Software Reuse: From Cloned Variants to Managed Software Product Lines

Christoph Seidl, David Wille, and Ina Schaefer

Abstract Many software systems are available in similar, yet different variants to accommodate specific customer requirements. Even though sophisticated techniques exist to manage this variability, industrial practice mainly is to copy and modify existing products to create variants in an ad hoc manner. This clone-and-own practice loses variability information as no explicit connection between the variants is kept. This causes significant cost in the long term with a large set of variants as each software system has to be maintained individually. Software product line (SPL) engineering remedies this problem by allowing to develop and maintain large sets of software systems as a software family.

In this chapter, we give an overview of variability realization mechanisms in the state of practice in the industry and the state of the art in SPL engineering. Furthermore, we describe a procedure for variability mining to retrieve previously unavailable variability information from a set of cloned variants and to generate an SPL from cloned variants. Finally, we demonstrate our tool suite DeltaEcore to manage the resulting SPL and to extend it with new functionality or different realization artifacts. We illustrate the entire procedure and our tool suite with an example from the automotive industry.

1 Introduction

Modern software exists in many similar *variants* that realize slightly different functionality to accommodate specific customer requirements. For example, the automotive industry allows customers of their cars to freely decide whether an optional alarm system should be included or whether the power windows should be manual or automatic. The selected configuration impacts the software required

C. Seidl (✉) · D. Wille · I. Schaefer
Technische Universität Braunschweig, Institute of Software Engineering and Automotive Informatics, Braunschweig, Germany
e-mail: c.seidl@tu-braunschweig.de; d.wille@tu-braunschweig.de; i.schaefer@tu-braunschweig.de

Y. Dajsuren, M. van den Brand (eds.), *Automotive Systems and Software Engineering*, https://doi.org/10.1007/978-3-030-12157-0_5

to operate the car as significant parts of the program logic may differ. Program logic may be specified by source code or, on a more abstract level, by design models such as *function block diagrams (FBDs)* [19], *MATLAB/Simulink*[1] models, or *Rational Rhapsody*[2] statecharts. In consequence, *variability* stemming from different configuration options has to be manifested in various realization artifacts by what is called a *variability realization mechanism* to reflect the differences in program logic.

Variability realization mechanisms used in practice, such as copying and modifying specific software systems through *clone-and-own*, are readily available and do not require additional tools or changes in development process and/or management structure. However, in the long run and with a growing number of variants, these approaches do not support sustainable managed software reuse (see Sect. 3.1). Software product lines (SPLs) [29] offer concepts and facilities for managed software reuse by treating a set of closely related systems as software family. However, there is a gap between the state of practice used in industry and the state of the art for managed software reuse of SPLs in academia with regard to variability realization mechanisms. Furthermore, manually adopting an SPL approach for a grown set of cloned software variants entails significant effort and cost [24].

To remedy these problems, in this chapter, we first give an extensive overview of both the state of practice and the state of the art in variability realization mechanisms. We then describe a procedure we devised to transition from the grown structure of cloned variants to managed reuse of SPLs that analyzes the cloned variants and generates the artifacts for an SPL largely automatically. Finally, we demonstrate our tool suite DeltaEcore to realize the described transition procedure for practical application in order to significantly reduce effort and cost of adopting a managed reuse strategy for grown systems with many variants.

Due to their relevance for industrial practice, we specifically focus on variability realization mechanisms and an associated reverse engineering procedure. To retrieve high-level variability information, such as conceptual *features*, from related product variants, other approaches exist [30]: Feature location techniques analyze natural-language requirements documents [46], product maps describing the software's composition [32, 39], or existing model-based products [50, 51]. However, these approaches do not consider fine-grained variability within realization artifacts or do not generate an SPL as we do.

The structure of this chapter is as follows: Sect. 2 provides an overview of SPL terminology and introduces a running example from the automotive domain. Section 3 explains and contrasts the state of practice and the state of the art in variability realization mechanisms. Section 4 describes our procedure to transition from grown software systems with multiple variants to managed reuse by first analyzing the individual clones and then generating an SPL from the realization artifacts and the

[1] http://www.mathworks.com/products/simulink/.

[2] https://www.ibm.com/us-en/marketplace/rational-rhapsody/.

collected information. Section 5 introduces our tool suite DeltaEcore used to create and maintain an SPL. Finally, Sect. 6 closes with a conclusion.

2 Background

In the following section, we provide background on software product lines and provide a running example to illustrate concepts and techniques throughout the chapter.

2.1 *Software Product Lines*

A software product line (SPL) [8, 29, 43] is an approach for managed reuse where a set of closely related software systems is perceived as a software family consisting of *commonalities* and *variabilities*. Commonalities constitute the functionality contained in all systems of the software family, and variabilities constitute the functionality that sets apart the individual software systems. Individual software systems of the software family are created by combining the commonalities with a selection of the variabilities. However, not all combinations of variabilities are necessarily valid as technical or economical concerns may prohibit certain combinations.

In an SPL, commonalities and variabilities of artifacts as well as the rules governing potential combinations of variabilities are represented on different levels of abstraction. In the *problem space* [9], variabilities are represented on a mere conceptual level with no regard to their technical realization, for example, as names with a description of functionality that can be used to communicate with nontechnical stakeholders, such as customers or management. In the *solution space* [9], variabilities are represented with their effect on realization artifacts, for example, source code or models. Figure 1 depicts an overview of the essential constituents of an SPL and their relations.

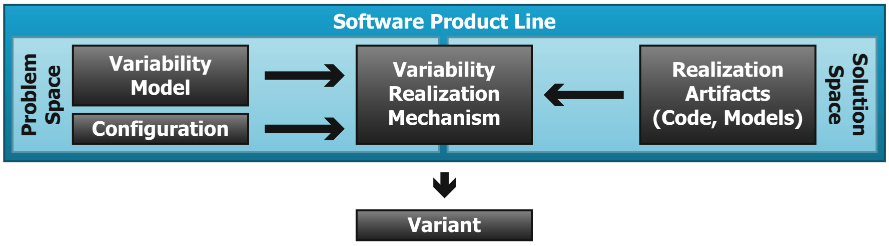

Fig. 1 Overview of the essential constituents of a software product line

The main constituent of the problem space is a *variability model*, which governs the valid combinations of variabilities by providing configuration rules. Various notations for variability models exist, such as feature models [9, 20], decision models [28], orthogonal variability models (OVMs) [29], and variability specifications (VSpecs) of the Common Variability Language (CVL) [16]. Despite the wide variety of available notations, feature models are by far the most commonly used in industrial practice [5] so that we elaborate on this notation.

A feature model is a hierarchical decomposition of a variable software system into *features*. A feature is a user visible functionality that, usually, is configurable, that is, may be selected or deselected [6]. Features may be *mandatory*, so that they have to be selected, or *optional*, so that they may be deselected. The root feature is implicitly considered mandatory. Furthermore, features may be grouped into *alternative groups*, so that exactly one of the contained features has to be selected, or *or groups*, so that at least one of the contained features has to be selected. Configuration constraints for a feature only apply if their respective parent feature is selected. For a concrete example of a feature model, see Fig. 2 in Sect. 2.2 for our running example. A configuration defined on the feature model is a selection of features that satisfies all configuration constraints.

The solution space constitutes the realization of all possible software systems of the software family. Realization artifacts may be of a wide variety of languages: Source code in different programming languages may specify program logic. Implementation models, such as statecharts, may define program flow on a higher level of abstraction. Design models, such as class diagrams or component diagrams, may represent parts of the architecture. Documentation material, such as user and developer manuals, may provide information for employing or extending software systems. The selection of variabilities as part of a configuration may have an effect on each of these artifacts as, depending on the choice of variabilities, certain functionality may be present in or missing from the software system. This yields the need to change the design, implementation, and documentation of the respective software system. To perform the respective changes, SPLs employ a *variability realization mechanism*, which takes as input a configuration from a variability model to then collect, adapt, and assemble relevant parts of realization artifacts.

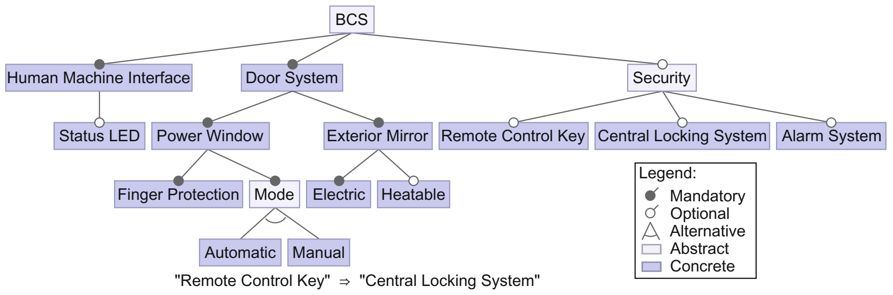

Fig. 2 Excerpt from the *BCS* feature model

The procedure of transforming a configuration into a variant of the SPL is referred to as *variant derivation*. A *variant*[3] or *product* of the SPL is the software system associated with a specific conceptual configuration. The variant derivation procedure differs greatly depending on the concrete type of variability realization mechanism employed by the SPL, which we cover in detail in Sect. 3.

2.2 Running Example Automotive Body Comfort System

To illustrate our techniques, we use a running example from an automotive *body comfort system (BCS)* SPL [27] along with its realization as statecharts, which comprises functionalities such as (automatic) power windows and exterior mirror control. In Fig. 2, we show an excerpt of the feature model for the BCS where the full SPL comprises 27 features and 11,616 valid product variants. The depicted feature model comprises different parts of the functionality that are common to all product variants, such as the car's `exterior mirror` or the `human machine interface`. In addition, different optional features exist representing additional functionality, such as `electric` or `heatable` exterior mirrors as well as a `central locking system (CLS)`.

Individual features may be implemented differently depending on the selection of other features. For example, the `CLS` feature exists in alternative implementations depending on different `power window (PW)` modes. In Fig. 3, we show two statechart implementations of the `CLS` feature consisting of the `cls_unlock` and `cls_lock` states with corresponding transitions.

