Chapter 9
Evolution of Software Product Lines

Goetz Botterweck and Andreas Pleuss

Summary. A *Software Product Line* (*SPL*) aims to support the development of a family of similar software products from a common set of shared assets. SPLs represent a long-term investment and have a considerable life-span. In order to realize a return-on-investment, companies dealing with SPLs often plan their product portfolios and software engineering activities strategically over many months or years ahead. Compared to single system engineering, SPL evolution exhibits higher complexity due to the variability and the interdependencies between products. This chapter provides an overview on concepts and challenges in SPL evolution and summarizes the state of the art. For this we first describe the general *process* for SPL evolution and general *modeling concepts* to specify SPL evolution. On this base, we provide an overview on the state-of-the-art in each of the main process tasks which are *migration* towards SPLs, *analysis* of (existing) SPL evolution, *planning* of future SPL evolution, and *implementation* of SPL evolution.

We thank Howell Jordan for comments on an early version of the text. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

9.1 Introduction

A *Software Product Line* (*SPL*) aims to support the development of a family of similar software products from a common set of shared assets [193, 688, 910]. By applying SPL practices, organizations are able to achieve significant improvement in time-to-market, engineering and maintenance costs, portfolio size, and quality [193]. SPLs have been commercially applied in many industry domains [784] including embedded systems, web and mobile applications.

SPLs represent a long-term investment and have a considerable life-span. Moreover, SPLs scale to a considerable size and are often embedded in larger structures, i. e. they consist of subsystems and are part of a larger supersystem. Hence, changes on an SPL can have a complex impact not only on the whole product family but also on related systems. When a change is introduced, inconsistencies are unavoidable until the change has been propagated through the system and related systems. Since usually multiple parties are involved and there are multiple changes, this can easily lead to further inconsistencies. All this needs to be taken into account when considering potential changes.

Because of the long term perspective, size, and complexity, organizations dealing with SPLs need to address evolution in a systematic fashion. In this chapter, we give an overview of such systematic approaches to SPL evolution. Our goal is to provide the reader with an introduction and given an overview of the field, which allows to identify more specialized literature that provides more details of a particular aspect.

As a background, Section 9.2 summarizes the main ideas of SPLs and Section 9.3 covers basic concepts for SPL evolution. Then, Section 9.4 gives an overview of approaches to SPL evolution. More concretely, we cover modeling of SPL evolution, processes for SPL evolution, migration towards SPLs, as well as the analysis, planning and implementation of SPL evolution. Section 9.5 concludes the chapter with an overview of remaining research challenges and final thoughts.

9.2 Software Product Lines

An SPL aims to support the development of a whole family of software products through systematic reuse of shared assets [193, 688, 910]. By an *asset* we refer to any artifact that is part of the software production process, such as an architecture, a software component, a domain model, a requirements document, a formal specification, documentation, a plan, a test case, or a process description [193].

As an example for an SPL consider online shop software: while different online shops usually differ from each other – e. g. in the supported payment methods, shipping options or article types – their underlying concepts are very common and can be implemented from reusable assets. Hence, a company offering a spectrum of online shop applications can use SPL techniques to achieve systematic reuse by (1) first identifying and creating the required reusable assets and (2) deriving the individual products (i. e. different online shop implementations) from the assets created in the first step in a systematic way.

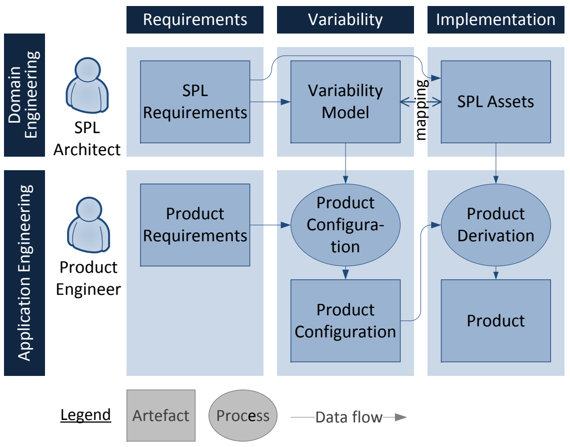


Fig. 9.1: Software Product Line Engineering (SPLE) framework

SPL engineering (*SPLE*) provides concepts on how to develop SPLs. A basic SPLE framework is shown in Figure 9.1: SPLE approaches often distinguish between *domain engineering* and *application engineering*[1]. Domain engineering deals with creating (and maintaining) the whole SPL. First, requirements for the SPL are elicited and the scope of the SPL is defined, i. e. a definition which potential products are to be supported. The variability between potential products is captured in a *variability model*. It defines the available variants, e. g. different payment methods and shipping options in an online shop, and the allowed combinations. To allow the creation of products the variants identified in the variability model need to be implemented by reusable SPL assets. A *mapping* is then specified to define which variant from the variability model is implemented by which assets.

Figure 9.2 shows example domain engineering models from an fictitious SPL for online shop software (*e-shop*). The left-hand side shows the variability model. There are several variability modeling approaches that could all be applied here, e. g. *feature models* [757], *decision models* [755] or *the orthogonal variability model* [688]. In this chapter we use feature models as a basis for the discussion. Other approaches are conceptually similar [208] and could be applied in a similar fashion.

A *feature* is a "distinguishable characteristic of a concept (e. g. component, system, etc.) that is relevant to some stakeholder of the concept" [207]. The example model shows features of an e-shop such as support for a Catalog, a Search function, and different ArticleTypes.

[1] Some approaches use different terms, like *core asset development* and *product development*, but provide essentially a similar distinction.

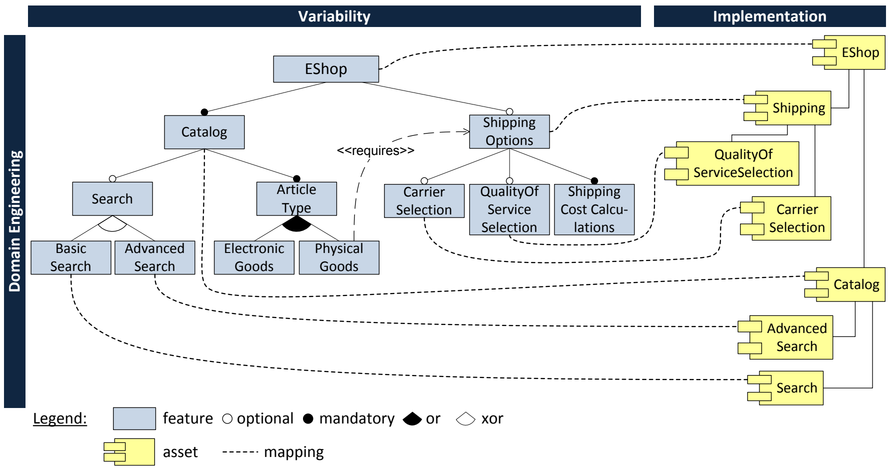


Fig. 9.2: Example domain engineering models: A feature model (left) and associated reusable implementation assets (right).

A feature model specifies all features supported by the SPL and the dependencies between them. Features are structured in a tree hierarchy. Additional constraints express further restrictions on which features can be selected or eliminated when specifying a concrete product. *Mandatory* features must always be selected if their parent is selected while *optional* features are facultative depending on the choices of the user. Features can also be grouped into *or-groups* (if the parent is selected at least one child must be selected) or *xor-groups* (if the parent is selected exactly one child must be selected). Selecting a child feature mandates that its parent is selected as well.

The feature model shown in Figure 9.2 specifies that each e-shop must support a Catalog (mandatory feature) which may include a Search function (optional) and must include an ArticleType feature (mandatory) from which at least one child has to be selected (or-group).

In addition, cross-tree constraints can be defined between arbitrary features in the model like *requires* (selecting a features requires to select another one) or *excludes* (two features mutually exclude each other). In the example, a requires constraint defines that PhysicalGoods always requires ShippingOptions to be selected.

The features in a feature model have to be implemented by reusable assets, represented by software components on the right-hand side of Figure 9.2. Additional mappings specify which features are implemented by which assets. In practice, these mappings are not always one-to-one but more complex. Moreover, features are usually mapped to different types of assets which in combination specify the complete implementation, including the *product line architecture* (*PLA* [130]), code fragments, test cases, and documentation.

