Towards Projectional Editing for Model-Based SPLs

Dennis Reuling
dreuling@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Christopher Pietsch
cpietsch@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Udo Kelter
kelter@informatik.uni-siegen.de
Software Engineering Group, University of Siegen, Germany

Timo Kehrer
timo.kehrer@informatik.hu-berlin.de
Humboldt-University of Berlin, Germany

ABSTRACT

Model-based software product lines (MBSPLs) are implemented using various variability mechanisms. These are commonly categorized whether they separate features *virtually* (e.g., using annotations) or *physically* (e.g., using modules). Each of these mechanisms comprises advantages and disadvantages regarding MBSPL development, maintenance and analysis. To date, MBSPL developers have to choose *upfront* which variability mechanism to use, and the chosen mechanism including its drawbacks is bound to the MBSPL's entire lifecycle. In contrast, *projectional editing* has recently shown very promising potential of making the development of classical SPLs (e.g., implemented in C/C++) more flexible. *User-editable projections* allow developers to switch *fluidly* between different variability mechanisms based upon a common internal representation known as variational abstract syntax tree.

In this paper, we report on ongoing work on the projectional editing of MBSPLs, which is challenged by a set of additional requirements. We lay the foundation for different editable projections using a common *variational abstract syntax graph (vASG)* as internal representation. This vASG is used for a fine-grained variability representation of EMOF-based models. We demonstrate the feasibility of our approach by incorporating different variability mechanism projections (150% models and delta modules) and modeling languages (Ecore class diagrams and UML state machines) used in existing MBSPL case studies.

CCS CONCEPTS

• **Software and its engineering → Model-driven software engineering**; **Software product lines**; *Software evolution*;

KEYWORDS

Model-based Software Product Line Engineering, Projectional Editing, Virtual Separation, Physical Separation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VaMoS '20, February 5–7, 2020, Magdeburg, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377030

ACM Reference Format:
Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2020. Towards Projectional Editing for Model-Based SPLs. In *Proceedings of Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems, Magdeburg, Germany, February 5–7, 2020 (VaMoS '20),* 10 pages.
https://doi.org/10.1145/3377024.3377030

1 INTRODUCTION

Software product line engineering (SPLE) tackles the complexity of variant-rich software systems by handling common and variable parts in a family of variants using *features* [9, 11]. In various domains, traditional source code and programming languages (like C/C++) have been replaced by (executable) models and modeling languages (like UML or Simulink), leading to model-based software product lines (MBSPLs) [1]. Different variability mechanisms have been proposed [2, 10, 14, 29] for implementing MBSPLs.

Annotative approaches mark model elements and their properties with presence conditions under which they are valid. Hence, *all* configurations are superimposed into *one* so-called 150% model [10], similar to preprocessor macros (e.g., *ifdef* directives in C/C++) known from code-based SPLs [1]. Thus a specific configuration can be derived by removing all model parts whose presence conditions are not fulfilled for this configuration. With such a *virtual* separation of features, developers can use existing tools and processes for implementing variability [16, 23], and a superimposed representation enables the efficient application of various family-based quality assurance techniques [33, 42, 43]. Furthermore, annotative approaches offer a straightforward SPL adoption path by *adding* variability to legacy systems without restructuring the core functionality [37].

In contrast, modular variability approaches are based upon *structuring* the product line implementation along features [1]. Each feature (combination) is implemented in a separate module, a specific configuration is *composed* by integrating [3] or applying [30, 39] its modules. Such a *physical* separation offers potential reuse among modules and fosters feature-oriented development. Furthermore, due to the tight integration of features as first-class citizens, a modular representation enables feature-oriented analyses [12] (e.g., regarding feature interactions [1] and redundancy among modules [31]).

To summarize, each variability mechanism comprises advantages and disadvantages regarding MBSPL development, maintenance and analysis. Thus, developers may want to switch between these mechanisms, depending on the (changing) development context during the lifecycle of an MBSPL. Although several approaches combine different variability mechanisms [15, 17, 22], this increases the complexity and developers who are not aware of all the variability mechanisms do not perceive as many benefits as expected [16, 20, 21]. Instead, *projectional editing* integrates different mechanisms independently and has recently shown promising results for code-oriented SPLs [5, 28]. The key concept is to use a *common* variational structure as *internal* representation [6] that is edited using different projections as *external* representations. Developers may switch between different projections at any time, and all changes are propagated directly to the internal representation and thus synchronized among all other projections. In this paper, we leverage these concepts from code-oriented SPLs and lift them to model-based SPLs:

- We analyze the additional challenges and requirements for projectional editing of MBSPLs regarding a) syntactical well-formedness and b) available variability mechanisms, as compared to state-of-the-art approaches known from code.
- We propose a new internal variability representation (vASG) which adheres to these requirements and which is expressive enough for supporting a) different variability mechanisms and granularity levels and b) any EMOF-based modeling language.
- We demonstrate the feasibility of our approach by implementing different user-editable projections and apply them to realistic MBSPL case studies.

2 BACKGROUND AND MOTIVATION

We first provide the necessary background and discuss requirements for projectional editing for MBSPLs. Therefore, we introduce an MBSPL, called *Expression Product Line (EPL)*, which has been adapted from [4] and serves as example throughout the paper.

2.1 Running Example

Fig. 1 shows the *problem space* of the EPL in terms of a *feature diagram*. It consists of the two and-groups *Data* and *Operations* and the alternative-group *Type*. While the first one declares three kinds of expressions, namely literals (*Lit*), addition (*Add*) and negation (*Neg*), the second group defines the operations *Print* and *Eval*, which can be performed on an expression. A literal and the return type of the operation eval can be either an *Integer* or a *Float*, as declared by the alternative-group. Mandatory features represent the *commonalities* of the MBSPL, while optional and alternative features represent the *variability* of the MBSPL.

In the remainder, we illustrate the realization of the *solution space* following an annotative and a modular variability approach.

