Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Sweden

Damir Nešić
KTH Stockholm
Sweden

Salome Maro
Chalmers | University of Gothenburg
Sweden

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

Jan-Philipp Steghöfer
Chalmers | University of Gothenburg
Sweden

ABSTRACT

Highly configurable systems are complex pieces of software. To tackle this complexity, hundreds of dedicated analysis techniques have been conceived, many of which able to analyze system properties for all possible system configurations, as opposed to traditional, single-system analyses. Unfortunately, it is largely unknown whether these techniques are adopted in practice, whether they address actual needs, or what strategies practitioners actually apply to analyze highly configurable systems. We present a study of analysis practices and needs in industry. It relied on a survey with 27 practitioners engineering highly configurable systems and follow-up interviews with 15 of them, covering 18 different companies from eight countries. We confirm that typical properties considered in the literature (e.g., reliability) are relevant, that consistency between variability models and artifacts is critical, but that the majority of analyses for specifications of configuration options (a.k.a., variability model analysis) is not perceived as needed. We identified rather pragmatic analysis strategies, including practices to avoid the need for analysis. For instance, testing with experience-based sampling is the most commonly applied strategy, while systematic sampling is rarely applicable. We discuss analyses that are missing and synthesize our insights into suggestions for future research.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; • **General and reference** → *Empirical studies*;

KEYWORDS

Highly configurable systems, software product lines, analysis

ACM Reference Format:
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.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3238201

In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238201

1 INTRODUCTION

Engineering a highly configurable system allows addressing varying customer needs, reducing time-to-market of new system variants, and experimenting with new ideas. Large configurable systems can easily exhibit thousands of configuration options, leading to almost infinite configuration spaces. Software product lines (SPLs) [1, 20], software ecosystems [12, 15], and personalization-capable systems—especially in the automotive, avionics, telecommunication or power-electronics domain—are common examples of highly configurable systems. The Linux kernel [14] boasts around 15,000 of configuration options, supporting different hardware architectures, software features or runtime environments ranging from Android phones to large supercomputer clusters.

Engineering highly configurable systems is challenging due to variability—the number of configurations and system variants grows exponentially with the number of configuration options. Consider the popular analogy by Krueger et al. [35], where a system with 320 Boolean, non-constrained configuration options has more configurations than estimated atoms in the universe. Over the last decades, many development techniques for highly configurable systems have been conceived, mainly in the field of product line engineering. While its development concepts have well been adopted in industrial practice—consider the product line hall of fame

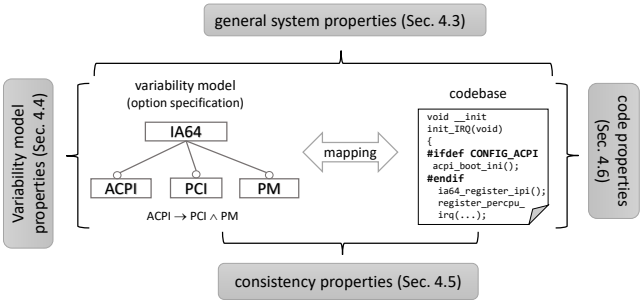

Figure 1: Architecture of a highly configurable system and categories of typical properties to assure or analyze

(splc.net/hall-of-fame) and case studies [61, 69]—this is much less clear for product-line *analysis* techniques.

Figure 1 shows a typical architecture of highly configurable systems. It consists of source code with configuration mechanisms (here, preprocessor directives), a specification of configuration options (a.k.a., *features* [10]) and their dependencies in a variability model (here, a feature model [14, 30]), and a mapping between both (typically the build system). Properties to assure relate to: the whole system with all configurations (*general system properties*, e.g., reliability or performance), the codebase (*code properties*, e.g., option scattering or nesting), the option specification (*variability model properties*, e.g., absence of dead options), and the mapping (*consistency properties*, e.g., absence of dead source code).

Analysis techniques for highly configurable systems differ from traditional analyses. Even though, the latter can be used on individual variants (e.g., for optimization) when the system is configured by the system vendor, using traditional analyses is not sufficient to detect errors that pertain to all possible variants, especially when customers configure it. Yet, even when the vendor is in control of the configurations, applying traditional analyses, such as testing, still requires effective configuration-sampling strategies [21, 53].

Creating analysis techniques for highly configurable systems by lifting single-system analyses (e.g., model checking or type checking) [47, 66], has received substantial attention over the last decade. Such *variability-aware analyses* [66] elicit a certain system property *for all possible configurations*. A recent survey [66] identified 123 analyses from the literature, including lifted type-checking, static-analysis, and model-checking techniques. Alone for feature models, an eight year old survey [8] identified—in 53 publications—30 different analyses, realized upon a multitude of solvers with different logical representations (e.g., propositional logic).

Although a vast variety of such analyses has been proposed in the literature [6–8, 66], it is largely unknown whether and how these are used. Specifically, we lack empirical data whether the proposed analyses for highly configurable systems are adopted in practice, whether they address actual needs or find errors, or what analysis strategies are actually applied.

We present a study on the needs and practices for analyzing highly configurable systems in industry. We combined a survey with 27 employees—of companies ranging from less than ten to over 200 employees working on highly configurable systems ranging from less than 25,000 lines of code to over one million lines of code, containing between ten to over 10,000 configuration options—with in-depth interviews of 15 survey participants. Our study design relied on categorizing analysis techniques from the literature and identifying properties analyzed by them; we used these to elicit the need for and the severity of analyzing the properties. We also elicited industrial practices. Since it is intrinsically difficult to objectively understand the real practices and map them to the state of research, we triangulated results from the survey and interviews, steering the latter based on the survey responses, and carefully analyzing the results iteratively.

Our research questions are:

- *RQ1: What are important properties of highly configurable systems that should be analyzed?* We elicited the perceived severity and reasons for analyzing the properties we identified from the literature (in the categories shown in Figure 1) and those expressed by the practitioners.
- *RQ2: What are industrial analysis practices?* We asked our participants about established (textbook) analysis tools and techniques, and additional practices they apply. We also dug deeper into specific ones to understand them qualitatively.

We contribute: (i) empirical data on the needs and state-of-practice of analyzing configurable systems, (ii) synthesized insights organized in categories inspired by the architecture of highly configurable systems and a classification of existing analyses from the literature, (iii) a discussion of our study results and their implications for researchers and practitioners, and (iv) a replication package with further study details in an online appendix [65].

We proceed by discussing the background and related work on the analysis of highly configurable systems in Section 2, followed by the design and data analysis procedure for our survey and interviews in Section 3. We report results in Section 4, organized as insights covering the perceived relevance of various analyses and reported challenges. We summarize the impact of our study and propose possible directions for research in Section 5. We discuss threats to validity in Section 6 and conclude in Section 7.

2 BACKGROUND AND RELATED WORK

We introduce highly configurable systems and software product lines, to discuss analyses that have been proposed in the literature, as well as works related to our study.

