

Filtered Model-Driven Product Line Engineering with SuperMod: The Home Automation Case

Felix Schwägerl^(✉), Thomas Buchmann, and Bernhard Westfechtel

Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany
{felix.schwaegerl, thomas.buchmann,
bernhard.westfechtel}@uni-bayreuth.de

Abstract. Software Product Line Engineering promises to increase the productivity of software development. In the literature, a plan-driven process has been established that is divided up into domain and application engineering. We argue that the strictly sequential order of its process activities implies several disadvantages such as increased complexity, late customer feedback, and duplicate maintenance. SuperMod is a novel model-driven tool based upon a filtered editing model oriented towards version control. The tool provides integrated support for domain and application engineering, offering an iterative and incremental style of development. In this paper, we apply SuperMod to a well-known case study, the Home Automation System product line. We learn that the tool supports a broad variety of iterative and incremental development processes, ranging from phase-structured to feature-driven. Furthermore, it can mitigate the disadvantages of the traditional software product line development process.

Keywords: Software product line engineering · Software development process · Filtered editing · Model-driven engineering · Home automation example

1 Introduction

Software Product Line Engineering (SPLE) aims at systematic development of a family of software products by exploiting the variability among members thereof [1]. Core assets of different products are provided as the *platform*. Commonalities and differences among products are captured in *feature models* [2]. In the literature [3], a two-stage SPLE process is proposed (cf. Fig. 1): (1) During *domain engineering (DE)*, platform and variability model are defined. A *mapping*, e.g., *presence conditions* [4], specifies which part of the platform realizes which feature(s). (2) In *application engineering (AE)*, variability is resolved by specification of a *feature configuration*, and a product with the desired features is derived in a preferably automated way. For the definition of the platform, two distinct approaches exist: Using *positive variability*, a common *core* is defined to which specific features may be added. *Negative variability* proposes to specify

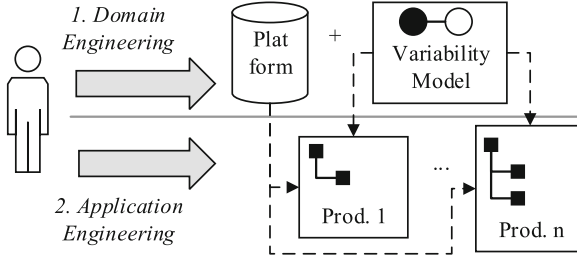


Fig. 1. The two-stage SPLE process as defined in the literature [3].

the platform as *superimposition* of product variants, from which elements must be removed to obtain a specific product.

Version Control (VC) has become indispensable for software engineers to control software evolution and to coordinate changes among a team. Version control systems (VCS) such as *Git* [5] or *Subversion* [6] provide an iterative three-stage editing model, which is shown in Fig. 2: (1) A developer *checks out* a specific revision of a software project from a *repository*. A copy of the project is created in the local *workspace*. (2) In the workspace, the developer *modifies* the project by implementing new functionality or by fixing bugs. (3) To make these modifications persistent and available to others, the developer *commits* his/her changes to the repository as a new revision.

Model-Driven Software Engineering (MDSE) [7] considers *models* as first-class artifacts, using well-defined languages such as the *Unified Modeling Language (UML)* [8]. Many model-driven applications are built upon the *Eclipse Modeling Framework (EMF)* [9]. The combination of MDSE with VC or SPLE is subject to many research activities, resulting in the integrating disciplines *Model Version Control* [10] and *Model-Driven Product Line Engineering (MDPLE)* [11], which improve tool support by raising the abstraction level of the artifacts subject to version control or variability.

Previous Work. In [12], we have elaborated a *conceptual framework* for the integration of SPLE and VC based on MDSE. The framework addresses the

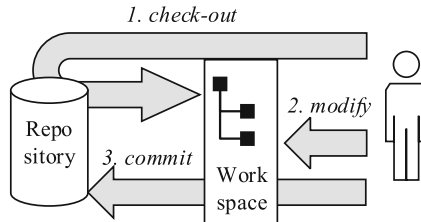


Fig. 2. The iterative three-stage editing model proposed by version control systems [5,6].

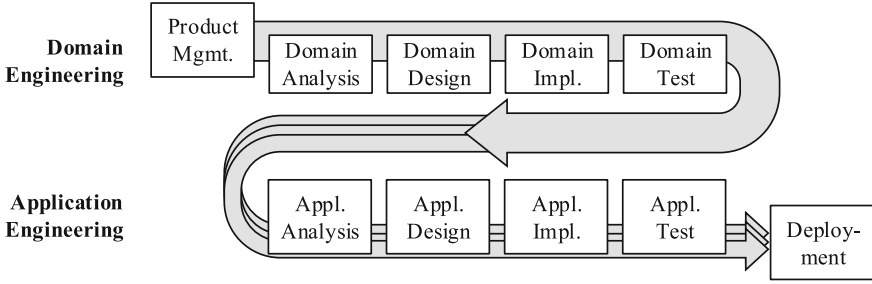


Fig. 3. Detailed phases of the traditional SPLE process as defined in the literature [3].

incremental development of a SPL in a single-version workspace using a filtered editing model that fully automates variability management. In addition to a revision graph, which describes evolution, a feature model and feature configurations are used to express logical variability. In [13], we have presented SuperMod, a model-driven tool that realizes the conceptual framework, allowing to develop a software product line in a single-version workspace step by step using the familiar version control metaphors *update*, *modify*, and *commit*.

Contribution. The current paper explores the development processes underlying existing SPLE tools relying on unfiltered editing on the one hand, and the impact of SuperMod’s filtered editing model on development processes on the other hand. We apply SuperMod to a well-known SPLE example, the *Home Automation System (HAS)* product line [3], illustrating the following key observations:

- Using a filtered editing model, product lines may be developed in an iterative and incremental way, relaxing the strictly sequential order of DE and AE.
- By applying all changes representatively within one product variant, complexity is reduced when compared to multi-variant editing.
- Tool support for DE and AE is integrated, allowing to postpone the decision whether a change is product-specific or in the scope of multiple products until *commit*.
- The adaptation of the VCS-oriented editing model allows to propagate product-specific changes back to the product line.
- SuperMod is flexible with respect to the used SPLE process, ranging between phase-structured and feature-driven domain engineering.

Roadmap. Section 2 is dedicated to SPLE processes. Section 3 sketches the tool SuperMod used to carry out the HAS case study in Sect. 4. Related work is outlined in Sect. 5. Finally, in Sect. 6, open questions are discussed, before the paper is concluded.

