Towards a Better Understanding of Variability Evolution

by

Leonardo Passos

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor in Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

Abstract

Highly-configurable software systems often leverage variability modeling to achieve systematical reuse and mass customization. Although facilitating variability management, variability models do not eliminate the variability in other artifacts. In fact, evolving a system's variability is far from trivial, as variation points spread across different artifacts, possibly at multiple locations—evolving a single feature may affect many variation points. To make matters worse, existing approaches for variability evolution have been largely criticized in practice, as industry-based reports claim them as ineffective.

Ineffective support appears to be a direct consequence of lacking an in-depth understanding of how variability evolution happens in practice. For instance, most of the existing research focuses on variability evolution as it happens in variability models only, ignoring how its evolution relate to other artifacts (e.g., build and implementation files). Moreover, when validating new variability evolution approaches, researchers often rely on randomly generated models, or in some situations, even on fictitious cases. Studies that do account for variability evolution across different artifacts do so in the context of small systems, which are unlikely to be representative of the complexity typically found in large-scale subjects.

Understanding variability evolution is a pre-requisite for properly supporting it in practice. As the former is yet immature, we seek to advance the existing understanding by performing an in-depth analysis of variability evolution in large, complex, and real-world systems in the systems software domain.

As a starting point, we perform an exploratory analysis over a sample of the Linux kernel evolution, one of the largest and longest-living configurable system. Motivated by the impact of pattern analysis in modern software engineering (e.g., refactoring patterns), we set to mine evolution patterns from the Linux kernel commit history. Specifically, our patterns focus on the variability evolution induced by adding or removing features in the variability model, capturing how other artifacts (e.g., Makefiles and code) coevolve as a consequence. We identify 23 variability-coevolution patterns, from which we crosscheck their properties with the current literature, evidencing limitations in existing approaches, as well as providing insights for improving existing tools and helping to shape future ones. Additionally, we also observe how developers implement new features, finding feature scattering as a recurrent practice. This is particularly interesting, as feature scattering is often criticized in practice. We argue that scattering is not necessarily bad if used with care—in fact, as with the Linux kernel case, existing systems have shown that it is possible to achieve long-term evolution while accepting some level of feature scattering. The limits of

feature scattering, however, are currently unknown. This is not surprising, as no empirical study investigates feature scattering across the evolution of large and long-lived software systems.

From our exploratory analysis of the Linux kernel, we perform further assessments to strengthen our understanding.

First, we set to increase the external validity of our patterns by validating them in the context of three other systems: axTLS, Toybox, and uClibc. We find that our patterns cover as much as 64 % of all feature additions and removal cases across the evolution of our three chosen subjects—altogether, our validation spans a period of over 20 years of evolution. Moreover, we find 14 patterns whose use goes beyond Linux. In fact, we claim them as general cases within the systems software domain.

Second, seeking a better understanding of feature scattering limits, we return our attention to the Linux kernel evolution. Different from the mining of patterns, our analysis considers the entire snapshot of the Linux kernel commit history, covering almost eight years of evolution. Scoped to the scattering of device-driver features, the most common feature type in the Linux kernel, we set to identify empirical limits within the codebase, including the proportion of scattered features, as well as identifying typical scattering degrees. We also note specific feature types which appear to be more prone to scattering. While we do not claim the limits we find as universal, our study provides evidence that scattering can go as far as the limits we observe in the Linux kernel implementation.

Acknowledgements

This thesis would not have been possible without my family support. I thank Itamar das Chagas Passos, my dad, for his great effort in providing me, my brothers Luis and Milton, and my sister Thaisa with great educational opportunities, a reality that back home is still an exception rather than a rule. Special thanks also goes to Milsy Teixeira Passos, my dear mom, whose care and love have always been present. I would also like to thank Aline Ferreira, who, back in 2010, supported the early planning of my Ph.D.

I am also grateful to my advisor, Professor Krzysztof Czarnecki, who has given me all the freedom to perform my work, yet, constantly guiding me through the revolt waters of research. Professor, you are truly an inspiration, and I sincerely thank you for all your support. I also thank Professor Andrzej Wąsowski and Professor Sven Apel for their constant feedback and appealing discussions. Nicolas Dintzner and Leopoldo Teixeira shall not be forgotten; without your contributions, this work would have never happened.

I thank all my lab mates from the Generative Programming Laboratory at the University of Waterloo. You have made my journey very joyful; being at the lab was never an obligation, but rather a personal need. Special appraisal goes to Thorsten Berger, Jianmei Guo, Ed Zulkoski, Pavel Valov, and Atri Sarkar. I have a special bond with you all, and hope that our friendship shall last for the many years to come, regardless of potential geographical distances. I also thank Professor Vijay Ganesh, a great friend that I was very lucky to meet, although at a late stage in my Ph.D.—rumour has it, good wine is always served last.

Last, but not least, I thank Haldun Sural, Ricardo Console, and the entire Majeed family for their warm welcoming when I first came to Canada.

Table of Contents

List of Tables **x**

List of Figures **xii**

1 Introduction **1**

1.1 Main Research Question 3

1.2 Motivating Example 3

1.3 Research Outline 8

1.3.1 Step One: Linux Kernel as an Exploratory Case Study 10

1.3.2 Step Two: Further Assessments 11

1.3.3 Research Contributions 12

1.4 Thesis Outline 14

1.5 Conventions 14

1.6 Publications 15

1.7 Awards 15

2 Variability Evolution **18**

2.1 Variability Model Evolution 18

2.2 Mapping Evolution 20

2.3 Implementation Evolution 21

2.3.1 Evolution in Annotative-based Approaches 21

2.3.2 Evolution in Modular-based Approaches 23
2.3.3 Others Approaches 25
2.4 Variability Coevolution 25
2.5 Conclusion 27

3 Linux as an Exploratory Case Study 28

3.1 The Three Spaces of the Linux Kernel 28
3.1.1 Variability Model 29
3.1.2 Mapping 30
3.1.3 Implementation 31
3.2 Evolution Overview 33
3.3 Research Question 33
3.4 Methodology 34
3.4.1 Scoping 35
3.4.2 Data Collection 35
3.4.3 Mining Process 39
3.4.4 Review Process 44
3.4.5 Pattern Inference 46
3.4.6 Pattern Representation 47
3.5 Results 49
3.5.1 Non-Inferred Feature Addition Patterns 51
3.5.2 Inferred Feature Addition Patterns 66
3.5.3 Non-Inferred Feature Removal Patterns 71
3.5.4 Inferred Feature Removal Patterns 75
3.5.5 Non-Patterns in the Additions and Removals Samples 75
3.6 Discussion 76
3.6.1 Variability Evolution Support Requires Accounting for the Coevolution of Variability Models with Related Artifacts 76

3.6.2 New Theories Must Account for Feature Retirement 78

3.6.3 High Modularity, Low Scattering Degree, Localized Scattering, and Disciplined Annotations are Key Evolution Principles 78

3.7 Threats to Validity 80

3.8 Conclusion 82

4 Variability-Coevolution Patterns: Beyond the Linux Kernel 83

4.1 Research Questions 84

4.2 Methodology 85

4.2.1 Subject Selection 85

4.2.2 axTLS Conversion to Git 90

4.2.3 Scoping 90

4.2.4 Data Collection 94

4.2.5 Pattern Identification Process 98

4.2.6 Manual Identification of Patterns 102

4.2.7 Review Process 103

4.2.8 Generality Inference 107

4.3 Results 108

4.3.1 Catalog Expressiveness 108

4.3.2 Pattern Generality 111

4.4 Discussion 111

4.4.1 Different Systems May Realize Patterns with Different Mechanisms 112

4.4.2 Feature Modularity Directly Relates to a Project's Notion of Feature 115

4.4.3 There Appears to Exist a Balance Between the Number of AVOMF Instances Adding Extensions and the Scattering Degree Values of such Features 117

4.4.4 Most AVONMF Instances Relate to Scattered Features, but with Low SD. 118

4.4.5 Annotations Tend to be Disciplined 119

4.5 Threats to Validity 121

4.6 Conclusion 122

5 **Towards Better Understanding Feature Scattering: A Longitudinal Analysis of Linux Kernel Device-Driver Features** **123**

5.1 Research Questions 124

5.2 Methodology 125

5.2.1 Scoping 125

5.2.2 Identifying Driver Features 125

5.2.3 Data Collection 126

5.2.4 Data Analysis 127

5.3 Results 127

5.3.1 Scattered versus Non-Scattered Features 128

5.3.2 Local versus Global Scattering 130

5.3.3 Scattering Degrees 132

5.3.4 Causes of Scattering 135

5.4 Threats to Validity 140

5.5 Conclusion 142

6 **Conclusion** **145**

6.1 Summary 145

6.2 Future Work (Research Agenda) 146

References **150**

List of Tables

1.1 Popular variability-aware research subjects 17

3.1 Distribution of source code file types (averages taken from the v2.6.12–v3.9 release range) 31

3.2 Commit statistics 37

3.3 Activities performed by each participant in each sample 47

3.4 Collected patterns and their frequency 50

3.5 Description of the top-level folders of the Linux kernel source code tree . . 52

3.6 Mapping of the kernel's top-level directories and its subsystems 53

3.7 Frequency of non-inferred patterns per subsystem (additions sample) . . . 54

3.8 Frequency of non-inferred patterns per subsystem (removals sample) 72

4.1 Target population of systems in the systems software domain 87

4.2 Distribution of source code file types (averages taken from the entire evolution of the target subjects) 88

4.3 Size of our three subjects of analysis 89

4.4 Mapping of axTLS login names to corresponding Git users 90

4.5 Stable release identification and sorting criteria 91

4.6 Releases within our scope of analysis 93

4.7 Release time ranges 94

4.8 Feature additions and removals population of each target subject 95

4.9 Commit statistics for our three chosen subjects 99

4.10 Number of excluded addition and removal cases 101

4.11 Correspondence between participants in the Linux kernel case study and the ones validating our catalog of patterns 102

4.12 Activities performed by each participant in the analyses of axTLS and Toybox (I: pattern identification; R(P): review of the patterns that P identifies) 105

4.13 Activities performed by each participant in the analyses of uClibc (I: pattern identification; R(P): review of the patterns that P identifies) 106

4.14 Commit window inconsistencies stemming from participants' reviews . . . 106

4.15 Pattern identification inconsistencies stemming from participants' reviews . 106

4.16 Matching cases among all additions and removals 109

4.17 Counting of pattern instances 112

4.18 AVOMF instances introducing extensions 117

4.19 (Non-)scattered cases among AVOMF instances introducing extensions . . 118

4.20 *SD*-values of AVOMF instances introducing extensions 118

4.21 Extensions among AVONMF instances 119

4.22 (Non-)scattered cases among AVONMF instances 119

4.23 *SD*-values of AVONMF instances 119

4.24 Syntactic level of extensions among AVOMF instances introducing *ifdefs* . 120

4.25 Syntactic level of AVONMF extensions 120

5.1 Summary statistics of (non-)scattered driver features (relative) 129

5.2 Summary statistics of (non-)scattered driver features (absolute) 129

5.3 Summary statistics of locally and globally scattered driver features (relative) 131

5.4 Summary statistics of locally and globally scattered driver features (absolute) 131

5.5 Relative and absolute growth of scattered outlier features 134

5.6 Summary statistics of the average *SD*-value of scattered driver features . . 135

5.7 Relationship between being a platform-driver feature and scattering location 139

5.8 Relationship between being an infrastructure-driver feature and scattering location 139

List of Figures

1.1 Variability model of the automotive example 4

1.2 Variability model and the related artifacts of the automotive example (snapshot t_0) 5

1.3 Merge of YRSModel1 into YRSModel2 (snapshot t_1) 6

1.4 Simplification of the software structure (snapshot t_2) 8

1.5 Addition of new ABS-related features (snapshot t_3) 9

3.1 The three spaces in the Linux kernel and their interaction with Kbuild . . 30

3.2 Kernel growth evolution 34

3.3 Database creation process 36

3.4 Commit window example 40

3.5 Patch adding the Davinci DM646x EVM driver (primary commit) 41

3.6 Commit window sizes 43

3.7 Commits changing the kernel memory-based chunk allocation 45

3.8 Patch adding NEED_PER_CPU_KM 46

3.9 Definition of Merge Visible Optional Feature into Sibling 49

3.10 Definition of Add Visible Optional Modular Feature 56

3.11 Example of Add Visible Optional Guard Modular Feature 57

3.12 Definition of Add Visible Optional Non-Modular Feature 58

3.13 Definition of Add Visible Optional Compilation Flag Feature 60

3.14 Example of Add Visible Optional Compilation Flag Feature 61

3.15 Definition of Add Visible Mandatory Value-Based Feature 61
3.16 Example of Add Visible Mandatory Value-Based Feature 62
3.17 Definition of Featurize Compilation Unit to Visible Optional Feature . . . 64
3.18 Example of Featurize Compilation Unit to Visible Optional Feature 65
3.19 Definition of Featurize Code Fragment to Visible Optional Feature 66
3.20 Example of Featurize Compilation Unit to Visible Optional Feature 67
3.21 Example of Add Visible Optional Abstract Feature 69
3.22 Example of an invisible optional abstract feature 70
3.23 Definition of Merge Visible Optional Feature into New One 73
3.24 Example of Merge Visible Optional Feature into New One 74

4.1 Handling the evolution of the Kconfig language 92
4.2 ToyBox: patch adding ls 96
4.3 ToyBox probing example 97
4.4 ToyBox: git query to locate the primary commit adding ls 99
4.5 Commit window sizes of axTLS, ToyBox, and uClibc 100
4.6 Pattern matching priorities (additions) 103
4.7 Pattern matching priorities (removals) 104
4.8 Amount of toy features against other kinds 107
4.9 Pattern matching results 110
4.10 Selection statement as a replacement for `#ifdef` 114
4.11 AVOGMF instance using a Makefile conditional (axTLS) 115
4.12 AVOGMF instance using a Makefile conditionals and a directory variable (uClibc) 115

5.1 Feature distribution per subsystem 126
5.2 Data extraction process 127
5.3 Growth of (non-)scattered driver features 128
5.4 Growth of locally and globally scattered driver features 130

5.5 *SD*-values of scattered driver features 133
5.6 Growth of outlier scattered features 134
5.7 Evolution of the Gini coefficient of the *SD*-value of scattered driver features 134
5.8 Scattering location of sampled (non-)platform drivers 138
5.9 Scattering degree levels in (non-)infrastructure driver features 141
5.10 Ranking of outlier features 142
5.11 Ranking of features among non-infrastructure outliers 143

6.1 Patch validation 149

Chapter 1

Introduction

Highly-configurable software systems (a.k.a *variant-rich*) offer a large degree of configurability, allowing users to tailor a target system according to their preferences and needs. The high degree of configurability arises from the variability of the artifacts of the system, meaning that they can be configured for use in a particular context [65]. Once configured, the target system varies its behaviour or structure accordingly, leading to a specific *variant*. Examples of such systems span different domains, including database management systems [14, 76, 113, 114], SOA-based applications [10], operating systems [7, 19, 21], and industrial software product lines.[1]

As large and complex variant-rich systems have considerable numbers of points of variabilities, these systems often describe them in terms of *features*, employing *variability models* to explicitly capture user-relevant features and their associated constraints; when doing so, these systems become *variability-aware*. Features, in this case, denote either functionality chunks (coarse-grained variability) or fine-grained configuration parameters. Features declared in the variability model may then be referenced in related software artifacts (e.g., Makefiles and C source code) by means of explicit *variation points*. Examples of variation points include `#ifdef` C pre-processor directives (a particular kind of code annotation [7]) and conditional build rules. Feature referencing, in turn, allows different artifacts to vary according to specific configurations (feature selections and their associated values).

As with other types of software, variability-aware systems must evolve to meet changing requirements, platforms, and other environmental pressures. In addition to the inherent complexity of evolving large systems, variability-aware systems with high number of features impose yet another challenge: the evolution of their underlying variability.

[1] `http://splc.net/fame.html`

Along with the complexity of their variability models, which may include thousands of features [18, 126], large variability-aware systems have hundreds of variation points spread across build artifacts, models, source code, and other artifacts.

So far, understanding variability evolution has not been thorough. Most of the existing knowledge is partial, as many researchers study variability evolution in variability models only [4, 44, 64, 71, 91, 109, 123, 133], ignoring its connection to the evolution of related artifacts (e.g., build files and code). Moreover, the validation of variability evolution techniques often relies on randomly-generated variability models or evolution scenarios that do not come from real-world systems [64, 71, 133], posing a reliability threat. The few existing studies aiming to understand variability evolution across different artifacts and in the context of real subjects rely on small systems, which are unlikely to reflect the complexity typically found in large systems. For instance, Neves et al. [104] study the coevolution of variability models and related artifacts in real software product lines (SPLs), but their subjects have less than 50 features. In addition, their analysis limits to refinement changes (i.e., changes that do not affect the behaviour of the system). Scoping investigation to refinement cases, however, is too restrictive in practice; as we discuss later in the text, feature modification and retirement are too frequent to be ignored.

The lack of a thorough understanding of variability evolution coincides with poor variability evolution support. As Chen et al. point out [29, 30], the few existing approaches claiming to support variability evolution are ineffective in practice. Further triangulation of data sources confirms such belief [9]:

> *"Variability evolves as a result of adding, deleting, or updating variation points and variants. However, we found little support for systematically and sufficiently supporting evolution in variability models and other related artifacts."*
>
> —Babar et al., IEEE Software, 2010.

From the stated sources of evidence, we assume a causal relationship between lacking a thorough variability evolution understanding and inadequate variability evolution support. To us, a thorough understanding is the same as understanding variability evolution as it happens in practice. Hence, we phrase the following assumption.

Main Assumption. Poor variability evolution support follows from a lack of understanding of how variability evolution happens in real-world settings.

Upon the validity of our assumption, adequate variability evolution support can only be achieved if one sets to understand variability evolution as it happens in practice. That leads to the research goal of this thesis.

Main Research Goal. Advance the understanding of variability evolution as it happens in real-world settings.

1.1 Main Research Question

To achieve our research goal, we state a broad research question, which we distill in later chapters.

Main Research Question: How does variability evolution occur in real-world settings?

1.2 Motivating Example

To illustrate variability evolution and what one can learn by studying it, we present an example inspired by the automotive industry, a domain known for having a complex and large variability [18, 136]. Our example also sets the basic terminology for the remainder of this text. The example system comprises a set of features controlling specific hardware devices in a car, namely braking, stability control, and a yaw sensor. Its structure is given in terms of a variability model, whose features are realized in implementation artifacts (C code in our example). A mapping (Makefiles in our case) then associates features to specific compilation units (.c files). Structuring a system in terms of a variability model, a mapping, and an implementation space is typical in variability-aware systems, and resembles the structure of many SPLs [7, 37].[2]

The variability model describes the common and variable characteristics of different instances of our car system (see Figure 1.1). Following the FODA-notation, a well-known

[2] A typical SPL is structured as follows: (i) *problem space*: captures all features in a target product portfolio; (ii) *solution space*: realizes all features in the target SPL; and (iii) *mapping*: maps features to their corresponding implementation.

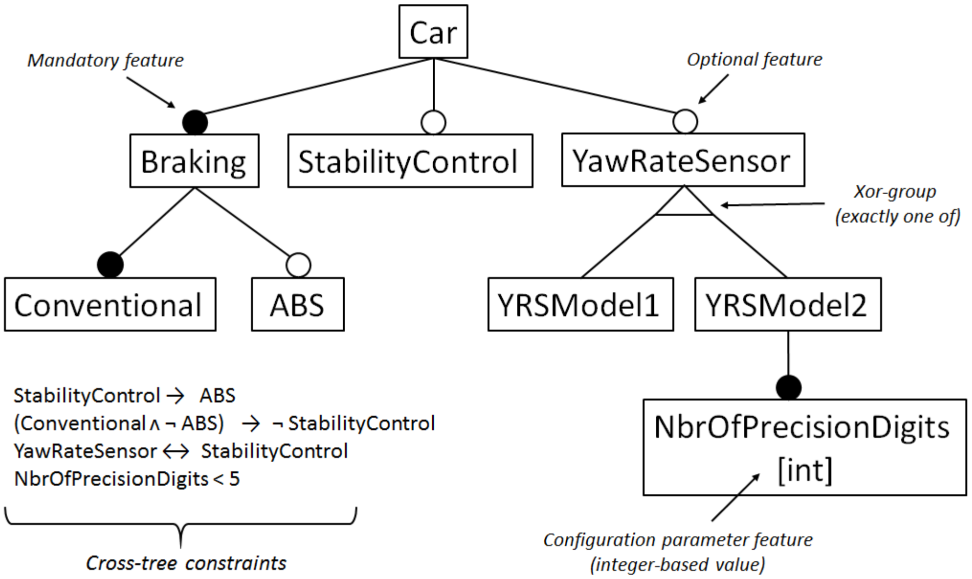


Figure 1.1: Variability model of the automotive example

representation for variability models [20], features (shown as rectangles) are put in a parent-child hierarchy, where a child feature always implies the presence of its parent (e.g., ABS implies the presence of Braking). Features can be made optional—denoted by an empty circle—or mandatory, denoted by a filled circle (e.g., every car has Braking, but may not have StabilityControl). In the example, both Conventional and ABS can be present at the same time, but Conventional is mandatory, whereas ABS is not. This is due to safety; if both features are present, and if ABS fails, the car automatically switches to conventional braking. Features can also be placed in a group. In the example, the sensors YRSModel1 and YRSModel2 are in an xor-group stating that exactly one of them can be present at any given time. Both sensors are based on the same technology, with the difference that YRSModel1 has yaw-rate prediction support, whereas the second model has better precision measure, allowing setting the number of decimal digits (NbrOfPrecisionDigits). In addition to the dependency constraints imposed by the hierarchy, cross-tree constraints define specific dependencies across the parent-child hierarchy, and may also define value restrictions involving multiple features [108]. For simplicity, we have illustrated a special case controlling the value of a single feature (NbrOfPrecisionDigits).

In addition to the variability model, the system contains Makefiles and C source code (see Figure 1.2); the latter realizes the system's features, whereas Makefiles map features in the variability model to their associated compilation units. To abstract over such rules, we

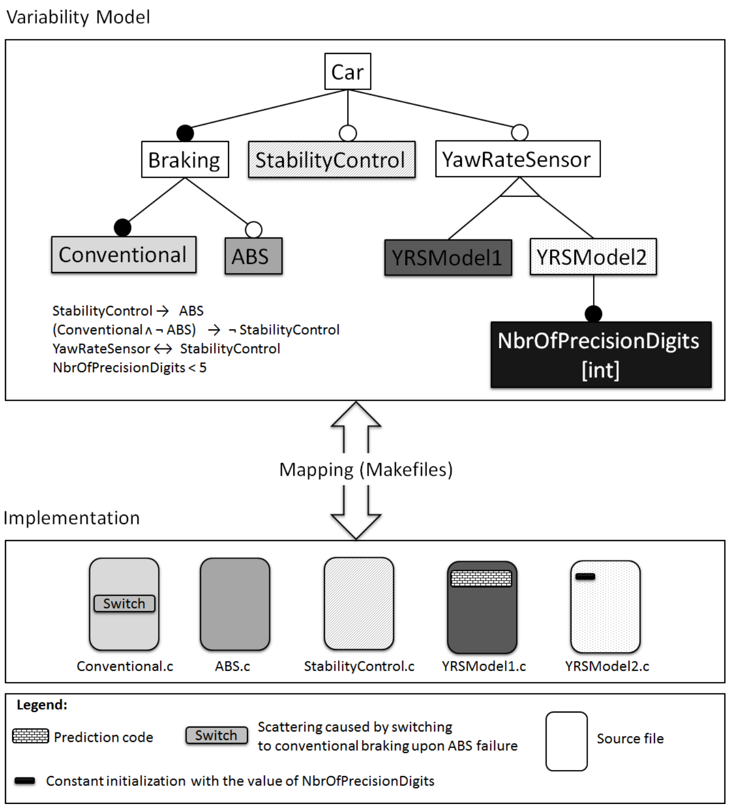


Figure 1.2: Variability model and the related artifacts of the automotive example (snapshot t_0)

show these mappings in terms of specific pattern fills (e.g., as feature Conventional binds to Conventional.c, both share the same background color—light gray). Note, however, that not all features are completely modularized: the implementation of Conventional has a logic that enables it as the main braking controller upon the failure of ABS. In that case, the presence of ABS creates explicit variation points in Conventional.c, causing ABS to be *scattered across* Conventional.

Evolution Scenario 1 Due to the cost of maintaining two similar, but independent yaw-sensors, stakeholders of our example system decide to merge YRSModel1 into the existing feature YRSModel2. As such, they copy the prediction code of the former into the latter, and remove YRSModel1 from the variability model. This change leads to a new snapshot of the system (t_1), shown in Figure 1.3. In order to eliminate the unnecessary

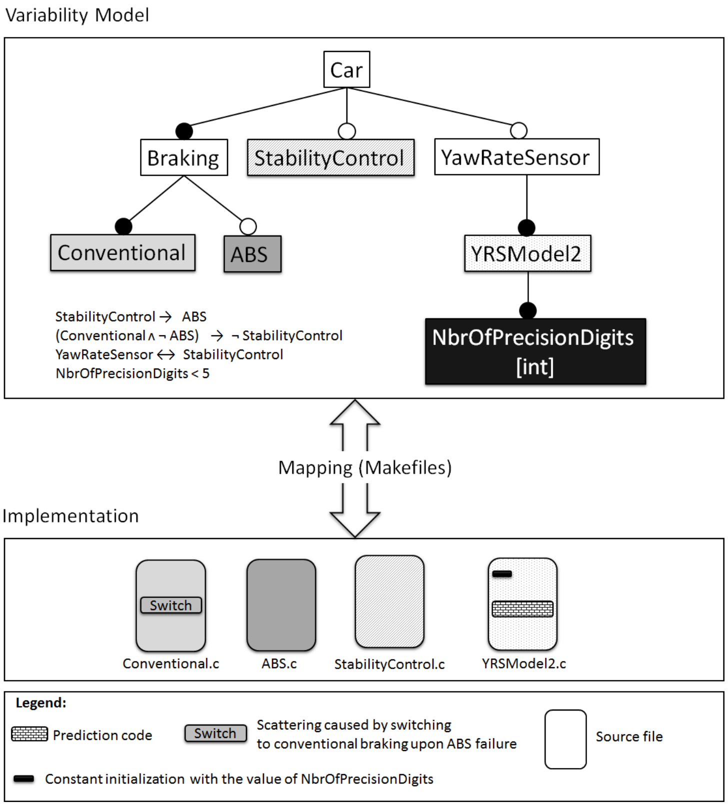


Figure 1.3: Merge of YRSModel1 into YRSModel2 (snapshot t_1)

chain of mandatory features that start from YawRateSensor, stakeholders lift YRSModel2 to YawRateSensor, and subsequently rename YRSModel2.c to YawRateSensor.c. Figure 1.4 shows the resulting snapshot. At each step, the mapping is updated accordingly.

