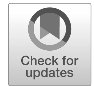

Jörg Christian Kirchhof, RWTH Aachen University
Michael Nieke, TU Braunschweig
Ina Schaefer, TU Braunschweig
David Schmalzing, RWTH Aachen University
Michael Schulze, pure-systems GmbH

18

[KNS+21] J. C. Kirchhof, M. Nieke, I. Schaefer, D. Schmalzing, M. Schulze:
Variant and Product Line Co-Evolution.
In: Model-Based Engineering of Collaborative Embedded Systems, pp. 333-351, Springer, Jan. 2021.
www.se-rwth.de/publications/

Variant and Product Line Co-Evolution

Individual collaborative embedded systems (CESs) in a collaborative system group (CSG) are typically provided by different manufacturers. Variability in such systems is pivotal for deploying a CES in different CSGs and environments. Changing requirements may entail the evolution of a CES. Such changed requirements can be manifold: individual variants of a CES are updated to fix bugs, or the manufacturer changes the entire CES product line to provide new capabilities. Both types of evolution, the variant evolution and the product line evolution, may be performed in parallel. However, neither type of evolution should lead to diverging states of CES variants and the CES product line, otherwise both would be incompatible, it would not be possible to update the CES variants, and it would not be possible to reuse bug fixes of an individual variant for the entire product line. To avoid this divergence, we present an approach for co-evolving variants and product lines, thus ensuring their consistency.

© The Author(s) 2021

W. Böhm et al. (eds.), *Model-Based Engineering of Collaborative Embedded Systems*,
https://doi.org/10.1007/978-3-030-62136-0_18

18.1 Introduction

Product line engineering

Configurability and variability play a pivotal role for collaborative embedded systems (CESs). Individual configurations enable customization and flexibility while, optimally, allowing a high degree of reuse between different *variants*. Product line engineering is an approach that enables mass customization for families of similar (software) systems [Schaefer et al. 2012]. During *domain engineering (DE)*, commonalities and variabilities of variants of a product line—that is, its configured product instances—are typically captured in terms of *features* [Pohl et al. 2005]. A feature represents increments to the functionality of products. *Variability models,* such as feature models [Kang et al. 1990], organize features and the relationships between them. Features are mapped to realization artifacts, such as code, models, or documentation. During *application engineering (AE)*, a variant is derived by defining a configuration that consists of selected features [Pohl et al. 2005]. Using this configuration and the feature-artifact mapping, the resulting artifacts can be composed to form a variant.

Variability for collaborative embedded systems

For collaborative embedded systems (CES), supporting and managing variability is crucial. Typically, a CES is developed once and deployed for different customers and in different environments. Thus, a CES must accommodate customer-specific requirements and be applicable in different environments. Developing these different CES variants individually does not scale economically. Moreover, separate variant development is bad practice as the different variants inevitably diverge from each other, which results in incompatibilities, bugs/errors, and significantly higher maintenance effort [Pohl et al. 2005].

Modifying derived variants

The optimal situation is that all variants are created, maintained, and updated during DE using the product line artifacts and the variability model. In practice, however, customers often require adaptations or updates for their variant, with the adaptations or updates being implemented by changing only this particular variant during AE. For instance, a CES is deployed for one specific customer and this customer requires changes at short notice or implements their own changes. This has several advantages: first, the complexity of implementing such changes is comparably low as the impact on other variants does not have to be considered; second, the time required to deploy new changes and thus the costs are low as well.

This procedure is particularly interesting for variability of CESs. Typically, a CES is used in multiple different CSGs by different companies. Thus, changes to a CES product line require a lot of effort as the impact on all possible variants and the CSGs that use the CES must be considered. Consequently, required changes are implemented directly in a CES variant that is used in a particular CSG.

Diverging changes of product lines and their variants

However, this procedure comes at the cost of lost compatibility between the product line and the changed variant. If product line artifacts are updated, it is unclear whether these changes affect the modified variants and, even worse, it is unclear how to merge the changes at DE level with changes at AE level. As a result, the product line and the modified variants diverge. Consequently, respective variants are not updated if the product line is updated, and other variants cannot benefit from changes that have been made at variant level.

Propagating product line changes to modified variants

To overcome these limitations, we provide an approach that enables engineers to modify variants at AE level while keeping these changes and changes at DE level synchronized. The first part of the approach propagates updates from DE level to modified variants. To this end, an internal repository is automatically maintained. The variants originally derived from the DE level are stored in this repository. If the product line is changed, a three-way-merge mechanism compares the original variant, the updated variant derived from the updated product line, and the modified original variant. As a result, updates from the product line level are merged into the modified variant. Thus, the variant users benefit from product line updates but are still able to modify their variant individually.

