Principles of Feature Modeling

Damir Nešić
Royal Institute of Technology
Stockholm, Sweden

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany

Ştefan Stănciulescu
ABB Corporate Research
Baden-Dättwil
Switzerland

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT

Feature models are arguably one of the most intuitive and successful notations for modeling the features of a variant-rich software system. Feature models help developers to keep an overall understanding of the system, and also support scoping, planning, development, variant derivation, configuration, and maintenance activities that sustain the system's long-term success. Unfortunately, feature models are difficult to build and evolve. Features need to be identified, grouped, organized in a hierarchy, and mapped to software assets. Also, dependencies between features need to be declared. While feature models have been the subject of three decades of research, resulting in many feature-modeling notations together with automated analysis and configuration techniques, a generic set of principles for engineering feature models is still missing. It is not even clear whether feature models could be engineered using recurrent principles. Our work shows that such principles in fact exist. We analyzed feature-modeling practices elicited from ten interviews conducted with industrial practitioners and from 31 relevant papers. We synthesized a set of 34 principles covering eight different phases of feature modeling, from planning over model construction, to model maintenance and evolution. Grounded in empirical evidence, these principles provide practical, context-specific advice on how to perform feature modeling, describe what information sources to consider, and highlight common characteristics of feature models. We believe that our principles can support researchers and practitioners enhancing feature-modeling tooling, synthesis, and analyses techniques, as well as scope future research.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software reverse engineering**.

KEYWORDS

Feature models, modeling principles, software product lines

ACM Reference Format:

Damir Nešić, Jacob Krüger, Ştefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338974

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338974

1 INTRODUCTION

Software *product lines* are portfolios of system variants in an application domain. They are typically engineered as an integrated software platform that exploits the commonality and manages the variability among the variants [21, 66, 78], to facilitate systematic software reuse, enhance maintenance, and reduce costs. However, product lines are inherently complex because they integrate the *features*—many with complex dependencies—of all possible system variants [27, 30] in the codebase of one software platform. To handle this complexity, features need to be managed and, therefore, modeled.

Feature models [11, 24, 46, 68], introduced almost three decades ago, are the most common and popular notation for modeling the features of a software product line. Engineers use them to describe features and their dependencies in an intuitive, tree-like structure. Product-line users derive individual variants—represented by a valid combination of features and their values—from the feature model. To this end, feature models offer various modeling concepts, such as optional and mandatory features, a feature hierarchy, feature groups, and cross-tree constraints, as illustrated in a toy example in Fig. 1 (explained in detail in Sec. 2). Feature modeling is supported by major product-line engineering tools [5], such as pure::variants [15], Gears [53], and FeatureIDE [77]. Many researchers, practitioners, and tool vendors have introduced additional concepts, such as feature attributes, feature cardinalities [6, 7, 25] or non-propositional constraints [14, 65], thus gradually increasing the expressiveness of feature models.

Feature modeling has been intensively considered in research and applied in practice [9, 11, 59], as witnessed by over 5,000 citations of the original publication on feature modeling [46], the software-product-line community's hall of fame (splc.net/hall-of-fame), experience reports [74, 78], and textbooks [4, 21, 66, 78]. To complement feature-modeling tooling, researchers contributed hundreds of feature-model analyses [7, 63, 76], refactorings and management techniques [1, 2], as well as automated synthesis techniques [3, 72]. Most feature models are created manually

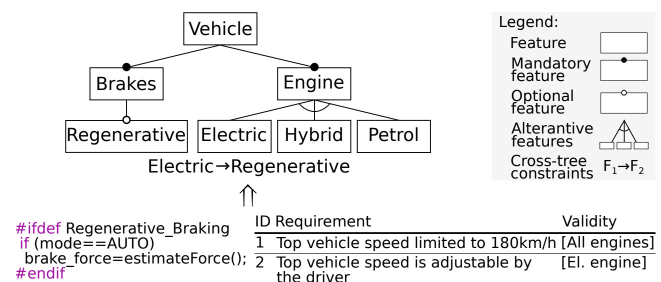

Figure 1: A feature model (top) with representations of features in code (bottom left) and requirements (bottom right).

and often in the context of re-engineering legacy variants into a software product line [11].

Despite decades of research, there is only limited knowledge on how feature models are, or should be, designed in practice. Despite numerous surveys, experience reports, and case studies [9, 11, 59], there is no modeling methodology for feature models that could be used to systematically design feature models based on established practices. It is not even clear whether practices that have been reported adhere to recurrent principles, and, if so, to what extent. We are only aware of one, 20-years old selection of design guidelines based on individual author experiences [58]. A modeling methodology, or at least a set of consolidated principles representing best practices, would not only help practitioners design feature models, but would also help the research community conceive better modeling tools, processes, and to scope future research.

Our long-term goal is to establish a modeling methodology and process for feature modeling. In this paper, we present a study of feature-modeling practices we identified from a systematic literature review of 31 papers and ten interviews with experts who created feature models in small to ultra-large projects with several thousands of features. By triangulating from these two sources, we collected practices performed by practitioners, analyzed them, and synthesized them into 34 general feature-modeling principles organized into eight categories (modeling phases): preparation and planning, training, information sources, model organization, modeling, dependencies, model validation, and maintenance and evolution. In detail, we contribute:

- 34 feature-modeling principles originating from practice;
- a detailed discussion of each principle with usage guidelines;
- an online replication package with more details (e.g., list of 654 papers identified during snowballing; the relevant interview and paper quotes for each principle).[1]

Overall, our principles are a first step towards defining a feature-modeling process and can help practitioners as well as researchers to understand best practices of feature modeling.

2 BACKGROUND AND MOTIVATION

In this section, we describe typical feature modeling concepts and the contexts in which feature modeling is applied. We refer to Fig. 1, in which we show a toy feature model from the automotive domain, to illustrate the relation of features to requirements and code. Real-world feature models are significantly larger, reportedly comprising up to 15,000 features in current editions of the Linux kernel, which relies on its own notation of feature models [13, 14].

2.1 Feature Modeling

Feature models organize features [8, 55], which abstractly represent the commonality and variability of variants in a product line. Being intuitive entities, features are also used beyond software product lines in agile development processes for single systems (e.g., feature-driven development, SCRUM or XP) as a means to communicate and maintain an overall understanding of the system. Feature models have been introduced as part of the FODA method [46] and quickly became popular due to their simple notation. One of the most valuable concepts is the *feature hierarchy*, as it structures features and allows users to navigate in the model. Beyond such ontological semantics, feature models express the set of all possible variants (configuration-space semantics) by using constraints as follows.

Feature Modeling Concepts. Features can be *mandatory* (e.g., Engine in Fig. 1) or *optional* (e.g., Regenerative) within a product line. Optional features can be enabled or disabled for individual variants, subjected to further constraints. The feature hierarchy imposes constraints in the form of parent-child implications, mainly to prevent the selection of features that would not have any effect when the parent feature is not selected. Features can have *types*. Most notations restrict features to Boolean, while others offer various types for features [12, 14, 71]. The original FODA notation defined features as Boolean, but also defines the notion of feature attributes—dedicated variables associated with features for holding non-Boolean values (e.g., integer, real values). Similarly to FODA, other feature-modeling notations have adopted the concept of feature attributes [7]. Whether non-Boolean values are useful typically depends on the domain. In our example, all features have Boolean values. *Feature groups* impose constraints on the grouped features. OR and XOR groups are most common, while MUTEX groups are rare [14]. Some notations have introduced arbitrary bounds (called group cardinalities) specifying the minimum and maximum numbers of features that can be selected from a group. In our example, the features Electric, Hybrid, and Petrol belong to an XOR group. Finally, all feature dependencies that cannot be expressed using these concepts can be declared as *cross-tree constraints.* These are typically propositional expressions over features, but some feature-modeling notations allow arithmetic or string constraints. Our example contains a very simple constraint: Electric implies the selection of Regenerative. Feature models are configured by assigning values to features, typically in an interactive configuration process supported by a configurator tool.

