Thesis for The Degree of Licentiate of Engineering

# Understanding Variability-Aware Analysis in Low-Maturity Variant-Rich Systems

Mukelabai Mukelabai

Division of Software Engineering
Department of Computer Science & Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden, 2020

**Understanding Variability-Aware Analysis in Low-Maturity Variant-Rich Systems**

Mukelabai Mukelabai

Technical Report No 209L
ISSN 1652-876X
Department of Computer Science & Engineering
Division of Software Engineering
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2020.

*"Our knowledge can only be finite, while our ignorance must necessarily be infinite."*
*- Karl Popper*

# Abstract

*Context:* Software systems often exist in many variants to support varying stakeholder requirements, such as specific market segments or hardware constraints. Systems with many variants (a.k.a. variant-rich systems) are highly complex due to the variability introduced to support customization. As such, assuring the quality of these systems is also challenging since traditional single-system analysis techniques do not scale when applied. To tackle this complexity, several variability-aware analysis techniques have been conceived in the last two decades to assure the quality of a branch of variant-rich systems called software product lines. Unfortunately, these techniques find little application in practice since many organizations do not use product-line engineering techniques, but instead rely on low-maturity *clone&own* strategies to manage their software variants. For instance, to perform an analysis that checks that all possible variants that can be configured by customers (or vendors) in a car personalization system conform to specified performance requirements, an organization needs to explicitly model system variability. However, in low-maturity variant-rich systems, this and similar kinds of analyses are challenging to perform due to (i) immature architectures that do not systematically account for variability, (ii) redundancy that is not exploited to reduce analysis effort, and (iii) missing essential meta-information, such as relationships between features and their implementation in source code.
*Objective:* The overarching goal of the PhD is to facilitate quality assurance in low-maturity variant-rich systems. Consequently, in the first part of the PhD (comprising this thesis) we focus on gaining a better understanding of quality assurance needs in such systems and of their properties.
*Method:* Our objectives are met by means of (i) knowledge-seeking research through case studies of open-source systems as well as surveys and interviews with practitioners; and (ii) solution-seeking research through the implementation and systematic evaluation of a recommender system that supports recording the information necessary for quality assurance in low-maturity variant-rich systems. With the former, we investigate, among other things, industrial needs and practices for analyzing variant-rich systems; and with the latter, we seek to understand how to obtain information necessary to leverage variability-aware analyses.
*Results:* Four main results emerge from this thesis: first, we present the state-of-practice in assuring the quality of variant-rich systems, second, we present our empirical understanding of features and their characteristics, including information sources for locating them; third, we present our understanding of how best developers' proactive feature location activities can be supported during development; and lastly, we present our understanding of how features are used in the code of non-modular variant-rich systems, taking the case of feature scattering in the Linux kernel.
*Future work:* In the second part of the PhD, we will focus on processes for adapting variability-aware analyses to low-maturity variant-rich systems.

### Keywords

Variant-rich Systems, Quality Assurance, Low Maturity Software Systems, Recommender System

# Acknowledgment

I would like to thank my research supervisors Thorsten Berger and Jan-Phillip Steghöfer for their patient guidance and enthusiastic encouragement through out my research work. Both have taught me many things with regard to research, teaching in higher education, and career development. In particular, Thorsten, as my main Supervisor, has been very instrumental in providing me with several opportunities such as, international collaboration with top researchers in my field, peer review experience for several venues, and participating in locally organizing the 22nd International Systems and Software Product Line conference in the role of Proceedings Chair. These opportunities have not only enhanced my academic prowess but have also been useful in letting me gain an understanding of the broad spectrum of research problems in my field. In addition, Thorsten has organized several social outings for our research group, such as kayaking in Gothenburg, hiking in the islands of Saltholmen, and taking tree climbing obstacle courses in Partille; these were fun and helped relieve stress many times. On the other hand, the advice and feedback given by my co-supervisor Jan-Phillip was very helpful in both the various research I undertook and in balancing my academic and social life.

I am extremely grateful and honored to be working at the Software Engineering division comprising such an internationally-rich combination of talented academic and administration personnel. Without such a friendly and conducive working environment, this research work would not have been possible. I would like to thank colleagues in the *EASE* research lab at the Software Engineering division, Daniel Strüber and Jakob Krüger for their collaboration. I also thank colleagues I share my office with, Wardah Mahmood, Khaled Al, and previously Abdullar Mamoon and Hiva, for their support and social moments shared.

Lastly, but not the least, I also extend special thanks to my wife, Amanda Hims Mukelabai, for her support and understanding through out my work, and to my God through whose providence and sustenance I have been able to conduct my research.

This research was partially supported by the ITEA project REVaMP[2] funded by Vinnova Sweden (2016-02804) and by the Swedish Research Council Vetenskapsrådet (257822902).

# List of Publications

## Appended publications

This thesis is based on the following publications:

[A] M. Mukelabai, D. Nešić, S. Maro, T. Berger, J. Steghöfer "Tackling Combinatorial Explosion: A Study of Industrial Needs and Practices for Analyzing Highly Configurable Systems"
*Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 155-166. 2018.*

[B] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, T. Berger "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.*

[C] M. Mukelabai, T. Berger, J. Steghöfer "FeatRacer: Locating Features Through Assisted Traceability"
*Planned to be submitted for publication.*

[D] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, J. Padilla "A Study of Feature Scattering in the Linux Kernel"
*IEEE Transactions on Software Engineering (2018).*

# Other publications

The following publications were published during my PhD studies, or are currently in submission/under revision. However, they are not appended to this thesis, due to contents overlapping that of appended publications or contents not related to the thesis.

[a] M. Mukelabai, B. Behringer, M. Fey, J. Palz, J. Krüger, T. Berger "Multi-View Editing of Software Product Lines with PEoPL."
*IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 81-84. 2018.*

[b] M. Mukelabai "Verification of Migrated Product Lines."
*Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 2, pp. 87-89. 2018.*

[c] D. Strüber, M. Mukelabai, J. Krüger, S. Fischer, L. Linsbauer, J. Martinez, T. Berger "Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-rich Systems."
*Proceedings of the 23rd International Systems and Software Product Line Conference-Volume A, pp. 177-188. 2019.*

[d] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, T. Berger "Towards a Better Understanding of Software Features and Their Characteristics: a Case Study of Marlin."
*Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, pp. 105-112. 2018.*

[e] T. Thüm, L. Teixeira, K. Schmid, E. Walkingshaw, M. Mukelabai, M. Varshosaz, G. Botterweck, I. Schaefer, T. Kehrer "Towards Efficient Analysis of Variation in Time and Space."
*Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, pp. 57-64. 2019.*

# Research Contribution

The following were my contributions to each paper, listed according to the Contributor Roles Taxonomy (CRediT)[1]. Where necessary, the degree of contribution is specified as 'lead', 'equal', or 'supporting'.

In Paper A, I led the design of the methodology, formal analysis of the survey and interview data, validation (reproducibility), and visualization (presentation) of the data. I participated equally in the collection of interview data but led the survey data collection. I also participated equally in writing the original draft but led the review and editing of the paper.

In Paper B I led the investigation of our subject systems, participated equally in the design of the methodology and formal analysis of the data, and played a supporting role in writing of the original draft.

In Paper C I developed the recommender system, led in the roles of data collection, formal analysis, writing the original draft, and participated equally in the design of the methodology and validation of the results.

Lastly, in Paper D, I participated equally in the formal analysis of the survey and interview data, led the writing process for both the original draft and the review, and was responsible for data curation.

[1] https://casrai.org/credit/

# Contents

# Chapter 1

# Introduction

Software often needs to be customized to address varying stakeholder requirements, such as specific market segments or hardware constraints. To achieve this, organizations create variants of their systems, resulting in a portfolio of software variants that need to be maintained. Each variant in the portfolio is distinguished by the set of features comprising it. Developers commonly use features to communicate and manage functional and quality aspects of their software as well as to reuse and adapt existing variants to new requirements [1,2]. In practice, variant-rich systems are of varying maturity levels with regard to how they are engineered and managed; from ad hoc *clone&own* to integrated configurable systems.

Variant-rich systems pervade and greatly enhance modern life; as such, customers demand high quality for these systems as well as they do for software products resulting from traditional single-system development. However, the complexity of variant-rich systems demands quality assurance techniques that take into account variability. Thus, over the last decades, hundreds of dedicated variability-aware analysis techniques [3, 4] have been conceived, many of which are able to analyze system properties for all possible variants, as opposed to traditional, single-system analyses. Unfortunately, these techniques target well established integrated platforms (right side of Figure 1.1) that have variability concepts, such as feature models and feature-to-asset traceability, implemented and formally specified. Yet, many industrial variant-rich systems are still immature [5, 6] and can hardly use these analysis techniques. Hence, to facilitate their adoption, different challenges need to be addressed; for instance, features comprising different variants need to be explicitly documented and traced to software assets that implement them.

Figure 1.1 illustrates two contrasting approaches to engineering and managing variants of a system. The first and most commonly used is called *clone&own* (left side of Figure 1.1) in which developers clone an existing project and adapt it to new requirements to create a new variant. This method is convenient and requires little upfront investment, hence its wide adoption in industry [5, 6]. However, ad hoc *clone&own* imposes significant maintenance overheads as the number of variants grows; for instance, when fixing bugs in multiple variants. The second and most mature approach is to use a *configurable and integrated platform* (right side of Figure 1.1) that uses software

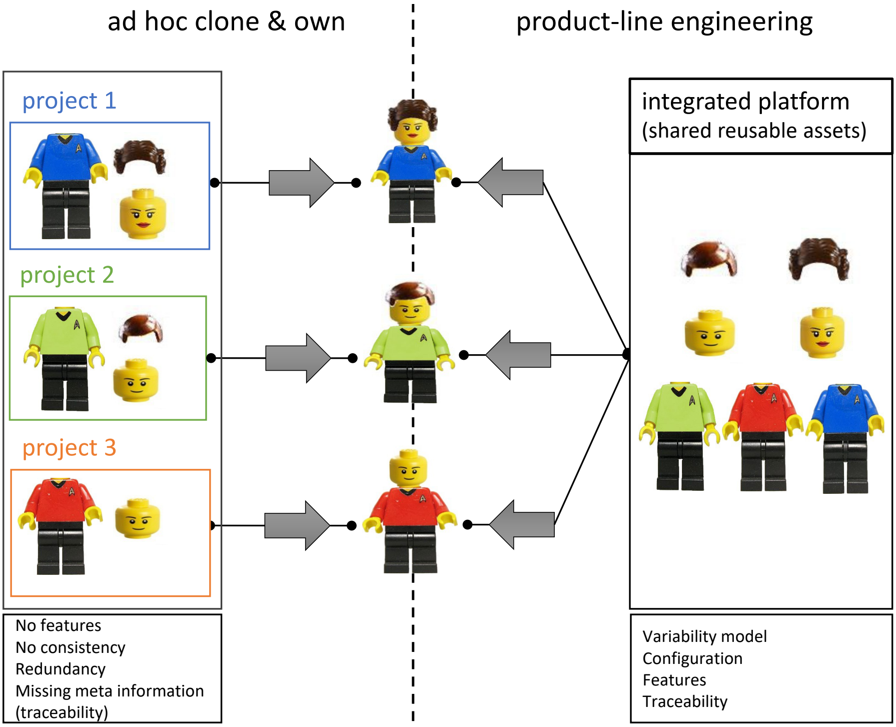


Figure 1.1: Ad hoc *clone&own* strategy vs configurable and integrated platform strategy

product line engineering (SPLE) [1,7,8] methods to engineer and automatically derive variants. With this approach, rather than independently developing variants, reusable assets are engineered and configured to generate desired variants based on selected features. Examples of variant-rich systems developed using the product-line approach include highly configurable systems, such as the Linux kernel and software product lines found in the avionics, automotive, industrial automation, and telecommunications domains. While this approach offers several advantages over *clone&own*, such as better product quality and faster time to market, it is difficult to adopt in practice, because, firstly, it may be difficult for an organization to foresee market needs a priori, and secondly, it requires a substantial upfront investment for the organization to make its code-base configurable and implement variability concepts such as feature modeling [2] and asset-to-feature traceability. Thus, most organizations begin with *clone&own* and later migrate gradually to a configurable platform, albeit the migration is costly and risky [9–11].

The overarching goal of the PhD is to facilitate variability-aware quality assurance in low-maturity variant-rich systems. To work towards this goal, in this thesis we aim to gain a better understanding of variability-aware analysis in low-maturity variant-rich systems. To that effect, we conduct three knowledge seeking studies and one solution-seeking study, whose contributions are summarized in Figure 1.2. As a first step, we investigate the state-of-practice w.r.t analysis of variant-rich systems, using the following research question:

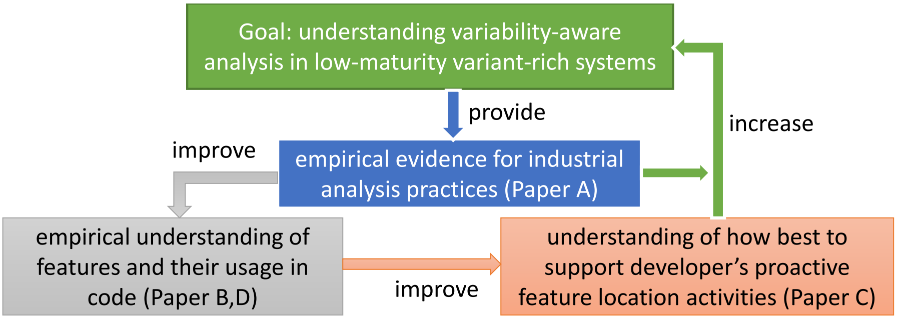

Goal: understanding variability-aware analysis in low-maturity variant-rich systems
provide
increase
improve
empirical evidence for industrial analysis practices (Paper A)
empirical understanding of features and their usage in code (Paper B,D)
improve
understanding of how best to support developer's proactive feature location activities (Paper C)

Figure 1.2: Summary of paper contributions to achieving the aim of the thesis

RQ1: *What are industrial practices for assuring the quality of variant-rich systems and what properties are assured?* (Paper A)

As stated above, several variability-aware analysis techniques have been proposed over the last two decades. We seek to understand whether these techniques are adopted in practice, whether they address actual needs, and what strategies practitioners actually apply to analyze variant-rich systems. The results of this investigation are reported in Paper A and form the basis for corresponding research questions.