Lifting variant changes to product line level

The second part of the approach propagates changes from AE level to DE level. First, changes at variant level are identified. In the next step, the features that are affected by these changes are identified. This is particularly important to allow these changes to be propagated to product line level. However, this task is challenging as, typically, the information about which part of a variant stems from which feature is not preserved when a variant is derived. Finally, the variant changes are transferred semi-automatically to the respective product line at DE level. To this end, regression deltas between original artifacts and modified artifacts are computed and mapped to the respective feature at DE level. As a result, product line artifacts are updated with the most recent changes at the AE level without the need for additional costs to redevelop the variant changes for the entire product line.

18.2 Product Line Engineering

Feature models to represent variability

In product line engineering, features are typically captured in variability models. The most prominent variability model type is a feature model [Batory 2005], [Kang et al. 1990]. Feature models capture the abstract functionality of a product line as features and organize them in a structured tree. Thus, the feature tree has exactly one root feature and can have multiple child features. Each feature, except for the root feature, has exactly one parent feature — that is, the feature tree is an acyclic graph. This tree defines basic relationships between features — that is, a feature can only be selected if its parent feature is selected. Additional constraints can be defined by using feature types or cross-tree constraints in propositional logic with features as variables. In *feature-oriented programming (FOP)*, each feature is implemented separately [Prehofer 1997]. Thus, artifacts, such as code, models, or documentation, that realize a specific feature are developed. In addition, artifacts that are necessary to enable the collaboration of multiple features must be implemented as well.

Variability at implementation level

To realize the variability that artifacts express, there are different mechanisms and notations that establish a feature-artifact mapping. With *annotative* or *negative* approaches, parts of artifacts are marked with feature expressions that define the feature combinations in which they should be used [Schaefer et al. 2012]. If a feature is not selected, its annotated artifact parts are removed. A prominent example of the annotative method is C/C++ preprocessor annotations. With *compositional* or *positive* variability, distinct artifacts for each feature (combination) are implemented that are composed later [Schaefer et al. 2012]. For instance, plug-in systems can be used with a distinct plug-in for each feature. Finally, *transformational* approaches, such as *delta-oriented programming (DOP)* [Clarke et al. 2010], are a combination of the positive and negative approaches. They enable specification of deltas that define changes to artifacts that add, delete, or modify parts of the respective artifacts.

Deriving variants during application engineering

During AE, variants of a product line are derived [Pohl et al. 2005]. To this end, configurations are defined that consist of selected features of the feature model. To derive a concrete variant from such a configuration, a generator uses this configuration, the feature-artifact mapping, and a concrete variability realization mechanism. This variability realization mechanism is specific to the notation used to implement feature artifacts, such as preprocessors, plug-ins, or DOP, and transforms the product line artifacts to match the selected

configuration. For preprocessors, this means removing all annotated parts that do not match the current feature configuration. For additive approaches, such as plug-ins, this means composing all artifacts of the selected features to form a variant. For transformational approaches, such as DOP, the deltas that are mapped to the selected features are collected and their change operations are applied.

Similar to other systems, product lines evolve to meet new requirements or to fix bugs [Schulze et al. 2016]. To this end, feature artifacts and their mapping are modified at DE level and variants can be updated by triggering a new generation at AE level. In theory, this is the optimal way to perform product line evolution. However, in industrial practice, this is often infeasible or simply not done. Consequently, variants are modified at AE level to match specific requirements, to fix bugs, or to be updated. This results in a divergence of product line and variants which we address with the approach presented.

18.3 Propagating Updates from Domain Engineering Level to Application Engineering Level

This section is largely based on [Schulze et al. 2016].

18.3.1 The Challenge of Propagating Updates

Deriving variants and performing customer-specific modifications

To illustrate the process and the resulting problems of propagating updates from DE to AE, we present an abstract overview of variant derivation in conjunction with the evolutionary process described in Figure 18-1. The *Product Line Assets* boxes depicted act as placeholders for different artifacts and each *Variant A* box represents all artifacts belonging to variant A. The creation of a specific customer variant *A* starts with the derivation step at T0, which is symbolized in the figure by Step ①. This step basically consists of multiple actions