The described evolution scenario highlights the risks of focusing exclusively on the evolution of the variability model: by comparing the original variability model at snapshot t_0 and the one resulting from the first change (snapshot t_1), one may conclude that the first sensor is no longer supported. Comparing t_0 and t_2 only by diffing their variability models is even more misleading, as one might conclude that the capabilities of YRSModel1 and YRSModel2 no longer exist in t_2. In contrast, observing the variability evolution across different artifacts leads to a different picture: since YRSModel2 subsumes YRSModel1 in supported capabilities, snapshots t_1 and t_2 have the same capabilities as t_0.

As this evolution scenario demonstrates, techniques that capture evolution as it occurs only at the variability model are unlikely to fully reflect the actual evolution of the system.

For instance, existing edit-reasoning techniques [71, 128, 133] do not account for changes in other artifacts outside the variability model. As such, they are unable to provide a correct result in the face of changes whose semantics depends on edits in different artifacts. The same can be said about the recovery and maintenance of traceability links.

As there are many possibilities of how variability may evolve across different artifacts, studying it in real-world settings helps to understand what evolution scenarios occur in practice (e.g., a merge), what causes them to occur (e.g., feature similarity), how changes are made (e.g., merging at the variability model level, and copying code at the implementation level) and how they impact existing techniques (e.g., edit-reasoning and tracing techniques). This evolution scenario, in fact, is motivated from our current findings when conducting case studies of real-world and complex variability-aware systems, which we report in Chapters 3 and 4.

Evolution Scenario 2 As our system continues to evolve, it incorporates new ABS controllers. In addition to the ABS controller from snapshots t_0–t_2, which supports a single speed sensor, stakeholders of our car system add three new controllers (see Figure 1.5): ABS2S (ABS with two speed sensors), ABS3S (ABS with three speed sensors), and ABS4S (ABS with 4 speed sensors). When doing so, stakeholders rename the ABS feature to ABS1S and add a new abstract feature ABSType to root an xor-group over all four controllers. The addition of new ABS types affects StabilityControl, as its skid control mechanism depends on the target ABS kind. Due to prohibitive costs, stakeholders of our car example do not fully modularize each ABS type; instead, the code of each ABS controller is scattered across StabilityControl. Compared to alternative solutions (e.g., design patterns, aspects, etc), scattering does not require knowledge of complex programming language constructs, in addition to having little upfront investment [7]. Scattering, however, has long been argued as an undesirable situation [52, 82, 84, 125]; the intermingling of scattered features with different implementation parts can lead to ripple effects, while requiring developers to be kept in constant sync, hindering parallel development. Aware of such trade-offs, stakeholders of our car system still choose the immediate benefits of feature scattering (e.g., low cost and simple programming language knowledge); by decreasing the immediate development time compared to adopting a modular solution, engineers release our example car system according to its planned schedule.

Our evolution scenario exemplifies that scattering is not necessarily bad, provided it is used with care. Currently, however, there are no empirical studies investigating the limits of feature scattering in the continuous evolution of large-scale variability-aware software systems. Chapter 5 improves the current state-of-affairs with a longitudinal feature scattering case study.

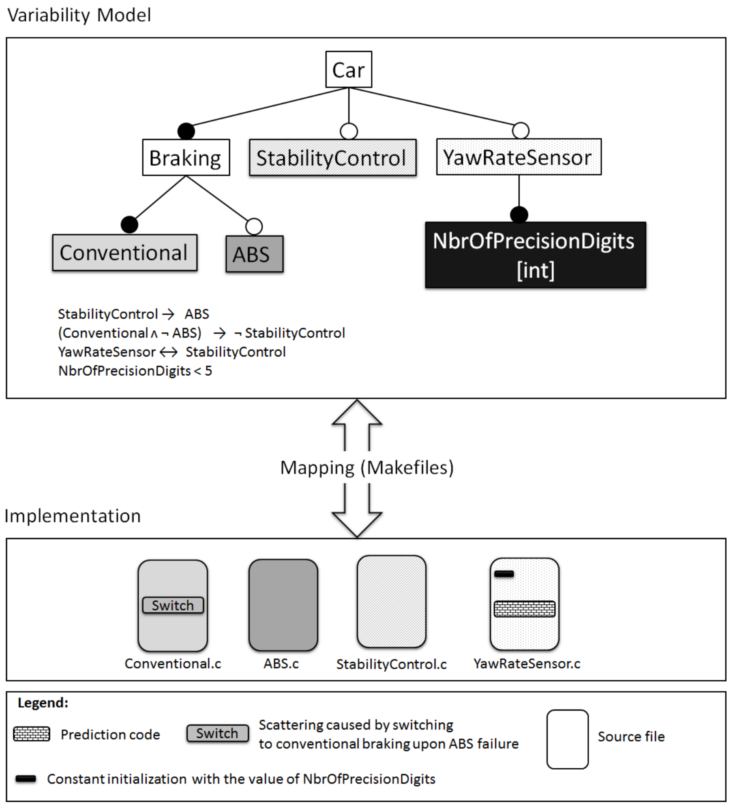


Figure 1.4: Simplification of the software structure (snapshot t_2)

1.3 Research Outline

To answer our main research question, we perform an exploratory case study of a real-world and complex variability-aware software system: the Linux kernel. From our exploratory analysis, we collect findings for further assessment, leading us to verify our understanding in three other systems and in Linux itself.

Three reasons justify choosing the Linux kernel as a starting point of investigation:

1. The Linux kernel is one of the largest and longest-living variability-aware system in existence today, comprising over 13,000 features and more than 10 million source lines of code (SLOC).[3] As Table 1.1 shows, such numbers are far beyond the complexity

[3] SLOC (Source Lines of Code) is given by LOC (Lines of Code) minus the number of empty lines. LOC, in turn, is given by the total number of lines, including blank ones.

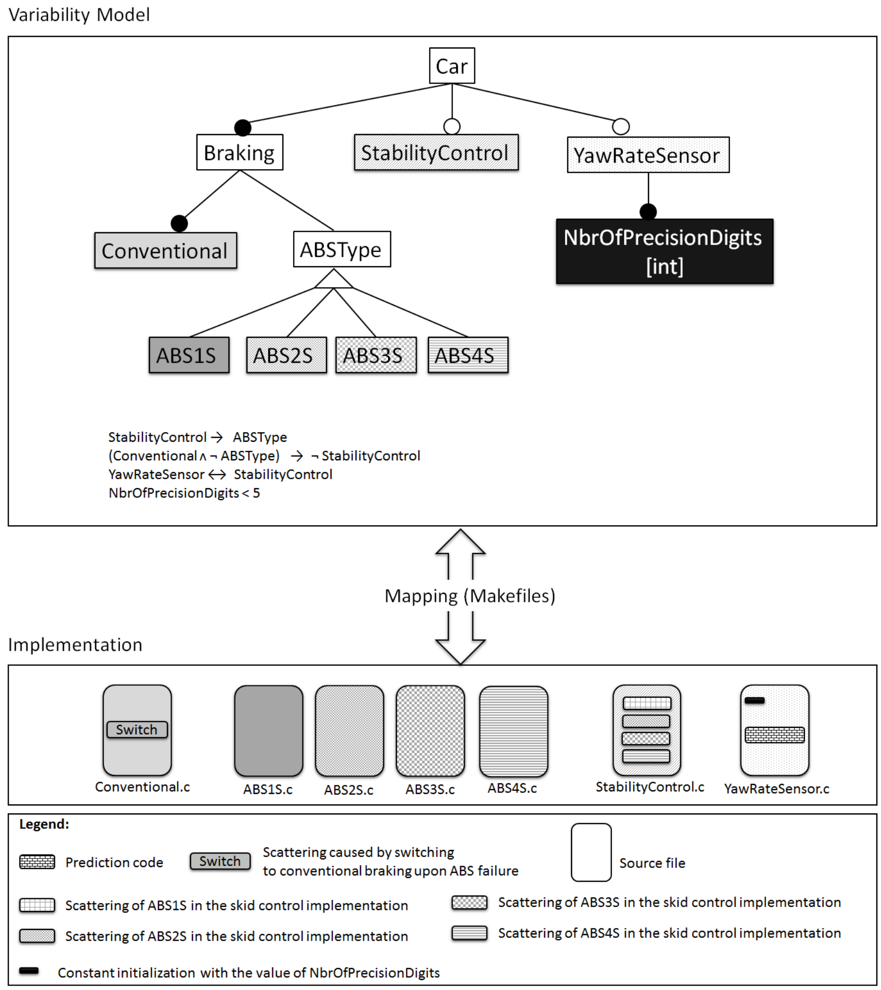

Variability Model
Car
Braking
StabilityControl
YawRateSensor
Conventional
ABSType
NbrOfPrecisionDigits [int]
ABS1S
ABS2S
ABS3S
ABS4S
StabilityControl → ABSType
(Conventional ∧ ¬ ABSType) → ¬ StabilityControl
YawRateSensor ↔ StabilityControl
NbrOfPrecisionDigits < 5
Mapping (Makefiles)
Implementation
Switch
Conventional.c
ABS1S.c
ABS2S.c
ABS3S.c
ABS4S.c
StabilityControl.c
YawRateSensor.c
Legend:
Prediction code
Switch
Scattering caused by switching to conventional braking upon ABS failure
Source file
Scattering of ABS1S in the skid control implementation
Scattering of ABS3S in the skid control implementation
Scattering of ABS2S in the skid control implementation
Scattering of ABS4S in the skid control implementation
Constant initialization with the value of NbrOfPrecisionDigits

Figure 1.5: Addition of new ABS-related features (snapshot t_3)

of other commonly researched subjects, as reported by the SPL2Go website.[4] Hence, in comparison, the Linux kernel is likely to be more representative of the complexity found in real-world settings than any other subject thus far considered.

2. Structurally, the Linux kernel resembles many other variability-aware systems, including both open-source [21, 96, 100] and industrial-based solutions [17, 69]. Hence, understanding the Linux kernel variability evolution is of potential interest to a whole set of systems.

[4] `http://spl2go.cs.ovgu.de`

3. Due to the its public evolution history, the Linux kernel as a case study favors independent verification of research results, increasing the validity of one's investigation.

Next, we outline our two major research steps towards our research goal. The distilling of the main research question and related methodological details are deferred to Chapters 3, 4, and 5.

1.3.1 Step One: Linux Kernel as an Exploratory Case Study

From the assumption that poor variability evolution support follows from a lack of knowledge of how it happens in practice, we set to understand variability evolution in the context of the Linux kernel, a real-world example of a large and complex variability-aware system. To cope with the sheer size of the kernel and its evolution history, we scope investigation to the cases where variability evolution is induced by the addition of new features or the removal of existing ones. By sampling the Linux kernel evolution, we observe how developers add and remove features from the variability model and how they subsequently change related artifacts (e.g., build files and C source code). Our goal is to mine *emergent variability-coevolution patterns*. Emergent, in this case, means that we do not prescribe the patterns we extract when mining the commit history. Rather, they arise from the context under analysis [35]. For each pattern we mine, we also track its usage—for instance, by observing how developers apply the pattern, the variability mechanisms underlying the pattern usage, and how instances of the pattern relate to the Linux kernel architectural decomposition.

Our interest in identifying variability-coevolution patterns stems from the practical impact that pattern research has had in modern software engineering. Patterns have been used as a main medium for documenting established practices, including reusable design [58], architectural styles [56, 85, 122], refactoring opportunities [57], network topologies [130], security integration strategies [120], common user interface design [135], etc. By capturing the practices in a given domain, patterns favor vocabulary standardization, contributing to better communication and facilitating training. Patterns have also driven better tool support (as we aim to). For instance, most modern integrated development environments (e.g., Eclipse,[5] Netbeans,[6] and Visual Studio[7]) automate many refactoring patterns, such as those in Fowler et al. [57].

[5] http://www.eclipse.org

[6] https://netbeans.org

[7] https://www.visualstudio.com

Our notion of pattern adheres to the *Rule of Three*, which states that a pattern is any real-world solution whose application occurs at least three times.[8] The rule provides a common ground for operationalizing pattern identification. Many researchers adopt it, making the rule a well-accepted practice in pattern analysis [57, 85, 103]. To avoid bias, it is common practice to further restrict the rule to assure three distinct sources of evidence—e.g., three distinct developers must apply a pattern, or it must appear in the evolution history of at least three different systems, etc [85]. When mining the Linux kernel commit history, we only report a pattern when we assure that at least three distinct developers apply it.

1.3.2 Step Two: Further Assessments

The exploratory analysis of the Linux kernel reveals two main findings: (i) a catalog of 23 coevolution patterns, covering five main evolution scenarios—adding features from completely new elements, creating features out of existing elements (featurization), renaming existing features, merging others, and retiring unmaintained and/or buggy features; (ii) preliminary evidence of feature scattering as common practice in Linux kernel development.

The first finding is of particular interest to tool builders, as our patterns capture evolution cases that other studies in the literature do not report. Moreover, our patterns evidence changes that cannot be correctly handled by existing variability evolution techniques. Consequently, our catalog serves as an initial benchmark for evaluating existing tools, or may help devise better ones. From the Linux kernel alone, however, we cannot evidence whether our patterns are general, nor whether they are expressive enough to explain the variability coevolution in other systems. **Hence, a further assessment consists in validating our patterns in terms of their expressiveness and generality in systems other than the Linux kernel**.

To validate our patterns, we carefully choose three other variability-aware subjects within the same domain of the Linux kernel—the systems software domain. Our target subjects are: axTLS (a client/server TLSv1 SSL library), ToyBox (a combination of individual command-line utilities), and uClibc (a C library optimized for embedded systems). To assess expressiveness, we investigate the extent that our patterns explain the coevolution induced by adding or removing features in the variability models of our three chosen subjects. By matching patterns against the evolution of axTLS, Toybox, and uClibc, we find that our patterns explain 64 % of all the additions and feature removals in our scope. When assessing generality, we restrict the *Rule of Three* by requiring the evidence that

[8] `http://c2.com/cgi/wiki?RuleOfThree`

at least three distinct developers apply any given pattern, as well as that the pattern appears in the evolution of at least three systems. We claim each pattern satisfying such a constraint as a general case within our target domain. If a pattern does not satisfy the previous constraint, but its inverse pattern does, we also claim the former as general. In total, we find 14 general patterns.

In the case of our second finding, our patterns suggest scattering as a common practice in the Linux kernel development. Such finding is of major importance. On one hand, intermingling features with different implementation parts can harden evolution—e.g., by causing ripple effects. Consequently, scattered features may significantly increase the maintenance effort of a system [7, 112]. In contrast, feature scattering allows developers to overcome design limitations when extending a system in unforeseen ways [112], or when circumventing modularity limitations of programming languages, which impose a dominant decomposition [8, 127, 131]. In other cases, the cost of modularizing features might be initially prohibitive or simply too difficult to be handled in practice [78]. Feature scattering, on the other hand, requires little upfront investment [7].

The amount of scattering Linux kernel developers are willing to accept, possible influencing factors, and how scattering relates to the evolution of the system as a whole are not known at this point. Such knowledge is key in contributing to widely accepted practices governing feature scattering. It also advances knowledge towards a general scattering theory, which could serve as a guide to practitioners—for instance, in identifying implementation decay, assessing the maintainability of a system [55], identifying scattering patterns [54], or setting practical scattering thresholds [111]. At the time of this writing, no empirical study has investigated feature scattering evolution in the context of a large and long-lived software system. **Thus, as a further research step, we analyze scattering along the Linux kernel evolution**, performing a longitudinal analysis of almost eight years of the kernel's 20-year history. To handle the Linux kernel complexity, we scope our analysis to features in the driver subsystem, which we identify as the largest and fastest growing kernel part.

1.3.3 Research Contributions

Our work provides the following research contributions:

- A methodology to extract patterns from the evolution history of variability-aware systems. We design our methodology based on the organization of the Linux kernel around three main artifact types: variability model, build files, and C source code.

Since other variability-aware systems follow the same organization, including both open-source [21, 100] and industrial product lines [17], our methodology is likely to be replicable in the study of other systems.

- A detailed study of how variability models coevolve with different artifacts in the context of a large, complex, and continuously evolving variability-aware software system: the Linux kernel.

- A taxonomy for the observed coevolution, organized as a catalog of 23 variability-coevolution patterns.

- An assessment of the expressiveness of our coevolution patterns.

- The identification of 14 general patterns within our original catalog.

- A set of principles on how to implement scattered features using C pre-processor annotations, along with possible alternatives. Altogether, the principles we report ease maintenance and evolution, while also being beneficial to other variant-rich systems employing C pre-processor annotations.

- Empirical evidence of the limitation of state-of-the-art variability evolution techniques when handling evolution scenarios captured by some of our patterns.

- Empirical evidence in favor of a new theory for software-product-line evolution. While many of our patterns are captured by the existing theory of software-product-line refinement [22], feature-retirement patterns are not. Since the latter are too frequent to be ignored, a new theory should be devised to account for feature retirement.

- A methodology for identifying pattern instances in the repositories of variability-aware systems. We successfully apply our methodology to identify pattern instances across the evolution of axTLS, Toybox, and uClibc.

- Three publicly available datasets covering, (i) the mining of patterns in the Linux kernel sample; (ii) the mining of pattern instances in axTLS, Toybox, and uClibc; (iii) scattering-related data from almost eight years of Linux kernel evolution. Altogether, readers may use our datasets as replication packages, as benchmarks for tools, or as baselines for further analyses.

- Descriptive statistics aimed at understanding the state-of-practice of feature scattering in the Linux kernel.

- An inspection and classification of 170 scattered device drivers from the Linux kernel, from which we test hypotheses to verify possible factors influencing where a feature is scattered across or its scattering degree (a measure of a feature's spread in code).
- An *online appendix* [1] with full access to our datasets, R scripts, and the source code of all our custom-made tools.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents related work investigating variability evolution. Chapter 3 presents our exploratory analysis when investigating variability-coevolution patterns in the Linux kernel. Chapters 4 and 5 build on the findings of the Linux kernel case study. Specifically, Chapter 4 investigates the external validity of our Linux kernel coevolution patterns, assessing their expressiveness and generality across the evolution of three variability-aware systems (axTLS, Toybox, and uClibc). Chapter 5 seeks to better understand feature scattering by performing a longitudinal study of Linux kernel device-driver features. In particular, we investigate how scattering evolves over time, whether practical limits exist, possible influencing factors, etc. We present our final remarks in Chapter 6, as well as outlining a research agenda for future work.

1.5 Conventions

This thesis employs the following language and typesetting conventions:

- When we present summary statistics, the mean should always be understood as the arithmetic mean. So does the average, a term that we use interchangeably with the mean.
- Our notion of a feature is always of a configurable option declared in a variability model.
- When writing code snippets, we use the `monotype font`.
- When first introducing a term, we write it in *italics*.
- We typeset references to source code elements (e.g., functions), feature names, and filenames with Serif font.

- We generically refer to C pre-processor annotations `#ifdef`, `#ifndef`, `#if`, and `#elif` as *ifdefs*.

1.6 Publications

This thesis contains material from the following publications:

- Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. *Coevolution of Variability Models and Related Software Artifacts: A Fresh Look at Evolution Patterns in the Linux Kernel.* In Empirical Software Engineering, Springer, 2015.

- Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Túlio Valente: *Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers.* In 14th International Conference on Modularity, ACM, 2015.[9]

- Leonardo Passos and Krzysztof Czarnecki. *A Dataset of Feature Additions and Feature Removals from the Linux Kernel.* In 11th Working Conference on Mining Software Repositories (Data track), IEEE/ACM, 2014.

- Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. *Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel.* In 17th International Software Product Line Conference, ACM, 2013.

- Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, Jianmei Guo, and Claus Hunsen. *Feature-Oriented Software Evolution.* In 7th International Workshop on Variability Modelling of Software-intensive Systems, ACM, 2013.

1.7 Awards

During the development of this thesis, we received two awards at the 14th International Conference on Modularity:

[9] The Modularity conference was previously named AOSD—Aspect-Oriented Software Development.

- Best research paper.
- Bronze medal in the Student Research Competition. The competition was sponsored by Microsoft Research.

Table 1.1: Popular variability-aware research subjects

Name	Language	Number of features	Size (LOC)
Arithmetic Expression Evaluator	Haskell	25	460
Battle of Tanks	Java	144	1,975
Chat HyperJ	Java	7	758
DesktopSearcher	Java	22	3,779
Elevator C	C	6	877
Elevator Java	Java, AspectJ	6	1,046
Email JML	Java, JML	9	1,233
Email System C	C	9	258
Email System Java	Java, AspectJ	9	1,233
FAME DBMS	C++	14	5,000
Graph Product Line	Jak, HTML	18	1,350
Lampiro	JavaME	11	45,000
Mine Pump C	C	7	279
Mine Pump Java	Java, AspectJ	7	580
Mobile Media	JavaME	14	5,700
Mobile RSS Media	JavaME	14	20,000
NetBot	C#	6	1,579
Prevayler	Java	5	80,000
Prop4J	Java	17	2,051
Pynche	Python	12	2,400
SQL Parser	AntLR DSL	4	60
Tank War	Java	37	5,578
Vistex	Java	16	2,405

Chapter 2

Variability Evolution

This chapter summarizes most of the relevant works investigating variability evolution. We divide presentation according to the evolution of three different *spaces* (or artifact types), namely variability model, mapping (e.g., build files), and implementation files (e.g., C source code). Section 2.1 discusses the works studying variability model evolution, followed by a presentation of the works targeting the evolution of the mapping (Section 2.2) and implementation files (Section 2.3). Section 2.4 presents the relevant research focusing on how variability evolves across different spaces. Section 2.5 concludes the chapter.

2.1 Variability Model Evolution

Most of the existing research on variability evolution focuses on the evolution of variability models.

Foundational work investigates the semantics of variability models, defining it in terms of a satisfiability problem [11, 94]. In the case of Boolean variability models, whose features users either select or exclude from configuration, researchers translate variability models into propositional logic formulae or CSP-based specifications, applying SAT, Binary Decision Diagrams, or CSP solvers to perform different types of analyses as the input models evolve [16]. Example analyses include consistency checks, dead feature detection, counting variants, interactive guidance during configuration, or fixing models and configurations [16, 143, 145]. Recent configuration fixing approaches also tackle non-Boolean variability models [142, 149]. Opposed to Boolean variability models, non-Boolean ones allow features to store numeric or string values. Their analyses, however, require support for numeric and string theories, generally backed by the use of SMT Solvers.

Different techniques aim at supporting variability model changes. Czarnecki et al. define operations to specialize variability models [38, 39, 83]—a variability model specializes another if the set of variants of the former is a subset of the latter. Alves et al. [4] investigate variability model refactorings, presenting a catalog of refactoring templates (operations). Each template either preserves the original set of derivable variants or increases it with new variants. In the latter case, the resulting variability model generalizes the one preceding the template application. Thüm et al. [133] propose an edit-based reasoning technique for assessing how variability model changes affect the set of possible variants. The authors classify changes according to four categories: specialization, generalization, refactorings, and arbitrary edits (none of the previous three). Different from Alves et al., the refactoring notion of Thüm concerns changes that do not add new variants, nor exclude existing ones. At the core of their edit reasoning, the authors efficiently translate the original variability model, as well as the one resulting from the changes, into a satisfiability constraint problem, encoding it in conjunctive normal form (CNF). By avoiding an exponential explosion of CNF clauses, Thüm's approach handles large models, showing scalability with randomly-generated models with up to 10,000 features. Moreover, their reasoning does not require variability models to have the same set of features, as generalization includes new ones, whereas specialization remove others. This contrasts to previous work analyzing equivalence or specialization of variability models [71, 128]. Guo et al. [64] define a set of primitive operations for evolving variability models. Assuming the initial version of the model to be consistent, the authors propose consistency checks for change requests, providing additional operations to restore consistency.

Despite the great effort in studying variability model evolution, most techniques lack empirical validation with realistic models, which are often times unavailable to researchers. Lacking realistic large scale models, researchers generally resort to using randomly-generated models—e.g., as in Thüm et al. [133] and Guo et al. [64]. To change figures, She et al. [123] propose the Linux kernel variability model as a realistic benchmark directly accessible to researchers. By analyzing various metrics (e.g., branch factor, cross-tree constraint ratio, depth, etc), the authors show that the Linux kernel variability model surpasses the complexity of most models within the research community. Lotufo et al. [91] extend She's work with a longitudinal analysis of the Linux kernel variability model targeting the x86 hardware architecture. The authors also present evolution scenarios and operations that developers commonly face when evolving the target model. Similarly, Passos et al. [108] analyze the constraints in 116 variability models of the eCos real-time operating system [96], providing a benchmark for non-Boolean model analyses. Berger et al. [21] compare different variability models from real-world systems in the systems software domain, including the Linux kernel and eCos.

To better understand the Linux kernel variability model evolution, Dintzner et. al. [44] propose an approach to extract and classify changes from historical versions of the Linux kernel variability model. The authors validate their analyses in different versions of the Linux kernel, targeting 21 different hardware architectures.

