What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines

Thorsten Berger[1], Daniela Lettner[2], Julia Rubin[3], Paul Grünbacher[2], Adeline Silva[4], Martin Becker[4], Marsha Chechik[5], Krzysztof Czarnecki[1]
[1]University of Waterloo, [2]Johannes Kepler University Linz, CD Lab MEVSS, [3]Massachusetts Institute of Technology, [4]Fraunhofer IESE, [5]University of Toronto

ABSTRACT

The notion of *features* is commonly used to describe the functional and non-functional characteristics of a system. In software product line engineering, features often become the prime entities of software reuse and are used to distinguish the individual products of a product line. Properly decomposing a product line into features, and correctly using features in all engineering phases, is core to the immediate and long-term success of such a system. Yet, although more than ten different definitions of the term *feature* exist, it is still a very abstract concept. Definitions lack concrete guidelines on how to use the notion of features in practice.

To address this gap, we present a qualitative empirical study on actual feature usage in industry. Our study covers three large companies and an in-depth, contextualized analysis of 23 features, perceived by the interviewees as typical, atypical (outlier), good, or bad representatives of features. Using structured interviews, we investigated the rationales that lead to a feature's perception, and identified and analyzed core characteristics (facets) of these features. Among others, we found that good features precisely describe customer-relevant functionality, while bad features primarily arise from rashly executed processes. Outlier features, serving unusual purposes, are necessary, but do not require the full engineering process of typical features.

1. INTRODUCTION

Software Product Line Engineering (SPLE) approaches rely on identifying and explicitly managing commonalities and variabilities of a product portfolio. These commonalities and variabilities are often captured in an abstract manner using entities called *features*. The use of features is motivated by the fact that customers and engineers often speak of product characteristics in terms of features a product has or delivers. A feature is usually defined as "a logical unit of behavior specified by a set of functional and non-functional requirements" [7] or "a distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder of the concept" [9]. In fact, many additional definitions of the term *feature* can be found in the literature [1, 17, 32, 15, 25, 18, 23, 20, 8, 31].

Yet, companies still face difficulties in deciding when to introduce a feature, determining the right level of granularity for a feature, and defining the aspects that should be taken into consideration when engineering features. Without this knowledge, using SPLE concepts and the numerous existing tools for managing product line features is problematic. In fact, all authors of this paper—when presenting feature-related engineering or analysis techniques—are commonly faced with the question: "What is a feature?"

In this paper, we aim to address this issue by empirically investigating the experiences of three successful industrial companies that develop software product lines (SPLs) and explicitly manage features. We conducted a qualitative study to elicit, understand, and describe features managed by the companies. We also describe the companies' perspective on their successes and failures in managing features.

Our main goal is to improve the empirical understanding of the notion of features in industry, by providing insights into the range of real-world feature definitions and usages. We rely on semi-structured interviews, whose design and analysis was guided by two main research questions:

RQ1: What reasons cause companies to perceive a feature as typical, atypical, good or bad? We studied concrete examples of features by asking our interviewees for typical, atypical (outlier), good, and bad exemplars, and by diving into the reasons for such classification. Our intention was to be as open as possible, trying to disambiguate existing perceptions of features among our interviewees.

RQ2: What are important characteristics of features? When discussing each feature, we asked the interviewees to describe its different *facets*: intrinsic qualities of a feature, such as its purpose within the software lifecycle or its binding time. Using feature facets as the basic terminology allowed us to structure the discussion, to compare the features across companies, and to organize our findings.

We present first-hand opinions of industrial practitioners on practices contributing to the development of features that are perceived as typical, successful or failing. In addition to narrative descriptions of features and their classification rationales, we provide an in-depth cross-case analysis of all the features. In summary, we contribute: (i) a set of *facets* that can be used as a terminology for describing and comparing features (Table 2); (ii) reasons (rationales) for

SPLC 2015, July 20 - 24, 2015, Nashville, TN, USA

ISBN 978-1-4503-3613-0/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2791060.2791108

Table 1: Interview participants

	part.[1]	role	exp.[2]	features
Keba	A	developer	12	LINMovement, Oscilloscope, Euromap, SilentMode
	B	product manager	19	ProfiNetSlave, Wizard, ManualConfiguration, UserGuidance
	C	developer	3	LanguageTranslation, ProductionOverview, DataManager, HeatUpOptimization
Opel	D	team lead/ architect	5	LaneKeeping, ParkAssist, EmergencyBraking
Danfoss	E	architect	4	Torque, CascadeController, ProductG, PowerUpFastFuncs
	F	team lead	8.5	Wobbler, FieldBus, ResetFix, BoardSupportPackage

[1] participant (interviewee) [2] experience with the product line in years

a feature being classified as typical, outlier, good, or bad (Table 4); (iii) a range of values for different facets of concrete features engineered in industry (Sec. 6); and (iv) a set of core observations from the cross-case analysis that have practical impact on engineering features for SPLs.

We proceed by outlining our research methodology in Sec. 2. We then introduce the subject companies and their product lines in Sec. 3. We address RQ1 in Sec. 4 and 5, where we present features and their classification rationales. In Sec. 6, we address RQ2 with the cross-case analysis of the feature facets. Finally, we discuss threats to validity in Sec. 7 and related work in Sec. 8, and conclude in Sec. 9.

2. RESEARCH METHODOLOGY

We describe how we selected the companies for our study and present our approach to designing and conducting the interviews and analyzing the results.

2.1 Company Selection

To conduct our study, we focused on companies that (i) develop SPLs; (ii) explicitly record, track, and manage features—both common and variable ones; and (iii) maintain an active collaboration with at least one of the authors of this paper. Our selection criteria ensured that we consider "meaningful" examples that are of general interest to the SPL community, and that allowed us to reliably interpret the findings based on our understanding of the companies' product lines and their organizational context.

We selected the three companies Keba, Opel, and Danfoss from the domains of industrial automation, automotive, and power electronics. For each company, we selected up to three interviewees, covering a range of roles, such as product manager, architect, and developer. For Opel, we interviewed a single person. Overall, we collected data about 23 features: twelve from Keba, three from Opel, and eight from Danfoss. Table 1 summarizes our interviewees, their roles, their experience with the product line, and the features they described.