Annotative Variability Mechanism. Fig. 2 shows the 150% model of the EPL as an annotated Ecore class diagram [1], the Ecore annotations attached to classes, their properties and references are used to specify commonalities and variabilities. Model elements without annotations represent the commonalities consisting of the

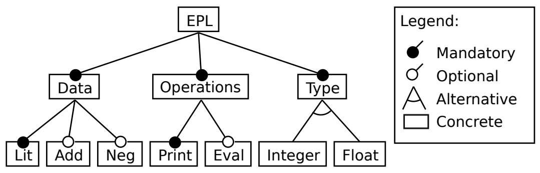

Figure 1: Feature model of the problem space of the Expression Product Line (EPL), slightly adapted from [4].

interface Exp that declares a method print() and the class Lit implementing this method. The optional and alternative features, i.e., the variabilities, are realized by model elements annotated with a *presence condition*. For instance, the class Lit declares the property value two times but with different types, namely *Integer* and *Float*. Given a valid configuration comprising the feature *Integer*, the property annotated with *Float* is not included in the final model. As another example, the feature *Eval* is realized by two variants of the method eval that differ in their return types (both in the interface Exp and the implementing class Lit). The respective elements are annotated with the presence conditions *Eval and Integer* as well as *Eval and Float*.

Note that, in general, the granularity of variability for annotative approaches depends on syntactical constraints of the domain modeling language. Regarding our example, multiplicity constraints of the Ecore metamodel (e.g., each method may have at most one return type) results in duplicated elements for the method eval and property value, thus limiting re-use among these elements.

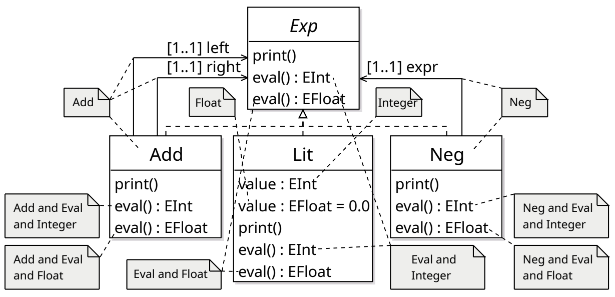

Figure 2: Implementation of the EPL as an annotated Ecore class diagram serving as 150% model.

Modular Variability Mechanism. As a modular variability mechanism, we use *delta modeling*, a language-independent transformational approach in which the commonalities and variabilities of an MBSPL are realized by specifying a *core model* and a set of *delta modules* [38]. The core model typically represents a valid configuration of the MBSPL, while delta modules specify model transformations using predefined language-specific *delta operations* that may add, modify ore remove model elements. Therefore, an operation supplies several parameters for passing arguments like context elements and attribute values. For the sake of clarity, we refer to delta operations with passed arguments as *delta actions*. To relate a delta module to one or several features, it is equipped with an *application condition*, i.e., a propositional formula over a subset of all available features.

[1]https://www.eclipse.org/modeling/emf/

Fig. 3 shows the core model of the EPL realizing all mandatory features (*Lit* and *Print*) as well as the alternative feature *Integer*. Listing 1 shows the delta module *DEval* following the textual syntax of the SiPL-framework [29, 31, 32]. Its application condition (*when-clause*) consists of the feature *Eval* (line 1), which means that the delta module is applied when the feature *Eval* is selected in a valid configuration. The set of available delta operations is determined by the document type (line 2), i.e., the namespace uri of the modeling language's metamodel. The delta module adds the operation `eval` to the interface `Exp` (lines 4-8) and class `Lit` (lines 13-17) with `EInt` as return type (lines 9-12, 18-21). Note that in Ecore class diagrams, operations are referred to as `EOperations`, and interfaces are actually classes (referred to as `EClasses`) which are declared to be an interface.

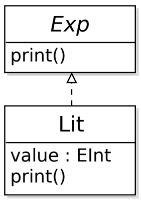

Figure 3: Core model of a delta-based EPL implementation.

The delta module *DFloat*, which is shown in Listing 2, realizes the alternative feature *Float* by modifying the type of the property `value` in the class `Lit` (lines 4-7).

Listing 3 shows the delta module *DEvalFloat* that realizes the feature interaction of the feature *Eval* with *Float*. Its application condition is a conjunction of both features (line 1). It changes the return type of the operation `eval` in the interface `Exp` (lines 4-7) and the class `Lit` (lines 8-11) from `EInt` to `EFloat`.

Listing 1: DEval

```
1  delta DEval when "Eval" {
2    docType : "http://www.eclipse.org/emf/2002/Ecore";
3
4    new_EOperation as expEval =
         addEOperationToEOperationsOfEClass(
5      object eClass : epl.Exp,
6      value name : "eval",
7      ...
8    );
9    setETypeOfEOperation(
10     object srcEOperation : expEval,
11     object tgtEClassifier : ecore.EInt
12   );
13   new_EOperation as litEval =
         addEOperationToEOperationsOfEClass(
14     object eClass : epl.Lit,
15     value name : "eval",
16     ...
17   );
18   setETypeOfEOperation(
19     object srcEOperation : litEval,
20     object tgtEClassifier : ecore.EInt
21   );
22 }
```

Listing 2: DFloat

```
1  delta DFloat when "Float" {
2    docType : "http://www.eclipse.org/emf/2002/Ecore";
3
4    changeETypeOfEAttribute(
5      object srcEAttribute : epl.Lit.^value,
6      object tgtEClassifier : ecore.EFloat
7    );
8  }
```

Listing 3: DEvalFloat

```
1  delta DEvalFloat when "Eval and Float" {
2    docType : "http://www.eclipse.org/emf/2002/Ecore";
3
4    changeETypeOfEOperation(
5      object srcEOperation : epl.Exp.eval,
6      object tgtEClassifier : ecore.EFloat
7    );
8    changeETypeOfEOperation(
9      object srcEOperation : epl.Lit.eval,
10     object tgtEClassifier : ecore.EFloat
11   );
12 }
```

Note that the delta module *DEvalFloat* depends on the delta module *DEval*, the operation `eval` must be added before its type can be set or altered. The SiPL-framework provides several analysis functions to automatically detect such interrelations between delta modules, they do not need to be managed manually. Moreover, note that the level of granularity for specifying variable parts depends on the available delta operations. In our implementation of the EPL, we use the approach and supporting tool presented by Kehrer et al. [19, 36] to derive a basic set of delta operations from the Ecore metamodel. Since this set includes a delta operation for changing the return type of the operation of a class in an Ecore class diagram, no redundant definition of the operation `eval` is needed in a delta-oriented implementation of the EPL, as opposed to the annotative implementation using a 150% model.