2.2 Mapping Evolution

In this thesis, we interpret the mapping as a set of build rules, which associate source files to a set of enabling features.

Generally, variability in build files follows a simple approach [7]. Exploiting the imperative nature of build files, developers write selection statements or pre-processor directives to vary build behaviour according to different target configurations. Other systems, however, prevent developers from doing so. For instance, in systems employing the variability modeling language of the eCos operating system [142, 149], developers write build rules as feature attributes [139]. Upon a feature selection, build rules are put together in a compositional manner, resulting in a variant-specific build file. Although it is reasonable to assume that build rules as feature attributes are likely to easy maintenance, empirical evidence is lacking. This is not surprising, as few researchers set to investigate build files from a variability perspective. Exceptions, however, do exist, mostly in the context of Linux kernel Makefiles.

Berger et al. [19] and Nadi et al. [102] parse Linux kernel Makefiles to statically extract presence conditions of compilation units—a presence condition is a Boolean expression over feature names [36, 101]; in the case of Makefiles, it controls under what configurations a file is compiled or not. Connecting the presence conditions stemming from Makefiles to the propositional semantics of variability models allows detecting build inconsistencies, such as determining which files will never compile. Makefile parsing often depends on specific idioms of how developers write build rules. The work of Berger and Nadi are not an exception. Thus, their results are as good as the extent that developers follow Linux kernel idiomatic styles. Dietrich et al. [41] also extract presence conditions of Linux kernel compilation units. The authors, however, do not parse target Makefiles. Instead, their approach instruments the Linux kernel compilation to enable one feature at a time, keeping track of the files getting compiled upon each feature selection. While an approximation, avoiding the parse of Makefiles allows their approach to process different releases throughout the Linux kernel evolution.

Seeking to statically extract configuration knowledge from Makefiles, Zhou et al. [151] present an early prototype for determining the presence condition of each compilation unit in a target project. Their work builds on top of SYMake [129], which employs symbolic execution to conservatively analyze all possible execution paths of a Makefile script.

2.3 Implementation Evolution

There are different approaches to encode variability in implementation-based artifacts, each with different trade-offs. In the following, we summarize the major techniques, along with related work. For a through comparison among different techniques, we refer readers to Apel et al [7].

2.3.1 Evolution in Annotative-based Approaches

Annotative approaches are one of the simplest methods for variability encoding. They are also one of the most widely used mechanisms for creating variable source code. Examples of annotative approaches include C pre-processor conditionals (e.g., `#ifdef`) or selection statements to branch a program's control-flow. Both approaches are examples of *if* conditions. Another form of annotation consists in using libraries of compile-time constructs, written, for example, with meta-programming facilities. In that direction, Czarnecki and Eisenecker propose different compile-time control structures, implementing them with C++ templates [37].

As an alternative to textual annotations, Kästener et al. [75] propose the use of code coloring. In their approach, features map to individual colors; using a custom-made tool (CIDE), users color the lines of code that implement a particular feature. In the case where a line supports multiple features, all colors apply. In the latter case, distinguishing combinations of features can be particularly difficult. In [77], Käster et al. evaluate colored-based annotations when refactoring two existing systems into product lines. From their evaluation, the authors argue in favor of annotative approaches for developing product lines whose features associate with fine-grained code fragments. The authors also discuss facilities in CIDE to aid developers, including feature navigation, projection facilities, and the exporting of colored code as feature modules.

Unarguably, annotation-based approaches provide a simple mechanism for variability encoding. In the long-term evolution of a system, however, simplicity may come with a price. While requiring little pre-planning and upfront investment [7], if not used with care, annotations may lead to extensive scattering and feature intertwining (a.k.a, *tangling*) [87, 89]. Scattering and tangling have long been under suspicion to negatively impact quality and maintainability—for example, by causing bugs [48, 79] or ripple effects [111]. Others also argue that annotations decrease program comprehension, as they clutter control-flow reasoning [51, 75, 86, 125].

To prevent decay due to extensive use of annotations, Queiroz et al. [111] propose the use of threshold values to keep feature scattering and tangling within acceptable limits. From the observation that code-related metrics often follow heavy-tailed distributions [15, 92, 110, 144], the authors argue that scattering and tangling thresholds should not be based on a single limit value (e.g., mean); instead, they should be *relative*. A relative threshold for a metric M defines a percentage p and a limit k such that at most $p\,\%$ of source code entities have $M \leq k$ [105]. From p, it follows that at most $(100 - p)\,\%$ of code elements have $M > k$. Since there are varying levels for p and k, the authors analyze a corpus of 20 long-lived C pre-processor-based systems from different functional domains, extracting empirical thresholds for scattering and tangling-related metrics. The authors find scattering distribution to be highly-skewed, but note that tangling is mostly uniform across their target subjects. In both cases, the authors report between 80 % and 85 % of features having low scattering and tangling values, respectively. Thus, the authors argue that at most 15–20 % of features should be highly scattered or tangled with other features.

Other researchers seek understanding the practice of using C pre-processor annotations when encoding a system's variability. Liebig et al. [87, 89] investigate different metrics to assess the complexity of *ifdef* annotations in a corpus of 40 systems. Among their findings, the authors state that most annotations in the source code of their subjects enframe entire functions or code blocks—e.g., entire selection statements or loops, suggesting that such annotations could be encoded with alternative techniques favoring better modularity (e.g., aspects). Hunsen et al. [69] investigate the complexity of C pre-processor annotations in both open-source and industrial-based systems. Their research shows that both types of systems share similar complexity. Hence, studies focusing on the variability complexity as given by C pre-processor annotations in open-source systems are likely to be representative to what occurs in industrial settings.

From an engineering perspective, different researchers aim at improving existing tool support to parse, type-check, and refactor annotative-based variant-rich systems. For instance, Gazzillo et al. [60] provide a complete variability-aware parser that copes with all the complexity of the C pre-processor. Opposed to other tools that take C pre-processor-based code as input [24, 59, 62, 106, 141], their solution does not parse source code by processing one configuration at a time, nor does it rely on heuristics or specific coding idioms. Similar to Garrido, Kästener et al. [79] also implement a variability-aware parser, along with subsequent variability-aware static analyses [80, 90]. Recent advances in variability-aware refactoring follows directly from better parsing support [88].

2.3.2 Evolution in Modular-based Approaches

Ideally, all features should be modular, in a one-to-one mapping style. Different approaches seek such a goal. We explain the main ones in the following.

GenVoca/AHEAD. Batory et al. [14] is among the first ones exploring the ideas of feature modularity, investigating mechanisms to represent and compose features to derive specific variants. The authors propose GenVoca, an algebraic model for Feature-Oriented-Programming [13], later generalizing it into AHEAD—Algebraic Hierarchical Equations for Application Design [12]. Different from GenVoca, AHEAD works not only with source code, but with all kinds of software artifacts. Despite their elegance, GenVoca and AHEAD have been confined to academic realms [7]. Consequently, there are no empirical studies investigating evolution of real-world GenVoca/AHEAD-based systems.

Aspect-Oriented Programming (AOP). Outside academic setups, researchers at Xerox propose the Aspect-Oriented-Programming paradigm [82].

AOP aims at modularizing so called *cross-cutting concerns*, which, due to the tyranny of the dominant decomposition of existing programming languages [8, 127, 131], favor the modularization of certain concerns in detriment of others [97]. Thus, cross-cutting concerns are inherently scattered across the code, with varying scattering degrees. It is worth noting that concerns comprise not only features (in the sense of how we use the term), but also requirements, design elements, design patterns, and programming idioms [48, 54].

AOP modularizes cross-cutting concerns in the form of aspects, which hook to existing declarations by adding new elements (e.g., an attribute to a class) or intercepting specific execution points to add desired functionalities (e.g., validate input parameters before a method call). In AOP terminology, extensions to existing declarations are called *advices* and interceptors are known as *pointcuts*. Examples of typical cross-cutting concerns include tracing [25] and exception handling [26].

Compared to other existing compositional approaches, AOP has support across different languages, such as AspectJ (extends Java), AspectC (extends the C-language), AspectC++ (extends C++), etc. Different frameworks also support AOP, including enterprise-based solutions, such as Spring[1] and JBoss.[2]

[1] `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html`

[2] `http://docs.jboss.org/aop/1.3/aspect-framework/reference/en/html_single/`

Different empirical works investigate AOP from a software evolution perspective. For instance, Käster et al. [76] report their experience in refactoring Berkeley DB into an aspect-based product line. Opposed to common belief, the authors notice that, as the number of aspects grow, code readability and maintainability decrease; overall, they argue against aspect adoption as a mainstream technique for refactoring legacy systems into product lines. Figueiredo et al. [54] investigate 13 cross-cutting patterns and their impact on the design stability in seven releases of three target applications, with implementations in Java and AspectJ. The authors notice that some patterns are not easily expressed with typical AOP constructs. Moreover, they find certain cross-cutting patterns to strongly indicate change-proneness of certain classes; for other patterns, such relation is not evident. Apel [6] investigates 11 AspectJ academic-based systems. In his set of subjects, the author finds that 2 % of the target code requires advanced cross-cutting mechanisms, whereas 12 % requires basic ones; the remainder 86 % comprises object-oriented code. Thus, Apel argues that simple AOP language mechanisms are expressive enough to modularize most of the cross-cutting concerns in the set of studied subjects. Munoz et al. [99] perform a similar analysis by studying 38 open-source AOP-based systems. The authors also conclude that AOP-language constructs are underused, while also noting that developers use aspects in a cautions way.

While striving to achieve better modularity, AOP is not free of criticism. For instance, due to glue-code, optimizations, and/or code obfuscation, debugging is often times hard [150]. Moreover, aspects may hinder control-flow reasoning, as they may extend a base code in many and often unknown ways—in AspectJ, for instance, hooking does not follow pre-defined interfaces. Furthermore, AOP allows violating basic modularity principles, breaking information hiding and module interfaces [7]. Additional evolution problems include composition ordering and the difficulty of coevolving pointcuts with base code [97].

Delta-Oriented Programming (DOP). Delta-Oriented-Programming is a recent programming paradigm aiming to support the development of software product lines.

In a traditional DOP-lifecycle, developers first create a complete working system—the core module. Deltas then add or modify programming elements in the core module. Like the C pre-processor, but unlike aspects, deltas may also remove code.

When deltas inadequately modify the core module, resulting variants may be ill-formed. Avoiding such problem requires reasoning over separate deltas and the core module, potentially hardening maintenance and programming comprehension [7].

Like GenVoca and AHEAD, DOP does not have significant industrial adoption. Thus, there is no evidence of its use in building large and complex real-world product lines.

2.3.3 Others Approaches

In addition to annotative and modular-based approaches for variability encoding and their different trade-offs, other approaches also exist.

At a basic level, program arguments (e.g., command line parameters) allow users to vary the behaviour of a given system upon its call. Likewise, configuration files, such as property files in Java projects, are also an alternative. In either case, variability is diluted in code, requiring feature location techniques to allow proper variability management [28, 40, 45, 49, 95, 124, 137, 146].

Code cloning is also an alternative for supporting variability. With the aid of a version control system (e.g., Git,[3] Mercurial[4], or Subversion (SVN)[5]), developers clone an existing branch, adapting it according to a set of requirements. As the system evolves and need dictates, developers may also merge-and-refactor branches [7, 117]. Despite a rapid mechanism for developing experimental features, cloning has long been argued to harden software evolution—variability management in cloned variants only adds further complexity. Consequently, existing software product-line-engineering adoption approaches discourage cloning [66, 72]. Nonetheless, certain industry niches perceive cloning as favorable, mainly due to its initial low adoption costs and freedom for independent modifications [46]. Acknowledging that some organizations rely on cloning as a mainstream practice, Rubin et al. [115, 116] propose a framework to organize development, maintenance, and merge-refactoring knowledge of specific variants and the system as a whole.

2.4 Variability Coevolution

Aiming to combine the flexibility of code-cloning with the benefits of traditional product-line engineering (e.g., [31]), Antkiewicz et al. [5] propose a virtual platform for incremental product-line adoption. The platform comprises seven levels of governance, from adhoc-clown-and-own (L0) to a product line engineering with a fully integrated platform (L6). As a target systems evolves through different levels, developers coevolve reusable assets, ultimately, deriving a variability model.

Other researchers set focus on detecting inconsistencies arising from evolving the different spaces of the Linux kernel. For instance, Tartler et al. [132] detect inconsistencies

[3] `https://git-scm.com/`

[4] `http://mercurial.selenic.com/`

[5] `http://subversion.apache.org/`

between the Linux kernel variability model and *ifdefs* in code (e.g., an #ifdef whose condition cannot be satisfied given the set of cross-tree constraints). Nadi et al. [102] extend Tartler's framework to additionally detect inconsistencies in Makefiles (e.g., a build rule is dead due to an inconsistency with the constraints in the variability model). Their framework is also more accurate; when detecting inconsistencies in *ifdefs* in code, they take into account the constraints in the Linux kernel variability model as well as those in Makefiles. The works of Tartler et al. and Nadi et al. are one of the few targeting variability coevolution in the large. Generally, researchers rely on small target subjects or even employ artificial examples. Neither approach shall be representative of the complexity of real-world settings.

Neves et al. [104] manually extract evolution templates (operations) from the commit history of a small product line (< 50 features), validating their catalog against later releases of their original product line and in two other smaller systems. Their templates conform to the refinement theory in [22], assuring that old variants can still be mapped to variants in the product line resulting from a template application. Refinement changes are behaviour preserving, allowing systems to *safely evolve*. Such a notion, however, does not capture feature retirement, which necessarily shrinks the set of possible variants.

Holdschick [67] presents change operations between variability models and functional models in the automotive domain, seeking better understanding coevolution challenges. Similarly, Seidl et al. [121] provide a set of evolution scenarios and mapping operators to reestablish the correct binding of different spaces in a software product line. In both works, the authors do not provide empirical evidence supporting the need of their operators nor scenarios.

Schulze et al. [119] present a catalog of 23 refactoring patterns for evolving DOP-based product lines. Among other things, their catalog supports the coevolution of variability models and deltas. Their patterns are based on those from Fowler et al. [57], but adapted to the context of DOP. From such a lifting, the authors assume that their patterns are expressive enough to capture refactorings from real-world settings. Ultimately, however, it is unknown whether the evolution of real-world DOP-based systems resembles the evolution of object-oriented ones, casting doubt on their catalog expressiveness. Although the authors provide a prototype to automate their proposed refactorings, they do not test it against real-world cases.

2.5 Conclusion

When revisiting related work, we evidence a lack of empirical studies investigating variability evolution in the context of large and complex real-world systems. Most of the existing efforts focus on the evolution of variability models alone, with supporting tools being validated with randomly-generated models or small case studies. Few studies set to investigate the mapping evolution—thus, tool support is not yet mature. In contrast, we notice great advances at the implementation space, notably better support for variability-aware parsing.

In the coevolution arena (our focus), few studies set to investigate how variability models coevolve with other related artifacts. Hence, mature tool support is still lacking, as existing tools are backed by little or no empirical evidence supporting them.

Chapter 3

Linux as an Exploratory Case Study

This chapter discusses the Linux kernel as an exploratory analysis of variability evolution. Among others, we present the overall structure of the Linux kernel and how we mine coevolution patterns from the kernel's evolution history. We mine 23 patterns, detailing their usage and discussing related findings that we use to guide further investigations, which we present in later chapters.

Chapter Organization. Section 3.1 explains the Linux kernel structure and how it links to the notion of a feature. In Section 3.2, we discuss the Linux kernel evolution and its relation to the evolution of the kernel's feature set. Section 3.3 presents our research question and underlying assumption. We detail our research methodology in Section 3.4, presenting results in Section 3.5. Section 3.6 discusses how our patterns align with our main research goal, detailing key findings. Section 3.7 argues about possible threats to validity, followed by our conclusions in Section 3.8.

3.1 The Three Spaces of the Linux Kernel

The Linux kernel is structured in three spaces [19, 42, 101]: (i) it has a variability model, which explicitly captures user-related configuration options and their associated constraints; (ii) a mapping, which links features to specific compilation units; (iii) an implementation space, which realizes all the kernel's features. Such structure is not exclusive to Linux; rather it is also found in other open-source variability-aware systems [21, 100] and industrial product lines [17]. Following the steps in Figure 3.1, we describe how each of these spaces works and how they are connected.

3.1.1 Variability Model

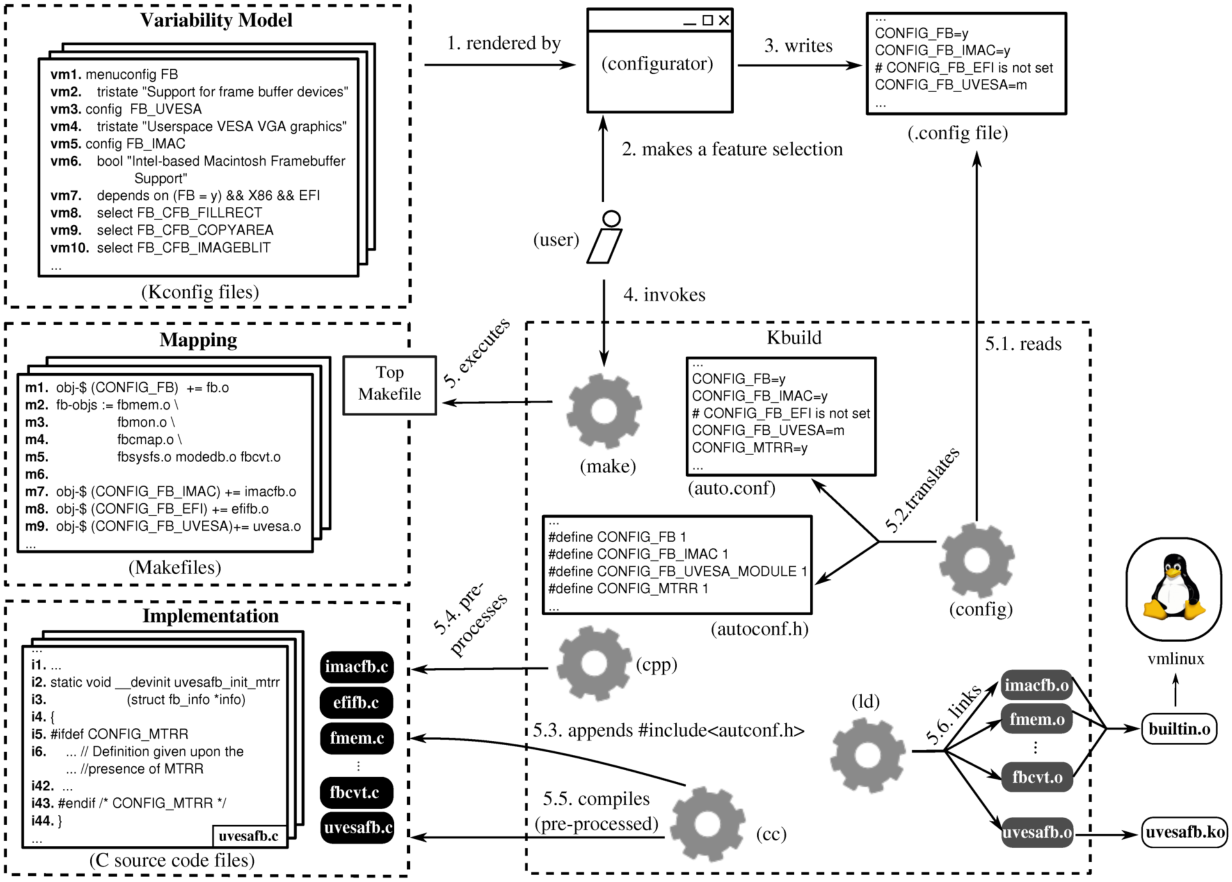
The Linux kernel variability model comprises a set of files written in the Kconfig language [81]. A configurator renders (step 1) a tree of features from Kconfig files that are available for the user's platform (i.e., processor family). From it, users select features that should be present in the resulting kernel (step 2).

As shown in the excerpt in Figure 3.1, the features in the Linux kernel variability model are generally config declarations (lines vm3 and vm5). In our example, FB (the parent of all frame-buffer-related features)[1] and FB_UVESA (a generic frame-buffer driver) are tristate features (lines vm2 and vm4). They can be absent (n) or present either as dynamically loadable kernel modules (m) or by being statically compiled into the resulting kernel (y). Boolean features are also possible (line vm6), assuming either y or n as value. Other types include integer and strings (not shown).

In Kconfig, features may contain attributes. The prompt attribute is a short text describing the feature (lines vm2, vm4 and vm6). The configurator uses the prompt to render feature nodes in the hierarchy (the absence of a prompt makes a feature invisible to users). A default attribute (not shown) provides an initial value of the corresponding feature, which can be later changed during configuration. Two specific attributes define cross-tree constraints: depends on and selects. The depends on attribute (line vm7) is a dependency condition that, if satisfied, allows users to select the feature with this attribute. Not all dependencies result in cross-tree constraints, as the Linux kernel configurator uses some dependencies as a means to define the parent of a feature. A select attribute is a reverse dependency that enforces the immediate selection of one or more target features. For example, selecting FB_IMAC causes the immediate selection of FB_CFB_FILLRECT, FB_CFB_COPYAREA, and FB_CFB_IMAGEBLIT (lines vm8–vm10).

Once the user finishes selecting features and setting their parameters, the configurator saves the configuration in a .config file (step 3). The latter is a sequence of feature-name=value lines; when writing feature names, the configurator prefixes them with CONFIG_. Prefixed names are then referenced in the mapping and source code as a means to react to the presence of specific features.

[1] https://www.kernel.org/doc/Documentation/fb/framebuffer.txt



```
vm1. menuconfig FB
vm2.    tristate "Support for frame buffer devices"
vm3. config  FB_UVESA
vm4.    tristate "Userspace VESA VGA graphics"
vm5. config FB_IMAC
vm6.    bool "Intel-based Macintosh Framebuffer
                Support"
vm7.    depends on (FB = y) && X86 && EFI
vm8.    select FB_CFB_FILLRECT
vm9.    select FB_CFB_COPYAREA
vm10.  select FB_CFB_IMAGEBLIT
...
```

```
...
CONFIG_FB=y
CONFIG_FB_IMAC=y
# CONFIG_FB_EFI is not set
CONFIG_FB_UVESA=m
...
```

```
m1.  obj-$ (CONFIG_FB)  += fb.o
m2.  fb-objs := fbmem.o \
m3.                fbmon.o \
m4.                fbcmap.o \
m5.                fbsysfs.o modedb.o fbcvt.o
m6.
m7.  obj-$ (CONFIG_FB_IMAC) += imacfb.o
m8.  obj-$ (CONFIG_FB_EFI) += efifb.o
m9.  obj-$ (CONFIG_FB_UVESA)+= uvesa.o
...
```

```
...
CONFIG_FB=y
CONFIG_FB_IMAC=y
# CONFIG_FB_EFI is not set
CONFIG_FB_UVESA=m
CONFIG_MTRR=y
...
```

```
...
#define CONFIG_FB 1
#define CONFIG_FB_IMAC 1
#define CONFIG_FB_UVESA_MODULE 1
#define CONFIG_MTRR 1
...
```

```
...
i1. ...
i2. static void __devinit uvesafb_init_mtrr
i3.                 (struct fb_info *info)
i4. {
i5. #ifdef CONFIG_MTRR
i6.     ... // Definition given upon the
        ... //presence of MTRR
i42.  ...
i43. #endif /* CONFIG_MTRR */
i44. }
...
```

Figure 3.1: The three spaces in the Linux kernel and their interaction with Kbuild

3.1.2 Mapping

In the Linux kernel, the mapping between features and compilation units occurs mostly inside Makefiles. Kbuild, the kernel build infrastructure,[2] controls the whole compilation process of the kernel. To build a kernel image according to a given configuration, users invoke make (step 4), which triggers the execution of the top Makefile at the root of the Linux kernel source code tree (step 5). The top Makefile then invokes config, which in turn reads the configuration file (step 5.1) and translates it to two other files (step 5.2): auto.conf, later used by make, and autoconf.h, later used by the C pre-processor (cpp).

The top Makefile controls vmlinux (the resident kernel image) and dynamically loadable

[2] https://www.kernel.org/doc/Documentation/kbuild/

Table 3.1: Distribution of source code file types (averages taken from the v2.6.12–v3.9 release range)

File type	Mean (%)
C implementation file	43
C header file	39
Assembly	4
Other	14

kernel modules (LKMs), i.e., modules that can be loaded at runtime. To build vmlinux, Kbuild first builds all the object files stored in core-y, libs-y, drivers-y, and net-y variables, as stated in the top Makefile:

```
1 vmlinux := $(core-y) $(libs-y) $(drivers-y) $(net-y) ...
2 ...
3 drivers-y += drivers/ main/
```

These variables denote lists of object files to which further elements can be appended. When appending directories (line 3 above), Kbuild recursively runs the Makefile in each of the listed directories and generates all objects of a special list: obj-y (similarly, obj-m is a list for dynamically loadable modules). Objects are conditionally added to such a list by replacing y with a feature name. As shown in the Makefile of Figure 3.1 (line m7), imacfb.o is added to obj-y if FB_IMAC is set to be y in the auto.conf file (the same applies to FB_EFI and FB_UVESA, lines m8–m9). Kbuild attempts to compile object files by locating a corresponding C file with a matching name. If such file does not exist, Kbuild uses a list named after the object file and suffixed with either -y or -objs. In our example, the FB feature associates with the set of objects in the fb-objs list (lines m2–m5 in Figure 3.1); there is no fb.c file in the Makefile's directory.

3.1.3 Implementation

The implementation space contains the operating system code, comprising the realization of the features declared in the variability model, along with supporting code. As summarized in Table 3.1, the kernel's code base comprises mostly C implementation and header files.