Our findings from RQ1 reveal that many organizations still need more systematic and explicit variability management to be able to apply state-of-the-art variability-aware analysis techniques. Furthermore, features play a pivotal role in variability-aware analysis since they are commonly used to communicate and manage both common and variable functionalities of variant-rich systems: For instance, activities such as change-impact analysis, selection of test cases targeting specific features, and tracing failed test cases to affected variants or features, all require explicit feature documentation. In a variant-rich system, maintenance tasks, such as bug fixing, that may be easier to perform in single systems, are complicated on account of variability and require analysis techniques that can, for instance, indicate to a developer which feature combinations or variants are affected by a specific bug. Yet, in low-maturity variant-rich systems, this information necessary for analysis is missing; features and their locations in source code assets are poorly documented and developers often recover them retroactively — a.k.a., *feature location* — when faced with maintenance tasks such as bug fixing (see Section 1.1.3). In fact, feature location is considered one of the most common activities of software developers [12–15]. Hence, in our second and third research questions, we look into providing this missing information, mainly features and their locations. We formulate the second research question as:

RQ2: *What information sources are useful for recovering and locating features and their characteristics–feature facets?* (Paper B)

With RQ2, we seek an empirical understanding of features and how to recover them from source-code assets. Several automated techniques have been proposed to recover features and their locations [15–17]. However, these techniques generally exhibit low accuracy, need substantial effort to suit specific projects, and often only exploit a single source of information, such as execution traces or code comments. Moreover, recovering feature characteristics (henceforth referred to as facets), such as rationale or architectural responsibility of a feature, is even more difficult, since their corresponding information sources are largely unknown and developers may have varying understanding of these facets. Hence, in Paper B we seek an empirical understanding of information sources we can utilize to locate features and their facets, and strategies to exploit these information sources. This knowledge can prove useful for improving automated feature location techniques and consequently variability-aware analysis techniques.

Furthermore, considering the inaccuracy of automated feature location techniques and inefficiency of retroactive manual feature location [18, 19], we seek to understand how best developer's feature location activities can be supported during development. To that effect, we formulate the following research question:
RQ3: *How best can developers' feature location activities be pro-actively supported during development?* (Paper C)

Given that retroactive manual feature location recovery has been shown to be effort-heavy even for small systems ranging from 2k to 73k LOC [18], and that automated techniques are unreliable in practice, an alternative approach [20, 21] has been proposed that allows developers to actively record feature locations within software assets by annotating them with feature names. Particularly, the technique by Ji et al. [20] was shown, in a simulation study [20], to have low maintenance effort, and that the benefits of annotating assets outweigh the costs thereof, especially for variant-rich systems with many cloned variants. However, for very large systems, even this approach can potentially overwhelm developers, leading them to sometimes forget to annotate assets. Thus, in Paper C, we seek to understand how best developers can be supported in feature location activities by means of embedded annotations and a recommender system.

Lastly, considering that many variant-rich systems use non-modular mechanisms, such as preprocessor directives (e.g., `#ifdef`), to implement variability in source code, we seek to understand how features are used in such systems. In particular, we focus on one instance of variability-aware analysis of the source code that measures how feature code is spread across the code base — a.k.a., feature scattering. Consequently, we formulate the following research question:
RQ4: *How does feature scattering evolve and what are practices and circumstances leading to it?* (Paper D)

Scattered features significantly increase system maintenance efforts [1, 22], since they hinder program comprehension, can lead to ripple effects and require frequent developer synchronization, which challenges parallel development. By understanding how features are used in code w.r.t. scattering, our study presented in Paper D aims to play a role in creating a widely accepted set of practices to govern feature scattering and eventually serve as a guide to practitioners–for instance, in identifying implementation decay [23], and assessing the maintainability of a system [24].

We proceed by discussing the background on maturity of variant-rich systems, quality assurance and feature traceability in Section 1.1, followed by the methodology in Section 1.2. We summarize the four papers comprising this thesis in Section 1.3. We then discuss our main findings in Section 1.4, threats to validity in Section 1.5, and present the conclusion in Section 1.6 and future work in Section 1.7 .

## 1.1 Background

In this section we briefly introduce concepts of quality assurance for variant-rich systems, maturity levels, feature location and traceability, as well as machine learning.

### 1.1.1 Quality Assurance for Variant-rich Systems

Software quality assurance is a broad term that refers to a set of validation and verification activities carried out during the software engineering process to ensure that the resulting software products meet and comply with the quality standards of an organization [25]. Quality assurance activities target both software products and their related artifacts such as documentation for requirements, configuration, design, and tests [26]. Quality properties assured may include performance, security, maintainability, reliability, and consistency, depending on an organization's priorities, while methods of quality assurance range from dynamic analyses such as testing [25] to static analysis such as manual code inspections [27].

Nowadays, most software systems are large and complex, consisting of millions of lines of code. This complexity is even greater in variant-rich systems, since organizations do not only work with one but several variants. Consequently, quality assurance for variant-rich systems demands analysis techniques that account for their variability; for instance, that all possible variants that can be configured by customers (or vendors) conform to specified performance requirements. To this end, several such variability-aware analyses [3, 28, 29] have been conceived in the past few decades.

In general, variability-aware analyses for variant-rich systems target four main aspects of their general architecture: Figure 1.3 illustrates this architecture together with categories of typical quality properties that can be assured. The features distinguishing the variants of the system are declared in a *feature specification — a.k.a., variability model* (here a feature model [2, 30], but could also be a textual configuration or properties file). The feature specification may also describe relationships between the features, for instance, here, selecting feature ACPI requires that PCI and PM are also selected. For a configurable variant-rich system, the *code base* is *mapped* to the feature specification by means of configuration mechanisms (here, preprocessor directives, e.g., `#ifdef`) that determine which parts of the code base constitute a given variant based on the selected set of features. However, recall that for an organization using *clone&own*, variants are not realized through configuration mechanisms but through version-control techniques such as forking and branching.

With this overall architecture, quality properties to assure for variant-rich systems relate to:

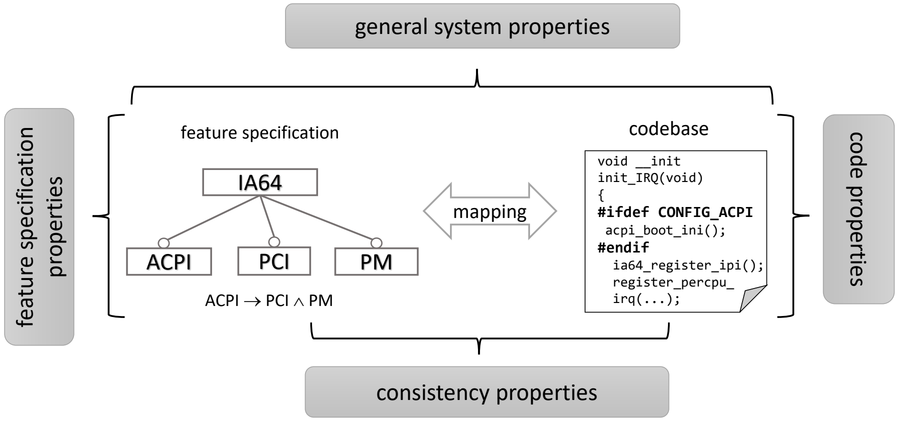
**General system properties for the whole system and its variants**. Similar to traditional single systems, all possible variants of a system must comply with set quality criteria for properties such as behavioral correctness, security, reliability, safety and performance. These qualities can be assured through dynamic analyses such as testing [4, 31, 32] or static analysis such as model checking [33] and deductive verification [34]. Even though, traditional single-system analysis techniques may be used on individual variants (e.g., for optimization) when the system is configured by the system vendor, these analyses are not sufficient to detect errors that pertain to all possible variants (in a configurable variant-rich system), especially when customers configure it (e.g., a customer selecting features for their car). Thüm et al. [3] present a survey of a class of static analyses, called *variability-aware analyses*, that target variant-rich systems typically by lifting single-system analyses. These analyses can assure properties such as type-safety [35], performance [36], or absence of unwanted feature interactions [37, 38] for the whole system, individual variants or features.

**The feature specification**. Used to describe both the common and variable features of a system as well as relationships among them, a feature specification can be informal (e.g., a listing of features in a spreadsheet) or formal (e.g., a feature model [2]). A vast amount of analyses have been proposed in the literature for formal feature specifications, especially feature models; Benavides et al. [29] present a survey of over 30 such analyses targeting feature models. These analyses can check, for instance, that the specification is satisfiable (i.e., at least one valid variant exists given the feature dependencies and constraints), or that any given variant satisfies feature constraints (i.e., a variant does not contain invalid combinations e.g., a car should either be manual or automatic transmission but not both). Our study in Paper A is complementary to Benavides et al.'s survey, since we match these analyses to industrial needs.

**The code base**. Source code exhibits several structural properties [39, 40] that can be assured. Such properties include, for instance, scattering degrees of features (to what extent a feature's implementation is spread across the code base), coding standards, layout/style guides, and deep nesting of conditional statements such as *if* statements or preprocessor macros such as `#ifdef`. These properties, if not quality-assured, have potential to hinder program comprehension and challenge parallel development, thus, substantially increasing maintenance efforts. Our study presented in Paper D investigates one of these properties — feature scattering — to provide insights on circumstances leading to it and developer practices for coping with it.

**The mapping**. To prevent inconsistencies between the feature specification and implementation artifacts, several consistency-related analyses [41, 42] have been proposed. Consistency checks include, for instance, that the feature specification is consistent with the code (e.g., that all features have related code or vice versa); that constraints in the source code are consistent with feature constraints (e.g., that there is no dead code [43]); or that the feature specification is consistent with requirements. Some of the proposed analysis techniques use SAT solvers to check for consistency, while a few others rely on model checking [44] and theorem proving [45]. However, due to lack of formal specifications for some artifacts, e.g., requirements, manual inspections are also widely used.

Unlike Thüm et al's [3] and Benavides et al.'s [29] surveys, which identify the state-of-the art, our study presented in Paper A investigates the state-of-practice — offering insights about the adoption and potential challenges when using existing variability-aware analyses for the above four aspects of



```
void __init
init_IRQ(void)
{
#ifdef CONFIG_ACPI
 acpi_boot_ini();
#endif
  ia64_register_ipi();
  register_percpu_
  irq(...);
```

Figure 1.3: Architecture of a configurable variant-rich system and categories of typical properties to assure

variant-rich systems.

The above analysis techniques presuppose the existence of formally specified features, configurable source-code, and established traceability links between features and implementation artifacts. Yet, most variant-rich systems are developed using ad hoc *clone&own*, with poorly documented features.

### 1.1.2 Maturity of Variant-rich Systems

Variant-rich systems are portfolios (or families) of related software products targeting different market segments or requirements. On account of how variability is managed and how products are derived, variant-rich systems differ in maturity; from less mature systems using ad hoc *clone&own* to mature ones using software product-line engineering methods to engineer and derive variants. Despite the benefits of the product-line approach, often, a 'big bang' transition from *clone&own* to a configurable platform is perceived costly and risky [9–11]. Hence, incremental migration is more desirable. Antkiewicz et al. [46] propose six governance levels, illustrated in Figure 1.4, that would allow an organization to make such a transition in a minimally invasive way. Moreover, these levels are also indicative of the maturity of a variant-rich system since advancing from one level to the next is considered an incremental realization of a configurable platform with incremental benefits and investment. At level 0, the organization uses *ad hoc clone&own* with no notion of features or reuse; only a single variant is derived from each project. At level 1, *clone&own* is used with *provenance*; development teams record provenance information about original projects and per cloned asset, e.g., that assetB is a *cloneOf* assetA. This information facilitates change propagation and bug fixes among cloned assets. At level 2, *clone&own* is used with *features*; teams declare features and map them to assets that implement them — a.k.a asset-to-feature traceability. The use of features provides for functional decomposition of projects and reasoning about their co-evolution. At level 3, the organization uses *clone&own with configuration* in which case teams can introduce constraints among features to exclude invalid

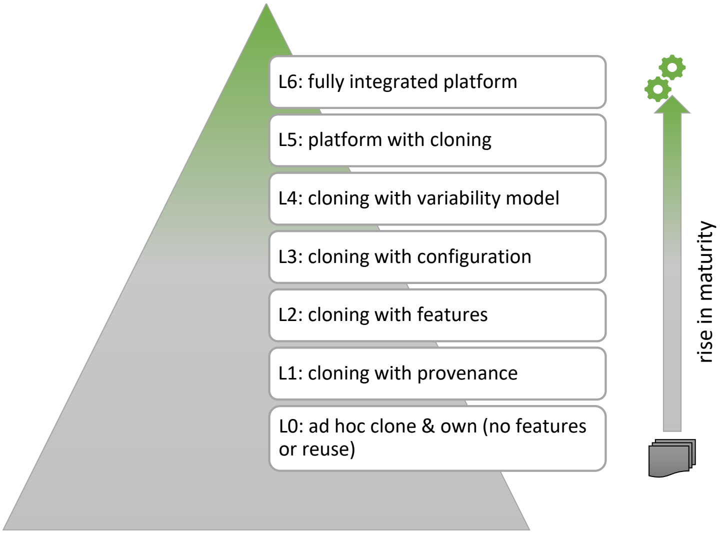

L6: fully integrated platform
L5: platform with cloning
L4: cloning with variability model
L3: cloning with configuration
L2: cloning with features
L1: cloning with provenance
L0: ad hoc clone & own (no features or reuse)
rise in maturity

Figure 1.4: Maturity levels of variant-rich systems [46]

combinations as well as disable or enable specific features for individual projects and derive variants based on selected subset of features. This level minimizes cloning while maximizing reuse potential. At level 4, *clone&own* is used with a central feature model that teams use as a reference point for creating new projects and propagating changes, while taking into account feature constraints. At level 5 the organization uses product-line engineering with a platform to configure and derive new variants but at the same time supports merging of existing cloned projects into the platform. Existing projects are used to harvest features that may be integrated into the platform. Lastly, at level 6, the organization uses a fully integrated platform from which variants are automatically derived. Level 6 is a superset of all the previous levels and is the target of many quality assurance techniques for variant-rich systems.