3 BASIC REQUIREMENTS

Following the ideas presented in [6], projectional editing relies on a common variational structure over a set of projections. In order to support projectional editing for MBSPLs, such a variational structure needs to fulfill the following criteria:

C1 - Representation. The variational structure must support any kind of variability representation used for MBSPLs. This includes positive and negative variability in the sense that any model element can be explicitly declared to be *absent* or *present* in the final model, according to its condition. Furthermore, combinatorial variability (e.g, feature interactions or derivatives [1]) must be supported.

C2 - Consistency. In contrast to [5], syntactical constraints defined by the metamodel of the underlying domain modeling language must be considered in the variational structure, especially in the context of visual models [18]. This ensures that all valid configurations (according to the variability model) are either syntactically well-formed or that invalid model fragments violating metamodel constraints are marked accordingly.

C3 - Granularity. The granularity of variability must be independent of the modeling language and its syntactical constraints. This is crucial for supporting fine-granular reuse and/or variability among features. In fact, this may not be possible without relaxation of some syntactical constraints of the modeling language (e.g., upper bounds of multiplicity constraints).

4 APPROACH

We now describe our approach for projectional editing of MBSPLs, which addresses the requirements presented in the previous section. Following the notions introduced in [5], we differentiate between

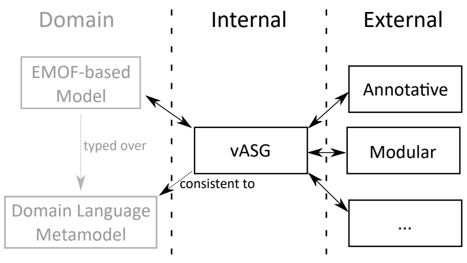

Figure 4: Overview of the model representations used by our approach.

internal and *external* representations of variability. A conceptual overview of our approach is shown in Fig. 4, which we will describe in more detail in the remainder of this section.

4.1 Internal Representation

Abstract Syntax Graph. We assume EMOF-based representations [2] of models, using the Eclipse Modeling Framework (EMF) as an implementation technology that serves as the de-facto standard implementation of EMOF. EMF models (referred to as EMOF-based model in Fig. 4) are basically object graphs whose objects and references are typed over a metamodel (referred to as domain language metamodel in Fig. 4).

Since references in EMF object graphs are only implicitly represented by their source and target objects (and thus can neither be identified nor annotated), our notion of an Abstract Syntax Graph (ASG) used as basis for our internal representation differs from the usual EMF representation by treating references as first class citizens. Such an ASG can be decomposed into *atomic model entities* (*AME*) [27], namely *Elements*, *Attributes* and *References*:

- *Elements* represent the objects of an EMF object graph. Their type is drawn from the language metamodel (i.e., an instance of `EClass`), and they may contain other elements.
- *Attributes* of model elements represent the properties (referred to as structural features in EMF) of an element. Their type is declared by the language metamodel (i.e., by an instance of `EAttribute`), and they have a value drawn from the domain of the corresponding type declaration.
- *References* between elements represent the references of an EMF object graph as a first class citizen. Their type is drawn from the language metamodel (i.e., an instance of `EReference`), and they have a source and a target *Element*. A *Reference* may have an opposite *Reference* in order to represent bidirectional references.

Fig. 5 shows an excerpt of the ASG of our example model introduced in Fig. 3. The class `Exp` is decomposed into 7 atomic model entities. It contains three *Attribute* entities `name`, `abstract` and `interface` as well as one *Reference* `eOperations` that references another *Element* `print`. In turn, the element `print` has one *Attribute* specifying its `name`. Since we work with Ecore class diagrams in this example, types are drawn from the Ecore metamodel, e.g., element `print` if of type *EOperation* (not shown in Fig. 5).

[2]https://www.omg.org/spec/MOF/2.5.1

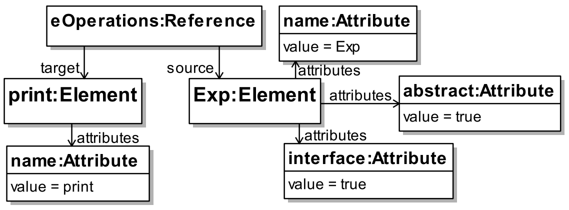

Figure 5: AMEs of the element Exp.

Variational Abstract Syntax Graph (vASG). We now extend our notion of an ASG to support variability. The conceptual structure of our *Variational Abstract Syntax Graph* (vASG) is shown in Fig. 6. For a given MBSPL, the `SuperimposedModel` contains all AMEs (see abstract `Entity` in the center), each being one of the three kinds of entities described earlier in this section.

According to criterion **C1** (see Section 3), our vASG additionally introduces an `Annotation` object that may be attached to any AME. In case of simple boolean conditions, one may use the *body* to denote the *presence* of an AME in case of a selected feature (e.g., the AME representing class `Neg` in Fig. 2 for feature *Neg*). Additionally, arbitrary formulas over features may be expressed as well, and we distinguish among presence and absence conditions, the latter are needed for explicitly excluding certain entities from a projection (e.g., deletions in delta modules) as we will later discuss in more detail in Sec. 4.2.

Regarding criterion **C2**, we enforce certain syntactic consistency constraints to be fulfilled by a vASG. The minimum level of consistency required by projections is that models are at least editable in their external representation [18]. Thus, mandatory properties of model elements (e.g., multiplicities with a lower bound greater than 0) as well as elementary EMF constraints (e.g., each element except the root element has exactly one container) are enforced by our vASG representation.