2.1 Highly Configurable Systems

Highly configurable systems offer configuration options (a.k.a., features [10, 18, 36] or calibration parameters) that can be of different types (e.g., Boolean or integer). Options are used in implementation artifacts (e.g., source code, test cases, requirements), either to parameterize functionality or control variation points. The latter can be realized using different mechanisms, including preprocessor directives for conditional compilation (e.g., *#IFDEF*s), option/feature toggles (e.g., simple *IF*s), build systems or component frameworks.

The available options and their dependencies need to be declared, either in a dedicated variability model (explained shortly), a textual configuration/properties file or a database; or informally, such as in a spreadsheet. Models can be input to a configurator tool that guides users to a valid configuration, which binds variation points or parameterizes the system. We generally refer to option specifications as *variability models*.

Software product lines—portfolios of system variants in a specific domain—are typically realized as a highly configurable system. The options (called *features* or *decisions*) are described in a dedicated variability model, such as a feature [11, 14, 30] or a decision [23, 60] model. Note that we use the terms *option* and *feature* synonymously. Popular, feature-model based, configurator tools are pure::variants, Gears, FeatureIDE or the Linux kernel configurator [5, 14]. The research field of product-line engineering has conceived a large number of analysis techniques.

2.2 Analysis Techniques and Tools

Analysis techniques for highly configurable systems have been studied for almost two decades by now.

Dynamic Analyses. As we will show, one of the most popular dynamic analysis for highly configurable systems in industrial practice is testing [25, 39, 51]. Several testing techniques for configurable systems are summarized by Engstrom et al. [25], with rather few that have been evaluated through experiments or industrial case studies, including techniques for integration and system testing [56] as well as performance testing [55].

Testing is always specific to one configuration, Since it is often impossible to test all configurations, configuration sampling strategies have been proposed [21, 28, 45, 53], including random sampling or more systematic sampling strategies that try to enable each option or certain option combinations (e.g., n-wise feature interaction sampling) [53].

Static Analyses. In order to effectively analyze all possible configurations, a class of static analyses, called *variability-aware analyses*, has been developed over the last decade, typically by lifting single-system analyses, such as dataflow analysis [42, 58], model checking [37] or deductive verification [68]. Also defect prediction using machine learning was proposed [54]. According to Thüm et al's survey [66], these analyses can check properties of the whole system (*family-based* analysis) or of individual options in isolation (*feature-based* analysis). Examples of properties are type-safety [31], performance [27], temporal properties [19], or absence of unwanted feature interactions [2, 3, 16, 29]. Unlike Thüm et al.'s survey [66], which identifies the state-of-the art, our study aims at presenting the state-of-practice—offering insights about the adoption and potential challenges when using existing variability-aware static analyses.

Variability Model Analyses. A vast number of analyses has been proposed for variability models (especially feature models) as well as analyses to check consistencies between variability models and implementation artifacts. Recall Benavides et al.'s survey [8]. Among the 30 different analyses on feature models are, for instance, checking model satisfiability (i.e., at least one configuration exists), checking validity of configurations, counting or enumerating configurations, and finding dead features. The majority of these analyses has been evaluated on rather small, artificial models [14, 33]. Our study is complementary to Benavides et al.'s survey [8], since we match these analyses to industrial needs. Finally, there are also analyses that reason about feature-model edits [67], as well as analyses that elicit quality properties on models using various model metrics [9], such as the maintainability of a model [4].

Consistency and Code Analyses. Furthermore, to prevent inconsistencies between variability models and implementations artifacts, consistency-related analyses have been proposed [38, 48, 50, 57, 63, 64, 70], many relying on SAT solvers, for instance, to check that code constraints are consistent with variability-model constraints (e.g., to find dead code [63, 64]), but there are also analyses using theorem proving [59] or model checking [26]. Some of these have been applied in practice [34]. Furthermore, assuring structural code properties is also relevant. Studies on such properties focus mainly on the structure of variation points [40, 41, 71], but do not elicit needs or practices related to source-code analyses.

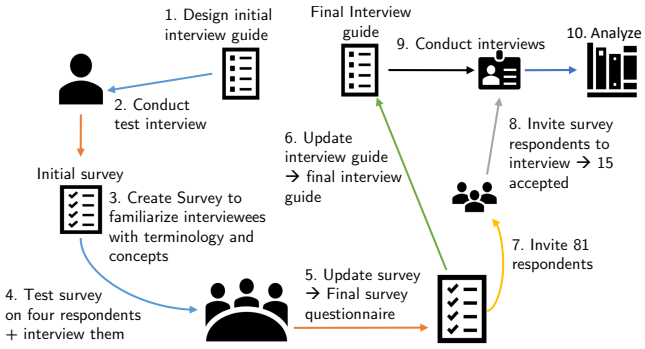

Figure 2: Study design

Analysis Tools. As shown in a recent survey [46], many of the analyses above are tool-supported, but all are research protoypes. Some industry-scale evaluations of specific tools have been conducted on industrial systems, such as consistency checking [70]. Other evaluations rely on open-source systems, such as the Linux kernel or other systems software [21, 32, 43]. Even though these evaluations show the effectiveness of a technique, they do not investigate its practical importance.

3 METHODOLOGY

Our study design aimed at obtaining insights about analyses that are applied in practice and about the needs of practitioners. Thereby, we also strive to obtain insights about the relevance of analyses that have been proposed in the literature. Recall that most of these proposed analyses are automated, yet, we do not narrow our scope to these only, but also consider manual analyses. To this end, we studied literature surveys, specifically Chastek et al. [17], Benavides et al. [8], and Thüm et al. [66]. We designed a structure, both for the survey questionnaire and the interview guide, based on a classification of analysis techniques for product lines, largely inspired by Chastek et al. [17] and the architecture of highly configurable systems, as we illustrated in Figure 1. This structure comprised: (i) analysis of general system properties, (ii) testing practices and challenges, (iii) analysis of the variability model, (iv) consistency analysis between variability model and implementation artifacts (e.g., code); (v) analysis of implementation artifacts. The typical properties, analyses, tools, and challenges identified in the literature were distributed in these categories. Specifically, instead of asking about particular analysis techniques, which are likely unknown to the participants, our focus was on properties regarding the five categories. For instance, we extracted eight (in our opinion) most relevant properties of variability models (explained shortly in Section 4.4), for which analyses had been proposed, from Benavides et al.'s literature survey [8].

From experience, we expected many terms from the literature to be unknown to industrial participants, so we developed the questionnaire and interview guide iteratively with five industrial test participants. Figure 2 shows our methodology. First, we designed an interview guide based on the categories and conducted a pilot interview. The experiences led to the first version of the survey questionnaire, which we then tested with four industrial

participants from the automotive, power electronics, and camera systems domain. We then continued refining the terminology and slightly adjusting the scope. For instance, we replaced product-line terminology with more common terms (e.g., *product line* became *highly configurable system*, variability model became configuration specification). Furthermore, we added testing (and configuration sampling), as the most frequently expressed analysis technique by the pilot participants. We conducted further pilot interviews with these participants, learning that a survey questionnaire before the interview helps to familiarize interviewees with the terminology and structure, and so allowed going deeper into certain aspects, reducing unnecessary clarifications.