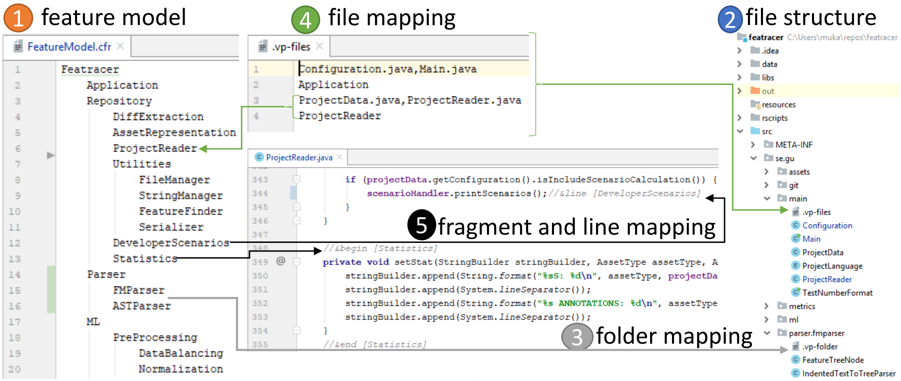
### 1.1.3 Feature Location and Traceability in Low-Maturity Variant-Rich Systems

Developers commonly use features to define, manage, and communicate functionalities of a system. Unfortunately, the locations of features in code and other characteristics of features (a.k.a., facets), relevant for evolution and maintenance, are often poorly documented. When a system evolves over time, the knowledge about features, their facets, and their locations often fades and has to be recovered— an activity known as *feature location*. In fact, feature location is considered one of the most common activities of developers [12–15].

Owing to poor documentation, developers recover features retroactively when need arises; for instance, during a maintenance task such as bug fixing or when integrating clones. Manual feature location [12, 47] is time consuming, takes substantial effort, and has been empirically demonstrated [12] to be inefficient even for small systems with sizes ranging from 2k–73k lines of code . To this end, several studies have been conducted, applying different techniques to automatically retrieve feature locations. Rubin et al. [15] present a survey of 24 automated feature location techniques whose underlying methods include formal concept analysis [48], latent semantic indexing [49], term frequency–inverse document frequency (tf-idf), and hyper-link induced topic search (HITS) [50]. In general, these techniques exhibit low accuracy when used in practice, and often exploit only one source of information such as code comments or execution traces. Our study presented in Paper B aims to improve feature location and recovery techniques by providing an empirical understanding of information sources we can utilize for these purposes, strategies to exploit these information sources, and the facets of features.

Organizations can establish traceability between features and their corresponding software assets in either of two ways: either they record feature traceability information during the development of the features (*the eager strategy*), or they retroactively recover such information when needed (*the lazy strategy*) [20]. In the former, developers record feature traces actively while memory of such information is still fresh (e.g., when performing tasks related to a feature or shortly after), while in the latter, developers retroactively recover feature locations by reading through the code or applying automated techniques. As indicated above, the *lazy strategy* is inefficient or inaccurate whether done manually or automatically. When using the eager strategy, organizations can either store traceability information *externally*, e.g., in a database, or *internally* together with the assets. Using external storage is challenging [51] since it requires a universal way of addressing locations and also relies on tools (such as FEAT [22]) to alleviate the burden of constantly updating the feature locations as the code base evolves. On the other hand, using internal storage requires a mechanism for embedding [20, 21] and a mechanism for extracting and visualizing [52] the traceability information inside the software assets.

In Paper C, we present a study in which we seek to understand how best to support developer's traceability efforts, using the embedded annotation technique proposed by Ji et al. [20]. Figure 1.5 illustrates this technique using code examples from our implementation of FeaTracer — our FEAture TRACEability Recommender system. The annotation technique comprises i) a textual feature model (feature specification) giving a hierarchical list of all features; with indentation indicating hierarchy (part 1); ii) textual mapping files that annotate files (part 4) and folders (part 5) and are put into the folder hierarchy of a project (part 2); and iii) fragment (block) or line-level feature annotations that are put as comments into source code, irrespective of the programming language (part 5). Using this annotation system, the developer is able to trace the locations of features, which can later be useful for several maintenance tasks such as bug fixing, refactoring or even integration of clones. Furthermore, these annotations can be exploited by tools, such as FeatureDashboard [52], to provide several code and feature metrics relevant for maintenance and quality assurance. Some of these metrics include scattering degree (extent to which a feature's implementation is spread across the code base), tangling degree (extent to which a feature's implementation is mixed with implementation of other features) of features, and nesting depth of annotations [53]. Documenting features and their locations immediately and continuously during development benefits from the developers' fresh knowledge and using the embedded annotation approach provides that annotations co-



```
Featracer
    Application
    Repository
        DiffExtraction
        AssetRepresentation
        ProjectReader
        Utilities
            FileManager
            StringManager
            FeatureFinder
            Serializer
        DeveloperScenarios
        Statistics
    Parser
        FMParser
        ASTParser
    ML
        PreProcessing
            DataBalancing
            Normalization
```

```
Configuration.java,Main.java
Application
ProjectData.java,ProjectReader.java
ProjectReader
```

```
            if (projectData.getConfiguration().isIncludeScenarioCalculation()) {
                scenarioHandler.printScenarios();//&line [DeveloperScenarios]
            }
        }

    //&begin [Statistics]
    private void setStat(StringBuilder stringBuilder, AssetType assetType, A
        stringBuilder.append(String.format("%sS: %d\n", assetType, projectDa
        stringBuilder.append(System.lineSeparator());
        stringBuilder.append(String.format("%s ANNOTATIONS: %d\n", assetType
        stringBuilder.append(System.lineSeparator());
    }
    //&end [Statistics]
```

Figure 1.5: Embedded feature annotations

evolve with assets, for instance, when assets are cloned or moved to different locations within the project [20]. However, despite being cheap and robust, this approach requires developers to annotate assets continuously and that they do not forget too many annotations [20]. Moreover, for large systems, developers may get overwhelmed and tend to forget to annotate their source code. To understand how best developers can be supported to trace features during develpment, we implemented and systematically evaluated a recommender system (FeaTracer) that catches cases when annotations are missed during commit revisions, and reminds developers to annotate by suggesting the missed feature locations. FeaTracer uses state-of-art machine learning algorithms to analyze commit change-sets and make recommendations. In this case, each asset can be mapped to multiple features, hence, the machine learning classification algorithms we apply are capable of multi-label learning.

### 1.1.4 Recommender Systems and Multi-label Classification

Recommendation systems are widely used in several domains such as e-commerce and entertainment to suggest, for instance, items to purchase or movies to rent; but also in the field of software engineering [54, 55], they are used for tasks such as suggesting bug-fixes, code snippets, and associated requirements. Underlying these systems are machine learning algorithms that either predict continuous (regression) or discrete (classification) values. The algorithms can either be supervised (i.e., need training examples to predict new instances) or un-supervised (i.e., use clustering). For supervised machine learning, a training dataset consists of example instances with their characteristics (a.k.a features, metrics, or attributes) mapped to target classification labels or regression values. In our study (Paper D), we use classification algorithms: the instances to be classified are source-code assets such as folders, files, code fragments, and lines of code, and their target class-labels are the software features that they implement. For instance, in Figure 1.5, files *ProjectData.java* and *ProjectReader.java* are both mapped to feature *ProjectReader*. Hence in a training dataset consisting of files as training examples, the target class labels for both file-instances would be *ProjectReader*. Similarly, a training dataset with code fragments as instances would have the block of code, *line*

Table 1.1: Example dataset with four attributes and three labels

| | CSM | SLD | NEA | COMM | server | client | bubbleGraph |
|---|---|---|---|---|---|---|---|
| $E_1$ | 0.5678 | 0.5 | 4 | 5 | 1 | 0 | 1 |
| $E_2$ | 0.2346 | 1 | 2 | 4 | 1 | 0 | 0 |
| $E_3$ | 0.9678 | 0.6 | 4 | 8 | 0 | 1 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| $E_n$ | 0.452 | 0.354 | 2 | 5 | ? | ? | ? |

last three columns are the labels (software features mapped to instances)

*349 to 354 in file ProjectReader.java*, mapped to target label *Statistics*. These instances exhibit several characteristics (classification metrics) which are used by the learning algorithms to predict a feature location for an asset that is not annotated. Examples of these metrics include the cosine similarity of the text within each asset, and the structural location distance between assets of the same feature. Intuitively, the set of metric values (*a.k.a feature vector* in machine learning terminology) for each unlabeled asset instance is used as input to the learning algorithm to predict the set of software features (class labels) that the asset should be annotated with, based on patterns learned from the metric values of the training examples previously presented to the learning algorithm.

Thus, formally, in supervised machine learning, a classification task involves learning from examples associated with one or more labels and later making predictions of labels for unknown instances. Let $D$ be a multilabel dataset (MLD) composed of $N$ examples $E_i = (x_i, Y_i), i = 1..N$. Each example $E_i$ is associated with a feature vector $\mathbf{x_i} = (x_{i1}, x_{i2}, ..., x_{iM})$ described by $M$ features (metrics) $X_j, j = 1..M$, and a subset of labels $Y_i \subseteq L$, where $L = \{y_1, y_2, ...y_q\}$ is the set of $q$ labels. Table 1.1 presents an example MLD with labels from one of our subject systems studied in Paper C — Clafer Web tools; with the feature vector $\mathbf{x} = (CSM, SLD, NEA, COMM)$ and the set of labels $L = \{server, client, bubbleGraph\}$. Here, each learning example ($E_i$) could be a folder, file, fragment, or line of code, depending on the chosen abstraction level of the dataset. Example $E_1$ has for its feature vector $\mathbf{x_1} = (0.5678, 0.5, 4, 5)$ and its labelset $Y_1 = \{server, bubbleGraph\}$. In this scenario, the multi-label classification task comprises generating a classifier $H$ which, given an unseen instance $E = (x, ?)$ (c.f, $E_n$ in Table 1.1), is capable of accurately predicting its subset of labels $Y$ , i.e., $H(E) \rightarrow Y$. The classification task is called *binary* if the output is *YES/NO*, *multi-class*, if, from the set of labels $L$, only one can be associated with the unseen instance (i.e., $|Y| = 1$), and *multi-label* if $|Y| \geq 1$. Multi-label learning finds its application in several domains, such as text mining [56], protein analysis [57], and media classification through pattern recognition [58].

There are two approaches to accomplishing a Multi-label classification task: *problem transformation* and *algorithm adaptation*. The former works by producing, from a MLD, a group of datasets that can be processed with traditional single-label classifiers, while the latter aims to extend existing algorithms to handle multi-label classification problems [59]. Examples of *problem transformation* approaches include Binary Relevance and Label Powerset, while Instance-based Logistic Regression and Multi-label k-Nearest Neighbors are some examples of learning algorithms adapted for multi-label problems. Our study uses classifiers from both approaches.

For any given unknown instance, a multi-label classifier outputs either i) a *bipartition* of relevance for each label (i.e. TRUE for relevant and FALSE for not relevant), or ii) a *ranking* of labels according to their relevance for the instance, or iii) both. All the classifiers used in our study are capable of giving both outputs.

## 1.2 Methodology

In this section we detail the methodology we used to address the research questions formulated for this thesis.

As stated above, the overarching goal of this thesis is to better understand variability-aware analysis in low-maturity variant-rich systems. To accomplish this, we worked in close collaboration with industry to gain insights into state-of-practice, challenges and needs, and also engineered solutions meeting our objectives.

For Paper A, we conducted an empirical assessment of the needs and practices for assuring variant-rich systems — highly configurable systems in particular. We combined a survey with 27 employees of companies from 8 countries with in-depth interviews of 15 of the survey participants. The company sizes ranged 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 features. Our study design relied on categorizing analysis techniques from the literature and identifying properties analyzed by them (c.f Figure 1.3); we used these to elicit the need for and the criticality 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.

For Paper B we aimed at understanding what information sources are available and suitable for recovering feature locations and facets. To address this, we conducted an exploratory study on two open-source systems: Marlin, a variability-rich 3D printer firmware, and Bitcoin-wallet, an Android application for bitcoins, both of which comprise several information sources and variability mechanisms.

For Paper C we aimed at understanding how best to support developers in tracing feature locations during development using a recommender system — FeaTracer. To this end we followed the design science approach [60] to design, develop and validate FeaTracer through several internal iterations. We evaluated several multi-label classification algorithms and metrics through different experiments and configurations using five open-source systems with feature-annotated source code. The five systems comprised the four clones of Clafer Web tools [61] (ClaferMooVisualizer, ClaferConfigurator, ClaferIDE, and CommonPlatformUITools) and Marlin[1] 3D-printer firmware. Even though Marlin uses only variability annotations (i.e., preprocessor directives that wrap only optional features) and not the embedded feature annotation approach

[1] http://marlinfw.org

that traces all features whether mandatory or optional (see Section 1.1.3), still, our study leading to Paper B revealed that Marlin's development process is mostly feature-oriented and that the majority of its features is optional, hence developers wrap them with preprocessor macros. Thus, in the absence of more systems using the embedded annotation approach, we considered Marlin to be a suitable subject system to evaluate FeaTracer.

For Paper D we aimed at understanding how features are used in non-modular variant-rich systems. Particularly, we focused on understanding what circumstances lead to feature scattering and how developers cope with it. For this we conducted a case study of the Linux kernel — one of the longest-lived highly configurable systems with over 13,000 features and over 10 millions SLOC. We first conducted a longitudinal analysis of the source code covering almost eights years of evolution of the kernel to investigate trends of feature scattering (from version 2.6.12 to 3.9). We then complemented this analysis with a survey involving 74 kernel developers and maintainers, and interviews with 9 of them. The survey and interviews were focused on understanding developer practices, circumstances, and perceptions of feature scattering.

## 1.3 Summary of Papers

We now present a summary of each of the papers comprising this thesis by briefly stating its aim and contributions.