2.2 Interview Design

Feature Facets. The goal of our study was to collect examples of features developed in industrial practice, and to outline the reasons for specific features being considered good, bad, typical, or outliers. To gain insights into these questions, we conducted a set of semi-structured interviews with employees of the studied companies.

We structured our interviews around *feature facets*—qualities of features that we aimed at exploring, such as lifecycle purpose and binding time. These facets were initially defined by consulting the existing literature [19, 7, 8, 10], and then further refined based on our previous collaborations with the studied companies. More specifically, three of the authors created within-case writeups of the general architecture and the organizational structure of each company, which were then used to guide the definition of the facets. The resulting set of facets, along with their description and clarifying examples, are given in Table 2.

Interview Process. We started the interview with questions about our interviewees themselves, including (i) their professional background, (ii) how long they have been working in the current profession, (iii) how long they have been involved with the product line, (iv) their role in the product line, and (v) the number of features they were involved with.

We then asked each interviewee to describe three to four features, providing guidance in the selection process. We asked for one *typical*, one *outlier*, one *good*, and one *bad* feature. Our goal was to be as open as possible, leaving it to our interviewees' judgment which criteria to use for selecting a feature for each type. Yet, to provide some guidance, we gave some hints, for instance, that a good feature could be one that is well-received and popular with customers, commercially successful, or on time, on budget, easily reusable, or has a low defect count. For a bad feature, we said it could be one that is problematic, troublesome, difficult to develop, confusing, buggy, or one that destroyed user confidence, damaged the brand, or showed unexpected behavior. An outlier is a feature whose properties are rarely observed in other features. Finally, a typical feature is neither especially good nor bad, and not an outlier in any sense.

For each feature described by the interviewees, we asked about the reasons why they considered it to be typical, good, bad, or outlier. We then asked to discuss the feature from the perspective of each facet. When interviewees had difficulties answering our questions, we used a "by example" strategy, providing possible answers to our facet questions, as described in Table 2. In the case of surprising responses, we dug deeper by asking specific questions, trying to elicit the underlying reasons for such responses.

2.3 Data Collection and Analysis

Each of the conducted interviews was fully recorded, with the obtained recordings lasting between 68 and 117 minutes. In addition to these, core answers to our facet-related questions were summarized by the interviewer during the interview itself. These summaries were further cross-checked against the recording by an author who did not participate in the corresponding interview. Such reviews were used to verify the summaries and to augment them when needed. The obtained information was used to describe the company-specific features in terms of their facets in Sec. 6.

Moreover, we created full transcripts for parts of the interviews that discuss the rationale behind considering a feature as good, bad, typical, or outlier. We applied open coding [2] to identify the main concepts related to this classification, which are discussed and exemplified in Sec. 4 and 5.

3. SUBJECT COMPANIES

We now provide background information about our subject companies and the product lines they develop.

3.1 Keba: Industrial-Automation Provider

Keba AG is a medium-scale company producing injection

Table 2: Variety of feature facets elicited in interviews

facet	question	examples
Rationale	Why does the feature exist?	Business reasons (e.g., customer request, market demand), regulatory needs (e.g., export restriction, country codes), aspects of the technical environment (e.g., platform, OS, library, installation environment) or social aspects (e.g., usage context, user needs)
Level	At what organizational level does the feature exist?	Customer-facing features (e.g., those managed by product managers) or technical features (e.g., logical and physical architecture, or implementation-level features)
Nature	What is the nature of the feature?	Primarily a unit of functionality (e.g., to characterize system capabilities, behavior, or data), a unit of variability (e.g., an optional functionality) or a configuration/calibration parameter
Scope	What is the scope of the feature?	Local to one component of a system or cross-cutting (i.e., scattered across architectural components)
Architectural responsibility	What is the architectural responsibility of the feature?	Addresses user-interface requirements, encapsulates some application logic, or infrastructure-level tasks
Lifecycle	Does the feature have a purpose for the lifecycle of a system?	Testing, debugging, build, optimization, packaging, deployment, simulation, or monitoring
Definition and approval	How has the feature been defined and approved?	Feature elicitation workshops with customers, systematic studies of similar system, or market analyses
Representation	Which artifacts/tools are used to define the feature?	Feature models, configuration tools, code-level configuration options, product maps, or spreadsheets
Use	In what ways is the feature used in the organization?	Defining a product line's scope, explaining a system to a customer, changing behavior at runtime, or supporting software composition
Dependencies	What are the dependencies to other features?	Dependencies between features (e.g., require or exclude), dependencies across levels (e.g., to logical and physical components)
Implementation and deployment	Which languages and technologies have been used to implement and deploy the feature?	Programming languages used, build-time integration of libraries, feature deployment in app stores
Inclusion/Exclusion	Which mechanisms are used for including or excluding the feature?	Configuration tool, configuration file, user preferences at runtime
Binding time	At what stage is the feature included into the product?	Compile, build, load, or run time
Responsibility	Which people, roles, or teams are in charge of the feature?	Application engineers developing customer-specific features, platform engineers developing core functionality
Position in hierarchy	What are the features above (if any) and the features below (if any)?	Concrete feature names or not applicable
Testing	Which methods and tools are used for testing the feature?	Automated component test suites or manual system integration tests
Evolution	How did the feature change over time?	Changed frequently (e.g., daily, weekly or monthly), mostly stable, rolled out, or retired
Metrics	Which metrics are used to characterize the feature?	Number of products in which a feature exists, number of feature instances per product
Quality and performance	Which non-functional characteristics are important for the feature?	Reaction time, power consumption, or efficiency of a feature implementation

molding machines, energy appliances, and robotics solutions used for industrial automation [22]. Around 700 people are employed at the head office in the company's home country Austria, while a branch in China exists for the Chinese market. Keba's industrial-automation solutions include hardware, software, and tools. We focus on the software product lines for injection-molding and robotics solutions. Keba ships about 7,000 injection-molding solutions to 25 customers per year. Four resellers are available for the injection-molding branch. Furthermore, Keba sells about 1,800 robotics solutions to about 30 customers per year and works with six resellers related to the robotics branch.

Architecture. Layered technological platforms exist in diverse variants to meet requirements in different market segments. Keba's automation platform is organized as a product line, and different variants are derived to develop domain solutions for injection molding, robotics, and energy automation. The platform for injection-molding machines provides an application framework, while the robotics platform uses a DSL-based approach for programming robots. The different layers and their interfaces strongly influence the development process: multiple system platforms are derived from a system platform architecture to support multiple runtime systems. Domain solutions are built on top of each system platform, by exploiting the interfaces of the platform. Products are defined by adding new functionality on top of the domain solutions using cloning. Products are fine-tuned using configuration parameters during installation and setup.