3.1 Survey Questionnaire

Design. We designed the survey questionnaire to take no more than 25 minutes to complete. It first elicited brief respondent ethnographics (e.g., professional background or experience), then general characteristics of the configurable subject system (e.g., size, domain, and architecture), and then covered the five analysis categories in detail. Finally, we asked about general challenges. The whole questionnaire is available in our online appendix [65].

All questions were optional to allow respondents to leave out those questions they did not have the right technical background in or did not know sufficient details about. Most questions were closed-ended; for some we asked for additional elaborations. The questions about the importance of analyzing specific properties used a four-point scale asking participants to rate the importance of analyzing the property: *not necessary*; *nice to have, but not critical*; *critical*; and *don't know*. This was followed by questions on which property is assured and how.

Respondents. Trading the number of responses for an increased confidence in them, we sent 81 invitations to our industrial partners (manufacturers, consulting companies, tool vendors), also asking them to kindly forward the invite to colleagues or collaborators.

We received 54 responses (response rate: 66.7 %). 43 % were complete responses and 57 % were partial. From the latter, we disqualified 50 % for the lack of responses to any of the analysis sections (even when ethnographics, system characteristics, and challenges were filled in), so only 27 (50 %) of the 54 respondents were considered. This indicates that many developers, who work on highly configurable systems, are not familiar with respective analyses at all. Excluding these enhances the confidence in our results. In the remainder, we refer to individual respondents as S1, S2, and so on.

3.2 Interviews

Design. The follow-up, semi-structured interviews relied on an interview guide largely following the questionnaire structure. We first verified basic ethnographics with the interviewee and characteristics of the subject system, which helped getting into the interview. For the main part, our strategy was to first briefly analyze the survey responses for each section and then going into depth about certain properties perceived important or other aspects we found relevant. The sections and our preliminary analysis provided a structure, but we let the interview flow freely. This way, we obtained additional insights into analysis needs, practices, and challenges. For instance, we asked why a certain property was perceived as critical, why it was analyzed or not, and what would be needed to do so (e.g., presentation or scalability requirements). We also engaged the interviewee to discuss the feasibility of applying existing analyses she might not be aware of.

Interviewees. We invited 23 survey respondents who completed all questionnaire sections for interviews, among which 15 accepted. The interviews were conducted in person or via phone, typically lasting around one hour or slighly longer, totaling to 17 hours altogether. The interviews were recorded, stored securely, and transcribed. In the remainder, we refer to individual interviewees using I1, I2, and so on.

3.3 Data Analysis and Interpretation

We analyzed the survey quantitatively by creating diagrams and manually aggregating responses to questions, and qualitatively by inspecting responses to the open-ended questions. For obtaining trends about the importance of certain properties, we created violin plots (note, the diagrams in the remainder omit the "don't know" responses) by assuming a continuous scale, carefully interpreting these trends, triangulating with qualitative data. Violin plots indicate where most answers of the four-point scale are; for instance, if most answers regarding the relevance of a particular property is towards critical, the plot will be thick at that point. The black dot indicates the median. To analyze the interviews, we used open coding [62] (a method from grounded theory) [22]: three authors coded the interviews iteratively, continuously discussing and refining the codes, but also having a coding workshop with all authors, until the codes were finalized. Based on the codes, we triangulated results from the survey and interviews; for instance, crossing information that a certain system property is perceived as critical, with the details from the respective interview. We report the results by first providing details about the systems, survey respondents, and interviewees, and then by using a narrative style for the main findings. While the relatively low number of survey responses and interviews did not allow statistical hypothesis testing, we cross-checked for support in the survey data when interesting conjectures emerged during analysis.

4 RESULTS

We present our results by first describing our survey respondents, interviewees, and their configurable systems. Tables 1 and 2 summarize the latter two. We then provide findings related to our five main analysis categories. We limit descriptive results and exact statistics, which can be found in our appendix [65], but instead highlight and discuss our main findings. Table 3 summarizes the needs and practices observed.

4.1 Survey Respondents and Interviewees

The majority of survey respondents comprised developers (70 %), followed by software architects (48 %) and team leaders (44 %). Almost half had over ten years of experience working with highly configurable systems, a third between five and ten years. The ethnographics of our interviewees largely resembled those of the survey respondents (only the average experience deviated slightly).

Table 1: System characteristics stated by survey respondents

domain		system size (LOC)	
automotive	27 %	< 25,000	4 %
industrial automation	20 %	25,000–50,000	4 %
aerospace&defense	12 %	50,001–100,000	12 %
consumer electronics	8 %	100,001–500,000	19 %
telecommunication	8 %	500,001–1,000,000	27 %
office	8 %	> 1,000,000	35 %
other	20 %		
team size		**number of conf. options**	
< 10	27 %	10–50	15 %
10–50	39 %	51–100	31 %
51–100	19 %	101–500	12 %
101–200	4 %	501–1,000	8 %
> 200	12 %	1,001–10,000	19 %
		> 10,000	15 %
configurable artifacts		**option specification**	
source code	89 %	configuration file	44 %
models	70 %	variability model	41 %
test cases	52 %	in source code	41 %
runtime configuration file	52 %	in configurator tool	33 %
requirements	48 %	database	22 %
other	19 %	other	11 %
options with dependencies		**variability mechanism**	
none	20 %	option/feature toggles	67 %
1–25 %	36 %	configurable build system	67 %
26–50 %	20 %	conditional compilation	48 %
51–75 %	12 %	other	30 %
76–100 %	12 %	component/service framework	26 %

4.2 System Characteristics

Table 1 summarizes our subjects' highly configurable systems, which cover a wide range of domains, mainly automotive, industrial automation, and aerospace and defense, among others. The largest systems (15 % of them) exhibit over 10,000 configuration options. As expected, the most frequent configurable artifact is source code (89 %), followed very frequently by models (70 %), but half of the survey respondents also mentioned test cases, requirements, as well as runtime configuration files.

Since 70 % of the survey respondents stated that they configure models, we asked our interviewees about the kinds of models. These were domain-specific models, such as aircraft simulator models (I4) in the aerospace and defense domain, and Simulink models in the automotive domain (I15).

Close to a quarter of the respondents does not declare dependencies between configuration options, while the majority of those who declare them only does so for 26–50 % of the options. For instance, interviewee I2 stated that only few dependencies, mostly optional, are modeled to reduce complexity. These dependencies are domain, software, and hardware constraints (I1). For the systems where dependencies are partly or not at all modeled, our interviewees explained that this is because: (i) undeclared dependencies do not cause problems due to the small number of variants (I3, I6, I13), (ii) relationships for some options are determined at runtime (I4), (iii) there is lack of time and money, since setting up rules and calculating possibilities or consequences for certain rules takes too much time (I4), or (iv) there are too many implicit dependencies in the source code (I5) that cannot be expressed.