To accommodate for criterion **C3**, in contrast to lower bounds of multiplicity constraints, we do not enforce the vASG to comply to *upper* bounds defined by a metamodel's reference types and follow the idea of *superimposing* [3] similar elements to a single unified one. To achieve this, our vASG incorporates a local signature for uniquely identifying equal and similar entities using (exchangeable) `SignatureCalculators`. This way, a `SuperimposedElement` fosters reuse by subsuming redundant entities, i.e., their properties and annotations are unified. In contrast, each `SuperimposedElement` may contain arbitrary many entities with different properties, e.g., different values for a "name" attribute. This allows for fine-granular variability even for single-valued properties, independently of the underlying domain modeling language. For example, one may define the signature of an `Attribute` as the signature of its container `SuperimposedELement`, its type and its value. The computation of a *meaningful* signature is a complex problem itself, similar to matching problems known from model version and variant management (see [41] for a survey regarding model comparison). In our approach, the computation algorithm is an exchangeable component that can be used to adapt the vASG representation to a given modeling language.

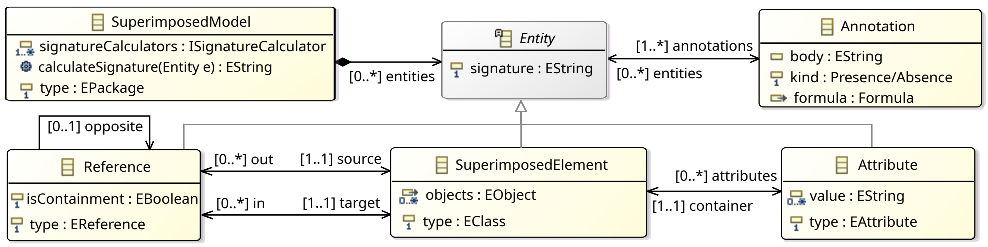

Figure 6: Conceptual structure of the vASG.

Figure 7 shows an excerpt of the vASG for the EPL introduced in Sec. 2. The *SuperimposedElement* `Lit` represents the respective class along with its contained elements, namely the operations `print` and `eval` that are connected via the *Reference* of type `eOperations`, and the *Element* called `value` that is connected via the *Reference* of type `eStructuralFeature`. The annotations consist of a set of *presence conditions*, that is a disjunction of propositional formulas that specify to which features or combinations thereof an element is connected. For instance, the reference of type `eType` from the *SuperimposedElement* `eval` to the element `EInt` is annotated with *Eval and Integer*, while the element `eval` is annotated with *Eval* as well as *Eval and Integer*, which can be read as *(Eval) or (Eval and Integer)*. Although operations may have only one return type according to the Ecore metamodel, our variational structure integrates two dedicated `eType` references for element `eval`, thus ensuring both re-use and consistency.

4.2 External Representations

We now demonstrate the feasibility of our approach by implementing two different user-editable projections, either virtually or physically separating features. To this end, we describe how to project variability from the vASG to the respective external representations and vice versa.

Annotative Projection. Basically, transforming the internal representation to an external 150% model proceeds as follows: Each superimposed element yields an element in the the domain model,

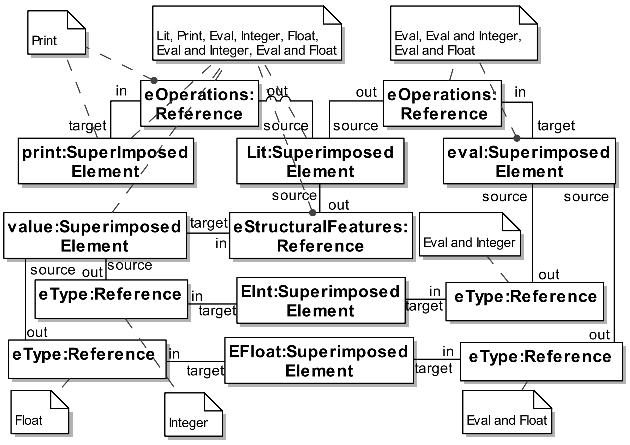

Figure 7: Excerpt of the vASG for the EPL.

i.e., an object in the EMF object graph of the 150% model. `Reference` and `Attribute` entities result in references and object attributes of the EMF object graph of the 150% model. For attaching variability information in the 150% model, we use an exchangeable annotation mechanism. In our example, we use `EAnnotations` as described in Sect. 2. In case of UML models, annotations may be implemented based on `Comment` objects (see Sect. 5).

However, in contrast to source code-based SPLs [5], we may not be able to apply a one-to-one transformation from the vASG to the 150% model. Due to criterion **C3**, our internal representation allows for violating metamodel constraints regarding upper bounds of multiplicity constraints and thus cannot be projected without preprocessing.. First, we analyze if any entitiy violates such a constraint. If so, we *duplicate* the "conflicting" entities and all their properties as well as incoming references, similar to the variability normalization concept presented in [34]. As duplicating elements may introduce further constraint violations (e.g., multiplicities are violated for the container), this process is repeated until no constraint violations are found. Regarding our example vASG from Fig. 7, element `eval` needs to be duplicated due to the two `eType` references. This yields two superimposed elements for `eval` having an `eType` reference to `EInt` and `EFloat`, respectively. Furthermore, the incoming `eOperations` Reference is duplicated as well, thus element `Lit` owns two operations named `eval`.

Importing an 150% model into our vASG essentially creates one superimposed element for each object in the EMF object graph of the 150% model, each of which is annotated according to the external annotations. Usually, not all external elements are annotated, either by developers' intention or due to an implementation bug [1, 16]. As our approach and its analyses require all entities to be annotated in the vASG (e.g, for ensuring criteria **C2**), we additionally propagate annotations along AMEs based upon syntactical constraints defined by the metamodel (similar to [13]). For example, the vASG element representing class `Add` in Fig. 2 is enriched with all annotations of contained elements (both `eval` operations), i.e., *Add and Eval and Integer* as well as *Add and Eval and Float*. Such propagation is also done for all contained entities, including references (based upon their source and target) as well as attributes (based upon their container elements). The final vASG contains all elements from the original 150% model, enriched with variability information for all AMEs. Finally, the vASG is *postprocessed* automatically for ensuring criterion **C3**, using a given signature calculator as described in Sect. 4.1. This is necessary as the original 150% model may contain (needed) duplicates, which can be eliminated in our vASG representation for more fine-granular re-use. For