During *application engineering* (see Figure 9.1), concrete products are developed based on the assets provided by the SPL. A product is defined by a *product config-*

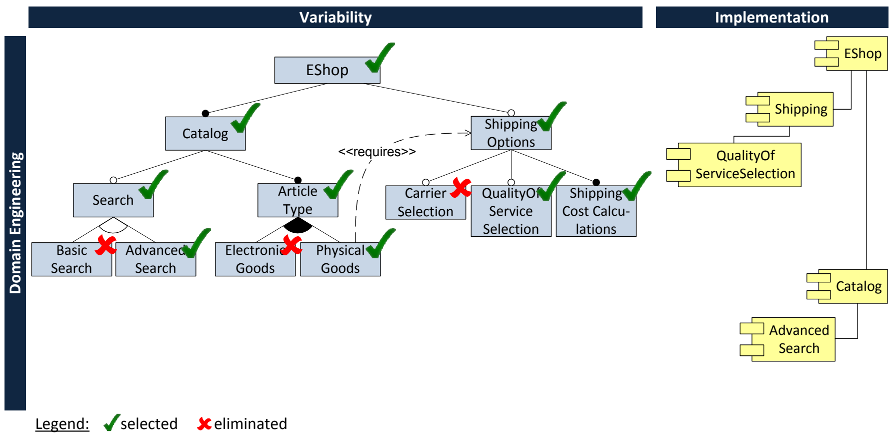


Fig. 9.3: Example product definition during application engineering: A product configuration (left) and the resulting product implementation (right).

uration, which resolves the variability by selecting from the given variants while considering the defined constraints. In the case of a feature model, this is done by selecting or eliminating features. Based on the product configuration and the feature mappings it is then possible to derive the resulting product (*product derivation*).

Figure 9.3 shows an example for one particular product configuration created during application engineering. Here, a concrete product of the example SPL is defined by selected and eliminated features (left-hand side of Figure 9.3). The sample product configuration defines an e-shop that supports AdvancedSearch, PhysicalGoods, and ShippingOptions with QualityOfServiceSelection while BasicSearch, ElectronicGoods and CarrierSelection have been eliminated from the product. The corresponding implementation for this product (right-side of Figure 9.3) is then derived based on the mappings defined during domain engineering. For instance, the component implementing AdvancedSearch is included into the product implementation while the component Search, which implements BasicSearch, is eliminated.

Often, SPL concepts are combined with techniques from model-driven software development [795]. In a model-driven SPL, the product derivation is realized by model transformations that, ideally, generate the complete product together with all documentation, test cases, etc., in a fully automated way [349, 901]. However, a more extensive use of modeling frameworks, as required for automation, can also lead to a higher maintenance effort [238]. For instance, changes in a metamodel might require all existing model instances to be migrated (*co-evolution*, see Chapter 2).

9.3 Characteristics of SPL Evolution

SPL evolution faces several challenges caused by the characteristics of SPLs:

- *Long life-span*. On the one hand, an SPL is a long-term investment that pays off the more, the more products are derived from the SPL. On the other hand, an SPL must evolve to reflect new and changed requirements for its products. Hence, an SPL will often evolve to a greater extent and over a longer period of time than the single products.
- *Large size and complexity*. As an SPL represents a whole family of products, it is of larger size and complexity than its individual products. Usually, multiple teams are involved in its creation and maintenance. Hence, knowledge can be more distributed and evolution of different the parts of an SPL can happen at different speeds.
- *More interdependencies*. Due to the systematic reuse in an SPL, there are more interdependencies between software assets. For instance, changes on the SPL level (e. g. a bug fix in a reusable asset) can affect many individual products created based on the SPL, and new requirements on individual product level can require changes with the whole SPL (e. g. substituting a reusable asset).

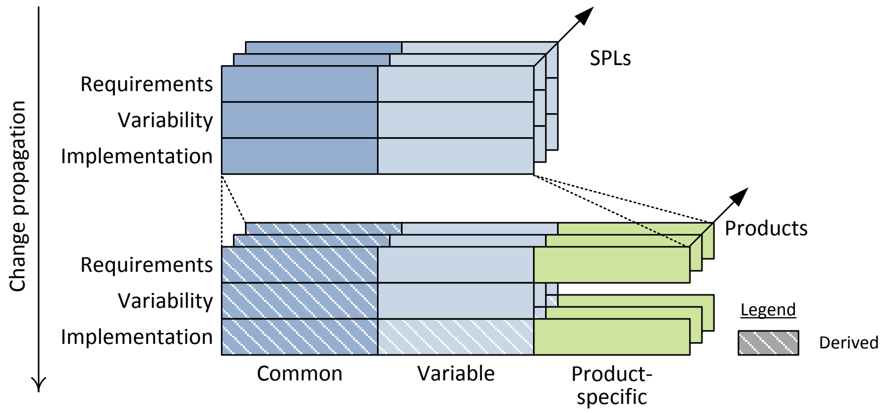


Fig. 9.4: SPL levels and assets subject to evolution.

The additional complexity in an SPL is partly caused by the different abstraction layers that have to be considered together (see Figure 9.4). In an SPL, one has to distinguish between the SPL (upper part in Figure 9.4) and its products (lower part). In addition, there can also be multiple SPLs to manage complex product portfolios [753], for instance, (i) to modularize development of very large systems into multiple SPLs with a shared architecture (*program of product lines* [230]), (ii) to handle very large variability by specifying main variability decisions on a top-level SPL while lower level SPLs specialize this further (*hierarchical product* *lines* [130]), or (iii) to support reuse across multiple domains by multiple SPLs that share some assets (*product populations* [873]). Such very large systems are sometimes developed not only by the organization's internal developers but by a whole developer community, including external developers and third-party contributors, leading to so-called *software ecosystems* (see Chapter 10).

SPLs and products, as any other software, can be defined by assets on different levels of abstraction, from requirements to the final implementation. As indicated in Figure 9.4, different abstraction levels need to be considered both in the SPL and its products: on SPL level there are the requirements and implementation for the whole SPL. In addition, variability between the products is defined. On product level there are the product requirements that influence the product-specific variability resolution (i. e. the product configuration) and the corresponding implementation.

To analyze the impact of evolutionary changes, the assets in an SPL can be further classified into three categories:

- *Common assets* are defined on the SPL level. They are part of all products and are hence directly derived from the SPL (derived assets are represented by shaded areas in Figure 9.4). For instance, in the example from Figure 9.2, the feature Catalog is defined as mandatory child feature of each e-shop. Hence, the feature Catalog, the associated requirements, and the corresponding asset Catalog are by definition part of every product.
- *Variable assets* are defined on the SPL level as well. They are part of some products depending on each product's configuration. Hence, on product level, there must be a variability decision about each variable asset (e. g. selecting or eliminating a feature) which is driven by the requirements for the particular product. The assets within the product's implementation are then derived according to this decision (i. e. including or excluding variable assets into the product). For instance, the Search feature is optional in the e-shop example, so it has to be decided on product level (based on the product's requirements) whether to include it or not.
- *Product-specific assets* are used to add functionality to individual products, e. g. some customer-specific functionality not supported by the reusable assets in the SPL. Hence, product-specific assets and their corresponding requirements reside on product level only and there is no variability configuration for them. An example in the e-shop might be a customized search function optimized for a specific article type. Usage of product-specific assets should ideally be minimized within an SPL approach as it diminishes reuse and increases maintenance effort. However, depending on the market and the business model it is not always possible to reject product-specific requirements.

Considering Figure 9.4, evolution can occur on three different levels (see [753, 809]). On the level of the *set of SPLs*, new SPLs can be added or deprecated ones can be deleted. In addition, SPLs can be merged, for instance, due to an acquisition or if SPLs become similar over time [753]. SPLs can also be split, e. g. when parts of the SPL are likely to evolve in a different direction in the future [809].

On the level of the *set of products*, new products can be added (by product derivation) and old deprecated ones can be deleted. Basically, adding new products should not require any changes to the SPL or other products. However in practice, as pointed out in [407], new feature combinations in a product configuration can sometimes lead to unforeseen effects in the implementation (e. g. *feature interactions* [133]) which then require implementation changes.

On the level of *single assets*, assets can be added, deleted, or modified. Changes have to be propagated towards lower levels of abstraction (e. g. from requirements to implementation). Changes on common assets are performed on the SPL level and affect all derived products. Changes on variable assets that are performed on the SPL level affect all products where the respective variants are selected. On the product level, variable assets are added or removed by changing the product configuration. Changes on product-specific assets affect only individual products.

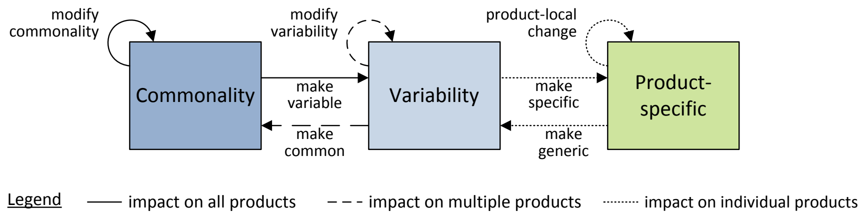


Fig. 9.5: Types of changes on assets (based on [753]).

A specific type of change in SPLs is “moving” assets between the categories common, variable, and product-specific. Figure 9.5 shows the different types of changes and their impact (based on [753]). For instance, common functionality can be made variable if it should be excluded from some products. Usually this requires changing the implementation (to make it variable) which then affects all products. Making a variable asset common influences at least those products that did not contain the asset before. Making a variable asset specific or a product-specific asset generic influences only the specific product.