2 Software Product Line Development Processes

2.1 The Traditional SPLE Process

The de-facto standard SPLE process has been sketched in the introduction. Figure 3 shows both sub-processes, domain and application engineering, being equally structured by the typical software development activities *analysis*, *design*, *implementation*, and *testing*. Prior to DE stands an additional activity, *product management*, where the scope of the product line is planned, including economical considerations. AE is applied repeatedly for each product; in the traditional SPLE process, it strictly follows DE and re-uses artifacts developed there, i.e., the outcomes of domain analysis, design, implementation, and testing. The sub-process terminates with the deployment of particular products.

The benefits of SPLE are obvious: Rather than developing products from scratch, they may be configured and refined based upon an existing platform. The more products are contained in the product line, the higher the return of investment will be. However, we argue that the traditional SPLE process suffers from a couple of disadvantages:

1. **Necessity of Additional Tools.** To manifest the captured variability in the platform, the toolchain must be extended by mapping tools in the case of negative variability, or composers or transformation languages in the case of positive variability. Obviously, additional tools require additional training effort and imply new sources of error. In the case of MDSE, tools need to be generic with respect to the used modeling language, which immediately leads to undesirable compromises concerning, e.g., the representation of model elements in concrete syntax.
2. **Complexity of the Multi-variant Platform.** Domain engineering requires the developers to keep track of all artifacts of the SPL. This raises complexity particularly concerning the implementation of variation points. Assuming that designing a good architecture is already a challenge for single system development, domain design and implementation become even more complex and error-prone. In many MDSE approaches, multi-variant models are constrained with single-version rules.
3. **Duplicate Maintenance.** Many tools, particularly in the context of MDPLE, aim at fully automated AE by reducing it to a simple configuration step. Frequently, product maintenance causes duplicate maintenance effort. For instance, a bug report may be at first glance specific to a single product, but then become relevant to different members of the product line. Technically, the problem is caused by the automated configuration of products being a one-way road. *Round-trip* support between DE and AE is urgently required.

2.2 Iterative and Incremental Software Product Line Engineering

Iterative SPLE. In analogy to the waterfall model for single-system development, the traditionally applied sequential SPLE process has soon been extended

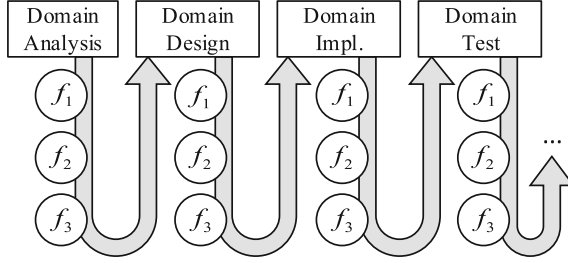


Fig. 4. Phase-structured domain engineering. The identifiers f_i refer to different features and their connected realization artifacts in the platform.

by feedback loops and *iterations*, making SPLE more flexible. Gomaa’s *double spiral* development model [11] allows for alternations between the activities of DE and AE, which are executed in intertwined spirals. Similarly, Clements and Northrop [1] define an iterative SPLE process consisting of three main activities, namely *Core Asset Development*, *Product Development*, and *Management*, which coarsely correspond to DE, AE, and product management, respectively. It is assumed that all three activities are performed in parallel, evolving both the platform and individual products continuously.

Iterative SPLE still assumes that DE is performed in a strictly sequential way as shown in Fig. 4. In the beginning of each iteration, during *domain analysis*, several features are introduced. These features are further designed, implemented, and tested during the subsequent activities. This implies a *phase-structured* domain engineering process, which typically consists of long-running iterations that have to be planned extensively in advance.

Phase-structured SPLE processes allow to maintain an overview of the overall product line, easing architectural decisions necessary to anticipate variation points. For this purpose, *multi-version editing tools* are employed, e.g., preprocessor languages [14] in source-code centric approaches and mapping tools [15, 16] in MDPLE.

Incremental SPLE. *Feature-Oriented Software Development (FOSD)* summarizes a plethora of different techniques and paradigms for the development of variational software in general, and SPL in particular [17]. In the sub-discipline *Stepwise and Incremental Software Development (SISD)* [18], features are described as *refinements* or *layers* of an existing software system and consecutively added to the platform as separate *increments*. The implied *feature-driven* and incremental realization of domain engineering is sketched in Fig. 5 as a counterpart to the phase-structured way. By introducing one feature at a time, this results in comparatively short-running iterations.

In the FOSD context, feature-driven development is preferred over phase-structured approaches. Rather than focusing on multi-variant architectural decisions and explicitly modeling variation points, product changes associated with

a specific features are described in a preferably fine-granular way, e.g., by using composition [19, 20] or aspect-oriented techniques [21].

2.3 SPLE Processes with SuperMod

During the transition from phase-structured to feature-driven SPLE, the performed iterations become smaller. Accordingly, the distinction between domain engineering and application engineering is blurred. The tool SuperMod presented in Sect. 3 provides a filtered editing model, which makes multi-variant artifacts transparent to the SPL engineer by uniformly supporting DE and AE. An increment is performed *representatively* in a particular product variant and then propagated to the platform. Increments correspond to change sets, each referring to a partial feature configuration, which may be developed over multiple iterations. SuperMod is compatible with SPLE processes ranging between phase-structured and feature-driven. The disadvantages of the traditional process listed in Sect. 2.1 are addressed as follows:

1. **Familiar VCS and SPL Metaphors.** SuperMod is added to the toolchain as a new tool, implying the aforementioned difficulties. However, SuperMod's user interface relies on familiar concepts such as version control metaphors (*check-out* and *commit*) and established SPL abstractions (*feature models* and *configurations*).
2. **Filtered Editing.** Changes are generally performed on single-variant products, which eases architectural decisions. Variation points are created automatically and transparently. In the case of MDPLE, the variability of the invisible multi-variant model is unconstrained.
3. **Automatic Propagation of Changes.** After having finished an iteration, the performed changes are propagated to the platform automatically, removing the necessity of duplicate maintenance. In SuperMod, there is technically no distinction between DE and AE. Only at *commit* time, the user must decide whether a change is product-specific or global.