The `ManPW` variant (cf. Fig. 3a) is employed with a manual power window, whereas the `AutoPW` variant (cf. Fig. 3b) is used with an automatic power window.

In terms of implementation, the main difference between the two variants is that, during a transition from `cls_unlock` to `cls_lock`, the `ManPW` variant is only disabled (i.e., `pw_enabled=false`) when the window is closed completely (i.e., `pw_pos==1`). Otherwise, it is still possible to manually close the window. However, the `AutoPW` variant is disabled independent of the position of the window, which is automatically closed by generating a corresponding command (i.e., `GEN(pw_but_up)`). We use these variants of the `CLS` in the remainder of this chapter to illustrate different variability realization mechanisms as well as our variability mining and SPL generation techniques.

[3]Note that some publications [29] use a different definition of the term *variant* to describe one concrete option for a specific variation point, for example, a specific value for a configurable parameter.

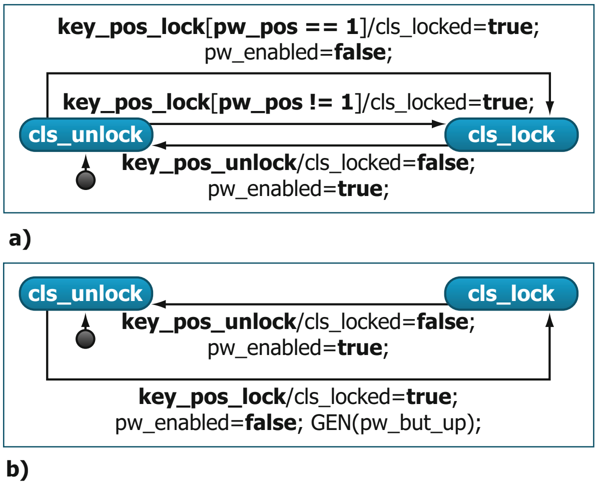
key_pos_lock[pw_pos == 1]/cls_locked=true;
pw_enabled=false;
key_pos_lock[pw_pos != 1]/cls_locked=true;
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
a)
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
key_pos_lock/cls_locked=true;
pw_enabled=false; GEN(pw_but_up);
b)

Fig. 3 Two statechart variants associated with the CLS feature. (**a**) ManPW variant of the central locking system (CLS). (**b**) AutoPW variant of the central locking system (CLS)

3 Variability Realization Mechanisms

Different techniques exist to represent changes associated with conceptual features in realization artifacts, such as source code or models. In the following, we describe the state of practice of variability realization in industry and survey the state of the art by explaining in detail different variability realization mechanisms from SPL engineering.

3.1 State of Practice in Variability Realization

In industrial practice, a couple of ad hoc variability realization mechanisms or existing language constructs and modularization concepts (not originally intended to capture variability) are used to encode variability. In the following, we summarize the most widely applied techniques for variability realization in the solution space, while the survey by Berger et al. [5] focuses on variability modeling in the problem space.

- **Clone-and-Own** (or copy-paste-modify) [12, 21]: Developers copy existing models or source code of product variants and modify the respective artifacts until a new variant is obtained. This new variant is then stored under a new name and can be deployed in the same way as existing variants. This process can be repeated for each variant to be developed. The advantage of this approach is that it is very lightweight and saves development effort in the short term. No special modeling notation or tool support is required. However, with an increasing number of variants, the variability in the set of cloned variants becomes difficult to manage as each software system has to be maintained in isolation. In particular, for debugging and maintenance, undocumented variability becomes an obstacle.

- **Conditional Compilation** [26]: In programming languages, conditional compilation techniques allow deriving the implementation for a specific code variant during compilation by appropriately selecting values for preprocessor macros. Conditional compilation is most prominently used within C/C++ where code blocks can be enclosed in `#ifdef` directives that are omitted for compilation if the corresponding constant is set to false. Conditional compilation is a widely used approach within the programming language community and offers very flexible means to obtain custom-tailored code for specific variants. However, it leads to code fragmentation and scattering of variability which is difficult to maintain and debug.
- **Variability Encoding** [44]: Variability in models or source code is encoded by standard programming/modeling language constructs that are originally intended for choice within the control flow during execution. For instance, in programming languages, variability can be encoded using if statements where the if condition is a configuration parameter, such as a specific feature. In MATLAB/Simulink, switch-case statements or variant subsystem blocks are used to capture variability [45]. While variability encoding does not require specific language or tool support to express variability, it is limited to expressing variability in software behavior (in contrast to variability in its structure), and the binding of variability is shifted to runtime which means that the complete code base has to be deployed in all cases which may be disadvantageous for resource-constrained devices or for protecting intellectual property. Additionally, choices due to variability and choices due to program behavior are mixed, which violates the separation of concerns principle and hinders maintenance and debugging.
- **Parametrization**: Variability of a system is captured by setting specific parameter values for system variables. Parametrization in the automotive domain is, for example, used in characteristic curves or maps, such as for engine control, set during calibration phases. Alternatively, electronic control units (ECUs) incorporate a set of behavioral variants that can be configured by parametrization. After the car is readily built, a parametrization string is entered such that the software variant matches the built variant of the car. This requires that all possible variants are already encoded within the ECU. The parametrization strings are often kept in spreadsheets, which complicates analyses such as finding out if the software for a specific car variant is configurable at all. Furthermore, configuration errors may only be detected during system execution, which significantly complicates debugging.
- **Components and Plug-In Frameworks** [41]: Variability on the architectural level can be represented by component or plug-in frameworks. Variants can be obtained by composing different component variants from a component library or by using different plug-ins in plug-in frameworks such as Eclipse. The advantage of component libraries and frameworks is that variability is

modularly encapsulated within components. However, variability is subject to the granularity of the components. Hence, fine-grained variability in behavior or structure cannot be captured. Instead, for each (even only fine-grained) change due to variability, a new variant of a component is needed which leads to redundancy and replicated code in the component/plug-in library.

3.2 State of the Art in Variability Realization Mechanisms

Despite ad hoc variability realization mechanisms used in industrial practice and their individual shortcomings, variability realization mechanisms of SPLs support managed reuse. These variability realization mechanisms can be distinguished into three principle groups: *annotative*, *compositional*, and *transformational* [34]. In the following sections, we elaborate on each type.

3.2.1 Annotative Variability Realization Mechanisms

Annotative variability realization mechanisms[4] [22, 34] utilize annotations to denote parts of the realization artifact that belong to a particular feature. As a consequence, with annotative variability realization mechanisms, a single artifact contains all possible variations of one realization asset affected by variability, often referred to as a *150% model*. For example, a C++ class may contain multiple definitions of a method with the same signature (similar name of the method with same number and type of parameters and return value) with different bodies for individual features, where each definition is wrapped in a preprocessor statement (`#ifdef`) that only enables the respective method when a particular feature is selected. Hence, when used in a disciplined manner, the aforementioned conditional compilation as well as variability encoding may be viewed as annotative variability realization mechanisms (see Sect. 3.1).

For annotative variability realization mechanisms, the connection between a conceptual variability model and the annotations is established through naming conventions, for example, features of a feature model may have the same name as presence variables of annotations. During variant derivation, the presence of elements from the variability model is then resolved to Boolean values for annotations. The realization artifacts are reduced from a 150% model to an artifact of the intended variant by removing those annotated parts of the realization artifact whose conditions in the annotations are not satisfied. The result is a variant containing only the intended functionality. Figure 4 depicts an example of an annotative variability realization mechanism where the two variants of Fig. 2 are represented as a 150% model and the `AutoPW` variant with automatic power window is derived as example.

[4]Annotative variability realization mechanisms are also referred to as *subtractive* or *negative*.

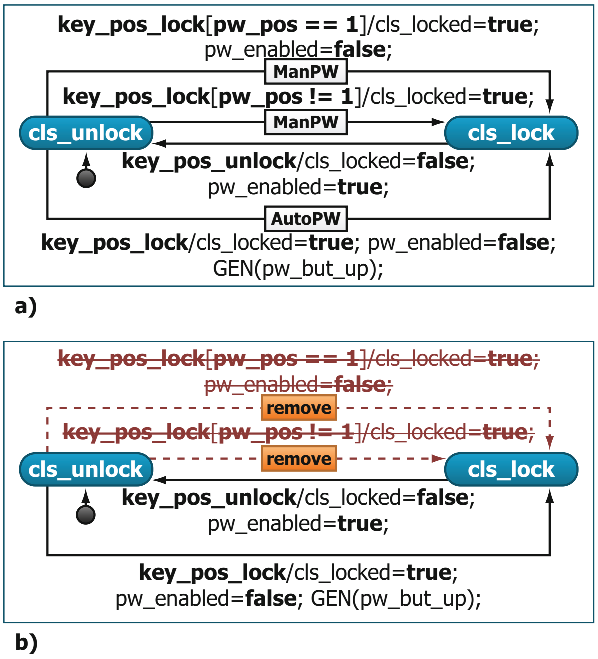
key_pos_lock[pw_pos == 1]/cls_locked=true; pw_enabled=false;
ManPW
key_pos_lock[pw_pos != 1]/cls_locked=true;
ManPW
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false; pw_enabled=true;
AutoPW
key_pos_lock/cls_locked=true; pw_enabled=false; GEN(pw_but_up);
a)
~~key_pos_lock[pw_pos == 1]/cls_locked=true; pw_enabled=false;~~
remove
~~key_pos_lock[pw_pos != 1]/cls_locked=true;~~
remove
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false; pw_enabled=true;
key_pos_lock/cls_locked=true; pw_enabled=false; GEN(pw_but_up);
b)

Fig. 4 Example of an annotative variability realization mechanism. (**a**) 150% model. (**b**) Variant derivation by removing parts of the 150% model

Annotative variability realization mechanisms pose certain requirements to be applicable: If annotations are internal to realization artifacts, constructs for adequate annotation have to either be included in the realization language (e.g., if statements) or in another embeddable language (e.g., the C++ preprocessor). Furthermore, if annotations are external to realization artifacts (e.g., through a specific annotation model), elements need to be referenceable from outside of the realization artifact. Finally, the specification of the 150% model requires full knowledge of all possible variations at design time.