Feature Modeling in Practice. Given the need for upfront investments [20, 54], product lines are rarely developed from scratch [11]. Instead, most organizations start with ad hoc reuse practices, such as clone & own [18, 31], which are simple and cheap, but impose maintenance problems in the long run. As such, most product lines are adopted by re-engineering variants into an integrated platform, typically referred to as re-active (or extractive adoption) as opposed to proactive adoption [52]. Feature modeling is conducted during such migrations. Typically, as reported in experience reports [44, 45, 56], different artifacts are compared between individual variants, such as requirements and source code, to identify differences, which are abstracted into features and organized into a feature model.

2.2 Modeling Principles

For individual modeling languages, the literature offers modeling guidelines and principles. For example, there are guidelines for UML class diagrams [33], business process-models [61], or implementation principles for *domain-specific languages* [62, 75, 79]. There are established principles for software design, such as *divide and conquer* (break the problem into smaller parts) and *software layering*, where layers manage different concerns. Architecture design is also supported by established guidelines, such as the principle *high cohesion and low coupling* or the principle *information hiding*, both

[1] https://bitbucket.org/easelab/featuremodelingprinciples

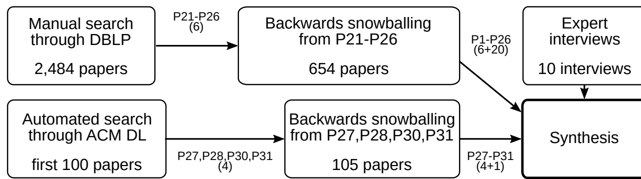

Figure 2: Process for identifying relevant data sources

of which increase modularity and facilitate the maintenance of a software system. Such general design principles are rather abstract and become practically applicable only when placed in the context of a particular language or a technique, such as modeling of object-oriented or aspect-oriented systems. Although we expect that feature modeling practices adhere to these general principles, it is unclear how to apply these principles to feature modeling. For example: what are the details of applying *high cohesion and low coupling* in a feature model, what information sources to take into account for modeling, or what organizational structure is needed?

3 METHODOLOGY

To identify modeling principles, we relied on two sources. First, following the structure of a systematic literature review according to the *snowballing method* [81], we identified papers that report about feature-modeling practices. Second, we conducted ten interviews with feature-modeling practitioners. These practitioners included vendors of feature-modeling tools who have reported their practices when guiding organizations during feature modeling. We analyzed both types of sources to identify modeling practices in either source, triangulated these practices, and synthesized them into principles.

3.1 Literature Review

To identify relevant papers, we relied on a lightweight literature review process because systematic literature reviews [51] spend significant effort on calculating various statistics about identified papers, whereas our primary goal was a qualitative analysis.

Search Method. As automated searches in digital libraries face several issues [17, 70], we decided to rely on the DBLP library and to manually analyze a set of relevant venues. To identify the initial set of papers, we analyzed publications from research and industry tracks in the proceedings of the last five instances (as of June 2018) of the following conferences and workshops: ASE, ESEC/FSE, FOSD, ICSE, ICSME, ICSR, MODELS, SANER, SPLC, and VaMoS.

As we show in Fig. 2, we analyzed 2,484 papers in this step. By reading their titles, abstracts, and if necessary the body of a paper, we identified papers P21–P26 (marked * in Tbl. 1) as the ones containing relevant feature-modeling practices. We used papers P21–P26 as the initial set for backwards snowballing. To this end, we analyzed each reference from P21–P26 by reading the title, abstract, and if necessary the referenced paper itself, thus leading to new relevant papers. In turn, we also analyzed the references of newly identified papers until the reference lists of all relevant papers were exhausted. Through the snowballing process, we identified papers P1–P20 (cf. Tbl. 1), and together with the initial papers (P21–P26), their reference lists contained 654 papers. To increase the confidence that we identified all relevant papers, we searched the *ACM Guide to Computing Literature* with the search string *(+Experience Report +(Variability OR Feature) +(Specification OR Model))*. We then sorted the results according to *relevance*, as all other sorting criteria are less appropriate, for example, date of publication or number of citations. By reading titles, abstracts, and if necessary the full papers, we analyzed the first 100 results and identified papers P27, P28, P30, P31 as relevant ones (marked ** in Tbl. 1). Then, we used these four papers as the initial set for another backwards-snowballing round that led to the identification of paper P29. The reference lists of papers P27–P31 contained 105 papers.

Selection Criteria. While analyzing the venues and applying backwards snowballing, we used the following inclusion criteria:

i) The paper is written in English.
ii) The paper describes the application of feature modeling where either (1) practitioners report their experiences, (2) researchers analyze open-source or proprietary systems and report identified practices or (3) tool-vendors or educators report experiences during feature-modeling training.
iii) The paper is either a reviewed paper or a technical report.

Besides conference and journal publications, we also included technical reports, because they can often include valuable industrial insights even without ensuring quality by peer review.

Data Extraction. From each paper, we extracted all details about the reported feature-modeling practices. In particular, we focused on lessons learned that helped us to assess whether a practice was helpful for the organization or not. The extraction was done by two of the authors, while the other two authors checked if the extracted material is sufficiently detailed, understandable, and relevant.

3.2 Expert Interviews

We complemented the practices reported in the literature by conducting semi-structured interviews with ten participants from nine organizations. The interviews were designed and conducted independently from the literature review in order to avoid introducing bias towards any practice. Each interviewee had several years of experience with feature modeling and they were recruited among our industrial partners and companies (e.g., tool vendors) having a close relationship to the research community. Each interview lasted around an hour on average, only one was considerably shorter, around 25 minutes. We first elicited the context of feature modeling in the form of characteristics of the project (e.g., domain, kind of system, team structure, implementation technologies), when the interviewee was from a company applying feature modeling. For the tool vendors, we asked about typical contexts. Second, we asked about the practices used for creating and evolving the models, including the involvement of different roles and characteristics of the resulting models (e.g., size, shape, and constraints). Finally, we elicited perceived benefits and challenges of feature modeling, among others, to get an understanding whether the reported practices were successfully applied or not. The interviews were conducted in person, via phone or Skype, recorded, and transcribed.

3.3 Synthesis of Principles

Our goal was to establish a set of principles. We define a principle as a generalized practice that has the following three characteristics: *a)* it concerns a well-established feature modeling concept; *b)* it is

Table 1: Overview of the 31 paper we identified. Single and double asterisks indicate papers that were used as initial sets for the first and second round of snowballing, respectively. Dashes indicate information that was not explicitly stated.