4.3 Analysis of General System Properties

Reliability (67 %), performance (65 %), absence of feature interaction (65 %), behavioral correctness (62 %), and safety (44 %) are perceived as the most important system properties. As shown in Figure 3, assuring these properties for each configuration is primarily perceived as critical, unlike properties such as security or maintainability. While the differences are small and likely not statistically significant, the reported percentages confirm the expected relevance of these properties, which have been considered in the literature. Furthermore, for example in the case of reliability, two survey respondents (S1, S6) and two interviewees (I2, I5) perceived reliability as encompassing other properties such as safety and security. For instance, a system that crashes due to some "unforeseen combinatorial path through the code" is deemed unreliable, insecure, and not meeting safety standards (S1).

Table 2: Interviewee ethnographics

	role	exp.[1]	domain	size[2]
I1	Team leader, Domain expert, System architect	5-10	industrial automation	100K-500K
I2	Department lead	> 10	industrial automation	> 1M
I3	Developer, Team leader, Domain expert, Software architect	> 10	aerospace & defense	100K-500K
I4	Developer, Configuration Manager	> 10	aerospace & defense	> 1M
I5	Team leader, Software architect	5-10	automotive	> 1M
I6	Developer, Domain expert, Software architect	> 10	telecommunication	2.5M
I7	Project manager, Product manager	5-10	banking/insurance	500K-1M
I8	Developer	5-10	automotive	50K-100K
I9	Developer, Modeler, System architect	5-10	automotive	
I10	Developer, Software architect	5-10	automotive	500K
I11	Developer	5-10	heat pumps	
I12	Developer	> 10	automotive	
I13	Developer, Team leader	> 10	automotive	1.2K
I14	Developer, Domain expert, System owner, System architect	> 10	network cameras	50M
I15	System owner	5-10	automotive	> 1M

[1] experience with the system in years [2] system size in lines of code

We observed strong performance requirements, which are confirmed to be challenging to assure for all configurations. Their violation affects safety and can have serious consequences (I5, I12, S6). I12: *"We have time-critical performance [requirements], deviation of 2ms is a fail."* S6: *"Our system is used to test telecom equipment under stress. It is critical that the performance report from our system is accurate given any configuration or combination of configurations."*

With the exception of cost constraints, all other system properties (shown in Figure 3) we asked about are assured by, at least, 29 % of the survey respondents, who confirmed assuring one or more properties. As expected, reliability is assured by the majority (79 %), followed by behavioral correctness (71 %), performance (50 %), and safety constraints (43 %). Safety was largely related to *legislation*, especially in the industrial automation and automotive (e.g., carbon emission or vehicle safety) domain (I2, I10).

None of our survey respondents or interviewees assured cost constraints, but some have expressed the lack of and the need for tools that would help in cost analysis, for instance, tools to analyze the cost of adding or removing features (I11, S17), or analyze feature resource-consumption (I1, I14).

Testing and manual reviews are the most prominent analysis practices. All survey respondents in fact do testing (100 %) for assuring relevant system properties, followed by manual reviews

Table 3: Overview of reported needs and associated practices

reported analysis needs	reported analysis practices[1]
analysis of general system properties (Section 4.3)	
analyze reliability, performance, behavioral correctness, safety, unwanted feature interactions	testing (100 %), manual reviews (67 %), formal methods (20 %)
configuration sampling for testing	experience-based sampling (67 %), systematic sampling (13 %), random sampling (7 %)
alleviate need for complex analyses	modularization of features to enable feature-based testing using unit tests or to enable feature-interaction testing using integration tests; modularization limits test runs
targeted test case selection	–[2]
traceability between failed test cases and configuration options	exploit recorded database snapshots of configuration option values
variability model analyses (Section 4.4)	
analyze configuration validity, model satisfiability, number of configurations, validity of edits	manual review (64 %), scripts (55 %), testing (18 %), configurator tool (18 %)
analyze impact of quick fixes and small changes on variability model	–[2]
reduce test effort	analyse possible configurations
identify technical debt, lack of coding discipline or missing knowledge	analyse anomalies of variability model (redundant constraints or dead options)
consistency analyses (Section 4.5)	
assure consistency of the variability model	manual review (80 %), automated techniques (40 %), testing (30 %)
fast, flexible automated consistency checking embedded in continous deployment toolchain	–[2]
integration of tools handling different artifacts	manual work (e.g., snapshots and manual import/export)
source-code analyses (Section 4.6)	
variability-aware static analysis tools	non-variability-aware static analysis tools (80 %), manual reviews (70 %), proprietary scripts (40 %)
reduce need for complex analyses	adhere to coding standards (90 %), adhere to company-specific style guides (80 %)
avoid deeply nested ifs and #ifdefs	code reviews
feature interaction analyses (Section 4.7)	
show absence of feature interactions	manual reviews, extensive testing
analyze feature interactions in loosely coupled architectures	–[2]

[1] percentage of survey respondents stating the practice (where applicable)
[2] no practice reported

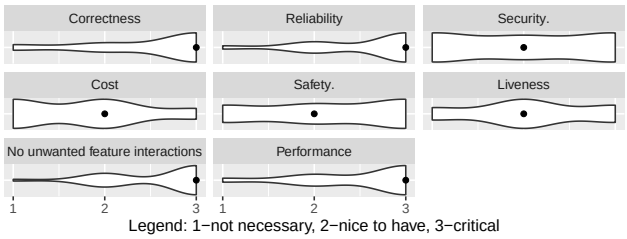

Figure 3: Importance of assuring general system properties

(67 %). Only 20 % use formal methods, such as model checking. The manual review process usually entails manual inspection of one or more artifacts by the domain experts, for instance, I12 reports that absence of feature interaction is assured by manually comparing feature specifications, and trying to come up with potentially unwanted interactions.

70 % of our survey respondents use regression testing. A practice here is to perform regression testing only for a few selected main configurations (I2), usually 3–4, because regression tests can be time-intensive (two days in the case of I2) and the results are analyzed manually by developers. On the other hand, I6 stated that their regression tests are run on the stable release of the main branch and are intended to test that *"all the features in the stable branch still work reliably."*

Additionally, unit and integration testing are also used (I1, I6, I11). For I6, unit tests are performed by function developers and designers to verify user-visible functional requirements, after which integration testing is performed for components to include testing of "non-functional requirements." Similarly for I11, test scripts are written for features, when activated and when disabled, for individual modules, and for integrated modules I11: *"We have test scripts for both the small modules and also all modules together in the main module [...] But when we implement a new feature that is configurable, then we should do a test script for the feature that it works as it should and we should also do a test script that doesn't do anything when it's not configured."*

Experience-based configuration sampling is dominant, while systematic sampling is usually not applicable. 87 % of survey respondents sample configurations by considering known feature interactions, largely based on developer experience (67 %). Only 13 % use systematic sampling (e.g., sample all combinations of two options), and 7 % use random sampling.