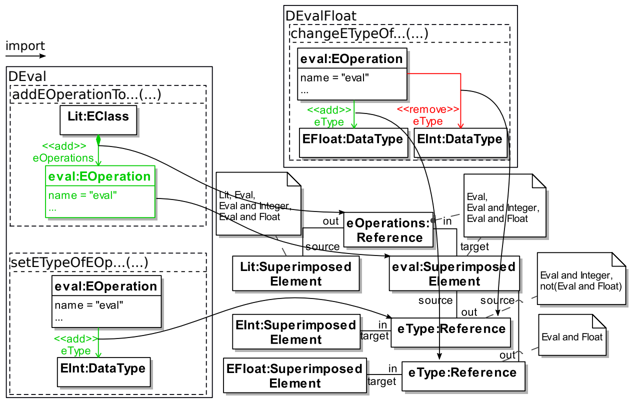

Figure 8: Illustration of importing the delta modules *DEval* and *DEvalFloat* into the vASG of the EPL.

example, both attributes `value` in class `Lit` in Fig. 2 can be superimposed due to their similarity, thus resulting in one superimposed element in the vASG (see Fig. 7, left).

Modular Projection. For importing a delta module into the vASG, each delta action is analyzed regarding its effect onto the ASG. Here, we make use of the graph-based specification of delta operations provided by SiPL [30]. A delta operation is specified declaratively by a parameterized graph transformation rule. Rule graph elements may be attached with a *change action* (`add` or `remove`), elements without change action are considered to be preserved. Context elements and attribute values may be passed as parameters. Figure 8 illustrates the import of the delta modules *DEval* and *DEvalFloat* into the vASG of the EPL (attributes are omitted for the sake of readability). We use colors to highlight change actions of delta operations, green marked rule elements represent additions, while red marked elements represent removals. The other elements represent context elements. The delta module *DEval* adds the operation `eval` to the class `Lit` and sets its type to `EInt`, while the delta module *DEvalFloat* changes the type of the added operation from `EInt` to `EFloat`.

In a first step, we import the core model and annotate each entity with a conjunction of all features realized by the core model. After that, we check if the delta module to be imported depends on other delta modules. The depending delta module may require the presence or absence of entities that are added or removed by other delta modules. In this case, the required delta modules must be imported first in order to resolve all context elements for depending delta actions. For instance, the delta module *DEvalFloat* requires that the element `eval` exists and that its type is set to `EInt`. This is done by the delta module *DEval*, which in turn requires the element `Lit` being part of the core model. Thus, the delta module *DEval* must be imported before *DEvalFloat*. To import a delta module, we iterate over all its delta operations' change actions, a suitable order in which these change actions are applicable can be inferred from the delta operations' declarative specification [18]. Graph elements that are to be created by a delta action are annotated with the delta module's application condition serving as presence condition. Graph elements that are to be removed by a delta action are annotated by the delta module's negated application condition serving as absence condition. Imported graph elements are illustrated by unidirectional arrows in Figure 8. For ensuring criterion **C2**, analogously to the import of a 150% model, the new annotations are propagated to all contained elements along all attributes and outgoing references having the same annotation as the containing element .

To export a delta module from the vASG, we exploit SiPL's facility to derive a delta module from a model difference [29, 32]. Given the application condition of the delta module to be derived, we export an original and a modified model from the vASG, serving as input for the difference calculation that yields the delta module. The modified model comprises all entities annotated with the application condition. For instance, given the application condition *Eval and Float*, the modified model contains the elements `Lit`, `eval` and the references `eOperations` from `Lit` to `eval` as well as `eType` from `eval` to `EFloat` [3] (see Figure 8). To obtain the original model, we start from the modified one and (i) drop those elements that are exclusively annotated by the application condition and (ii) include those elements that are annotated by the negated application condition. When deriving the difference from the original to the changed model, elements dropped in step (i) result in creations while elements included in step (ii) result in deletions, all retained elements serve as context for the respective delta actions. For instance, for exporting the original model for *DEvalFloat*, we start with the modified model as described above, discard the reference `eType` from `eval` to `EFloat` (exclusively annotated with *Eval and Float*), and include the reference `eType` from `eval` to `EInt` (annotated with *not(Eval and Float)*). The resulting difference between the original and modified model for *DEvalFloat* consists of an `eType` reference from `eval` to `EInt` that is to be deleted and an `eType` reference from `eval` to `EFloat` that is to be created. These changes are specified by the delta actions of *DEvalFloat* shown in Figure 8.

5 EXPERIMENTAL EVALUATION

We evaluate our approach w.r.t. the following research questions:

(RQ1) Feasibility. Is the internal representation suitable to fulfill all criteria for projectional editing of MBSPLs?

(RQ2) Implications. How does switching between variability mechanisms impact the implementation characteristics of the MBSPL?

(RQ3) Fluidity. Can variability mechanisms be changed fluidly by a developer in the case of larger MBSPLs?

5.1 Study Subjects

In our experiments, we work with two MBSPLs that are implemented using different modeling languages and variability mechanisms. We selected these subjects as they are publicly available and have been originally developed significantly differently by either separating features *physically* or *virtually*.

EPL. The EPL, introduced in Sect. 2.1, is based upon Ecore class diagrams and has been originally developed using a modular approach [4]. Additionally, we have re-implemented the EPL using an annotative approach (see Fig. 2) for comparison purposes. The problem space specifies *16* valid configurations. The solution space

[3] Note that the elements `EInt` and `EFloat` represent "remote" elements from another model (in this case, the Ecore metamodel). Elements from another model are not annotated, they are always present as long as they are referenced by a non-remote entity.

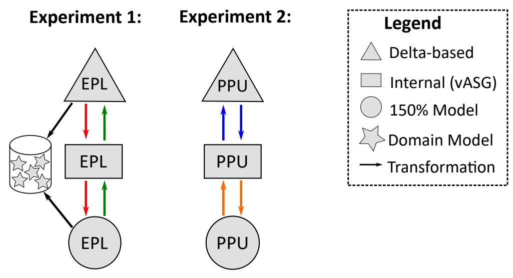

Figure 9: Study design of our experiments.

of the annotative implementation comprises a 150% model consisting of *55* elements. The modular realization includes the core model and *9* delta modules. The core model comprises *5* elements, while the delta modules consist of *2.4* delta actions on average.