Keba uses a wide range of variability mechanisms to support product derivation: Platform and product variants are created by exploiting interfaces to hook in new functionality; by adding, exchanging or reloading modules; by defining specific I/O-ports; by modifying parameters to influence program behavior; by adapting configuration files to change system behavior and performance as well as by pre-processing code to integrate specific product variants during compilation.

Organizational Structure. Dedicated teams maintain the automation platform and the domain-solution platforms. Project teams then work with customers on individual products. There are also external developers contributing code—for example, domain engineers working for OEMs, and application engineers working for resellers. This has a significant impact on Keba's development process, challenging platform evolution in particular. Domain-solution engineers regularly review and prioritize features for upcoming releases.

3.2 Opel: Car Manufacturer

Opel is a German subsidiary of GM—a large car manufacturer operating across 157 countries, comprising 202,000 employees, and having sold 9 million vehicles in 2011.

Opel has software product lines aligned with the mechanical product lines and the engineering culture of cars. The product lines discussed in this paper are part of an initiative at GM called Next Generation Tools (NGT) [14], created to handle the complexity introduced as a by-product of new technologies, such as hybrid and alternative-fuel engines. SPLE plays a major role in the NGT and is implemented following the *Second Generation PLE (2GPLE)* approach, in which features are treated as first-class citizens. Vehicles can now be described in terms of a *bill-of-features*, which facilitates the communication between business, marketing, and engineering units. The tool used for modeling the features is BigLever GEARS. The development process is organized in five different levels covering feature model, requirements,

logical architecture, technical architecture, and deployment. Development activities correspond to the "V-Model": The process starts with defining requirements and architecture for the future vehicle, and at the bottom of the V-Model is the creation of hardware and software components that correspond to the requirements specification.

Architecture. Opel uses a system-of-systems architecture managed as a hierarchical product line of product lines. It covers domains, subsystems, functions, functional elements, and components (being aggregations of functional elements). The software components are as general as possible to allow flexibility with respect to variations. The manufacturing process is driven by the selection of features and part numbers of physical car components that determine which ECUs (electronic control units) are in the car. The ECUs contain the feature implementation, and their presence determines whether a feature is available. The software components are specifically configured for every produced vehicle. In this calibration process, the Vehicle Option Codes—parameters determining the startup of optional software components installed by ECUs in the car—are saved to a flash database.

Organizational Structure. There are different teams, including individual teams for each product line; they are referred to as a body of knowledge—the teams have specialized knowledge about the instantiation of their product line. There is also the concept of a feature owner, referring to the main technical contact person in charge of a feature. The feature owner also models the feature in GEARS.

3.3 Danfoss: Component Producer

Danfoss is a large producer of electronic and mechanical components for industrial and consumer applications. It has 23,000 employees globally, distributed across 56 factories in 18 countries. We focus on Danfoss Drives, a subdivision producing frequency converters (drives), which are used in a wide range of applications, such as for HVAC or for winding machines in the textile industry. Consequently, the drive firmware has a high degree of variability, as the motors to be controlled vary significantly. While the variability had initially been handled using a clone&own approach [12], Danfoss later adopted an SPLE approach by migrating the cloned products into an integrated platform [16].

Architecture. Danfoss has multiple product lines, each realized with a typical embedded-platform architecture and a codebase of a few million lines of C/C++ code. Our focus, the frequency-converter product line, consists of a platform, which realizes 14 main products, complemented with additional repositories of 30 extensions ("sub-products"). The platform's variability is realized using the C/C++ preprocessor by referencing static features in conditional-compilation directives (e.g., #IFDEF); by generating build files; and by using dynamic parameters that influence the run-time of a concrete product.

Features are defined in a feature model and mapped to source files using the commercial tool pure::variants from pure::systems. Upon creating a configuration (i.e., a selection of features), pure::variants generates the build files and the set of parameters belonging to the product-specific configuration. Not all parameters are present in all products, and the configuration of these parameters (e.g., limits and default values) varies considerably across products.

Initially, only variability in source code was managed. Following positive experience, variability management was extended to other artifacts, such as requirements and test cases [30]—enabling the derivation of variants of these artifacts by configuration and improving traceability.

Organizational Structure. A dedicated team involving software architects from all development sites supports and maintains the platform. Application-engineering teams contribute new functionality of the drives. However, there is no real split between application and domain engineering. The platform team is also responsible for the feature-model development, and each feature is assigned to a feature owner. From the product perspective, there are product managers who determine which features will go into a product. When there is a change request for a particular feature, the product manager has to contact the feature owner.

4. TYPICAL AND OUTLIER FEATURES

We now introduce concrete examples of the features we studied, beginning with typical and atypical (outlier) features. We discuss the rationales behind our interviewees' classification, complementing them with quotations. Table 3 shows all features with their ID and the respective classification-rationale codes. These codes—indicated by `special formatting` in the following descriptions—are further explained in Table 4.

4.1 What is a Typical Feature?

Most interviewees argued that typical features represent core functionality of the `domain` and are, thus, a prime prerequisite for a company's business. The feature Danfoss.Wobbler is an example: It was not explicitly requested by a customer, but is regarded essential in the textile industry domain as it allows frequency converters to operate properly without producing waves. Another feature providing core functionality is Keba.ProfiNetSlave, implementing inter-machine communication based on the Industrial Ethernet standard Profinet.

Other typical features are either generally demanded by the market or requested by a specific customer. For instance, the feature Opel.LaneKeeping satisfies a very common `market demand` expressed nowadays by car buyers. `Customer request`s are also considered as common examples, such as expressed by interviewee **E:** *A typical feature is one requested by a customer, a new motor control for example. It's a functional one.* He refers to Danfoss.Torque, which extends existing motor-control functionality.

Keba.LanguageTranslation is another example of a typical feature, as it was realized following a management decision (`internal standard`) to offer certain system localizations by default to support customers in specific regions. It was also marked as typical since it represents `ubiquitous` functionality affecting not only the EasyNet control-station program for which it was initially conceived, but now almost all parts of the product line.