At the code level, variability is expressed in terms of C pre-processor *ifdefs*. These annotations control the compilation of specific code fragments, henceforth referred as *extensions*. The conditions of *ifdef* annotations are essentially Boolean expressions over feature names, making the compilation of extensions dependent on specific configurations. The extensions of a feature trace back to the *ifdef* conditions referencing its identifier; counting the latter defines the *scattering degree* (*SD*) of a feature [7, 87, 89, 111]. The scattering degree metric is an indirect measure of the number of potential places that a developer may edit upon changing a feature of interest [111]. A feature is scattered when its *SD*-value is at least 2. A single extension ($SD = 1$) does not qualify a feature to be scattered, as it has no spread in the source code. In our example, MTRR is a scattered feature. In release v3.2 of the kernel, in addition to the scattering at line i5, it also adds 112 extensions elsewhere. The *SD* metric falls under the umbrella of absolute metrics that count the number of source code entities relating to a given feature. In contrast, relative metrics assess feature-scattering relative to the code size of extensions [47]. Existing research [48] comparing absolute metrics with relative ones shows that the former correlate better with defects, which justifies our choice for the *SD* metric throughout this thesis.

Upon a user configuration, Kbuild proceeds to pre-processing the kernel source code. To do so, Kbuild adds an inclusion directive to autoconf.h in each target source file (step 5.3). Such header file contains definitions for all the features listed in the .config file. In autoconf.h, Kbuild encodes macros as follows: all features in the .config file result in pre-processor symbols with the same name; tristate features selected as modules are suffixed with _MODULE; macros of selected Boolean/tristate features are set to 1; integer/string features, if present, lead to macros whose values match those given during configuration.

Given the macro definitions in autoconf.h, the C pre-processor evaluates all *ifdef* conditions, deciding which code blocks to include and which to remove (step 5.4). Then, the C compiler compiles the resulting code (step 5.5). From the example configuration in Figure 3.1, pre-processing uvesafb.c results in a non-empty body of the __devinit uvesafb_init_mtrr function (lines i6–i42), as CONFIG_MTTR is a defined macro in autoconf.h.

The last step in the compilation process links the object files in obj-y, merging them into a built-in.o file (step 5.6). The parent Makefile then links the latter into the vmlinux image. Tristate features set to m, in contrast, are not linked to the final kernel image. Rather, each object file in the obj-m list results in a dynamically loadable kernel module (.ko file).

3.2 Evolution Overview

The Linux kernel evolves continuously. Analyzing a snapshot of the kernel's Git repository[3] evidences such a fact. As shown in Figure 3.2, from June 2005, when the first stable release (v2.6.12) was put under Git control, to April 2013 (release v3.9), the Linux kernel SLOC of its C-based implementation[4] increased by 159 %, with an average growth of 2.6 ± 1.5 % between each consecutive stable release pair. The short-hand 2.6 ± 1.5 % denotes an arithmetic mean of 2.6 % with standard deviation of 1.5 %. The kernel's feature set, shown in Figure 3.2b, displays a similar trend, and strongly correlates with SLOC growth (Pearson product-moment correlation $r = 0.996$).[5] Since v2.6.12, it increased by 177 %, growing 2.8 ± 1.4 % between stable releases. The latest kernel release in the considered snapshot (v3.9) contains over 13,000 features implemented in more than 33,000 C files, amounting to over 10 million SLOC. These C files contain over 34,000 *ifdefs* that explicitly refer to at least one feature in the variability model.

3.3 Research Question

Our exploratory analysis of the Linux kernel aims at assessing how variability evolves across different spaces. Specifically, we want to mine emergent variability-coevolution patterns from the kernel's evolution history. Thus, we ask the following research question:

RQ. What variability-coevolution patterns emerge from the Linux kernel evolution?

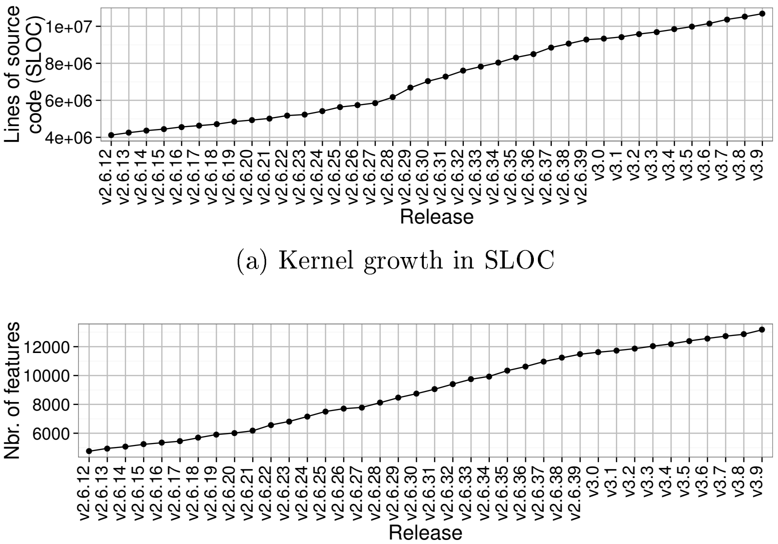
At the core of our research question lies the assumption that variability-coevolution patterns do exist.

Assumption. There exists variability-coevolution patterns that emerge from the Linux kernel evolution history.

[3] `git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git`

[4] This includes all known C implementation and header files, including non-standard extensions, such as: .inc. .uc, pgc, etc.

[5] Since the release range v2.6.12–v3.9 is a large subset of the kernel release population, the presented correlation coefficient is necessarily significant.



(a) Kernel growth in SLOC

(b) Kernel growth in number of features

Figure 3.2: Kernel growth evolution

Two main points favor our assumption: (i) as Figure 3.2 shows, feature addition closely follows SLOC growth. Thus, adding new features is likely to require the variability model to coevolve with the implementation space; (ii) the Linux kernel has well-known intrinsic structures, including a plugin architecture [34, 140] and a three-space organization. Thus, it must be the case that such structures impose a strict mechanism on how developers add new features to the kernel and how they subsequently remove features later in time. Hence, some patterns are likely to occur as a by-product of the existing structures.

3.4 Methodology

This section details our research methodology to mine coevolution patterns. We discuss our scoping decisions and data collection procedure, followed by an explanation of how we mine patterns, while also inferring others. We also present a notation for representing patterns.

3.4.1 Scoping

To cope with the size of the Linux kernel and its long evolution history, we narrow our investigation to the coevolution induced by the addition and removal of features names. In particular, we observe how developers coevolve Makefiles and C source code when adding or removing features from the x86 variability model, mining emergent variability-coevolution patterns. The choice of the x86 hardware architecture follows from the fact that its variability model is representative of how the Linux kernel variability model evolves as a whole, as both display the same growth pattern [91].

To allow mining coevolution patterns, we first collect the entire set of added and removed features from the variability model namespace of the x86 architecture. This is performed by calculating the feature set difference of the x86 variability models of consecutive stable kernel releases.[6]

To list the features in the variability model of a given release, we adapt the Kconfig infrastructure shipped in the Linux kernel source code. Such infrastructure, however, varies across different releases, as the the Kconfig language evolves along the way. We note, however, a stability period between releases v2.6.26 and v3.3. To facilitate implementation, we take such release range as our target population, allowing us to have a single Kconfig parser. Only recently we have been able to parse any Kconfig file; the latest version of our infrastructure can now process all Kconfig files from release v2.6.12 onwards. Section 4.2.3 provides further details.

3.4.2 Data Collection

From the definition of the target population, we take the union of all added features in the v2.6.26—v3.3 release range as the *additions population*; likewise, the *removals population* is given by the union of all removed feature names in the given release interval. Between releases v2.6.26 and v3.3, the number of additions (4,112) is four times bigger than the amount of removals (1,002). These numbers are consistent with other works [44, 91] showing that feature additions in the Linux kernel exceed feature removals. Given the feature names in each population, we select two random samples: one comprising 6.5 % (268) of feature additions, and another with 13 % (132) of feature removals. Such sizes follow from the effort we can afford when manually analyzing each sample, a cost and labour intensive activity.

[6] Unstable releases are suffixed with *-rc* (e.g., v2.6.32-rc1), whereas stable ones are not.

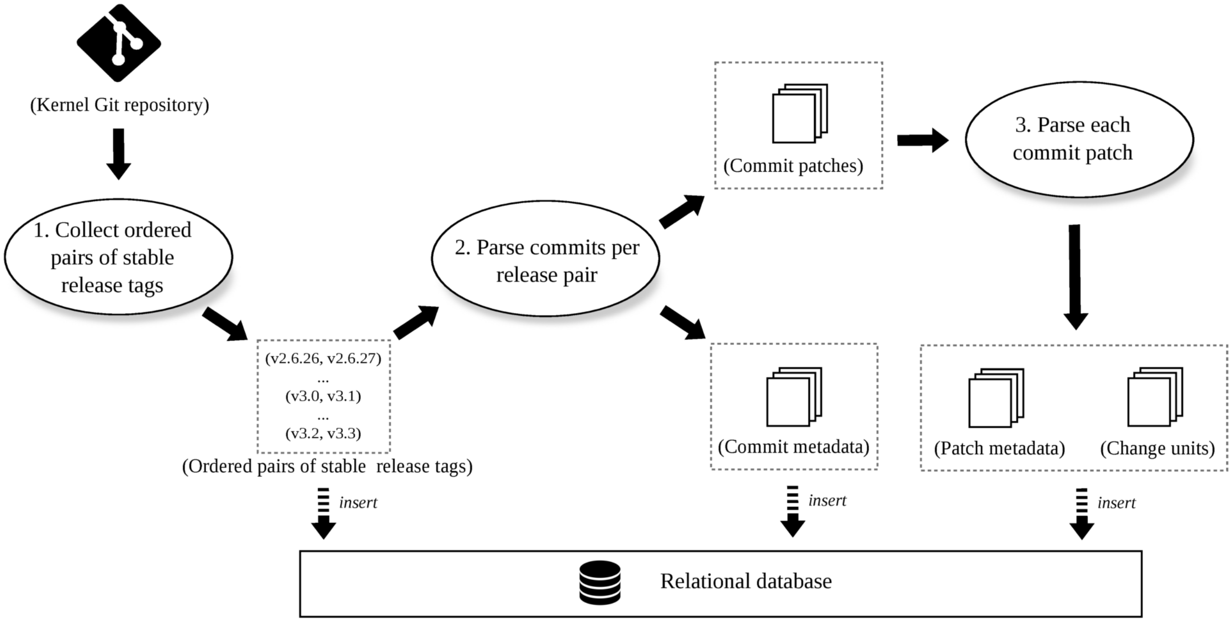


Figure 3.3: Database creation process

An entry in the additions sample is a pair of the form (f, r_{i+1}), where r_{i+1} adds a feature name f that does not exist in the previous stable release r_i. An entry (f, r_{i+1}) in the removals sample means that release r_{i+1} no longer contains f, although r_i does. A feature f in either of the entries is referred as *primary feature*—a primary object in our investigation.

To obtain the patch adding or removing a primary feature, we must first locate its corresponding commit, referred to as *primary commit*. To that end, we use a custom-made tool to create a relational database from the Linux kernel Git commit history. Figure 3.3 depicts the process to populate our database. First (step 1), we enumerate all stable releases saved in the commit history, storing them as ordered sequential release pairs of the form (r_i, r_{i+1}). In step 2, we parse all commits between the releases of each release pair,[7] storing the commit author name and email, the commit message, the commit hash, etc. Next (step 3), we parse the patch of each commit from step 2, saving associated metadata (e.g., the name of the changed file, whether the file is new, removed, or renamed, etc) and any *feature change units*. A feature change unit is a change that adds or removes a feature name in a Kconfig file. For each change unit in our database, we also store the

[7] If r_i and r_{i+1} are two consecutive releases, `git log --no-merges` r_i..r_{i+1} lists all commits between r_i and r_{i+1}, excluding those that result from merging branches.

Table 3.2: Commit statistics

Release range: v2.6.26 to v3.3	
Nbr. of commits	176,449
Nbr. of commits changing Kconfig files	10,205
Nbr. of commits adding/removing features	5,704
Nbr. of distinct primary commits in our two samples	359

name of the feature it adds or removes. In all steps, we link data accordingly: each patch metadata and change unit record links to a corresponding commit record, which in turn, links to a specific release pair.

With the database in place, retrieving the primary commit of a primary feature becomes a simple matter of issuing an SQL-query: if a feature f is in the feature set difference of r_{i+1} and r_i, then there exists a primary commit with a change unit adding f. Such commit, in turn, associates with the release pair (r_i, r_{i+1}). Likewise, if f is in the difference of the feature sets of r_i and r_{i+1}, then there exists a primary commit with a change unit removing f. As before, the retrieved commit associates with the release pair (r_i, r_{i+1}).

In the database, a primary commit associates with one or more primary features. Primary features may also have two or more associated primary commits, but we restrict it to be exactly one to facilitate analysis. Taking f as primary feature, we find the following cases that lead to two or more primary commits in the target population:

T1 In addition to x86, f is also in the namespace of other architectures (e.g., sparc, powerpc, etc), being declared in Kconfig files specific to such CPUs. Therefore, adding or removing f happens in all architectures that support it, having different commits for different architectures (generally, one per architecture type). When facing multiple commits targeting different architectures, we select the one concerning x86 (our scope of analysis).

T2 A commit adds f, another removes it (e.g., by reverting the first change), and a third adds f again. Likewise, a commit may remove f, a second add it, and a third remove it again. In both cases, we take the primary commit to be the last one in the series, regardless of which sample f originates from.

T3 A commit adds f to a child Kconfig file. A parent Kconfig file then includes the child one by means of an include instruction. Later, another commit replaces the inclusion instruction by the declaration of f itself (another addition). When having two possible primary commits as described, we take the first one, as the second does not affect the namespace; rather, it only relocates f's declaration.

T4 A commit first adds f, followed by another commit creating an additional configuration option f (in Kconfig, it is possible for a feature to be declared twice). Similar to the previous case, the namespace is not changed. As before, we take the first commit as the primary one.

T5 Due to the distributive nature of the kernel development, patches may be submitted more than once. Consequently, different commits may have equal patches. For example, a patch submitted to the kernel mailing list may be accepted by a developer, who commits it to his local copy of the kernel repository. Prior to pushing it to the remote site, the developer pulls from the remote copy to retrieve any updates. Meanwhile, another developer also accepts the change, and prior to pushing it, he also performs a pull to fetch any remote updates. Note that both pulls do not retrieve the accepted change, as it has not been pushed by either developer. Then, the second developer pushes his changes, followed by the push of the first developer. As a result, the remote repository now has two exact patches, each with a different commit hash.

T6 There are two or more commits removing f, with each commit holding a different patch. As an example, consider the case where a commit copies f to a new location in the repository, resulting in a duplicate declaration. A new feature is then introduced, generalizing the capabilities of f. As the generalized feature supersedes the original and the copied features, both must be removed. The developer, however, separates such removal in two commits. The first one contains the removal of the original feature; the second commit contains the patch adding the generalized feature, together with the removal of f's copy. When facing multiple removals, we take the latest one. Likewise, it also happens that two different commits add a feature f in distinct ways. For example, a developer sends to the mailing list a patch adding f, which eventually gets accepted. Later to his first submission, the same developer re-submits the patch with further enhancements.

In the kernel repository, feature additions and removals that link to multiple primary commits are infrequent. In our samples, we only find three additions (two cases of T1 and one case of T5) and two commits removing the same primary feature (T5).

Forcing a primary feature to have exactly one primary commit means that we work with the same number of primary commits as our sample sizes; hence, there are 268 and

132 primary commits relative to added and removed features, respectively. Since some primary commits concern more than one primary feature, the number of distinct primary commits (359) is lower than the sum of the two sample sizes. Table 3.2 puts these statistics into context.[8] The number of distinct commits in our two samples equals to 6 % of all commits that either add or remove features in the v2.6.26–v3.3 release range. Commits adding or removing features in turn, is a subset of the commits changing Kconfig files; the former represents 56 % of all the commits in the latter set. Commits that necessarily change Kconfig files are a particular piece of the kernel evolution history, accounting for approximately 6 % of all commits in the given release range. Overall, the two samples cover 0.2 % of all commits between releases v2.6.26 and v3.3.

Knowing all primary commits, we proceed to mine variability-coevolution patterns.

3.4.3 Mining Process

We apply two major steps when mining the coevolution pattern best explaining the addition or removal of a primary feature: *Commit Window Retrieval* and *Commit Window Categorization and Clustering*. We describe each step in the following.

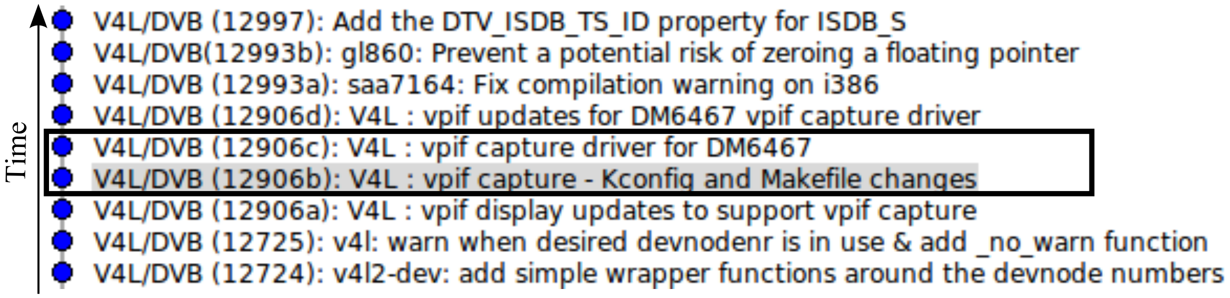
Commit Window Retrieval. A primary commit only guarantees to retrieve changes in the variability model. Thus, the induced coevolution of feature additions and removals may require inspecting other commits to capture related changes outside the variability model. For that matter, we rely on a *commit window* to expand the search scope for changes in related artifacts.

A commit window is a sequence of commits that in addition to the primary commit, may include commits preceding or following the primary one. To exemplify a commit window, consider the addition of the `CAPTURE_DAVINCI_DM64X_EVM` feature.[9] As shown in Figure 3.4, the primary commit (highlighted in gray and labelled as 12906b) is part of a sequence of commits changing the V4L and DVB subsystems,[10] as stated in the commit log messages. The primary commit patch is shown in Figure 3.5. A patch is a textual diff recording added (prefixed with "+") and removed lines (prefixed with "-"). Lines without prefix provide context to ease understanding. In the example, the primary commit adds a Kconfig entry (Figure 3.5, lines 8–11) and a new build rule to compile `vpif_capture.c` (line

[8] The numbers in the table do not account for commits that merge branches.

[9] `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=89803d83`

[10] V4L/DVB: Video for Linux/Digital Video Broadcasting



```
V4L/DVB (12997): Add the DTV_ISDB_TS_ID property for ISDB_S
V4L/DVB(12993b): gl860: Prevent a potential risk of zeroing a floating pointer
V4L/DVB (12993a): saa7164: Fix compilation warning on i386
V4L/DVB (12906d): V4L : vpif updates for DM6467 vpif capture driver
V4L/DVB (12906c): V4L : vpif capture driver for DM6467
V4L/DVB (12906b): V4L : vpif capture - Kconfig and Makefile changes
V4L/DVB (12906a): V4L : vpif display updates to support vpif capture
V4L/DVB (12725): v4l: warn when desired devnodenr is in use & add _no_warn function
V4L/DVB (12724): v4l2-dev: add simple wrapper functions around the devnode numbers
```

Figure 3.4: Commit window example

15). Since the feature's compilation unit is not added in the primary commit, we set to expand it to the point where such an addition occurs, if at all. The commit following the primary one (label 12906c) adds `vpif_capture.c`; thus, we expand the commit window to include it. The window, however, does not include commit 12906d, as it simply updates the code of the previously added compilation unit (e.g., changing a `static` variable to be `extern`). The information stemming from the update is irrelevant, as it does not add new knowledge on the structure of the change in place. Thus, the resulting window comprises commits 12906b and 12906c, as shown by the rectangle in Figure 3.4.

Strictly, the boundaries of a commit window are only limited by the total number of commits in the evolution history. Furthermore, selecting which commits should be part of a commit window is ultimately a subjective process.

To mitigate subjectivity, we expand a commit window by including commits that have the same commit message *label* as the primary one, and that necessarily precede or follow it. For example, in Figure 3.4, all commits changing the V4L and DVB subsystems are labelled with "VL4/DVB", and thus, are potential candidates to be included in the resulting commit window. Following sequences of commits sharing the same label, however, does not necessarily retrieve commits related to the primary feature under investigation (e.g., it may include commits relative to a sibling feature of the primary feature, both belonging to the same part of the kernel). To avoid large windows with unrelated commits, we define four main expansion rules for including commits sharing the same label of a primary commit:

E1 Include commits that add/remove compilation units known to be mapped to the primary feature.

E2 Include commits whose changes affect files mapped to the primary feature, provided such changes add relevant information w.r.t. the structure of the change in place.

```
drivers/media/video/Kconfig

config DISPLAY_DAVINCI_DM646X_EVM
   help
-     Support for DaVinci based display device.
+     Support for DM6467 based display device.

+config CAPTURE_DAVINCI_DM646X_EVM
+   tristate "DM646x EVM Video Capture"
+   depends on VIDEO_DEV && MACH_DAVINCI_DM6467_EVM
+   ...

drivers/media/video/davinci/Makefile

+obj-$(CONFIG_CAPTURE_DAVINCI_DM646X_EVM) += vpif_capture.o
```

Figure 3.5: Patch adding the Davinci DM646x EVM driver (primary commit)

E3 Include commits whose changes add/remove compile-time variation points that reference the primary feature.

E4 Include commits that modify the declaration of the primary feature in the variability model.

Initially, we apply these rules to expand the commit windows of features in the additions sample only. Starting with the primary commit, we allow a commit window to grow as large as needed, but stop its expansion whenever we meet one of the following boundary conditions: (a) the commits in the current window provide enough context to understand the changes related to the primary feature. For instance, to understand the addition of CAPTURE_DAVINCI_DM64X_EVM we are only required to extend the commit window up to the point where vpif_capture.c is added, but not further; (b) we reach a large sequence of commits that do not share the same label as the primary commit. In this case, we consider the change of the primary feature to be over. The rationale of first expanding commit windows of features in the addition sample follows from our assumption that commit windows of features in the removals sample are likely to be smaller; if true, the maximum commit window size in the additions sample works as an upper bound for the commit window size of features in the removals sample. Our assumption relies on the fact that removing features should be done at once, in a single commit, as developers should not leave dead code behind, nor break the system compilation. Additions, on the other hand, may span more than a single commit, as adding incremental chunks agrees with Git's principle *commit early, commit often.*[11]

We find that commit windows of added primary features have at most 28 commits, although in most cases it has a single one (the primary commit). For defining the commit windows of removed features, we conservatively increase the 28-limit to 40, as an attempt to avoid loosing any commits. Upon the validity of our previously stated assumption, however, commit windows in the removals sample should never reach such a limit. In fact, they do not. After applying the four expansion rules, while respecting boundary conditions and a maximum commit window size of 40, we find that almost every commit window in the removals sample has size one. Few commit windows (6) have more than one commit; three commit windows have two commits, while the remaining three have four, five, and 14 commits, respectively. Overall, commit windows are small in both samples (see Figure 3.6). In the additions sample, an average commit window has 1.9 commits, whereas in the removals sample, the mean is 1.2.[12] In both samples, the median commit window size is one. Therefore, in the case of the Linux kernel, determining the size of commit windows is not difficult, as a typical commit window contains only the primary commit of the feature under investigation.

Commit Window Categorization and Clustering. Within each retrieved commit window, we move to manually inspect all the changes it contains, initially categorizing it as *addition*, *removal*, *split*, *merge*, or *rename* of the primary feature. Windows with the same category are then clustered together. Note that classifying commit windows require us to ignore changes unrelated to the primary feature. Lines 5–6 in Figure 3.5 show a simple example. More complex unrelated changes occur when a commit window contains patches that, in addition to the primary feature, also add or remove other features. In this case, we set focus on patch parts that explicitly associate with the primary feature (e.g., a code fragment guarded by an *ifdef* condition referring to primary feature, a C file whose compilation depends on selecting the primary feature, etc), or that relate to it as a consequence of the change under investigation (e.g., a new *ifdef* condition is created for a new feature, which in turn, results from the rename of the primary one).

The relevant changes inside each window are then taken as a whole, which we capture

[11] `http://sethrobertson.github.io/GitBestPractices`

[12] These values are calculated as follows: for the additions sample, we sum the size of all its commit windows (502), and divide the result by the number of added primary features (268). Likewise, in the removals case, we sum the total number of commits in the commit windows in the corresponding sample (155), and divide it by the number of removed primary features (132).