ID	Ref	Venue	Year	Research Type	Technique/Tool	Domain	Context	#Principles
P1	Kang et al. [48]	SPE	1999	Industrial case study/Method	FODA/FORM	Telecommunications	—	8
P2	Griss et al. [37]	ICSR	1998	Experience report/Method	FODA, UML	Telecommunications	—	6
P3	Kang et al. [47]	AnSE	1998	Experience report/Method	FODA/FORM	Multiple	—	6
P4	Cohen et al. [22]	—	1992	Industrial case study	FODA	Defense	Extractive	10
P5	Hein et al. [38]	SPLC	2000	Industrial case study	FODA, UML	Automotive	Extractive	22
P6	Lee et al. [57]	SPLC	2000	Industrial case study	Feature model	Elevators	Extractive	2
P7	Kang et al. [49]	Software	2002	Experience report	FODA/FORM	—	—	4
P8	Lee et al. [58]	ICSR	2002	Experience report	FODA	—	—	13
P9	Kang et al. [50]	—	2003	Industrial case study/Method	FODA/FORM	Inventory system	Extractive	7
P10	Sinemma et al. [73]	SPLC	2004	Industrial case study	COVAMOF	Multiple	Extractive	3
P11	Gillan et al. [35]	VaMoS	2007	Experience report	—	Telecommunications	Extractive	2
P12	Hubaux et al. [41]	SPLC	2008	Open source case study	OmniGraffle	E-government	Extractive	4
P13	Schwanninger et al. [69]	MODELS	2009	Industrial case study	Pure::variants	Industrial automation	Extractive	8
P14	Berger et al. [13]	ASE	2010	Open source case study	Kconfig/CDL	Linux/eCos	—	4
P15	Dhungana et al. [29]	JSS	2010	Industrial case study	Decision oriented	Industrial automation	Extractive	5
P16	Hubaux et al. [40]	VaMoS	2010	Literature review	Feature diagrams	—	—	4
P17	Berger et al. [14]	TSE	2013	Open source case study	Kconfig/CDL	Multiple	—	4
P18	Berger et al. [11]	VaMoS	2013	Survey	Multiple	Multiple	Multiple	5
P19	Manz et al. [60]	ICGSE	2013	Industrial case study	Feature model	Automotive	Extractive	4
P20	Berger et al. [10]	IST	2014	Open source case study	Kconfig/CDL	Systems software	Proactive	4
P21*	Berger et al. [9]	MODELS	2014	Interview study	Multiple	Multiple	Multiple	5
P22*	Chavarriaga et al. [19]	SPLC	2015	Industrial case study	SPLOT	Electrical transformers	Extractive	10
P23*	Gaeta et al. [34]	SPLC	2015	Industrial case study	SysML	Avionics	Extractive	3
P24*	Lettner et al. [59]	MODELS	2015	Industrial case study	FeatureIDE	Industrial automation	Extractive	15
P25*	Fogdal et al. [32]	SPLC	2016	Industrial case study	Pure::variants	Industrial automation	Extractive	5
P26*	Pohl et al. [67]	ICSE-SEIP	2018	Industrial case study	Pure::variants	Automotive	—	5
P27**	Boutkova et al. [16]	SPLC	2011	Experience report	Pure::variants	Automotive	Extractive	6
P28**	Nakanishi et al. [64]	SPLC	2018	Experience report	FODA/FORM	—	Teaching SPL	1
P29	Iwasaki et al. [43]	SPLC	2010	Experience report	FORM	Network equipment	Extractive	3
P30**	Derakhshanmanesh et al [28]	RE Journal	2014	Experience report	Pure::variants	Automotive	Extractive	5
P31**	Hofman et al. [39]	SPLC	2012	Experience report	FODA UML profile	Healthcare	Extractive	3

applicable to an arbitrary domain; and *c)* it is *up-to-date* and relates to modern tools and modern software-engineering practices.

We extracted nearly 190 instances of practices by reading through the identified papers and the interview transcripts. We wrote down each practice, which typically amounted to one to three sentences, keeping traceability to the original source. Then, while considering the modeling context, we iteratively merged redundant practices. Specifically, it was necessary to recapitulate how the overall modeling was done and what the characteristics of the model and project were. If these were not the same, the practices were strong candidates for generalized principles, as they seem to be applicable in various contexts. Likewise, if practices were contradicting, we reviewed the context and decided whether the practice should be made conditional upon the context. For instance, cross-tree constraints are regularly modeled in systems software, while many of the interviewees stated to avoid constraints modeling. After closer analysis, we realized that this depends on who configures the feature model. If this is done by a domain expert from the company itself, investing the effort into the typically expensive constraint modeling does not pay off, because employees are typically aware of these constraints. In contrast, if feature-model configuration is performed by end users, this requires a model with correct and complete constraints to prevent derivation of invalid configurations. During the iterative definition of principles, we discussed them among the authors until consensus of the formulation was reached and a category was decided. Initially, the categorization was inspired by Lee et al. [58], but the final categorization emerged from the set of included principles.

Table 2: Overview of our interviewees and their domains.

ID	Role	Domain	#Feature	#Principles
I1	Consultant	— (tool vendor)	—	14
I2	Consultant	— (tool vendor)	—	10
I3	Consultant	— (tool vendor)	—	6
I4	Architect	Automotive	≥1000	6
I5	Architect	Ind. automation	≤1000	6
I6	Arch./develop.	Web shops	≤40	5
I7	Consultant	— (various)	—	4
I8	Team leader	E-learning	≥1000	11
I9	Architect	Automation	≤100	9
I10	Team leader	Automotive	—	5

4 DATA SOURCES

Identified Papers. We identified 31 papers reporting relevant practices, as we show in Tbl. 1. The majority (14) of these papers are industrial case studies, followed by experience reports (10), open-source case studies (4), one survey, one interview study, and one literature review. The papers cover a period of over 20 years, are published in various venues, and use over ten different feature modeling notations. Moreover, feature modeling has been applied to 14 different domains. Most studies (18) are extractive (cf. Sec. 2.1), one is proactive, two have multiple contexts, and one relates to

teaching. The far right column contains the number of instances of principles we identified from each paper.

Interviewees. In Tbl. 2, we characterize our interviewees together with the domains in which they performed feature modeling. Six of them are *consumers* of feature modeling tooling, meaning practitioners applying feature modeling for the variants of their company (I4, I5, I6, I8, I9, I10). The remaining four interviewees are rather *producers* of such tooling (one was an independent consultant), and they mainly acted as consultants for companies adopting product-line engineering and feature modeling (I1, I2, I3, I7).

Interviewees **I1**, **I2**, and **I3** work in different roles for companies offering feature-modeling and product-line-engineering tooling. All of them have decades of experience in consulting companies in transitioning to product lines, mainly in the embedded systems domain. **I4** is a software architect in a large (≤99,000 employees) car manufacturer producing over 400,000 cars per year from three main platforms. I4 is involved in feature modeling and variability management in this product line. **I5** is a software architect responsible for variability management at a large (≤25,000 employees) vendor of electronics and mechanical components for end-user and industrial applications. The company has a large portfolio of variants, many of which were originally derived with the clone & own approach. **I6** is a department lead, acting as a software architect and a developer, in a small (≤50 employees) consulting company that delivers customized web-based e-commerce and enterprise applications. Specifically, I6 was involved in the design of a feature model, and the development of the target system. Around 400 to 500 possible variants can be generated from this product line. **I7** is a consultant who sells methodology and approaches for solving customer problems. I7 has experiences in using feature models with two different customers. **I8** is a project lead, who led the development of a large software migration project in the domain of university course and staff management. **I9** leads an architecting team and is a solution architect in a large international company (≤140,000 employees) providing solutions in industrial automation. For the needs of this study, I9 discussed the experiences of creating a product line with around 15 variants. **I10** is a software team leader in a large company (≤30,000 employees) in the automotive domain. I10 is responsible for managing configurable software for an embedded device, with an exponential number of possible variants that can be sold as applications to customers.

Interviewees' Product Lines. We briefly summarize the characteristics of the software product lines that the *consumers* of feature-modeling tooling engineer, reporting the sizes of the corresponding feature models in Tbl. 2. In the domain of **I4**, several hierarchically organized feature models exist, where the ones higher in the hierarchy contain hundreds and the lower ones thousands of features. Furthermore, because the company relies on different suppliers, the variability is implemented by different variability mechanisms. The product line of **I5** relies on the most traditional set-up with a single, moderately sized feature model, and where the variability is primarily bound statically at build time, among others, using the C preprocessor (e.g., *#ifdefs*). The domain of **I6** is modeled in a single feature model and the variability mechanism is a Java preprocessor developed in-house. **I8**'s domain comprises a large feature model with very few Boolean features, where the variability is mainly bound at runtime. The domain of **I9** relies on a single, moderately sized feature model, and the variability mechanism is a configurable build system for C++ based components. Finally, in the domain of **I10**, several models exist to represent the common components and their versions, as well as fine-grained features of those components, and configuration parameters for each feature. The system self-configures at boot time using a configuration file.

5 IDENTIFIED PRINCIPLES

We now present the 34 feature-modeling principles we synthesized from our data sources. Each principle defines *what* should be done or *how* it should be done. Furthermore, for each principle, we discuss *why* it is relevant and applicable. We grouped the principles into eight phases and ordered them in the most probable sequence they would be applied when building feature models. The phases in Sec. 5.1–5.3 capture principles relevant for pre-modeling activities, such as planning or identifying information sources. Phases in Sec. 5.4–5.7 capture those relevant for the actual modeling and validation of the obtained feature model. Finally, Sec. 5.8 captures those relevant for post-modeling activities, that is, maintenance and evolution. The acronyms used to label principles correspond to section titles and order of appearance within the section.