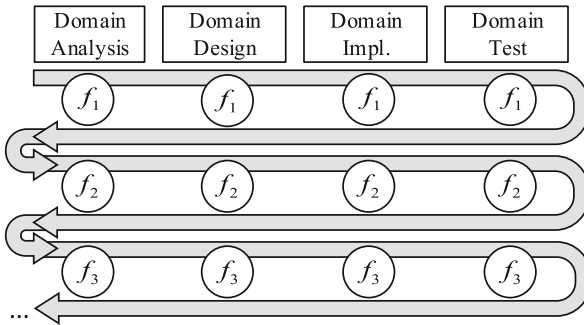


Fig. 5. Feature-driven domain engineering.

3 The Tool SuperMod

This section briefly describes SuperMod [13], a model-driven tool that allows to develop software product lines in an iterative and incremental way as proposed in the previous section. First, we explain theoretical foundations of the tool. Thereafter, SuperMod’s architecture and editing model are sketched and the operations *check-out*, *modify*, and *commit* are redefined. The tool is available for evaluation as Eclipse plug-in (see installation instructions at the end of this paper). Currently, SuperMod is restricted to single-user operation; support for team collaboration is scheduled for future releases.

3.1 Underlying Principles

SuperMod realizes the conceptual framework presented in [12], which integrates MDSE, SPLE, and VC. The framework in turn specializes the *uniform version model* [22], adding higher-level representations for both the version space (i.e., feature models and revision graphs) and the product space (i.e., EMF models). Below, the core concepts of UVM and its extensions are described informally.

- **Options:** An *option* is a temporal or logical property of a software system, which may or may not be included in a specific product version. In SuperMod, two kinds of options exist: *revision options* and *feature options*.
- **Choices:** A *choice* denotes a single valid version by assigning a selection (*selected* or *deselected*) to each of the existing options. Choices are used as *read filters*, i.e., they describe product versions available in the workspace.
- **Ambitions:** An *ambition* denotes a set of versions as a subset of all valid versions. Ambitions are used as *write filters* in order to delineate the scope of a product change performed in the workspace. In contrast to a choice, an ambition may contain *unbound* options, to which the change is immaterial.
- **Version Rules:** The set of available choices and ambitions is constrained by a set of *version rules*, logical expressions over the option set. Version rules are used, e.g., in order to implement constraints such as mutual exclusion imposed by feature models, or to designate subsequent revisions.
- **Visibilities:** A visibility is a logical expression over the option set, which is attached to an element of the feature or domain model. In order to test an element’s presence in a specific version, the bindings specified by the respective choice are applied. Visibilities are modified automatically during the operation *commit*.

3.2 Tool Architecture and Editing Model

Both the architecture and the editing model of SuperMod are inspired by *distributed VCS* [5]. The traditional VCS architecture is extended as follows: Firstly, the feature model is an additional artifact varying along the temporal dimension. Secondly, the domain model varies along two dimensions, the revision graph and the feature model. Figure 6 illustrates the remarks below.

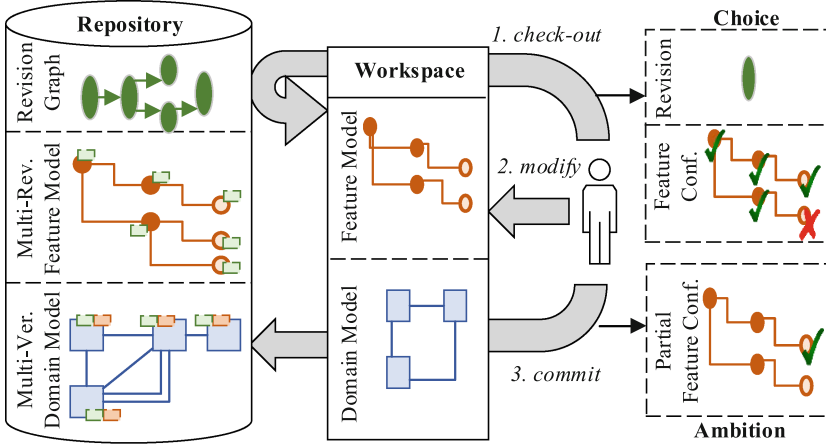


Fig. 6. SuperMod tool architecture and editing model.

Repository. A *repository* is a persistent storage transparently linked to a software project under VC. Developers communicate with it by means of the metaphors *check-out* and *commit*. A SuperMod repository consists of three layers.

- The *revision graph* is a directed acyclic graph that describes the temporal history of a SuperMod project. The graph is extended automatically each time a new revision has been committed. For each revision, a *revision option* is introduced transparently together with a *version rule* that realizes the relationship to the predecessor revision.
- The *multi-version feature model* plays a dual role: Firstly, its evolution is controlled by the revision graph. Secondly, each feature is mapped to a *feature option*, such that the feature model provides an additional version model. Feature model constraints are mapped to *version rules* transparently [12].
- The *multi-version domain model* describes the superimposition of the versioned project. Although the term “domain model” is used here, the project may comprise a file hierarchy containing model or non-model resources. Within the *visibilities* of domain model elements, both revision and feature options may occur.

Workspace. A SuperMod workspace contains the currently selected version of the domain model in its single-version representation. EMF models are represented as instances of their custom Ecore-based metamodel(s). Plain text and XML files are made available in their ordinary format, allowing SuperMod users to utilize their preferred single-version editing tools. During the sub-process *modify*, they may also edit the feature model, e.g., by introducing new features or relationships.

Version Specification. A version in the temporal dimension corresponds to a single revision. As mentioned above, *feature configurations* specify choices and ambitions in the logical dimension. When referring to an iterative and incremental development process (cf. Sect. 2.2), version specification happens in the beginning and at the end of each iteration. A feature configuration is specified on the current revision of the feature model. When provided as an ambition, the feature configuration may be *partial*¹ and typically binds only few features, in many cases only one feature. The *effective choice/ambition* is formed during check-out/commit as conjunction of the temporal and logical component.

3.3 Check-Out, Modify, and Commit

In the following, the operations *update*, *modify*, and *commit* known from VCS are redefined on top of SuperMod's architecture and editing model (cf. Fig. 6).

Check-Out. Like in ordinary VCS, the operation *check-out* is provided to select a specific version (the *choice*) from the repository, which is then copied to the workspace:

- The user selects a *revision* as the temporal component of the choice. The *feature model* is filtered by the revision, and made available for modification in the workspace.
- The user specifies a completely bound feature configuration, which forms the logical component of the choice. The *effective choice* is recorded persistently.
- The domain model is filtered by the effective choice and *exported* into the local workspace. The *export* transformation translates multi-version resources into their specific single-version representation, e.g., plain text or XMI files.
- The filtered and exported contents are made available in the workspace.