### 1.3.1 Paper A

To address varying stakeholder requirements, organizations often create several variants of their systems. These variants are either realized through the *clone&own* approach or by means of a configurable platform. The latter constitutes a group of variant-rich systems that are highly configurable, such as software product lines [62,63] and personalization-capable systems — especially in the automotive, avionics, telecommunication or power-electronics domain. Highly configurable systems are complex pieces of software that exhibit thousands of configuration options (features), leading to almost infinite configuration spaces (possible number of variants). One such example is the Linux kernel [64] boasting of around 15,000 configuration options, supporting different hardware architectures, software features or runtime environments ranging from Android phones to large supercomputer clusters. Thus, engineering highly configurable systems is challenging due to variability — the number of configurations and system variants grows exponentially with the number of configuration options.

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 been well adopted in industrial practice — consider the product line hall of fame (splc.net/hall-of-fame) and case studies [65,66] — this is much less clear for product-line *analysis* techniques. However, hundreds of dedicated analysis techniques have been conceived, many of which are able to analyze system properties for all possible system variants, 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 feature specifications and artifacts is critical, but that the majority of analyses for feature specifications (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.

Our main contributions comprise: (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 (Figure 1.3) 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* [67].

### 1.3.2 Paper B

Developers commonly use features to define, manage, and communicate functionalities of a system [1,2]. Unfortunately, the locations of features in code and other characteristics (feature facets), relevant for evolution and maintenance, are often poorly documented [68]. Since developers change and knowledge fades with time, such information often needs to be recovered. In fact, feature location [16, 69–72] is one of the most common and expensive activities in software engineering [12–14, 20]. Several automated techniques have been proposed to recover features and their locations [16, 17, 71, 73]. Unfortunately, these techniques generally exhibit a low accuracy, need substantial effort (e.g., calibration and adaptation for specific projects), and often only exploit a single source of information, such as execution traces or code comments. Other feature facets, such as the rationale or architectural responsibility of a feature, are even more difficult to extract, as corresponding information sources are largely unknown and developers may have varying understandings of these facets.

Hence, to improve techniques for feature location and for recovering feature facets, we need to improve our empirical understanding of features. This includes knowledge about information sources we can utilize for these purposes, about strategies to exploit these information sources, and about the facets of features. Particularly interesting are *modern open-source projects* that are developed on software-hosting platforms, such as GitHub and BitBucket, which provide additional capabilities for maintaining and documenting a project. Such platforms boast a richness of different information sources (e.g., pull requests, change logs, release logs, commits, Wikis, issue trackers) from which such information can be recovered — and that can be present in similar form in industrial settings.

However, it is largely unknown from what information sources features, their locations, and their facets can be recovered. We present an exploratory study on identifying such information in two popular, variant-rich, and long-living systems: The 3D-printer firmware Marlin and the Android application Bitcoin-wallet. Besides the available information sources, we also investigated the projects' communities, communications, and development cultures. Our results show that a multitude of information sources (e.g., commit messages and pull requests) is helpful to recover features, locations, and facets to different extents. Pull requests were the most valuable source to recover facets, followed by commit messages, and the issue tracker. As many of the studied information sources are, so far, rarely exploited in techniques for recovering features and their facets, we hope to inspire researchers and tool builders with our results.

Overall, we contribute: (i) an *analysis of the development process* of the open-source systems Marlin and Bitcoin-wallet; (ii) a *set of consolidated search patterns* to identify and locate features; (iii) *empirical data on the facets* of the identified features in both systems; and (iv) an *online appendix*[2] containing the feature fact sheets, feature models, and annotated code bases.

### 1.3.3 Paper C

Our study in Paper A showed that one major reason why quality assurance of low-maturity variant-rich systems is challenging is that essential meta-information, such as relationships between features and their implementation in source code, is often missing. To recover this information, developers carry out an activity called *feature location* in which they retroactively trace features to their implementation. Feature location is one of the most common activities in software engineering [16, 69–72]. Unfortunately, it is also a very expensive activity when performed manually [12–14, 20]. Even though several automated techniques have been proposed to recover features from source code, feature location remains a challenging problem in software evolution and maintenance since the proposed techniques are often inaccurate when used in practice, or demand much effort from developers [16, 17, 71, 73]. We argue that features are very domain-specific entities; each project uses its own notion, and developers have a different understanding of features and use them differently across projects [68]. Hence, to effectively trace them to their implementation, developers have to record feature-asset traceability information themselves; but one question remains: how to record this information.

We propose a new technique and tool (FeaTracer) to tackle the feature location problem, relying on some core ideas: (i) embedded annotations [20, 21] that wrap source-code assets with feature names; these annotations are easy to apply and known to naturally co-evolve with software assets, thus reducing maintenance effort [20], (ii) continuous recording of annotations by developers during development, and (iii) learning from those recordings to support developers in tracing feature implementation while the knowledge is still fresh in their minds.

We conducted several experiments aimed at understanding how best FeaTracer can support developers' feature location activities through machine-learning-based

[2] https://bitbucket.org/rhebig/jss2018/

recommendations. Our study relies on running our prediction experiments on the revision history of repositories with annotated source code assets. Our first subject system is a set of four cloned projects, collectively called **ClaferWebTools** [74], whose development history has been simulated [20] by adding embedded annotations of features at each revision. The four projects are ClaferMooVisualizer (viz), ClaferIDE (ide), ClaferConfigurator (config), and ClaferCommonPlatformUITools (tools); the last project being an integration of the first three. ClaferWebTools is predominantly JavaScript-based. Our simulation experiments cover 3 years of its development (2012-2014), comprising a total of 742 commits. However, we only generated datasets for commits with annotated assets — 351 commits. The second system is **Marlin** — a 3D printer firmware exhibiting rich variability through preprocessor annotations. Even though Marlin does not use embedded annotations that wrap both mandatory and variable features, but uses preprocessor annotations (e.g., #ifdef) that wrap only variable features, our study in Paper B found that Marlin's development process and culture is feature oriented and that most of its features are optional, hence developers wrap them with preprocessor directives. Thus, in the absence of many systems with embedded annotated assets, we deem Marlin a suitable candidate for evaluating FeaTracer. In Marlin's case, we focused only on boolean features. Our analysis covers 2 years of its development (2011-2012), comprising 500 commits authored by 36 developers.

To align our evaluation with a real development scenario, we simulated development by training classifiers on data generated from each $n$th commit and made predictions for all assets that were changed in the subsequent $n+1$th commit. For instance, we trained using the first commit, and predicted in the second, then trained in the second commit and made predictions for all changed assets in the third commit, and so on. The predictions were made at four different granularity levels (folder, file, fragment (multiple consecutive lines), and line-level). We then analyzed the performance of the classifiers and selected the best performing for each granularity level.

Overall, we contribute: (i) a technique and tool called FeaTracer to alleviate the feature-location problem, (ii) empirical data on what multi-label learning algorithm and classification metrics best support feature location for what source code granularity level, and (iii) a replication package with further study details in an online appendix [67].

### 1.3.4 Paper D

Scattering of feature code is commonly perceived as an undesirable situation [75–78]. Scattered features are not implemented in a modular way, but are spread over the code base, possibly across subsystems. The tangling of scattered features with different implementation parts can lead to ripple effects and require frequent developer synchronization, which challenges parallel development. Scattered features may significantly increase system maintenance efforts [22,79]. Yet, feature scattering is common in practice [39,40,80] as it allows developers to overcome design limitations when extending a system in unforeseen ways [22] or when circumventing modularity limitations of programming languages, which impose a dominant decomposition [81–83]. In other cases, the cost of modularizing features might be initially prohibitive or simply too difficult to be handled in practice [84]. In contrast, feature scattering requires little upfront investment [79], although maintenance costs may rise as the system evolves. Many long-lived and large-scale software systems have shown that it is possible to achieve continuous evolution while accepting some extent of feature scattering. Examples span different domains, such as operating systems, databases, and text editors [39, 80, 85].

Surprisingly, there are no empirical studies investigating feature scattering in large and long-lived software systems. Such studies are key in creating a widely accepted set of practices to govern feature scattering and may eventually contribute to a general scattering theory, which could serve as a guide to practitioners — for instance, in identifying implementation decay [86], assessing the maintainability of a system [24], identifying scattering patterns [87] or setting practical scattering thresholds [80].

To contribute to a deeper understanding of feature scattering and its evolution, we present a case study of one of the largest and longest-living software systems in existence today: the Linux kernel. Its features are manifested as compile-time configuration options that users select when deriving customized kernel images. The Linux kernel is the operating system kernel upon which free and open-source software operating system distributions, such as Ubuntu, OpenSUSE, Fedora and Android, are built. Its deployment goes beyond traditional computer systems, such as personal computers and servers, to embedded devices, such as routers, wireless access points, and smart TVs, as well as to mobile devices. Introduced in 1991, the Linux kernel boasts over twenty-seven million source lines of code (mostly written in C), and 12,000 contributors from more than 200 companies. Our analysis covers evolution, practices, and circumstances leading to feature scattering. We first conducted a longitudinal analysis of the source code to obtain feature scattering trends and followed up with a survey of 74 kernel developers and interviews with 9 of them.

Due to the sheer size of the kernel, we scoped our longitudinal analysis to features of the *driver* subsystem, which we identified as the largest and fastest growing kernel subsystem. We analyzed the scattering of driver features within and across the device-driver subsystem and followed up with developers and maintainers through a survey and interviews, to understand their practices, circumstances, and perceptions of feature scattering. To obtain a broader set of opinions on general issues of scattering, the survey and interviews were not limited to the driver subsystem.

Our contributions comprise: (i) a *dataset covering almost eight years of the evolution of feature code scattering* extracted from the Linux kernel repository (from version 2.6.12 to 3.9). It serves as a replication package, as a benchmark for tools, and for further analyses; (ii) *Empirical data from a survey and interviews* aimed at understanding the state of practice of feature scattering in the Linux kernel; (iii) An *online appendix* [67] with further details on our dataset, scripts to analyze the data, and additional statistics.

## 1.4 Results

We now answer our research questions based on our contributions.

**RQ1:** ***What are industrial practices for assuring the quality of***

***variant-rich systems and what properties are assured?*** **(Paper A)**

To answer this question, we conducted a survey with 27 practitioners and followed up with interviews with 15 of them. We elicited the perceived severity and reasons for analyzing the properties we identified from the literature (in the categories shown in Figure 1.3) and those expressed by the practitioners. Furthermore, 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.

The subject systems represented by our survey respondents and interviewees were from a wide range of domains, mainly automotive, industrial automation, and aerospace and defense. Their sizes ranged from less than 25,000 lines of code to over one million lines of code, containing between ten to over 10,000 features. This 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, 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.

**Properties assured**. With regard to properties that are assured, first, the severity that our practitioners express for the common properties suggested in the literature confirms their relevance for highly configurable systems. For instance, reliability, performance, behavioral correctness, and safety are the top properties perceived highly critical to assure (possibly influenced by domains of our subject systems). However, most of the analyses targeting properties of the feature specification are not seen as important by our practitioners. Figure 1.6 shows survey responses indicating which of the top 8 feature specification analyses from Benavides et al.'s survey [29], including three change-impact analyses on feature specifications [88], are perceived critical by our survey respondents. As indicated, only two of those eight properties are considered critical to assure, the first being whether the specification is satisfiable (at least one valid variant exists given feature constraints) and the second being that any given configuration (variant) is valid (i.e., satisfies constraints). Furthermore, 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 between the feature specification and source code) is considered highly critical, as well as identifying unwanted feature interactions.

**Practices**. With regard to practices, 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 feature specifications required

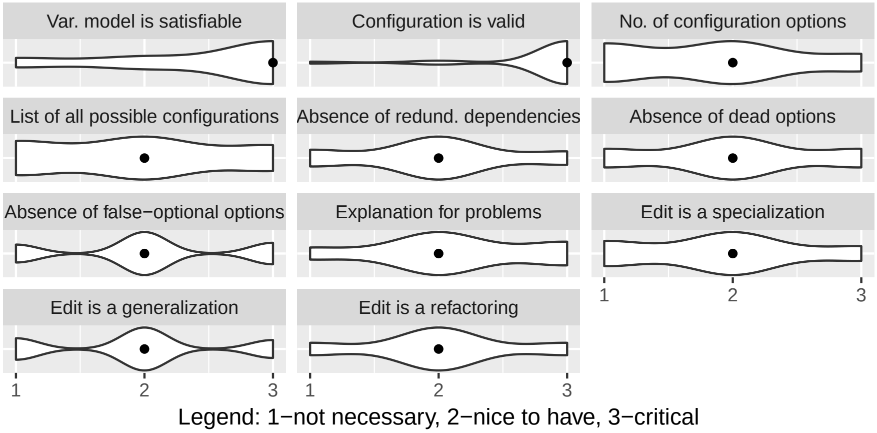


Figure 1.6: Importance of assuring various properties of the feature specification

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.

**RQ2:** ***What information sources are useful for recovering and locating features and their characteristics–feature facets?*** **(Paper B)**

To address this question we conducted an exploratory study (Paper B) on Marlin, a variability-rich 3D printer firmware, and on Bitcoin-wallet, an Android application for bitcoins, both of which comprise several information sources and variability mechanisms. First we studied the feature-development processes exercised by the Marlin and Bitcoin-wallet developers, to understand how features are developed in the two systems and communities. Next we systematically investigated what information sources help locating features, and to what extent. To this end, we focused on the differences between optional and mandatory features, as especially mandatory features are challenging to locate — variability annotations, such as `#ifdef`, only wrap optional features. Next, we manually analyzed the Marlin and Bitcoin-wallet Github documentation, such as release logs, and source code to investigate what search strategies help recovering features. As we adapted similar search strategies for each information source, we consolidated these into common patterns. Lastly, we investigated what information sources help identifying feature facets, and to what extent.