Of course, changes on SPL level take only effect on an existing product if the product is re-derived, e. g. to release a new version of the product that includes the changes made on SPL level. It depends on the company strategy and the importance of a change (e. g. important bug-fixes) if and when changes on SPL level are propagated to existing products.

To summarize this section, we can say that evolving an SPL can be particularly complex as one has to consider (1) both the SPL and its products and (2) the variability of the assets.

9.4 Approaches to SPL Evolution

In this section we will give an overview of the state-of-the-art in SPL evolution. To give the reader some orientation, Figure 9.6 shows a graphical summary of the areas that we will cover.

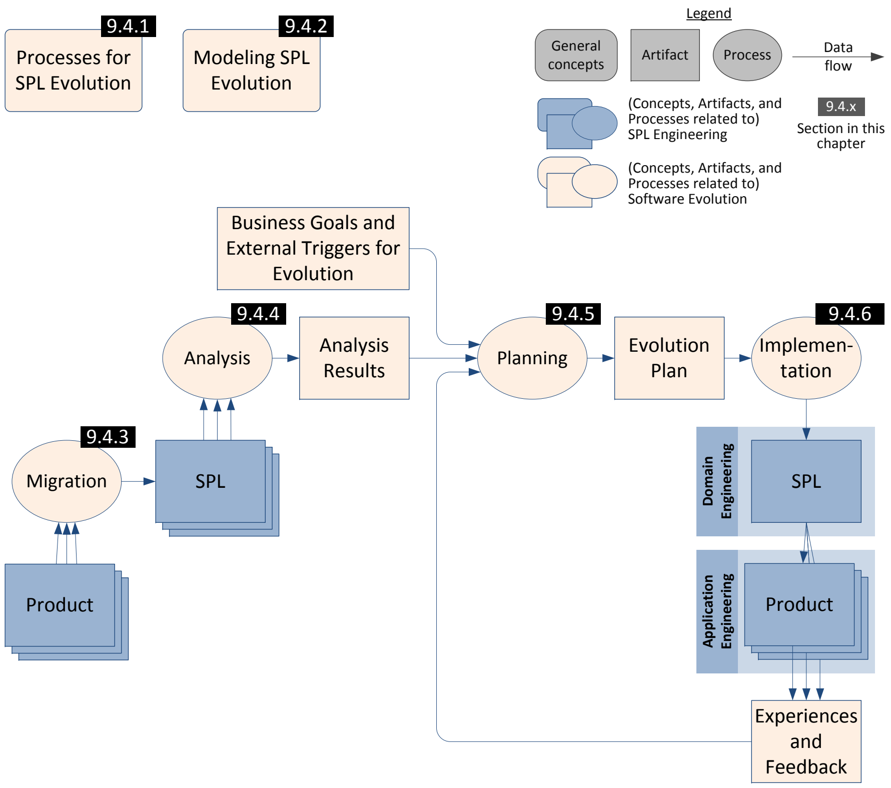


Fig. 9.6: Overview of activities and aspects in the evolution of SPLs.

First, we will address general concepts, i. e. *process models for SPL evolution* (Section 9.4.1) and *modeling techniques for SPL evolution* (Section 9.4.2). We will then roughly follow a process of evolution, as follows: An SPL is often initiated through the *migration* of existing products into an SPL (Section 9.4.3). A second step in initiating SPL evolution is the *analysis* of past evolution (Section 9.4.4), which leads to an overview of which changes happened in previous evolution steps. Subsequently, SPL evolution is performed by iterations of *planning* future evolution (Section 9.4.5), and *implementing* it (Section 9.4.6). In each iteration, the *evolution plan* is updated and the change relative to the earlier version implemented to reach the next evolution step of the SPL. Experiences and feedback are then used as input to the planning of the next iteration.

9.4.1 Process Models for SPL Evolution

In the context of SPL evolution, the literature provides various suggestions for processes and methods. These range from evolution-oriented extensions of general SPLE frameworks [85] to methods that address a particular aspect of SPL evolution, e. g. the mining of legacy assets [655].

9.4.1.1 Process framework for SPL evolution

We will now introduce a process framework for SPL evolution, which is based on the generic framework for SPLE (Figure 9.1) introduced earlier in Section 9.2. We extend and refine that to cover the specific aspects of SPL evolution (see Figure 9.7).

Just like in the basic SPLE framework, we vertically distinguish activities for the creation of the product (*domain engineering*) and the derivation of products (*application engineering*). Horizontally, we distinguish various types of artifacts, for instance, *requirements*, *features* (as a common type of variability specification), and *implementation*. On a higher level, i. e. *method engineering*, we deal with the set-up and configuration of a process and organizational structures (the method) in the other two layers.

To handle evolution, this framework includes activities taking care of adaptation and change. This begins with the initial setup through *method configuration* ❶ according to the particular context. Here we have to take into account specific *method requirements* given by the context, which influence the process structures on domain engineering and application engineering level. For instance, in domains that deal with co-design by various disciplines (e. g. mechanics, electronics, software) we might have to execute and synchronize multiple parallel design activities. Similarly, in domains with regulated software we might have to include special review activities into the process. These variations are not shown in Figure 9.7, but give examples of why an adaptation of the process might be necessary.

Once this setup has been completed, the activities of domain engineering and application engineering are performed. As their main objectives these activities aim to create the SPL (in domain engineering) and derive products from it (in application engineering). However, as side results they also initiate change and evolution: The activities of *product configuration* and *product derivation* yield information on *mismatches and suggested changes*, e. g. when the current SPL is not able to cover all *product-specific requirements*. In some cases the engineers might decide to implement *product-specific assets* to overcome these gaps between the current capabilities of the SPL and product-specific requirements.

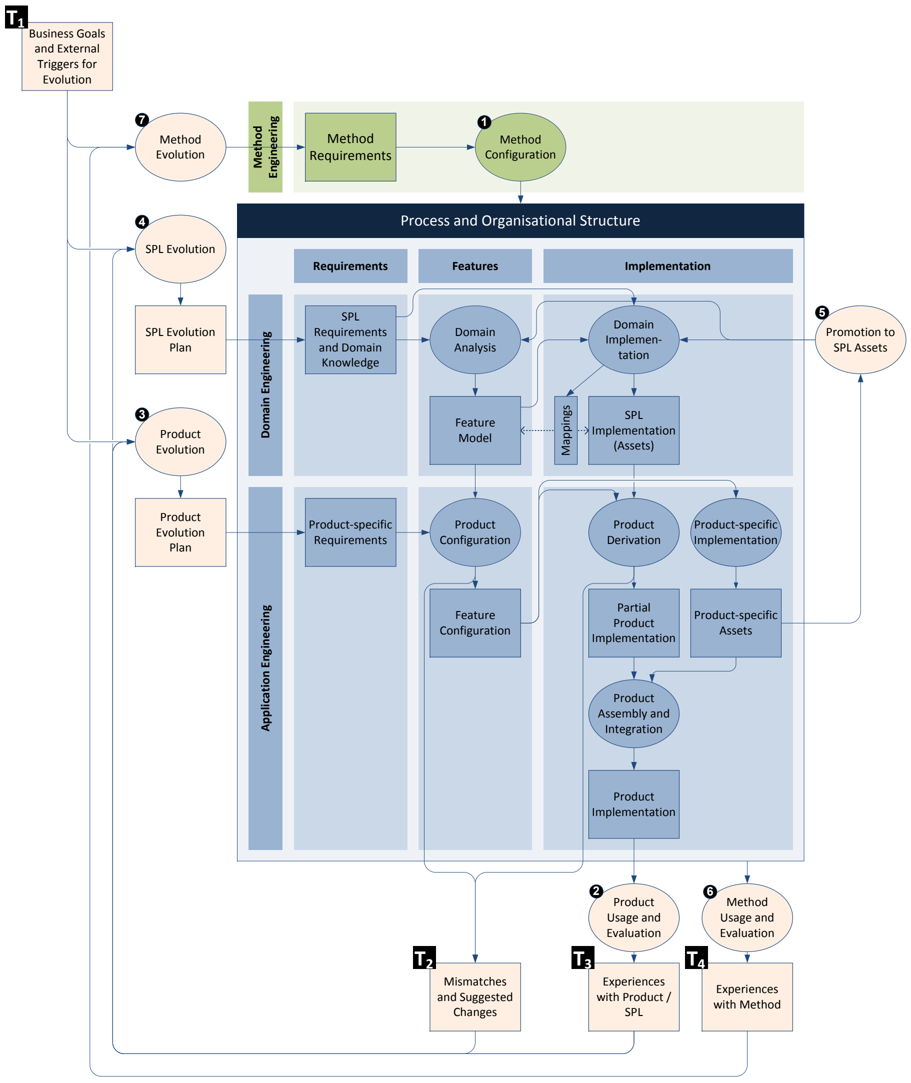


Fig. 9.7: Process framework for SPL evolution.