Modify. The user may *modify* both the filtered feature model and the filtered domain model within the workspace. For domain model resources, arbitrary editors may be used. For the feature model, the command *Edit Version Space* is offered, which delegates to a specific model editor for the current feature model revision.

Commit. The operation *commit*, the counterpart to *check-out*, propagates changes performed in the workspace to the repository under a user-specified scope (the *ambition*):

¹ Our notion of partial feature configuration only implies that there exist unbound features. This differs from the concept of *staged configurations* (as introduced, e.g., in [4]), which need to be specified in a top-down way, introducing parent-child selection constraints.

- A new *revision* is created as the successor of the revision specified for the choice and selected in the temporal component of the ambition. Within the given revision of the feature model, the logical component of the ambition is user-specified as a *partial feature configuration*. For consistency, it is required that the set of versions described by the effective ambition include the recorded choice.
- The original state of the workspace version is temporarily restored by applying the recorded choice to the repository. The new state is generated by *importing* (the inverse of *export*) the current workspace into its multi-version representation.
- *Differences* are computed between the original and the new workspace state.
- Inserted elements are copied into the repository.
- The *visibilities* of inserted/deleted feature model elements are updated automatically by adding/subtracting the temporal component of the ambition to/from the existing visibility.
- In analogy, the *visibilities* of inserted/deleted domain model elements are updated by adding/subtracting the *effective ambition*.

4 The Home Automation Case Study

We apply the tool SuperMod presented in Sect. 3 to the standard example of a product line for Home Automation Systems from [3]. The example is divided up into a phase-structured and a feature-driven part. First, the activities *analysis* (Sect. 4.1), *design* (Sect. 4.2), and *implementation* (Sect. 4.3) are executed, realizing an initial DE iteration. During implementation, a command-line application is developed based on the generated source code. Due to space restrictions, the activity *testing* has been omitted. In the second part, we transition into feature-driven DE, extending the product line by a new feature ensuing from a customer request (Sect. 4.4). Last, we present our observations and refer back to the SPLE processes from Sect. 2.

For analysis and design, we rely on UML *use case*, *activity*, *package*, and *class diagrams* [8], using the GMF²-based UML modeling tool *Valkyrie* [23] and its Java code generator. The remarks below are illustrated by screencasts available on our web pages; please follow the link provided at the end of this paper.

4.1 Requirements Analysis

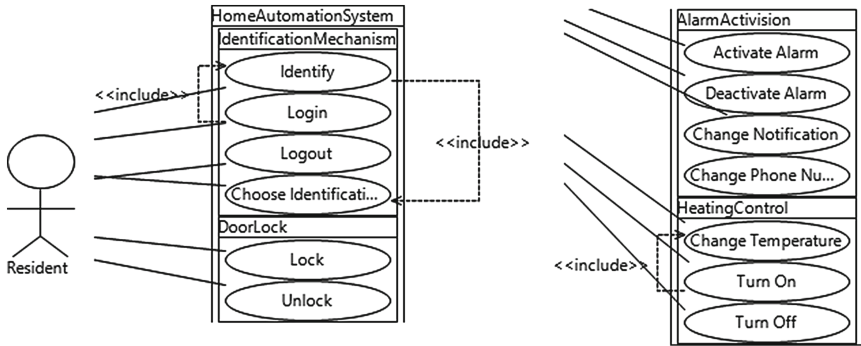
Requirements analysis is split into two phases. To begin with, residents' interactions with the HAS are documented in a use case diagram. Subsequently, one use case is representatively refined by means of an activity diagram.

After having initialized a Valkyrie project and having connected it to SuperMod version control, the first phase is started with an empty use case diagram. In consecutive iterations, we add actors, components, use cases, and relationships as summarized in Table 1. The table also shows that the feature model

² *Graphical Modeling Framework*, <http://www.eclipse.org/modeling/gmp/>.

Table 1. Commit history of the use case diagram.

Rev.	Ambition	Changes to feature model	Changes to use case diagram
1	H.A.S.	added feature H.A.S.	added actor Resident and component H.A.S.
2	Id.Mech.	added feature Id.Mech.	added component Id.Mech., contained use cases, includes, and connected use links
3	DoorLock	added feature DoorLock	added comp. DoorL., use cases Lock and Unlock
4	DoorLock	—	added missing use links for Lock and Unlock
5	AlarmAct.	added feature AlarmActivision	added component AlarmActivision, contained use cases, and connected use links
6	SMSTo-Owner	added features Ac.Sig., Vid.S., PoliceInf., and SMSToOwner	added use case Change Phone Number
7	Heat.Cont.	added feature Heat.Cont.	added component Heat.Cont. and contents

**Fig. 7.** The use case diagram of the HAS example after revision 7, shown in a variant that includes all mandatory and optional features available.

is developed simultaneously, introducing new features on demand in order to delineate the scope of the respective changes. Figure 7 shows a variant of the final use case diagram.

During the second analysis phase, the feature **IdentificationMechanism** is further refined by adding three concrete mechanisms, namely **Keypad**, **Magnetic-Card**, and **FingerprintScanner**. These are collected in an OR-group, meaning that at least one mechanism must be chosen in a valid configuration. In case several mechanisms are available, one of them must be chosen during identification. The

Table 2. Commit history of the activity diagram for **Identify**.

Rev.	Ambition	Changes to feature model	Changes to activity diagram
8	H.A.S.	added XOR groups below DoorL. and HeatingCont.	—
9	Id.Mech.	—	initialized diagram, added initial and final nodes, Choose Mech., decision/merge nodes, and flows
10	Keypad	added OR group with Keypad, Mag.Card, Fp.Scanner	added action KeypadIdentification and incoming/outgoing flow
11	Mag.Card	—	added action M.C.Id. and incoming/outgoing flow
12	Fp.Scan.	—	added action Fp.Id. and incoming/outgoing flow

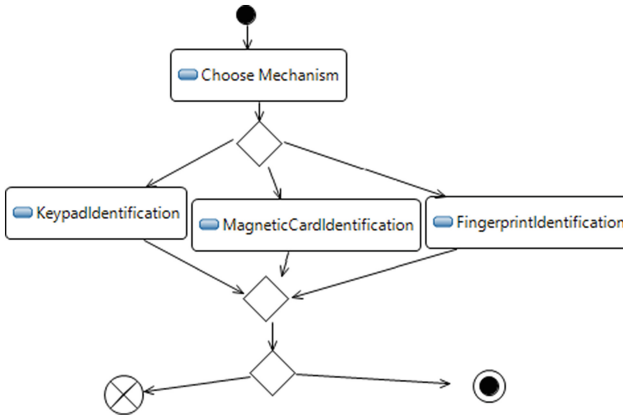


Fig. 8. The activity diagram of the use case **Identify** after revision 12, shown in a variant that includes all sub-features of **IdentificationMechanism**.

available selection should be restricted by the active features; this is realized in revisions 10 until 12 shown in Table 2. The resulting activity diagram is shown in Fig. 8; Fig. 9 shows the refined feature model.