**Development process**. Marlin has a well-defined and structured development process for features and bug fixes. Several steps are concerned with quality assurance and, while everyone can contribute an issue or implement it, a subset of contributors is responsible for accepting them. Besides ensuring quality, this process also serves as detailed documentation and allows tracking changes and decision-making processes. We found that the primary means of communication are issue trackers and pull requests. Moreover, pull requests

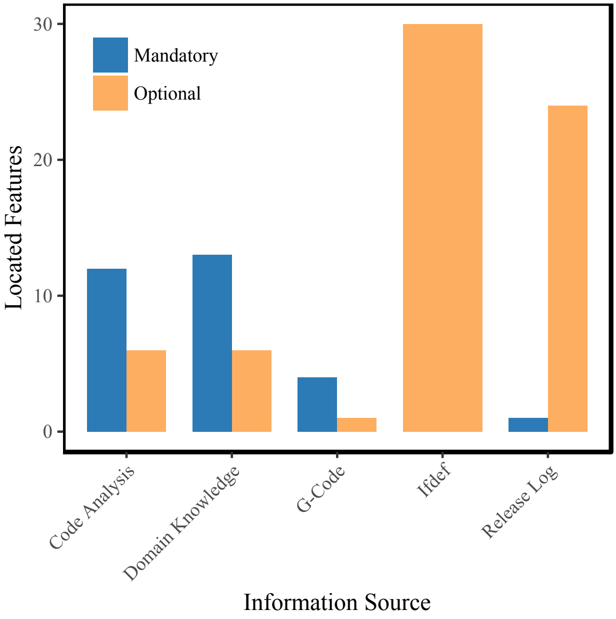


Figure 1.7: Information sources used to locate features in Marlin.

are linked to the release log, in which developers track development, quality improvements, and bug fixes of each release. Interestingly, pull requests are labeled and categorized by Marlin's developers, for example, as `PR:Bugfix`, `PR:Coding Standard`, and `PR:New Feature`. This illustrates the potential for improving automation for feature location and for recovering feature facets based on such modern information sources. Still, while we found a common notion of features for the Marlin community around which the communication is structured, this may not be the case for other systems.

It seems interesting to test techniques based on natural language processing to connect artifacts, such as source code, commits, and discussions — aiming to identify and locate features as well as their facets. If a common terminology is established in projects, this may allow to considerably improve automated analyses of legacy systems. Marlin has been developed for more than seven years (excluding its predecessors) and comprises more than 4,600 forks. Thus, this development process (and terminology) seems to be established and ensures constant, qualitative implementation of new features, while allowing the integration of third-party developers.

Our analysis of Bitcoin-wallet indicates that the same process is not applied on all open-source projects. However, Bitcoin-wallet has far fewer contributors, forks, and issues, indicating less popularity compared to Marlin. Thus, the differences in the development processes may not be due to a strict hierarchy or a developer keeping all responsibility, but simply due to the different scales.

**Information sources and search strategies**. Due to the existing notion of features being optional, Marlin's developers do not provide much information about mandatory features on the software-hosting platform or the release log. Consequently, these information sources are not suitable for locating this type of features. Besides the actual source code and its elements, mainly domain knowledge helped to identify mandatory features of Marlin — in our case heavily based on constructing the actual hardware. As a result, we argue that feature-location techniques can be improved by considering different types of documentation while analyzing the source code. Especially comments seem interesting, as they are directly connected to the corresponding source code in most cases. However, several questions arise, for example, how to ensure that the used documentation is maintained simultaneously with the code [89, 90]. Other domain-specific information sources may be helpful, such as the G-Code[3] commands in our study, but also require domain knowledge to identify them. Ultimately, we found five complementary information sources that were helpful to identify and locate features in projects that are maintained on software-hosting platforms, which we show in Figure 1.7:

- Domain knowledge (e.g., building two printers)
- Release log (i.e., pull requests, commits)
- Code analysis (i.e., comments, dependencies)
- `#ifdef`annotations
- G-Code commands

Using these information sources and a combination with other artifacts, such as models or requirements, can facilitate identifying and locating both types of features. In particular, we experienced that domain knowledge is necessary to identify features and to find their locations.

Unfortunately, Bitcoin-wallet does not provide such a rich set of entry points for feature location. Especially the missing linkage between the release log and code, the limited variability representation, and missing notion of features made it challenging to analyze the code. Unsurprisingly, we found it more challenging to track information for most features in Bitcoin-wallet compared to Marlin.

Our analysis indicates that different information sources require adapted search strategies, but can then facilitate the analysis. Consequently, we also have to adapt automated techniques accordingly. Regarding the artifacts we considered, this is rather unsurprising: Source code is differently structured and provides additional sources compared to the release log and its connected artifacts, except for the code differences stored in each commit. Still, the release log proved to be an effective and cheap way to identify and locate optional features in Marlin.

**Feature facets**. We investigated information sources for seven facets [68] namely: *rationale* (why a feature is introduced) , *architectural responsibility* (what part of the system a feature belongs to), *definition and approval* (how a feature is defined and approved for consideration), *binding time and mode* (when the feature is determined to be included in a variant, e.g., statically through preprocessor directives), *responsibility* (responsible developer), *evolution* (e.g., releases in which the feature is included), and *quality and performance* (quality properties targeted by the feature).

Overall, we found that different information sources can be helpful for each feature facet. Most of these information sources are only available in

[3] https://marlinfw.org/meta/gcode/

modern software-hosting platforms, but provide good opportunities to improve automated techniques to recover feature facets. Still, as comparing Marlin and Bitcoin-wallet illustrates, the usability of each information source for a facet depends heavily on its usage, the development process, community, and domain of the system.

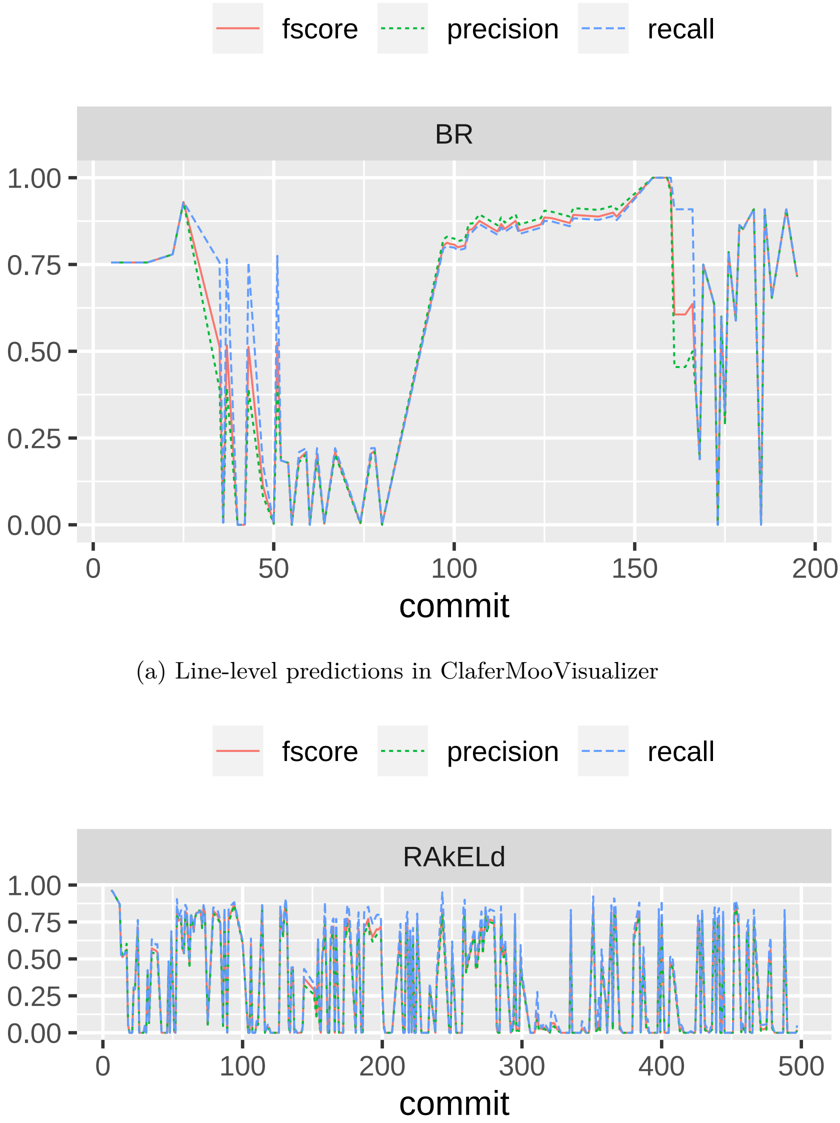
**RQ3:** ***How best can developers' feature location activities be proactively supported during development?*** **(Paper C)** We addressed this research question by designing and evaluating a recommender system (FeaTracer) that offers suggestions for missed feature locations when developers forget to annotate assets during development. We evaluated FeaTracer using two open-source systems: a web-based application for managing cloned variants called ClaferWebTools (JavaScript-based), and a 3D-printer firmware called Marlin (C/C++-based). Our goal was to understand (i) which of the four source-code granularity levels (folder, file, fragment, and line-level) is more reliable when offering suggestions for missed feature locations, (ii) what multi-label learning algorithm is more accurate when predicting feature locations, and (iii) which language-independent metrics best characterize assets of a given feature to offer more accurate recommendations? To characterize features and their related assets, FeaTracer uses four metrics: cosine similarity (CSM) that compares how similar the textual content of assets belonging to a given feature is; source-code location distance (SLD) that measures how close to each other assets of a given feature are; number of existing annotations (NEA) that counts the total number of features implemented by each asset belonging to a given feature; and, number of commits (COMM) in which each asset belonging to a given feature has been changed — this measures how often assets of a given feature are changed.

To investigate which granularity level is more reliable, we generated training and test datasets at four granularity levels (folder, file, fragment, and line level) and evaluated the performance of different classifiers on predictions made at each level. We also analyzed characteristics of the different datasets, such as number of learning examples, average number of labels per instance (a.k.a, label cardinality), and imbalance ratio (i.e., ratio between the most frequent and rare labels). To investigate the most accurate multi-label algorithm for predicting feature locations, we evaluated the performance of 5 multi-label classification algorithms from the MULAN Java library, namely, Binary relevance (BR), Label Powerset (LP), Instance-Based Logistic Regression (IBLR), Multi-label k-Nearest Neighbors (MLkNN), and Random k-Labelsets of disjoint sets (RAkEL-d). We arrived at these five after an exploratory stage in which we filtered out others, e.g., those classifiers that could not work due to exceptions we could not fix, or were too slow to fit our recommendation scenario which requires fast feedback to the developer. Since our aim is to support developers with recommendations whenever they commit changes to their repositories, we did not perform cross-validation but instead based our evaluation of the classifiers on actual predictions for all assets changed in every $n + 1$th commit, using training data from the $n$th commit. Lastly, we applied feature (metric) selection techniques to understand which metrics best characterize features and their related assets to offer more accurate recommendations. We used multi-label feature selection methods proposed by Spolaôr et al. [91], which rely on the *filter approach* [92]. Filter methods are widely used in research related to multi-label learning [93]. They use general characteristics of the dataset to select some features and exclude others, independently of the learning algorithm. As such, they may not choose the best features for specific learning algorithms. For each pair of the training and test dataset, we alternated the combination of metrics and evaluated the performance of our selected classifiers.

Overall, we found that line-level predictions are more accurate and reliable since lines of code offer more learning examples for FeaTracer than fragments, files and folders. For instance, in one of our evaluation projects (ClaferMooVisualizer), the average number of fragments in a training dataset was 9, over a course of 192 commits, while that of lines was 248, and that of files was 2. Nonetheless, FeaTracer offers recommendations for all granularity levels. Furthermore, we observed that file and folder level datasets had the lowest imbalance ratios (we recorded zero imbalance ratio for all ClaferWebTools), and that line-level datasets were more imbalanced than fragment-level datasets. Of all our evaluation projects, ClaferMooVisualizer was the least imbalanced; which could explained why we had higher accuracy scores here than in other projects. With regard to best performing classifier, we found that different classifiers are best suited for the different granularity levels. For instance, Label Powerset is more suited for file and folder level predictions (achieving an average accuracy of 81 % in ClaferMooVisualizer), while Binary Relevance (BR) and Random k-Labelsets of disjoint sets (RAkELd) are more suited for fragment and line level predictions (achieving an average recall of 61 % and precision of 57 % on line level predictions), and were found to be the most robust against few learning examples. This result indicates that no one classifier is best for all granularity levels and that FeaTracer may benefit from ensemble approaches that combine different classifiers. The use of feature selection methods did not yield significant benefits, except in one project (ClaferIDE) where there was a 30 % increase in accuracy when we used the top two metrics instead of all four. In general, however, we found that the best results were obtained when all four metrics were used. Our metric selection technique showed CSM and SLD to be the top two metrics for folder, file and fragment-level datasets, while NEA and SLD were ranked the top two metrics for line-level datasets.

As shown Figure 1.8, we observed certain developer practices that affect the accuracy of FeaTracer, these being either refactorings that significantly changed characteristics of the learned assets, or the addition of several new assets outnumbering existing assets. For instance, in the line-level datasets for Marlin (Figure 1.8b), we observed that from the 4th commit to the 13th commit, the number of annotated lines of code is steady between 99 and 172. However, from the 14th commit the number rises to 774. From the 19th commit, there is another sharp rise from 727 to 4,904 annotated lines added, hence, the noticeable sharp drops in prediction accuracy we observed. We observed similar patterns in ClaferWebTools as shown for ClaferMooVisualizer in Figure 1.8a. This development pattern is expected since developers often refactor code, clone, or import files into their projects, hence, we believe our accuracy values are within reasonable range to be able to support developers in tracing features to their implementation.

**RQ4:** ***How does feature scattering evolve and what are practices and circumstances leading to it?*** **(Paper D)** We addressed this research question in Paper D by empirically investigating the impact of feature scattering



(a) Line-level predictions in ClaferMooVisualizer

(b) Line-level predictions in Marlin

Figure 1.8: Trends of line-level predictions

on the maintenance of a large and long-lived software system — the Linux kernel. The first part of our study (longitudinal study) targeted analysis of the kernel's code to investigate trends of feature scattering (how feature scattering evolves with the evolution of the kernel), and the second part, as a follow-up to the first, targeted developers of the Linux kernel to investigate their perception of feature scattering and its impact on maintenance effort of the kernel, and how they cope with it.

For the longitudnal analysis, we covered almost eight years of evolution of the kernel, from verison 2.6.12 to 3.9, in which period the kernel saw steady growth from 4,752 features to 13,165 features. As stated above (Section 1.3.3), we scoped our analysis to the source code of *driver* subsystem, which is the largest subsystem of the kernel. To understand how feature scattering evolves, first, we analyzed the relative and absolute growths of scattered and non-scattered driver features — for instance, to understand whether the proportion of scattered features is increasing, decreasing or stable. Next, we analyzed how the growth of locally scattered features differs from globally scattered features. To this end we analyzed the relative and absolute growths of driver features that are scattered (i) within the *driver* subsystem only (local scattering) and (ii) across, at least, another subsystem (global scattering). We aimed at understanding how scattering is related to the kernel's architecture. Lastly, we investigated the extent to which feature code scattering evolves over time. Here, we analyzed the extent (degree) of the scattering of feature code, aiming at understanding the underlying distribution and possible thresholds, as well as how this degree relates to local and global scattering.