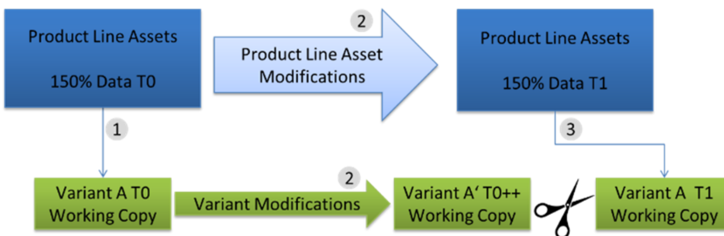

Fig. 18-1: *Challenges of DE and AE co-evolution*

(e.g., selecting features, transforming corresponding artifacts, generating the variant) to be performed for each artifact type, such as requirements, source code, models etc. The result is a working copy for the derived variant that constitutes the base for further development as the product line is not usually able to deliver the entire functionality customers want. Hence, changes to particular artifacts, such as add, remove, and modify, take place on the derived variant at AE level, leading to a customer-adapted and, usually, functionality enriched variant (represented as *Variant A'* in Figure 18-1).

Product line level changes and incompatibilities with variant modifications

Beside modifications on variants' working copies, changes also take place on the entire product line (i.e., DE level) — for example, through maintenance activities such as bug fixing or functionality extension in order to satisfy emerging market needs. The changes at both levels are made simultaneously and in an unsynchronized manner (marked with ② in the figure). In general, this is not a problem and often even desired in industry as it allows variants of different customers to develop at their own speed. However, a problem arises if a derived variant requires further functionality or bug fixes from the product line. This means that the same derivation process of Step ① is performed again at T1 (Step ③), which results in a newly generated working copy for that variant, and as a side effect, all variant modifications (②) on *Variant A* are lost, since the artifacts are replaced by the DE level versions.

The loss of essential changes performed at AE level (visualized by scissors in Figure 18-1) is a major concern for real-world product lines due to the resulting increased time and cost of recreating the changes.

18.3.2 Artifact Evolution and Co-Changes

Basic artifact modifications

Three basic operations can be part of an evolutionary task, regardless of the artifacts affected:

- **Add:** An artifact (e.g., a requirement, code, model, etc.) is added — for example, to extend functionality.
- **Remove:** An artifact is removed — for example, because it became irrelevant.

- **Modify:** An artifact is adapted according to changing circumstances — for example, due to legal issues.[1]

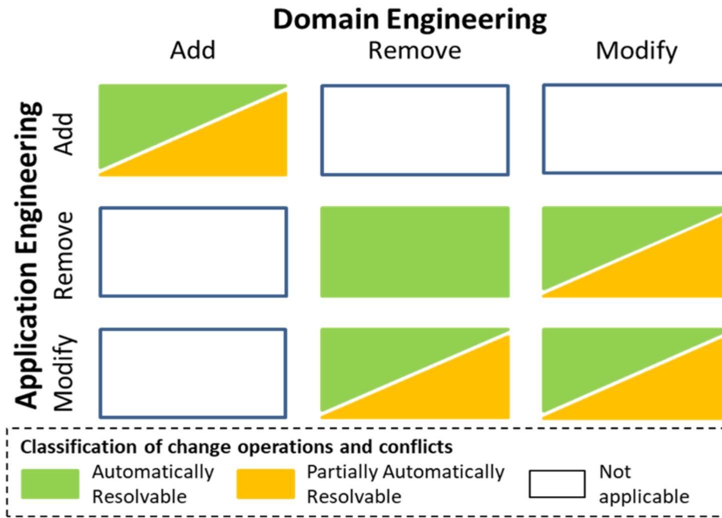

Fig. 18-2: *Co-change operations between DE and AE and their effects*

These types of changes happen at both DE and AE level respectively, and it is only if a change was made on an artifact that exists at both levels that we call it a co-change. Such co-changes can lead to a conflict if an artifact was modified at both levels at the same location but in different ways. In order to preserve the co-changes made at the AE level during update propagation, we have to a) detect, b) classify, and if possible, c) (automatically) resolve each conflicting co-change. The matrix in Figure 18-2 visualizes all possible cases and helps to classify the possible co-changes. As depicted, there are also some cases that can never occur (e.g., an addition of a new artifact at DE level being removed at AE level), other cases that can be fully resolved (e.g., removal of the same artifact at both levels), and cases that can be (partially) automatically resolved (e.g., a modification of a DE level artifact that was removed at AE level). However, before we can classify or even resolve changes, the initial detection of a co-change is key for the subsequent steps.

---

[1] While this operation can be considered as a combination of the two basic operations add and remove, its semantics is important for determining conflicts. Hence, we treat this operation separately.