4.2 Design

The static structure of the HAS product line is also developed in two phases. After modeling an initial package diagram, specific packages are refined by class diagrams.

Table 3 indicates that the package diagram (see Fig. 10) is developed in an iterative and incremental way by realizing one feature after another. Variation

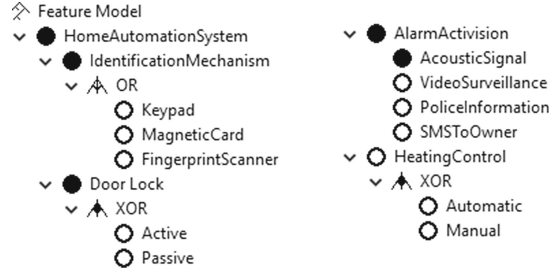


Fig. 9. The feature model after revision 12, shown in SuperMod’s feature model editor. Filled circled denote mandatory features, empty circles optional child features. OR groups require the selection of at least one, XOR groups of exactly one child feature.

Table 3. Commit history of the package diagram.

Rev.	Ambition	Changes to package diagram
13	HomeAutomationS.	added package has and contained class HomeAutomationSystem
14	Ident.Mechanism	added package identification, class Id.Mech., and interface IMechanism
15	Keypad	added class Keypad
16	MagneticCard	added class MagneticCard
17	FingerprintScanner	added class FingerprintScanner
18	DoorLock	added package doorLock and interface IDoorLock
19	Active	added class ActiveLock
20	Passive	added class PassiveLock
21	AlarmActivision	added package alarm, class AlarmAct., and interface IAlarmService
22	AcousticSignal	added class AcousticSignal
23	VideoSurveillance	added class VideoSurveillance
24	PoliceInformation	added class PoliceInformation
25	SMSToOwner	added class SMSNotifier
26	HeatingControl	added package heating and contained interface IHeatingControl
27	Automatic	added class heating::Automatic
28	Manual	added class heating::Manual

points are anticipated by sketching the use of appropriate *design patterns* such as *strategy* and *command* [24], which are subsequently refined by class diagrams. Here, we refrain from introducing new features during the design phase, although permitted in general.

As shown in Table 4, the package identification is refined by a class diagram,

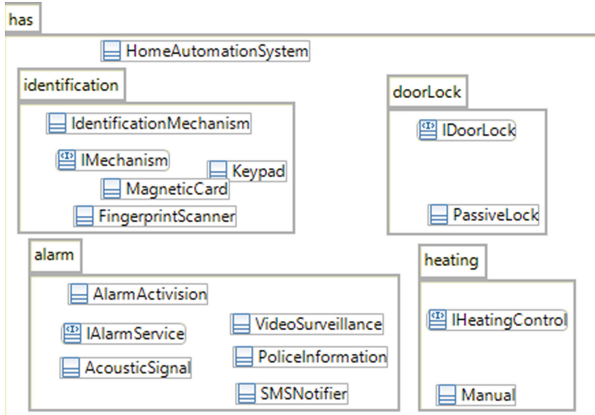


Fig. 10. The package diagram after revision 28. The shown product variant does not include features *Active* and *Automatic*, thus not classes `doorLock::ActiveLock` and `heating::Automatic`, either.

exemplifying the realization of variation points during design. In revision 29, general details are added to the class `IdentificationMechanism` as well as to the interface `IMechanism` that realizes the *command* pattern. Its specific realizations are added subsequently and scoped with the respective feature. In this example, the only necessary changes are to make the respective command classes realize `IMechanism` (see Fig. 11). In fact, more details could have been added to the classes here. Furthermore, similar refinements might have been applied to the packages `doorLock`, `alarm`, and `heating`.

4.3 Implementation

In our model-driven product line, the static part of the source code can be derived from the artifacts developed in the *design* phase using *Valkyrie*’s code generator. The main class `HomeAutomationSystem` shall contain the main executable as command-line application. Below, we confine the presentation to the

Table 4. Commit history of the class diagram refining package identification.

Rev.	Ambition	Changes to class diagram for package identification
29	Ident.Mechanism	initialized diagram, detailed class <code>Ident.Mech.</code> and interface <code>IMech.</code>
30	Keypad	added interface realization originating from class <code>Keypad</code>
31	MagneticCard	added interface realization originating from class <code>MagneticCard</code>
32	FingerprintScanner	added interface realization originating from class <code>FingerprintScanner</code>

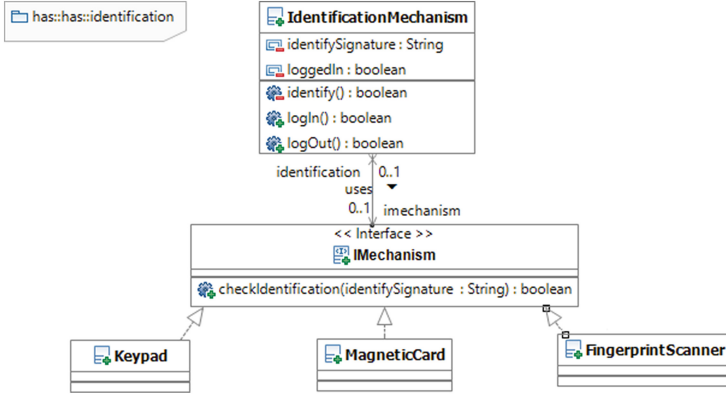


Fig. 11. The class diagram that refines package `identification` in its state after revision 32, with features `Keypad`, `MagneticCard`, and `FingerprintScanner` selected.

implementation of the method `identify()` of class `IdentificationMechanism`, which implements the activity diagram from Fig. 8.

Table 5. Overall commit history of the *implementation* phase.