A variety of tools for managing SPLs is based on an annotative variability realization mechanism: BigLever's Gears[5] [25] and pure-system's pure::variants[6] [7] are industrial tools for managing SPLs. FeatureIDE[7] [42], Clafer[8] [3], and FeatureMapper[9] [17] are tools for managing SPLs stemming from academia.

[5]http://biglever.com/solution/product.html.

[6]http://pure-systems.com/Products.html.

[7]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide.

[8]http://clafer.org.

[9]http://featuremapper.org.

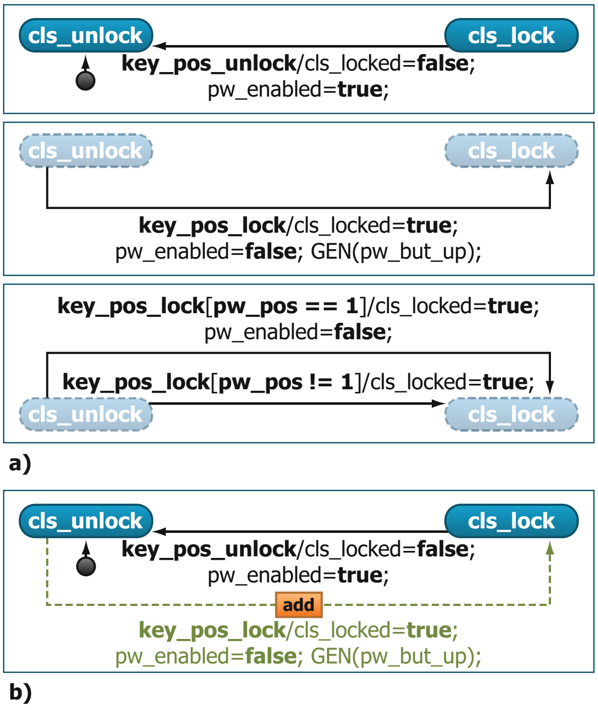
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
cls_unlock
cls_lock
key_pos_lock/cls_locked=true;
pw_enabled=false; GEN(pw_but_up);
key_pos_lock[pw_pos == 1]/cls_locked=true;
pw_enabled=false;
key_pos_lock[pw_pos != 1]/cls_locked=true;
cls_unlock
cls_lock
a)
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
add
key_pos_lock/cls_locked=true;
pw_enabled=false; GEN(pw_but_up);
b)

Fig. 5 Example of a compositional variability realization mechanism. (**a**) Core model and two units of composition. (**b**) Variant derivation by adding units of composition to the core model

3.2.2 Compositional Variability Realization Mechanisms

Compositional variability realization mechanisms[10] [4, 22, 34] represent a variable software system as a common *core model* and multiple *units of composition*. The core model comprises the realization of functionality common to all variants. The units of composition contain the realization of individual configuration options, usually on the granularity level of a single feature, but it is principally possible to define finer-grained units of composition. During variant derivation, relevant units of composition are collected and combined with the core model to form a variant containing valid realization artifacts. Figure 5 depicts an example of a compositional variability realization mechanism where the two variants of Fig. 2 are represented as a common core model with two units of composition and the `AutoPW` variant with automatic power window is derived as example.

It is possible that, in isolation, neither the core model nor the individual units of composition are valid artifacts with regard to the syntax of the language in

[10]Compositional variability realization mechanisms are also referred to as *additive* or *positive*.

which they are specified as they may define partial and incomplete information. For example, a statechart may define a core model only containing the states where the units of composition contain various transitions that will connect the states in different variants.

Compositional variability realization mechanisms require certain conditions: The functionality affected by variability has to be accessible for modification through composition, which makes certain structures of realization artifacts more favorable. For example, in source code, it may be complicated to replace fragments of methods through composition so that smaller methods are beneficial for composition. Furthermore, a component-based software architecture [41] or a plug-in framework allows for easier alignment of features with units of composition but poses restrictions on the architecture (see Sect. 3.1).

Compositional variability realization mechanisms do not necessarily have to know all possible variation in advance as new units of composition may be added later on. However, in contrast to 150% models of annotative variability realization mechanisms, which store all variations of a realization artifact in a single element, the (possibly many) units of composition of compositional variability realization mechanisms lead to an increased scattering, which may increase maintenance effort.

Compositional variability realization mechanisms are used in different approaches and tools: *feature-oriented programming (FOP)* [4] captures modifications to artifacts resulting from a different feature in a *feature module*. A feature module uses superimposition to express changes to a realization artifact by either adding new or overriding parts of existing realization artifacts or, potentially, utilizing the previous content. As an example, a feature module for a C++ class may provide an alternative implementation for an existing method and, as part of the new implementation, call the previous definition of the method. During variant derivation, the features selected in a configuration are resolved to the respective feature modules via name matching, which are then used to compose the core model of the SPL with the units of composition. FOP is implemented in various tools, such as the AHEAD Tool Suite[11] [4] or FeatureHouse[12] [2]. Furthermore, FeatureIDE [42] may be configured to use a compositional variability realization mechanism.

Aspect-oriented programming (AOP) [23] may be perceived as a compositional variability realization mechanism when employed within an SPL. An *aspect* captures (potentially cross-cutting) concerns of a software system as additions to various significant locations of the targeted realization artifact called *join points*. Individual features may be realized as aspects that are then combined with the core model of the SPL by *weaving* the additions of relevant aspects into the respective join points. AOP is utilized for the realization of variability in SPLs in various different approaches [1, 13, 15].

[11] http://cs.utexas.edu/~schwartz/ATS/fopdocs.

[12] http://infosun.fim.uni-passau.de/spl/apel/fh.

3.2.3 Transformational Variability Realization Mechanisms

Transformational variability realization mechanisms represent variabilities as transformations that restructure a *base variant* of an SPL to a specific *target variant* that constitutes the functionality associated with the selected features of one particular configuration. Sequences of calls to transformation operations may be grouped into *transformation modules* if they have a sufficiently high level of cohesion. A transformation module may realize variability of an entire feature or parts thereof. Transformations may have different complexity: On an atomic level, transformations may be perceived as addition, modification, and removal of elements of the addressed realization artifact. On a more complex level, these atomic operations may be synthesized to form compound operations, for example, to remove an element and all depending references.

During variant derivation, the base variant is copied and relevant transformation modules are collected and sequentially applied to transform the base variant to the intended target variant. In the process of transformation, newly required functionality is added, and functionality of the base variant that is rendered redundant due to the selected configuration is removed. Figure 6 depicts an example of a transformational variability realization mechanism where the `ManPW` variant with the manual power window of Fig. 2 is used as base variant with transformations to create additional variants and the `AutoPW` variant with automatic power window is derived as example.

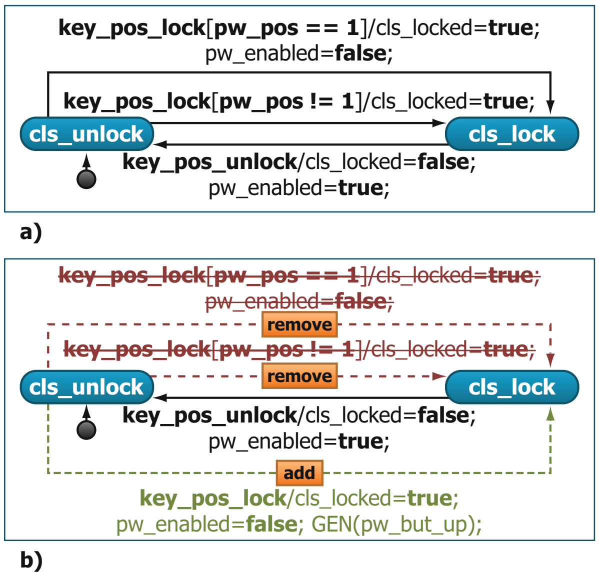
key_pos_lock[pw_pos == 1]/cls_locked=true;
pw_enabled=false;
key_pos_lock[pw_pos != 1]/cls_locked=true;
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
a)
~~key_pos_lock[pw_pos == 1]/cls_locked=true;~~
~~pw_enabled=false;~~
remove
~~key_pos_lock[pw_pos != 1]/cls_locked=true;~~
remove
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
add
key_pos_lock/cls_locked=true;
pw_enabled=false; GEN(pw_but_up);
b)

Fig. 6 Example of a transformational variability realization mechanism. (**a**) Base variant. (**b**) Variant derivation by adding, modifying, and removing elements

The base variant of an SPL may, in principal, be selected arbitrarily from the set of products. However, the choice of the base variant greatly influences the number of transformations to create the other variants and, thus, the complexity of the SPL. Hence, a number of considerations may influence which product is selected as base variant of the SPL depending on the intended use: Selecting the product most commonly purchased by customers aligns with business practices when functionality of other products is viewed as deviation from the common product. Selecting the product created first aligns with the development process as subsequent changes may directly be represented as delta modules. Selecting the product with the most features aligns with product configuration where undesired functionality may be deselected.

There are multiple approaches that can be perceived as transformational variability realization mechanisms if applied in the context of an SPL: Model transformation may be employed as transformational variability realization mechanism if all realization artifacts can be perceived as models, for example, as instances of meta-models (see Sect. 4). In this case, model transformation operations are employed to alter realization artifacts of the SPL. Model transformation operations may either be provided as *general purpose transformation operations* [35, 38] that are agnostic of the realization language or as *domain-specific transformation operations* [31] that are tailored to the respective realization language.