5.1 Planning and Preparation

PP_1: Identify relevant stakeholders. (P4). We identified this principle in P4, according to which the relevant types of stakeholders in the beginning (called *sources* in P4) are domain experts with detailed knowledge about the variants or platform that shall be modeled. Depending on the domain and organizational processes, these stakeholders can include diverse roles (e.g., architects, application engineers, project managers, requirements engineers). Identifying these stakeholders is one of the precursors for obtaining domain knowledge (IS_1) and conducting workshops (M_1). Another type of stakeholders is the modelers who will perform the actual modeling and subsequent model maintenance (called *producers* in P4). The roles that typically belong to this group are system and software architects as well as product managers, since their everyday work includes creating abstract system models. Identifying these stakeholders is required for training modelers (T_1). Finally, users of the feature model are another type of stakeholders (called *consumers* in P4). Depending on the purpose of the feature model (PP_3), this group can include diverse roles, from product managers to developers and any role in between. Understanding who the users of a feature model are will inform the decisions about model decomposition (PP_4), feature identification (M_6), and creating views (M_9)

PP_2: In immature or heterogeneous domains, unify the domain terminology. (P5, P6, P8, P9, P27). To facilitate the modeling process and model comprehension, it is beneficial to unify the terminology used by stakeholders and provide descriptive terms (e.g., for feature names). P8 advises that *"in an immature or emerging domain, standardize domain terminology and build a domain dictionary. If not done, different perceptions of domain concepts could cause confusion among participants in modeling activities and lead to time-consuming discussions."* Similarly, P27 explicitly states that *"the first step during the introduction of the feature-based variability modeling*

was the definitions for the all relevant terms [...] [as a] precondition for the successful collective work."

PP_3: Define the purpose of the feature model. (P12, P23, P24, P26, P30). Users of feature models can be divided into two categories. First, feature models can support design and management of a product line, for instance, as an explicit model of the domain or as a tool for *product-line scoping*. Second, feature models can support the actual product-line development, for instance, to coordinate teams developing different features or as an input for an automated build system and configurator. For example, P24 presents an industrial case study in which feature models are used to capture three different concerns, the *problem space*, the *solution space*, and the *configuration space*, which contains partial configurations of the problem-space feature model that are referenced by the solution-space feature model. Another example is P12, which presents an analysis of an open-source system in which a feature model supports system configuration. P12 notes that it was not clear if the feature model captures the *design time* or *runtime* variability. Therefore, if a feature model has multiple purposes, model views should be defined (M_9). Finally, a clear feature-model purpose avoids unnecessary effort, for example, by reducing discussions.

PP_4: Define criteria for feature to sub-feature decomposition. (P5, P8, P12, I4, I8). The semantics of the feature model hierarchy are not well-defined [23]; it can represent relations such as part-of, functionality decomposition, and feature responsibilities. Defining how and when a feature should be divided into sub-features facilitates achieving a consistent model that provides a single perspective on the product line (PP_3). According to P5, an indication of a good feature hierarchy is a low number of cross-tree constraints (MO_4). P12 confirms that there is no obvious way to build the feature model hierarchy and that prototyping is often required. Interview and paper sources report that the hierarchy represents functional decomposition (P8, I4, I8), which is in line with the most common interpretation of features, as we describe in M_6.

PP_5: Plan feature modeling as an iterative process. (P8, P22, P24). Not surprisingly, we learned that feature modeling should iteratively alternate between domain scoping and modeling. On the one hand, using an iterative process allows to gradually increase expertise in feature modeling; on the other hand it facilitates safe evolution of feature models, because the changes between feature model versions are typically not significant. For example, it allowed P22 to *"(1) train the domain experts using simpler models, (2) practice with them how to introduce new features and constraints, and (3) define practices to review and debug the models continuously."*

PP_6: Keep the number of modelers low. (P21, P22, P25, I1, I9, I10). Industrial practice shows that the number of stakeholders performing the modeling should be low, in some cases a single person. Typically, only few stakeholders have the overview domain-knowledge necessary to create the model. I1 states that *"it's usually the individual subsystem leaders or their architects or the lead designers in their subsystems that can capture the feature models."* I9 reports that only *architects* and *project managers* are involved in the modeling process. I10 also mentions that few domain experts are involved in this phase to define the needed behavior of the feature model and to discuss it with software experts.

5.2 Training

T_1: Familiarize with the basics of product-line engineering. (P25, P26, P29, I1). The stakeholders who will perform the modeling should familiarize themselves with the basics of product lines (e.g., product-line architecture, variant derivation) and especially with the notation of the used modeling technique and tool. It is beneficial to establish intuition about the correspondence between programming concepts used in every-day work, for example, classes or data types, and feature types and their graphical representations in the selected tool. I1 states that stakeholders can be trained by explaining, for instance, that Boolean features correspond to single checkboxs, and enum features to multiple checkboxes in the tool.

T_2: Select a small (sub-)system to be modeled for training. (P8). As we identified in P8, the initial system to be modeled should be small (see also PP_5), with a lot of commonality, and without strict deadlines regarding the release to production. This facilitates training of feature modeling and can improve feature-model acceptance, due to the ability to have arbitrarily fast feedback loops.

T_3: Conduct a pilot project. (P29, I1). The feature model training is ideally run as a guided exercise in the form of a pilot project that lasts several days, for instance, three days in the case of I1. The pilot project requires that the activities in principles T_1 and T_2 have been performed. I1 reports that in the context of extractive product-line adoption, it is necessary to guide the stakeholders that pose detailed knowledge about the variants, typically developers, towards understanding the differences between the variants in terms of domain concepts and not implementation-level differences. Often, when asked about the differences between variants, the developers respond with very specific, implementation-level answers. By asking for the reason of these differences multiple times (corresponds to M_3), developers will provide more and more abstract explanations, which is a core experience according to I1: *"every time they say something you say 'why', and now kind of abstract up one level you go why, and you know, after that 3 or 4 whys they will probably get to the essential feature."* Overall, analyzing around 20-50 variation points, identifying, and modeling corresponding features within the maximum of three days is a good target. Verifying that the obtained feature model can be configured as expected can be based on principle QA_2.

5.3 Information Sources

IS_1: Rely on domain knowledge and existing artifacts to construct the feature model.. (P1–P5, P15, P19, P22, P24, P25, P27, I1–I3, I9, I10). The sources of information to identify features are twofold. First, domain experts and stakeholders (PP_1) contribute their knowledge about the domain, existing systems, and customer needs. This type of knowledge is typically extracted and documented in workshops (M_1). I10 explains that *"There are certain application engineers [...] who are experts in making the link between what the behavior should be [...] and discuss that with the software experts to define what it should be doing."* Second, in the context of extractive product-line adoption, different engineering artifacts can be used to identify features and their dependencies, such as configuration files, domain vocabularies, specifications, manuals, contracts, documentation, and the actual source code. Features and their dependencies are typically extracted by identifying differences between the artifacts from different variants (cf. M_2, M_3). I1 explains how features are

retrieved manually: *"we look at source code clones/branches/versions to get the product differences (e.g., by looking at ifdef variables), and identify and extract features from these differences manually."*

5.4 Model Organization

MO_1: The depth of the feature-model hierarchy should not exceed eight levels. (P14, P17, P21, P28, I1–I5, I8, I9). While rarely made explicit in experience reports (only in P28), survey papers and most of our interviewees report that the feature-model hierarchy is typically between three to six levels deep. Feature models with a deeper hierarchy are typically split into several models (cf. MO_3). For example, I4 says that *"at most I've seen three levels deep"* similarly to I1: *"We usually don't see them going more than 3 levels deep."* I9 reports that *"It is more spread than deep I would say; It is just the way this model was evolving."* Even in the highly complex Linux kernel, the maximum hierarchy depth is eight levels (P17). Deep hierarchies are typically avoided in order to prevent deep sub-trees that the users must read and understand.