Rev.	Ambition	Changes to <code>IdentificationMechanism.java</code> or other source files
33	<code>HomeAutomationS.</code>	generated Java source code
34	<code>Ident.Mechanism</code>	added multi-variant implementation to method <code>identify()</code> (l. 96 – 101)
35	not <code>Fp.Scanner</code>	removed <code>FingerprintScanner.java</code> and line 99
36	not <code>MagneticCard</code>	removed <code>MagneticCard.java</code> and line 98
37	not <code>Keypad</code>	removed <code>MagneticCard.java</code> and line 97

Variability is achieved by making the declarations and usages of specific mechanism classes dependent on their respective features. As shown in Table 5 and Listing 1.1, after the initial code generation run in revision 33, *negative variability* is simulated: In revision 34, a multi-variant implementation is provided. We then connect the variable constructor calls and the concrete implementation classes to their respective features by applying the *negative implementation*, i.e., by removing the corresponding source code file and the statement containing the constructor call, and by committing against the negation of the respective ambition³. In order to perform these deletions, it is necessary to switch to a suitable choice where the respective features are deselected, e.g., the choices presented in the screencast.

³ Equivalently, we could have applied the positive realization and committed it against positively bound features; however, this would have required three additional code generation increments.

```

95  private void identify() {
96      List<IMechanism> mechs = new LinkedList<>();
97      mechs.add(new Keypad());
98      mechs.add(new MagneticCard());
99      mechs.add(new FingerprintScanner());
100     IMechanism mech = (...) // choose interactively
101     return mech.checkIdentification(getIdentifySignature());
102 }

```

Listing 1.1. Implementation of the method `IdentificationMechanism.identify()` in revision 34.

4.4 Handling a New Customer Request

So far, our SPL has been developed in a phase-structured way, following the classical development activities *analysis*, *design*, and *implementation*. Now, we demonstrate how SuperMod allows to quickly react to a new customer request that cross-cuts all three development activities; we realize the increment in one single iteration.

The customer requests to extend the list of identification mechanisms available in the HAS product line by a new, *biometric* mechanism that uses existing iris scanner hardware and drivers. We check-out the latest revision of the HAS project, choosing the customer’s product variant, which currently includes all sub-features of `IdentificationMechanism`. Then, we handle the request as follows (due to space restrictions, we cannot present the modified artifacts here; please refer to the screencasts):

- **Analysis:** It is obvious that a new feature `Biometric` must be introduced into the OR-group below `IdentificationMechanism` (cf. Fig. 9). The request does not affect the use cases, but the activity diagram that details the use case `Identify` (cf. Fig. 8): We add a new action `BiometricIdentification` and connect it to the decision/merge node in analogy to the existing identification actions.
- **Design:** We add a new class `Biometric` as well as a realization of the interface `IMechanism` to the class diagram shown in Fig. 11. This transparently extends the package diagram (cf. Fig. 10).
- **Implementation:** The (incremental) code generation is re-invoked, creating a new source file `Biometric.java`. To the implementation of method `IdentificationMechanism.identify()` (cf. Listing 1.1), we add the following statement after line 99:


```
mechs.add(new Biometric());
```
- **Deployment:** The current iteration is finalized by committing all pending changes to the repository under revision 38. As logical ambition, we specify a partial configuration that selects only the new feature `Biometric`. Hence, the performed modifications hold for future variants that include this feature. At last, the current product variant is deployed to the customer, without the need for an additional AE run.

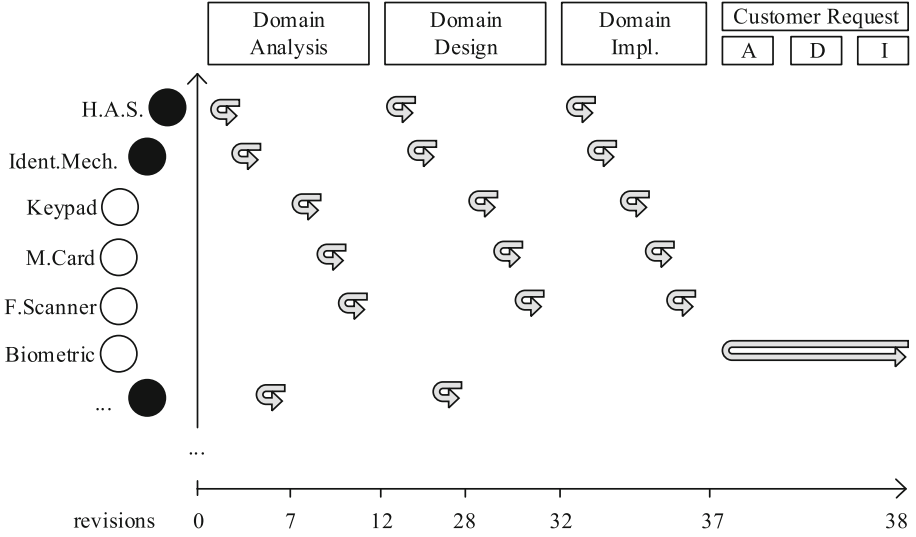


Fig. 12. Summary of the HAS example: Iterations and increments performed during specific development activities, aligned with the temporal (x-axis) and logical dimension (y-axis).

4.5 Results and Observations

Key Figures. Our example SPL has evolved over a total of 38 iterations, distributing as follows: 12 iterations for analysis, 20 for design, 5 for the implementation of a cut-out of the functionality, and one additional iteration for the new customer request. In total, the product line contains approximately 100 model elements, from which 17 source code files have been derived, the largest of which contains 137 lines of code. The final feature model contains 17 features, 10 of which are optional. The entire version management has been performed by specifying 6 choices (cf. screencasts) and 38 ambitions, respectively, during *check-out* and *commit*.

According to the mechanisms described in [12, 13], 410 *visibilities* have been added to elements, attributes, and links contained in the transparent multi-variant UML model (not including its graphical representation). Necessarily, the same number of *feature expressions* or *presence conditions* would have to be manually specified when using an explicit mapping model in a tool relying on positive variability, e.g., [15] or [16].

Remarks on SPLE Process and Tools. Figure 12 summarizes the relevant cut-out of the example. When considering DE as a whole, one could sum up revisions 1 until 37 as one process iteration, including the DE activities *analysis*, *design*, and *implementation*. Technically, a multitude of iterations have been performed using fine-grained *check-out/commit* cycles in order benefit from

SuperMod’s automated variability management and filtered editing model. Noteworthy, the iterations belonging to each phase are arranged roughly diagonally when referring to the temporal and logical dimension.