4.2 What is an Outlier Feature?

Not surprisingly, there is less agreement in what constitutes an outlier feature, with almost everyone giving different reasons. Neither did we observe any overlap in the rationales between classifying a feature as typical or as an outlier.

For instance, the feature Keba.SilentMode enables installations without user interaction. It is realized as a hidden command-line option of the setup program and only used internally for testing; it does not provide a core functionality for customers (`deployment`).

Table 3: Features and their classification rationales

	company	feature ID	feature description	classification rationale
typical	Keba	LINMovement	linear movement command of a robot	`domain`
	Keba	ProfiNetSlave	implementation of the PROFINET standard for the fieldbus communication stack	`domain`, `market demand`
	Keba	LanguageTranslation	translation of the EasyNet control-station program into several languages	`ubiquitous`, `market demand`, `internal standard`
	Opel	LaneKeeping	driver assistance to keep the lane, including active steering	`domain`, `market demand`
	Danfoss	Torque	enables high starting-torque for permanent-magnet machines	`customer request`
	Danfoss	Wobbler	smooths the movement of electric motors (to avoid waves) in the textile industry	`domain`
outlier	Keba	SilentMode	non-interactive ("silent") installation procedure	`deployment`
	Keba	UserGuidance	user guidance for device configuration	`placeholder`, `usability`
	Keba	HeatUpOptimization	optimizes heat-up procedures of multiple machines by distributing power peaks	`optimization`, `startup`
	Danfoss	PowerUpFastFuncs	moves the execution of some functions from the flash drive to the RAM	`optimization`, `build`
	Danfoss	BoardSupportPackage	new Hardware Abstraction Layer (HAL) to improve board support	`evolvability`, `maintainability`
good	Keba	Oscilloscope	software oscilloscope for recording signals	`error-free`
	Keba	Wizard	wizard-based configuration support for the initial robot setup	`popular with customers`
	Keba	ProductionOverview	historical overview on the production process (operation modes and "shots")	`popular with customers`, `popular with developers`
	Opel	ParkAssist	automated steering in parking situation	`well modularized`
	Danfoss	CascadeController	enables control of multi-drive (pump) setup to manage pressure or level	`well implemented`, `well modularized`
	Danfoss	FieldBus	enables the fieldbus communication stack for frequency converters	`distinct functionality`, `well modularized`, `thoroughly tested`
bad	Keba	Euromap	enables the Euromap protocol support in the fieldbus communication stack	`test challenges`, `frequent changes`
	Keba	ManualConfiguration	manual configuration of EncoderBox	`market demand`, `rushed development`, `customer complaints`
	Keba	DataManager	export/import of low-level machine data	`rushed development`, `workaround`, `variability`
	Opel	EmergencyBraking	autonomous emergency braking	`highly cross-cutting`
	Danfoss	ProductG	implements product-specific features for one particular customer	`highly cross-cutting`
	Danfoss	ResetFix	fixes a defect in another feature (Reset Counter)	`defect fix`, `frequent changes`

Another interesting example is the feature Keba.UserGuidance, which serves as a `placeholder` for product managers to plan future `usability` improvements regarding device-configuration support. Specifically, users requested better dependency resolution and choice propagation in the configurator to catch misconfigurations early, which otherwise could only be discovered at system startup. Our interviewee also classified this as a bad feature since it is too vague (explained shortly in Sec. 5.2) and needs refinement. **B:** *We didn't really know how to improve it. That's why we are using it as a placeholder for projects where one needs to improve something. [...] Basically, it's an accumulative feature.*

Two of the outlier features—Keba.HeatUpOptimization and Danfoss.PowerUpFastFuncs—were only introduced for the `optimization` of non-functional aspects as explained by an interviewee: **E:** *Outliers are technical features for tuning and tweaking performance. It's not really a feature from the drive perspective, but we can configure a drive to be faster or slower. We can tweak the product to indirectly fulfill the customer requirements.* Furthermore, these two outliers only have a specific lifecycle purpose—they control the `startup` or the `build` process of the system. For instance, Danfoss.PowerUpFastFuncs improves system performance by moving functions from flash to RAM using a dedicated compiler macro.

Finally, Danfoss introduced the mandatory feature BoardSupportPackage, a more intelligent hardware abstraction layer (HAL), to reduce the number of variants. Third-party board vendors urge Danfoss to update the boards by increasing the price for old boards. The more robust abstractions provided by this feature account for improved `maintainability` and `evolvability` to quickly support new boards. This feature is not visible to the customer and only exists at the architecture and development level. However, it needed to be approved by the product management and other engineering teams to verify that it does not negatively affect existing business logic. In this light, considering this new HAL as a feature makes it a *unit of maintenance*, which various teams can use for communication and management can use for planning.

5. GOOD AND BAD FEATURES

We also studied features perceived as good or as bad by our interviewees. Similar to the discussion above, we now introduce examples of such features, describe the underlying classification rationales, and provide illustrative quotations.

5.1 What is a Good Feature?

In two cases, our interviewees mentioned customer satisfaction (`popular with customers`) as their prime rationale for considering a feature as good. For instance, the feature Keba.Wizard allows performing the initial robot setup and installation within a few minutes by avoiding error-prone manual configuration. Keba.ProductionOverview provides monitoring capabilities for injection molding machines, including an overview of operation modes and the history of production sequences. In particular, customers like the possibility to inspect variables during system operation. The feature is also highly valued by Keba developers who use it for diagnoses (`popular with developers`).

Features also need to provide a `distinct functionality` to the product line. On the other hand, ambiguous features not meeting this criterion are considered as bad features (explained shortly in Sec. 5.2). Recall the outlier feature Keba.UserGuidance, which was only vaguely understood by customers and the management.