The reasons why systematic sampling is not used are: (i) there are too many meaningless combinations to test (I5) and (ii) important configurations are usually known (I1, I2, I3, I6). Since most companies rely on testing (see above), experience is used to sample relevant configurations. While declaring dependencies in a variability model to rule out meaningless combinations could reduce the configuration space among which to systematically sample, even this would leave too many meaningless configurations. Moreover, declaring such dependencies is already perceived as challenging owing to the high effort required to do so. I5: *"In general, that's one of the concerns when the modeling is being done, the effort to foresee all the possible constraints and dependencies sometimes is too high to have really any effect."*

We dug deeper into the kind of experience used for sampling. According to our interviewees, it comes from: typical use cases or a representative set of configurations that customers use (I1, I2, I3, I6), standard or critical configurations for a company (I1, I2), knowledge of failures that occur in the after-market (I2), and relevant test cases frequently executed based on developer knowledge of what might go wrong (I5). For instance, S3: *"Our regression test are based upon the automated version of our qualification tests. Some parts have been made specifically to catch some typical issues we have seen."*

In some cases, combinations of these experiences are used. S22: *"[...] includes both configuration options (V8 engine, long-haulage truck, etc.) and testing conditions (hot climate, mountainous terrain, etc.). [...] the testing department would usually define some configurations/tests based on experience to increase testing coverage."* It is also possible to combine systematic and experience-based sampling due to a very large configuration space. S6: *"We try to test all combinations (automated tests), but usually some combinations are not*

possible due to large variability, therefore, we often prio some tests to automate, and that is usually based on experience."

Feature- and family-based analyses are used, but in a very limited way. Different strategies are employed to avoid sophisticated analyses. Modularization of features was stated by 70 % of survey respondents as one such strategy. Three interviewees (I1, I6, I11) use *unit testing* for individual modules, and *integration testing* for integrated modules. I1: *"We have two approaches. One is [...] to cover feature by feature that we make sure that it works [...] as intended. And then we have what we call confident testing where we more look at the applications where you have bundles of features that are working together. [...] not necessarily covering all the requirements of every feature but covering [...] the interplay between the features [...]"* Based on the classification of analysis strategies by Thüm et al. [66], we found these to be pragmatic cases of feature-based analysis, where configuration options are analyzed in isolation, and family-based analysis, where interactions of configuration options are analyzed together. However, among our participants, this only works for coarse-grained features that are modularized. We found no evidence of any participant being able to do this for cross-cutting or more fine-grained features.

Effective testing requires targeted test case selection and traceability. The needs with respect to testing can be roughly categorized into test case selection and traceability. The former is separate from sampling strategies (where configurations are selected), since it pertains to the selection of the *relevant test cases* targeting different code qualities such as criticality or safety (I1), as well as selecting test cases that target specific features (I5). This is often challenging because tests themselves are rarely configurable (I2). Traceability is relevant for understanding which configuration options were selected when a concrete test case failed (I3, I5). This can be addressed by using a database snapshot of configurations and their values to identify which configurations are affected by bugs reported under a specific customer configuration (I5).

4.4 Variability Model Analyses

Satisfiability of a variability model and validity of a configuration are perceived as the most critical properties. Recall that we elicited the eight most relevant variability-model properties (cf. Section 3) for which analyses have been conceived in the literature [8]. As Figure 4 shows, only two properties are perceived as critical, by the median of respondents: *validity of a configuration*—an existing configuration adheres to the variability model (85 %)—and *satisfiability of the model*—at least one configuration exists for the model (56 %).

We asked interviewees why these two properties are deemed more critical despite them being available in many configurator tools. They stated that these two properties are most critical due to quick fixes or one-off changes made to the variability model for some customers (I2). Such changes have the potential to easily invalidate the model, resulting in the system not working at all (I5). Consequently, expensive and time-consuming (and reportedly sometimes financially risky) rework has to be done to ensure that the configurable system works (I5, I7).

The properties of the variability model are mostly assured through manual reviews. The survey responses show that only four of the properties from Figure 4, excluding those related to edits, are assured: (i) *validity of a configuration* and (ii) *satisfiability of a specification*, both assured by 85 % of the respondents, (iii) *number of possible configurations* by 33 %, and (iv) *list of all possible configurations* by 17 %. The validity of variability-model edits is assured by 33 % of the respondents. The majority (64 %) assures these properties through *manual reviews*, 55 % use *scripts* (e.g., to collect statistics on configuration options and their references), 18 % use *testing* (e.g., regression testing), and only 18 % use the capabilities of *configurator tools*. Notably, I7 uses model checking to assure the validity of configurations. Since only 28 % of the survey respondents stated to specify configuration options in a configurator tool, and that over 85 % specify them directly in source code or use a textual configuration file, it is not surprising that these properties are mostly assured manually or through tests.

Change-impact analysis of model changes is important, but should not be limited to the model. While all interviewees often make edits to their variability models, e.g., adding a new configuration option or changing dependencies, surprisingly, analyzing if such edits lead to more configurations (generalization), fewer configurations (specialization) or maintain the number of configurations (refactoring)—typical analysis reported in the literature [8, 67]—is not perceived as important. A minority of survey respondents perceives this analysis as critical (20 % for generalization, 17 % for refactoring, and 16 % for specialization), while the majority perceive it as nice to have (58 % for refactoring, and 48 % for specialization and generalization) or unnecessary (28 % for specialization, 24 % for generalization, and 17 % for refactoring). Interviewee I5 explained that such analysis is not critical as long as relevant configuration options get activated and the system operates as expected. Two interviewees, however, perceived this analysis as critical, since it is necessary to gauge the impact of the change (I10) or to exhaustively test a new addition (I15).

From such explanations, we learned that analyzing the impact of edits to variability models means more than knowing how the number of configurations is affected. It is more important to analyze the impact on other artifacts or the whole system. Indeed, five interviewees requested specific analyses: (i) assess how a model-change impacts existing configurations (I10, I15), implementation artifacts, and system complexity (I1, I14), (ii) assess impact of option name change without changing its structure and vice-versa (I1), (iii) assess impact of splitting an option into two or more features (I1), or (iv) easily understand the impact of switching options on and of (S16). None of these analyses currently exists.

Finally, one interviewee explained the need for enhanced change-impact analysis as a tradeoff of adopting a highly configurable system (instead of using clone&own [24] for realizing variants. I1: *"We do not clone, but the complexity moves from doing so to the [variability model] [...] where a need for a change potentially affects all previous (as well as future) configurations, and it can be hard to know the impact and to 'fix' previous configurations."*

Knowing the list and number of possible configurations helps reducing test effort. Although not considered critically important

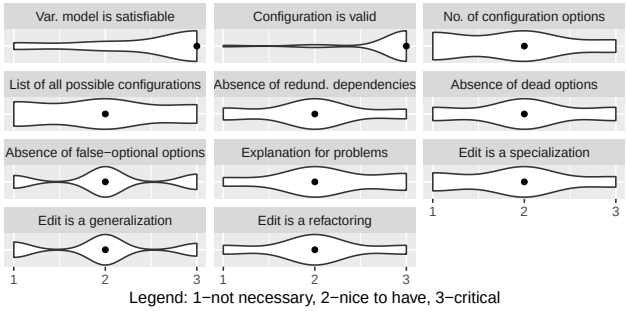

Figure 4: Importance of assuring various properties of the variability model

by the majority of survey respondents, the *list of possible configurations* and *number of possible configurations* were considered critical by 23 % and 15 %, respectively. As we learned, they help reducing test effort. I12: *"Every new combination of parameters that we do results in about three month's time of testing. So, everything we can do to keep down the amount of configurations is gold for us. So, I would say that, that is not only nice to have, that is critical."*

Anomalies of a variability model (redundant options, false optional options, and dead options) indicate a lack of discipline in adherence to coding guidelines and a lack of knowledge of a system. Since it is perceived critically important by 21 % of our survey respondents, we asked the respective interviewees why they perceived the analysis of anomalies of a variability model as critical. Two interviewees stated that redundant constraints and dead options indicate: a lack of discipline in adhering to coding guidelines (I6), missing knowledge of one's own system (I1), and that false options indicate a bad smell or technical debt (I1).