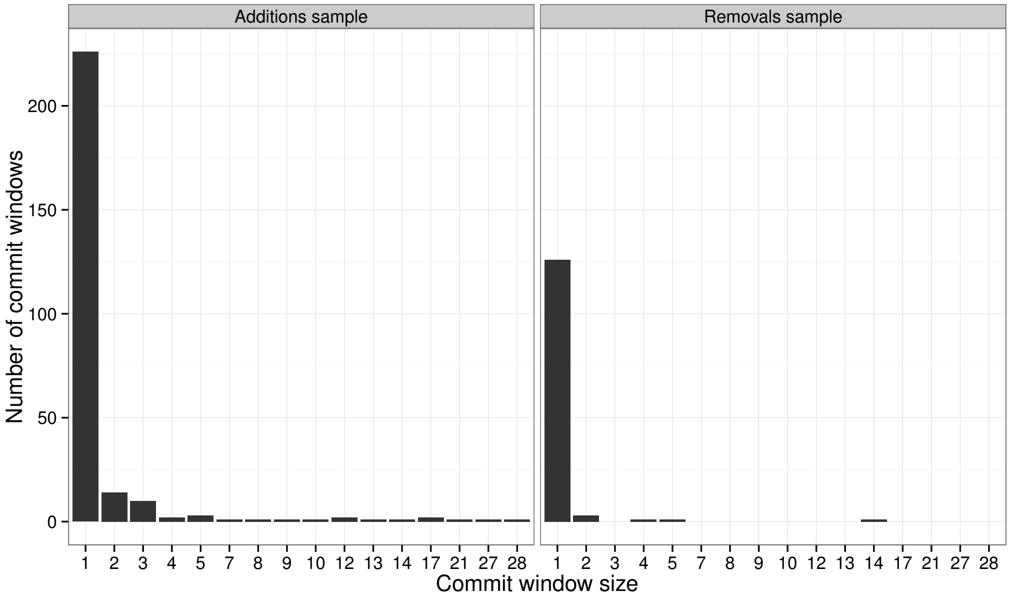


Figure 3.6: Commit window sizes

as a *before-state* (what exists before the change) and *after-state* (what exists after the change). At this stage, we create specialized subcategories to represent commit windows with similar before and after states, capturing common characteristics of how developers modify primary features and their cross-tree constraints. Such characteristics include, but are not limited to:

a) Visibility: Feature is promptable in the configurator or not.

b) Type: Whether the feature is a *switch* (i.e., Boolean/tristate) or a *value-based* feature (int/string) [21].

c) Computed defaults.

d) Mandatory.

e) Whether the feature causes the addition of compile-time variation points, and in which spaces.

f) Whether the feature contains associated compilation units.

g) Whether the feature adds compilation flags.

We then re-cluster results accordingly and discard clusters with less than three instances, or clusters respecting such threshold, but with less than three distinct contributors. These two key criteria conform to the *Rule of Three*, a widely adopted operationalization for pattern identification [57, 85, 103]. Keeping only the clusters whose changes have been applied by at least three distinct developers provides three or more distinct sources of evidence, preventing bias towards an specific developer coding style. Moreover, our operationalization makes pattern identification independent of a given sample size. To differentiate among contributors, we use the author's name and email, as recorded in the metadata of each commit. Once we cannot further subcategorize clusters, we set to extract a common structure among the before and after states of all commit windows in each obtained cluster. Each resulting structure defines a pattern.

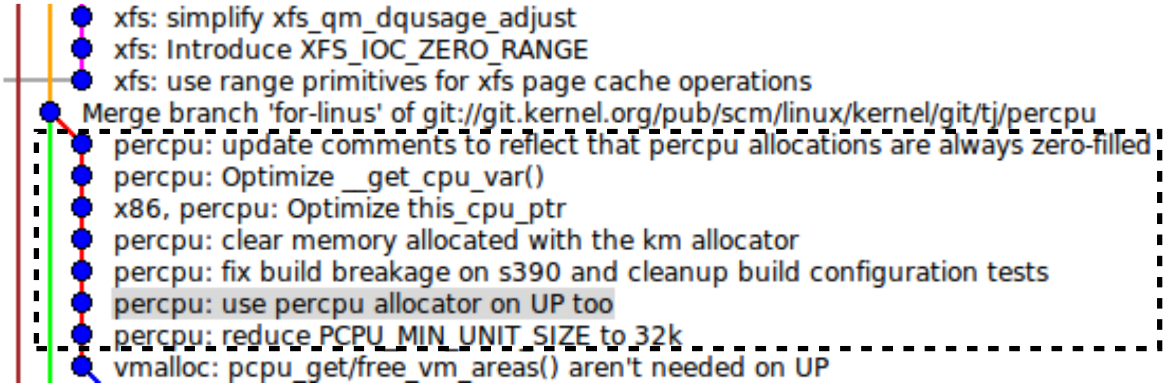
In total, we examine 657 commits in all commit windows, where 502 relate to features in the additions sample and the remaining 155 to features in the removals sample. In some cases, however, we cannot derive a full understanding of the changes relative to a primary feature. As an example, consider the addition of the NEED_PER_CPU_KM feature to the kernel memory management subsystem.[13] Figures 3.7 and 3.8 show the addition's primary commit (highlighted in gray) and its corresponding patch fragment, respectively. Since the newly added feature is computed (it is assigned its default value upon the validity of its depend on clause) and not visible, users cannot configure it directly. Thus, the identifier NEED_PER_CPU_KM must be referred elsewhere for the feature to be useful. However, expanding the initial commit window to include commits sharing the same label of the primary commit (shown as a dashed rectangle in Figure 3.7) does not show any reference addition. Hence, as we cannot fully understand the change in place, we exclude NEED_PER_CPU_KM from further analysis. Overall, when facing doubt, we exclude 4.5 % (12) of the features in the additions sample; in the removals sample, the exclusion rate is 8.3 % (11).

We execute our mining process in two rounds, having two participants performing the mining: P_1 and P_2. Participant P_1, the author of this thesis, is a proficient Linux user with past experience in the analysis of feature evolution in the Linux kernel [107]; participant P_2 has expertise in variability model evolution [64].

3.4.4 Review Process

To assure the quality of our results, after each mining round, we perform extensive reviews to mitigate possible human errors. In addition to P_1 and P_2, two other participants aid

[13] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=bbddff05



```
xfs: simplify xfs_qm_dqusage_adjust
xfs: Introduce XFS_IOC_ZERO_RANGE
xfs: use range primitives for xfs page cache operations
Merge branch 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/tj/percpu
percpu: update comments to reflect that percpu allocations are always zero-filled
percpu: Optimize __get_cpu_var()
x86, percpu: Optimize this_cpu_ptr
percpu: clear memory allocated with the km allocator
percpu: fix build breakage on s390 and cleanup build configuration tests
percpu: use percpu allocator on UP too
percpu: reduce PCPU_MIN_UNIT_SIZE to 32k
vmalloc: pcpu_get/free_vm_areas() aren't needed on UP
```

Figure 3.7: Commits changing the kernel memory-based chunk allocation

reviewing: P_3 and P_4. Participant P_3 has experience in investigating variability evolution in small-sized real-world software product lines [104]; P_4, in turn, has an extensive background in Linux kernel variability model evolution [44].

During review, at least two participants review the initial mining results, questioning whatever points they judge necessary. If reviewers raise awareness for a possible error, a *consensus phase* follows. In that case, the participant responsible for the mining and the ones performing the review defend they points, and eventually reach a final agreement. Table 3.3 summarizes the role of each participant; the assignment of a role to a participant follows directly from a participant's availability at the time of this study.

In the first mining round, we analyze 206 additions and 101 removals cases of our collected sample. At this point, P_1 and P_2 are the ones mining coevolution patterns (indicated with an 'M' in the corresponding table cell). P_1 and P_2 perform their analysis while also consulting each other to clarify and discuss arising issues. After completing their analysis, both participants set to review each other's work (shown with an 'R' in the corresponding table cell). To strengthen our findings, participants P_3 and P_4 perform two additional reviews. As before, potential inaccuracies are spotted and settled based on argumentation.

In the second round, we analyze the remaining cases in our sample, namely 62 feature name additions and 31 removals. At this step, only P_1 mines patterns, followed by a review of P_3 and P_4. As before, in the case of inconsistencies, a consensus follows.

In both rounds, the reviews of P_3 and P_4 evidence few inaccuracies. P_3's review suggests five errors; three are minor comments, whereas the remaining two do raise suspicion that two commits might not be instances of the pattern they have been initially assigned to.

```
mm/Kconfig

+config NEED_PER_CPU_KM
+        depends on !SMP
+        bool
+        default y

mm/Makefile

-ifdef CONFIG_SMP
-obj-y += percpu.o
-else
-obj-y += percpu_up.o
-endif

mm/percpu-km.c

-#ifdef CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK
+#if defined(CONFIG_SMP) && \
  defined(CONFIG_NEED_PER_CPU_PAGE_FIRST_CHUNK)
 #error "contiguous percpu allocation is incompatible..."
 #endif

```

Figure 3.8: Patch adding NEED_PER_CPU_KM

Consensus, however, does not confirm such conclusion, and P_3 is convinced otherwise. The review of P_4, in turn, suggests a 7 % and 14 % inconsistency rate in the analysis of the primary features in the additions and removals samples, respectively. Consensus confirms them all, leading to a correction of our initial results.

3.4.5 Pattern Inference

When mining coevolution patterns, we assure that each pattern has at least three instances in our samples and a minimum of three distinct contributors. When we cannot satisfy either or both of these two conditions, we set to infer patterns. We employ two inference rules:

I1 There exists a pattern adding a given feature, but no inverse pattern exists in the

Table 3.3: Activities performed by each participant in each sample (M: Mining patterns; R(P): Review of the patterns mined/reviewed by participant P; R(P, Q): Review of the patterns mined/reviewed by participants P and Q)

Participant	Round 1		Round 2	
	Additions (206)	Removals (101)	Additions (62)	Removals (31)
P_1	M + R(P_2)	M + R(P_2)	M	M
P_2	M + R(P_1)	M + R(P_1)	–	–
P_3	R(P_1, P_2)	R(P_1, P_2)	R(P_1)	R(P_1)
P_4	R(P_1, P_2, P_3)	R(P_1, P_2, P_3)	R(P_1, P_2, P_3)	R(P_1, P_2, P_3)

removals sample. From the fact that every added feature should be eventually removed, and that such removal can be achieved by simply following the opposite steps performed when adding the feature, we take the inverse of any addition pattern to be an inferred removal if it is not already in our catalog of patterns.

I2 There exists a pattern in the removals sample, but an inverse pattern is not reported in the additions sample. From the rationale that a feature can only be removed if it is first added, and that such addition can be achieved by following the inverse steps of its removing pattern, we take the inverse of any removal pattern to be an inferred addition if it is not already in our catalog of patterns.

These rules are not exhaustive, and other patterns can be inferred by additional rules (e.g., by composing patterns). However, we restrict inference to rules I1 and I2 on the basis that the existence of their inferred patterns is suggested by the reported inverse non-inferred patterns. When reporting our patterns, we clearly distinguish which are inferred and which are not.

3.4.6 Pattern Representation

Each pattern we mine captures a common structure among the before and after states of all commit windows in a given cluster resulting from our categorization and clustering step. To facilitate understanding, we represent these structures graphically, devising a special notation. As an example, consider a particular instance of a merge pattern operating on the two framebuffer-related features presented in Section 3.1: FB_IMAC and FB_EFI. Both features are children of FB. Due to their similarity, developers decide to merge the two features, adding the capabilities of FB_IMAC into the implementation of FB_EFI. To avoid capability redundancy, developers remove FB_IMAC from the variability model, mapping and implementation.[14] Figure 3.9 captures such a pattern. As the figure shows, a pattern denotes a transition from a before-state to an after-state, which results from the application of the prescribed change. The transition is represented by an arrow (shown in the middle); the before-state is on the left of the arrow; the after-state follows it. In each state, the pattern captures key characteristics in the variability model, build files, and source code.

We express the variability model in a FODA-based notation, together with the set of the existing cross-tree constraints (i.e., CTC). Since FODA [74] is a simple, intuitive and widespread notation praised by both researchers and practitioners [20], we can abstract over many specific details of Kconfig, while reaching a larger audience. In the before-state of Figure 3.9, two optional sibling features exist: f_1 (matches FB_IMAC) and f_2 (matches FB_EFI). To explicitly report that these features are visible (promptable) during configuration, we use a corresponding attribute (shown inside square brackets).

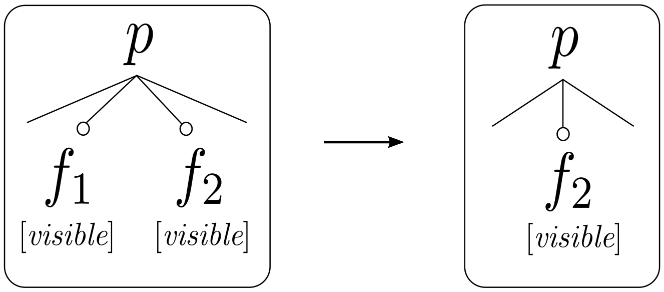
We capture the mapping M as a sequence of build rules defined by the following syntax:

$$\begin{aligned} M &::= \langle R^{+} \rangle \\ R &::= (E, R, R) \mid \text{object files}^{+} \mid \text{directory}^{+} \mid \text{compilation flag}^{+} \mid \epsilon \end{aligned}$$

In a conditional build rule (e, r_1, r_2), e is an expression E over feature names; r_1 is another build rule R executed in case e evaluates to true; r_2 is an alternative build rule if e does not hold. The shorthand form (e, r_1) is used when r_2 is empty. Unconditional rules are either a sequence of object files, a non-empty list of directories, one or more compilation flags, or an empty rule. The pattern in Figure 3.9 shows two build rules: $(f_1, f_1.o)$ and $(f_2, f_2.o)$, stating that the presence of f_1 and f_2 triggers the compilation and linkage of their corresponding compilation units (imacfb.c and efifb.c in the example). For simplicity, this representation does not distinguish dynamically loadable modules from objects to be statically linked against the kernel.

Similarly to the mapping space, we capture the implementation (I) as a sequence of code block triples (e, c_1, c_2), where e is a macro-based expression over feature names and c_1 and c_2 are themselves code block triples. As before, simplifications are possible: c denotes an unconditional code block and (e, c_1) is a conditionally compiled code block without an alternative. In case an entire compilation unit implements a feature, we draw a square in the code space (e.g., matching imacfb.c and efifb.c, respectively).

[14] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=7c08c9ae



CTC

$$M = \langle \dots (f_1, f_1.o) \dots \dots (f_2, f_2.o) \dots \rangle$$

$$I = \langle \dots \boxed{f_1} \dots \boxed{f_2} \dots \rangle$$

$CTC\text{'} = CTC_{[f_2 \backslash f_1]}$

$$M\text{'} = \langle \dots (f_2, f_2.o) \dots \rangle$$

$$I\text{'} = \langle \dots \boxed{f_2} \dots \rangle$$

$$\textit{with } f_2 > f_1$$

Figure 3.9: Definition of Merge Visible Optional Feature into Sibling (MVOFS)

In all spaces, we use ellipses ("...") to ignore unrelated elements that do not affect the features under analysis.

The after-state of the merge pattern in Figure 3.9 removes f_1 from all three spaces (removal is generally denoted by omitting elements previously shown in the before-state). The set of cross-tree constraints is then rewritten (CTC') such that every reference to f_1 becomes a reference to f_2. Besides referential integrity, such rewrite guarantees that all constraints imposed by f_1 are now imposed by f_2 as well (no constraint is lost). Furthermore, the compilation unit of f_2 continues to support the capabilities of f_1, plus its own, which we denote as $f_2 > f_1$.

3.5 Results

We find 23 patterns from the analysis of our two samples; of those, 19 are non-inferred, while four are inferred. Table 3.4 lists all the patterns and their usage frequency. Non-inferred patterns abstract over the changes in clusters with at least three commit windows; moreover, the total number of primary commit contributors in the commit windows inside the cluster of a non-inferred pattern is always greater or equal to three. In the table, non-inferred patterns are identified by a *No*-value in the *Inferred* column. In contrast, inferred patterns denote the cases where we find a non-inferred pattern, but we lack evidence of

Table 3.4: Collected patterns and their frequency

	Feature additions sample			Feature removals sample		
	Pattern	Frequency	Inferred?	Pattern	Frequency	Inferred?
1	AVOMF	124	No	RVOMF	22	No
2	AVOGMF	11	No	RVOGMF	12	No
3	AVONMF	32	No	RVONMF	10	No
4	AVOCFF	4	No	RVOCFF	0	Yes
5	AVONMCFF	3	No	RVONMCFF	0	Yes
6	AVOAF	2	Yes	RVOAF	6	No
7	AVMVF	3	No	RVMVF	3	No
8	AIMF	12	No	RIMF	3	No
9	ACINMF	2	Yes	RCINMF	3	No
10	FCUTVOF	10	No	MVOFNO	3	No
11	FCFTVOF	4	No	MVOFS	3	No
12	RNM	11	No	RNM	18	No
	Total	218		Total	83	
	Sample (%)	81%		Sample (%)	63%	

an inverse pattern in the complementary sample. The inverse pattern, however, is likely to exist in the Linux kernel evolution. Inferred patterns are identified by a *Yes*-value in the *Inferred* column. To illustrate inference, consider the case where a developer adds a visible (promptable) feature controlling a specific compilation flag, as prescribed by the AVOCFF pattern (row 4). Following such addition, it should also be the case that the same feature should be later removed in the course of evolution, although we may not see it as part of our removals sample. In the later case, we infer the pattern.

We discuss all the patterns in the following, except for rename (RNM), which we omit due to its simplicity.[15] We also present a brief discussion over changes that do not lead to patterns.

[15] Basically, a rename just updates all references to a given feature name f to a new name f_N in all spaces where f appears. Note that renaming does not cause any behavioural change, nor does it change the set of cross-tree constraints.

3.5.1 Non-Inferred Feature Addition Patterns

We find nine non-inferred patterns in the additions sample. They concern two specific situations: (i) adding a new feature from completely new elements (AVOMF, AVOGMF, AVONMF, AVOCFF, AVONMCFF, AVMVF, and AIMF); (ii) adding a new feature created out of existing elements—featurization (FCUTVOF and FCFTVOF). Altogether, they capture how the mapping and implementation change upon adding a new feature in the variability model namespace.

Add Visible Optional Modular Feature (AVOMF)

A visible and optional modular feature increases the user configuration space by providing a functionality unit that can be optionally present in the resulting kernel. *Modularity*, in this case, assures that some of the capabilities of the new feature have their own compilation unit(s).

As shown in Figure 3.10, the pattern adds a new optional and visible feature f in the variability model, along with its associated cross-tree constraints (CTC_f). A build rule then relates the feature presence to its compilation units, whose files are added to the implementation space. The addition of CAPTURE_DAVINCI_DM646X_EVM, previously discussed in Section 3.4, is an instance of this pattern.

Most primary features in the additions sample (46 %) fit into this pattern. To verify where the instances of this pattern add features to, we map the primary features of AVOMF to each kernel subsystem. According to Corbet et al. [33], there are seven subsystems in the kernel: arch, core, driver, firmware, fs, misc, and net. Greg Kroah-Hartman, the main developer of the Linux kernel stable branch,[16] provides a mapping between files in the code base of the Linux kernel and the subsystems reported by Corbet et al.[17] We take Hartman's mapping to be expert knowledge, reusing it without modifications. The mapping between the kernel source code tree to its associated subsystems is summarized in Table 3.6; a bullet in a given cell indicates that at least one file in a given folder (row) maps to the corresponding subsystem (column). By applying Hartman's mapping to each file in the Linux kernel source code tree, we take the subsystem of a feature to be the same of its enclosing Kconfig file. Once we associate each feature with a single subsystem, we count the number of pattern instances adding primary features to each kernel subsystem (see Table 3.7). In the case of the AVOMF pattern, its instances add features to the following subsystems:

[16] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS

[17] https://raw.github.com/gregkh/kernel-history/master/scripts/genstat.pl

Table 3.5: Description of the top-level folders of the Linux kernel source code tree (based on [23, 93, 140])

Folder	Description
arch	Architecture (CPU) dependent code
block	I/O scheduling algorithms for block devices
crypto	Cryptography-related algorithms
Documentation	Brief descriptions of each part of the implemented kernel
drivers	Device drivers of different devices classes
firmware	Device firmware needed by certain drivers
fs	Defines the virtual file system abstraction, along with concrete file systems
include	Kernel header files
init	Kernel boot and initialization
ipc	Support for inter-process communication (IPC)
kernel	The main kernel code (architecture independent)
lib	Library (helper) routines
mm	Memory management support
net	Implementation of network protocols
samples	Different code examples
scripts	Different scripts for building the kernel
security	The security framework of the kernel, known as LSM (Linux Security Modules), supporting different access control models [148]
sound	The Linux sound subsystem and related device drivers
usr	Implementation of initramfs, a RAM-based root filesystem required by the startup process; the first process (init) runs on top of it
tools	Tools for building the kernel and helper programs useful for kernel developers
virt	Virtualization support

- *Device driver* (driver): 93.6 % of the instances in this pattern concern the addition of device drivers (i.e., features that are “plugged-in” to the kernel to support different hardware). This high frequency is in line with previous work [53, 61, 70, 91] stating that Linux kernel evolution is mainly driven by the addition of new device driver-related features.

Table 3.6: Mapping of the kernel's top-level directories and its subsystems

Source code folder	Subsystems						
	arch	core	driver	firmware	fs	misc	net
arch	●						
block		●					
crypto			●				
Documentation						●	
drivers			●				
firmware				●			
fs					●		
include	●	●	●			●	●
init		●					
ipc		●					
kernel		●					
lib		●					
mm		●					
net							●
samples						●	
scripts						●	
security			●				
sound			●				
usr						●	
tools						●	
virt		●					

- *Architecture specific code* (arch): 2.4 % of the instances of this pattern add modules that are specific to a given hardware architecture. For example, one instance adds support for injecting machine checks when testing the kernel for the x86 architecture. Such functionality is used by kernel developers when performing quality assurance.

- *File system* (fs): 1.6 % of AVOMF features relate to adding file system functionalities, including support for integrity tests and compression support (LZO) for the Squash

Table 3.7: Frequency of non-inferred patterns per subsystem (additions sample)

Pattern	Distribution across systems						
	arch	core	driver	firmware	fs	misc	net
AVOMF	3	1	116	0	2	0	2
AVOGMF	0	0	9	0	0	0	2
AVONMF	7	2	19	0	3	0	1
AVOCFF	0	2	2	0	0	0	0
AVONMCFF	2	0	1	0	0	0	0
AVMVF	0	1	2	0	0	0	0
AIMF	0	0	11	0	0	0	1
FCUTVOF	0	0	10	0	0	0	0
FCFTVOF	0	1	3	0	0	0	0
RNM	0	0	10	0	1	0	0

file system.[18]

- *Network* (net): 1.6 % of the features of this pattern provide network capabilities, such as extending a network protocol with a new functionality. One specific case adds probing support for incoming SCTP packets.

- *Core functionality* (core): 0.8 % of the features of this pattern add a module to the core subsystem. An example is self-test for 64-bit atomic instructions.

Instances of this pattern are either tristate (91 %) or Boolean. The dominance of tristate features follows a trend in most of the patterns related to modular features, evidencing a strong relationship between the two. This association is unlikely to be accidental, as modular tristate features provide flexibility to cover different requirements and configuration purposes. For example, in embedded platforms where hardware can be anticipated, tristate features can be statically linked against the final kernel; in other situations, when hardware configuration varies, tristate features can be compiled as LKMs and loaded as needed.

It is worth noting that a modular feature can still have extensions elsewhere. Such cases, however, are not frequent. After checking out the last commit in the commit window of each AVOMF instance and counting the number of extensions associated with each primary

[18] http://squashfs.sourceforge.net/

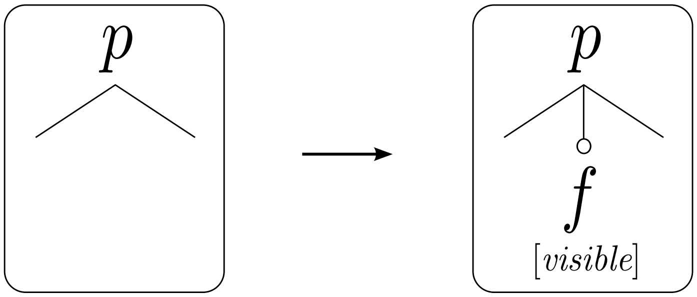
feature in C header and implementation files, we find only 11 features introducing *ifdefs*; in total, we identify 14 extensions. All extensions are *disciplined* [89], i.e., their *ifdefs* annotate code fragments matching entire syntactic units in the host programming language (C). Similar to previous studies [77, 87], we also assess the level at which developers write their extensions. Our analyses show that 64.3 % (9) of extensions add code at the global level (e.g., declaring a new macro, variable, function, structure, etc), while 28.6 % (4) add code at the function level (e.g., by adding statements inside a function);[19] there is a single case (7.1 %) where a feature adds an extension at the type level (e.g., annotating the declaration of a field within a struct type). The number of extensions per feature is also low: of all 11 AVOMF features with extensions, eight have $SD = 1$; the remaining three have $SD = 2$. Thus, only 2 % of AVOMF cases concern scattered features i.e., features with $SD \geq 2$. Two of the three scattered features are *locally scattered*. Stated otherwise, they are scattered across the same subsystem as their containing one (driver and fs, respectively). The other scattered feature, in contrast, is *globally scattered*—it has at least one extension outside its containing subsystem (driver). Specifically, we find it to have two extensions in the arch subsystem.

AVOMF usage suggests that most features in the Linux kernel are completely modular. When not fully modular, extensions are limited, and so is their scattering degree. The lack of AVOMF features in fs that are globally scattered outside fs is likely to follow from the *Virtual File System* abstraction layer, whose role is to shield other parts of the kernel against new file systems and their supported feature set. Analogously, device drivers tend to be fully modular; most tend to be simply "plugged-in" to the system, registering themselves as handlers to specific events (e.g., hardware interrupts) [34].

Add Visible Optional Guard Modular Feature (AVOGMF)

This pattern is a specialization of AVOMF. However, we distinguish between the two and count them separately because the structure of AVOGMF plays an important role in the compilation process. In addition to the changes imposed by AVOMF, the AVOGMF pattern requires that f acts as a compilation guard over an entire directory, controlling whether the compilation process should recursively descend to that location. As such, it contains an additional mapping rule in the parent Makefile:

[19] The function level also includes the cases where developers annotate elements at specific positions within an array initialization or fields in the case of struct instantiations.