Features have also been rated as good if they are perceived as `well implemented` and `error-free`. For instance, the feature Keba.Oscilloscope provides signal charts and two-dimensional plots for monitoring and diagnosing robotics solutions. According to our interviewee **A:** *it just always worked.* General statements about the implementation aspect further emphasize the absence of surprising feature interactions and adherence to architectural rules: **E:** *A good feature fulfills the requirements, but does not introduce any bugs on the way or impact existing features of the product line. [...] It has to follow the architecture rules and the coding style.*

Features are also considered as good if they are `well modularized`, not cross-cutting multiple components. For

Table 4: Expressed feature-classification rationales

	rationale	description	occ.[1]
typical	`domain`	core domain functionality	4
	`ubiquitous`	common feature affecting many assets	1
	`market demand`	required to be competitive in the market	4
	`internal standard`	management decision for an internal standard	1
	`customer request`	requested by a specific customer	1
outlier	`deployment`	only supports system deployment	1
	`placeholder`	placeholder for future improvements	1
	`usability`	improves usability for customers	1
	`optimization`	optimizes a non-functional aspect (e.g., power consumption, performance)	2
	`startup`	controls or affects system startup	1
	`build`	controls or supports build process	1
	`evolvability`	improves future system evolvability	1
	`maintainability`	improves system maintainability	1
good	`popular w/customers`	positive customer feedback	2
	`well modularized`	feature implementation limited to module	3
	`popular w/developers`	supports system diagnosis	1
	`error-free`	no or very few defects since inception	1
	`well implemented`	implementation adheres to architecture rules and coding styles	1
	`distinct funct.`	graspable concept that is easily understood	1
	`thoroughly tested`	high confidence in correctness	1
bad	`rushed development`	implemented under pressure	2
	`workaround`	compromise during implementation	1
	`customer complaints`	bad feedback from the market	1
	`frequent changes`	implementation modified frequently	2
	`highly cross-cutting`	scattered feature implementation	2
	`test challenges`	difficult or laborious to test	1
	`variability`	need to make a feature optional	1
	`defect fix`	defects that became features	1

[1] number of occurrence of the rationale in interviews

instance, the feature Opel.ParkAssist is not scattered across multiple components, therefore significantly limiting coordination effort between the suppliers, who commonly implement the components.

5.2 What is a Bad Feature?

Among our interviewees, bad features are usually the result of time pressure and `rushed development` as well as compromises made during implementation (`workaround`). For instance, the features Keba.ManualConfiguration and Keba.DataManager were implemented too hastily, resulting in low implementation quality. The feature Keba.ManualConfiguration, supporting the configuration of an encoder box to reduce wiring costs, was developed under time pressure and released prematurely to selected customers. However, this led to negative market feedback (`customer complaints`) as expressed by interviewee **B:** *There was an extreme pressure from the customer side to support this feature. [...] We sensed that customers would not agree with [the complicated configuration], but due to pressure we realized it. [...] Would have been better to realize the feature with complete tool support before release.*

A related problem are highly volatile features (`frequent changes`), such as the feature Opel.EmergencyBraking, which required continuous improvements and extensions to support an increasing number of deceleration profiles and object types recognized on the road.

Scattered and `highly cross-cutting` feature implementations led to a bad perception of two features: Danfoss.ProductG realizes a request for one specific customer, requiring many little tweaks to the code. Opel.EmergencyBraking is also considered as a highly crosscutting feature. It is also more complex than the typical feature Opel.LaneKeeping due to different deceleration profiles and object types as well as more sub-variants and parameters for calibration. However, cross-cutting features are not necessarily considered bad, as we will discuss in Sec. 6.2.

Interestingly, the necessity to make a feature optional (`variability`) has also led to issues. The feature Keba.DataManager supports copying mold and protocol data between an injection molding machine and a PC, allowing modification of the machine cycle. Due to safety concerns of a key customer, the feature had to be made optional, resulting in high effort.

We also observed features that originated from defects (`defect fix`). This happened for the feature Danfoss.ResetFix, where a bug fix for a reset functionality of counters was defined as a new feature. It fixes an unintended feature interaction between a counter feature and a feature providing a reset functionality for the counter, resulting in incorrect counting. In general, defining bug fixes as features helped Danfoss to support customers who were used to the incorrect behavior and did not wish a mandatory bug fix.

Finally, both duplicate and superfluous features were reported as bad features. Danfoss reported that the same functionality—for instance, the feature ResetFix from above—was implemented twice due to a lack of coordination between the application-engineering teams. Superfluous features are those that were developed but never used, effectively wasting development effort. None of our studied features belongs to this category, but Danfoss reported such features: **E:** *The worst case scenario is a feature which has been implemented but not used at all. It happens.*

6. CROSS-CASE ANALYSIS

In our study, we also conducted a cross-case analysis by investigating the various facets (cf. Table 2) of all 23 features. We now discuss the results of this analysis and explicitly formulate the core observations.

6.1 Rationale, Level, and Nature

`Rationale`. In all subjects, typical, good, and bad features were mainly introduced for business-related reasons, such as general market demand and customer requests. However, the introduction of certain operational modes (e.g., Keba.Oscilloscope, a feature adding logging and monitoring support) or regulatory requirements (e.g., ECE-R 79 requirements on steering interventions and automatically commanded steering for Opel.ParkAssist) were mentioned as well.

Outlier features were often the result of a technical concern that had to be addressed. Indirectly, these features also realize customer requirements. Their prime rationale was the support for dedicated `Lifecycle` tasks of a system, such as deployment, debugging, monitoring, configuration, system startup, or usability improvements.

`Level`. We observed that the notion of features is used at all organizational levels. For almost all of our features, traces exist at all levels. For Keba, these levels comprise product management, architecture, and development. Opel has similar levels, ranging from the feature-definition level at the top via the requirements, the logical and physical architecture, down to the deployment level. Similarly, for Danfoss, nearly every feature is identifiable at the business level down to all other levels.

We also found that most of the features were leaf features, which were very concrete and easy to describe, and top-level features defined in product maps. Intermediate features were fuzzier and more abstract, making it difficult for the practitioners to talk about them. Two interviewees explicitly stated this point, for instance, **F:** *Product features*

on a top-level are good features, which describe a specific functionality [...] Yet, intermediate features are frequently used, both for grouping purposes (Danfoss) or for defining functionalities with different variants (Keba and Opel).

Observation 1: ***Outlier features.*** *Features do not only address functional or non-functional concerns that end up in a product. Features are also used for atypical purposes, such as supporting a system's lifecycle.*

Outlier features are an important part of the development process. Yet, they do not need to be developed according to the full feature development process. In other words, they do not exist on all levels. For instance, Keba.UserGuidance solely exists at the product-management level and is used internally. Keba.SilentMode exists only at the development level; it is a hidden command-line option used only by service engineers during setup. Also, Danfoss.PowerUpFastFuncs exists only at the architecture and development levels.