Revision 37 may be considered as an initial major revision of the product line, after which we transition from phase-structured to feature-driven development in order to integrate *customer feedback* more flexibly. The change performed in revision 38 is realized in the customer’s product variant and then added to the product line transparently by committing the change against the new feature Biometric. This way, *duplicate maintenance* is *avoided* by the tool-level integration of DE and AE.

In sum, the example has shown that SuperMod is compatible with both a phase-structured and a feature-driven style of iterative and incremental SPLE development. In addition to a *reduced version management overhead*, the example has demonstrated a *minimal planning effort* when referring to particular iterations; features are introduced on demand. The advantages of *tool independence* and *unconstrained variability* discussed in [12] can also be reproduced in the HAS example.

5 Related Work

This paper continues a series of previous publications on the SuperMod project and its foundations. In [12], the underlying conceptual framework for the integration of VC, SPLE and MDSE has been defined. The paper also contains a general overview of literature concerning the integrating disciplines *Model-Driven Product Line Engineering* [11], *Model Version Control* [10], and *Software Product Line Evolution* [25]. In [13], the tool SuperMod has been presented using the standard example of a product line for *graphs* [26]. Additionally, the paper contains a comparative domain analysis of VC and SPLE and aligns SuperMod with different tools that share VC and SPLE concepts. In this section, we compare our work to different iterative and/or incremental approaches to SPLE and to other occurrences of the HAS case study.

The Home Automation System example has been introduced by Pohl et al. [3] to illustrate different activities of the traditional SPLE process (cf. Fig. 3). The authors stress the importance of the activity *domain analysis*, where an initial feature model is produced. During *domain design* and *domain implementation*, a reference architecture and core implementation assets are constructed. In *domain testing*, component tests are written. These artifacts are then filtered and composed during corresponding *application engineering* activities, concluding with testing the product using respective component tests. As opposed to our version of the HAS example, the original version has been developed in a strictly phase-structured way.

Among others, the authors of [27] have observed that agile principles potentially increase the applicability of SPLE while reducing time to market. They present a bottom-up, test-driven approach inspired by *Extreme Programming* [28]. After defining a test case specific to a new feature, its realization is incorporated to the product line using systematic refactoring techniques. However,

the presented solution is not as highly automated as the SuperMod approach. Furthermore, when compared to our example, the iterations are still relatively long-running. Presumably, SuperMod can also meet the requirements of agile SPLE.

In [29], an approach to filtered (*projectional*) editing of multi-variant programs is described. Like in our work, the motivation is a reduction of complexity gained by hiding variants not important for a specific change to a multi-variant model. Visibilities are managed automatically, but in contrast to our approach, the *choice* always equals the *ambition*. Furthermore, the restriction of a *completely bound choice* does not exist since the user operates on a *partially filtered* product which still contains variability. The wider the ambition, the more variability information is kept in the workspace, increasing maintenance overhead especially for wide ambitions.

Völter et al. [21] apply *aspect-oriented* techniques such as modularization and composition in order to realize the HAS example using positive variability. The platform is described at a high level of abstraction using a custom domain-specific language. During product derivation, artifacts belonging to the selected features are composed. This leads to a reduction of complexity with respect to architectural decisions of the modular artifacts, but raises new problems when it comes to conflicting composition rules.

In [15, 30], the HAS example has been used to demonstrate consistency mechanisms of the MDPLE tool FAMILIE, which relies on negative variability and unfiltered editing. The tool allows to connect a manually developed multi-variant domain model to a feature model in a dedicated *mapping model* using *feature expressions*, and to automatically configure products. Contradictions among feature expressions may lead to inconsistencies within the mapping model. The presented solutions, which include a domain specific language for repair actions, are interactively controlled by the SPL engineer. In contrast, SuperMod makes both the multi-variant model and feature expressions transparent, and product conflicts are resolved by the user in batch mode.

The HAS example is frequently referred to in the context of dynamically reconfiguring systems, which can be considered as product lines that use run-time variability. An example is provided in [31], where the platform itself is described using MDSE techniques. When compared to our compile-time based solution, time to market is even shorter. However, consistent component interaction must be manually ensured.

6 Discussion

Having conducted two standard examples with the tool SuperMod and having defined an appropriate development process, we are now able to discuss the potential research impact as well as the limitations of our proposed approach. The following open questions will also stimulate future research directions.

How Steep is SuperMod’s Learning Curve? In the beginning, SuperMod’s editing model seems quite unfamiliar, in particular to SPL engineers who are used to unfiltered editing approaches, where they fully control the multi-variant architecture. According to our own experience, planning the iterations and learning to specify a correct ambition are the most challenging parts. We have observed that small iterations and frequent commits require a fair amount of discipline. The effective training effort of SuperMod remains to be experimentally quantified and compared to unfiltered SPLE approaches.

Do We Still Require Unfiltered Editing? Intentionally, SuperMod users never get in touch with multi-version artifacts, since they always operate in a single-version view. This reduces complexity, but also awareness of the variability present in the overall product line. In some situations, one wants to inspect or modify the multi-version artifacts in an unfiltered way, e.g., in order to revise erroneously specified ambitions. A compromise between filtered and unfiltered editing is *partially filtered editing* [29]. However, preprocessor-like variability annotations are technically hard to realize for graphically represented models.

Where are Organized Reuse and Variation Points? SPL are based on the principle of *organized reuse*. When developing the multi-variant architecture, variation points are planned in advance and explicitly realized using the features of the respective programming or modeling language, e.g., inheritance. SuperMod does not require to explicitly model and document variation points; on the contrary, they are completely transparent to the user. This fact is in turn linked to the advantage of reduced complexity and the disadvantage of limited awareness of variability [12]. Furthermore, in SPLE, features are typically introduced in the beginning during *product management*. In contrast, our approach dedicates the decision, when to introduce new features, to the user.

How to Control the Multi-variant Architecture? This question is linked to the preceding two answers. Due to the single-version view and the fact that variation points are transparent, the challenge of designing a multi-variant architecture never arises. However, this also removes the chance to control the architecture, e.g., by refactoring. Does this result in a “worse” multi-variant architecture? Provided that it is transparent to the user anyway, is a “good” multi-variant architecture important at all? The properties of automatically constructed multi-variant architectures need to be further investigated.