MO_2: Features at higher levels in the hierarchy should be more abstract. (P3, P4, P8, P9, P13, P14, P17, P21 P27, P30, I8). We found that the higher a feature is in the feature-model hierarchy, the more it is visible to the customers or it represents a more abstract domain-specific functionality. Features in middle levels usually represent functional aspects, while bottom-level features capture detailed features of technical concerns, such as the build process, hardware, libraries, and diagnostics. In P3, the levels in a feature model are (from highest to lowest abstraction): capability features, operating environment features, domain technology features, and implementation technique features. P2 uses the RSEB [36] methodology, which makes a distinction between *architectural* and *implementation* features. This principle is strongly connected to PP_4 and ensures a consistent, comprehensible model.

MO_3: Split large models (P4, P5, P12, P13, P15, P19, P22, P24, P26, P27, P30, P31, I1, I2, I8) **and facilitate consistency with interface models** (P5, P31). Several sources state that large feature models with thousands of features should be decomposed into smaller ones. For example, features representing implementation details and features representing user-visible characteristics should be placed in different feature models (compliant with MO_2). P15 states that *"our first brute-force approach was to put all model elements into one single model. This did not scale [...] It also became apparent [...] a single variability model is inadequate to support evolution in the multi-team development."* However, splitting large models raises consistency maintenance issues. An interesting principle for consistency maintenance is to identify the features that participate in inter-model dependencies and extract them into a separate feature model, the so-called *interface* feature model (P5) or a *feature dependency diagram* (P31). Then, the inter-model dependencies are isolated and easier to maintain. Even if the goal is to create a single feature model, it can be easier to create several smaller feature models, which capture the concerns of different stakeholders, and merge them later. Such smaller models can also be used as prototypes that are refined during iterative modeling (PP_5).

MO_4: Avoid complex cross-tree constraints. (P11, P20, P21, P31, I2–I7). Cross-tree constraints allow adding dependencies between subtrees of a feature model. However, complex constraints, typically in the form of arbitrary Boolean formulas, hamper comprehension, maintenance, and evolution of the model—and can make it harder to understand configuration problems. Because of that, almost all interviewees reported that they avoid cross-tree constraints or use simple ones that are typically supported by tools for feature modeling, such as *requires, excludes, or conflicts*. If constraints cannot be simplified, P21 reports that some companies capture complex constraints in the the feature-to-asset mapping by using the concept of *presence conditions* [80]. P31 reports a practice where each feature involved in a cross-tree constraint is *tagged*, to indicate that there are additional constraints affecting the selection of this feature.

MO_5: Maximize cohesion and minimize coupling with feature groups. (P1, P3, P14, P22, P24, I6). We identified the explicit structural principle that feature groups should represent related functionalities, while abstract features should be used for structuring. A high cohesion within a group and low coupling to other groups (absence of cross-tree constraints) indicates that the features belong together. P1 and P3 also state that *and* groups on a high level and *or* groups on a low level indicate high reuse (cohesion), while high *alternative* groups indicate limited reuse.

5.5 Modeling

M_1: Use workshops to extract domain knowledge. (P8, P15, I1, I3, I9). Workshops are used extensively to initiate feature modeling, and two papers, as well as two of our interviewees, report them as being the most efficient way to start. To this end, stakeholders identified in the planning phase (PP_1) as the ones with detailed knowledge about the systems are asked to describe and model variants based on their experiences. With the purpose of identifying features, the workshops should be used to determine why differences between different variants exist. The question *"Why does this difference exist?"* should be repeatedly asked until the answers converge to an abstract reason that represents a feature (I1).

M_2: Focus first on identifying features that distinguish variants. (P8, P9, P15, P27, I1, I2). According to multiple sources, it is easier for most stakeholders to describe the features that distinguish variants from each other rather than focusing on the commonalities. For example, P15 states: "*The variability of the system was therefore elicited in two ways: moderated workshops with engineers [...] to identify major differences and [...] used automated tools to understand the technical variability at the level of components by parsing existing configuration files.*" P8 argues for this principle by stating that "*Products in the same product line share a high level of commonality. Hence, the commonality space would be larger to work with than the difference space.*"

M_3: Apply bottom-up modeling to identify differences between artifacts. (P8, P15, P27, I1–I3). Different artifacts (IS_1) can be analyzed to identify the differences between existing variants. As reported by I1, I3, and I2, source code files are typically the first artifacts to be analyzed, and the analysis can be done automatically by *diff* tools (P15, I2). However, the differences are typically interpreted manually, for example, in workshops (M_1), as bottom-up modeling is the central technique for understanding extractive product-line adoption (I1).

M_4: Apply top-down modeling to identify differences in the domain. (P5, P21, I2). For a *top-down* analysis, workshops (M_1) focus

more on domain experts, project managers, and system requirements, as explained by I2: "*Top-down is successful with domain experts, more abstract features.*" The features that emerge from the top down analysis typically represent commonalities or abstract features that help with feature model structuring (MO_5).

M_5: Use a combination of bottom-up and top-down modeling. (P13, P21, I2). Due to the different results that can emerge from bottom-up and top-down analyses (M_2), it is highly recommended to combine both strategies. A top-down analysis results in more abstract (higher-level) features, while a bottom-up analysis provides insights on more fine-grained (lower-level) features. Thus, combining both strategies increases completeness and provides a checking mechanism, for example, whether all features are implemented in the systems. For instance, P13 reports that "*the feature model was built in a top-down and a bottom-up manner. [...] The user visible features became top level features, while internal features either ended up in the lower level of the feature model or in separate, more technical sub-domain feature models.*"

M_6: A feature typically represents a distinctive, functional abstraction. (P5, P8, P9, P18, I1, I3, I6, I7, I9). While some works use feature models to represent non-functional properties (e.g., performance requirements or physical properties, such as color), most sources and interviewees emphasize that features should represent functional abstractions (P5, P18, I1, I3, I7, I9). A concrete example, mentioned by I3, is not to model the *CPU type* as a feature, as it does not represent externally visible functionality—it should rather be a feature attribute. Features should be used to capture the externally visible characteristics of a system, most often by abstracting a set of functional requirements (P8).

M_7: If needed, introduce spurious features. (P12). We identified in P12 that a "*spurious feature represents a set of features [...] that are actually not offered by current software assets, which is arguably paradoxical when modeling the provided software variability.*" For example, in a phone conference system, if a language is selected for which there is no translation, then the *default language* should be used. A feature that represents all languages with Unavailable Translation (UT) can be declared in order to express the following cross-tree constraint: UT *requires* DefaultLanguage. Note that contrary to the majority of feature definitions, UT represents a functionality that the system does *not* posses.

M_8: Define default feature values. (P14, P17, I8). We found in three sources that it can be beneficial to define default feature values if the configuration space of a feature model is very large. Then, deriving a configuration becomes a *reconfiguration* problem. Interestingly, in each source that used default values, the feature modeling tool also supported *visibility conditions* for features (P17). In these sources, the default value of a feature can be modified only if the corresponding visibility condition is satisfied.

M_9: Define feature-model views. (P19, I8). Not all parts of a feature model are equally relevant to all stakeholders. Feature model views can unclutter the model representation for selected stakeholders according to their needs. P19 reports that "*not all features and associated artifacts are relevant for an individual engineer [...] we realized user-specific views by development phase and abstraction specific feature models within a hierarchical feature model*". The *abstraction specific* view, referred to in P19, is a partial configuration of one feature model that can be referred to from another feature model. A technique called *profiles*, reported by I1, uses the same mechanism to expose a subset of all features from one feature model to another.

M_{10}: Prefer Boolean type features for comprehension. (P14, P17, P22, P23, I2, I5–I7, I9). Most interviewees reported that the vast majority of features are of type Boolean. I8 is a notable exception, reporting mostly non-Boolean features, representing some more application-logic-related specifications that were moved into the feature model. However, I7 considers that the main strength of feature modeling is that it is a "*nice way of organizing configuration switches*", thus confirming the strong preference for Boolean features. We note that there are domains, such as operating systems, where high numbers of numerical features exist (P14, P17).