For the follow-up study with kernel developers, first, we investigated both possible causes and circumstances leading to feature scattering by analyzing the survey and interview data. We also identified and asked the interviewees about examples of scattered code that they developed and that we identified as such in the kernel's codebase. Furthermore, we studied whether certain kinds of features are more likely to be scattered. Second, we analyzed the survey respondents' and interviewees' reported practices for coping with feature scattering and whether developers consciously maintain a scattering threshold for the number of scattered features or for the features' scattering degrees.

Overall, we found that: First, the majority of driver features can actually be introduced without causing scattering and that the number of scattered features remains proportionally nearly constant throughout the kernel's evolution. We also found that scattering is not limited to subsystem boundaries and that the implementation of the majority of scattered *driver* features is scattered across a moderate number of four to eight locations in the code. Second, that developers introduce scattering in the Linux kernel, among other reasons, to avoid code duplication and to support hardware variability, backwards compatibility, and code optimization. We also learned that the features that are most prone to scattering are those relating to platform devices — devices that cannot be discovered by the CPU as opposed to hotplugging ones. And third, that developers try to avoid feature scattering mostly by adhering to coding guidelines that alleviate the problems of preprocessor use (e.g., use of static in-line functions) and refactoring existing code to, for instance, improve system architecture, but the majority do not consciously maintain a scattering threshold.

## 1.5 Threats to Validity

In this section we discuss threats to the validity of our research based on definitions by Wohlin et al. [94].

### 1.5.1 Construct Validity

For RQ1, we used concepts and terms that our survey participants and interviewees could understand to mitigate potential misinterpretations. For instance, we used the concept of highly configurable system to ensure that all practitioners could describe their practices without the need to adopt a specific terminology and 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. We also iteratively developed our questionnaire and our interview guide using pilot runs with industrial participants. To ensure completeness, participants could provide additional information.

For RQ4, our measurement of feature scattering in code relied on a simple metric (*SD*), that is low level and measures the parts related to feature code as specified by the original developers (using pre-processor directives). Hence we consider it as a reliable measurement of feature scattering.

### 1.5.2 Internal Validity

For RQ1, 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. All subjects were very open about their current limitations and had no incentive to present their current practices in a better light. Even though the interviews were conducted by different researchers, the recordings were exchanged for transcription and for coding to avoid potential biases.

For RQ2, since we did not involve original system developers to perform feature location tasks, we mitigated the risk of potential bias by having two authors become domain experts for each system; for instance, by assembling two different kinds of 3D printers (i.e., Delta, Cartesian) for Marlin, which differ in their mechanics and algorithms. We also performed domain, system, and community analyses, during which different authors extensively read documentations (e.g., about G-Code commands) and meta-data (e.g., issue tracker) available in the GitHub repositories. The source code was also analyzed in pairs, which includes cross-checking of the code understanding and of the locations.

For RQ3, to mitigate the risk that bugs in FeaTracer impact results, we performed extensive reviews to ensure that metrics and related accuracy measures are calculated as expected. All three authors held several meetings to review the implementation of FeaTracer and results obtained from the evaluation.

For RQ4, firstly, we performed extensive code reviews to mitigate the threat of bugs in our custom-made tool impacting our results for the longitudinal code analysis. Secondly, to avoid limiting conclusions to individual perspectives, the survey covers a broad range of roles of respondents that contribute to more than one subsystem of the Linux kernel. In addition, owing to the substantial technical and industrial experience of our interviewees, our work provides both a general perspective on feature scattering as well as in-depth insights on technical issues.

### 1.5.3 External Validity

For RQ1, 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.

For RQ2, we only considered two systems (Marlin and Bitcoin-wallet), which may differ from others. However, Marlin is a substantial case, and as an embedded system, it shares characteristics with many other embedded systems. In fact, preprocessors are used similarly in almost all open-source and industrial C/C++ systems [95]. Similarly, Bitcoin-wallet is an Android application that shares common characteristics with others and, thus, should also be representative.

For RQ3, even though our evaluation relies on two systems from two domains — web application development with JavaScript (ClaferWebTools), and embedded systems development with C/C++ (Marlin) — FeaTracer's feature location approach is language independent and can be applied to any project in any domain. Since each system and domain is different, FeaTracer can easily be adapted to each specific project by letting it learn the annotations of the project and use them for feature location recommendations. For systems with large numbers of labels, FeaTracer can be configured to use more scalable classifiers only, such as RAkELd, instead of Binary Relevance or others that may not be appropriate.

For RQ4, our study had only one subject system (the Linux kernel). Still, it is one of the largest open-source projects in existence today. Furthermore, our focus on device drivers is justified by the insight that it is the largest and most vibrant subsystem of the Linux kernel. Despite this focus, we study scattering not only within this subsystem, but also investigate how device-driver features affect the other subsystems of the kernel. Furthermore, the majority (66 %) of our survey respondents and interviewees work as professional developers in different companies besides contributing to the Linux kernel. Hence the insights they provided on feature scattering may not be specific only to the Linux kernel but may be applicable to other systems.

### 1.5.4 Conclusion Validity

For RQ1, 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.

To enhance the repeatability and reliability of our study answering RQ2, we provide the data set with feature locations and all other data in an online appendix.[2] We argue that other researchers can replicate our study, but may derive other results. For example, due to Marlin's evolution, they may categorize features differently, or include additional information sources (e.g., developers).

## 1.6 Conclusion

This thesis aims at understanding variability-aware analysis in low-maturity variant-rich systems. To fulfill this goal, we conducted a combination of three knowledge-seeking studies and one solution seeking study. Firstly, we present an investigation of industrial practices and needs for analyzing variant-rich systems (particularly, highly configurable systems), to improve our understanding of how and whether existing analysis techniques are applied in practice. This study reveals that most existing variability-aware analysis techniques can not be applied in industrial practice because feature specifications required for such analyses do not exist or are not expressed in a form that can be used as input. Secondly, we present information sources and search strategies that can be useful for recovering feature locations and feature facets, thereby contributing to a better empirical understanding of features that is relevant for developers and automated feature location techniques. This study reveals that several information sources, such as commit messages and pull requests, are helpful to recover features, locations, and facets to different extents. However, pull requests are a most valuable source to recover facets, followed by commit messages, and the issue tracker. Thirdly, we present results of our investigation to understand how best developers can be supported when tracing feature locations during development by means of a recommender system. We show that lines of code are more reliable for recommendations since they offer more learning examples for classification algorithms than fragments or files do. We also show that different classifiers perform best for different granularity levels; for instance that Label Powerset performs better for file and folder levels while Binary Relevance performs better for predicting line-of-code annotations. Hence the use ensembles may improve results. Furthermore, some practices, such as commit policy (sizes of change-sets), can negatively impact prediction accuracy. Lastly, using a case study of feature scattering in the Linux kernel, we present our understanding of how features are used in non-modular variant-rich systems. We show that, even though full modularity is difficult to achieve, still, even in large and long-lived systems, such as the Linux kernel, the majority of features is introduced without causing scattering, and that developers scatter feature code to address a performance-maintenance tradeoff (alleviating complicated APIs), hardware design limitations, and avoid code duplication.

## 1.7 Outlook to the Second Part of the PhD Project

As stated above, several variability-aware analysis techniques have been proposed that either operate statically [3, 33] (e.g., type checking, model checking, and theorem proving) or dynamically [4, 31, 32] (e.g., testing). Many of these techniques are single-system analysis that have been lifted (made variability-aware) to operate on product lines (level 6 of Figure 1.4). Different strategies have been applied by each technique to reduce analysis effort when considering variants of a product line. Thüm et al. [3] present a survey of and discuss these different analysis strategies. For instance, some techniques (a.k.a., family-based) apply the analysis to the whole product line by analyzing domain artifacts (reusable software assets) and take into account feature constraints from the variability model (formal feature specification), while others (a.k.a product-based) analyze individual generated products. Depending on the strategy used, some of these analysis can be adapted to low-maturity variant-rich systems. For instance, level 0 (ad hoc *clone&own*) variant-rich systems might benefit from product-line analysis techniques that use the product-based strategy and incorporate techniques for exploiting redundancy and improving scalability. However, it is unclear how this can be done since all these analyses target configurable platforms with formal feature specifications.

As a motivating example, we consider the study of Sattler et al. [96] in which they lift traditional data-flow analysis approaches to analyze and represent data flows of all possible combinations of apps for purposes of detecting privacy data leaks. Considering that mobile apps often process private data that may be leaked to untrusted parties, data-flow analyses are conducted to detect such leaks in the communication between components within an app and across apps. Certain combinations of apps, however, potentially create data-flows that are hard to detect when analyzing individual apps. Even though sophisticated tools exist that can analyze data-flows within individual apps (e.g., IccTA [97]) and across apps (e.g., DidFail [98]), none of them scale to large combinations of apps. This limitation mainly lies in the data-flow representation — the tools do not exploit redundancies (i.e., common parts of the graph such as commonly used *intents*) and do not consider variability, e.g., that an app can be installed or not. As such, both IccTA and DidFail rely on the assumption that the set of app components is known, invariable, and rather small [96]. To overcome this limitation, Sattler et al. used the concept of *presence conditions* [99], borrowed from product-line analysis [3] as well as variational data structures [100], to compress the data-flow graph and make it variational such that its generation and analysis scales better than its non-variational form (e.g., as used in DidFail). In this case, presence conditions were used to predicate the presence or absence of apps in a data-flow.

We believe that by investigating approaches used by studies such as Sattler et al.'s and other existing lifted product-line analysis techniques, we can devise common principles to guide the application of these variability-aware analyses (or their strategies) to low-maturity variant-rich systems. Therefore, for our future work we aim to *(i) investigate the principles governing the adaptation of existing variability-aware analysis techniques from single system analysis*, and *(ii) devise a process (or proof-of-concept framework) on how these principles can be applied to low maturity variant-rich systems.* With this framework, we can describe, for instance, what process can be used by organizations at level 0 (no reuse), or level 2 (cloning with features), etc., to adopt specific analyses. As such, the framework can serve as a guide to practitioners and tool developers as well as drive further research. While a plethora of variability-aware analysis techniques exist, our plan is not to design an adoption process framework for all, but rather focus on the most commonly used techniques — with industrial relevance. For instance, our study in Paper A found that testing is widely used, followed by static analysis, and that model checking and theorem proving are hardly used at all.

Furthermore, we also plan to integrate FeaTracer as a plug-in for a real-world IDE, such as IntelliJ IDEA or Eclipse, and evaluate its performance through experiments with human subjects over an extended development period (e.g., one month). In addition, we will also investigate developer practices when using the embedded feature annotation approach to record feature locations, and thus gain deeper insights on practicality as well as possible enhancements.

# Bibliography

[1] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines.* Springer, 2013.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” Software Engineering Institute Carnegie-Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, “A classification and survey of analysis strategies for software product lines,” *ACM Computing Surveys*, vol. 47, no. 1, pp. 6:1–6:45, Jun. 2014.

[4] E. Engström and P. Runeson, “Software product line testing—a systematic mapping study,” *Information and Software Technology*, vol. 53, no. 1, pp. 2–13, 2011.

[5] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, “An exploratory study of cloning in industrial software product lines,” in *CSMR.* IEEE, 2013.

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, “A survey of variability modeling in industrial practice,” in *VAMOS*, 2013.

[7] L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[8] U. P. Schultz and M. Flatt, Eds., *Generative Programming: Concepts and Experiences, GPCE’14, VÃďsterÃěs, Sweden, September 15-16, 2014.* ACM, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2658761

[9] F. Stallinger, R. Neumann, R. Schossleitner, and S. Kriener, “Migrating towards evolving software product lines: Challenges of an SME in a core customer-driven industrial systems engineering context,” in *PLEASE*, 2011.

[10] H. P. Jepsen, J. G. Dall, and D. Beuche, “Minimally invasive migration to software product lines,” in *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings*, 2007, pp. 203–211. [Online]. Available: http://dx.doi.org/10.1109/SPLINE.2007.30

[11] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, “Reengineering legacy applications into software product lines: A systematic mapping,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 2972–3016, 2017.

[12] J. Wang, X. Peng, Z. Xing, and W. Zhao, “How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study,” *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[13] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, “Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 420–432, Jun. 2007.

[14] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, “The concept assignment problem in program understanding,” in *ICSE*, 1993.

[15] J. Rubin and M. Chechik, “A survey of feature location techniques,” in *Domain Engineering.* Springer, 2013, pp. 29–58.

[16] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, “Feature Location in Source Code: A Taxonomy and Survey,” *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[17] A. Olszak and B. N. Jorgensen, “Understanding Legacy Features with Featureous,” in *Working Conference on Reverse Engineering*, ser. WCRE. IEEE, 2011, pp. 435–436.

[18] J. Wang, X. Peng, Z. Xing, and W. Zhao, “How developers perform feature location tasks: a human-centric and process-oriented exploratory study,” *Journal of Software: Evolution and Process*, vol. 25, no. 11, pp. 1193–1224, 2013.

[19] J. Krüger, T. Berger, and T. Leich, “Features and How to Find Them: A Survey of Manual Feature Location,” in *Software Engineering for Variability Intensive Systems: Foundations and Applications.* LLC/CRC Press, 2019, pp. 153–172.

[20] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, “Maintaining Feature Traceability with Embedded Annotations,” in *International Systems and Software Product Line Conference*, ser. SPLC. ACM, 2015, pp. 61–70.

[21] M. Seiler and B. Paech, “Using tags to support feature management across issue tracking systems and version control systems,” in *REFSQ*, 2017.

[22] M. P. Robillard and G. C. Murphy, “Representing Concerns in Source Code,” *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, 2007.

[23] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo, “Feature-Oriented Software Evolution,” in *International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS, 2013.

[24] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto, “On the maintainability of aspect-oriented software: a concern-oriented measurement framework,” in *CSMR*, 2008.