Detecting and classifying co-changes

Since an evolution is performed simultaneously at both levels, detecting where a change happened and what type of change it was is essential to enable informed decisions in the subsequent steps. Considering the variants' derivation process in Figure 18-1, a comparison of the artifacts of *Variant A'* with *Variant A* at T1 might be a solution, since a change can easily be detected if an artifact differs between both versions. However, this simple approach is not sufficient to detect the level at which the change happened. More problematically, the most difficult case cannot be uncovered in this way — that is, a case where the same artifact was changed in a different way in both versions. This means that with this two-way comparison, in general, no information about the origin (*Variant A'*, *Variant A* at T1, or both versions) or the kind of change can be retrieved.

The problem of the two-way comparison is that it lacks a common base to compare both variants with. In the derivation process in Figure 18-1, the original working copy *Variant A* at T0 constitutes this common base from which both variants originate. Given this common base, we can use a three-way comparison to obtain the changes between DE and AE. This enables us to compare the evolved variants of DE and AE level not only with each other, but also with their origin — that is, the common base at time T0. As a result, we can determine precisely which change operations were performed on the respective variant. We can therefore classify the changes according to our matrix and thus identify possible conflicts.

Resolving changes

With a full classification for each conflicting co-change, the resolution can be reached partially or full automatically, depending first on the nature of the co-change and second on the resolution strategy — for example, if one level takes precedence during conflict resolution. For most of the cases, this allows a fully automatic resolution. For those cases where conflict resolution needs user assistance, there are often tools that allow for adequate visualization and even merging of the conflict. If such tool support is not available, the user must resolve the conflict by hand, which is in any case the last resort.

18.3.3 Changes to the Variant Derivation Process

Necessity of a common base for three-way comparison

The detection of any possible co-change requires the application of a three-way comparison of the artifacts of three different versions (*Variant A* at T0 and T1, as well as *Variant A'*) of product line variants. However, in the scenario in Figure 18-1, not all the three required

versions are available explicitly. Basically, only *Variant A'* is available and *Variant A* at T1 can be generated from the product line artifacts in their current state. Retrieving the common base version of those two versions is more sophisticated. Generally, two approaches are conceivable to solve this problem as follows.

Regenerating a common base from the product line

In the first approach, the base version is regenerated from the product line, which requires a snapshot of the product line, including generators employed at the point in time when the previous base version was generated (i.e., time T0 in Figure 18-1). Provided that the product line is published in fixed release versions, these snapshots can easily be retrieved even if application engineers have no access to interim versions. However, if there are no such release versions, a snapshot of the entire product line must be created every time a variant generation process is triggered on a changed product line.

Saving generated variants as a common base in a repository

In the second approach, each variant generated is saved in a, possibly local, repository to keep it for later use. This approach is shown in Figure 18-3. Between the DE level and the working copy of a specific variant at AE level, a new level for the repository is introduced that is transparent for application engineers. When application engineers derive a specific variant A for the first time at T0, it is stored automatically in the internal repository for that variant (Step ①). The working copy is initially just cloned from that version (Step ②). Over time, the product line and *Variant A* are changed independently of each other (Step ③). Then, at T1, application engineers want an update of their working copy to synchronize with the current product line version. During that update propagation, a new version of *Variant A* is derived and stored in the internal

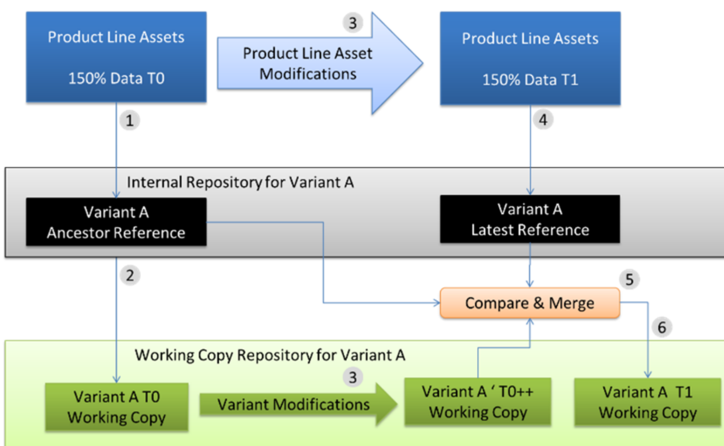

Fig. 18-3: *Solution for co-evolution and propagating updates from DE to AE*

repository (Step ④), but this version is not shown to application engineers directly. Instead, a three-way comparison (Step ⑤) is performed between the two versions in the repository (the ancestor reference as common base and the latest reference) and the working copy version *Variant A'*. As discussed above, most merges are done without user interaction and it is only for conflicts that cannot be resolved that application engineers must decide which changes should be applied. The result is an updated working copy with merged changes of the DE and AE level (Step ⑥). This update process can be repeated each time the product line is changed.