CTC $\qquad$ $CTC' = CTC \cup CTC_f$

$M = \langle \ldots \rangle$ $\qquad$ $M' = \langle \ldots (f, f.o) \ldots \rangle$

$I = \langle \ldots \rangle$ $\qquad$ $I' = \langle \ldots \boxed{f} \ldots \rangle$

Figure 3.10: Definition of Add Visible Optional Modular Feature (AVOMF)

$$M' = \langle \ldots \overbrace{(f, f/)}^{\textit{in parent Makefile}} \ldots \underbrace{(f, f.o)}_{\textit{in child Makefile (inside f/)}} \ldots \rangle$$

This rule instructs Kbuild to enter a child directory f upon the presence of that feature. Once Kbuild enters the f folder, it processes a Makefile with the rule on how to build f itself. Note that the condition over $f.o$ in the build rule in the child Makefile is redundant. Developers, however, tend to include it to prevent others from interpreting that the compilation of $f.o$ is not subject to the presence of the f feature. The addition of the device driver supporting Realtek's© 8192 network adapter illustrates this (see Figure 3.11):[20] in the parent Makefile (top snippet in the figure), Kbuild assesses whether RTL8192SE is present. If so, it enters the rtl8192se directory and processes the child Makefile there (bottom snippet); in that case, RTL8192SE's presence enables the compilation of all objects in the rtl8192se-objs list.

This pattern comprises 4 % of all additions, and two idioms result from its usage: (a) developers create guard modular features to control the compilation of a single feature, whose implementation is given by the files in the guarded directory. This represents 82 % of the instances of this pattern, where all instances add features to the driver subsystem;

[20] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=85e09b40

```
drivers/net/wireless/rtlwifi/Makefile
```

```
+obj-$(CONFIG_RTL8192SE) += rtl8192se/
...
```

```
drivers/net/wireless/rtlwifi/rtl8192se/Makefile
```

```
+rtl8192se-objs := dm.o fw.o hw.o led.o phy.o rf.o \
+                  sw.o table.o trx.o
+
+obj-$(CONFIG_RTL8192SE) += rtl8192se.o
...
```

Figure 3.11: Example of Add Visible Optional Guard Modular Feature (AVOGMF)

(b) a guard modular feature roots a subtree in the variability model with, at least, one modular descendant feature. All modular features in the subtree reside in the f directory. All the instances of the AVOGMF pattern that relate to this idiom usage add features to the `net` subsystem.

Add Visible Optional Non-Modular Feature (AVONMF)

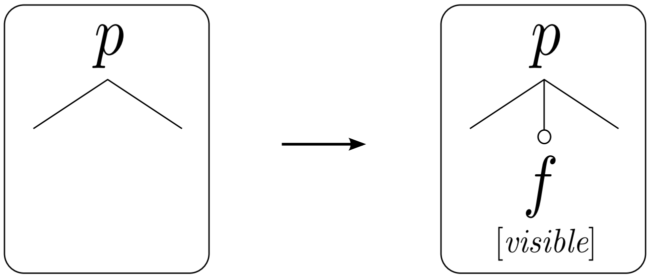
This pattern concerns the addition of features that do not fit inside a module, but rather reside in an existing host code; 12 % of the additions instances match this pattern.

As shown in Figure 3.12, this pattern adds a visible optional feature in the variability model, while not changing the mapping. The implementation changes by including code extensions (C_1) whose compilation depends on f. Alternative fragments, as given by C_2, may be absent.

This pattern serves the purpose of extending existing capabilities in code. The following patch snippet illustrates this:[21]

```
+#ifdef CONFIG_SQUASHFS_4K_DEVBLK_SIZE
+#define SQUASHFS_DEVBLK_SIZE 4096
+#else
+#define SQUASHFS_DEVBLK_SIZE 1024
+#endif
```

[21] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=7657cacf



CTC $\qquad\qquad\qquad$ $CTC' = CTC \cup CTC_f$

M $\qquad\qquad\qquad$ M

$I = \langle \dots \rangle$ $\qquad\qquad\qquad$ $I' = \langle \dots (\dots f \dots, C_1, C_2) \dots \rangle$

Figure 3.12: Definition of Add Visible Optional Non-Modular Feature (AVONMF)

If SQUASHFS_4K_DEVBLK_SIZE (matches f) is present, the block size of the Squash file system is set to four kilobytes; otherwise it is set to one kilobyte.

Following the same analysis in AVOMF, we measure the scattering degree of AVONMF features and check whether their *ifdefs* relate to a disciplined usage. We find 62 % (20) of the instances of this pattern to be scattered features ($SD > 1$), while the remaining ones (12) have $SD = 1$. In total, there are 135 extensions, with a median of two extensions per feature (min = 1, max = 36). As it occurs with AVOMF, the extensions of AVONMF features are disciplined: 40.7 % (55) occur at the global level, 39.3 % (53) appear at the function level, while 7.4 % (10) and 11.9 % (16) are at the type and block (e.g., adding a statement to a *for* loop) levels, respectively. There is a single case of undisciplined usage (0.7 %), where an *ifdef* guards a piece of a statement. This distribution of disciplined annotation percentages is similar to the one reported by Liebig et al. when investigating 40 pre-processor-based systems [87]. As we find only one case of a statement annotation, we strengthen their claim that fine-grained extensions are not frequent in practice. When investigating scattering location, we find that 60 % of scattered AVONMF features are locally scattered, with a maximum SD of eight and a median of three. A lower proportion (40 %) is globally scattered, although displaying higher scattering degrees (min = 2, median = 5.5, and max = 36). These results suggest that, in the Linux kernel, global scattered features have higher SD-values when compared to locally scattered ones.

In contrast to modular features, in 94 % of the instances of this pattern, f is a Boolean feature. Since it does not introduce any compilation unit (and thus, no build rules), it is not possible to directly control whether f should be statically present in the resulting kernel or whether it should be possible to load f dynamically at runtime. The only situation in which f is tristate is when it contains a reverse dependency to a modular tristate feature f_s; if declared as Boolean, f would cause f_s to be statically compiled into the resulting kernel, and thus, breaking the flexibility of the runtime variability related to f_s. However, visible optional non-modular tristate features are rather infrequent, as only two instances appear in our sample; one of them has no selection towards another tristate feature, and thus, provides no benefit over a Boolean declaration.

Most instances of the AVONMF pattern add features to the driver subsystem (59.3 %), although less frequently than AVOMF instances. In the remaining, 21.9 % relate to adding features in arch, 9.4 % in fs, 6.3 % in core, and 3.1 % in net.

Add Visible Optional Compilation Flag Feature (AVOCFF)

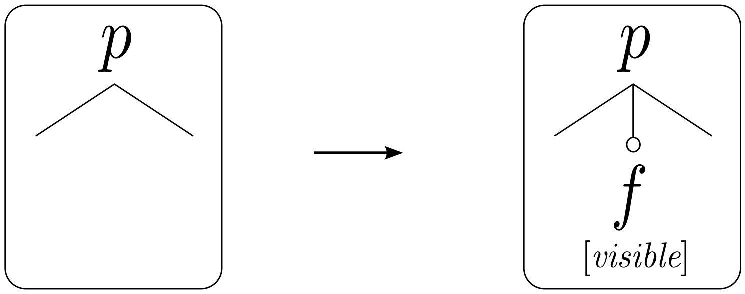
This pattern captures the addition of features that exist with the sole purpose of enabling specific compilation flags; it comprises 1 % of all additions in our sample. The purpose of the pattern is to expose a compilation flag that enables specific diagnostic capabilities, such as profiling and debug messages. Figure 3.13 shows the pattern, and an example is given in Figure 3.14.[22] Selecting USB_DWC3_VERBOSE, a new feature added to the Kconfig model (Figure 3.14, lines 3–8), defines the macro symbol VERBOSE_DEBUG, which is then referred in code, controlling whether calls to specific debug routines should be in the post-processed file. The definition of VERBOSE_DEBUG occurs by adding the compilation flag -DVERBOSE_DEBUG to the C flags list (ccflags).

In the investigated sample, half of the AVOCFF instances add features to core, while the remaining add features to driver.

Add Visible Optional Non-Modular Compilation Flag Feature (AVONMCFF)

This pattern is a composition of AVONMF and AVOCFF. It is not accounted in neither AVONMF nor AVOCFF, as the former does not change the mapping, whereas the latter does not affect the implementation. To cover both types of changes, we introduce the new pattern AVONMCFF, which is equivalent to the composition of the two base patterns. The result of the AVONMCFF pattern in the after state is a new visible optional feature in the variability model, and a new compilation flag whose activation is subject to the presence of the newly added feature, together with *ifdefs* in code that refer to it. Since the

[22] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=72246da4



CTC $\quad$ $CTC' = CTC \cup CTC_f$

$M = \langle \ldots \rangle$ $\quad$ $M' = \langle \ldots (f, \textit{add a compilation flag c}) \ldots \rangle$

I $\quad$ I

Figure 3.13: Definition of Add Visible Optional Compilation Flag Feature (AVOCFF)

new feature does not hold a compilation unit of its own, it is non-modular. The pattern has three instances, corresponding to 1% of the additions sample size. All three instances are Boolean, adding features to arch (2) and driver (1).

Add Visible Mandatory Value-Based Feature (AVMVF)

This pattern, shown in Figure 3.15, covers the addition of a mandatory visible value-based feature (integer or string). As the feature is just a place-holder for a value, it does not add any cross-tree constraint, nor any compilation unit, preserving both CTC and M. The feature is, however, referred in the implementation when initializing specific parts of the code. Figure 3.16 exemplifies the pattern.[23] In the example, developers add RCU_BOOST_PRIO as a new feature in the variability model, setting it to depend on RCU_BOOST. Such dependency, however, is not a cross-tree constraint. Rather, the dependency is used by the Linux kernel configurator to place RCU_BOOST_PRIO as a child of RCU_BOOST. At the implementation level, RCU_BOOST_PRIO is referred in kernel/rcutiny_plugin.h as a means to properly define a macro with the same name as the feature (lines 20–24), but lacking the CONFIG_ prefix in code. The defined macro is then referred in kernel/rcutiny.c (line 32) when initializing a scheduling parameter.

Three instances of our sample (1 %) fall into this pattern, adding features to core (1) and driver (2).

[23] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=24278d14

```
drivers/usb/dwc3/Kconfig

+config USB_DWC3_VERBOSE
+   bool "Enable Verbose Debugging Messages"
+   depends on USB_DWC3_DEBUG
+   help
+      Say Y here to enable verbose debugging messages on
+      DWC3 Driver.
+
...

drivers/usb/dwc3/Makefile

+ccflags-$(CONFIG_USB_DWC3_VERBOSE) += -DVERBOSE_DEBUG
...

```

Figure 3.14: Example of Add Visible Optional Compilation Flag Feature (AVOCFF)

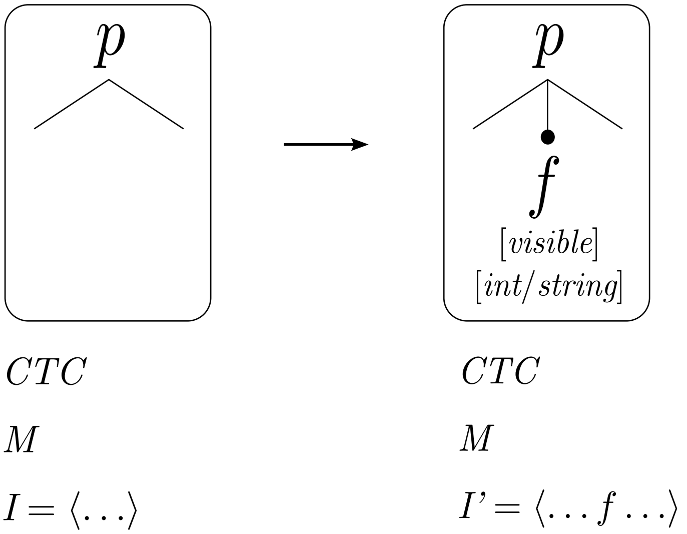

p
p
f
[*visible*]
[*int*/*string*]
CTC
M
$I = \langle \ldots \rangle$
CTC
M
$I' = \langle \ldots f \ldots \rangle$

Figure 3.15: Definition of Add Visible Mandatory Value-Based Feature (AVMVF)

Add Internal Modular Feature (AIMF)

Internal modular features are not directly exposed to users during configuration, as they are invisible (non-promptable). Such features exist to provide a common infrastructure to

```
init/Kconfig

+config RCU_BOOST_PRIO
+  int "Real-time priority to boost RCU readers to"
+  range 1 99
+  depends on RCU_BOOST
+  default 1
+  help
+     This option specifies the real-time priority to which
+     preempted RCU readers are to be boosted.  If you are
+     working with CPU-bound real-time applications, you
+     should specify a priority higher then the highest-priority
+     CPU-bound application.
+
...

kernel/rcutiny_plugin.h

...
+#ifdef CONFIG_RCU_BOOST
+#define RCU_BOOST_PRIO CONFIG_RCU_BOOST_PRIO
+#else /* #ifdef CONFIG_RCU_BOOST */
+#define RCU_BOOST_PRIO 1
+#endif /* #else #ifdef CONFIG_RCU_BOOST */

kernel/rcutiny.c

...
 static int __init rcu_spawn_kthreads(void)
 {
...
+  sp.sched_priority = RCU_BOOST_PRIO;
+  sched_setscheduler_nocheck(rcu_kthread_task, SCHED_FIFO, &sp);
   return 0;
 }
...
```

Figure 3.16: Example of Add Visible Mandatory Value-Based Feature (AVMVF)

other features, which in turn select them by means of reverse dependencies. Overall, this pattern comprises 4 % of all additions in our sample.

This pattern describes how internal modular features are added: as with other modular features, the variability model, mapping, and implementation change to accommodate the new feature (referred as f_1). However, two key characteristics arise: (i) f_1 is invisible; (ii) an additional constraint states that another feature f_2 selects f_1 (represented as an implication). Thus, the cross-tree constraints in the after-state are: $CTC' = CTC \cup CTC_{f_1} \cup \{f_2 \rightarrow f_1\}$.

Except for one feature in net, all other instances of AIMF concern the addition of driver-related features (92 %).

Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

Featurization occurs when existing elements are exposed as new features. One specific kind of featurization is when an existing compilation unit, initially subject to the presence of a feature p, becomes associated with its own feature, which is in turn created as a result. Such situation occurs in 4 % of additions.

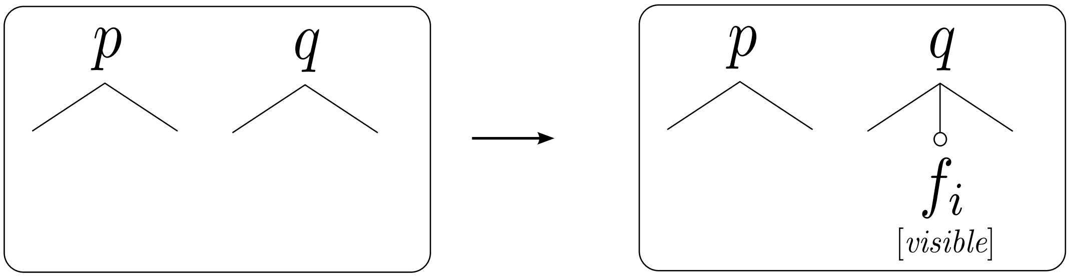
In the extracted pattern, illustrated in Figure 3.17, a feature p controls a set of object files $f_1.o \ldots f_n.o$. One of these objects, however, is not essential to the functionality provided by p; rather, its capability is optional. In this case, $f_i.o$ is featurized, i.e., a new feature f_i is created to control whether $f_i.o$ should be compiled or not. The new feature, in turn, is placed in the variability model under an existing feature q. Note that features p and q may or not be the same. Upon the creation of f_i, $f_i.o$ is then removed from the list of objects controlled by p. Featurizing $f_i.o$ gives users a finer-grained control over the configuration process, while decreasing the granularity of p. That prevents unnecessary functionality to be shipped in the resulting kernel, and in turn, improves its memory usage and boot time. The example shown in Figure 3.18 illustrates the featurization of me4000.o, previously controlled by COMEDI_PCI_DRIVERS, into the new feature COMEDI_ME4000.[24]

All 10 instances of the FCUTVOF pattern add features to the driver subsystem.

Featurize Code Fragment to Visible Optional Feature (FCFTVOF)

In this featurization pattern (see Figure 3.19), an unconditional code fragment C_0 becomes conditionally compiled and bound to the presence of a newly added feature f. To cover

[24] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f1d7dbbe



CTC

$M = \langle \ldots (p, f_1.o \ldots f_n.o) \ldots \rangle$

$I = \langle \ldots \boxed{f_1} \ldots \boxed{f_n} \ldots \rangle$

$CTC' = CTC \cup CTC_{f_i}$

$M' = \langle \ldots (p, f_1.o \ldots f_{i-1}\ f_{i+1} \ldots f_n.o)$
$\ldots (f_i, f_i.o) \ldots \rangle$

$I = \langle \ldots \boxed{f_1} \ldots \boxed{f_n} \ldots \rangle$

with $1 \leq i \leq n$

Figure 3.17: Definition of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

the case where f is not present, an alternative piece of code is given (C_1). When C_1 is not empty, the goal of the pattern is to provide an alternative behavior to an already existing implementation. Otherwise, the pattern extracts optional behaviour, decreasing the footprint of the resulting object code, which improves overall performance.

This FCFTVOF pattern covers 1 % (4) of the sampled additions, and for the most part (3) it concerns the featurization of code fragments in driver-related features. Figure 3.20 provides an example of the featurization of volume-related functions in the subdriver of the ACPI ALSA driver for ThinkPad©.[25] If THINKPAD_ACPI_ALSA_SUPPORT is present (a newly added feature), the volume-subdriver registers support for volume capabilities (not shown) and successfully initializes, as given by the return value in its init function (Figure 3.20, line 25); otherwise, THINKPAD_ACPI_ALSA_SUPPORT is not present, and volume capability-functions are not compiled in the resulting driver, causing the initialization of the volume-subdriver to fail, as given by the return value one (line 34). The commit log message of the patch confirms that the featurization is motivated by performance optimization:

[25] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=ff850c33

```
drivers/staging/comedi/Kconfig
```

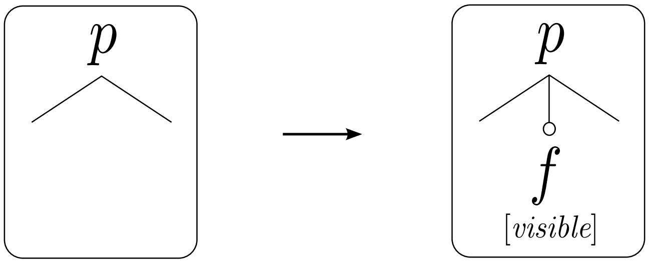
```
menuconfig COMEDI_PCI_DRIVERS
   tristate "Comedi PCI drivers"

+config COMEDI_ME4000
+   tristate "Meilhaus ME-4000 support"
+   help
+     Enable support for Meilhaus PCI data acquisition cards
+     ME-4650, ME-4670i, ME-4680, ME-4680i and ME-4680
...
```

```
drivers/staging/comedi/drivers/Makefile
```

```
-obj-$(CONFIG_COMEDI_PCI_DRIVERS) += me4000.o
+obj-$(CONFIG_COMEDI_ME4000) += me4000.o
...
```

Figure 3.18: Example of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)



CTC $\qquad$ $CTC' = CTC \cup CTC_f$

M $\qquad$ M

$I = \langle \ldots C_0 \ldots \rangle$ $\qquad$ $I' = \langle \ldots (\ldots f \ldots, C_0, C_1) \ldots \rangle$

Figure 3.19: Definition of Featurize Code Fragment to Visible Optional Feature (FCFTVOF)

> *"Allow the user to choose through Kconfig if the Console Audio Control interface (aka "volume subdriver") should be available or not. This not only saves some memory, but also allows the thinkpad-acpi driver to be built-in even if ALSA is modular when the console audio control interface is not wanted..."*

3.5.2 Inferred Feature Addition Patterns

We infer two patterns in the additions sample: *Add Visible Optional Abstract Feature (AVOAF)* and *Add Computed Internal Non-Modular Feature (ACINMF).* Both inferred patterns are bellow the threshold of three instances, but they have a corresponding inverse non-inferred pattern in the removals sample. The existence of an inverse non-inferred pattern in the removals sample suggests the inferred ones.

Add Visible Optional Abstract Feature (AVOAF)

This inferred pattern concerns the addition of *abstract features*, i.e., features that are exclusive to the variability model, and thus, are not referred in other spaces [134]. All the four cases of adding abstract features in the unexcluded portion of the additions sample relate to Boolean and optional features, but only half are visible. Thus, this pattern is

```
drivers/platform/x86/Kconfig

+config THINKPAD_ACPI_ALSA_SUPPORT
+   bool "Console audio control ALSA interface"
+   depends on THINKPAD_ACPI
+   depends on SND
+   depends on SND = y || THINKPAD_ACPI = SND
+   default y
+   ---help---
+     Enables monitoring of the built-in console audio output
+     control (headphone and speakers), which is operated by
+     the mute and (in some ThinkPad models) volume hotkeys.
...

drivers/platform/x86/thinkpad_acpi.c

+#ifdef CONFIG_THINKPAD_ACPI_ALSA_SUPPORT
...
static int __init volume_init(struct ibm_init_struct *iibm)
 {
    ...
    vdbg_printk(TPACPI_DBG_INIT,
                "initializing volume subdriver\n");
    ...
    return 0;
}
...
+#else /* !CONFIG_THINKPAD_ACPI_ALSA_SUPPORT */
+#define alsa_card NULL
+ ...
+static int __init volume_init(struct ibm_init_struct *iibm)
+{
+   printk(TPACPI_INFO, "volume: no ALSA support...\n");
+   return 1;
+}
+#endif
```

Figure 3.20: Example of Featurize Compilation Unit to Visible Optional Feature (FCUTVOF)

under our set threshold, as it has only two instances. However, as we report an inverse non-inferred pattern (RVOAF) in the removals sample, we classify these two visible features as part of an inferred pattern in the additions sample.

Interestingly, all abstract features in the addition sample are leafs in the variability model (as opposed to being internal nodes). In the cases where these abstract features are visible, their addition aims at capturing a configuration aspect that other features rely on. These features, in turn, do affect the mapping and/or implementation. Figure 3.21 illustrates this:[26] the addition of the visible optional abstract feature RD_XZ in the misc subsystem (lines 4–12) captures whether users want support for initial RAM disk compression. An initial RAM disk (initrd) is an initial root file system loaded as part of the kernel booting process, providing a minimal set of directories and executables that support the booting process (e.g., the insmod executable will be called to load different kernel modules, such as device drivers) before the actual file system is mounted. An initial RAM disk is kept as a compressed file, which is then uncompressed during the boot and placed in the primary memory (RAM). Upon the selection of RD_XZ, a reverse dependency selects DECOMPRESS_XZ, causing decompress_unxz.o to be compiled in a supporting library for the kernel (line 20). The other instance of this inferred pattern concerns the addition of an IPV4 feature in net.

The other two situations of adding abstract features relate to invisible ones. The two invisible features are capability abstractions [21] over the target hardware architecture for the kernel. Figure 3.22 illustrates this:[27] HAVE_KERNEL_GZIP (line 3) abstracts over gzip compression support of the target kernel image. As this functionality is not specific to x86, another feature KERNEL_GZIP exists, and its selection depends on the existing support of the target hardware architecture. Hence, x86 explicitly states its supported capabilities by selecting them, which includes HAVE_KERNEL_GZIP (line 23). Although this situation is actually prescribed in the Kconfig manual,[28] it was not found recurrent in our sample, and thus, we do not report it as a pattern. Moreover, it cannot be inferred, as we do not report an inverse pattern in the removals sample.

Add Computed Internal Non-Modular Feature (ACINMF)

This inferred pattern concerns the addition of a feature that is not promptable, and thus, it is invisible to users. Its presence is computed from a constraint setting the default value

[26] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=3ebe1243

[27] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=2e9f3bdd

[28] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

```

usr/Kconfig

+config RD_XZ
+   bool "Support initial ramdisks compressed using XZ"
+        if EMBEDDED
+   default !EMBEDDED
+   depends on BLK_DEV_INITRD
+   select DECOMPRESS_XZ
+   help
+      Support loading of a XZ encoded initial ramdisk or cpio
+      buffer. If unsure, say N.
+
...

lib/Makefile

+# XZ
 lib-$(CONFIG_DECOMPRESS_LZMA) += decompress_unlzma.o
+lib-$(CONFIG_DECOMPRESS_XZ)   += decompress_unxz.o
+
...
```

Figure 3.21: Example of Add Visible Optional Abstract Feature (AVOAF)

```
init/Kconfig

+config HAVE_KERNEL_GZIP
+  bool
+

 config KERNEL_GZIP
-   bool "Gzip"
+   depends on HAVE_KERNEL_GZIP
    help
      The old and tried gzip compression. Its compression ratio
      is the poorest among the 3 choices; however its speed
      (both compression and decompression) is the fastest.

...

arch/x86/Kconfig

config X86
   select HAVE_GENERIC_DMA_COHERENT if X86_32
   select HAVE_EFFICIENT_UNALIGNED_ACCESS
   select USER_STACKTRACE_SUPPORT
+  select HAVE_KERNEL_GZIP
...
```

Figure 3.22: Example of an invisible optional abstract feature

of the feature. The added feature is referred in code by means of *ifdefs*; as it does not have a compilation unit, the feature is non-modular. Computed internal features exist to encapsulate specific constraints, which simplifies the encoded variability; instead of repeating the constraint at each variation point that it is needed, developers encapsulate it in a single feature, which facilitates later maintenance when updating the constraint. Two instances of this inferred pattern appear in the additions sample, and both concern Boolean features being added to arch and core, respectively.

3.5.3 Non-Inferred Feature Removal Patterns

Non-inferred patterns in the removal sample capture how the mapping and implementation spaces change, if at all, upon the removal of an existing feature in the variability model. Excluding rename (RNM), we report nine non-inferred patterns in the removals sample, from which seven capture retirement situations directly matching their counterpart in the additions sample: *Retire Visible Optional Modular Feature (RVOMF)*, *Retire Visible Optional Guard Modular Feature (RVOGMF)*, *Retire Visible Optional Non-Modular Feature (RVONMF)*, *Retire Visible Optional Abstract Feature (RVOAF)*, *Retire Visible Mandatory Value-Based Feature (RVMVF)*, *Retire Internal Modular Feature (RIMF)*, and *Retire Computed Internal Non-Modular Feature (RCINMF)*. Among these, retirement patterns removing visible optional features and affecting the implementation space account for most removal cases. The inverse addition patterns matching these removal patterns show the same trend. Thus, both trends suggest that the kernel evolution is mainly driven by adding or removing visible optional features with some associated implementation. Moreover, as observed in the additions sample, most removal patterns relate to features in the driver subsystem (see Table 3.8).