Business priorities and the combinatorial explosion determine the criticality of addressing anomalies of a variability model. While anomalies of a variability model were perceived as critical by some interviewees, others did not share this view. Two (I2 and I12) stated that they do not consider anomalies, such as dead options, because: (i) the top management's business priorities might prefer investing time and money into adding new features over addressing dead options, for instance, and (ii) due to a large configuration space caused by the combinatorial explosion, addressing anomalies of the variability model might introduce new bugs that might take considerable effort to fix.

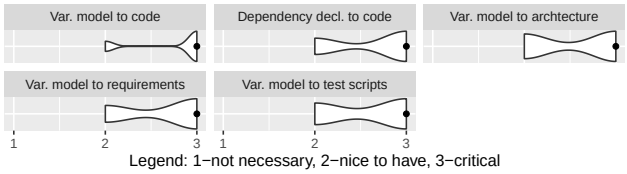

Figure 5: Importance of assuring consistencies between variability model (especially constraints) and other artifacts

4.5 Consistency Analyses

As Figure 5 shows, all elicited consistency properties were merely perceived as critical by the survey respondents: *variability model to source code* (76 %), *dependency declarations to source code* (58 %), *variability model to requirements* (56 %), *variability model to architecture specification* (48 %), and *variability model to test scripts* (47 %).

Consistency between variability model and source code is seen as most critical. Six interviewees explained that such inconsistencies lead to variants that do not implement the selected configuration or more than the configuration specifies. While research on assuring such consistency exists [38, 49, 52, 57, 64], such approaches typically need substantial adaptation and calibration effort. Instead, according to I14, effective solutions should: be more informative or much faster than testing, be flexible and allow inconsistencies during intermediary stages of development, and allow continuous delivery by differentiating between which inconsistencies would or would not impact the next product release.

Manual review is the primary means of analyzing consistency. 80 % of survey respondents manually inspect different artifacts in order to assure consistency. 40 % use automated techniques and(30 % use testing, that is, if a product behaves as expected, then it is assumed that the dependencies between implementation artifacts are consistent. Examples of reported automated techniques include: an automated build process that checks for dependencies (S3), a code generator that checks for inconsistencies between the variability model and source code (S15), and a tool that checks for architectural violations given a variability model (S22).

Assuring consistency is also a tool-integration problem. Six interviewees pointed out that they use automated consistency-assurance techniques. For instance, I6 uses a script to check if options used in the source code are defined in the variability model; I5 checks consistency between the variability model and test cases by mapping test cases to functionalities under which they are tested for each configuration (this mapping is partly tool-supported); I6 uses functional tests to check consistency between the variability model and a requirements database. Connecting variability modeling tooling to, mainly, requirements management tools (I6), and tools for deriving variants (I4), is primarily a tool-integration problem. An example is described by I4 who raises the need for a tool that would track each configuration-option selection across the complete tool chain, when deriving a variant.

4.6 Source-Code Analyses

As shown in Figure 6, only two code properties were perceived as critical in the survey: *absence of deep nesting of #IFDEFs and #IFs* (40 %) and *low scattering degree of configuration options* (44 %). The remaining properties were all perceived as nice to have. We asked the interviewees to elaborate their answers regarding the criticality of different code properties.

Manual reviews and non-variability-aware static analysis tools are the primary means of analyzing source code. 70 % of the survey respondents assure code properties using manual reviews, followed by 40 % who use in-house scripts. Even though, 80 % assure code properties using static analysis tools (e.g., Coverty, Code Sonar or PC-Lint), none is variability-aware.

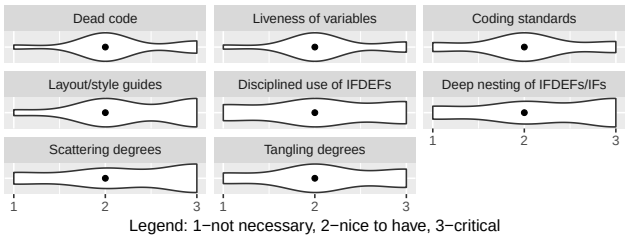

Figure 6: Importance of assuring code properties

Adherence to coding standards can alleviate the need for code analysis. The survey asked about coding guidelines, specifically about the adherence to coding standards such as MISRA, and company-specific code style guides. Although both were judged as nice to have, the former is assured by 90 % and the latter by 80 % of the survey participants. Considering that MISRA is a *de facto* standard, the high percentage assuring compliance is not surprising. However, the reason for enforcing adherence to company-specific code style guides varies. Interviewee I1 stated that such rules are crucial for enabling the collaborative development among geographically distant teams. Other participants create rules to facilitate script-based static code-analysis, for instance, I10: *"There are sometimes [...] static variables that are used in one C-function and then they are used again in another C-function, which makes it really difficult for our parsing to see what is going on."*

Deep nesting of *IF*s and *#IFDEF*s is avoided. Although perceived as critical, avoiding deep nesting is in most cases only assured through code reviews based on aforementioned coding standards (I1, I5, I2, I10). Another practice seen is to create explicit rules limiting the nesting to three levels. Other interviewees stated that decisions are made on a case-by-case basis, even though, they admit that the consequences of deeply nested structures can be serious (I2, I1, I10). I10: *"the cyclomatic complexity in the tool that we use goes to the maximum because there [are] a lot of ways through the code. So the potential of something going wrong is very high."*

4.7 Feature Interaction

Although feature interaction was not a separate category in the survey, interviews regularly included discussions about the importance and the difficulties that hinder such analysis. The main observation from the survey is that 65 % of the respondents perceive absence of feature interaction as a critical property—in line with the 74 % of respondents stating that sometimes specific combinations of options cause problems.

Absence of feature interactions is also primarily analyzed by testing or manual reviews. In cases when the absence of feature interaction is assured, it is usually done through manual reviews (I9, I12) or extensive testing (I10, I11). Still, recall the limited sampling observed, disregarding more systematic coverage criteria. Four interviewees stated that assuring the absence of feature interactions would be nice, but not a *game changer* (I3). Reasons for not perceiving feature interaction as a big problem are the manageable size of embedded software, organizational culture that emphasizes feature modularization, and mandatory regression testing.

Analyzing feature interactions in large, loosely coupled systems is challenging. A specific challenge for feature interaction in large systems that have loosely coupled modules was explained by I14: *"if you look into larger systems, [...] different processes [...] communicate over IPC (Inter Process Communication) and then, you cannot do this analysis with a compiler."* This can be extended to distributed embedded systems where a feature implementation is distributed across several computational units.