M_{11}: Document the features and the obtained feature model. (P4, P12, P30, I8–I10). Several interviewees explicitly emphasized the importance of documenting the feature model, while most other interviewees implicitly mentioned the importance of documentation, such as I8: "*we've put a lot of effort into extensively documenting all options and immediately document in the editor.*" This principle is related to PP_2 and PP_5, because any new terminology (i.e., features, constraints) should be continuously unified and documented (e.g., the rationale for introducing the new element). I10 mentions that the implementation itself often is the documentation, which creates problems when variants should be refactored into a platform. Therefore, I10 suggests that feature documentation and feature implementation should be simultaneous activities.

5.6 Dependencies

D_1: If the models are configured by (company) experts, avoid feature-dependency modeling. (P21, I1, I2, I5–I8). The majority of interviews suggested that in real-world feature modeling, identifying dependencies is expensive and if explicitly modeled, the maintenance of the feature model becomes complex. Moreover, the interviewees reported that the experts configuring the model are typically aware of the undeclared dependencies (e.g., I6: "*There were some cross-tree dependencies [...], but they weren't in the model (the one configuring the model needed to know them)*") and I8: "*[dependencies are] typically not modeled*"). Consequently, dependency modeling is avoided or the modeled dependencies are rather simple, for instance, *requires* and *excludes* (e.g., I1: "*Very few cross-tree constraints [...] typically requires and conflicts*"). An estimate of interviewee I3 is that, if explicitly modeled, around 50% of features would be involved in dependencies. It also seems that, in order to compensate for the absence of explicit dependencies, custom dependencies with domain-specific semantics are introduced. For example, after a new system release, a new feature can be *recommended* by another feature. We remark that the sources advising against feature-dependencies modeling either consider small to medium-sized feature models (several tens to several hundreds of features) or feature models whose configuration spaces define a relatively small number of configurations (several tens).

D_2: If the main users of a feature model are end-users, perform feature-dependency modeling. (P10, P14, P24, I4). Considering end-users instead of domain experts, we identified a principle that opposes D_1. If feature model configuration is done by end-users (related to the purpose of the feature model PP_3), or with large

feature models (with thousands of features), feature-dependency modeling should be performed. Thereby, it can be ensured that correct configurations are derived, and support for choice propagation and conflict resolution can be offered.

5.7 Quality Assurance

QA_1: Validate the obtained feature model in workshops with domain experts. (P8, P22, P24, I1). Because a feature model represents the complete domain, we found several sources that emphasize the benefits of involving various domain experts in the review. P24 states that different domain experts should participate because "*they can focus on their area of expertise, i.e., product management, architecture, or product configuration aspects. Our results further show that detailed domain expertise is required for defining the feature models [...].*" According to I1, aspects that should be discussed are "*what are the right names for the features, what are the right ways of structuring the features, try the process of first creating the new product [configuration] that never existed before [...].*" Also, it is beneficial if domain experts that have not been involved in the modeling review the feature model, to validate if the model is intuitive (P8).

QA_2: Use the obtained feature model to derive configurations. (P1, P3–P5, I1, I8). The obtained feature model should allow deriving configurations that represent variants whose artifacts were used for feature identification and feature model creation. Furthermore, we found statements that deriving new configurations that did not exist explicitly before, and verifying that they are meaningful with respect to existing implementation, is an indicator that the feature model faithfully represents the domain. Customers may be involved, as I8 reveals: "*We have a workshop with customer where we discuss how things need to be configured in detail to adhere to their domains.*" If the feature model does not allow deriving any new configurations besides the ones analyzed, this might be an indication that migrating to a product-line is not profitable.

QA_3: Use regression tests to ensure that changes to the feature model preserve previous configurations. (I9, I10). During two interviews, we found that, to ensure that an update of a feature model preserves the previously defined configurations, regression testing can be used. I9 explains that creating reference variants from the feature model helps covering the different combinations running in real-world systems, which are thoroughly tested within their continuous integration servers. This is further acknowledged by I10, who stated to perform "*testing, a lot of testing*", referring to regression testing, whenever there are variability related changes.

5.8 Model Maintenance and Evolution

Although some of the previous principles facilitate easier maintenance and evolution of feature models (MO_2–MO_5), in this section we present three additional principles.

MME_1: Use centralized feature model governance. (P13, P14, P21, P22, P26, P27, P29, I1, I4, I5). We identified in several sources that having a dedicated employee, or a dedicated team, which governs the feature model, ensures consistent and reliable evolution of the feature model. I1 stated that "*somebody [...] chief architect or whoever [...] becomes the lead product-line engineer, okay, so they really own the overall feature model.*" This is especially the case in closed environments where the enterprise has complete control over the feature model, in contrast to open-source or community-driven projects. In cases where several feature models exist, a dedicated team should maintain each model. Centralized model governance also facilitates strict access-control management.

MME_2: Version the feature model in its entirety. (P17, I4, I8, I9). Versioning individual features would probably lead to feature model inconsistencies, thus we found several advises to version the complete feature model. When a newer version of the feature model contains features that are obsolete, but are preserved for compatibility purposes, such features are marked as *deprecated* (I8).

MME_3: New features should be defined and approved by domain experts. (P22, I5, I8, I10). In order to introduce a new feature into the feature model, it is necessary to specify the feature, define how it is going to be tested, and to consider the implications of adding a new feature to the existing implementation. Consequently, a new feature should be approved by relevant domain experts. For example, I5 stated: "*we have to make sure that the work is done properly. Because [...] you can be stopped by simple technical issues, like we cannot merge back, because the branch is frozen.*"

6 DISCUSSION

As the majority of papers (cf. Tbl. 1) reports experiences of *extractive* product-line adoptions, the principles that most sources agree on relate to *bottom-up* feature modeling. Specifically, the principles agreed on most are about information sources from which features can be identified (PP_2, IS_1), feature-identification techniques (M_2, M_3), what the identified features represent (M_6, M_{10}), and what the desirable properties of the resulting feature models are (MO_1–MO_4).

Some of the identified principles are applicable only in particular modeling scenarios. For example, considering dependencies (i.e., D_1 and D_2), the choice to model feature dependencies depends on the purpose of the feature model, namely on the decision who configures the model. Moreover, some principles represent different ways of addressing the same issue (e.g., MO_3 and M_9). For this example, splitting a large feature model into smaller ones (MO_3) and defining feature-model views (M_9), both aims at separating the concerns of different stakeholders.

Choosing either principle comes with its own benefits and drawbacks. Defining views on a single feature model facilitates centralized model governance (MME_1) and avoids the need for interface models (P5, P31). In contrast, if multiple models are used, each of them will probably have a lower depth (MO_1), with less complex cross-tree constraints (MO_4), which facilitates maintenance. Despite the fact that principle MO_3 is supported by 15 sources, compared to only two supporting M_9, the reason for this might be that feature views require sophisticated tool support, which is rarely readily available. Namely, we know of only two research tools that support feature-model views [2, 42]. Although our results uncover that separation of concerns implies trade-offs, defining how to optimally separate the concerns of different stakeholders in different contexts requires further research.

Interestingly, the principles related to planning and training (PP_1–PP_6, T_1–T_3), do not have a high number of sources that support them. The majority of sources that support these principles are either industrial case studies or tool-vendor interviews. Such sources report experiences from case studies during which an external

person or organization guided the feature modeling and product-line adoption in a host company. Consequently, although feature models are considered to be an intuitive and simple modeling notation, companies must expect a certain amount of effort for training to clearly define the scope and purpose of a feature model.

The papers that we identified in our literature review (cf. Tbl. 1) report experiences on more than ten different notations for feature models. Moreover, they are concerned with 14 different application domains. We remark that among the diverse modeling notations, few were used repeatedly; namely FODA/FORM as the first notation that has been proposed (P1, P3, P7, P9, P28); and the tool pure::variants and its notation that is established in practice (P13, P25, P26, P27, P30). Similarly, there are a two prominent domains, namely automotive (P5, P19, P26, P27, P30) and industrial automation (P13, P15, P24, P25). However, none of the principles is specific to a particular notation or domain. In contrast, the principles we identified are general and applicable to any domain and feature-modeling notation.