18.3.4 Applicability and Limitations

Basically, our proposed classification scheme is general enough to be applicable with different scenarios and different artifacts in product line development. This is because our definition of both change operations and change conflicts is artifact-independent and we address the integration in the common product line development process. However, due to its general nature, our method requires some manual effort to be adapted for concrete product lines. Most importantly, the concrete artifacts that are subject to change operations must be defined and an instantiation of their granularity levels must be provided. The latter is of specific importance, because the granularity plays a pivotal role in deciding whether a conflict exists or not. Moreover, granularity levels are different for specific artifacts. For instance, for source code, it may be sufficient to distinguish between statement, block, and file level. In contrast, if we consider artifacts in a hierarchical structure, such as requirement specifications, different levels of granularity such as line, section, or subsection may be required to detect conflicts with a suitable accuracy. Finally, developers must specify how the conflict detection and resolution is integrated in the (most likely already existing) development process, for instance, which tools should be used for conflict detection. However, the aforementioned instantiation has to be done only once (when setting up or integrating with an existing product line engineering process) and can subsequently be used for the entire evolution process.

Finally, it is worth mentioning that, with our proposed classification, we focus mainly on syntactical changes. As a result, our classification does not ensure semantic correctness. However, we argue that syntactical correctness is the stepping-stone for consistent

co-evolution in product lines and thus for ensuring integrity of both DE and AE level.

18.3.5 Implementation

In our prototypical implementation, we have integrated the process described into pure::variants[2], the leading industrial variant management tool, which supports the development of product lines. This tool can manage different types of realization artifacts, either by means of generic modeling in the tool or by means of integration into external tools using specific connectors. The derivation process for variants is handled by an extensible set of transformations that are specific to the artifact type or external tool. These transformations are the connection point for our implementation. Since the chosen approach is generic, the prototypical implementation supports all types of artifacts as long as a three-way comparison is available for the specific artifact type. For example, for source code, the internal local repository is realized by simply creating folders for the ancestor as well as latest references, as can be seen in Figure 18-4 from the box in the upper left corner.

The three-way comparison and the merge are then executed using the three directories directly, while specifying the ancestor directory as the common base of the two others once. Thus, when an application engineer wants to update their working copy, they start a new derivation of the current variant, which leads to the generation of a new latest version, followed by triggering the compare and merge operation. If there are no conflicts that have to be resolved manually, the application engineer will get the merged result. If there are conflicts, the application engineer must resolve them by deciding which version—working copy or latest—they prefer to be in the merged result. At the end, the application engineer gets a merged version semi-automatically.

The prototypical implementation was presented to different customers and received a positive response, with many of those customers facing the challenges mentioned with regard to variant and

[2] www.pure-systems.com

product line co-evolution. Thus, our method addresses a highly relevant topic in the industrial domain.

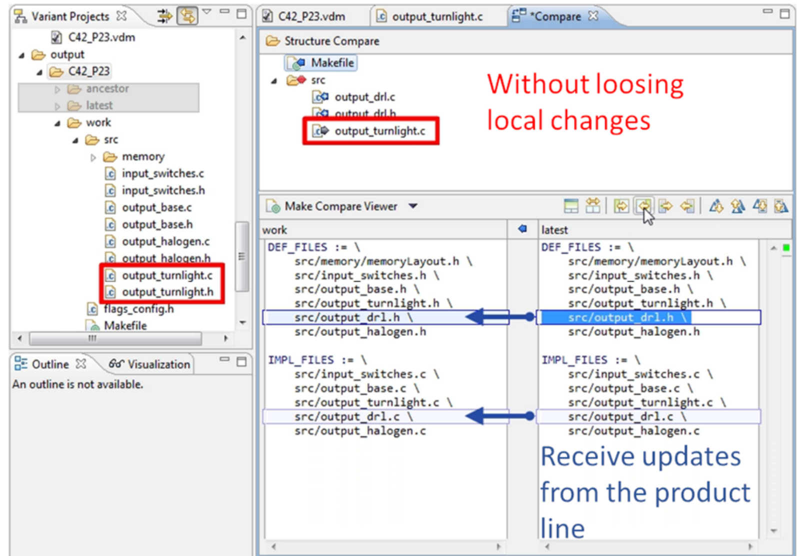

Fig. 18-4: *Updating a variant in pure::variants preserving local changes*

18.4 Propagating Changes from Application Engineering Level to Domain Engineering Level

18.4.1 The Challenge of Lifting Changes

Challenges in propagating changes from the AE level

Propagating updates from the AE level to the DE level produces a few challenges. Introducing changes from the AE level to the DE level may result in conflicts, as development may go ahead at the DE level as well. Detecting changes and applying them to DE level artifacts is made more complicated here, as, in feature-oriented programming, there is often a mapping between features and implementation artifacts. Depending on the variability specification mechanism used, reconstructing the feature mapping from AE level artifacts is often not straightforward. In constructive mechanisms - for example, when constructing a 150% model - references to features may still exist in AE level artifacts. Yet, with transformational approaches, feature references are usually removed during the generation of AE level artifacts. However, reconstructing this mapping on the AE level is crucial for assigning changes to the correct features.