Thus, outlier features are coordinated or implemented only by a subset of the typical roles involved (e.g., developers, architects, or product managers) and are in most cases not visible to the customer (Keba.HeatUpOptimization is an exception). Surprisingly, according to our interviewee, only Opel has no such outlier features. For the domain under consideration, all features currently represent functionality. However, due to its long engineering history, Opel has additional co-existing entities (e.g., basic software components) that might be used for this purpose. Investigating these entities and their relation to features is valuable future work.

`Nature`. The features we investigated were treated primarily as a unit of functionality to define system capabilities, behavior or data. This is often the case for mandatory features covering core functionality. Only as a secondary aspect are features also a unit of variability—when the functionality should be optional. Recall Keba.DataManager, where an allegedly mandatory addition (a new feature) had to be made optional, causing substantial development effort. In both cases, features do not only provide a unit of functionality, but can immediately serve as a unit of variability when necessary—without the need to introduce a new feature, but potentially with significant implementation effort.

Only one of our features (Danfoss.CascadeController) primarily offered parametrization to other existing features or functionality. Interestingly, Keba decided to consider the support for the configuration of other features as features themselves: Keba.UserGuidance, Keba.Wizard, and Keba.ManualConfiguration. Yet, recall that the first one was just a placeholder for future plans to improve the usability of the device configuration.

Finally, almost every feature came with further configuration (a.k.a. calibration) parameters to fine-tune it. Danfoss and Opel manage large parameter databases. For instance, at Danfoss, the feature model has about 1,000 features, whereas the parameter model has about 2,800 parameters. Parameters are either directly assigned to and controlled by features or can be stand-alone, as in the case of Danfoss. The latter arose for historical reasons—the parameter database existed before features and a software product line were adopted.

Observation 2: ***Features vs. parameters****. Parameters are not treated in the same way as features.*

Parameters are important entities managed by our companies in addition to features. Yet, the handling and the characteristics of parameters are different. Parameters do not have a process attached to them like features do, have no architectural responsibility and no dedicated responsible role (usually the feature owner is also responsible for parameters).

6.2 Scope and Architecture

`Scope`. Not surprisingly, we observe both localized and cross-cutting features, scattered over large parts of the product line. Half of Keba's and Danfoss' studied features are cross-cutting. For instance, Keba.Oscilloscope introduces logging and a global monitoring mode for operating the system, affecting many parts of the codebase. In Opel's active safety domain, almost all features are cross-cutting, with implementations being scattered over many components and ECUs. Among the three Opel features, only one (ParkAssist) is well localized in its current implementation. Another highly cross-cutting feature, spanning many domains, is Danfoss.ProductG. It is the result of one customer request whose realization needed many tweaks throughout the codebase.

Yet, while some features are bad features due to their highly cross-cutting nature, the scope of a feature is *not* a differentiator between good and bad features. As one interviewee explicitly explained, **F:** *If a feature is cross-cutting, that itself is not bad. There can be good reasons for a scattered feature implementation.*

Observation 3: ***Cross-cutting features.*** *Scattered feature implementations do not necessarily lead to problematic features.*

`Architectural responsibility`. The majority of the studied features across all companies contributes core business logic. At Keba, all of the features discussed affect the user interface (UI). At Opel, all three discussed features affect both substantial business logic and the UI. At Danfoss, most of the features (except the outliers) handle business logic, whereas only one feature (Danfoss.Wobbler) also contributes to the UI. This small number is not surprising given the small display panels built into frequency converters.

The outliers, and other features to a lesser extent, almost always affect the product-line infrastructure for a specific lifecycle purpose. However, recall the outlier feature Danfoss.BoardSupportPackage, which only contributes a new architecture (and some business logic). Defining this new architecture as a feature allowed internal communication and approval (explained shortly in Sec. 6.4), but also booking developers' time on realizing the feature. This was also explained by a Keba interviewee: **B:** *There are internal features [...] [used] for project controlling [...] [to communicate] how much of our time we invest into them.*

6.3 Process and Representation

`Definition and approval`. While a deep study of processes is beyond the scope of our work, we observed a diversity of processes. At Keba, the features we studied are defined by an internal project team, or existing specifications are used in case the feature implements an existing standard (e.g., for Industrial Ethernet). Sometimes, capabilities found in similar systems are also studied. Usually, no dedicated approval is necessary—none of the studied features required this. Opel follows the typical V-shaped software-engineering process: New features are defined in a so-called Advanced Technology Work project, comprising the elicitation of requirements and the building of a prototype vehicle. There is

special focus on safety-critical aspects; for instance, possible feature interactions are investigated. Features are commonly redefined based on customer clinics and field experience. At Danfoss, a feature is typically created based on input from customers and goes through a regular development process (requirements, analysis, etc.). Before going into a product, it has to be approved by the product owner. Danfoss' outliers are created without customer involvement.

It was surprising that the actual process was *not* a differentiator between good and bad features. However, as briefly discussed above (Sec. 5.2), our feature sample shows that time pressure is a clear indicator.

Observation 4: ***Immature features.*** *A rushed development process causes problematic features.*

Danfoss reported on bad experiences with an experiment called "time-boxing": **E:** *Because of the time pressure, we are told not to think, just to implement. [With time-boxing] there was a limited amount of time to do some things. It was OK to implement the code, but not to do any documentation.*

`Representation`. While Opel and Danfoss represent features in dedicated feature-modeling tools, the situation at Keba is more diverse. Keba defines high-level product features and their descriptions in product maps—matrices that allow comparing related products across numerous features—using the Polarion requirements management system and spreadsheets. Feature requests are also managed in an issue-tracking system, used by application engineers to communicate with domain engineers about future platform features. Keba also has a home-grown configuration tool that, relying on selecting and customizing features by developers, allows quickly cloning a product variant based on the domain platform. Developers further use configuration files to define lower-level features and parameters associated with features.

Opel uses GEARS for features and DOORS for requirements. Surprisingly, safety-critical dependencies are currently modeled in the requirements, not in the feature model. At Danfoss, every feature is represented in one central feature model managed via pure::variants (with current modularization attempts). Features are cross-linked to the parameter database and requirements are managed in CaliberRM.