We ordered the principles in the most reasonable sequence of steps that are needed to create a feature model, but this ordering does not represent a definitive modeling process. For example, some principles can be optional and some can be applied differently, depending on an organization's domain and context. Defining and validating a modeling process is part of our future work.

7 RELATED WORK

The most notable work that defines a set of principles for feature modeling is reported by Lee et al. [58]. Unlike that paper, we present a systematically collected set of principles from papers on industrial practices and experiences, supported by practices reported by feature-modeling practitioners. Other than Lee et al., who share their personal knowledge on creating feature models, we identified 31 papers to collect empirical evidence. Perhaps the most related papers among those we identified, regarding methodological support in terms of well-defined feature-modeling principles, are the works based on the FORM methodology [26, 47, 49, 50, 57]. The core of FORM is a layered feature model that is based on the original FODA method [46]. Besides the fact that the defined practices are based solely on the authors' experiences, we note that this research is two decades old and the set of practices does not provide guidelines for issues arising in modern feature modeling. For example, these works do not report on combining bottom-up and top-down modeling, model hierarchy definitions, and model views. Several industrial case-studies reported practices used for creating feature models with the purpose to adopt product lines [19, 34, 38, 59]. Because these case studies report experiences from specific domains and are based on specific technologies, the reported practices are rather specific. In contrast, our principles are either generally applicable, since we synthesized them from several practices, or we clearly state the context in which they apply. Other related papers are the surveys and case studies by Berger et al. [9–11, 13], which investigate the characteristics of successful industry-grade and open-source feature models. Although not primarily aiming at identifying modeling principles, these works provided insights into the characteristics of feature models, and thus helped us to understand which modeling principles will lead to such feature models.

8 THREATS TO VALIDITY

Construct Validity. To ensure correct interpretation of interview data, each interview started with an introduction where we tried to understand the project the interviewee worked with. In line with semi-structured interview methods, we allowed thorough explanations of the relevant terminology in the interviewees' domains, upon which we adjusted our question's terminology in order to detect intricate details about the modeling principles.

Internal Validity. To ensure completeness and avoid the need for forwards snowballing, the initial set of papers was identified manually by analyzing five instances of relevant conferences and workshops. To increase our confidence about completeness, we verified with an automatic search. To ensure that we can obtain insights about different sizes and usages of feature models, we selected the interviewees based on experience and the characteristics of their product lines. Finally, to ensure that we identify domain-independent, generally applicable principles, we interviewed tool-vendors and consultants with broad experience in feature modeling.

External Validity. To ensure that the identified principles can be used in arbitrary domains to create feature models with various purposes and sizes, we based the study on research papers, technical reports, interviews with practitioners, consultants, and tool providers, all of which reported experiences about diverse applications of feature modeling in practice. For the few principles that are applicable in a specific context, we state this context clearly.

Conclusion Validity. We claim that the extracted principles are general and are reasonable with regards to our analysis and scope. Our study is based on a systematic and reproducible method that relies on qualitative data obtained from literature and interviews with practitioners. To avoid misinterpretation of the collected data, we have jointly analyzed our sources, triangulated the principles between literature and interviews, and refined the principles and categories until we reached a consensus among all authors.

9 CONCLUSION

Feature modeling has received almost 30 years of attention in research and practice. However, a comprehensive modeling methodology is still missing. Towards such a methodology, we showed that recurrent modeling practices actually exist and that these can be synthesized into generalized principles. We presented a list of 34 feature modeling principles, synthesized from practices we identified in the literature and in interviews with ten feature-modeling practitioners. The presented principles cover eight categories, from training and planning the modeling activities, via identifying the relevant stakeholders and relevant information sources, to the actual modeling, model organization, model validation, as well as model maintenance and evolution. We hope that our insights help practitioners to design feature models and researchers to design methods and tools that are aligned with accepted modeling principles. As future work, we plan to synthesize an executable modeling process and validate it with our industrial partners.

Acknowledgments. Supported by Vinnova Sweden (ITEA project REVaMP^2 2016-02804, ECSEL project PRYSTINE 2018-01764) and the Swedish Research Council (257822902). We thank our interviewees as well as Robert Lindohf, Olivier Biot, and Slawomir Duszinsky for valuable feedback on earlier drafts of this paper.

REFERENCES

[1] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2010. Composing Feature Models. In *International Conference on Software Language Engineering (SLE)*. Springer, 62–81.

[2] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A Domain-Specific Language for Large Scale Management of Feature Models. *Science of Computer Programming* 78, 6 (2013), 657–681.

[3] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. 2012. Efficient Synthesis of Feature Models. In *International Software Product Line Conference (SPLC)*. ACM, 106–115.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines.* Springer.

[5] Rabih Bashroush, Muhammad Garba, Rick Rabiser, Iris Groher, and Goetz Botterweck. 2017. CASE Tool Support for Variability Management in Software Product Lines. *ACM Computing Surveys* 50, 1 (2017), 14:1–14:45.

[6] Don Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *International Software Product Line Conference (SPLC)*. Springer, 7–20.

[7] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010), 615–636.

[8] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 16–25.

[9] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *Model-Driven Engineering Languages and Systems Conference (MODELS)*. Springer, 302–319.

[10] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wąsowski, and Steven She. 2014. Variability Mechanisms in Software Ecosystems. *Information and Software Technology* 56, 11 (2014), 1520–1535.

[11] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, 7:1–7:8.

[12] Thorsten Berger and Steven She. 2010. Formal Semantics of the CDL Language. Technical Note. Available at http://www.cse.chalmers.se/~bergert/paper/cdl_semantics.pdf.

[13] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2010. Variability Modeling in The Real: A Perspective from the Operating Systems Domain. In *International Conference on Automated Software Engineering (ASE)*. ACM, 73–82.

[14] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions of Software Engineering* 39, 12 (2013), 1611–1640.

[15] Danilo Beuch. 2004. *pure::variants Eclipse Plugin User Guide.* pure-systems GmbH.

[16] Ekaterina Boutkova. 2011. Experience with Variability Management in Requirement Specifications. In *International Software Product Line Conference (SPLC)*. IEEE, 303–312.

[17] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. 2007. Lessons from Applying the Systematic Literature Review Process within the Software Engineering Domain. *Journal of Systems and Software* 80, 4 (2007), 571–583.

[18] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 625–634.

[19] Jaime Chavarriaga, Carlos Rangel, Carlos Noguera, Rubby Casallas, and Viviane Jonckers. 2015. Using Multiple Feature Models to Specify Configuration Options for Electrical Transformers: An Experience Report. In *International Software Product Line Conference (SPLC)*. ACM, 216–224.

[20] Paul C. Clements and Charles W. Krueger. 2002. Point / Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–31.

[21] Paul C. Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley.

[22] Sholom G. Cohen, Jay L. Stanley Jr., A. Spencer Peterson, and Robert W. Krut Jr. 1992. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain.* Technical Report CMU/SEI-91-TR-28. Carnegie-Mellon University.

[23] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley.

[24] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)*. ACM, 173–182.

[25] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process Improvement and Practice* 10, 1 (2005), 7–29.

[26] Eduardo S. de Almeida, Jorge C. C. P. Mascena, Ana P. C. Cavalcanti, Alexandre Alvaro, Vinicius C. Garcia, Silvio R. de Lemos Meira, and Daniel Lucrédio. 2006. The Domain Analysis Concept Revisited: A Practical Approach. In *International Conference on Software Reuse (ICSR)*. Springer, 43–57.

[27] Sybren Deelstra, Marco Sinnema, and Jan Bosch. 2005. Product Derivation in Software Product Families: A Case Study. *Journal of Systems and Software* 74, 2 (2005), 173–194.

[28] Mahdi Derakhshanmanesh, Joachim Fox, and Jürgen Ebert. 2014. Requirements-Driven Incremental Adoption of Variability Management Techniques and Tools: An Industrial Experience Report. *Requirements Engineering* 19, 4 (2014), 333–354.

[29] Deepak Dhungana, Paul Grünbacher, Rick Rabiser, and Thomas Neumayer. 2010. Structuring the Modeling Space and Supporting Evolution in Software Product Line Engineering. *Journal of Systems and Software* 83, 7 (2010), 1108–1122.