Prerequisites for propagating changes

Our goal is to lower the barrier for adopting changes to variants in product line engineering by supporting the propagation of changes from a variant's working copy to the product line. To adequately propagate changes to the DE level, we have to a) detect changes, b) make the feature information available at AE level, c) assign changes to features or the codebase, and d) resolve each conflicting co-change. We propose a process that detects changes in the working copy of a variant then maps them to the appropriate features and transfers them semi-automatically to the product line.

18.4.2 A Process for Lifting Changes

Approaches to propagating changes

Similar to updating the working copy of variants with changes from the product line, detecting co-changes requires a three-way comparison of the artifacts in questions when lifting changes to the product line. Here, two possible approaches are conceivable. In the first approach, changes in the working copy of the variant (*Variant A'*, see Figure 18-1) are detected by comparing it to its base version (*Variant A at T0*). The changes detected are then translated and applied to the base version of the product line (*Product Line Assets at T0*), resulting in a new product line version. These two versions are then compared with the updated product line (*Product Line Assets at*

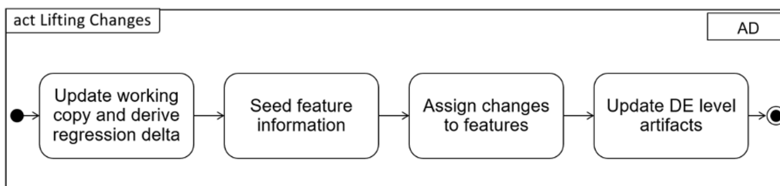

***Fig. 18-5:** Activities for propagating changes from the AE level to DE level*

T1) in a three-way comparison to detect and resolve conflicting co-changes. In the second approach, co-changes are instead detected and resolved on the AE level artifacts and only then translated and applied to the product line. This approach follows the process of updating the working copy of a variant (see Section 18.3.3) with changes from the product line, as co-changes are identified and resolved through a three-way differencing and merge on the three different variant versions.

Process description for propagating changes from the AE level

We follow the second approach, as this approach builds upon the previously proposed process for updating a variant. The proposed process for this approach is presented in Figure 18-5. It consists of four steps: first, we update the working copy of a variant with changes

in the product line through a three-way merge on the artifacts of the three different variant versions. In addition to resolving conflicting co-changes, we also calculate regression deltas between the new variant versions (*Variant A* derived from the *Product Line Assets* at T1 and the working copy of *Variant A resulting from the merge*). These regression deltas represent the changes detected that will be applied to the DE level artifacts. However, changes must first be assigned to their corresponding feature (*Seed feature information*), and for this we require access to domain knowledge at AE level. To this end, in the second step, we annotate AE level assets with feature information. These annotations are the input in the third step to assign each change to a corresponding feature. Finally, in the fourth step, we translate and apply changes to DE level artifacts. In the following, we focus on the second and third steps, which we present in more detail.

18.4.3 Deducing Feature Information

Conflicts when updating DE level artifacts

Conflicting co-changes must also be resolved if changes from the AE level are to be propagated to the DE level. Changes must also be assigned to a feature to be made available to other variants of the product line. However, developers at the AE level implement changes concerning the variant's configuration, and information about individual features is usually not available. Changes at AE level can change the implementation of existing features or the codebase (e.g., bug fixing) or add new features (implementation of new functionalities). Before we can assign changes to features, the changes must first be detected, and domain knowledge must be made available at AE level.

Underlying Model

General model description

Artifacts, their content, and their relationships can be represented abstractly as a graph $G = (V, E)$. Here, the set of vertices V represents artifacts or elements of artifacts in the desired granularity, and the set of typed edges $E = V \times V \times T$ represents their relationships, where T is the set of kinds of relationships identified. One possible realization of this data structure is object diagrams, which adequate transformations can extract directly from a development project and which we can employ to identify the impact of individual changes [Butting et al. 2018]. We use this data structure as an internal representation of model artifacts to abstract from concrete syntax changes.