Eventually, application engineering yields *product implementations*. Then, *product usage and evaluation* ❷ results in *experiences with the product and the SPL* respectively. These experiences combined with mismatches and suggested changes provide input to *product evolution* ❸ as well as *SPL evolution* ❹. The latter often includes the *promotion of product-specific assets to SPL assets* ❺. The execution of domain engineering and application engineering, i. e. *method usage and evaluation* ❻ yields *experiences with the method* and can trigger *method evolution* ❼, e. g. adaptation of the process and organizational structure. For instance, it might be decided that to improve product quality additional testing activities will be introduced on application engineering level.

9.4.1.2 Evolution strategies

The process framework in Figure 9.7 shows multiple ways how SPLs evolve: evolution can take place on different levels and be caused by different triggers.

Concerning the level of evolution, there is SPL Evolution (❹ in Figure 9.7) and product evolution ❸. A specific case is when product-specific assets are promoted to SPL level ❺. (Note that this also complies with the discussion in Section 9.3 where making a product-specific asset generic corresponds to promotion to SPL level.)

Concerning triggers for evolution, there are *business goals and external triggers* T_1 for evolution, *mismatches and suggested changes* T_2 resulting from product derivation, and *experiences with the product/SPL* T_3 (leaving aside method evolution here).

Deelstra et al. [230] and Schmid et al. [754] describe several SPL evolution strategies that commonly occur in practice. They can easily be related to our process framework by classifying them according to the level of evolution and the triggers. Figure 9.8 shows a taxonomy of evolution strategies where these strategies (or situations) are classified according to its trigger and the level of evolution on which it takes place. We describe each situation in the following.

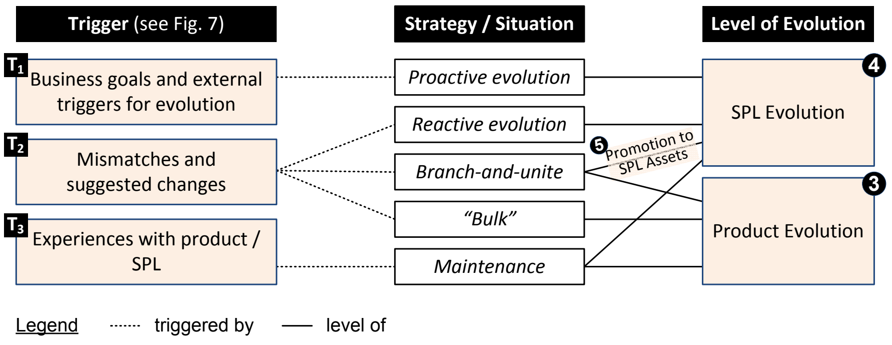


Fig. 9.8: Categorization of SPL evolution strategies with respect to trigger and level of evolution (from Figure 9.7).

Proactive evolution refers to proactively planning future requirements and adding them on the SPL level. This is a pure domain engineering activity where evolution planning is based on business goals and external triggers for evolution (such as market changes).

There are three common ways how to deal with mismatches and suggested changes that arise during product derivation:

Reactive evolution refers to integrating new requirements that arise during product derivation directly into the SPL, e. g. as variable assets. This means that reactive evolution is performed on SPL level. The advantages are the immediate possibility for reuse in future products and the avoidance of product-specific implementations or multiple branches. Highly automated approaches, like model-driven SPLs, often aim for this strategy to avoid product-specific implementations so that complete products can be derived automatically from the SPL. The disadvantages are required frequent changes on SPL level and the potential need to co-evolve already existing products. Also, creating product-specific functionality as a reusable asset can result in extra effort.

In the *branch-and-unite* approach, product-specific requirements are initially handled on product level, e. g. by creating a new product-specific branch. Later on, the product-specific branches can then be reunified with the SPL after releasing the product (promotion to SPL assets). In this way, the frequency of changes to the overall SPL can be reduced and emphasis can be put on the concrete product first. On the other hand the merge can become complex. A related concept is the grow-and-prune model which states that in large systems quick reaction to changes often requires copying and specialization (grow) and later on needs to be cleaned up by merging and refactoring (prune) [284].

The *bulk* situation occurs when an organization ends up with too many branches by evolving on product level only. This can lead to quality and maintenance problems and major effort is required to reintegrate the branches into the SPL.

Beside the strategies above that mainly deal with changing requirements, there are also other *maintenance* activities caused by experiences with the product and the SPL, like refactorings and correction of bugs that occur over time. The level of evolution then depends on whether the assets to be changed reside on SPL level or are product-specific.

9.4.1.3 Other process models

All SPLE frameworks described in the literature (e. g. [193, 688]) cover the main SPLE activities (*process and organizational structure* in the center of Figure 9.7). Some approaches extend this to address evolution on various levels (activities around *process and organizational structure*).

For instance, Bayer et al. [85] present PuLSE, a generic framework for SPLE, including the PuLSE-EM module, which covers evolution and maintenance. Based on information provided as a result of other PuLSE modules, PuLSE-EM accumulates knowledge and history information (e. g. a product configuration history and

PLA history) and restarts other modules (for scoping, domain engineering, and application engineering) with adaptations.

Similarly, the ConIPF method suggested by Hotz et al. [407] considers "mismatches" arising during product configuration and realization on the application engineering level and feeds them back into domain engineering where they are assessed and required changes are identified. These required changes are processed by an "evolution and development" activity, which leads to evolved and new assets as well as updates in configuration models.

There are several other SPLE methods that describe process structures for SPL evolution, e. g. [9, 176, 339, 809]. Further approaches which are more focused on the migration of existing groups of products towards SPLE are discussed later in Section 9.4.3.

9.4.2 Modeling Evolution and Change

A prerequisite to handle evolution in a systematic way is the ability to explicitly specify evolutionary changes. This is required during analysis of the evolution history of an SPL (to capture and specify observed changes), during planning of future evolution (to specify potential future changes and reason about them), and during implementation of evolution (to specify the changes to be realized).

On a lower abstraction level, like source code files, changes can be handled with the same tools as for single product development, like source code versioning systems. However, the higher the abstraction level (e. g. to view the evolution of an SPL as a whole), the more SPL specifics, like variability, need to be taken into account.

In earlier work in [134, 135, 686] we suggested feature models as a suitable means of abstraction to describe the overall evolution of an SPL, as features represent an SPL in a way that is meaningful to different stakeholders. Hence, evolution of an SPL is represented as a sequence of feature models over time. We will now first introduce an example and then discuss concepts for modeling evolution and change on this base.

Figure 9.9 shows a small example from the e-shop domain. It shows the four versions of the SPL's feature model at four different points in time, including historic evolution (2012) and planned future evolution steps (2014 and 2015). In this example, the version in 2012 supports only a Catalog and ShippingOptions with optional support for CarrierSelection. The version in 2013 (today) has been extended by support for Search which is available either as BasicSearch or, with extra costs, as an AdvancedSearch that supports a more intelligent search algorithm. The planned version for 2014 will distinguish between ElectronicGoods (which can be either shipped or downloaded directly) and PhysicalGoods that need to be shipped. Hence, ShippingOptions has become an optional feature and a cross-tree constraint has been added. In this version, AdvancedSearch will not be supported as it requires additional time to integrate it with the changes on Catalog. For 2015 it is planned to support an

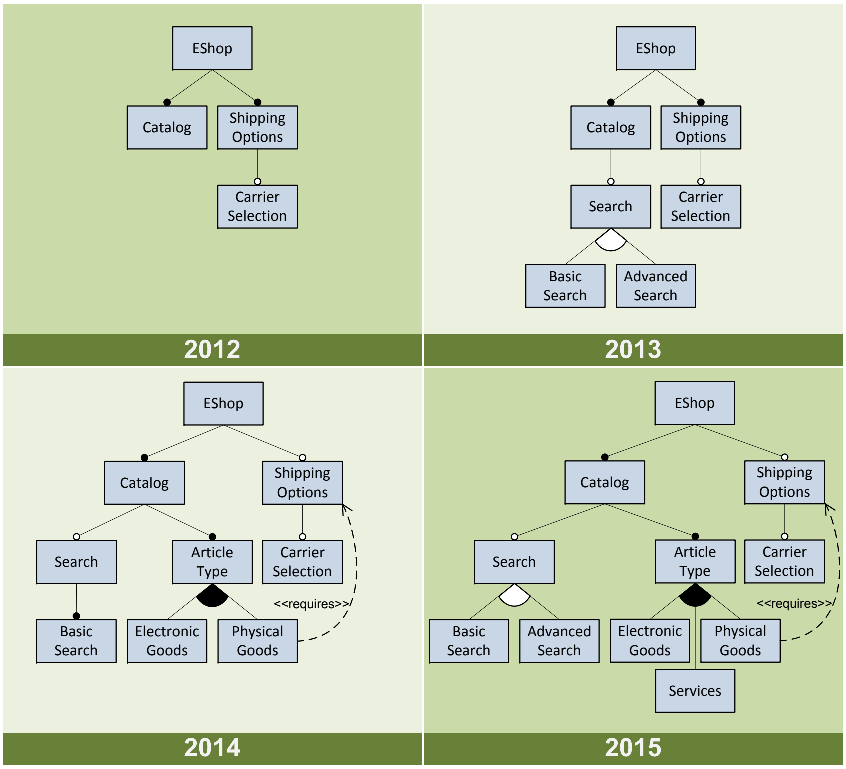


Fig. 9.9: Evolution of an SPL as a sequence of feature models

additional article type Services and to support AdvancedSearch again for all article types.