5 SUMMARY AND IMPACT

We now summarize our main results and discuss their implications for practitioners and researchers.

Our subject systems can be seen as very typical and substantial cases of industrial highly configurable systems from diverse domains and of varying scales. While their main characteristics (Section 4.3), including the configuration mechanisms and technologies they use, largely resemble those of systems used in empirical studies or evaluations of analysis techniques (e.g., open-source systems software), we observed a mismatch between typical assumptions made in the literature and the actual practitioners' needs. Certain development structures and system characteristics—often abstracted away when proposing new analysis technologies—appear to hinder many of the more sophisticated analysis techniques.

Identified Needs (RQ1). We both confirm and refute common assumptions. First, the severity that our practitioners express for the common properties suggested in the literature confirms their relevance for highly configurable systems. However, most of the variability-model-related analysis properties are not seen as important by our practitioners. The proposed change-impact analyses are not seen as sufficient, because they are confined to the model and its configuration space, not providing holistic insights on impacts on implementation artifacts. Assuring consistencies between artifacts (especially variability model and source code) is considered highly critical, as well as identifying unwanted feature interactions.

Identified Practices (RQ2). We observed (as expected) testing as the dominant practice. Interestingly, the configuration sampling criteria that are necessary for testing primarily rely on experience. Hardly any systematic sampling or random sampling is used. Our results also suggest that the latter are not even applicable given the configuration spaces that would still leave too many irrelevant variants. Furthermore, hardly any formal method is used (apart from limited model checking). Besides testing, manual work, such as code reviews, is exercised, because often the variability models required for more sophisticated analyses do not exist or are not expressed in a form that can be used as an input. The lack of integrated tool chains is also a factor, since artifacts required for performing analyses are managed in different tools. Interestingly, the experience of the developers and rules, such as coding standards, but also engineering practices such as modularization of code, often alleviate the need for sophisticated analyses of the highly configurable system.

Research Directions. Our results suggest to refocus some research efforts towards the actual needs our study identified. General research directions could be the following.

Improve engineering methods to alleviate the need for analyses. Since most analyses need substantial investment and adaptation towards the specific highly configurable system and its engineering

environment, investigating ways to avoid them in the first place is a worthwhile research avenue.

Conceive lightweight analyses that account for the diversity of artifacts and tool chains, or that are even independent of these. At least, authors of new analysis techniques should clearly state assumptions on how variability is expected to be modeled and manifested in the software artifacts.

Unify variability management and modeling concepts. We again [11, 13] observed a variety of variability and configuration mechanisms. Addressing this tool-integration problem appears to require unifying variability concepts with concepts from commonly used tools, especially version-control systems [44].

Conceive hybrid analyses that combine manual reviewing with variability-aware analyses. Recall that manual, case-by-case decisions on actions to be done upon analysis results cannot be avoided and that state-of-the-art analysis tooling requires substantial setup. Furthermore, variability-aware analysis are static by nature and, therefore, typically produce many false positives. Thus, hybrid analysis techniques could be efficient.

With respect to concrete analysis techniques, our results suggest the following main research directions.

Improve testing techniques for highly configurable systems, especially automated test-case generation, test-case selection based on quality properties (e.g., safety), and traceability management between test cases and configurations that failed tests.

Investigate experience-based configuration sampling, especially identify the sources of experience and map these sources to the effectiveness of finding bugs. This requires experiments and field studies. Furthermore, combining experience-based sampling and systematic sampling seems to be a worthwhile research direction based on our results.

Conceive change-impact analyses that are not confined to the variability model, but offer insights on the impact of changes to other artifacts when done to the variability model or to the implementation artifacts. Concisely and effectively presenting change impacts to engineers is another challenge.

Realize quicker and more flexible consistency checking, for instance, to support continuous integration and deployment. The feedback loop (from making changes to being alerted about inconsistencies) needs to be much shorter.

Conceive feature-interaction analyses for large and loosely coupled systems. These analyses should take more diverse artifact relationships into account and should go beyond static analyses currently offered by compilers.

Finally, our study again emphasized that regular feedback loops with practitioners are crucial, steering research efforts away from analysis techniques that might be low-hanging fruits, but are not perceived as needed in the real world.

6 THREATS TO VALIDITY

Construct Validity. Our survey and the interviews used the concept of highly configurable system to ensure that all practitioners could describe their practices without the need to adopt a specific terminology. We used terms such as *configuration option* to refer to the concept of feature, *configuration specification* to refer to variability model, and provided short explanations for non-trivial questions to mitigate potential misinterpretations. We iteratively developed our questionnaire and our interview guide using pilot runs already with industrial participants. To ensure completeness, participants could provide additional information. Finally, we used violin plots to visualize trends in the responses about the severity of different analyses. So, we interpreted the four-point scale as continuous with equal distances between points, but did not use this interpretation as a foundation for statistical analysis.

Internal Validity. We selected the participants based on their industrial and technical experience. This experience paired with the combination of survey and interviews provided both general perspectives on analysis techniques and on assured properties, as well as specific insights with respect to how analyses are performed and what the needs are. Since all subjects were interested in exploring new analysis techniques and were aware that their input might shape future research, they were very open about the current limitations and had no incentive to present their current practices in a better light. Finally, even though, the interviews were conducted by different researchers, the recordings were exchanged for transcription and for coding to avoid potential biases.

External Validity. All our study participants work with highly configurable systems of varying sizes and maturity, covering a wide range of domains. The needs we elicited and the insights we derived can be applied to highly configurable systems in similar domains. Some needs and practices reported are dependent on a concrete system, but we identified these and marked them accordingly if they were mentioned.

Conclusion Validity. Our qualitative analysis depends on our interpretation. However, we mitigated bias by collaboratively coding the interviews using open coding, cross-checking the codes, refining the codes, and conducting a coding workshop by all authors. We used triangulation and carefully formulated and verified insights and conclusions to enhance our study's validity.

7 CONCLUSION

We presented a study on the needs and practices of analyzing highly configurable systems. We studied substantial industrial cases covering a wide range of domains, development scales, and system complexities. Mapping existing research results to industrial needs and practices is intrinsically difficult, given the different cultures, terminologies, and system architectures in practice. Our focus was to deeply understand each case using expert interviews, while also going into a reasonable breadth (still focusing on response quality) with our survey. We found rather pragmatic practices and a surprisingly low adoption (and awareness) of academic analyses, even though, most of the studied companies have research collaborations. As future work, we intend to map needs to the state of the art from the literature and conceive a research agenda.

ACKNOWLEDGMENT

We thank all of our study participants, Vinnova Sweden (2016-02804) funding the EU ITEA project REVAMP^2, and the Swedish Research Council (257822902).

REFERENCES

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

[2] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. 2011. Detection of Feature Interactions Using Feature-aware Verification. In *ASE*.

[3] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-interaction detection based on feature-based specifications. *Computer Networks* 57, 12 (2013), 2399–2409.

[4] Ebrahim Bagheri and Dragan Gasevic. 2011. Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal* 19, 3 (2011), 579–612.