The *Common Variability Language (CVL)* [16] is an attempt at adding standardized variability to arbitrary modeling notations by overlaying variability information over realization assets of a software family. CVL provides a set of standard operations to manipulate realization assets of a software family in order to manifest changes associated with variability so that it can be regarded as providing a transformational variability realization mechanism. For example, the approach allows binding variability at one variation point, for example, by choice from a fixed set of options or by setting the value of a variable. Variant derivation facilities may be employed to manifest the effects of changes associated with variability in implementation artifacts.

Delta modeling [33] is the most prominent transformational variability realization mechanism for SPLs. In delta modeling, transformations are performed by invoking specific *delta operations*. Delta operations are provided by a *delta language*, which is a domain-specific variability language tied to the *source language* employed by realization artifacts it modifies, for example, *DeltaStateCharts* for statecharts. The delta language allows fine-grained control over which operations are available for realizing variability while, at the same time, allowing complex transformation operations specific to the source language of the artifacts that should be modified.

Delta modules encapsulate sequences of calls to delta operations with particularly strong cohesion, for example, because they realize a particular feature. Usually, multiple delta modules are required to realize a specific variant of an SPL. The order in which the delta modules are applied may influence the created variant. Hence, not necessarily all application orders of delta modules are valid as, for example, an element cannot be modified before it was added. To express these restrictions, *application-order constraints* may be specified, which state that a specific delta module can only be applied after another delta module was applied.

During variant derivation, first, the relevant delta modules are collected. This may be a manual selection of delta modules by a user of the SPL, or, if the SPL also employs a feature model, the selection of features of a configuration may be resolved to the set of associated delta modules. Then, the application-order constraints of delta modules are evaluated and used as input to a topological sorting algorithm, which creates one valid sequence of application. Finally, the base variant of the SPL is copied and the transformations are applied sequentially in the order determined for the delta modules to create the intended target variant of the SPL.

Transformational variability realization mechanisms have both as input and output a valid variant of the SPL. This is in contrast to annotative and compositional variability realization mechanisms, which depend on a specific software family representation as input with either a 150% model or a core model, which both potentially are not considered valid realization artifacts with regard to syntax and static semantics of the realization language. Hence, transformational variability realization mechanisms have various benefits: For one, provided that the configuration knowledge is sound, both the input and output of transformational variability realization mechanisms may be inspected with standard tools that depend on valid syntax and static semantics of the respective artifacts. In addition, the variability realization process of creating variants by starting out with one concrete software system and transforming it aligns closely with the common practice of companies to first develop an individual software solution on request and, later on, to transform it into an SPL with multiple products to sell off the shelf. Moreover, using transformations is flexible as not necessarily all features have to be known in advance and new features can be added seamlessly later on.

Finally, delta modeling as transformational realization mechanism can emulate the behavior of both annotative and compositional variability realization mechanisms when restricting the delta operations employed in delta modules to only remove or add elements, respectively. This well-formedness property is referred to as *monotonicity* of delta modules [10], and there are procedures to transform any delta-oriented SPL to a monotonous form [11]. Due to these beneficial properties, we employ delta modeling for our work and the explanations in the remainder of this chapter.

4 From Cloned Variants to Managed Software Product Lines

In this section, we explain a procedure to migrate from cloned variants in grown systems to managed reuse in SPLs. The presented approach is applicable to models of different block-based modeling languages, such as MATLAB/Simulink models, FBDs, and statecharts [18, 47, 49].

A common way to describe the notation of a modeling language is through a *metamodel* [14, 40]. In a metamodel, *metaclasses* describe the elements of the modeling language, *metareferences* the relations of these elements, and *metaattributes* the properties of each element. Concrete artifacts of the modeling language are then perceived as instances of the metamodel. The Eclipse Modeling Framework (EMF) provides *Ecore* as a notation to specify metamodels and offers a wide variety of tools on that basis: With Xtext[13] or EMFText,[14] models can be specified in a textual language, for example, source code. With GEF,[15] GMF,[16] or Graphiti,[17] it is possible to create visual editors so that models can be specified in a graphical language, for example, statecharts. With these tools, it is possible to perceive any realization language (e.g., the language for statecharts) as a metamodel and its artifacts (e.g., concrete statecharts) as models instantiating the metamodel. This has the benefit that artifacts may be defined in different representations and may stem from different sources but can still be handled uniformly as models of an explicitly defined metamodel. Predefined metamodels, grammars, and parsers exist for many popular languages to treat their source code as instances of Ecore models, such as *JaMoPP*[18] for Java or *srcML*[19] for C/C++. In addition, it is possible to define metamodels for further languages.

In Fig. 7, we show the metamodel for statecharts we utilize for the BCS running example. The metaclasses `State` and `Transition` represent the respective elements of the statechart notation, where elements of both are uniquely identified by the value of the metaattribute `id`. States may carry a `name` and transitions may specify `events` upon whose occurrence they prompt certain `actions` if the respective `guards` are satisfied. Metareferences define potential compositions of the respective elements. A `StateChart` consists of a number of `states`, where the `initialState` is designated by the specialized class `InitialState`. Furthermore, a state contains multiple `transitions` of which each references a `sourceState` and a `targetState`. Additionally, a state may be a compound in the sense that it may contain further states and, indirectly, transitions that define its behavior. Depending on the number of elements instances may reference, metareferences may be distinguished into *single-valued references* (e.g., `initialState` of `StateChart`) and *multivalued references* (e.g., `states` of `StateChart`), which is of relevance for our approach. The remaining elements of the metamodel in Fig. 7 capture variability as will be explained in Sect. 4.1. Finally, the *metamodel URI* is the unique identifier of the metamodel which can be utilized to retrieve the metamodel from a central registry, for example, when checking models for conformance with the respective metamodel. For the BCS running example, the metamodel URI is `http://www.tu-braunschweig.de/isf/states`.

[13]http://eclipse.org/Xtext.

[14]http://emftext.org/.

[15]http://eclipse.org/gef.

[16]http://eclipse.org/modeling/gmp.

[17]http://eclipse.org/graphiti.

[18]http://jamopp.org/index.php/JaMoPP.

[19]http://www.srcml.org/.

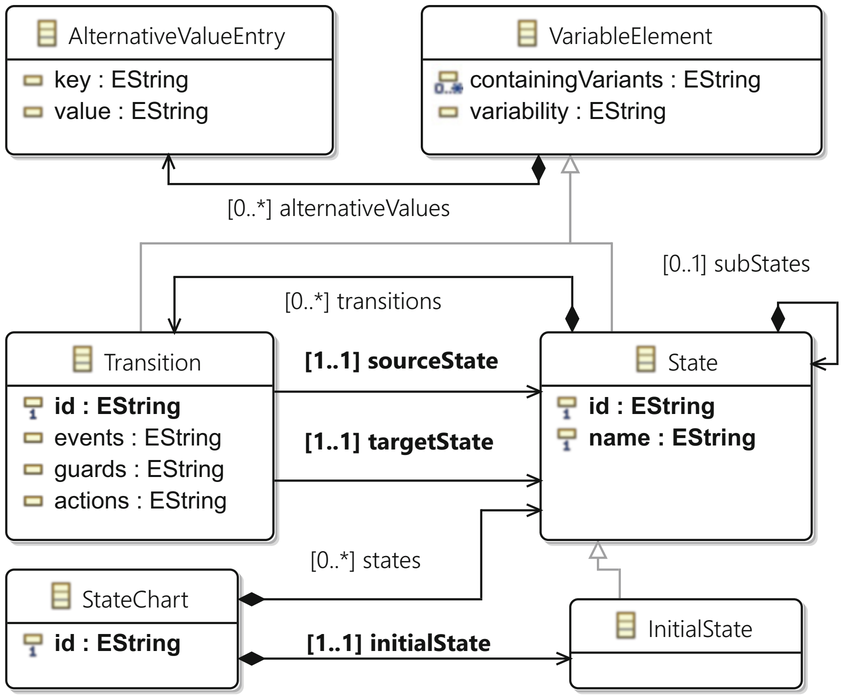

Fig. 7 Metamodel for the statechart notation used in the running example

Due to the benefits of employing metamodels, we heavily utilize model-based development. However, we do not require users of our mining and generation technology to use it as well because the model-based character is completely transparent and required inputs may be provided in textual languages. In Sect. 4.1, we explain how a set of cloned model variants can automatically be analyzed to extract variability information (i.e., common and varying parts between the variants). In Sect. 4.2, we show how to generate delta modules of a delta-oriented SPL from the cloned variants.

4.1 Mining Variability from Cloned Variants

To support transition from a set of cloned model variants to a managed reuse strategy, it is essential to identify variability relations between the variants. In Fig. 8, we show our family mining process, an approach to semiautomatically reverse-engineer variability information from a set of block-based model variants [18, 47, 49]. The approach relies on metamodeling techniques and first translates the input models in an instance of a metamodel specifically tailored to the modeling language

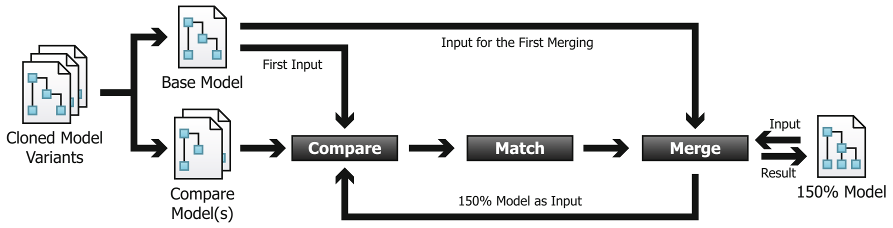

Fig. 8 Workflow for variability mining from cloned model variants

employed to realize variants, for example, for statecharts, the metamodel of Fig. 7. Furthermore, we created metamodels capable of handling *MATLAB/Simulink* models and *FBDs* [18, 47, 49]. The family mining algorithm consists of the following three phases. During the *Compare Phase* (cf. Sect. 4.1.1), models are compared and possible relations are identified. In the *Match Phase*, unambiguous one-to-one relations are selected from these comparisons. In the *Merge Phase*, the resulting relations are used to create a *150% model*.