Kernel maintainers retire features when: (a) the features are under staging (unstable features) for a long time, and there is no indication that they will gain enough quality to be merged into the main kernel. Reasons include broken, unmaintained, or buggy features, or non-adherence to development conventions; (b) the features break due to changes elsewhere and no effort is put to fixing them; (c) the features are not used and are unmaintained for a long time; (d) another feature supersedes an obsolete one, causing the latter to be retired.

Interestingly, 67 % of RIMF and RVMVF, 64 % of RVONMF, 50 % of RVOAF, and 27 % of the RVOMF instances are removed as a consequence of retiring the whole subtree containing them. This suggests that some forms of retirement occur in a coarse-grained manner and are triggered by the removal of a feature rooting an entire subtree, along with all its descendants.

Table 3.8: Frequency of non-inferred patterns per subsystem (removals sample)

Pattern	Distribution across subsystems						
	arch	core	driver	firmware	fs	misc	net
RVOMF	1	1	20	0	0	0	0
RVOGMF	0	0	12	0	0	0	0
RVONMF	0	0	10	0	0	0	0
RVOAF	0	0	4	0	0	0	2
RVMVF	0	0	2	0	0	0	1
RIMF	0	0	3	0	0	0	0
RCINMF	2	0	1	0	0	0	0
MVOFNO	0	0	3	0	0	0	0
MVOFS	0	0	3	0	0	0	0
RNM	0	0	16	0	1	0	1

Non-retirement patterns also exist, and capture cases where a feature is merged into another one. Two such patterns exist: *Merge Visible Optional Feature into New One (MVOFNO)* and *Merge Visible Optional Feature into Sibling (MVOFS)*. The instances of each merge pattern concern the merging of features in the driver subsystem. We present MVOFNO and MVOFS in the following.

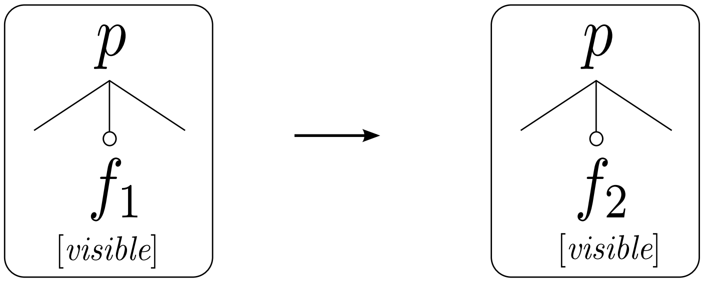
Merge Visible Optional Feature into New One (MVOFNO).

This pattern concerns the creation of a feature from an existing one, which is then enhanced with new code. Figure 3.23 illustrates the pattern. A feature f_1 is renamed to f_2, and its set of cross-tree constraints is replaced with a new set CTC_{f_2}. Furthermore, all references to f_1 are replaced by references to f_2 in all spaces. At the implementation level, $f_2 > f_1$ captures the enhanced code, meaning that f_2 supports all the capabilities of f_1, plus new ones.

Of all instances in the removals sample, 2 % (3) fit into this pattern and often relate to generalizing drivers to support a set of related hardware family.

As a concrete example, consider the merge of BATTERY_PALMTX into the new feature BATTERY_WM97XX supporting a whole family of chips.[29] As shown in the associ-

[29] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=4e9687d9



$$
\begin{array}{ll}
CTC = CTC_{others} \cup CTC_{f_1} & CTC' = CTC_{others_{[f_2 \backslash f_1]}} \cup CTC_{f_2} \\
M & M' = M_{[f_2 \backslash f_1]} \\
I & I' = I_{[f_2 \backslash f_1]} \\
 & \textit{with } f_2 > f_1
\end{array}
$$

Figure 3.23: Definition of Merge Visible Optional Feature into New One (MVOFNO)

ated patch (see Figure 3.24), developers drop the original cross-tree constraints and rename the previous feature in the variability model and mapping. Moreover, the code is updated with various information about the new driver (not shown). Note that in the example, the merge changes the associated help text, but it does not relate the new feature back to BATTERY_PALMTX. Thus, when users migrate towards a newer kernel with BATTERY_WM97XX, they may incorrectly conclude that BATTERY_PALMTX is no longer supported. Hence, merges can cause the false impression that some features cease to exist.

Merge Visible Optional Feature into Sibling (MVOFS)

This pattern covers the situation in which developers merge a visible optional feature into its sibling (see Figure 3.9), due to their similarity. The merging of FB_IMAC into FB_EFI, previously discussed in Section 3.4.6, exemplifies the pattern. This pattern aims at easing maintenance, as keeping two similar features might require a duplicate effort whenever a change occurs in either of them. As other merges, this pattern is responsible for 2 % (3) of all removals in the sample.

```
drivers/power/Kconfig
```

```
-config BATTERY_PALMTX
-   tristate "Palm T|X battery"
-   depends on MACH_PALMTX
+config BATTERY_WM97XX
+   bool "WM97xx generic battery driver"
+   depends on TOUCHSCREEN_WM97XX
    help
-     Say Y to enable support for the battery in Palm T|X.
+     Say Y to enable support for battery measured by WM97xx
...
```

```
drivers/power/Makefile
```

```
-obj-$(CONFIG_BATTERY_PALMTX) += palmtx_battery.o
+obj-$(CONFIG_BATTERY_WM97XX) += wm97xx_battery.o
...
```

Figure 3.24: Example of Merge Visible Optional Feature into New One (MVOFNO)

3.5.4 Inferred Feature Removal Patterns

We infer two removal patterns: *Retire Visible Optional Compilation Flag Feature (RVOCFF)* and *Retire Visible Optional Non-Modular Compilation Flag Feature (RVONMCFF).* Opposed to the inferred patterns in the additions sample, we do not find any instances of these two patterns. However, as these feature types are added as seen in the additions sample, it is reasonable to assume that one way of retiring such features is by performing the opposite steps of their addition.

3.5.5 Non-Patterns in the Additions and Removals Samples

The patterns reported in Table 3.4 cover most of the additions (81 %) and removals (63 %) we analyzed. However, not every change results in a pattern. Following the names defined in the previous section, the additions that do not match a pattern and are not excluded from analysis (38) fall into the following cases:

- Addition of guard features (5), i.e., features whose sole purpose is to guarantee the compilation of the content inside a given folder. Although such case respects our defined threshold of three instances, the changes in this cluster do not stem from three distinct developers.
- Addition of internal optional abstract features (2).
- Addition of computed internal modular features (2).
- Addition of a computed internal non-modular feature whose extension combines new code fragments with existing lines of code (1).
- Addition of an internal mandatory modular feature (1).
- Addition of internal mandatory non-modular features (2).
- Addition of internal optional non-modular features (2).
- Addition of a computed value-based feature, i.e., a value-based feature whose presence is computed (1).
- Different situations of exposing existing code as a feature (15).
- Distinct merge cases (4).

- Featurization of existing constraints in the variability model (2).
- Combination of the rename of a feature and the split of its compilation unit (1).

In the case of the 38 unexcluded instances in the removals sample that are not put as part of a pattern, we report the following situations:

- Removal of individual cases of internal features (4) not fitting RIMF nor RCINMF. A concrete example includes the removal of an internal optional compilation flag feature.
- Different cases where a feature becomes an integral part of the code, while being removed from the variability model (16).
- Different merge situations that do not lead to patterns (17).
- One split case.

Compared to the additions sample, removals tend to contain more merge-related changes, with a rich realization that leads to different ways on how to accomplish them. Consequently, few merge patterns arise.

3.6 Discussion

To align our results with our main research goal, we discuss how our variability-coevolution patterns advance the understanding of variability evolution in real-world settings.

3.6.1 Variability Evolution Support Requires Accounting for the Coevolution of Variability Models with Related Artifacts

Our patterns provide empirical evidence that any approach claiming to support variability evolution only at the level of variability models is likely to be ineffective in practice, as most observed additions and removals cases require changes outside the variability model. This contrasts with existing research, which has been mostly focused on the evolution of variability models only (see Chapter 2). Moreover, some changes can only be understood by inspecting the coevolution with other artifacts (e.g., as in the case of merges).

To illustrate our point, consider the edit-based reasoning technique proposed by Thüm et al. [133], a state-of-the-art solution for assessing how variability model changes affect the set of possible derivable variants. In such approach, changes are classified as:

- *Generalization*: The introduced changes in the variability model do not impact previous valid configurations. Thus, new variants can be derived, while preserving the original set of variants.
- *Specialization*: The changes decrease the original set of derivable variants.
- *Refactoring*: The set of derivable variants resulting from the changes remains the same.
- *Arbitrary edit*: None of the above.

Reasoning is performed by efficiently translating both the original variability model and the one resulting from the changes into a satisfiability problem; by avoiding an exponential explosion of CNF clauses, the proposed reasoning has been tested over large models, showing to scale with randomly-generated models with up to 10,000 features. Moreover, reasoning does not require variability models to have the same set of features, as generalization can include new ones, and specialization remove others. This contrasts to previous work analyzing equivalence or specialization of variability models with the same set of features [71, 128].

Despite scalability advances, the work of Thüm et al. is not sound. Specifically, their approach does not yield correct results if changes affect the feature set, but preserve the overall functionality with changes in other spaces. The merging of FB_IMAC into FB_EFI, discussed in Section 3.4.6, illustrates this situation. While FB_IMAC is removed from the variability model, FB_EFI supersedes the removed feature in the implementation space. Furthermore, since FB_EFI has the same cross-tree constraints as FB_IMAC, no constraint is lost (a renaming refactoring updates references to FB_IMAC to become references to FB_EFI). After the merge, all functionality remains the same, as support for FB_IMAC is now given by FB_EFI. Hence, the set of variants does not change. In contrast, Thüm's edit-reasoning reports such merge as a specialization, indicating a shrink in the set of variants—an incorrect result. Other reasoning techniques [71, 128] are even more limited, as they only reason over variability model edits that preserve the feature namespace. As the Linux kernel case shows, many cases do not fit into such situation.

3.6.2 New Theories Must Account for Feature Retirement

Our catalog shows that feature retirement comprises most of the patterns in the removals sample. Thus, removals are frequent in the evolution of the Linux kernel. While the MVOFS and MVOFNO patterns are captured by the only known theory of software-product-line refinement [22], retirement patterns are not. Thus, a new theory of product-line evolution that covers not only refinement, but also retirement situations is needed. As our catalog is the first of its kind, our patterns serve as a starting point for understanding specific types of feature removals that should be accounted by new theories.

3.6.3 High Modularity, Low Scattering Degree, Localized Scattering, and Disciplined Annotations are Key Evolution Principles

In our catalog, the two most frequent patterns are AVOMF and AVONMF, accounting for 58 % of all investigated feature additions. These two patterns reveal key principles governing the Linux kernel evolution.

The AVOMF pattern, the most recurrent pattern in our catalog, shows that most additions introduce modular features, i.e., features that have their own compilation unit(s). Moreover, AVOMF features tend to be fully modular, allowing developers to confine their implementation under well-defined interfaces (e.g., the driver-development API), causing changes to be localized and fostering parallel development—a key strategy in the distributed setting in which the kernel is developed. When features are not fully modular, the extensions they introduce are disciplined, aligning with the syntactic units of the C language. AVOMF instances introducing extensions are not heavily scattered across the kernel. Instead, they have low *SD* (2), with extensions generally placed in files in the same subsystem as their associated primary features (local scattering). The dominance of the AVOMF pattern and its localized scattering suggest that the Linux kernel evolution is in line with the system's software architecture.

The extensions introduced by non-modular features, as prescribed by AVONMF, are also disciplined. Like AVOMF, extensions tend to occur at the global and function levels. As argued elsewhere [87, 89], disciplined annotations facilitate maintenance activities, such as refactoring in the presence of *ifdefs*. Moreover, disciplined annotations at the global level can be rewritten with constructs of alternative techniques favoring better modularity—e.g., by using aspects [2]. In terms of locality, the proportion of locally scattered AVONMF features dominates the globally scattered ones; the latter, however, present higher scattering degrees. Nonetheless, both kinds have a low median *SD*-value (≤ 6).

The dominance of modular features, low *SD*, localized scattering, and disciplined annotations mitigate the challenges imposed by the use of *ifdef* annotations on program comprehension [51, 75, 86, 125] and on the potential of introducing bugs [50, 79]. While modularity is supported by the plugin architecture of the kernel, low scattering and disciplined annotations appear to follow directly from coding guidelines related to *ifdef* use:[30]

> *"Code cluttered with ifdefs is difficult to read and maintain. Don't do it. Instead, put your ifdefs in a header, and conditionally define 'static inline' functions, or macros, which are used in the code. Let the compiler optimize away the "no-op" case."*

The kernel development process also reinforces that understanding:[31]

> *"The C pre-processor seems to present a powerful temptation to some C programmers, who see it as a way to efficiently encode a great deal of flexibility into a source file. But the pre-processor is not C, and heavy use of it results in code which is much harder for others to read and harder for the compiler to check for correctness. Heavy pre-processor use is almost always a sign of code which needs some cleanup work [...] Conditional compilation with #ifdef is, indeed, a powerful feature, and it is used within the kernel. But there is little desire to see code which is sprinkled liberally with #ifdef blocks."*

which is further stressed by Linus Torvalds himself when rejecting a contributed patch:[32]

> *"Note that there is no way I will ever apply this particular patch for a very simple reason: #ifdef's in code [...] And make your #ifdef's be _outside_ the code. I hate code that has #ifdef's. It's a major design mistake [...] So please spend some time cleaning it up, I can't look at it like this."*

—Linus Torvalds, Wed, 8 Aug 2001 09:40:07 (fa.linux.kernel newsgroup)

Torvalds continues:

[30] https://www.kernel.org/doc/Documentation/SubmittingPatches

[31] https://www.kernel.org/doc/Documentation/development-process/4.Coding

[32] http://yarchive.net/comp/linux/ifdefs.html

"Having the #ifdef's outside the code tends to have two advantages:

- it makes the code much more readable, and doesn't split things up.

- you have to choose your abstraction interfaces more carefully, which in turn tends to make for better code.

Abstraction is nice - _especially_ when you have a compiler that sees through the abstraction and can generate code as if it wasn't there."

—Linus Torvalds, Wed, 8 Aug 2001 12:14:32 (fa.linux.kernel newsgroup)

Localized scattering appears to follow from the design of the Linux kernel software architecture, which seems able to encapsulate most features in their containing subsystem. However, the proportion of globally scattered features, as AVONMF instances indicate, is not negligible. This leads us to believe that there exists some features that do not fit well into the Linux kernel layered architecture or that are not easily modularized in the C language.

3.7 Threats to Validity

Following best practices in case study analysis [147], we analyze the trustworthiness of our results according to three threat types: construct, external, and reliability.

Construct Validity. Limiting non-inferred patterns to have at least three instances raises a threat to construct validity. We argue, however, that our choice guarantees the inclusion of less frequent patterns, while still defining a minimal threshold. It also prevents us from reporting non-inferred patterns over extreme outliers, which would represent rare evolution scenarios. To avoid bias towards personal change styles, the non-inferred patterns are also required to have three distinct sources of evidence, meaning that the patterns have been employed by at least three distinct developers. Inferred patterns, in contrast, do not guarantee the existence of at least three instances, nor three distinct sources of evidence. Thus, inferred patterns impose an additional threat. We argue, however, that it is logical to assume the existence of an inferred pattern, as long as we provide evidence that its inverse pattern is not inferred. The latter follows directly from our methodology. To prevent readers from interpreting inferred patterns as non-inferred ones, we clearly label them when reporting frequencies.

Manually mining patterns is an additional threat. We mitigate this threat by devising and following a methodology with a well-defined sequence of steps. Some steps, however, require subjectivity (e.g., defining the size of commit windows and cluster categories). Following best practices in case study research [118], we mitigate subjectivity by performing at least three extensive reviews, followed by a consensus analysis to assure consistency. We also document all the collected data and our analysis, making them publicly available for independent verification.

Last, but not least, we acknowledge that our patterns result from an indirect observation of what developers do. As such, despite the fact that we are able to explain most of the additions and removals in our samples, our catalog may not represent the evolution at the same abstraction level as perceived by kernel developers. Moreover, as our patterns directly follow from the analysis of the Linux kernel commit history, they cannot capture any practice occurring outside what is seen in the source code repository.

External Validity. Our scoping decisions threaten external validity. First, our study lacks *population generalizability.* Our analysis is scoped to additions and removals in the variability model of the x86 architecture, while observing how related artifacts coevolve as a result. Despite existing evidence that the x86 variability model follows a similar growth in comparison to the variability model of the whole kernel [91], it is not safe to claim that our patterns are representative of all hardware architectures supported by the Linux kernel. Similarly, we cannot claim that our patterns are representative of the entire Linux kernel evolution. Second, our patterns do not have *cross-population* generalizability. Stated otherwise, we cannot claim that our patterns are representative of feature additions and removals as found in other variability-aware software systems, open-source or not. As a first study of its kind, our work shall be succeeded by further assessments to verify whether the reported patterns are exclusive to the evolution of the Linux kernel or whether they are also found in other systems. Any system organized in terms of a variability model, a mapping, and source code relying on *ifdef* annotations, is a prospective candidate. Examples include other open-source Kconfig-based software systems [21, 100], the eCos real-time operating system [96], and even industrial software product lines [17].

Reliability Validity. As our methodology requires manual steps and even subjective judgment, the instructions on how to mine coevolution patterns must be as clear as possible. Otherwise, if participants have distinct interpretations when applying our methodology to mine a pattern from a given commit patch, results may not be consistent among different participants (*inter-item reliability* threat), or may even differ for the same participant when applying the methodology at distinct times (*test-retest* reliability threat). To mitigate reliability threat, we discussed different versions of our methodology with researchers in the field, collecting feedback and improving the clarity of our instructions. After reaching a final version of the methodology, we carefully added examples to illustrate specific points, leading it to its final format presented earlier in this chapter. Having independent reviews—as performed by participants P_3 and P_4—allows us to indirectly assess reliability. For instance, P_3 and P_4 show an agreement rate of at least 86 % with the performed mining results. These results suggest a low reliability threat for our methodology.

3.8 Conclusion

Mining a sample of the Linux kernel evolution history reveals 23 variability-coevolution patterns, including 19 non-inferred cases and four inferred ones. Our patterns show a close relationship between adding and removing feature names and evolving related artifacts. Some patterns empirically evidence the incorrectness of state-of-the-art approaches when handling specific evolution scenarios. In other cases, our patterns show the need of a more general theory of product-line evolution. In the context of the Linux kernel, patterns are mostly concerned with modular features, although scattering still occurs. The latter, however, tends to be localized, with low scattering degree values and with a disciplined use of annotations. Thus, Linux kernel developers seem to keep scattering under control.

Next, we investigate the expressiveness and the generality of our coevolution patterns (Chapter 4), followed by an in-depth analyses of feature scattering (Chapter 5). In the latter case, we want to understand the extent of feature scattering, practical limits governing its evolution, in addition to possible contributing factors.

Chapter 4

Variability-Coevolution Patterns: Beyond the Linux Kernel

Our catalog of patterns from Chapter 3 stems from mining a sample of the Linux kernel evolution history. Despite providing an insightful understanding of how variability models coevolve with other artifacts in the context of a large and complex real-world system, our catalog lacks evidence of whether its patterns take part in the evolution of systems other than Linux. To fill such a gap, this chapter validates our patterns in the context of three other variability-aware systems: axTLS, Toybox, and uClibc. We choose these three target systems by sampling well-known systems in the same domain of the Linux kernel—the systems software domain. Our validation investigates the expressiveness of our patterns in covering the induced variability coevolution when developers add or remove feature names in the variability models of our target subjects, along with the extent that our catalog generalizes beyond the Linux kernel case.

Chapter Organization. Section 4.1 presents our research questions. Section 4.2 discusses our methodology, explaining how we select three target subjects. The section also explains our scoping decisions and data collection procedure, in addition to our data analysis rationale. Section 4.3 presents our results, followed by some remarks of the patterns we validate (Section 4.4). We argue about threats to validity in Section 4.5, concluding the chapter in Section 4.6.

4.1 Research Questions

To validate our catalog, we ask two research questions. The first focuses on the expressiveness of our patterns; the second targets the generality of our catalog.

RQ. 1 Is our catalog of patterns expressive enough to capture the variability coevolution induced by the addition or removal of features in the variability models of existing variability-aware sytems?

To answer our first research question, we proceed as follows. Given our three target subjects, we mine all their releases within our scope, listing all the cases where developers add and remove feature names from the target variability models. For each addition and removal case, we analyze its induced coevolution on related artifacts, counting the number of cases that match a pattern in our catalog. The percentage of matching cases, i.e., patterns instances, defines our expressiveness measure.

RQ. 2 From our catalog, what variability-coevolution patterns are general?

At the core of our second question is the notion of *generality.* We operationalize the generality of a pattern as a conjunction of three conditions:

G1 A general pattern has at least three instances.

G2 At least three distinct developers author a general pattern.

G3 At least three distinct systems employ a general pattern.

While conditions G1 and G2 borrow from the same constraints in the Linux kernel case, the third condition allow us to generalize our patterns beyond Linux. Altogether, our operationalization adheres to the *Rule of Three*—it requires three distinct sources of evidence, both in the number of developers as in the number of systems.

When answering our second research question, we assume that our catalog does contain general patterns. Thus, our question focus on identifying which ones are indeed general.

Assumption. Our catalog of variability-coevolution patterns contains general patterns.

Our assumption follows from preliminary evidence supporting the generality of AVOMF. The generality of other patterns, whilst missing empirical evidence, follows from defendable arguments.

In the case of AVOMF, Neves et al. [104] report a change type whose structure closely matches our pattern. From the change type, the authors derive a corresponding template—*Add new optional feature*. Neves' templates, however, are not patterns. Opposed to our generality notion, their templates do not enforce three distinct sources of evidence, defined both in terms of systems as in terms of developers. Moreover, their templates impose behaviour preservation; our patterns do not. Nonetheless, it suffices to say that their study finds *Add new optional feature* instances in two systems; if we accumulate the evidence we find from the Linux kernel, we reach a total of three distinct systems applying the change, providing some evidence towards the generality of AVOMF.

We also argue for the generality of other patterns, although supporting empirical evidence is currently missing. For instance, consider the RNM pattern. Renaming is often employed when refactoring different elements (e.g., classes, functions, database table names, etc) in a given system; since features are also a target for renaming, it is unrealistic to believe RNM as exclusive to the Linux kernel evolution. The AVONMF pattern is also a pattern likely to be general, at least in the case of annotative-based variability-aware systems.

4.2 Methodology

This section describes our methodology in selecting our subjects of analysis, associated scoping decisions, and our data collection and analysis procedures.

4.2.1 Subject Selection

Our validation focuses on systems in the same domain of the Linux kernel—the systems software domain. In addition to the Linux kernel, Berger et al. [21] report 11 other major open-source and variability-aware systems in the systems software domain. Table 4.1 briefly describes each system. All systems follow a similar structure as the one in the Linux kernel, having a variability model, mapping, and implementation.

We take the systems in Table 4.1 as our target population of subjects. Employing our methodology to all systems in the table, however, is a daunting task. As we discuss later in the text, our methodology employs human judgment when matching the before and after states of commit windows of feature additions and removals against our catalog of patterns; manually inspecting the commit windows of all addition and removal cases across the entire evolution of all 11 target systems makes analyses a time-consuming activity, imposing a high research cost. As a trade-off, we set to collect a sample from the original set of 11 targets.

From the target population, we select three target systems in a non-random manner; together with the Linux kernel, that leads to a total of four systems to assess pattern generality, satisfying our third generality condition (G3). Selecting subjects results from the application of seven filters, which we apply sequentially. Such filters aim at decreasing the sample space to a final set of three subjects, while eliminating as many sources of bias and overlapping among target subjects as possible.

Our first filter concerns the target variability modeling language. Since our current pattern notation directly abstracts Kconfig constructs into a FODA-based representation, we restrict analysis to systems employing the Kconfig modeling language. From all the systems in Table 4.1, eCos is the only one not adopting Kconfig; instead, eCos represents features using the *Component Description Language* (CDL) [96]. Hence, we exclude eCos from our final set of subjects.

The second filter we apply selects systems whose source code repository is Git. Our filtering criterion allow us to reuse much of our existing infrastructure targeting the Git repository of the Linux kernel. Exceptionally, we also consider SVN-based projects, as their conversion to Git is allegedly straightforward [27]. Among the target systems, uCLinux does not satisfy our filter—its source code is in a CVS repository. Thus, we do not consider uClibc as a subject of analysis.

Third, we discard operating systems as subjects of analysis. Since our catalog stems from the Linux kernel operating system, subjects whose functionality overlaps with those in the Linux kernel could result in sample bias. As a conservative measure, we also exclude systems that deploy an operating system environment. From the third filter, we discard Fiasco, BuildRoot, EmbToolkit, and Freetz.