The remainder of this section describes how to model the changes between different versions of an artifact using the example of feature models above. Analogous to other areas like metamodel evolution (Chapter 2), there are two basic ways to specify such changes: 1) by modeling the *differences* between them (Section 9.4.2.1) or 2) by describing the performed modifications in terms of *change operators* (Section 9.4.2.2). Finally, Section 9.4.2.3 provides a more detailed example using a *combined approach.*

9.4.2.1 Difference Models

Approaches which are specifying the *differences* between versions work similar to approaches for program differencing [470] or common source code versioning systems that determine differences between versions of text-based files based on heuristics on the level of lines or characters. On the level of models, a *difference model* can be used that contains the changes between two versions in terms of added, removed, and modified elements.

Figure 9.10 shows an example for the evolution step from 2013 to 2014 in our e-shop example: the xor-group and its child feature AdvancedSearch has been removed. The features ArticleType, ElectronicGoods, and PhysicalGoods and their relationships and constraints have been added. In addition, ShippingOptions has been modified to become an optional feature. Context elements (represented by light color in Figure 9.10) are used to specify the locations in the model, e. g. where to add new elements.

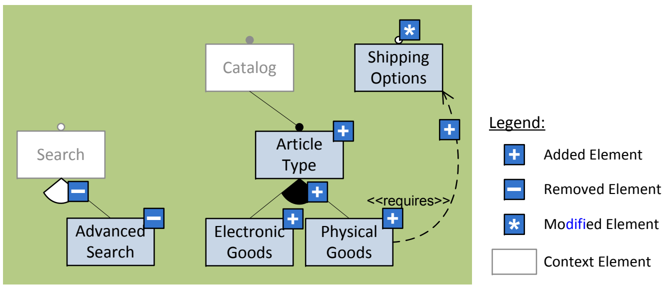


Fig. 9.10: Difference model for the evolution step from 2013 to 2014.

Several approaches have applied such concepts in context of SPLs: Acher et al. [5] provide a formal approach to identify the syntactic and the semantic difference between two feature models. Schäfer et al. [749] define a concept of difference models (called *delta models*) and apply it e. g. to specify multiple products in terms of differences to a core product. Hendrickson et al. [390] use difference models (called *change sets*) and relationships between them to specify the architecture of different products by combinations of change sets.

At this point, an important observation can be made: Specifying changes is not only relevant in context of evolution but also in context of variability, e. g. to specify the differences between multiple product variants in an SPL. In context of SPLE the latter is called *variability in space* while evolution can be considered as *variability in time*. Hence, it is not only possible to apply change modeling concepts to describe variability in an SPL (like Schäfer et al. and Hendrickson et al. mentioned above) but also to apply variability modeling concepts to specify evolution. An approach that makes use of this idea is EvoPL described later in Section 9.4.2.3.

9.4.2.2 Change Operators

The second basic concept to specify changes are *change operators*. A change operator describes an operation performed on a model to achieve a change. There are three atomic change operators *add*, *delete*, and *modify* that have the same semantics as the elements in difference models. However, the main difference is the possibility of more complex operators that allow to express richer semantics about a change like "*split feature f into f_1 and f_2*".

Semantically rich operators are usually defined for a specific modeling concept (e. g. feature models or metamodels, see Chapter 2) and can also be optimized for a specific purpose. For instance, in context of SPL refactoring Alves et al. [25] define a set of change operators on feature models that do not change the feature model's semantics (e. g. "*convert or to optional*" or "*push up node*"). Seidl et al. [761] define change operators in context of implementing SPL evolution which are discussed later in Section 9.4.6.

9.4.2.3 Combined Approach

An approach that combines concepts of difference models and change operators to model long-term evolution of an SPL is *EvoPL* [686]. It also leverages the idea of considering evolution as *variability in time* introduced above.

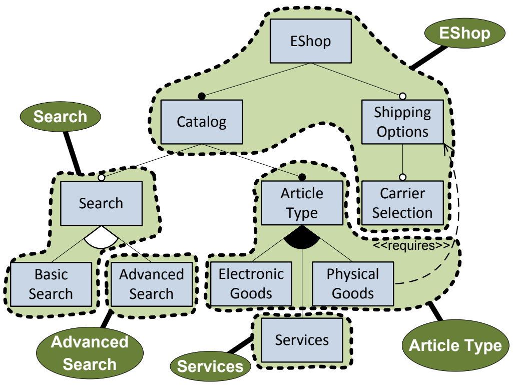
EvoPL focuses on feature models as main artifact to manage SPL evolution. The approach is intended to be used for both, analyzing past evolution (see Section 9.4.4) and planning future evolution (see Section 9.4.5).

In EvoPL, each feature model version is composed of model *fragments*. Figure 9.11a shows the fragments for the evolution in Figure 9.9. A fragment clusters related feature model elements that are added or removed only together during the same evolution step. The purpose of fragments is to raise the level of abstraction by representing multiple related elements. Each fragment has a unique name and is stored together with a context element (the parent feature) specifying its location within the overall feature model. In this way, each feature model at a certain point in time can be described as a composition of fragments.

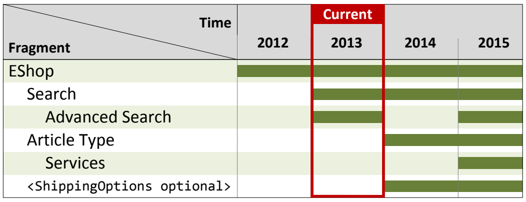
Changes within fragments, e. g. changing a feature from mandatory to optional or adding a cross-tree constraint, are specified by change operators (called *evoOperators*, see [686] for details) associated with the fragments. For instance, ShippingOptions are changed from mandatory to optional in 2014 which is defined by a change operator <ShippingOptions optional> that is applied to the versions for 2014 and 2015.

The overall evolution is then specified using the concept of "variability in time": The fragments and evoOperators themselves are stored in a specific kind of feature model (called *EvoFM*) that specifies their hierarchy and other dependencies between them. Each evolution step can then be represented by a "configuration" of the EvoFM, i. e. a selection of fragments and evoOperators that together make up a feature model. The evolution of a feature model can, hence, be represented by a sequence of EvoFM configurations.

We visualize this by a representation that we call *evolution plan* (Figure 9.11b). The horizontal dimension represents the time line; each column represents an evolution step. The vertical dimension represents the EvoFM; each row represents an EvoFM element, i. e. a fragment or an evoOperator (the latter denoted in angle



(a) Clustering into fragments



Fragment \ Time	2012	Current 2013	2014	2015
EShop	🟩	🟩	🟩	🟩
Search		🟩	🟩	🟩
Advanced Search		🟩		🟩
Article Type			🟩	🟩
Services				🟩
`<ShippingOptions optional>`			🟩	🟩

(b) Evolution plan

Fig. 9.11: Fragments and resulting evolution Plan

brackets). Each cell in the plan represents a configuration decision, i. e. whether the fragment or evoOperator is selected (i. e. applied) in that version or not.

For instance, the evolution plan in Figure 9.11b represents the evolution steps from Figure 9.9: In 2012, only the fragment EShop is selected (applied). In 2013, the fragments EShop, Search and AdvancedSearch are applied. In 2014, AdvancedSearch is no longer applied while ArticleType and the change operator <ShippingOptions optional> are applied. In 2015 all fragments and change operators are applied.

Figure 9.12 shows the overall workflow with EvoPL: A model transformation enables to automatically extract an evolution plan from a given sequence of feature models. The evolution plan is then used to plan future evolution by adding new evoConfigurations (and, if necessary, new fragments and evoOperators). Please note that fragments are never modified (as evoOperators are used instead) except for splitting fragments which can become necessary if in a future evolution step a subset of a fragment should be removed. Once planning of future versions has been finished, another model transformation supports automated composition of the resulting feature models. Due to the incremental nature of the model transformations it is possible at any time to update the evolution plan to include changes on feature model level and, in turn, to re-generate feature models after the evolution plan has been modified.