In the metamodel for statecharts presented in Fig. 7, we provide classes `VariableElement` and `AlternativeValueEntry` to store the determined variability information: In `VariableElement`, we allow to store the identified variability for compared elements (i.e., whether they are contained in all compared variants or represent variability only present in certain variants). In addition, we store the model variants containing the corresponding elements. In `AlternativeValueEntry`, we allow to define mappings from model names to alternative values (e.g., when we identify state names with minor deviations in two compared variants).

The following sections explain each of the steps for mining variability from cloned variants in details.

4.1.1 Compare Phase

Our family mining approach is realized in a pairwise manner and, thus, iteratively compares two models at a time. The algorithm starts comparing the models by selecting a *base model* (e.g., the smallest variant) and regards all remaining models as *compare models*. Next, the algorithm compares the selected base model with one of the compare models by analyzing the dataflow in the model. For our running example, we compare the two statechart implementation variants of the `CLS` feature (cf. Fig. 3). Starting from the *start elements* on the highest hierarchy level where data is introduced to the model or where the execution is started (i.e., the `cls_unlock` initial states from the variants with manual and automatic central locking system in Fig. 3), the algorithm separates the currently analyzed model hierarchy for both models into *stages*. These stages are created by analyzing the dataflow and contain only elements that are separated by the same number of edges (e.g., transitions in statecharts) compared to the start elements. For instance, the compare algorithm creates stage `S0` with state `cls_unlock` and stage `S1` with state `cls_lock` for both compared variants in Fig. 3. Depending on the employed modeling language, not only the analyzed model *nodes* (e.g., states for statecharts) are relevant but their connecting *edges* (e.g., transitions for statecharts) contain additional information worth considering during comparison. For example, as transitions in statecharts contain important execution information, they should be analyzed during family mining. Thus, two additional stages are created for both variants from Fig. 3 containing the outgoing transitions from the states `cls_unlock` and `cls_lock`, respectively.

Next, each stage from the base model is compared with its counterpart from the compare model by iterating and inspecting all possible combinations of the contained elements. For each comparison between two model elements, a so-called *compare element* is created storing the compared elements together with a *similarity value* calculated according to a user-defined *metric* [48, 49]. Such a metric allows to assign different weights to the properties of compared elements and, thus, allows to rank their influence on the model functionality. For instance, when comparing two transitions from Fig. 3, we could assign a higher weight to *actions* triggered by the transitions than to their *events* (i.e., the events triggering the transition's execution) and *guards* (i.e., conditions that have to be fulfilled in order to execute the transition) because actions trigger new events and the execution of other transitions. Consequently, these actions have a high influence on the semantics of a particular statechart. As the metric is adjustable to different settings, users can easily modify the weights to different needs. During the comparison of two elements, the similarity value is calculated by summing up the metric's weights according to the elements' similarity. To allow comparison of the calculated similarity values, we normalize the metric's values to the interval [0..1]. A concrete example for such a metric can be found in [48]. In cases where no counterpart exists for the comparison of elements in a stage, we create comparisons with *null*, which indicates that the respective element is optional as it is not present in some of the variants.

Our procedure natively supports comparison of hierarchical models by recursively starting the algorithm when comparing two hierarchical elements (e.g., two hierarchical states in statecharts or two subsystems in *MATLAB/Simulink*) [49]. Using this approach, we are able to calculate the overall similarity value of compared hierarchical elements by averaging the similarity value of their sub-elements.

4.1.2 Match Phase

The resulting list of compare elements might contain ambiguous relations between the compared model elements, because, during the comparison of stages, multiple combinations with the same model element might be created. For instance, during the comparison of the stages containing the transitions going from the `cls_unlock` state to the `cls_lock` state, the algorithm creates two compare elements. Both elements contain the corresponding transition from Fig. 3b comparing it with both possible variants from Fig. 3a.

As these ambiguous relations hinder to identify distinct one-to-one variability relations between the compared model elements, the algorithm traverses the list of all possible compare elements. For each contained compare element, it identifies all other compare elements sharing at least one of the compared elements (i.e., the element from the base model or the compare model). Afterward, the algorithm identifies a distinct match for these compare elements by selecting the compare element with the highest similarity value. All ruled-out compare elements are deleted and the algorithm continues until no unmatched elements are left. Thus, in our example, the algorithm matches the transition between the states `cls_unlock` and `cls_lock` in Fig. 3b with the transition containing the `pw_pos==1` guard in Fig. 3a because they have a higher similarity compared to the other possible transition containing the `pw_pos!=1` guard (i.e., a higher number of statements match). In case the algorithm cannot identify distinct relations for a compare element, it first sorts the conflicting elements to the end of the list. Using this approach, the algorithm tries to implicitly solve ambiguous relations automatically by first matching other elements. However, in some cases user intervention may be necessary to resolve the conflict. Elements that were ruled out completely from all compare elements (i.e., they have no matching partner) are regarded as optional elements and are added to the final list of matches in compare elements. For example, the ruled-out transition between the states `cls_unlock` and `cls_lock` in Fig. 3a has to be added as such an optional compare element to not lose information contained in the compared variants.

4.1.3 Merge Phase

The resulting list of distinctively matched compare elements can now be used to create a 150% model storing all implementation artifacts from the compared variants together with their identified variability (i.e., how the elements are related and in which models they are contained). The algorithm processes the list of distinct matches and creates the 150% model by merging the compare elements with a copy of the base model. For the merging process, we define the following mapping function to classify variability of compared elements:

$$rel(A, B) = \begin{pmatrix} similarity >= 0.95 & mandatory \\ 0 < similarity < 0.95 & alternative \\ similarity = 0 & optional \end{pmatrix} \tag{1}$$

This mapping function is adjustable to user preferences and defines default thresholds that were identified during an impact analysis of differing properties in compared elements with additional interviews on how similar these elements are regarded according to domain experts [48, 49]. The threshold of 95% categorizes two compared elements as *mandatory* (i.e., they are regarded as equal). However, we allow minor deviations between the elements because of the 5% interval up to 100% equality. For example, this allows us to regard elements as mandatory despite minor differences in their names. Mandatory elements do not have to be merged into the 150% model as they are already contained in the base model copy. However, we have to annotate differing values for the properties where they are not equal (e.g., the changed name). Otherwise, we lose information when creating the 150% model. Compare elements with a similarity value of 0% are regarded as *optional* (i.e., they are only contained in some of the variants) as they have no counterpart. Depending on whether the element was already contained in the base model copy, we have to copy the corresponding elements to the 150% model. All elements with a similarity value between the mandatory and optional threshold are regarded as *alternatives* (i.e., all variants contain exactly one of the possible alternative elements). Here, only the element that was not contained previously in the base model copy has to be copied to the 150% model. For all elements in the 150% model, we compare the model containing the corresponding element and explicitly store the variability identified according to the thresholds in Eq. 1. The resulting 150% model is used as an input for the iterative comparison with the next model.

In Fig. 9, we present the 150% model for our running example. As we can see, the algorithm correctly identified and annotated the models containing the different transitions (i.e., either the `ManPW` variant or the `AutoPW` variant). For readability reasons, we neglected the explicit variability annotations in Fig. 9. However, the algorithm correctly identifies that the annotated transition from the `AutoPW` variant is an alternative to the annotated transition with the `pw_pos==1` guard from the `ManPW` variant. The remaining annotated transition with the `pw_pos!=1` guard is identified as an optional element. All elements without annotations are regarded as mandatory as they are contained in both variants.

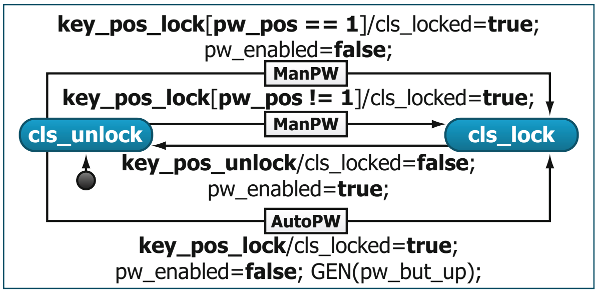
key_pos_lock[pw_pos == 1]/cls_locked=true;
pw_enabled=false;
ManPW
key_pos_lock[pw_pos != 1]/cls_locked=true;
ManPW
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false;
pw_enabled=true;
AutoPW
key_pos_lock/cls_locked=true;
pw_enabled=false; GEN(pw_but_up);

Fig. 9 Part of the 150% model showing the variability of the compared `CLS` variants

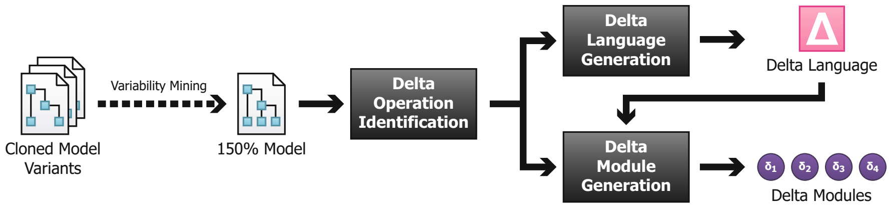

Fig. 10 Workflow for the generation of a delta-oriented SPL consisting of a delta language and multiple delta modules

4.2 *Generating a Delta-Oriented Software Product Line*

After creating a 150% model, we are now able to generate an SPL for managed reuse. In particular, we generate a delta-oriented SPL consisting of a delta language tailored specifically to the modeling language used to realize implementation artifacts and a set of delta modules using this delta language to specify the transformations to create the original set of variants from the base variant. Figure 10 depicts the workflow for SPL generation.

The workflow consists of three steps. First, the algorithm decides on which transformation operations are required by processing annotations of the 150% model (cf. Sect. 4.2.1). Then, the algorithm uses these operations to generate a delta language for the modeling language of the inspected variants (cf. Sect. 4.2.2).[20] Finally, using this delta language, delta modules are generated that describe the transformations from the base variant to all of the initially analyzed variants (cf. Sect. 4.2.3).