6.4 Use, Implementation and Deployment, and Product Derivation

`Use`. Most frequently, interviewees reported that features are used for explaining a system to a customer and for internal communication. **B:** *On the one hand it is the communication to the customers—which features we have. [...] On the other hand, it's also for communication with the development.*

Features were also frequently used for other closely related activities such as scoping to create awareness for feature reuse. **B:** *When planning a project, we say that we can do a project with those features. [Then] someone comes and says [...] we can realize automated tests with the existing features.*

Configuration was another common use of features. Some of Keba's and Danfoss' features also contribute a configuration interface, allowing a feature's parametrization by customers during setup or run-time. All of Opel's features also contribute a large variety of calibration parameters used for feature customization during manufacturing.

`Implementation and deployment`. We observed a large variety of implementation techniques. To implement features, Keba uses multiple programming languages (e.g., C, C++, C#, Java, .NET, IEC 61131-3) and a home-grown domain-specific language (TeachTalk), in which high-level robot-movement commands are declared. Danfoss' features are implemented in C and C++, partly also using home-grown DSLs. For deployment, Keba and Danfoss exploit binary and properties files; Keba also uses OSGi bundles and the TeachTalk scripts.

For the domain under consideration, Opel's features are mostly developed in C and typically deployed as AUTOSAR components. All of the studied features are implemented and validated by the ECU suppliers, who receive a specification defining the calibration parameters the component needs to support (for instance, to realize other cross-cutting features affecting this component). The resulting AUTOSAR components are integrated and validated on the vehicle level and deployed to the ECUs. Given that most components realizing features are developed by ECU suppliers, most of the development and integration effort is spent on calibration.

`Inclusion/Exclusion`. The mechanisms for including or excluding optional features are very diverse. At Keba, for instance, dedicated robot commands (e.g., Keba.LINMovement) are activated at startup time by TeachTalk scripts, which also allow fine-grained customization of the movement logic. Other features are selected in the home-grown configuration tool or activated via a command-line option, a preferences menu in the UI, or a dedicated description file. The latter can either be a file delivered only to certain customers to activate a feature (e.g., Keba.ManualConfiguration) or a hardware-description file activating a feature when a certain kind of hardware is present. Opel's mechanisms are driven by the calibration parameters, whose values determine the startup of the feature's components, or by the presence of hardware (if not present, the respective ECU and component are not included). Danfoss uses the tool chain provided by pure::variants, comprising feature selection and a subsequent build process driven by the C preprocessor as well as the family model of pure::variants for determining the respective source files to include. For the outlier Danfoss.PowerUpFastFuncs, compile macros are used to instruct the linker to move functions from flash memory to RAM.

6.5 Quality Assurance and Evolution

`Testing`. For Keba, manual system tests are more important than automated test procedures, which are primarily used at the levels of components. Opel's procedures follow the typical levels outlined in the "V" development process: component testing, integration testing, and vehicle testing. The software is tested in an environment; integration testing is usually done on a bench or on a hardware-in-the-loop platform. Afterwards, the software is validated in the vehicle. Danfoss conducts integration and regression tests for a fixed set of products that are actually sold. Features are not tested via component tests.

We noticed that cross-cutting features are problematic in cases where they involve manual testing processes. Such features can usually only be tested at integration time, potentially also requiring hardware, making them high risk.

Observation 5: ***Testing and feature scattering****. Scattered features that have to be tested manually are problematic.*

`Evolution`. The features at Keba were characterized as stable with core functionality remaining unchanged. Customizations, refinements or refactorings are made upon request.

For instance, the feature Keba.Oscilloscope was extended to support additional chart types, and the feature Keba.SilentMode was recently refactored and re-implemented using a different programming language. At Opel, features in the active-safety domain are very dynamic. Therefore, a major part of the evolution effort is spent on calibration, minor adjustments of features, and code refactorings. For the fast-evolving feature Opel.EmergencyBraking, engineers gradually added support for recognizing additional objects to trigger the automatic brake (e.g., stationary vehicle, pedestrian, or bicycle in front).

7. THREATS TO VALIDITY

External Validity. As with any case-study research, it might not be possible to generalize the results of our work beyond the considered cases. Thus, we carefully avoided making any generalizations, but rather presented an in-depth analysis of the selected cases. We also focused on large, influential companies, which we selected using well-defined criteria (cf. Sec. 2.1), and sought to obtain a diverse sample of features. This sampling approach is commonly known as *theoretic sampling* [13]. To get a broader perspective, we selected interviewees covering a range of roles in the studied companies. We thus believe that the observations reported in this paper are of value to the wider SPL community.

Internal Validity. We see two main threats to internal validity. First, we might have phrased our interview questions in a way that affected the participants' answers, especially in cases where specific examples were given. We attempted to mitigate this threat by performing a pilot study and refining our interview guide when we observed that our questions raised confusion. We also avoided providing examples of possible answers unless the participants experienced difficulties in addressing a raised question. Second, we might have misinterpreted the participants' answers and derived incorrect conclusions, threatening the reliability of our study. To mitigate this threat, all interviews were recorded and their summaries were cross-checked by one co-author who did not attend the interview. Unclear cases were discussed and some were further verified with the interviewees.

8. RELATED WORK

Feature Definitions. Many definitions of the term *feature* exist [1, 17, 32, 15, 25, 18, 23, 20, 8, 31], each of which emphasizes certain feature characteristics. For instance, Kang *et al.* [19] provide a definition covering implementation, testing, deployment, and maintenance of a feature. Bosch mentions functional and quality requirements specifying logical units of behavior [7]. Further definitions focus on features as user-visible aspects [17] or features as aspects that are valuable to a customer [24]. However, existing approaches do not combine multiple feature characteristics, nor do they describe relations among them. The feature facets presented in this paper can be useful as a terminology for describing different properties of features. Our observations also indicate possible dependencies between such properties.

Feature Identification. Scoping methods propose a top-down approach to determining the boundaries of a product line and are an important planning activity that may determine the success or failure of a product line effort [26]. Some scoping techniques ground the identification of the product line scope based on business objectives [11, 27]. Scoping methods also cover the identification of features in product lines, but guidance is typically very specific in this regard. While feature identification is not the primary aim of our work, the presented facets of 23 real-world features can be useful for organizations in their scoping activities.