[25] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement.* John Wiley & Sons, 2005.

[26] H. M. Sneed and A. Mérey, “Automated software quality assurance,” *IEEE Transactions on Software Engineering*, no. 9, pp. 909–916, 1985.

[27] D. L. Parnas and M. Lawford, “The role of inspection in software quality assurance,” *IEEE Transactions on Software engineering*, vol. 29, no. 8, pp. 674–676, 2003.

[28] D. S. Batory, D. Benavides, and A. R. Cortés, “Automated analysis of feature models: challenges ahead,” *Communications of the ACM*, vol. 49, no. 12, pp. 45–47, 2006.

[29] D. Benavides, S. Segura, and A. Ruiz-Cortés, “Automated analysis of feature models 20 years later: A literature review,” *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[30] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “Variability modeling in the systems software domain,” Tech. Rep., 2012, gSDLAB-TR 2012-07-06, University of Waterloo; superseded by the IEEE TSE journal publication "A Study of Variability Models and Languages in the Systems Software Domain". [Online]. Available: http://gsd.uwaterloo.ca/sites/default/files/vm-2012-berger.pdf

[31] J. Lee, S. Kang, and D. Lee, “A survey on software product line testing,” in *SPLC*, 2012.

[32] P. A. d. M. S. Neto, I. do Carmo Machado, J. D. McGregor, E. S. De Almeida, and S. R. de Lemos Meira, “A systematic mapping study of software product lines testing,” *Information and Software Technology*, vol. 53, no. 5, pp. 407–423, 2011.

[33] K. Lauenroth, K. Pohl, and S. Toehning, “Model checking of domain artifacts in product line engineering,” in *ASE*, 2009.

[34] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, “Family-based deductive verification of software product lines,” in *GPCE*, 2012.

[35] C. Kästner, S. Apel, T. Thüm, and G. Saake, “Type checking annotation-based product lines,” *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 3, pp. 14:1–14:39, Jul. 2012.

[36] J. Guo, D. Yang, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, “Data-efficient performance learning for configurable systems,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1826–1867, 2018.

[37] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, “Detection of feature interactions using feature-aware verification,” in *ASE*, 2011.

[38] S. Apel, A. von Rhein, T. Thüm, and C. Kästner, "Feature-interaction detection based on feature-based specifications," *Computer Networks*, vol. 57, no. 12, pp. 2399–2409, 2013.

[39] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *ICSE*, 2010.

[40] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of C code," in *AOSD*, 2011.

[41] D. M. Le, H. Lee, K. C. Kang, and L. Keun, "Validating consistency between a feature model and its implementation," in *ICSR*, 2013.

[42] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *ICSE*, 2014.

[43] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and Repairing Configuration Inconsistencies in Large Scale System Software," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 531–551, 2012.

[44] T. K. Satyananda, D. Lee, and S. Kang, "Formal verification of consistency between feature model and software architecture in software product line," in *ICSEA*, 2007.

[45] J. Greenyer, A. M. Sharifloo, M. Cordy, and P. Heymans, "Features meet scenarios: modeling and consistency-checking scenario-based product line specifications," *Requirements Engineering*, vol. 18, no. 2, pp. 175–198, 2013.

[46] M. Antkiewicz, W. Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, È. Stănciulescu, A. Wąsowski, and I. Schaefer, "Flexible product line engineering with a virtual platform," in *ICSE*. ACM, 2014.

[47] J. Krüger, T. Berger, and T. Leich, *Features and how to find them: a survey of manual feature location.* LLC/CRC Press, 2018.

[48] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations.* Springer Science & Business Media, 2012.

[49] T. K. Landauer, P. W. Foltz, and D. Laham, "An introduction to latent semantic analysis," *Discourse processes*, vol. 25, no. 2-3, pp. 259–284, 1998.

[50] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[51] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

[52] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron, "Florida: Feature location dashboard for extracting and visualizing feature traces," in *VaMoS*, 2017.

[53] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in forty preprocessor-based software product lines,” in *ICSE*, 2010.

[54] M. Robillard, R. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” *IEEE software*, vol. 27, no. 4, pp. 80–86, 2009.

[55] U. Pakdeetrakulwong, P. Wongthongtham, and W. V. Siricharoen, “Recommendation systems for software engineering: A survey from software development life cycle phase perspective,” in *The 9th International Conference for Internet Technology and Secured Transactions (ICITST-2014)*. IEEE, 2014, pp. 137–142.

[56] I. Katakis, G. Tsoumakas, and I. Vlahavas, “Multilabel text classification for automated tag suggestion,” in *Proceedings of the ECML/PKDD*, vol. 18, 2008, p. 5.

[57] S. Diplaris, G. Tsoumakas, P. A. Mitkas, and I. Vlahavas, “Protein classification with multiple algorithms,” in *Panhellenic Conference on Informatics*. Springer, 2005, pp. 448–456.

[58] M. R. Boutell, J. Luo, X. Shen, and C. M. Brown, “Learning multi-label scene classification,” *Pattern recognition*, vol. 37, no. 9, pp. 1757–1771, 2004.

[59] G. Tsoumakas and I. Katakis, “Multi-label classification: An overview,” *International Journal of Data Warehousing and Mining (IJDWM)*, vol. 3, no. 3, pp. 1–13, 2007.

[60] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS quarterly*, pp. 75–105, 2004.

[61] K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski, “Clafer: unifying class and feature modeling,” *Software & Systems Modeling*, vol. 15, no. 3, pp. 811–845, 2016.

[62] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[63] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[64] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “A study of variability models and languages in the systems software domain,” *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[65] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[66] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber, “Introducing pla at bosch gasoline systems: Experiences and practices,” in *SPLC*, 2004.

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

[68] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines," in *International Conference on Software Product Line*, ser. SPLC. ACM, 2015, pp. 16–25.

[69] A. Lozano, "An Overview of Techniques for Detecting Software Variability Concepts in Source Code," in *Advances in Conceptual Modeling. Recent Developments and New Directions.* Springer, 2011, pp. 141–150.

[70] W. K. G. Assunção and S. R. Vergilio, "Feature Location for Software Product Line Migration: A Mapping Study," in *International Software Product Line Conference*, ser. SPLC. ACM, 2014, pp. 52–59.

[71] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering.* Springer, 2013, pp. 29–58.

[72] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering Legacy Applications Into Software Product Lines: A Systematic Mapping," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2972–3016, 2017.

[73] A. Razzaq, A. Wasala, C. Exton, and J. Buckley, "The State of Empirical Evaluation in Static Feature Location," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 2:1–2:58, 2018.

[74] M. Antkiewicz, K. Bak, A. Murashkin, R. Olaechea, J. Hui, and K. Czarnecki, "Clafer tools for product line engineering." in *SPLC Workshops*, 2013, pp. 130–135.

[75] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with C news," in *USENIX*, 1992.

[76] G. Krone, M.; Snelting, "On the inference of configuration structures from source code," in *ICSE*, 1994.

[77] J.-M. Favre, "Preprocessors from an abstract point of view," in *ICSM*, 1996.

[78] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997.

[79] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-oriented software product lines: Concepts and Implementation.* Springer, 2013.

[80] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki, "The shape of feature code: an analysis of twenty c-preprocessor-based systems," *Software & Systems Modeling*, vol. 16, no. 1, pp. 77–96, 2017.

[81] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., "N degrees of separation: multi-dimensional Separation of Concerns," in *ICSE*, 1999.

[82] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information hiding interfaces for aspect-oriented design," in *ESEC/FSE*, 2005.

[83] S. Apel, T. Leich, and G. Saake, "Aspectual feature modules," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 162–180, 2008.

[84] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *SPLC*, 2011.

[85] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of variability models and related artifacts: a fresh look at evolution patterns in the Linux kernel," *Empirical Software Engineering*, 2015.

[86] L. Passos, K. Czarnecki, S. Apel, A. Wąsowski, C. Kästner, and J. Guo, "Feature-oriented software evolution," in *VaMoS*, 2013.

[87] E. Figueiredo, B. C. da Silva, C. Sant'Anna, A. F. Garcia, J. Whittle, and D. J. Nunes, "Crosscutting Patterns and Design Stability: An Exploratory Analysis," in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2009, pp. 138–147.

[88] T. Thum, D. Batory, and C. Kastner, "Reasoning about edits to feature models," in *ICSE*, 2009.

[89] B. Fluri, M. Wursch, and H. C. Gall, "Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes," in *Working Conference on Reverse Engineering*, ser. WCRE. IEEE, 2007, pp. 70–79.

[90] S. Nielebock, D. Krolikowski, J. Krüger, T. Leich, and F. Ortmeier, "Commenting Source Code: Is it Worth it for Small Programming Tasks?" *Empirical Software Engineering*, pp. 1–40, 2018.

[91] N. SpolaôR, E. A. Cherman, M. C. Monard, and H. D. Lee, "A comparison of multi-label feature selection methods using the problem transformation approach," *Electronic Notes in Theoretical Computer Science*, vol. 292, pp. 135–151, 2013.

[92] H. Liu and H. Motoda, *Computational methods of feature selection.* CRC Press, 2007.

[93] N. Spolaôr, M. C. Monard, G. Tsoumakas, and H. D. Lee, "A systematic review of multi-label feature selection and a new method based on label construction," *Neurocomputing*, vol. 180, pp. 3–15, 2016.

[94] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln, *Experimentation in Software Engineering.* Springer, 2012.

[95] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.

[96] F. Sattler, A. von Rhein, T. Berger, N. S. Johansson, M. M. Hardø, and S. Apel, "Lifting inter-app data-flow analysis to large app sets," *Automated Software Engineering*, no. 25, pp. 315–346, jun 2018.

[97] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 280–291.

[98] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, "Android taint flow analysis for app sets," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014, pp. 1–6.

[99] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness ocl constraints," in *Proceedings of the 5th international conference on Generative programming and component engineering*, 2006, pp. 211–220.

[100] E. Walkingshaw, C. Kästner, M. Erwig, S. Apel, and E. Bodden, "Variational data structures: Exploring tradeoffs in computing with variability," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, 2014, pp. 213–226.

[101] T. Berger, R.-H. Pfeiffer, R. Tartler, S. Dienst, K. Czarnecki, A. Wasowski, and S. She, "Variability mechanisms in software ecosystems," *Information and Software Technology*, vol. 56, no. 11, pp. 1520–1535, 2014.

[102] J. Bosch, "From software product lines to software ecosystems," in *SPLC*, 2009.

[103] C. W. Krueger, "New methods in software product line practice," *Communications of the ACM*, vol. 49, no. 12, pp. 37–40, Dec. 2006.

[104] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *SPLC*, 2015.

[105] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-21, 1990.

[106] M. B. Cohen, M. B. Dwyer, and J. Shi, "Interaction testing of highly-configurable systems in the presence of constraints," in *ISSTA*, 2007.

[107] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon, "Automated and scalable t-wise test case generation strategies for software product lines," in *ICST*, 2010.

[108] J. Midtgaard, C. Brabrand, and A. Wasowski, "Systematic derivation of static analyses for software product lines," in *MODULARITY*, 2014.

[109] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura, "A survey on the automated analyses of feature models," in *JISBD*, 2006.

[110] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: A requirements engineering perspective," in *FASE*, 2008.

[111] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a better understanding of software features and their characteristics: A case study of marlin," in *VaMoS*, 2018.

[112] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wasowski, "Three cases of feature-based variability modeling in industry," in *MODELS*, 2014.

[113] K. Schmid, R. Rabiser, and P. Grünbacher, "A comparison of decision modeling approaches in product lines," in *VaMoS*, 2011.

[114] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool features and tough decisions: A comparison of variability modeling approaches," in *VAMOS*, 2012.

[115] R. Bashroush, M. Garba, R. Rabiser, I. Groher, and G. Botterweck, "Case tool support for variability management in software product lines," *ACM Computing Surveys*, vol. 50, no. 1, pp. 14:1–14:45, Mar. 2017.

[116] A. Reuys, S. Reis, E. Kamsties, and K. Pohl, "The scented method for testing software product lines," in *Software Product Lines*. Springer, 2006, pp. 479–520.

[117] S. Reis, A. Metzger, and K. Pohl, "A reuse technique for performance testing of software product lines," in *SPLiT*, 2006.

[118] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry, "Test them all, is it worth it? a ground truth comparison of configuration sampling strategies," *arXiv preprint arXiv:1710.07980*, 2017.

[119] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *ICSE*, 2016.

[120] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dorre, and C. Lengauer, "Large-scale variability-aware type checking and dataflow analysis," Tech. Rep. MIP-1212, 2012.

[121] R. Queiroz, T. Berger, and K. Czarnecki, "Towards predicting feature defects in software product lines," in *FOSD*, 2016.

[122] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *ICSE*, 2010.

[123] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[124] P. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model composition in product lines and feature interaction detection using critical pair analysis," in *MODELS*, 2007.

[125] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer, "Is there a mismatch between real-world feature models and product-line research?" in *ESEC/FSE*, 2017.

[126] T. Berger and J. Guo, "Towards system analysis with variability model metrics," in *VAMOS*, 2014.

[127] E. Bagheri and D. Gasevic, "Assessing the maintainability of software product line feature models using structural metrics," *Software Quality Journal*, vol. 19, no. 3, pp. 579–612, 2011.

[128] A. R. Santos, R. P. de Oliveira, and E. S. de Almeida, "Strategies for consistency checking on software product lines: A mapping study," in *EASE*, 2015.

[129] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: Static analyses and empirical results," in *ICSE*, 2014.

[130] ——, "Where do configuration constraints stem from? an extraction approach and an empirical study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 820–841, 2015.

[131] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *EuroSys*, 2011.

[132] R. Tartler, J. Sincero, C. Dietrich, W. Schröder-Preikschat, and D. Lohmann, "Revealing and repairing configuration inconsistencies in large-scale system software," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 5, pp. 531–551, 2012.

[133] M. Vierhauser, P. Grünbacher, W. Heider, G. Holl, and D. Lettner, "Applying a consistency checking framework for heterogeneous models and artifacts in industrial product lines," in *MODELS*, 2012.