4.2.1 Delta Operation Identification

As we apply delta modeling, we intend to transform a *base variant (*`BV`*)* to a *target variant (*`TV`*)*. Our process allows users to freely select any variant contained in the input 150% model as `BV`. To determine appropriate delta operation calls for transformation to the respective `TV`, we have to decide for each element from the generated 150% model whether it has to be *added*, *removed*, or *modified* by calling the respective delta operation.

To identify appropriate operations to call, we use a decision process: The algorithm analyzes the annotations in the 150% model to identify in which variants the currently considered element is contained. In case it *is not* contained in `BV` and `TV` or it *is* contained in both variants but is regarded as *identical*, the process decides to not generate a delta operation. Two elements are regarded as identical

[20]To be exact, the algorithm generates a delta dialect, which can be used to generate the delta language (cf. Sect. 5).

Table 1 Decisions for Fig. 9 with `BV=ManPW` and `TV=AutoPW`

Element	Decision process
Transition 4	$\texttt{Decide} \xrightarrow{!in\ BV} \texttt{!in BV} \xrightarrow{in\ TV} \texttt{Add}$
Source state for transition 4	$\texttt{Decide} \xrightarrow{!in\ BV} \texttt{!in BV} \xrightarrow{in\ TV} \texttt{Set}$
Transition 3	$\texttt{Decide} \xrightarrow{in\ BV} \texttt{in BV} \xrightarrow[identical]{in\ TV\ \&} \texttt{Nothing}$
Source state for transition 2	$\texttt{Decide} \xrightarrow{in\ BV} \texttt{in BV} \xrightarrow{!in\ TV} \texttt{Unset}$
Transition 2	$\texttt{Decide} \xrightarrow{in\ BV} \texttt{in BV} \xrightarrow{!in\ TV} \texttt{Remove}$

if the family mining process marked them as mandatory without any alternative properties (e.g., for elements with minor name differences). In all other cases, we have to generate a delta operation for the transformation between both variants. If the analyzed element is only contained in `BV` and does not have an annotation for `TV`, it has to be *unset/removed* to generate the final variant. Similarly, the element has to be *set/added* when it is only contained in `TV` and not in `BV`. For elements that are contained in both variants but are *not similar*, the original element has to be *modified*.

In Table 1, we present the identification of delta operation calls for the 150% model in Fig. 9 with `BV=ManPW` (i.e., the variant in Fig. 3a) and `TV=AutoPW` (i.e., the variant in Fig. 3b). For space reasons, we limit the table to demonstrate an example for each type of identified delta operation (i.e., *add*, *set*, *remove*, *unset*, and *nothing*). However, for complete transformations, all elements have to be analyzed by the delta operation identification process. To allow easier reasoning on the decisions in Table 1, we have numbered the transitions of Fig. 9 consecutively from top (i.e., the transition with the `pw_pos==1` guard) to bottom (i.e., the transition with the `AutoPW` annotation). As we can see, the algorithm identifies correctly that transition `4` is not contained in `ManPW` and has to be *added* during a transformation to `AutoPW`. Similarly, transition `2` has to be *removed* as it is not contained in `AutoPW`. In addition, the algorithm identifies that the source states for transition `4` and `2` have to be *set* and *unset*, respectively. These operations are required to update the corresponding references because of the metamodel structure used to store the statechart variants in Fig. 7 (i.e., the `sourceState` reference is used to store the corresponding state). For transition `3`, the delta operation identification process correctly determines that the element does not have to be transformed as it is contained in both variants.

At present, we analyze atomic differences and identify atomic delta operations that modify a single value or reference. However, in the future we plan on identifying semantically richer delta operations by utilizing domain knowledge or knowledge of the semantics of the realization language, for example, a delta operation for statecharts that can remove a state as well as all its incoming and outgoing transitions to preserve well-formedness of the statechart.

4.2.2 Delta Language Generation

Using the described decision process, it is possible to determine which delta operations have to be called to perform the transformations to retrieve target variants `TVs` for all inspected cloned variants from the selected base variant `BV`. Depending on the selected `BV`, different delta languages are generated as only delta operations are considered that are needed to transform `BV` to the analyzed `TVs`. For example, when generating a delta language for the comparison of `BV=ManPW` with another `TV` that only contains additional states and transitions, no *remove* operation for transitions is generated. For each decision returned by the algorithm, we generate a corresponding delta operation and store it in a set to prevent generation of redundant operations. For instance, in our running example, this prevents the generation of multiple delta operations to *add* transitions to states.

In Listing 1, we present an excerpt from the delta language generated for the 150% model in Fig. 9 with `BV=ManPW` and `TV=AutoPW`. The excerpt contains the delta operations generated for the decisions in Table 1. This delta language was generated with our tool suite DeltaEcore (cf. Sect. 5) using information on the required type of delta operation, the transformed reference or attribute, and the containing class. The automatically generated names describe the functionality of the corresponding operation in a unique and descriptive way but they may be changed manually without impacting further generation (cf. Sect. 4.2.3)

4.2.3 Delta Module Generation

Using the generated delta language, it is now possible to define delta modules that contain transformations from one variant to another. Our algorithm automatically generates appropriate delta modules for the variability identified using the family mining algorithm (cf. Sect. 4.1). To start the delta module generation, our algorithm expects the selected `BV` and `TVs` from the 150% model generated during family

```
deltaDialect {
 configuration:
  metaModel: <http://www.tu-braunschweig.de/isf/states>;

 deltaOperations:
  addOperation addTransitionToTransitionsOfState(
     Transition value, State [transitions] element);
  setOperation setSourceStateOfTransition(State value,
     Transition [sourceState] element);
  removeOperation removeTransitionFromTransitionsOfState(
     Transition value, State [transitions] element);
  unsetOperation unsetSourceStateOfTransition(
     Transition [sourceState] element);
  // ...
}
```

Listing 1 Excerpt from the delta language generated for the running example

mining as well as a delta language providing operations to transform to the inspected variants. The used delta language could either be generated automatically using the approach described in Sect. 4.2.2 or could be specified manually.

Using the selected BV, the algorithm analyzes the decision of each element from the currently considered TV to identify the appropriate delta operation to apply. For this procedure, the adequate delta operation is determined from the provided delta language by looking up its type (e.g., *set*, *modify*) and the metamodel element it addresses as parameter. First, the number of possible delta operations to apply is reduced by filtering out all delta operations whose type does not conform with the decision. For example, the *set* decision in Table 1 reduces the number of possible operations for the delta language in Listing 1 to exactly one remaining element (i.e., the setSourceStateOfTransition operation). Then, the algorithm compares the references or attributes transformed by the remaining delta operations and the modified types with the needed transformation according to the generated decision (i.e., in our example setting the *sourceState* reference in the Transition class). Only after these additional checks, a corresponding delta operation call is generated to realize the needed transformation. It is worth noting that determining the respective delta operation to call is independent of the name of the operation so that the latter may be chosen freely before generation.

All generated delta operation calls for a transformation from a selected BV to a TV are stored in delta modules. In case of *add* operation calls, the corresponding constructor calls to create the element to be added are generated automatically and are also stored in the delta modules. In Listing 2, we show an excerpt for the delta module to transform BV=ManPW to TV=AutoPW. The excerpt contains all delta operation calls with corresponding constructor calls for the decisions from Table 1 (cf. Sect. 5.2).

Using the generated delta modules, it is now possible to derive variants from the defined BV that correspond to all originally inspected cloned variants. In the following Sect. 5, we explain our implementation of the tool suite DeltaEcore, which is used for the generation process and which allows automatic variant derivation from the specified delta modules.

```
delta "CLS-ManPW->CLS-AutoPW"
 dialect <http://www.tu-braunschweig.de/isf/states>
 modifies <CLS-ManPW.statechart> {
   unsetSourceStateOfTransition(<trans3>);
   removeTransitionFromTransitionsOfState(<trans3>,
      <cls_unlock>);
   Transition t = new Transition(id: "trans1",
      events: "key_pos_lock", actions: "cls_locked = true;
      pw_enabled = false; GEN(pw_but_up);");
   addTransitionToTransitionsOfState(t, <unlock>);
   setSourceStateOfTransition(<unlock>, <trans1>);
   // ...
}
```

Listing 2 Excerpt from the delta module generated for the running example

5 Realization as Tool Suite DeltaEcore

DeltaEcore[21] [36, 37] is a tool suite for variability management using the transformational variability realization mechanism delta modeling. It employs a model-based development process, which allows the tool suite to be tailored to specific implementation languages with low effort. DeltaEcore has three major application areas: *delta language creation* to adapt the tool suite to work with specific realization languages, *software product line definition* to apply a managed reuse strategy to a family of related software systems, and *variant derivation* to generate individual software products from the software product line.[22] Figure 11 provides an overview and the following sections elaborate on each of these application areas in detail.

5.1 Delta Language Creation

Before creating an SPL with DeltaEcore, suitable delta languages for all *source languages* used for the implementation have to be created, for example, a delta language for statecharts. A delta language provides dedicated operations to alter artifacts of the source language and, thereby, governs the level of control to these artifacts, for example, by providing operations to change the transitions of a state but not its id.

DeltaEcore assumes source languages to be available as an Ecore-based metamodel where the source language's elements are represented as metaclasses with references and attributes, which is feasible for both textual and graphical languages. For example, the statechart notation used throughout the running example is defined by the metamodel depicted in Fig. 7.

A delta language in DeltaEcore consists of two parts: the *common base delta language* and a *delta dialect*. The common base delta language is agnostic of the source language and defines language constructs shared by all delta languages, such as the definition of variables or requiring other delta modules. A delta dialect is specific to a source language as it references elements of the source language's metamodel to define delta operations for the respective source language. When specifying a delta module to alter an artifact of a specific source language, DeltaEcore combines the common base delta language with the respective delta dialect to provide an appropriate delta language.