7 Summary and Outlook

In this paper, we have revisited a well-established SPLE case study, the Home Automation System SPL. We have employed the tool SuperMod, which is focused on but not restricted to model-driven SPL. Its user interface is oriented towards version control by offering the metaphors *update*, *modify*, and

commit. Developers may evolve the SPL in a single-version workspace, while changes are propagated to the multi-version platform transparently, obviating the need for up-front, multi-variant design. The evolution of multi-variant product artifacts is mostly automated. SuperMod’s integrated tool support enables a round-trip between DE and AE, whose distinction is blurred by fine-granular *update/commit* cycles and by keeping products in the product line until deployment.

With respect to the underlying SPLE process, our example has demonstrated that SuperMod is compatible with different styles of iterative and incremental development, ranging from phase-driven domain engineering, which has been applied to create an initial major revision of the product line, to feature-driven development, which has been enforced to integrate customer feedback and to integrate the respective change to the product line transparently, without the need of duplicate maintenance.

When compared to state-of-the-art approaches, our presented solution minimizes both planning and maintenance effort. Furthermore, the amount of manually specified variability information is significantly lower. Using the presented procedure and tool, the SPLE developer may focus on product-specific design decisions, reducing the cognitive complexity in the domain engineering phase. SuperMod integrates well with existing tools, particularly in the EMF world.

Future work addresses extensions to SuperMod, including multi-user support, product conflict resolution, and difference representation. Furthermore, we aim to continue the experimental evaluation of our approach using a case study of industrial scale.

8 Accompanying Resources

The research prototype *SuperMod* is available as a set of Eclipse plug-ins under the Eclipse Public License. The plug-ins may be installed into a clean Eclipse Luna Modeling distribution using the following update site:⁴. The items *SuperMod Core* and *SuperMod Revision+Feature Layered Version Model* should be selected for installation. Furthermore, we provide several screencasts where SuperMod’s usage with both the *graph* example from [13] and the *HAS* example from this paper is demonstrated:⁵.

Acknowledgements. The authors give thanks to Marco Dmitrow for adapting the HAS case study in a master project and for valuable input for the improvement of SuperMod.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston (2001)

⁴ <http://btn1x4.inf.uni-bayreuth.de/supermod/update>.

⁵ <http://btn1x4.inf.uni-bayreuth.de/supermod/screencasts>.

2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute (1990)
3. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations. Principles and Techniques, Berlin (2005)
4. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: a progress report. In: International Workshop on Software Factories at OOPSLA 2005, San Diego, California, USA. ACM (2005)
5. Chacon, S.: Pro Git, 1st edn. Apress, Berkely (2009)
6. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly, Sebastopol (2004)
7. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering Management. Wiley, New York (2006)
8. OMG: UML Superstructure. Object Management Group, Needham, MA. formal/2011-08-06th edn. (2011)
9. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Upper Saddle River (2009)
10. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. *Int. J. Web Inf. Syst. (IJWIS)* **5**, 271–304 (2009)
11. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Boston (2004)
12. Schwägerl, F., Buchmann, T., Uhrig, S., Westfechtel, B.: Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J. (eds.) *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*, Angers, France, pp. 5–18. SCITEPRESS (2015)
13. Schwägerl, F., Buchmann, T., Westfechtel, B.: SuperMod - A model-driven tool that combines version control and software product line engineering. In: *ICSOFT-PT 2015 - Proceedings of the 10th International Conference on Software Paradigm Trends*, Colmar, Alsace, France, pp.5–18. SCITEPRESS (2015)
14. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. In: *Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering (ViSPL)*, pp. 303–313 (2008)
15. Buchmann, T., Schwägerl, F.: FAMILÉ: tool support for evolving model-driven product lines. In: Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., Tolvanen, J.P. (eds.) *Joint Proceedings of Co-Located Events at the 8th European Conference on Modelling Foundations and Applications*. CEUR WS, Building 321, DK-2800 Kongens Lyngby, pp.59–62. Technical University of Denmark (DTU) (2012)
16. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to Models. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pp. 943–944. ACM, New York (2008)
17. Apel, S., Kästner, C.: An overview of feature-oriented software development. *J. Object Technol.* **8**, 49–84 (2009)
18. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. In: *Proceedings of the 25th International Conference on Software Engineering, ICSE 2003*, pp. 187–197. IEEE Computer Society, Washington, DC (2003)

19. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model composition in product lines and feature interaction detection using critical pair analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
20. Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., Kulesza, U.: VML* – A family of languages for variability management in software product lines. In: van den Brand, M., Gašević, D., Gray, J. (eds.) *SLE 2009*. LNCS, vol. 5969, pp. 82–102. Springer, Heidelberg (2010)
21. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: *Proceedings of the 11th International Software Product Line Conference, SPLC 2007*, pp. 233–242. IEEE Computer Society, Washington, DC (2007)
22. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. *IEEE Trans. Softw. Eng.* **27**, 1111–1133 (2001)
23. Buchmann, T.: Valkyrie: A UML-based model-driven environment for model-driven software engineering. In: Hammoudi, S., van Sinderen, M., Cordeiro, J. (eds.) *Proceedings of the 7th International Conference on Software Paradigm Trends (ICSOFT 2012)*, pp.147–157. SCITEPRESS (2012)
24. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st edn. Addison-Wesley Longman, Amsterdam (1995)
25. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.* **78**, 1010–1034 (2013)
26. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Dannenberg, R.B. (ed.) *GCSE 2001*. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)
27. Ghanam, Y., Maurer, F.: Extreme product line engineering – refactoring for variability: a test-driven approach. In: Sillitti, A., Martin, A., Wang, X., Whitworth, E. (eds.) *XP 2010*. LNBIP, vol. 48, pp. 43–57. Springer, Heidelberg (2010)
28. Beck, K., Andres, C.: *Extreme Programming Explained: Embrace Change*, 2nd edn. Addison-Wesley Professional, Reading (2004)
29. Walkingshaw, E., Ostermann, K.: Projectional editing of variational software. In: *Generative Programming: Concepts and Experiences, GPCE 2014*, Vasteras, Sweden, 15–16 September 2014, pp. 29–38 (2014)
30. Buchmann, T., Schwägerl, F.: Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development, FOSD 2012*, pp. 37–44. ACM, New York (2012)
31. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer* **42**, 37–43 (2009)

Software Technologies

10th International Joint Conference, ICSOFT 2015,
Colmar, France, July 20-22, 2015, Revised Selected
Papers

Lorenz, P.; Cardoso, J.; Maciaszek, L.A.; van Sinderen, M.
(Eds.)

2016, XV, 431 p. 174 illus. in color., Softcover

ISBN: 978-3-319-30141-9