Variability Modeling. Variability modeling is essential for defining and managing the commonalities and variabilities in software product lines. A wide range of variability-modeling approaches has been proposed, including feature modeling [17], decision modeling [28], and orthogonal variability modeling [23]. Empirical studies report on experiences of applying variability modeling [6, 5]. Survey papers [8, 29, 3, 28, 10] compare variability-modeling approaches from different perspectives, which influenced the definition of feature facets in our study. For instance, a survey paper comparing feature models and decision models [10] uses ten dimensions to characterize the different techniques. Although the focus of our study was not on variability modeling, some dimensions described in this survey supported the definition of feature facets (e.g., applications, dependencies, binding time).

Feature-Oriented Engineering Methods. Feature-oriented software development (FOSD) is a programming concept for managing the construction, customization, and synthesis of software systems based on features as first-class citizens [1]. FOSD primarily addresses implementation-level aspects of features, whereas our aim was to empirically investigate a wider range of feature facets in different organizations.

Empirical Studies. Many experience reports about successful industrial product lines are provided by van der Linden et al. [31] and the SPL community's 'Hall of Fame'. While these provide valuable insights into economic, organizational, and process aspects of real-world product lines, only few details are given on the characteristics of individual features. Some experience papers provide more details about features at different levels of product lines. For instance, Lee et al. [21] report detailed experiences from developing an elevator control software product line comprising 490 features. Berger et al. [4] provide an analysis of features in 128 variability models, including metrics about feature types and feature dependencies. While these results helped us identify important facets of features, our aim was to complement existing empirical studies by conducting a qualitative study in companies and providing details about selected features.

9. CONCLUSION

We presented a qualitative study on the practical use of features in three large companies. The study provides a contextualized and in-depth analysis of 23 features in real-world settings—in organizations that manage features and explicitly track them. We reported insights into successful and failed practices of feature usage together with the respective conditions (RQ1) as well as a cross-case analysis on the range of feature definitions and usages in practice (RQ2).

What is a Feature? The notion of what a feature is varied widely across the three companies we studied. Yet, we observed a surprising consistency regarding what makes features good or bad. We also found that one of the most important characteristics of a feature is that it needs to represent a distinct and well-understood aspect of the system. We found that good features need to precisely describe customer-relevant functionality, that bad features primarily arise from rashly executed processes, and that cross-cutting features scattered over the codebase are *not* necessarily bad. We also

observed that outliers are necessary, but do not require the full engineering process of typical features. We hope that our results on the actual feature usage and on issues arising from it will be interesting for both practitioners and researchers.

Future Work. We plan to trace and study the lifecycles of features in more detail. Specifically, the insights we gained into the feature definition and approval process suggest that an in-depth study in this area would be highly valuable. Such a study should also capture the coordination among roles and teams required to engineer and evolve features. The paper used a set of facets for describing and communicating important characteristics of features. We plan to refine the facets as a basis for developing a language for describing features. Finally, we reported initial observations relating feature characteristics and their success. Further work should investigate the development of approaches that can help to predict when a feature is going to be good or bad.

Acknowledgments

We thank the companies, all our interviewees, and Manfred Schölzke for participating in our study. This work was partially supported by Keba AG and the Christian Doppler Forschungsgesellschaft Austria, the Artemis Joint Undertaking (grant 332830/2012-1), and the Ontario Research Fund.

10. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Techn.*, 8(5):49–84, 2009.

[2] E. R. Babbie. *The Practice of Social Research*. Cengage Learning, 13th edition, 2012.

[3] D. Benavides, S. Segura, and A. R. Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *J. Information Systems*, 35(6), 2010.

[4] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. Three Cases of Feature-Based Variability Modeling in Industry. In *Proc. MODELS*, 2014.

[5] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. VaMoS*, 2013.

[6] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Soft. Eng.*, 39(12), 2013.

[7] J. Bosch. *Design and Use of Software Architectures – Adopting and Evolving a Product-line Approach*. ACM Press, 2000.

[8] A. Classen, P. Heymans, and P.-Y. Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *Proc. FASE*, 2008.

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

[10] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *Proc. VAMOS*, 2012.

[11] J.-M. DeBaud and K. Schmid. A Systematic Approach to Derive the Scope of Software Product Lines. In *Proc. ICSE*, 1999.

[12] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. CSMR*, 2013.

[13] K. M. Eisenhardt and M. E. Graebner. Theory Building from Cases: Opportunities and Challenges. *Academy of Management J.*, 50(1):25–32, 2007.

[14] R. Flores, C. Krueger, and P. Clements. Mega-Scale Product Line Engineering at General Motors. In *Proc. SPLC*, 2012.

[15] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. 1997.

[16] H. P. Jepsen, J. G. Dall, and D. Beuche. Minimally Invasive Migration to Software Product Lines. In *Proc. SPLC*, 2007.

[17] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep., 1990.

[18] K. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 19(4), 2002.

[19] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Ann. Softw. Eng.*, 5:143–168, Jan. 1998.

[20] J. Lee and D. Muthig. Feature-oriented Variability Management in Product Line Engineering. *Commun. ACM*, 49(12):55–59, Dec. 2006.

[21] K. Lee, K. C. Kang, E. Koh, W. Chae, B. Kim, and B. W. Choi. Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice. In *Proc. SPLC*, 2000.

[22] D. Lettner, F. Angerer, H. Prähofer, and P. Grünbacher. A Case Study on Software Ecosystem Characteristics in Industrial Automation Software. In *Proc. ICSSP*, 2014.

[23] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer-Verlag, 2005.

[24] M. Riebisch. Towards a More Precise Definition of Feature Models. In *Modelling Variability for OO Product Lines*. 2003.

[25] J. Savolainen and J. Kuusela. Volatility Analysis Framework for Product Lines. In *Proc. ISSR*, 2001.

[26] K. Schmid. Scoping Software Product Lines: An Analysis of an Emerging Technology. In *SPLC*, 2000.

[27] K. Schmid, I. John, R. Kolb, and G. Meier. Introducing the PuLSE Approach to an Embedded System Population at Testo AG. In *Proc. ICSE*, 2005.

[28] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proc. VaMoS*, 2011.

[29] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and a Formal Semantics. In *Proc. RE*, 2006.

[30] K. Sierszecki, M. Steffens, H. H. Hojrup, J. Savolainen, and D. Beuche. Extending Variability Management to the Next Level. In *Proc. SPLC*, 2014.

[31] F. J. van der Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action*. 2007.

[32] P. Zave. FAQ Sheet on Feature Interactions. Available at http://www.research.att.com/~pamela/faq.html, 2004.