The common base delta language is provided entirely by DeltaEcore. However, the delta dialect has to be defined once for each source language, for example, a

[21] http://deltaecore.org.

[22] In this chapter, we focus on functionality of DeltaEcore regarding delta modeling. In addition, DeltaEcore also allows for a seamless integration of feature models, provides a graphical editor and configurator, supports integrated management of SPL evolution, and may be interfaced with other tools, such as FeatureIDE [42].

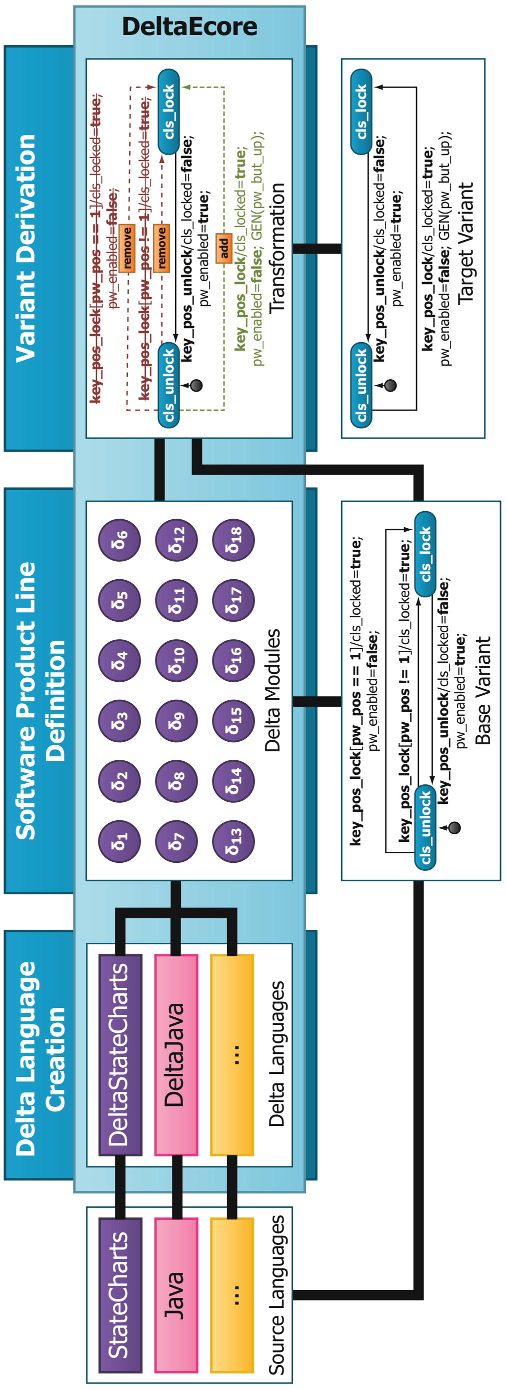
DeltaEcore
Variant Derivation
Software Product Line Definition
Delta Language Creation
StateCharts
Java
...
Source Languages
DeltaStateCharts
DeltaJava
...
Delta Languages
δ1 δ2 δ3 δ4 δ5 δ6
δ7 δ8 δ9 δ10 δ11 δ12
δ13 δ14 δ15 δ16 δ17 δ18
Delta Modules
key_pos_lock[pw_pos == 1]/cls_locked=true; pw_enabled=false;
key_pos_lock[pw_pos != 1]/cls_locked=true;
cls_unlock
cls_lock
key_pos_unlock/cls_locked=false; pw_enabled=true;
Base Variant
~~key_pos_lock[pw_pos == 1]/cls_locked=true; pw_enabled=false;~~
remove
~~key_pos_lock[pw_pos != 1]/cls_locked=true;~~
remove
add
key_pos_lock/cls_locked=true; pw_enabled=false; GEN(pw_but_up);
Transformation
key_pos_lock/cls_locked=true; pw_enabled=false; GEN(pw_but_up);
Target Variant

Fig. 11 Overview of DeltaEcore's major application areas for delta modeling

delta dialect for statecharts by specifying the signatures of the delta operations that should be provided. As the principle nature of altering implementation artifacts in delta modules is similar even across different languages, DeltaEcore provides seven types of *standard delta operations*:

- *Set* and *unset* operations alter values of single-valued references, such as the reference `initialState` of `StateChart`, by supplying a new value or resetting the reference to its default value, respectively.
- *Add* and *remove* operations alter values of multivalued references, such as the reference `states` of `StateChart`, by adding or removing a value to the set of values, respectively. *Insert* operations alter values of multivalued references with ordered values by adding a value to the ordered set of values at a specific position.
- *Modify* operations alter values of attributes, such as the attribute `name` of `State`, by supplying a new value, which, in contrast to altering values of references, is guaranteed to be free of side effects.
- *Detach* operations remove an element from its container so that it can be deleted from the model upon save if no other references to the element exist.[23]

Due to their uniform definition, the standard delta operations of DeltaEcore have defined semantics in terms of how they affect the artifacts of the source language. In addition to standard delta operations, it is also possible to supply *custom delta operations* with user-defined semantics, for example, to realize complex operations specific to the source language, such as removing a state along with all its incoming and outgoing transitions. As an example, we have already presented the delta dialect for the statechart notation used throughout the chapter in Listing 1 in Sect. 4.2.2.

A delta dialect may be defined in multiple ways: First, it may be defined entirely manually. Second, it may be generated by DeltaEcore by analyzing the structure of the source language's metamodel for suitable delta operations [36]. Third, it may be generated as part of mining variability of cloned variants as presented in Sect. 4.2.2 for statecharts. Furthermore, combinations of these approaches are also possible so that a generated delta dialect may be refined manually to supply specific user-defined operations. In combination with the common base delta language, the definition of a delta dialect suffices to create a delta language tailored to a specific source language.

[23]We refrained from defining a delete operation due to its potentially cross-cutting effects when resetting all references to the deleted element. However, a delete operation can be supplied manually with minimal effort if suitable for the source language.

5.2 Software Product Line Definition

To apply a managed reuse strategy to a family of related software systems, DeltaEcore supports the definition of an SPL based on delta modeling. For this purpose, one of the products of the family of software systems is designated as base variant, and all other variants are described in terms of transformations realizing the differences of the implementation artifacts to those of the base variant, which are captured in delta modules.

A delta module modifies a realization artifact (e.g., a statechart) through a sequence of calls to delta operations to add, modify, and remove elements. In DeltaEcore, the delta operations available for altering a realization artifact are provided by the delta dialect as part of the delta language for the respective source language (e.g., the delta dialect for statecharts).

As an example, we have already presented a delta module in Listing 2 in Sect. 4.2.3. In line 2, the delta dialect for the source language of the artifact to be altered is set by providing the URI of the language's metamodel, which DeltaEcore resolves to the appropriate delta dialect. In line 3, the statechart realizing the manual power window, which serves as base variant, is referenced to be modified by the delta module. In lines 4–12, a sequence of calls to delta operations alters the statechart to contain functionality related to the automatic power window.

To create complex variants, multiple delta modules may be used where each one encapsulates strongly coherent changes, for example, to realize one feature of the SPL. However, delta modules may not be completely independent of one another, for example, when one delta module alters an element that is created by another delta module. For this purpose, DeltaEcore allows for delta modules to specify *delta module dependencies*, which state that a delta module requires another delta module to be applied before its transformations can be invoked.

DeltaEcore also allows for specifying *application-order constraints*, which state that a certain delta module may only be applied after another delta module was applied. In contrast to delta module dependencies, application-order constraints do not entail that the referenced delta module is inevitably necessary so that application-order constraints are only evaluated should both delta modules be selected explicitly.

The set of all delta modules and their application-order constraints comprise the SPL, which allows for managed reuse and creation of individual products through variant derivation.

5.3 Variant Derivation

To create a concrete software product of the SPL defined in DeltaEcore, a variant derivation has to be performed. For this purpose, a set of delta modules has to be selected, which is then applied in a suitable order to transform a base variant to a target variant containing the intended functionality.

The initial set of delta modules is supplied by a user through selecting those delta modules that are associated with the functionality for the software product that differs from the base variant, for example, to enable the automatic power window.[24] DeltaEcore automatically completes the initial set of delta modules by adding all (transitively) required delta modules.

To ensure deterministic variant derivation, a valid application sequence for the relevant delta modules has to be determined. For this purpose, DeltaEcore performs a topological sorting, which takes into account delta module dependencies as well as application-order constraints posed upon those delta modules to determine an application sequence that satisfies all constraints.

DeltaEcore then copies the base variant of the SPL and applies the delta modules in the determined sequence to perform the transformations described by calls to delta operations within each delta module. The result is the target variant of the SPL containing the functionality of the intended software product. In DeltaEcore, the variant derivation procedure is fully automated so that users of the SPL only need to supply the initial set of delta modules intended to create a specific software product. Furthermore, it is also principally possible to use delta modules to perform changes specific to individual customers, for example, to customize a previously generated variant.

With the capacities of DeltaEcore for delta language creation, software product line definition, and variant derivation, it is possible to develop a set of closely related software systems as delta-oriented SPL. Combined with the variability mining and SPL generation approaches we presented in Sect. 4, it is possible to seamlessly migrate from the industrial practice of clone-and-own to a managed reuse strategy using an SPL.

6 Conclusion

In this chapter, we have demonstrated an approach for seamless transition from the industrial practice of creating software variants through clone-and-own to a managed reuse strategy with a delta-oriented SPL. We first reviewed the state of practice and state of the art in variability realization mechanisms. We then introduced our procedure for variability mining from variants created through clone-and-own to retrieve previously unavailable variability information. We demonstrated how our procedure generates a delta-oriented SPL from the mined variability information. With the resulting delta-oriented SPL, it is possible to maintain and create a large set of variants from a single set of managed variable artifacts.