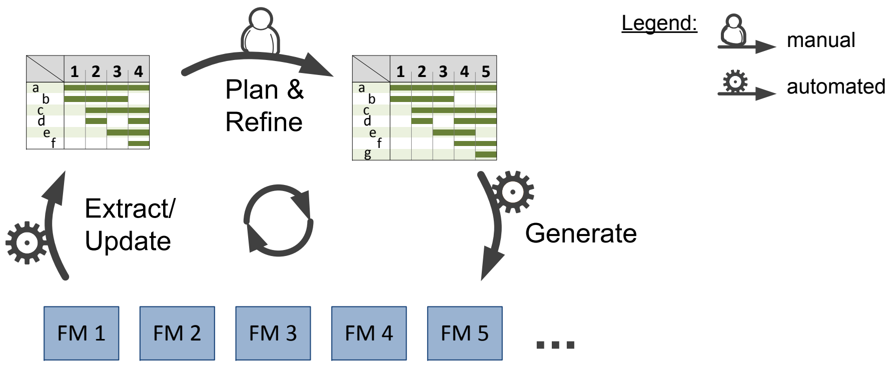


Fig. 9.12: Workflow and transformations on feature model level with EvoPL

An advantage of the evolution plan representation is its degree of abstraction. As demonstrated in [686], abstraction into fragments can significantly reduce complexity when dealing with large evolving feature models while the evolution plan provides a comprehensive overview of the different versions. Another advantage of the approach is its support for order-independent planning. Changes are not specified relative to the previous version (or a common baseline) but by selecting or eliminating fragments and change operators. This enables incremental planning of multiple versions in parallel or specifying a later version before its predecessors have been fully defined.

9.4.3 Migration to SPLE

In practice, the introduction of SPLE often arises when after some success in a market segment a company finds itself with a family of products. Hence, when discussing the *adoption* of SPLE, besides starting SPLE from scratch we have to consider approaches which evolve SPLs from legacy products and focus on *migration* and *mining of existing assets*. We can distinguish four types of SPLE adoption [228, 754]:

- *Independent* - A new SPL is created independently of any existing products.
- *Leveraged* - A new SPL is set up based on an existing one.
- *Project-integrating* - With an existing product base as background, a set of projects (developing new products) is selected to contribute to an SPL.
- *Reengineering* - From existing legacy products, assets are extracted and reengineered to contribute to a new SPL.

In addition, we can distinguish between *revolutionary* ("big bang") and *evolutionary* (incremental) models [130, 754]. This is somewhat orthogonal to the four adoption types, however, adoption types that take existing systems into account (*project-integrating*, *reengineering*) are most amenable for an evolutionary approach.

In the literature there are numerous approaches to SPL migration, e. g. [86, 140, 228, 284, 472, 778]. In the remainder of this section we will describe various aspects and activities of such migration approaches, i. e. *initiation of a migration project* (Section 9.4.3.1), *scoping* (Section 9.4.3.1), *variability analysis* (Section 9.4.3.3), *refactoring* (Section 9.4.3.4), *extraction of assets* (Section 9.4.3.5), and *assessment* (Section 9.4.3.6). While conceptually such aspects can be interpreted as a logical sequence of activities, in practice they are often performed in an iterative fashion (see, e. g. [86]). For instance, an initial analysis of variability among existing products might lead to a preliminary selection, which is followed by a more detailed variability analysis.

9.4.3.1 Initiation of the migration project

At the beginning of a migration project the relevant base information needs to be collected. This might include, e. g. information on product capabilities, evolution so far, existing software architectures, documents about scope and existing assets, and preliminary estimates of required changes (interface vs. deeper changes) [93, 785]. Relevant information can be extracted from artifacts or gathered by interviewing product experts, maintainers, and users [785].

Then, based on a first assessment an approach for the mining of assets can be drafted. It needs to be decided on which abstraction levels (e. g. features, components) the mining will happen and whether further processing (e. g. refactoring) is necessary.

A migration project must also consider business and organizational aspects. First, the advantages and potential drawbacks of the various options (current situation vs. introducing SPLE) need to be considered, e. g. with an estimation of costs and productivity benefits. Second, various organizational structures are possible. For instance, SPLE can be performed in product teams or in a separate dedicated SPL team [112].

9.4.3.2 Scoping and assessment of migration options

Similarly to general SPLE approaches [444, 785], in SPL migration the scope of the overall effort needs to be defined, i. e. deciding about which features to include in the SPL and which are out of scope. Since this might require additional input (e. g. a prioritization of features) the scoping might have to be performed after or in combination with other activities (e. g. after an initial variability analysis).

In scoping we can take a problem-oriented perspective (“What does the customer value most?”), but we also need to consider solution-oriented aspects (“What can we implement most easily?”) [778].

9.4.3.3 Variability analysis

The migration of existing products into an SPL often starts with an analysis of their commonalities and variabilities. This can be performed on various abstraction levels, e. g. (1) on the level of requirements, customer visible functionality and features or (2) on the level of implementation artifacts.

On higher abstraction levels we can apply techniques for *commonality and variability analysis* [688], e. g. an application-requirements matrix (table of products vs. requirements), priority-based techniques (requirements are rated by different stakeholders) or checklist-based analysis (collecting and analyzing requirements with the help of various checklists).

On more concrete levels, we can analyze implementation artifacts to *extract variability models*. When reverse engineering higher level variability models, we might have to apply heuristics and involve experts to (re-) construct their structure [773] (cf. Section 9.4.4.2).

Just like in general SPLE approaches, in a migration project we have to anticipate future changes. This includes product and feature planning, the anticipation of future features [778] and the analysis of consequences for the PLA and implementation assets.

9.4.3.4 Refactoring

Before the actual extraction of SPL assets is performed, it is often necessary to refactor existing artifacts [472], e. g. to remove accidental differences and increase commonalities. This can occur on various abstraction levels, e. g. when refactoring code [532, 857] or when restructuring the software architectures of existing products to prepare them for merging them into a shared PLA [228].

We have to distinguish such *preparative* and *transformative* refactoring (preparing assets for a migration, transforming them into SPL assets) from refactoring of the SPL after it has been established, e. g. on the conceptual and feature-model level [25, 128] or of PLAs [204].

Related techniques are feature-oriented restructurings, which are not predominantly aimed *towards* an SPL, but rather use feature-orientation as guiding concepts, e. g. with the help of regression tests [583] or when aiming to untangle and separate concerns on the implementation level [624].

9.4.3.5 Extraction of assets

One of the key artifacts when establishing SPL practices is a PLA. While in other scenarios it might be appropriate to design a PLA from scratch, we have to take a different approach when migrating existing products into an SPL. Here, techniques for architecture and component recovery can be applied, e. g. *Option Analysis for Reengineering* (*OAR*) [94] and *Mining Architectures for Product Lines*

(*MAP*) [655, 796]. In many cases, the approach will be to extract product-specific architectures [228] and then analyze and merge them, e. g. using techniques for model merging [174, 735].

In that course, mechanisms for variability realization [810] have to be selected and, based on earlier variability analysis, variation points have to be chosen.

Alongside the extraction of a PLA, the corresponding core asset implementations need to be extracted and refined [93, 94]. Here, we have to address the mapping of variability models onto implementation artifacts (*feature location*) [255], the identification of similar implementation artifacts (*clone detection*) [475, 587] and the analysis of feature implementations with respect to dependencies as well as interactions (*dependency analysis*, *feature interaction*) [42]. Dependencies that are detected on the implementation level need to be propagated up to higher abstraction levels (PLA, variability/feature model).

During a migration to SPLE, often SPLE activities and reverse engineering activities are performed side by side. For instance, Bayer et al. [86] suggest to integrate the reverse engineering of existing assets and the creation of SPL models via a "blackboard", i. e. a shared workspace allowing reengineering and SPL activities to exchange and incrementally enrich information.

9.4.3.6 Assessment

After the SPL infrastructure has been established, the created artifacts, in particular the PLA, should be evaluated [130]. Here, architecture evaluation methods [257] and analysis of selected product instances [130] can be applied. Of particular interest in the context of this chapter is the assessment of the PLA with respect to is evolvability/maintainability. Since the context (requirements, business goals) and the PLA will change over time, reevaluations should be performed [785].

The result of such a migration project provides input for subsequent activities. For instance, migration can provide a first draft of an evolution plan for the near future, based on features that do not exist yet but are anticipated.

9.4.4 Analyzing Evolution

This section discusses the analysis of the current status and the evolution history of an SPL as a basis for planning (Section 9.4.5) or to predict future evolution. We first briefly discuss repository mining techniques, then analysis approaches on the feature and architectural levels, and finally prediction of maintenance effort and evolution via simulation.

9.4.4.1 Mining software repositories

Approaches for *Mining of Software Repositories* (*MSR*) [216, 451] (cf. Chapter 5) process various data sources, e. g. source code repositories, bug databases, and mailing lists. Often different input sources are combined to gain better results. Approaches aim to uncover relationships and trends, e. g. using data mining techniques. Examples of extracted information are the growth of a system, change relationships between assets, or the reuse degree of components. Kagdi et al. [451] categorize MSR approaches into two types. Some approaches answer *market-based questions*, i. e. "If A occurs then what else occurs on a regular basis?" (resulting, e. g. in association rules). Other approaches answer *prevalence questions*, e. g. the number of functions reused or if a particular function was changed. Orthogonal to that, Kagdi et al. distinguish between approaches measuring *changes to properties*, i. e. calculating metrics for each version and then comparing over different versions, and approaches focusing on *changes to artifacts*. There is a large spectrum across levels of abstraction (e. g. features, architecture, source code) and granularity addressed by such approaches. SPL-specific issues, like variability or distinction between SPL and products, have received little attention to date.