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

[6] Don S. Batory, David Benavides, and Antonio Ruiz Cortés. 2006. Automated analysis of feature models: challenges ahead. *Commun. ACM* 49, 12 (2006), 45–47.

[7] David Benavides, Antonio Ruiz-Cortés, Pablo Trinidad, and Sergio Segura. 2006. A Survey on the Automated Analyses of Feature Models. In *JISBD*.

[8] 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.

[9] Thorsten Berger and Jianmei Guo. 2014. Towards System Analysis with Variability Model Metrics. In *VAMOS*.

[10] 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 *SPLC*.

[11] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS*.

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

[13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VAMOS*.

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

[15] Jan Bosch. 2009. From Software Product Lines to Software Ecosystems. In *SPLC*.

[16] Muffy Calder, Mario Kolberg, Evan H Magill, and Stephan Reiff-Marganiec. 2003. Feature interaction: a critical review and considered forecast. *Computer Networks* 41, 1 (2003), 115–141.

[17] Gary Chastek, Patrick Donohoe, Kyo Chul Kang, and Steffen Thiel. 2001. *Product line analysis: a practical introduction.* Technical Report CMU/SEI-2001-TR-001.

[18] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *FASE*.

[19] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. 2010. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *ICSE*.

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

[21] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. 2007. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA*.

[22] Juliet M Corbin and Anselm Strauss. 1990. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology* 13, 1 (1990), 3–21.

[23] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VAMOS*.

[24] 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 *CSMR*.

[25] Emelie Engström and Per Runeson. 2011. Software product line testing—a systematic mapping study. *Information and Software Technology* 53, 1 (2011), 2–13.

[26] Joel Greenyer, Amir Molzam Sharifloo, Maxime Cordy, and Patrick Heymans. 2013. Features meet scenarios: modeling and consistency-checking scenario-based product line specifications. *Requirements Engineering* 18, 2 (2013), 175–198.

[27] Jianmei Guo, Dingyu Yang, Norbert Siegmund, Sven Apel, Atrisha Sarkar, Pavel Valov, Krzysztof Czarnecki, Andrzej Wasowski, and Huiqun Yu. 2018. Data-efficient performance learning for configurable systems. *Empirical Software Engineering* 23, 3 (2018), 1826–1867.

[28] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2017. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. *arXiv preprint arXiv:1710.07980* (2017).

[29] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. 2007. Model Composition in Product Lines and Feature Interaction Detection using critical pair analysis. In *MODELS*.

[30] Kyo 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.

[31] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. 2012. Type Checking Annotation-based Product Lines. *ACM Transactions on Software Engineering and Methodology* 21, 3 (July 2012), 14:1–14:39.

[32] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *OOPSLA*.

[33] Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. 2017. Is There a Mismatch Between Real-world Feature Models and Product-line Research?. In *ESEC/FSE*.

[34] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Experimentation Workbench for Analyzing Software Product Lines. In *ICSE*.

[35] Charles W. Krueger. 2006. New Methods in Software Product Line Practice. *Commun. ACM* 49, 12 (Dec. 2006), 37–40.

[36] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *VaMoS*.

[37] Kim Lauenroth, Klaus Pohl, and Simon Toehning. 2009. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE*.

[38] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. 2013. Validating Consistency Between a Feature Model and its Implementation. In *ICSR*.

[39] Jihyun Lee, Sungwon Kang, and Danhyung Lee. 2012. A Survey on Software Product Line Testing. In *SPLC*.

[40] Jorg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE*.

[41] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *AOSD*.

[42] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dorre, and Christian Lengauer. 2012. *Large-Scale Variability-Aware Type Checking and Dataflow Analysis.* Technical Report MIP-1212.

[43] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. In *ESE/FSE*.

[44] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A Classification of Variation Control Systems. In *GPCE*.

[45] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *ICSE*.

[46] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. 2014. An Overview on Analysis Tools for Software Product Lines. In *SPLat*.

[47] Jan Midtgaard, Claus Brabrand, and Andrzej Wasowski. 2014. Systematic Derivation of Static Analyses for Software Product Lines. In *MODULARITY*.

[48] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*.

[49] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *ICSE*.

[50] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering* 41, 8 (2015), 820–841.

[51] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D McGregor, Eduardo Santana De Almeida, and Silvio Romero de Lemos Meira. 2011. A systematic mapping study of software product lines testing. *Information and Software Technology* 53, 5 (2011), 407–423.

[52] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. 2013. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *SPLC*.

[53] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. 2010. Automated and scalable t-wise test case generation strategies for software product lines. In *ICST*.

[54] Rodrigo Queiroz, Thorsten Berger, and Krzysztof Czarnecki. 2016. Towards Predicting Feature Defects in Software Product Lines. In *FOSD*.

[55] Sacha Reis, Andreas Metzger, and Klaus Pohl. 2006. A reuse technique for performance testing of software product lines. In *SPLiT*.

[56] Andreas Reuys, Sacha Reis, Erik Kamsties, and Klaus Pohl. 2006. The scented method for testing software product lines. In *Software Product Lines*. Springer, 479–520.

[57] Alcemir Rodrigues Santos, Raphael Pereira de Oliveira, and Eduardo Santana de Almeida. 2015. Strategies for Consistency Checking on Software Product Lines: A Mapping Study. In *EASE*.

[58] Florian Sattler, Alexander von Rhein, Thorsten Berger, Niklas Schalck Johansson, Mikael Mark Hardø, and Sven Apel. 2018. Lifting Inter-App Data-Flow Analysis

to Large App Sets. *Automated Software Engineering* 25 (jun 2018), 315–346. Issue 2.

[59] Tonny Kurniadi Satyananda, Danhyung Lee, and Sungwon Kang. 2007. Formal Verification of Consistency Between Feature Model and Software Architecture in Software Product Line. In *ICSEA*.

[60] Klaus Schmid, Rick Rabiser, and Paul Grünbacher. 2011. A comparison of decision modeling approaches in product lines. In *VaMoS*.

[61] 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 *SPLC*.

[62] Anselm Strauss and Juliet Corbin. 1990. Open Coding. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques* 2 (1990), 101–121.

[63] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *EuroSys*.

[64] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 531–551.

[65] The Authors. 2018. Online Appendix. https://sites.google.com/view/planalysis/.

[66] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A classification and survey of analysis strategies for software product lines. *Comput. Surveys* 47, 1 (June 2014), 6:1–6:45.

[67] Thomas Thum, Don Batory, and Christian Kastner. 2009. Reasoning about edits to feature models. In *ICSE*.

[68] Thomas Thüm, Ina Schaefer, Sven Apel, and Martin Hentschel. 2012. Family-based Deductive Verification of Software Product Lines. In *GPCE*.

[69] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.

[70] Michael Vierhauser, Paul Grünbacher, Wolfgang Heider, Gerald Holl, and Daniela Lettner. 2012. Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines. In *MODELS*.

[71] Bo Zhang, Martin Becker, Thomas Patzke, Krzysztof Sierszecki, and Juha Erik Savolainen. 2013. Variability Evolution and Erosion in Industrial Product Lines: A Case Study. In *SPLC*.