[134] C. Kröher, S. El-Sharkawy, and K. Schmid, "Kernelhaven: An experimentation workbench for analyzing software product lines," in *ICSE*, 2018.

[135] B. Zhang, M. Becker, T. Patzke, K. Sierszecki, and J. E. Savolainen, "Variability evolution and erosion in industrial product lines: A case study," in *SPLC*, 2013.

[136] J. Liebig, C. Kästner, and S. Apel, "Analyzing the discipline of preprocessor annotations in 30 million lines of c code," in *AOSD*, 2011.

[137] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, and G. Saake, "An overview on analysis tools for software product lines," in *SPLat*, 2014.

[138] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *ESE/FSE*, 2013.

[139] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *OOPSLA*, 2011.

[140] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel, "Product line analysis: a practical introduction," Tech. Rep. CMU/SEI-2001-TR-001, 2001.

[141] A. Strauss and J. Corbin, "Open Coding," *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*, vol. 2, pp. 101–121, 1990.

[142] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[143] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *CSMR*, 2013.

[144] L. Passos, J. Guo, L. Teixeira, K. Czarnecki, A. Wąsowski, and P. Borba, "Coevolution of variability models and related artifacts: A case study from the linux kernel," in *SPLC*, 2013.

[145] L. Linsbauer, T. Berger, and P. GrÃijnbacher, "A classification of variation control systems," in *GPCE*, 2017.

[146] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do You Remember This Source Code?" in *International Conference on Software Engineering*, ser. ICSE. ACM, 2018, pp. 764–775.

[147] C. Gacek and M. Anastasopoules, "Implementing Product Line Variabilities," *SIGSOFT Software Engineering Notes*, vol. 26, no. 3, pp. 109–117, 2001.

[148] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, and T. Berger, "Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin," in *International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS. ACM, 2018, pp. 105–112.

[149] A. Classen, P. Heymans, and P.-y. Schobbens, "What's in a Feature: A Requirements Engineering Perspective," in *Fundamental Approaches to Software Engineering*, ser. FASE. Springer, 2008, pp. 16–30.

[150] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The Love/Hate Relationship with the C Preprocessor: An Interview Study," in *European Conference on Object-Oriented Programming*, ser. ECOOP. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 495–518.

[151] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wąsowski, "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches," in *International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS. ACM, 2012, pp. 173–182.

[152] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, “A Study of Variability Models and Languages in the Systems Software Domain,” *IEEE Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[153] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, “An analysis of the variability in 40 preprocessor-based software product lines,” in *ICSE*, 2010.

[154] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, “Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers,” in *International Conference on Modularity*, ser. MODULARITY. ACM, 2015, pp. 81–92.

[155] M. Lillack, S. Stanciulescu, W. Hedman, T. Berger, and A. Wasowski, “Intention-Based Integration of Software Variants,” in *International Conference on Software Engineering*, ser. ICSE, 2019.

[156] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, “An Exploratory Study of Cloning in Industrial Software Product Lines,” in *European Conference on Software Maintenance and Reengineering*, ser. CSMR. IEEE, 2013, pp. 25–34.

[157] Ş. Stănciulescu, S. Schulze, and A. Wąsowski, “Forked and Integrated Variants in an Open-Source Firmware Project,” in *International Conference on Software Maintenance and Evolution*, ser. ICSME. IEEE, 2015, pp. 151–160.

[158] J. Krüger, L. Nell, W. Fenske, G. Saake, and T. Leich, “Finding Lost Features in Cloned Systems,” in *International Systems and Software Product Line Conference*, ser. SPLC. ACM, 2017, pp. 65–72.

[159] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, “Do Crosscutting Concerns Cause Defects?” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[160] M. P. Robillard and G. C. Murphy, “FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code,” in *International Conference on Software Engineering*, ser. ICSE. IEEE, 2003, pp. 822–823.

[161] S. Krieter, J. Krüger, and T. Leich, “Don’t Worry About It: Managing Variability On-The-Fly,” in *International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS. ACM, 2018, pp. 19–26.

[162] H. Abukwaik, A. Burger, B. Andam, and T. Berger, “Semi-Automated Feature Traceability with Embedded Annotations,” in *International Conference on Software Maintenance and Evolution*, ser. ICSME. IEEE, 2018, pp. 529–533.

[163] M. Rosenmüller, “Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines,” Ph.D. dissertation, University of Magdeburg, 2011.

[164] J. Lee and D. Muthig, “Feature-Oriented Variability Management in Product Line Engineering,” *Communications of the ACM*, vol. 49, no. 12, pp. 55–59, 2006.

[165] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchey, “An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry,” *Journal of Systems and Software*, vol. 91, pp. 3–23, 2014.

[166] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects,” in *European Conference on Object-Oriented Programming*, ser. ECOOP. Springer, 1997, pp. 419–443.

[167] B. Ray and M. Kim, “A Case Study of Cross-System Porting in Forked Projects,” in *International Symposium on the Foundations of Software Engineering*, ser. FSE. ACM, 2012, pp. 53:1–53:11.

[168] “Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines,” Electronic Industries Association, Standard, 1979.

[169] M. Antkiewicz, K. Bąk, A. Murashkin, R. Olaechea, J. H. J. Liang, and K. Czarnecki, “Clafer Tools for Product Line Engineering,” in *International Software Product Line Conference*, ser. SPLC. ACM, 2013, pp. 130–135.

[170] J. Martinez, W. K. G. Assunção, and T. Ziadi, “ESPLA: A Catalog of Extractive SPL Adoption Case Studies,” in *International Systems and Software Product Line Conference*, ser. SPLC. ACM, 2017, pp. 38–41.

[171] C. W. Krueger, “Easing the Transition to Software Mass Customization,” in *International Workshop on Software Product-Family Engineering*, ser. PFE. Springer, 2002, pp. 282–293.

[172] J. Wang, X. Peng, Z. Xing, and W. Zhao, “An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions,” in *International Conference on Software Maintenance*, ser. ICSM. IEEE, 2011, pp. 213–222.

[173] K. Damevski, D. Shepherd, and L. Pollock, “A Field Study of How Developers Locate Features in Source Code,” *Empirical Software Engineering*, vol. 21, no. 2, pp. 724–747, 2016.

[174] N. Wilde, M. Buckellew, H. Page, V. Rajilich, and L. T. Pounds, “A Comparison of Methods for Locating Features in Legacy Software,” *Journal of Systems and Software*, vol. 65, no. 2, pp. 105–114, 2003.

[175] H. Jordan, J. Rosik, S. Herold, G. Botterweck, and J. Buckley, “Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads,” in *International Conference on Program Comprehension*, ser. ICPC. IEEE, 2015, pp. 174–177.

[176] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[177] S. Apel and C. Kästner, "An overview of feature-oriented software development." *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[178] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski, "Three cases of feature-based variability modeling in industry," in *International Conference on Model Driven Engineering Languages and Systems.* Springer, 2014, pp. 302–319.

[179] S. Winkler and J. von Pilgrim, "A survey of traceability in requirements engineering and model-driven development," *Software & Systems Modeling*, vol. 9, no. 4, pp. 529–565, 2010.

[180] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski, "A Survey of Variability Modeling in Industrial Practice," in *International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS. ACM, 2013, pp. 1–8.

[181] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger, "Clone-based variability management in the android ecosystem," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2018, pp. 625–634.

[182] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla, "A study of feature scattering in the linux kernel," *IEEE Transactions on Software Engineering*, 2018.

[183] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[184] A. Burger and S. Gruner, "Finalist 2: Feature identification, localization, and tracing tool," in *SANER*, 2018.

[185] S. Zhou, S. Stanciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 105–116. [Online]. Available: https://doi.org/10.1145/3180155.3180205

[186] M. Seiler and B. Paech, "Documenting and exploiting software feature knowledge through tags," in *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019.*, 2019, pp. 754–777. [Online]. Available: https://doi.org/10.18293/SEKE2019-109

[187] F. W. Warr and M. P. Robillard, "Suade: Topology-based searches for software investigation," in *29th International Conference on Software Engineering (ICSE'07).* IEEE, 2007, pp. 780–783.

[188] J. Krüger, G. Calıklı, T. Berger, T. Leich, and G. Saake, "Effects of explicit feature traceability on program comprehension," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 338–349.

[189] S. Entekhabi, A. Solback, J.-P. Steghöfer, and T. Berger, "Visualization of feature locations with the tool featuredashboard," in *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B*, 2019, pp. 1–4.

[190] A. Pleuss, R. Rabiser, and G. Botterweck, "Visualization techniques for application in interactive product configuration," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, 2011, pp. 1–8.

[191] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code." in *SPLC (2)*, 2008, pp. 303–312.

[192] S. El-Sharkawy, N. Yamagishi-Eichler, and K. Schmid, "Metrics for analyzing variability and its implementation in software product lines: A systematic literature review," *Information and Software Technology*, vol. 106, pp. 1–30, 2019.

[193] P. Bille, "A survey on tree edit distance and related problems," *Theoretical computer science*, vol. 337, no. 1-3, pp. 217–239, 2005.

[194] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 432–441.

[195] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is my feature and what is it about? a case study on recovering feature facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019.

[196] G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas, "Mulan: A java library for multi-label learning," *Journal of Machine Learning Research*, vol. 12, pp. 2411–2414, 2011.

[197] F. Charte, A. Rivera, M. J. del Jesus, and F. Herrera, "Concurrence among imbalanced labels and its influence on multilabel resampling algorithms," in *International Conference on Hybrid Artificial Intelligence Systems*. Springer, 2014, pp. 110–121.

[198] F. Charte, A. J. Rivera, M. J. del Jesus, and F. Herrera, "Mlsmote: Approaching imbalanced multilabel learning through synthetic instance generation," *Knowledge-Based Systems*, vol. 89, pp. 385–397, 2015.

[199] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki, and M. T. Valente, "Feature scattering in the large: a longitudinal study of Linux kernel device drivers," in *MODULARITY*, 2015.

[200] P. Rothbauer, "Triangulation," *The SAGE encyclopedia of qualitative research methods*, vol. 1, pp. 892–894, 2008.

[201] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "A study of variability models and languages in the systems software domain," *Transactions on Software Engineering*, vol. 39, no. 12, pp. 1611–1640, 2013.

[202] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *ASE*, 2010.

[203] Kbuild, "The kernel build infrastructure," www.kernel.org/doc/Documentation/kbuild, last seen: Feb. 14th, 2015.

[204] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "The variability model of the Linux kernel," in *VaMoS*, 2010.

[205] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd. and Toshiba Corp., "Advanced configuration and power interface specification, revision 5.0," http://www.acpi.info/spec50a.htm, last seen: Feb. 14th, 2015.

[206] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the Linux Kernel a software product line?" in *OSSPL*, 2007.

[207] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wasowski, "Evolution of the Linux kernel variability model," in *SPLC*, 2010.

[208] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann, "Understanding Linux feature distribution," in *Proceedings of the 2nd Workshop on Modularity in Systems Software.* ACM, 2012, pp. 15–20.

[209] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux device drivers*, 3rd ed. O'Reilly, 2005.

[210] S. Venkateswaran, *Essential Linux device drivers*, 1st ed. Prentice Hall Press, 2008.

[211] M. W. Godfrey and Q. Tu, "Evolution in open source software: a case study," in *ICSM*, 2000.

[212] C. Izurieta and J. Bieman, "The evolution of FreeBSD and Linux," in *ESEM*, 2006.

[213] D. G. Feitelson, "Perpetual development: a model of the Linux kernel life cycle," *Journal of Systems and Software*, vol. 85, no. 4, pp. 859–875, 2012.

[214] J. Corbet, G. Kroah-Hartman, and A. McPherson, "Linux kernel development: how fast it is going, who is doing it, what they are doing, and who is sponsoring it," `http://www.linuxfoundation.org/publications/linux-foundation/who-writes-linux-2013`, last seen: Feb. 14, 2015.

[215] M. Eaddy, A. Aho, and G. C. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *ACoM*, 2007.

[216] M. Kasunic, *Designing an effective survey. Technical report, handbook CMU/SEI-2005-HB-004*. Software Engineering Institute, Carnegie Mellon University, 2005.

[217] M. Hubert and E. Vandervieren, "An adjusted boxplot for skewed distributions," *Computational Statistics & Data Analysis*, vol. 52, no. 12, pp. 5186–5201, 2008.

[218] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz, "Comparative analysis of evolving software systems using the gini coefficient," in *ICSM*, 2013.

[219] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski, "Feature-to-code mapping in two large product lines," in *SPLC*, 2010.

[220] M. T. Jones, "Anatomy of the Linux kernel," *IBM Developer Works*, 2009.

[221] D. S. Moore, G. P. McCabe, and B. Craig, *Introduction to the practice of statistics*, 6th ed. W. H. Freeman, 2009.

[222] S. Nadi and R. Holt, "The Linux kernel: a case study of build system variability," *Journal of Software: Evolution and Process*, vol. 26, no. 8, pp. 730–746, 2014.

[223] T. Chaikalis, A. Chatzigeorgiou, and G. Examiliotou, "Investigating the effect of evolution and refactorings on feature scattering," *Software Quality Journal*, pp. 1–27, 2013.

[224] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*, 1st ed. Springer, 1997.

[225] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: a case study using conditional compilation," in *CSMR*. IEEE, 2011, pp. 191–200.

[226] S. Paul, A. Prakash, E. Buss, and J. Henshaw, "Theories and techniques of program understanding," in *CASCON*. IBM Press, pp. 37–53.

[227] M. P. Robillard and G. C. Murphy, "Feat: a tool for locating, describing, and analyzing concerns in source code," in *ICSE, Demonstrations Track*, 2003.

[228] P. Oliveira, M. T. Valente, and F. P. Lima, "Extracting relative thresholds for source code metrics," in *CSMR-WCRE*, 2014.

[229] M. Voelter, J. Warmer, and B. Kolb, "Projecting a modular future," *IEEE Software*, vol. 32, no. 5, pp. 46–52, 2015.

[230] B. Behringer, J. Palz, and T. Berger, "Peopl: Projectional editing of product lines," in *ICSE*, 2017.

[231] M. Mukelabai, B. Behringer, M. Fey, J. Palz, J. Krüger, and T. Berger, "Multi-view editing of software product lines with peopl," in *ICSE, Demonstrations Track*, 2018.