[30] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. 2012. Strategies for Testing Products in Software Product Lines. *SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–8.

[31] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[32] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 252–261.

[33] Open Network Foundation. 2015. *UML Modeling Guidelines.* Technical Report ONF TR-514. Open Network Foundation.

[34] Jesús Padilla Gaeta and Krzysztof Czarnecki. 2015. Modeling Aerospace Systems Product Lines in SysML. In *International Conference on Software Product Line (SPLC)*. ACM, 293–302.

[35] Charles Gillan, Peter Kilpatrick, Ivor T. A. Spence, T. John Brown, Rabih Bashroush, and Rachel Gawley. 2007. Challenges in the Application of Feature Modelling in Fixed Line Telecommunications. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 141–148.

[36] Martin L. Griss. 1997. Software Reuse Architecture, Process, and Organization for Business Success. In *Israeli Conference on Computer Systems and Software Engineering (ICCSSE)*. IEEE, 86–89.

[37] Martin L. Griss, John Favaro, and Massimo d'Alessandro. 1998. Integrating Feature Modeling with the RSEB. In *International Conference on Software Reuse (ICSR)*. IEEE, 76–85.

[38] Andreas Hein, Michael Schlick, and Renato Vinga-Martins. 2000. Applying Feature Models in Industrial Settings. In *International Software Product Line Conference (SPLC)*. Kluwer, 47–70.

[39] Peter Hofman, Tobias Stenzel, Thomas Pohley, Michael Kircher, and Andreas Bermann. 2012. Domain Specific Feature Modeling for Software Product Lines. In *International Software Product Line Conference (SPLC)*. ACM, 229–238.

[40] Arnaud Hubaux, Andreas Classen, Marcilio Mendonça, and Patrick Heymans. 2010. A Preliminary Review on the Application of Feature Diagrams in Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. 53–59.

[41] Arnaud Hubaux, Patrick Heymans, and David Benavides. 2008. Variability Modelling Challenges from the Trenches of an Open Source Product Line Re-Engineering Project. In *International Software Product Line Conference (SPLC)*. IEEE, 55–64.

[42] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. 2013. Supporting Multiple Perspectives in Feature-Based Configuration. *Software & Systems Modeling* 12, 3 (2013), 641–663.

[43] Takashi Iwasaki, Makoto Uchiba, Jun Otsuka, Koji Hachiya, Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda. 2010. An Experience Report of Introducing Product Line Engineering across the Board. In *International Software Product Line Conference (SPLC)*. 255–258.

[44] Hans P. Jepsen and Danilo Beuche. 2009. Running a Software Product Line: Standing Still is Going Backwards. In *International Software Product Line Conference (SPLC)*. ACM, 101–110.

[45] Hans P. Jepsen, Jan G. Dall, and Danilo Beuche. 2007. Minimally Invasive Migration to Software Product Lines. In *International Software Product Line Conference (SPLC)*. IEEE, 203–211.

[46] Kyo C. Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report CMU/SEI-90-TR-21. Carnegie-Mellon University.

[47] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. *Annals of Software Engineering* 5, 1 (1998), 143–168.

[48] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, and Kwanwoo Lee. 1999. Feature-Oriented Engineering of PBX Software for Adaptability and Reuseability. *Software: Practice and Experience* 29, 10 (1999), 875–896.

[49] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe. 2002. Feature-Oriented Product Line Engineering. *IEEE Software* 19, 4 (2002), 58–65.

[50] Kyo C. Kang, Kwanwoo Lee, JaeJoon Lee, and SaJoong Kim. 2003. *Domain Oriented Systems Development: Practices and Perspectives*. CRC Press, Chapter Feature-Oriented Product Line Software Engineering: Principles and Guidelines, 29–46.

[51] Barbara A. Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University.

[52] Charles W. Krueger. 2001. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering (PFE)*. Springer, 282–293.

[53] Charles W. Krueger. 2007. BigLever Software Gears and the 3-Tiered SPL Methodology. In *Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA)*. ACM, 844–845.

[54] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 354–361.

[55] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.

[56] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 179–189.

[57] Kwanwoo Lee, Kyo C. Kang, Eunman Koh, Wonsuk Chae, Bokyoung Kim, and Byoung Wook Choi. 2000. Domain-Oriented Engineering of Elevator Control Software: A Product Line Practice. In *International Software Product Line Conference (SPLC)*. Kluwer, 3–22.

[58] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. 2002. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In *International Conference on Software Reuse (ICSR)*. Springer, 62–77.

[59] Daniela Lettner, Klaus Eder, Paul Grünbacher, and Herbert Prähofer. 2015. Feature Modeling of two Large-Scale Industrial Software Systems: Experiences and Lessons Learned. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE, 386–395.

[60] Christian Manz, Michael Stupperich, and Manfred Reichert. 2013. Towards Integrated Variant Management in Global Software Engineering: An Experience Report. In *International Conference on Global Software Engineering (ICGSE)*. IEEE, 168–172.

[61] Jan Mendling, Hajo A. Reijers, and Wil M. P. van der Aalst. 2010. Seven Process Modeling Guidelines (7PMG). *Information and Software Technology* 52, 2 (2010), 127–136.

[62] Daniel Moody. 2009. The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering* 35, 6 (2009), 756–779.

[63] Mukelabai Mukelabai, Damir Nešić, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems. In *International Conference on Automated Software Engineering (ASE)*. ACM, 155–166.

[64] Tsuneo Nakanishi, Kenji Hisazumi, and Akira Fukuda. 2018. Teaching Software Product Lines As a Paradigm to Engineers: An Experience Report in Education Programs and Seminars for Senior Engineers in Japan. In *International Systems and Software Product Line Conference (SPLC)*. ACM, 46–47.

[65] Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. 2011. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *International Software Product Line Conference (SPLC)*. ACM, 2:1–2:8.

[66] Klaus Pohl, Günter Böckle, and Frank J. van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer.

[67] Richard Pohl, Mischa Höchsmann, Philipp Wohlgemuth, and Christian Tischer. 2018. Variant Management Solution for Large Scale Software Product Lines. In *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. ACM, 85–94.

[68] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Semantics. In *International Requirements Engineering Conference (RE)*. IEEE, 139–148.

[69] Christa Schwanninger, Iris Groher, Christoph Elsner, and Martin Lehofer. 2009. Variability Modelling throughout the Product Line Lifecycle. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Springer, 685–689.

[70] Yusra Shakeel, Jacob Krüger, Ivonne von Nostitz-Wallwitz, Christian Lausberger, Gabriel Campero Durand, Gunter Saake, and Thomas Leich. 2018. (Automated) Literature Analysis - Threats and Experiences. In *International Workshop on Software Engineering for Science (SE4Science)*. IEEE, 20–27.

[71] Steven She and Thorsten Berger. 2010. Formal Semantics of the Kconfig Language. Technical Note. Available at http://www.cse.chalmers.se/~bergert/paper/kconfig_semantics.pdf.

[72] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *International Conference on Software Engineering (ICSE)*. ACM, 461–470.

[73] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, and Jan Bosch. 2004. COVAMOF: A Framework for Modeling Variability in Software Product Families. In *International Software Product Line Conference (SPLC)*. Springer, 197–213.

[74] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. 2004. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *International Software Product Line Conference (SPLC)*. Springer, 34–50.

[75] Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung, and Robert Heinrich. 2016. Challenges in the Evolution of Metamodels: Smells and Anti-Patterns of a Historically-Grown Metamodel. In *Workshop on Models and Evolution (WME)*. 30–39.

[76] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* 47, 1 (2014), 6:1–6:45.

[77] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. 2014. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming* 79 (2014), 70–85.

[78] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action*. Springer.

[79] Arie van Deursen and Paul Klint. 1998. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice* 10, 2 (1998), 75–92.

[80] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *International Conference on Software Engineering (ICSE)*. IEEE, 178–188.

[81] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *International Conference on Evaluation and Assessment in Software Engineering (EASE)*. ACM, 38:1–38:10.