In the future, we plan on incorporating domain knowledge to generate semantically richer delta operations to be used within the generated delta language and

[24]If a feature model is supplied, it is further possible to select a configuration from the feature model and have DeltaEcore resolve the selected features to the respective associated delta modules.

associated delta modules. Furthermore, we will also investigate how to generate an initial feature model to be used with the delta modules to also represent the problem space of the SPL in order to further increase usefulness for the industry.

References

1. Alves V, Matos P, Cole L, Vasconcelos A, Borba P, Ramalho G (2007) Extracting and evolving code in product lines with aspect-oriented programming. In: Transactions on aspect-oriented software development IV. Springer, Berlin, pp 117–142
2. Apel S, Kästner C (2009) An overview of feature-oriented software development. J Object Technol 8(5):49–84
3. Bąk K, Czarnecki K, Wąsowski A (2011) Feature and meta-models in Clafer: mixed, specialized, and coupled. In: Proceedings of the international conference on software language engineering (SLE), SLE '11. Springer, Berlin, pp 102–122
4. Batory D (2004) Feature-oriented programming and the AHEAD tool suite. In: Proceedings of the international conference on software engineering (ICSE), ICSE '04. IEEE, Piscataway, pp 702–703
5. Berger T, Rublack R, Nair D, Atlee JM, Becker M, Czarnecki K, Wąsowski A (2013) A survey of variability modeling in industrial practice. In: Proceedings of the international workshop on variability modeling in software-intensive systems (VaMoS), VaMoS '13. ACM, New York, pp 7:1–7:8
6. Berger T, Lettner D, Rubin J, Grünbacher P, Silva A, Becker M, Chechik M, Czarnecki K (2015) What is a feature?: a qualitative study of features in industrial software product lines. In: Proceedings of the international software product line conference (SPLC), SPLC '15. ACM, New York, pp 16–25
7. Beuche D (2012) Modeling and building software product lines with pure::variants. In: Proceedings of the international software product line conference (SPLC), SPLC '12. ACM, New York, pp 255–255
8. Clements PC, Northrop LM (2001) Software product lines: practices and patterns. Addison-Wesley, Boston
9. Czarnecki K, Eisenecker UW (2000) Generative programming: methods, tools, and applications. Addison-Wesley, Boston
10. Damiani F, Lienhardt M (2016) On type checking delta-oriented product lines. In: Proceedings of the international conference on integrated formal methods (iFM), iFM '16. Springer, Berlin, pp 47–62
11. Damiani F, Lienhardt M (2016) Refactoring delta oriented product lines to enforce guidelines for efficient type-checking. In: Proceedings of the international symposium on leveraging applications of formal methods, verification and validation (ISoLA), ISoLA'16. Springer, Berlin
12. Dubinsky Y, Rubin J, Berger T, Duszynski S, Becker M, Czarnecki K (2013) An exploratory study of cloning in industrial software product lines. In: Proceedings of the European conference on software maintenance and reengineering (CSMR), CSMR '13. IEEE, Piscataway, pp 25–34
13. Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Dantas F (2008) Evolving software product lines with aspects. In: Proceedings of the international conference on software engineering (ICSE), ICSE '08. IEEE, Piscataway, pp 261–270
14. Greenfield J, Short K (2003) Software factories: assembling applications with patterns, models, frameworks and tools. In: Proceedings of the international conference on object-oriented programming, systems, languages and applications (OOPSLA), OOPSLA '03. ACM, New York, pp 16–27

15. Groher I, Voelter M (2009) Aspect-oriented model-driven software product line engineering. In: Transactions on aspect-oriented software development VI. Springer, Berlin, pp 111–152
16. Haugen Ø, Møller-Pedersen B, Oldevik J, Olsen GK, Svendsen A (2008) Adding standardized variability to domain specific languages. In: Proceedings of the international software product line conference (SPLC), SPLC '08. IEEE, Piscataway, pp 139–148
17. Heidenreich F, Kopcsek J, Wende C (2008) FeatureMapper: mapping features to models. In: Proceedings of the international conference on software engineering (ICSE), ICSE '08. ACM, New York
18. Holthusen S, Wille D, Legat C, Beddig S, Schaefer I, Vogel-Heuser B (2014) Family model mining for function block diagrams in automation software. In: Proceedings of the international workshop on reverse variability engineering (REVE), SPLC '14. ACM, New York, pp 36–43
19. International Electrotechnical Commission (2009) Programmable logic controllers – part 3: programming languages. IEC61131-3 Standard
20. Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Carnegie-Mellon University Software Engineering Institute
21. Kapser C, Godfrey MW (2006) "Cloning Considered Harmful" considered harmful. In: Proceedings of the working conference on reverse engineering (WCRE), WCRE '06. IEEE, Piscataway, pp 19–28
22. Kästner C, Apel S, Kuhlemann M (2008) Granularity in software product lines. In: Proceedings of the international conference on software engineering (ICSE), ICSE '08. ACM, New York, pp 311–320
23. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J (1997) Aspect-oriented programming. ECOOP '97. Springer, Berlin
24. Krueger C (2002) Variation management for software production lines. In: Software product lines. Springer, Berlin, pp 37–48
25. Krueger CW (2008) The Biglever software gears unified software product line engineering framework. In: Proceedings of the international software product line conference (SPLC), SPLC '08. IEEE, Piscataway, pp 353–353
26. Liebig J, Apel S, Lengauer C, Kästner C, Schulze M (2010) An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the international conference on software engineering (ICSE). ACM, New York, pp 105–114
27. Lity S, Lachmann R, Lochau M, Schaefer I (2012) Delta-oriented software product line test models – the body comfort system case study. Tech. Rep. 2012-07, Technische Universität Braunschweig, Braunschweig
28. Muthig D, Atkinson C (2002) Model-driven product line architectures. In: Software product lines. Springer, Berlin, pp 110–129
29. Pohl K, Böckle G, van der Linden FJ (2005) Software product line engineering: foundations, principles and techniques. Springer, Berlin
30. Rubin J, Chechik M (2013) A survey of feature location techniques. In: Domain engineering: product lines, languages, and conceptual models. Springer, Berlin, pp 29–58
31. Rumpe B, Weisemöller I (2011) A domain specific transformation language. In: Proceedings of the international workshop on models and evolution (ME), ME '11
32. Ryssel U, Ploennigs J, Kabitzsch K (2011) Extraction of feature models from formal contexts. In: Proceedings of the international software product line conference (SPLC), SPLC '11. ACM, New York, pp 4:1–4:8
33. Schaefer I, Bettini L, Bono V, Damiani F, Tanzarella N (2010) Delta-oriented programming of software product lines. In: Software product lines: going beyond. Lecture notes in computer science, vol 6287. Springer, Berlin, pp 77–91
34. Schaefer I, Rabiser R, Clarke D, Bettini L, Benavides D, Botterweck G, Pathak A, Trujillo S, Villela K (2012) Software diversity: state of the art and perspectives. Int J Softw Tools Technol Transfer 14(5):477–495
35. Schmidt DC (2006) Model-driven engineering. Computer 39(2):25

36. Seidl C, Schaefer I, Aßmann U (2014) DeltaEcore – a model-based delta language generation framework. In: Modellierung, Modellierung'14, pp 81–96
37. Seidl C, Schaefer I, Aßmann U (2014) Integrated management of variability in space and time in software families. In: Proceedings of the international software product line conference (SPLC), SPLC '14. ACM, New York
38. Sendall S, Kozaczynski W (2003) Model transformation the heart and soul of model-driven software development. Tech. rep., Microsoft
39. She S, Lotufo R, Berger T, Wasowski A, Czarnecki K (2011) Reverse engineering feature models. In: Proceedings of the international conference on software engineering (ICSE), ICSE '11. IEEE, Piscataway, pp 461–470
40. Steinberg D, Budinsky F, Paternostro M, Merks E (2008) Eclipse modeling framework, 2nd edn. Addison-Wesley, Boston
41. Szyperski CA (1998) Component software - beyond object-oriented programming. Addison-Wesley, Boston
42. Thüm T, Kästner C, Benduhn F, Meinicke J, Saake G, Leich T (2014) FeatureIDE: an extensible framework for feature-oriented software development. Sci Comput Program 79:70–85
43. van der Linden F, Schmid K, Rommes E (2010) Software product lines in action: the best industrial practice in product line engineering. Springer, Berlin
44. von Rhein A, Thüm T, Schaefer I, Liebig J, Apel S (2016) Variability encoding: from compile-time to load-time variability. J Log Algebr Methods Program 85(1):125–145
45. Weiland J, Manhart P (2014) A classification of modeling variability in Simulink. In: Proceedings of the international workshop on variability modeling in software-intensive systems (VaMoS), VaMoS '14. ACM, New York, pp 7:1–7:8
46. Weston N, Chitchyan R, Rashid A (2009) A framework for constructing semantically composable feature models from natural language requirements. In: Proceedings of the international software product line conference (SPLC), SPLC '09. ACM, New York, pp 211–220
47. Wille D (2014) Managing lots of models: the FaMine approach. In: Proceedings of the international symposium on the foundations of software engineering (FSE), FSE '14. ACM, New York, pp 817–819
48. Wille D, Holthusen S, Schulze S, Schaefer I (2013) Interface variability in family model mining. In: Proceedings of the international workshop on model-driven approaches in software product line engineering (MAPLE), SPLC '13. ACM, New York, pp 44–51
49. Wille D, Schulze S, Seidl C, Schaefer I (2016) Custom-tailored variability mining for block-based languages. In: Proceedings of the international conference on software analysis, evolution, and reengineering (SANER), SANER '16, vol 1. IEEE, Piscataway, pp 271–282
50. Zhang X, Haugen Ø, Møller-Pedersen B (2011) Model comparison to synthesize a model-driven software product line. In: Proceedings of the international software product line conference (SPLC), SPLC '11. IEEE, Piscataway, pp 90–99
51. Zhang X, Haugen Ø, Møller-Pedersen B (2012) Augmenting product lines. In: Proceedings of the Asia-Pacific software engineering conference (APSEC), vol 1. IEEE, Piscataway, pp 766–771