9.4.4.2 Analyzing features

When analyzing the evolution of an SPL, feature-oriented analyses are of obvious interest. Various work on *feature location* [255] aims to establish traceability between features and assets that implement them (see Section 9.4.3.5). In an existing (model-driven) SPL, such traceability might already exist explicitly. However, in less structured SPLs, e. g. with many product-specific implementation parts, feature location can be essential for refactoring and migration (see Section 9.4.3).

Other approaches aim to reverse engineer the feature model, starting, e.g., from a set of unstructured features [773], the architecture [4], or even informal product descriptions [219]. One can expect that an automatically extracted feature model differs from one that is manually crafted by a human software architect; however there is not sufficient empirical data on this yet. To close this gap, Hsi and Pots [409] suggest to extract features from an application's user interface and to link them to code assets, e. g. via the operations called by user interface actions.

9.4.4.3 Architecture assessment

Most existing SPL-specific approaches for analysis of evolution address the assessment of the PLA. This can be useful both during creation of an SPL (see Section 9.4.3.6) as well as on existing SPLs. Maccari [549] applies the *Architecture Trade-off Analysis Method* (*ATAM*) [257] to assess the suitability of the PLA for future requirements.

Johnsson and Bosch [445] aim to quantify SPL aging. They measure average costs per maintenance task as well as the relative distribution of effort among adding components, adding functionality to components, and changes to existing functionality. They argue this can be used to detect architecture erosion and the related increase in maintenance effort. This can in turn be used to decide on the reorganization or retirement of SPLs.

9.4.4.4 Prediction based on simulation

Heider et al. [381] propose to simulate SPL evolution to predict the long-term development of maintenance effort and model complexity. The analysis is performed on a problem space model (i. e. decision or feature model), a solution space model, and dependencies within and between them. The simulation modifies the models with random operations based on probabilities defined in profiles. For instance, the "evolution profile" describes the type of evolution to be performed, like "continuous evolution", "refactoring", or "product placement" (i. e. changing mostly the problem space while keeping the solution space mostly unchanged). The evolution profile can be created based on existing evolution history.

9.4.5 Planning Evolution

This section deals with planning of evolution, i. e. how to decide about changes to the SPL to be implemented in the upcoming versions.

Usually important planning decisions on the evolution of complex software systems require careful consideration. Ad-hoc planning would bear the risk of deficiencies like insufficient anticipation of future requirements, lack of resources to realize new requirements, or loss of knowledge about previous decisions [736]. In single system engineering there are several research strands – like rationale management and release planning – that aim to reduce this risk by supporting systematic planning and decision-making. These concepts can be applied to SPLE.

An important prerequisite for deciding on future changes is to gain knowledge on the impact of a potential change. However, for an SPL this can be much more complex than in single system engineering due to the complexity of interdependencies between artifacts (see Section 9.3).

In this section we first discuss change impact analysis in SPLs and then present approach for decision-making in SPL evolution.

9.4.5.1 Change Impact Analysis

When deciding about a change we have to predict the required effort and potential pitfalls for its realization. This is supported by approaches for impact analysis [124, 173].

An important aspect of impact analysis is traceability, i. e. storing links between all logically related assets in the software development process to understand what other assets might be affected if an asset changes. An example are traces between a requirement and its implementation assets. In context of SPLs, the mapping between features and implementation assets (if fully specified) can be considered as a kind of trace link. However, traceability approaches consider additional types of links and often add some extra information to each link (like a rationale description).

For instance, Anquetil et al. [34] propose a traceability framework for SPLs. They propose four general categories of trace links: *Refinement* traceability relates artifacts from different level of abstraction like an element in the design model and its implementation. *Similarity* traceability relates artifacts at the same level of abstraction such as similar requirements that have some logical relationships or similar elements from different architectural views. *Variability* traceability relates artifacts as relevant for variability management like the mapping between a feature and its implementation. *Versioning* traceability relates successive versions of an artifact. As pointed out by Heider et al. [386], traceability needs to cover not only all assets on SPL level but also on product level. Other traceability approaches for SPLs can be found e. g. in [8, 442, 616].

Defining (and maintaining) trace links requires much effort, so there is a need for tool support. There are two ways how to acquire trace links in a tool-supported way: 1) ex-post by statically analyzing existing artifacts or 2) during development when artifacts are created. Heider et al. present *EvoKing* [383], an IDE for SPLs which supports both strategies. EvoKing supports monitoring evolution by keeping track of all assets and their relationships within an SPL. To provide some degree of abstraction, users can define the types of assets and relationships and how the tool interprets events like creating or modifying an asset of a certain type. Trace links are established according to user-defined rules. For instance, whenever a product configuration is created, a trace link is established to the underlying variability model. These rules can also be applied to existing artifacts. However, heuristics or statistical analysis to acquire trace links automatically have not been applied yet. Other approaches that aim to provide automated extraction of traceability links for SPLs are e. g. [442, 747].

Beside traceability approaches, there are only few other approaches for impact analysis in context of SPLs. Heider et al. [385] present an approach using regression testing to analyze the impact of changes on SPL level on products. Whenever the SPL is changed, the tool first analyses whether the existing product configurations need to be changed as well, e. g. whether configuration decisions need to be modified due to changes on the variability model. In a second step, the tool re-derives all products and compares them with their previous version and reports the differences.

In this way it provides instant feedback to developers about the consequences of a change on the SPL.

9.4.5.2 Decision making in SPL evolution

Planning evolution means to make decisions that may have essential impact on the future success. Concepts like the *QOC* approach (*Questions, Options and Criteria*) [550] provide general support for systematic decision-making. The first step in QOC is to define the issue on which to decide (*question*). Second, available solution *options* are identified and specified. In addition, *criteria* are defined by which the available options can be rated. Examples for criteria are the expected development effort (e. g. estimated by an impact analysis as above), market value, strategic benefit or risk. Each solution option is rated according to these criteria. Finally, an option is selected on this base. This allows systematic decision making and captures the reasons behind a decision.

Approaches in the area of *release planning* [736] apply such concepts to decide about new requirements (or features). For instance, when a set of new candidate requirements is given (e. g. due to customer requests and market analysis) they support to prioritize them and to select those to be implemented in the next release(s). Similar to QOC, criteria have to be defined by which the requirements can be rated. Usually ratings are performed by multiple stakeholders including e. g. prime customers. Criteria and stakeholders can be prioritized by assigning weights. Moreover, constraints can be defined to specify preconditions such as available resources (e. g. person months until next release) and dependencies between the candidate requirements (e. g. two requirements exclude each other). After each requirement has been rated according to the criteria, approaches like EVOLVE [643] automatically propose a candidate release plan that conforms to the defined constraints.

Similar concepts have been applied to SPLs for scoping (see Section 9.4.3.2), i. e. selecting which features from an existing set of related to include in an SPL. The PuLSE-Eco approach by DeBaud and Schmid [229] proposes to refine business goals into “benefit functions” (e. g. effort saved by making a feature reusable) which are decomposed into basic “characterization functions” (e. g. implementation effort in person months) by which each potential feature is judged. In this way the benefit of each feature is calculated as a base for the decision which features to include into an SPL.

Besides deciding about new features or requirements, which is similar to single system engineering, SPL evolution also needs SPL-specific decisions like 1) deciding about whether changes should become product-specific or be performed on SPL level and 2) about the variability of features on SPL level. In the following we describe an approach for each of these issues.

Heider et al. [382] address decision making about whether new requirements that arise on product level should be promoted to SPL level. Their tool EvoKing (see Section 9.4.5.1) provides SPL engineers an overview on new requirements that have arisen on product level. SPL engineers can then decide about each requirement to either lift it to SPL level or to assign it to developers on product level otherwise. In [380] the authors describe how this decision is supported by a Win-Win model negotiation approach. *Win-Win* [120] is a general approach similar like QOC but with a focus on brainstorming and negotiation: Different stakeholders define their objectives as *win conditions*. Win conditions where all stakeholders agree on are stored as *agreements*. Otherwise conflicts, risks or uncertainties are defined as *issues*. Stakeholders then brainstorm for *options* to resolve these issues and to explore trade-offs with the goal to find an option that can be turned into an agreement. In context of new SPL requirements, the win conditions are the new requirements proposed by SPL engineers and product engineers. Issues raise, for instance, when there are inconsistencies between the requirements on these two levels or between requirements of different products.

Thurimella and Brügge [843] address decision-making about the variability in SPLs. They apply similar concepts like in QOC. To decide about variability, the possible solution options are identified (e. g. whether a feature is mandatory or optional) and rated by criteria. The same principle is also applied to product configuration decisions where available variants can be considered as options.