PPU. The *Pick and Place Unit (PPU)* MBSPL [25, 44] is extracted from a case study for embedded software in industrial automation. The PPU manipulates work pieces of different materials by either transporting them to their destination or additionally handling them in some ways (e.g., sorting or stamping). The problem space is defined by a feature model specifying *69* valid configurations. The solution space consists of a 150% UML state machine model comprising 1178 elements.

5.2 Study Design and Methodology

We conducted two different experiments depicted in Fig. 9. Experiment 1 addresses research questions **RQ1** and **RQ2** using the EPL. We transformed the EPL from both original implementations (top and bottom) into our internal vASG representation (center) and projected them into the other variability mechanism (red and green arrows). To ensure correctness, we generated all valid configurations (left) using both representations. For reasoning about implementation characteristics, we compared the different internal and external representations with each other.

Regarding **RQ3**, we performed experiment 2 that measures the time consumption needed to switch between different projections of the PPU (which is substantially larger than the EPL). This includes the time needed for *unifying* equal elements in the vASG during import from an 150% model as well as the duplication of elements needed for exporting the vASG (see orange arrows). In case of our delta-based projection, we measure the time needed for generating/deriving delta-modules as well the transformation to the vASG (see blue arrows). The overall runtimes are split into loading and serializing EMF models as well as the actual transformation.

All subject systems and experimental data are available for reproducibility at the accompanying webpage [35].

5.3 Results & Discussion

RQ1: Feasibility. In our experiments, we have worked with different modeling languages (Ecore class diagrams and UML statemachines) and variability mechanisms (150% models and delta modules). Furthermore, both subject systems are using different kinds of variabilities (e.g., presence and/or absence conditions). Additionally, our subject systems contain feature interactions, which are realized by elements (resp. delta modules) annotated with complex formulas (resp. application conditions), e.g., a conjunction of several features. This provides confidence that our approach can be applied to further EMOF-based modeling languages and variability mechanisms as well (as required by criterion **C1**).

The same set of valid configurations can be generated from all the four external representations of the EPL, and each of these configurations yields a valid model. In other words, each of the generated models is consistent w.r.t. to the consistency constraints defined by the Ecore metamodel. This provides confidence that our approach respects consistency constraints that must be fulfilled in our context of projectional editing (as required by criterion **C2**).

Finally, the two imports conducted in terms of our first experiment yield the same vASG (cf. Table 1 columns 3 and 5), although both external representations used in the EPL case study support re-use at different levels of granularity. This provides confidence that our internal representation supports fine-granular re-use, even if the modeling language's consistency constraints would prevent the re-use of certain elements in external representations (as required by criterion **C3**).

RQ2: Implications. The most interesting implementation characteristics of the EPL in its different external representations are summarized in Tables 1, 2 and 3.

As expected, the internal representation allows for more re-use than the external representation of a 150% model (cf. Table 1 columns 2 and 3). This is due to the fact that certain consistency constraints of the language's metamodel are not enforced by our vASG. Furthermore, we see that the exported 150% model may contain (substantially) more annotations as compared to the original 150% model (cf. Table 1 columns 2 and 4). This is due to the fact that we *propagate* "missing" annotations during import into the vASG, thus they are included in the resulting external representation after exporting. To avoid this effect, however, core feature annotations may be suppressed during export if desired by the developer.

As for the delta-based representation, the core model as well as delta modules may differ depending on the transformation direction (see Fig. 9, red vs. green arrow). When creating an MBSPL from scratch, the core model may be chosen freely, which then impacts all delta modules as well. Table 2 gives an overview of the solution space of the original delta-oriented EPL implementation. We chose a core model comprising all mandatory features, the alternative feature *Integer* and the optional feature *Neg*. For realizing the remaining features, we implemented 9 delta modules, most of them only add or modify elements. Only the delta module *DNotNeg* removes all elements related to the feature *Neg*. Removals are one of the unique characteristics of delta modeling [38, 39]. When importing this implementation into the vASG (s. Fig. 9, red arrow), we get the same vASG as when importing the 150% model (cf. Table 1 columns 3 and 5). The only difference is the amount of annotations, as some elements are annotated with absence conditions due to the removals. Although our approach also supports absence conditions as annotations (see Sect. 4.2), these kinds of annotations were not used in the original 150% model of the EPL. Hence, when exporting the vASG obtained from the 150% model to delta modules (s. Fig. 9, green arrow), the core model consists of all core elements of the 150% model (all elements that are part of all configurations) and 10 delta modules without removals (see Table 3).

Table 1: Original 150% model of the EPL and vASG representations obtained from different external representations.

	Representation			
Type	150%	150% $\rightarrow$ vASG	vASG $\rightarrow$ 150%	Δ $\rightarrow$ vASG
Annotation	17	10	71	14
EAttribute	2	1	2	1
EClass	4	4	4	4
EDatatype	2	2	2	2
EOperation	12	8	12	8
EPackage	1	1	1	1
EReference	3	3	3	3
Total	**24**	**19**	**24**	**19**

Table 2: Original delta-oriented EPL.

Core Model	
Type	Elements
EAttribute	1
EClass	3
EDataType	1
EOperation	3
EReference	1
EPackage	1
Total	**10**

Delta-Module	Add	Rem	Mod
DAdd	4	0	1
DNotNeg	0	3	1
DEval	2	0	2
DNegEval	1	0	1
DAddEval	1	0	1
DFloat	0	0	1
DEvalFloat	0	0	2
DNegEvalFloat	0	0	1
DAddEvalFloat	0	0	1
Total	**8**	**3**	**11**

Table 3: Exported delta-oriented EPL (vASG $\rightarrow$ Δ).