Description of annotations and recommendation values

Besides the internal representation of model artifacts, we annotate elements (vertices) with features, which we store as a mapping $a: V \cup E \rightarrow F$, where F is the set of features. In our representation, the common codebase is mapped to the root feature, which is thereby represented as well. After the second phase (*Seed feature information*), each model element and each relationship of the base variant is annotated with exactly one feature. When assigning changes to features, we calculate recommendation values for each change and feature pair; that is, we calculate a mapping $r : C \times F \rightarrow [0, 1]$ that assigns to each pair *(c, f)* the probability that change c belongs to feature *f*. Here, $c \in C, and\ f \in F$, where C is the set of changes. Furthermore, we then calculate $r_{rem}(e, f), r_{add}(e, f),\ and\ r_{mod}(e, f)$, which state whether the removal, the addition, or the modification of a model element e may belong to a feature *f*.

Seeding Feature Information

Prerequisites for seeding feature information

Since changes in AE level artifacts are applied to model elements of implementation artifacts, information about which model elements belong to which feature is essential to allow informed decisions when assigning changes to features. While feature-oriented programming usually includes a feature mapping that assigns implementation artifacts or even model elements to features, this mapping is usually not available at AE level. The availability of the feature mapping at AE level depends on the variability mechanism and the variant generation process. If feature information is part of implementation artifacts at AE level, then even assigning changes to features may be trivial, as application engineers can implement changes in the scope of the corresponding feature directly.

Required domain knowledge

In most cases, feature information is not part of the resulting implementation artifacts. One example of this is transformational approaches, which transform some core model based on the selected features without traces of these transformations at AE level. As feature information is not available at AE level, we can instead reconstruct this information through the variant generation process. This can either be done directly during the initial variant generation or be recomputed from the product line. With the former, the feature information would have to be computed and derived for all variants, even if changes in a variant are never propagated to the product line. The latter would require the version of the product line, including generators employed at the point in time when the variant was generated. In either case, the goal is to annotate each model element

of the desired granularity of the unmodified variant at AE level with the corresponding feature.

Introducing new artifacts at AE level

In addition to the feature annotation derived from the product line, we require application engineers to annotate which major changes at AE level (e.g., the introduction of new artifacts) represent new features. Since these features are not (yet) known in the product line, it is otherwise not possible to distinguish between new features and changes to an existing feature. In contrast to variability mining, it is not possible to compare several variants to identify new features, since changes usually affect a single working copy of a variant. Instead, by partially annotating changes with a new feature, the full variant may be explored through further analysis. The resulting feature annotation of elements is used in the following to assign changes to specific features.

Assigning Changes to Features

Prerequisites for assigning changes to features

With a complete annotation of the original model elements with features, and incomplete information about new features, we can annotate the remaining changes with features through further analysis. Generally, this can only be achieved partially automatically through a recommendation engine. In some cases, annotating changes with features may be computed fully automatically depending on the quality of analyses employed, the unambiguity of the resulting annotations, and on conflicts in other variants when propagating changes to DE level artifacts.

Noteworthy feature relationships for the recommendation

As before, we focus on the three operations add, remove, and modify. Furthermore, we incorporate domain knowledge into our analysis; that is, we consider the *parent-child* relationship and the *requires* relationship of features. Using well-formedness rules together with domain knowledge enables us to limit the set of features that can contain a particular change. The concrete implementation, however, depends on the modeling language and variability specification mechanism used. The notes here provide the basis for implementing appropriate analyses for the respective circumstances.

Removal of model elements

A model element can only be removed in the feature that introduced it (the annotated feature) or in any of its dependent features. We call a feature f_1 dependent on a feature f_2 if f_1 is in a child-hierarchy of f_2 or if f_1 requires f_2. Dependent features can be removed only if the variability specification mechanisms support removing elements that have been introduced in another feature (e.g., transformational variability specification mechanisms). If model element e is removed at AE level, then $a(e) = f$ (model element e is

annotated with feature f) implies $r_{rem}(rem\ e, f) = 1$ (whether the removal of e may occur in feature *f*) and in the latter case, this also implies $r_{rem}(rem\ e, f_1) = 1$, where f_1 is dependent on feature *f*.

Modification of model elements

Similar to the removal of elements, a model element can only be modified in the feature that introduced it or in any of its dependent features. Therefore, if model element e is modified at AE level, then $a(e) = f$ (model element e is annotated with feature *f*) implies $r_{mod}(mod\ e, f) = 1$ and in the latter case, this also implies $r_{mod}(rem\ e, f_1) = 1$, where f_1 is dependent on feature *f*.