Basically, concepts like QOC or Win-Win can be applied to any particular evolution decision [842]. Besides the concrete approach used, any additional tacit knowledge underlying a decision (e. g. why an option was finally selected) should be explicitly documented by a *rationale description* to preserve the knowledge for future decision-making [824]. For instance, the EvoPL approach from Section 9.4.2.3 has been extended by support for decision-making by modeling high level goals, criteria, rationale, and the relationships between them [758].

As discussed in [759], it should be considered that planning information – such as goals, criteria, and rationale – evolves itself (e. g. changing business goals). It is useful to handle this evolution in a structured way as well (e. g. traceability of previous versions of a goal description) to preserve the information and understand previous decisions.

9.4.6 Implementing Evolution

Existing work on the implementation of SPL evolution aims to support realizing changes in a systematic way. It can be classified according to the abstraction layers in Figure 9.4. On the SPL level, changes to requirements lead to changes in the variability model. Several works aim to support changes to variability models and the associated mappings while preventing inconsistencies (Section 9.4.6.1). Other work focuses on realizing changes on the implementation level, e. g. by structuring the implementation according to features (Section 9.4.6.2). When an SPL has changed, this has to be propagated to existing products (Section 9.4.6.3).

9.4.6.1 Evolution of the variability model and its mappings to assets

Changes to requirements often lead to changes in the variability model, e. g. adding or removing features or splitting a feature to make some part of it variable. Also, the variability model has to be maintained itself, e. g. by restructuring to improve readability. Changes to the variability model can be (tool-) supported by change operators (cf. Section 9.4.2) to systematize changes and to update the mappings to implementation assets.

Thüm et al. [837] provide a tool that analyses changes performed on a feature model and classifies them into one of the four categories: (1) *refactoring*, not changing the set of valid products, (2) *generalization*, only adding products, (3) *specialization*, reducing the set of products, or (4) *arbitrary changes* otherwise:

Refactoring, as used by Thüm et al. [837], refers to changes of the feature model that do not change its semantics in terms of valid product configurations. The source and target feature model are then referred to as "equivalent" [803]. Such changes include, for instance, restructuring of the feature model, and changes that do not influence product configurations (e. g. renaming a feature).

Generalization refers to changes that only add products (i. e. valid configurations) to the SPL. All existing products remain valid and can still be derived from the SPL. Simple examples for such changes are adding a new optional variant or changing a feature from mandatory to optional. A catalog of feature model operations that preserve the set of products is presented in [25]. In contrast to [837], the authors call these types of changes "refactorings" or "refinements". Based on a formal notation for refinements introduced by Borba et al. [129], Neves et al. [642] specify several complex change operations for common behavior-preserving changes of SPLs which include not only the feature model itself but also the mappings and the associated assets. For instance, a new mandatory feature can be safely added to a feature model only if it represents functionality that is already part of all products (e. g. to convert it into an optional feature later on).

Specialization is mainly used during staged configuration [209], i. e. configuration performed in multiple steps, e. g. by various stakeholders. Variability is reduced in each step until it is completely resolved and exactly one product remains.

A tool that aims to support *arbitrary changes* is Feature Mapper [762]. It focuses on the co-evolution of feature models, implementation assets, and the mappings between them. On the level of feature models it supports several change operators, like "add feature", "split feature", "remove feature", or "remove feature and owned asset". On the level of assets (also represented as a model) changes depend on the concrete type of model. The authors provide some examples for UML models (e. g. "replace method with method object") and for models representing Java code (e. g. "extract method"). The authors classify three types of changes (focusing on consistency of feature mappings): (1) changes that only have effects within one model (like changing an optional feature to mandatory or renaming an asset); (2) changes that affect a model and the mapping (like "split feature" or "extract method"); and (3) changes that affect the model, the mapping, and the mapped model (like "remove feature and owned assets”). In the second and third case, the tool automatically updates the mappings to keep them consistent.

Beside the work on feature models, there are also approaches addressing other types of variability models, e. g. decision models and their associated assets [385].

The change operators and tools described above cover only a subset of possible changes to an SPL. Other changes that require manual implementation (like adding new functionality to an SPL) cannot be specified just in terms of predefined operators. However, tool support should indicate potential inconsistencies after a change. In general, this can be achieved by analyzing mappings between assets (similarly to Feature Mapper as described above) or additional traceability links between dependent assets (see Section 9.4.5.1).

Finally, after performing changes, the consistency should be checked between the different abstraction levels in the SPL as shown by Vierhauser et al. [892]. Guo et al. [355] show how to check the consistency of large feature models so that only those parts which are affected by an evolutionary change need to be checked again. In the context of formal validation of SPLs, Cordy et al. [199] provide a model-checking approach that supports evolution. They define a method to identify specific types of features and show that for such features, when added to an evolving SPL, only a subset of the products need to be model-checked again.

9.4.6.2 Evolution of assets

Work supporting changes on lower levels of abstraction mainly addresses mechanisms to increase modularity and maintainability of assets. Garg et al. [313] provide specific tool support to specify changes or multiple variants for SPL architectures. The presented tool *Ménage* represents visual architecture models in terms of components and connectors based on xADL 2.0. Variability and different versions are visually highlighted.

Other work addresses evolution on the code level. Here, one challenge is to modularize code so that changes on higher abstraction levels, e. g. new features, can be implemented with as few side effects as possible. For this, techniques similar to those for implementing variability in code can be used. For instance, aspect-oriented development can be used to implement cross-cutting features [902]. Loghran et al. [541] propose supporting evolution by a combination of aspect-oriented techniques and frames, which are hierarchically ordered code templates. They provide code examples showing how these “framed aspects” can be used to support reuse and the easier integration of new features.

Some work addresses evolution at runtime [339] (cf. Chapter 7.6). For this, software reconfiguration patterns are used that allow the configuration in component-based systems to be updated during runtime. The authors describe multiple reconfiguration patterns based on existing architectural patterns, e. g. “master/slave reconfiguration” or “centralized control reconfiguration”, and discuss how to perform evolutionary changes based on them.

After changing the implementation, the SPL has to be tested. Here, the testing strategy should take variability into account to avoid that all possible feature combinations and all existing products have to be tested again [256].

9.4.6.3 Propagating changes from the SPL to products

Heider et al. [384] provide tool support to propagate changes from the SPL to individual products. In theory, model-driven SPLs allow products to be regenerated after the SPL has changed. However, as Heider et al. point out, in practice product configuration can be a complex and time-consuming process which requires decisions by multiple stakeholders. Hence, configuration of different products and evolution of the SPL is often performed in parallel. After a variability model has changed, product configurations must be updated by considering the dependencies in the variability model, between the assets, and between variants and assets. Hence, updating the product configuration can be challenging. The authors address this with a tool that supports automated updates of products, resolves conflicts, and assists users in manually resolving conflicts based on trace data when automated update fails.

9.5 Conclusions

In this chapter, we provided an overview of basic concepts and state-of-the-art in SPL evolution, as well as a short introduction to our own work in feature-oriented software evolution. Many challenges remain.

We need to improve support for handling changes, this includes *understanding consequences of potential changes* (taking all dependencies into account, across abstraction layers) and better support for the *propagation of changes*, for instance techniques that tolerate inconsistencies and resolve them, while propagating the changes.

With increasing scale and complexity it becomes infeasible to adapt the whole system (i. e. the whole SPL) to change in a short time frame. Hence, when an organization aims to react to market events or urgent customer requests, we require *strategies and techniques to support fast adaptation*, e. g. with product-specific extensions, which are later propagated into the SPL. Here, we have to consider an oscillation between adaptation/extensions and creating a consolidated shared infrastructure (“grow-and-prune model”, [284]).

Tracing concepts in the problem space to the solution space, from high abstraction levels to details of the software design and lines of code, is a fundamental problem in software engineering. Such mappings are often not one-to-one and ambiguous. For instance, in SPLE and SPL evolution we have to map features to their implementations, a problem addressed by *feature location*. Here, (1) a feature is potentially implemented by multiple classes, a class potentially contributes to multiple features and (2) it is not a clear-cut decision whether a class is part of a feature's implementation.

Finally, we have to deal with *evolution for PLE "in-the-large"*, for instance in hierarchical SPLs or a systems-of-systems context, which requires propagation of changes up or down the systems hierarchy. When dealing with evolution of large SPLs in all its aspects (migration, analysis, planning, implementation, etc.) we need to consider the potential hierarchical structure of such systems. For instance, Gall et al. [308] report that the characteristics of the evolution of a particular subsystem deviated substantially from that of the main system, an effect that can be masked when we would only consider the system as a whole.

As a final thought, we concur with Dhungana et al. [241] who argue that SPLE should treat evolution as a normal case and not as the exception. Hence, improved concepts and techniques are required that are able to handle evolution of large software-intensive systems while taking the particular characteristics of SPLs into account. We believe that there lies great potential in a smart combination of automated and interactive techniques, which combine the best of both worlds–efficiency through automated mechanisms and guidance towards creative solutions through the capabilities of the human engineer.