Core Model	
Type	Elements
EClass	2
EOperation	2
EPackage	1
Total	**5**

Delta Module	Add	Rem	Mod
DAdd	4	0	1
DAddEvalFloat	1	0	1
DAddEvalInteger	1	0	1
DEvalFloat	2	0	2
DEvalInteger	2	0	2
DFloat	1	0	1
DInteger	1	0	1
DNeg	3	0	1
DNegEvalFloat	1	0	1
DNegEvalInteger	1	0	1
Total	**17**	**0**	**12**

RQ3: Fluidity. The runtimes of all the four projectional transformations conducted in terms of our second experiment using the PPU case study are shown in Table 4, averaged over 5 runs. The transformation from the vASG to the 150% model takes about 2 seconds, while the inverse direction takes about 10 seconds. This is due to the additional overhead of *unifying* equal elements in the vASG during import. The transformation of delta modules to the vASG takes *11* seconds, while the other direction takes *107* seconds. In case of longer runtimes (e.g., exporting the vASG to delta modules) most of the consumed time (64%) is due to EMF overheads for (de-)serialization of the underlying models. In particular, the difference between delta modeling and 150% models is caused by the number of saved EMF models, as the delta modules themselves are saved as separate models [29] compared to saving only one 150% model. Given the expected frequency of switching between variability mechanisms in realistic MBSPL development scenarios, we argue that fluidly switching between different external representations may be feasible in the case of larger MBSPLs, although this needs more in-depth investigation in future work.

5.4 Threats to Validity

Our selection of two MBSPLs may threaten the validity because the results might not be representative of other modeling languages and

Table 4: Results of Experiment 2.

	Consumed Time (s)		
Direction	Transformation	De/Serialization	**Total**
vASG $\rightarrow$ 150%	1.5	0.6	**2.1**
150% $\rightarrow$ vASG	9.1	0.8	**9.9**
vASG $\rightarrow$ Δ	38.1	68.8	**106,9**
Δ $\rightarrow$ vASG	10.7	0.7	**11.4**

variability mechanisms. Furthermore, we assume the implemented variability to be correct (e.g., no invalid annotations) and consistent to the feature model, which may not be realistic in other cases. Finally, the consistency level assumed is based upon our earlier work [18] as well as alignment with the delta module projection [30] and may need to be adapted for other projections and consistency levels.

6 RELATED WORK

Although several variability mechanisms have been proposed in the context of model-based SPLs [2, 10, 14, 29], there is no work on projectional editing for MBSPLs. Thus, most closely related to our work are code-oriented approaches [5–7, 28], which we extend regarding specific requirements of MBSPLs. In contrast, (different) variability mechanisms are integrated in [15, 22], yet without allowing to *switch* between different mechanisms but by using them in parallel. Kästner et al. [17] try to mitigate this by *refactoring* between physical and virtual separation, although no common variation structure is used as in our approach. Projectional editing is remotely related to *multi-view* editing, but variability is either out of scope [8] or used for consistency-checking [26]. So-called variation control systems [24, 40] also differentiate between internal and external representations, which is similar to our ongoing work regarding *variant projection* (see next section).

7 CONCLUSION AND ONGOING WORK

In this paper, we leveraged concepts of projectional editing of code-centric SPLs and lifted them to model-based SPLs. In contrast to the code-centric scenario, projectional editing of model-based SPLs comes with a set of additional requirements which are addressed by our notion of a variational abstract syntax graph and according bi-directional transformations into external representations.

To reduce complexity, which is not the aim of switching between variability mechanisms, we plan to integrate the so-called *variant projection* (see [5, 28]) into our framework, which renders a single configuration of the MBSPL in its external representation. Moreover, our consistency analysis as well as annotation propagation is based upon annotation strings without inspecting the formula in detail. We want to take single terms and the feature model into consideration for reasoning about consistency and correctness. In earlier work we demonstrated the drawbacks of current delta-based quality analyses due to their pair-wise strategy [31]. We plan to use the succinct vASG representation for efficient variability-specific analyses [30] for higher-order conflict detection.

ACKNOWLEDGMENTS

This work was partially supported by the DFG (German Research Foundation) within the projects CoMoVa (grant nr. 330452222) and VariantSync (grant nr. KE 2267/1-1).

REFERENCES

[1] S. Apel, D. Batory, C. Kästner, and G. Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation.* Springer Science & Business Media, Berlin Heidelberg.

[2] Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. 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.* 4–19. https://doi.org/10.1007/978-3-642-02408-5_2

[3] S. Apel, C. Kästner, and C. Lengauer. 2013. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Transactions on Software Engineering* 39, 1 (Jan 2013), 63–79. https://doi.org/10.1109/TSE.2011.120

[4] Benjamin Behringer and Moritz Fey. 2016. Implementing Delta-oriented SPLs Using PEoPL: An Example Scenario and Case Study. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016).* ACM, New York, NY, USA, 28–38. https://doi.org/10.1145/3001867.3001871

[5] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17).* IEEE Press, Piscataway, NJ, USA, 563–574. https://doi.org/10.1109/ICSE.2017.58

[6] Benjamin Behringer and Steffen Rothkugel. 2016. Integrating Feature-based Implementation Approaches Using a Common Graph-based Representation. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16).* ACM, New York, NY, USA, 1504–1511. https://doi.org/10.1145/2851613.2851791

[7] Thorsten Berger, Markus Völter, Hans Peter Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of Projectional Editing: A Controlled Experiment. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, New York, NY, USA, 763–774. https://doi.org/10.1145/2950290.2950315

[8] Antonio Cicchetti, Federico Ciccozzi, and Thomas Leveque. 2012. A hybrid approach for multi-view modeling. *Electronic Communications of the EASST* 50, 0 (July 2012). http://journal.ub.tu-berlin.de/eceasst/article/view/738

[9] P. Clements and L. Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc.

[10] K. Czarnecki and M. Antkiewicz. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *ACM International Conference on Generative Programming and Component Engineering (Lecture Notes in Computer Science)*, Vol. 3676. Springer-Verlag, 422 – 437.

[11] K. Czarnecki and U. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley.

[12] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information & Software Technology* 106 (2019), 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

[13] Sandra Greiner and Bernhard Westfechtel. 2019. On Determining Variability Annotations In Partially Annotated Models. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2019, Leuven, Belgium, February 06-08, 2019.* 17:1–17:10. https://doi.org/10.1145/3302333.3302341

[14] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K Olsen, and Andreas Svendsen. 2008. Adding standardized variability to domain specific languages. In *2008 12th International Software Product Line Conference.* IEEE, 139–148.