Addition of model elements

Any domain-specific or general-purpose language supports relationships between model elements, where relationships between two elements can be expressed by the relation $R \subseteq E \times E$, where $(e_1, e_2) \in R$ states that model element e_1 relates to model element e_2 in some way. Common relationships are containment relationships and references to other elements. Examples of the former are classes in Java that contain fields and method declarations. An example of the latter are transitions between two states in an automaton that reference their source and target state. Model elements must be introduced in the same feature that introduces a relationship on that feature, or in any of that feature's parent features - that is, if there is a relationship (e_1, e_2) between model element e_1 and e_2, and $a((e_1, e_2)) = f$ (the relation is annotated with feature *f*), then $r_{add}(add\ e_1, f) = 1$, $r_{add}(add\ e_2, f) = 1$, $r_{add}(add\ e_1, f_1) = 1$, and $r_{add}(add\ e_2, f_1) = 1$ for all features f_1 in the parent-hierarchy of feature *f*.

Calculating overall recommendation value

We compute the overall recommendation r for each change with $r(e, f) = r_{rem}(e, f) + r_{add}(e, f) + r_{mod}(e, f)$ by merging the recommendations of r_{rem}, r_{add}, and r_{mod}. The highest recommended feature f for each model element e is returned by the recommendation engine.

18.4.4 Applicability and Limitations

The proposed update process and the proposed recommendation mechanism are general enough to be applicable for different variability specification mechanisms and can be realized for different modeling languages. This is because we generally regard models as constructs consisting of model elements and relationships between these elements. Implementation of the recommendation mechanism and of the update process for different modeling languages, however, requires additional implementation effort, as for each modeling language, we have to identify possible relationships between artifacts

and extract these to transfer them into the recommendation engine. Furthermore, the proposed recommendation mechanism considers all modeling elements and changes to be equally important. If this is not desired, then weights must be defined for these elements. Moreover, domain engineers still have to manually merge changes into the product line artifacts, as recommendations provide only a general idea as to which features particular changes can be applied to. Here, the domain engineers' decisions can be used to limit the decision space further and update recommendations. Updating product line artifacts with changes from the AE level may and will cause conflicts in existing variants. Developers must integrate the process for propagating changes into the product line's development process and define how conflicts across variants will be resolved. Finally, the accuracy of the recommendations depends on the granularity of the overlying model, the maturity of the analysis, and the differencing algorithms employed. Here, we consider only syntactic changes, but algorithms that analyze semantic changes could also be used to enhance recommendations.

18.5 Conclusion

Variability and configurability play a pivotal role for CESs and CSGs. Product line engineering is an approach for structured reuse and management of CES and CSG variability. To meet new requirements, product lines evolve, and their variants can be updated accordingly. However, in industrial practice, individual variants are modified, which yields the threat of incompatibility. In this article, we proposed an approach to keep product lines and their variants synchronized. With this approach, the benefits of performing evolution at both product line level and variant level are combined. With a high degree of automation, engineers can perform evolution at variant level without the drawback of a high manual effort to synchronize the product line with the modified variant. Consequently, our contributions make product line engineering more applicable for industrial practice.

18.6 Literature

[Batory 2005] D. Batory: Feature Models, Grammars, and Propositional Formulas. In: International Conference on Software Product Lines, Springer, Berlin, Heidelberg, September 2005, pp. 7-20.

[Butting et al. 2018] A. Butting, S. Hillemacher, B. Rumpe, D. Schmalzing, A. Wortmann: Shepherding Model Evolution in Model-Driven Development. In: *Modellierung (Workshops), 2018,* pp. 67-77.

[Clarke et al. 2010] D. Clarke, M. Helvensteijn, I. Schaefer: Abstract Delta Modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, ACM, 2010, pp. 13-22.

[Kang et al. 1990] K.C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study (No. CMU/SEI-90-TR-21). Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst., 1990.

[Pohl et al. 2005] K. Pohl, G. Böckle, F. van der Linden: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer 2005, ISBN 978-3-540-24372-4.

[Prehofer 1997] C. Prehofer: Feature-Oriented Programming: A Fresh Look at Objects. In: ECOOP'97 - Object-Oriented Programming, 11th European Conference, Springer, 1997, pp. 419-443.

[Schaefer et al. 2012] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela: Software Diversity: State of the Art and Perspectives. In: International Journal on Software Tools for Technology Transfer, Volume 14, Number 5, Springer, 2012, pp. 477-495.

[Schulze et al. 2016] S. Schulze, M. Schulze, U. Ryssel, C. Seidl: Aligning Coevolving Artifacts Between Software Product Lines and Products. In: Proceedings of the Tenth International Workshop on Variability Modelling of Software-Intensive Systems, ACM, 2016, pp. 9-16.

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.