[15] Jose-Miguel Horcas, Alejandro Cortiñas, Lidia Fuentes, and Miguel R. Luaces. 2018. Integrating the Common Variability Language with Multilanguage Annotations for Web Engineering. In *Proceedings of the 22Nd International Systems and Software Product Line Conference - Volume 1 (SPLC '18).* ACM, New York, NY, USA, 196–207. https://doi.org/10.1145/3233027.3233049

[16] Jose-Miguel Horcas, Mónica Pinto, and Lidia Fuentes. 2019. Software Product Line Engineering: A Practical Experience. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A (SPLC '19).* ACM, New York, NY, USA, 164–176. https://doi.org/10.1145/3336294.3336304

[17] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. *SIGPLAN Not.* 45, 2 (Oct. 2009), 157–166. https://doi.org/10.1145/1837852.1621632

[18] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. 2013. Consistency-preserving edit scripts in model versioning. In *IEEE/ACM 28th International Conference on Automated Software Engineering (ASE).* 191–201.

[19] Timo Kehrer, Gabriele Taentzer, Michaela Rindt, and Udo Kelter. 2016. Automatically deriving the specification of model editing operations from meta-models. In *International Conference on Theory and Practice of Model Transformations.* Springer, 173–188.

[20] Jacob Krüger, Gül Calikli, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* ACM, New York, NY, USA, 338–349. https://doi.org/10.1145/3338906.3338968

[21] Jacob Krüger, Kai Ludwig, Bernhard Zimmermann, and Thomas Leich. 2018. Physical Separation of Features: A Survey with CPP Developers. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18).* ACM, New York, NY, USA, 2042–2049. https://doi.org/10.1145/3167132.3167351

[22] Jacob Krüger, Ivonne Schröter, Andy Kenner, Christopher Kruczek, and Thomas Leich. 2016. FeatureCoPP: Compositional Annotations. In *Proceedings of the 7th International Workshop on Feature-Oriented Software Development (FOSD 2016).* ACM, New York, NY, USA, 74–84. https://doi.org/10.1145/3001867.3001876

[23] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10).* ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819

[24] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017).* ACM, New York, NY, USA, 49–62. https://doi.org/10.1145/3136040.3136054

[25] Malte Lochau, Johannes Bürdek, Sascha Lity, Matthias Hagner, Christoph Legat, Ursula Goltz, and Andy Schürr. 2014. Applying model-based software product line testing approaches to the automation engineering domain. *at-Automatisierungstechnik* 62, 11 (2014), 771–780.

[26] Roberto Erick Lopez-Herrejon and Alexander Egyed. 2010. Detecting Inconsistencies in Multi-view Models with Variability *(ECMFA'10).* Springer-Verlag, Berlin, Heidelberg, 217–232. https://doi.org/10.1007/978-3-642-13595-8_18

[27] J. Martinez, T. Ziadi, T. F. Bissyande, J. Klein, and Y. L. Traon. 2015. Automating the Extraction of Model-Based Software Product Lines from Model Variants. In *International Conference on Automated Software Engineering (ASE 15).* 396–406.

[28] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krueger, and Thorsten Berger. 2018. Multi-view Editing of Software Product Lines with PEoPL. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18).* ACM, New York, NY, USA, 81–84. https://doi.org/10.1145/3183440.3183499

[29] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL - A Delta-Based Modeling Framework for Software Product Line Engineering. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* 852–857. https://doi.org/10.1109/ASE.2015.106

[30] Christopher Pietsch, Udo Kelter, Timo Kehrer, and Christoph Seidl. 2019. Formal foundations for analyzing and refactoring delta-oriented model-based software product lines. In *Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, Paris, France, September 9-13, 2019.* 30:1–30:11. https://doi.org/10.1145/3336294.3336299

[31] Christopher Pietsch, Dennis Reuling, Udo Kelter, and Timo Kehrer. 2017. A tool environment for quality assurance of delta-oriented model-based SPLs. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2017, Eindhoven, Netherlands, February 1-3, 2017.* 84–91. https://doi.org/10.1145/3023956.3023960

[32] Christopher Pietsch, Christoph Seidl, Michael Nieke, and Timo Kehrer. 2019. *Model Management and Analytics for Large Scale Systems.* Elsevier, Chapter Delta-Oriented Development of Model-Based Software Product Lines with DeltaEcore and SiPL: A Comparison. accepted.

[33] Dennis Reuling, Udo Kelter, Johannes Bürdek, and Malte Lochau. 2019. Automated N-way Program Merging for Facilitating Family-based Analyses of Variant-rich Software. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 13 (July 2019), 59 pages. https://doi.org/10.1145/3313789

[34] Dennis Reuling, Malte Lochau, and Udo Kelter. 2019. From Imprecise N-Way Model Matching to Precise N-Way Model Merging. *Journal of Object Technology* 18, 2 (2019), 8:1–20. https://doi.org/10.5381/jot.2019.18.2.a8

[35] Dennis Reuling, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2020. Accompanying materials for this paper. http://pi.informatik.uni-siegen.de/Projekte/sipl/doc/vamos2020/.

[36] Michaela Rindt, Timo Kehrer, and Udo Kelter. 2014. Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools. *Demos@MoDELS* 14 (2014).

[37] J. Rubin, K. Czarnecki, and M. Chechik. 2015. Cloned product variants: from ad-hoc to managed software product lines. *International Journal on Software Tools for Technology Transfer* 17, 5 (2015), 627–646.

[38] Ina Schaefer. 2010. Variability Modelling for Model-Driven Development of Software Product Lines. In *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings.* 85–92.

[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*, Jan Bosch and Jaejoon Lee (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 77–91.

[40] 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). ACM, New York, NY, USA, 822–827. https://doi.org/10.1145/2970276.2970288

[41] M. Stephan and J. R Cordy. 2013. A Survey of Model Comparison Approaches and Applications.. In *Modelsward*. 265–277.

[42] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (June 2014), 45 pages.

[43] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards efficient analysis of variation in time and space. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*. ACM.

[44] B. Vogel-Heuser, C. Legat, J. Folmer, and S. Feldmann. 2014. *Researching Evolution in Industrial Plant Automation: Scenarios and Documentation of the Pick and Place Unit*. Technical Report TUM-AIS-TR-01-14-02. TU München.