In tune with the previous case, our fourth filter eliminates systems with overlapping functionalities; when conflicting cases exist, we keep a single system within each conflicting group. From the remaining five systems, we find an overlap between BusyBox and ToyBox. We are aware that ToyBox is a spin-off of the BusyBox project. The former allegedly has

Table 4.1: Target population of systems in the systems software domain

System	Description	Repository
axTLS	A client/server TLSv1 SSL library with small memory footprint URL: `http://axtls.sourceforge.net`	SVN
BuildRoot	A configurable tool for building a complete and bootable Linux environment for an embedded device URL: `buildroot.uclibc.org/`	Git
BusyBox	A tool to link UNIX® selectable command-line utilities into a single executable URL: `http://www.busybox.net`	Git
CoreBoot	An open-source BIOS implementation URL: `http://www.coreboot.org`	Git
eCos	A real-time operating system with selectable user-space packages URL: `http://ecos.sourceware.org/`	CVS/Hg
EmbToolkit	A tool for building tool chains for embedded Linux system development URL: `https://www.embtoolkit.org`	Git
Fiasco	An L4 real-time micro-kernel URL: `http://os.inf.tu-dresden.de/fiasco`	SVN
Freetz	A firmware-extension for AVM FritzBox routers URL: `http://freetz.org`	SVN/Git
ToyBox	An alternative implementation of BusyBox, allegedly better designed and simpler to extend with new command-line utilities URL: `http://www.landley.net/toybox`	Git
uClibc	A configurable library for C-development in embedded Linux systems URL: `http://uclibc.org`	Git
uClinux	A port of the Linux kernel targeting CPUs lacking a memory management unit URL: `http://www.uclinux.org`	CVS

Table 4.2: Distribution of source code file types (averages taken from the entire evolution of the target subjects)

System	File type (Mean %)			
	C implementation file	C header file	Assembly	Other
axTLS	60.5	26.4	0	13.1
Toybox	87.1	6.5	0	6.4
uClibc	50	39.5	9.9	0.6

a cleaner and simpler implementation. Its creator, Rob Landley, explains:[1]

> *"When I was maintaining BusyBox there was tons of stuff I wanted to rewrite, and my goals were to make the code simple, small, fast, and full-featured. In that order. Simple was more heavily weighted than any other concern, each increase in speed and features or reduction in size had to justify the added complexity. I treated complexity as a cost, and wanted to get the best bang for the buck.*
>
> *BusyBox has wandered away from that, simplicity is now less important than small size, increased speed, or added features. They've kept the size under a megabyte, but the code's full of magic symbols and #ifdefs. The entry point to the whole program is now buried in a subdirectory near the end of a large file inside the #else case of an #ifdef, and that entire main() function doesn't contain a single line of code that isn't inside one of four other #ifdef blocks.*
>
> *One of my goals with ToyBox is that if you're just learning C programming, reading ToyBox should be a reasonable real-world introduction to the language."*

Inspecting the posts in LWN.net portal, one of the premier news and information websites in the open-source community, shows ToyBox to be gaining momentum. In fact, ToyBox is now part of the Android Open Source Project, replacing Google's toolbox.[2] BusyBox had also been considered as an official replacement for toolbox, but licensing proved to be an issue. Due to its increasing importance and supposedly better implementation, we favor ToyBox as a target subject.

[1] `http://tinyurl.com/Inside-the-ToyBox-An-interview`

[2] `https://lwn.net/Articles/629362/`

Table 4.3: Size of our three subjects of analysis

System	SLOC		Nbr. of features	
	min	max	min	max
axTLS	18,516	36,521	74	95
Toybox	10,671	38,846	84	249
uClibc	112,694	263,122	70	297

Our fifth filter aims at eliminating managerial bias. As such, we verify that no two systems share the same head maintainer; if so, we select exactly one system among conflicting ones. This filter has no effect—after inspecting the documentation of the systems available at this step, we do not find any managerial overlap.

The sixth filter we apply selects systems with at least two stable releases. Our rationale is as follows. A pair of stable releases (r_i, r_{i+i}) allows identifying features in r_{i+1}, but not in r_i; thus, we can enumerate the corresponding feature addition cases. Likewise, having at least two stable releases allows identifying removals cases, i.e., the set of features in r_i, but not in r_{i+1}. CoreBoot fails to satisfy our criterion; at the time of our selection, we only find version 4.0 in CoreBoot's master branch.

Lastly, the seventh filter in our selection process guarantees a sample size of three. To do so, it randomly selects three subjects from the remaining set of systems. This filter, however, has no effect. The application of the previous six filters already leads to a final set of three systems. Consequently, our sample is purely non-random.

Similar to the Linux kernel case, all chosen subjects are predominantly written in C (see Table 4.2), with pre-processor annotations referring to features declared in the variability model. The chosen subjects, however, are smaller than the Linux kernel, both in terms of code size as well as in their number of features—see Table 4.3. The target subjects, however, are considerably higher than the ones in the SPL2Go website, which lists commonly used research subject snapshots. Contrasting Table 1.1, which provides summary statistics of the systems in SPL2Go, with the statistics in Table 4.3 shows that axTLS, ToyBox, and uClibc have at least 89 % more features than 96 % of the systems in the SPL2Go listing. A similar trend occurs for SLOC; our three chosen subjects have at least 87 % more source lines of code than the same percentage of the systems in Table 1.1. Altogether, these statistics reiterate our concern in studying real-software systems, as they seem more likely to be representative of real-world complexity than often chosen small research subjects.

Table 4.4: Mapping of axTLS login names to corresponding Git users

SVN login	Git user	
	Name	Email
cameronrich	Cameron Rich	camster444@gmail.com
ehuman	Eric Human	EHu@directv.com
olereinhardt	Ole Reinhardt	ole.reinhardt@embedded-it.de

4.2.2 axTLS Conversion to Git

After our filtering process, there is a single non-Git based project in our final sample: axTLS. Our Linux-based infrastructure, however, only operates on Git-based repositories. Thus, reusing it requires converting the evolution history of axTLS into Git. To perform the conversion, we rely on the svn2git tool.[3] Its use requires mapping each SVN login into a Git-based user, i.e., a username and email pair. We create such mapping by first collecting the login names in the axTLS SVN commit history. We then mine the axTLS mailing list to find names closely matching the login names, extracting the corresponding email addresses. Table 4.4 shows our final mapping.[4] With the latter, svn2git successfully creates a Git repository for axTLS. The resulting repository is available at our online appendix [1].

4.2.3 Scoping

Once all subjects are under Git, we set to inspect all feature additions and removal cases across consecutive stable releases. As in the Linux kernel, the configuration of each subject requires setting a target architecture. For consistency, our CPU of scope is the same as the one we choose for our Linux kernel case—x86.

Another factor influencing scoping concerns the extent that we can generalize our Linux-based infrastructure to handle the specifics of each of our chosen subjects. After understanding the structure of each target subject, we notice that their main differences lie in their build systems; although closely resembling KBuild, variations exist.

[3] Conversion command: `svn2git http://svn.code.sf.net/p/axtls/code/ -no-minimize-url -verbose -nobranches -authors axtls.authors`

[4] We save our mapping in the axtls.authors file; we use it as an argument to svn2git (see footnote above).

Table 4.5: Stable release identification and sorting criteria

System	Stable release identification criteria
axTLS	The project's website does not provide a list of stable releases. Inquiring the head maintainer of the project clarifies the issue; all tags in the project's SVN repository label stable releases. We then map each SVN tag to its corresponding one in our Git repository. Tags in the SVN repository are already sorted according to chronological order
ToyBox	We assume that all the compressed files in the downloads section of ToyBox's website are stable releases. We then list all compressed files, sorting them according to their last modification date. We then map each release file to its corresponding release tag in Git, obtaining a chronological order of release tags
uClibc	Same criteria as in ToyBox

One variation type regards distinct project conventions and default settings. Examples include different name referencing schemes (e.g., in uClibc, if `FOO` is a feature name, references in Makefiles and C code are written as `__FOO__`; in ToyBox, it becomes `CFG_FOO`), distinct variability model names (e.g., in axTLS, Kconfig files are named `Config.in`), different system paths, particular ways for identifying and ordering stable releases, etc. To handle the variations as the ones above, we parameterize our tools accordingly.

A second type of variation follows from the different versions of the Kconfig language in each project and/or release. Our existing Linux-based infrastructure requires parsing Kconfig files to identify added and removed feature names between stable releases. Its implementation, however, only parses variability models adhering to the Kconfig language as given in the v2.6.26–v3.3 Linux kernel release range. To reuse our Linux-based infrastructure for axTLS, ToyBox, and uClibc, our existing parser must handle the different versions of the Kconfig language. There are two major approaches for generalizing our parser: (i) create a *one-size-fits-all* parser, whose automaton recognizes a unified version of the Kconfig language given at each stable release in the evolution history of our three subjects, or (ii) generate multiple parsers, one for each version of Kconfig within a target project and release.

In the one-size-fits-all solution, a parser results from the merge of different grammars, one for each release of Kconfig. Merging also combines associated semantic actions, commonly employed in Yacc-like grammars [3], such as Kconfig's. Merging, however, has considerable drawbacks: (a) when developers release new versions of Kconfig, one needs to incorporate changes in the source code of the one-size-fits-all parser, leading to intrusive maintenance; (b) unifying different Kconfig releases means diffing each grammar snapshot,

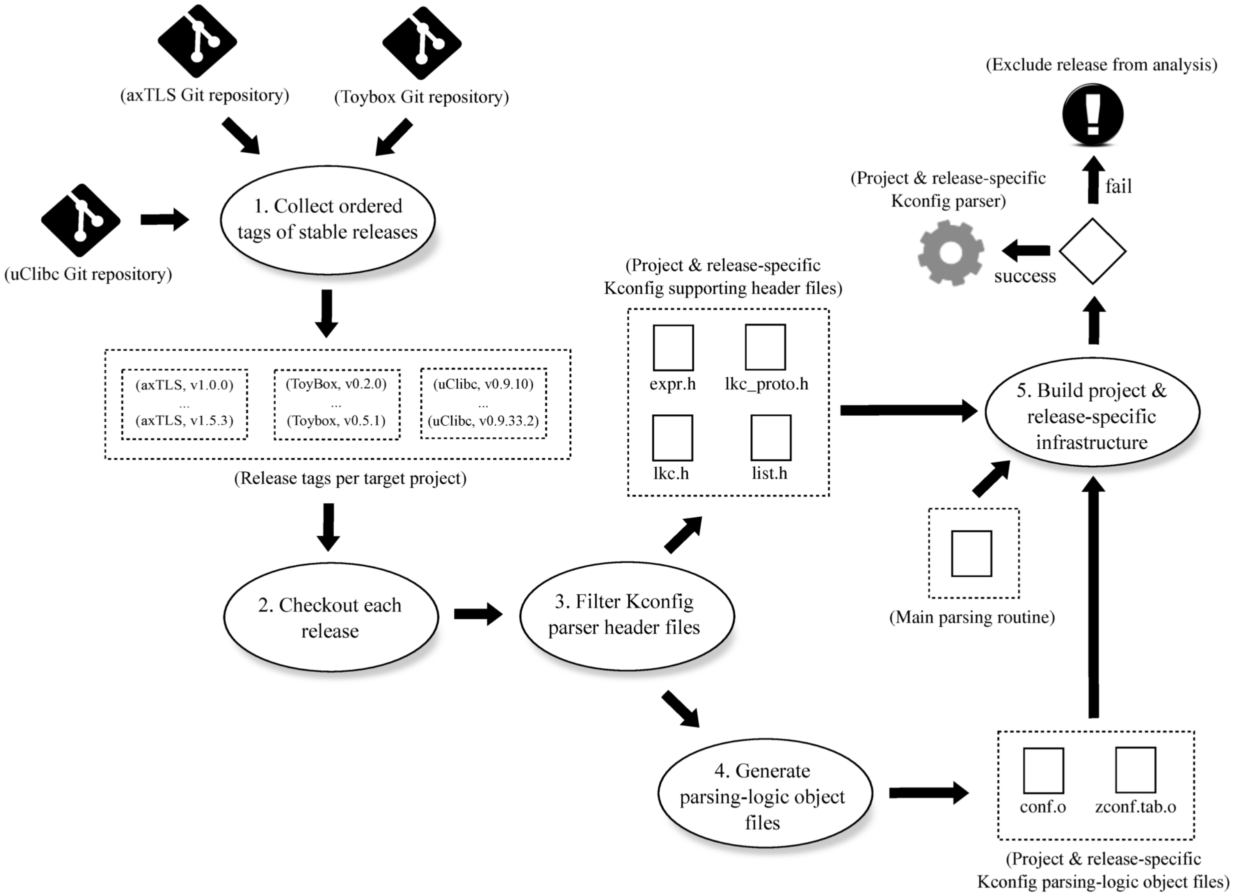

(axTLS Git repository)
(Toybox Git repository)
(uClibc Git repository)
1. Collect ordered tags of stable releases
(axTLS, v1.0.0)
...
(axTLS, v1.5.3)
(ToyBox, v0.2.0)
...
(Toybox, v0.5.1)
(uClibc, v0.9.10)
...
(uClibc, v0.9.33.2)
(Release tags per target project)
2. Checkout each release
3. Filter Kconfig parser header files
(Project & release-specific Kconfig supporting header files)
expr.h
lkc_proto.h
lkc.h
list.h
4. Generate parsing-logic object files
conf.o
zconf.tab.o
(Project & release-specific Kconfig parsing-logic object files)
(Main parsing routine)
5. Build project & release-specific infrastructure
success
fail
(Project & release-specific Kconfig parser)
(Exclude release from analysis)

Figure 4.1: Handling the evolution of the Kconfig language

a laborious and error-prone activity; (c) the Kconfig Yacc grammar specification is full of semantic actions, whose implementation varies across different versions of the language. Thus, language constructs may have conflicting behaviour across their evolution. Consequently, semantic unification is far from trivial.

The alternative solution is to create many parsers, one for each release of Kconfig. We favor such approach, as we are able to fully automate it. Figure 4.1 depicts our solution. Given the target repositories, we list the stable releases of each subject, sorting them in chronological order (step 1). Table 4.5 summarizes how we identify and sort stable releases. Following the sorting output, we checkout each release (step 2). Within a release snapshot, we locate its supporting Kconfig infrastructure, filtering the header files exporting the Kconfig API (step 3). There exists at most four headers across all releases in our three

Table 4.6: Releases within our scope of analysis

System	Stable releases			
	All	Supporting Kconfig	Releases in our scope	Coverage (%)
axTLS	v1.0.0–v1.5.3 (20)	v1.0.0–v1.5.3 (20)	v1.0.0–v1.5.3 (20)	100
ToyBox	v0.2.0–v0.5.1 (10)	v0.2.0–v0.5.1 (10)	v0.2.0–v0.5.1 (10)	100
uClibc	v0.9.10–v0.9.33.2 (41)	v0.9.16–v0.9.33.2 (35)	v0.9.16–v0.9.33.2 (35)	85

target projects. This follows from the fact that the Linux kernel implements its Kconfig infrastructure in a limited number of files; moreover, their names do not change as the kernel evolves. As our subjects clone the Linux kernel Kconfig infrastructure and preserve their filenames, we can promptly identify Kconfig-related source files. From the tracing of the Kconfig API, we set the header inclusion path of the GNU C compiler accordingly and invoke the subject's build system to compile the command-line conf configurator. There are different ways to generate conf—e.g., by invoking make allyesconfig, which creates a configuration attempting to select as many features as possible. Building conf generates two specific object files: conf.o and zconf.tab.o (step 4). The first object (conf.o) contains configuration logic; the second (zconf.tab.o) implements parsing routines. Altogether, they realize the functions in the four Kconfig header files. We link the two object files with a custom-made main parsing function (step 5), producing a release and project-specific parser matching the same release as those of conf.o and zconf.tab.o. Our custom-made parsing function calls a stable subset of the Kconfig API, which we determine by experimenting with different API releases across the Linux kernel evolution.

To facilitate usage, we hide all project and release-specific parsers under a façade. Given a target project and release of interest, the façade delegates parsing to the corresponding release-specific implementation.

With our multiple-parser approach in place, we scope analyses to all releases for which we successfully generate a parser. As Table 4.6 shows, our scope covers all 20 and 10 stable releases of axTLS and ToyBox, respectively. In the case of uClibc, we support all releases from v0.9.16, the first uClibc release adopting Kconfig—all six versions (v0.9.10–v0.9.15) preceding v0.9.16 are not in our scope. In total, given the set of releases of out target systems, our investigation spans a period of over 20 years of evolution (see Table 4.7).

Table 4.7: Release time ranges

	Release range in our scope	Time range
axTLS	v1.0.0–v1.5.1	01 Jul/2006–19 Nov/2014
ToyBox	v0.2.0–0.5.1	12 Feb/2012–19 Nov/2014
uClibc	v0.9.16–0.9.33.2	09 Nov/2002–15 May/2012
	Total (years)	20.42

Our multiple-parser-based infrastructure overcomes the parsing restrictions of our Linux kernel analysis, which employs a single parser approach. With the new infrastructure, we can now parse Kconfig models in all releases from v2.6.12, the first kernel release under Git, up to v3.9, the last stable release at the time of our experiments in Chapter 3.

4.2.4 Data Collection

Data collection follows the same procedure as in the Linux kernel case (see Section 3.4.2). In summary, given a target subject, we iterate over its sorted pairs of consecutive stable releases; for each pair (r_i, r_{i+1}), we calculate the feature set difference of the two releases, identifying new feature names along with removed ones. The union of all added features defines a subject's additions population; likewise, its removals population consists of all removed feature names between each consecutive stable release pair. Table 4.8 shows the size of each subject population. As it happens with the Linux kernel, the additions population dominates removals; overall, we find five times more additions than removal cases.

As in the Linux kernel case, we also create a database from the parsing of commit patches in the evolution history of each target subject. The database stores metadata of all commit patches, along with the changes adding and/or removing primary features at specific release pairs. The database allows us to later query the primary commit of a primary feature.

Different from axTLS and uClibc, the database for Toybox is incomplete. Although the ToyBox database captures all primary features (cases of feature additions or removals), it does not store all associated primary commits. Missing data follows from the fact that a large portion of the ToyBox variability model is dynamic, i.e., some features are generated when building the system configurator. Thus, parsing changes in commit patches adding or

Table 4.8: Feature additions and removals population of each target subject

System	Subject's population size		Total
	Feature additions	Feature removals	
axTLS	35	16	51
Toybox	199	34	233
uClibc	287	53	340
Total	521	103	624

removing features only accounts for the features in the static part of the ToyBox variability model; all dynamically generated features lack a corresponding primary commit in the database. To overcome such limitation, we devise a mechanism for retrieving the primary commits of dynamic features. To understand it, we first explain the ToyBox build system.

In ToyBox, the main unit of extension is a *toy*, i.e., a command-line utility implementation. A toy is a single C file in the toys subsystem;[5] each C-toy file is named after the toy it implements. As an example, consider the ls toy. Figure 4.2 shows its overall structure. As lines 4–27 show, each toy contains a header block comment declaring all its configuration options. When invoking the ToyBox configurator (e.g., make menuconfig), the build system iterates over all C files in the toys subsystem, collecting all configuration options in all header comments; the build script then generates a new Kconfig file (generated/Config.in) containing all collected configuration options. The root Kconfig file, in turn, includes the generated one. When doing so, Toybox places toys in the variability model by following their filesystem location. For instance, as the implementation of ls is under the posix folder in toys, Toybox generates a Posix commands menu entry, adding config LS as its child, along with all other toys in the posix folder.

In addition to toys, Toybox also supports *probe-related features*, which result from the probing of the underlying compilation environment. Toybox uses probing to check support for specific functions, header files, flags, etc. For each probe, the build system generates a corresponding probe feature, whose default value stores whether the probe succeeds or fails. The fragment of the Toybox build script in Figure 4.3 demonstrates such a process. Probing results from calling the probeconfig function; from its output, the build system creates the generated/Config.probed Kconfig file (line 26). Executing probeconfig sequentially

[5] The toy subsystem maps to a folder with the same name under the root directory of the ToyBox source code.

```
toys/posix/ls.c

/* ls.c - list files
 *
 * Copyright 2012 Andre Renaud <andre@bluewatersys.com>
 * Copyright 2012 Rob Landley <rob@landley.net>
...
USE_LS(NEWTOY(ls, ...))

config LS
  bool "ls"
  default y
  help
    usage: ls [-ACFHLRSacdfiklmnpqrstux1] [directory...]
    list files
    ...
    output formats:
    -m comma separated
    -o like -l but no group

    sorting (default is alphabetical):
    -f unsorted
    ...


*/

void ls_main(void)
{
  char **s, *noargs[] = ".", 0;
  ...
  if (CFG_TOYBOX_FREE) free(TT.files);
}
```

Figure 4.2: ToyBox: patch adding ls

```
1
2  scripts/genconfig.sh
3
4  probesymbol()
5  {
6    $CROSS_COMPILE$CC $CFLAGS -xc -o /dev/null $2 - 2>/dev/null
7    [ $? -eq 0 ] && DEFAULT=y || DEFAULT=n
8    rm a.out 2>/dev/null
9    echo -e "config $1\n\tbool" || exit 1
10   echo -e "\tdefault $DEFAULT\n" || exit 1
11 }
12
13 probeconfig()
14 {
15   # Probe for container support on target
16   probesymbol TOYBOX_CONTAINER << EOF
17     #include <linux/sched.h>
18     int x=CLONE_NEWNS|CLONE_NEWUTS|CLONE_NEWIPC|CLONE_NEWNET;
19
20     int main(int argc, char *argv[])  return unshare(x);
21
22 EOF
23 ...
24 }
25
26 probeconfig > generated/Config.probed || \
27 rm generated/Config.probed
```

Figure 4.3: ToyBox probing example

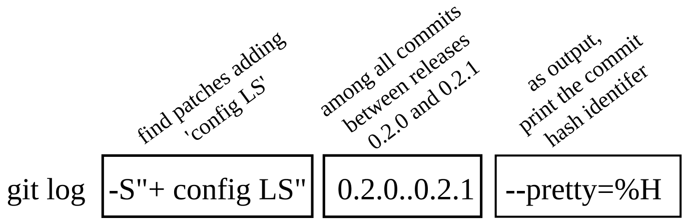
probes specific capabilities—for example, container support. To perform a single probe, probeconfig calls the probesymbol function (line 16), whose implementation is at lines 4–11. When probesymbol executes, it generates an internal configuration option (in our example, TOYBOX_CONTAINER) whose default value matches whether a program argument string (the probing test), as given in lines 17–21, successfully compiles. After probing all the capabilities of interest, the root Kconfig file includes generated/Config.probed.

With the knowledge of how Toybox dynamically generates a subset of its features, we devise a mechanism to retrieve the primary commit of all features in Toybox. Given an added or removed feature between two consecutive releases, we first query the ToyBox database. If the query returns a result, the feature is necessarily part of the static fragment of the Toybox variability model. If absent, we take the feature to be dynamic. To identify the primary commit of a dynamic feature, we use a custom-made tool directly querying Toybox's Git history. Figure 4.4 shows the resulting query when retrieving the primary commit adding the LS feature. Our tool generates the query as follows. From a release pair (r_i, r_{i+1}) associated with a given primary feature, our tool instructs Git to use r_i and r_{i+1} as release boundaries. In our example, the addition of LS associates with the (v0.2.0, v0.2.1) release pair, as LS is in the feature set of v0.2.1, but not in the feature set of v0.2.0; thus; v0.2.1 adds the feature. After defining search boundaries, our tool checks out the release adding or removing the primary feature of interest—in our example, the tool checkouts the v0.2.1 release of Toybox. Next, it determines whether the primary feature is a toy or whether it is a probe-related feature. To be a toy, there must exist a single C file declaring the feature; otherwise, the feature is probe-related—probeconfig generates it. Knowing the correct feature kind, our tool extracts the exact string declaring the feature. For toy features, such string stems from the header block comment in the C file implementing the toy; otherwise, probeconfig contains the string declaring the feature. In our example, LS is a toy, as toys/posix/ls.c declares it. Moreover, the header comment block in ls.c declares the feature as "config LS". From the previous data, our tool builds a Git query to find all commit patches within the release pair boundaries adding the extracted declaration string.

All queries, regardless if they operate on a subject's relational database or on the ToyBox Git repository, may return multiple primary commits. Following the same rules of the Linux kernel case methodology (see Section 3.4.2), we select exactly one primary commit. Thus, the number of primary commits equates to the number of added and removed feature cases. In total, we retrieve 624 primary commits, 521 relative to feature additions and 103 for removed ones. Since some primary commits associate with more than a single primary feature, in total, we retrieve 349 distinct primary commits. Table 4.9 contextualizes such statistics. Our coevolution analysis covers approximately 12 %, 14 %, and 2 % of the entire evolution history of axTLS, Toybox, and uClibc, respectively; in total, our analysis covers 4.4 % of their entire combined evolution.

4.2.5 Pattern Identification Process

At the core of our investigation lies the ability to identify pattern instances. As such, we devise a manual identification process consisting of two steps: *Commit Window Retrieval* and *Pattern Matching*. We describe each step in the following.



```
git log -S"+ config LS" 0.2.0..0.2.1 --pretty=%H
```

find patches adding 'config LS'
among all commits between releases 0.2.0 and 0.2.1
as output, print the commit hash identifer

Figure 4.4: ToyBox: git query to locate the primary commit adding ls

Table 4.9: Commit statistics for our three chosen subjects

System	Nbr. of commits		
	Entire evolution	Adding/removing features	Distinct primary commits
axTLS	205	51	25
ToyBox	1,114	233	161
uClibc	6,620	340	160
Total	7,939	624	346

Commit Window Retrieval. This step follows the same rationale as the *Commit Window Retrieval* when mining coevolution patterns from the Linux kernel Git repository (see Section 3.4.3). There are, however, two main differences:

1. While we follow the same expansion rules E_1–E_4 as in the Linux kernel case, we do not forbid including commits whose label differs from the primary one. Our decision follows from the fact that developers in axTLS and Toybox do not label commit messages.

2. When retrieving the commit window of a primary feature, we always limit the number of commits to at most 40. This contrasts to the commit windows of feature additions in the Linux kernel, which we allow to grow as much as needed. Our choice of at most 40 commits follows from our past experience of analyzing patches adding or removing features to the Linux kernel; generally, we find commit windows to have a single commit. Our chosen subjects strengthens such understanding. As Figure 4.5 shows, commit windows in axTLS, ToyBox, or uClibc have an average size of 1.24 commits. Within additions (521), only 6 % (32) of commit windows have size greater

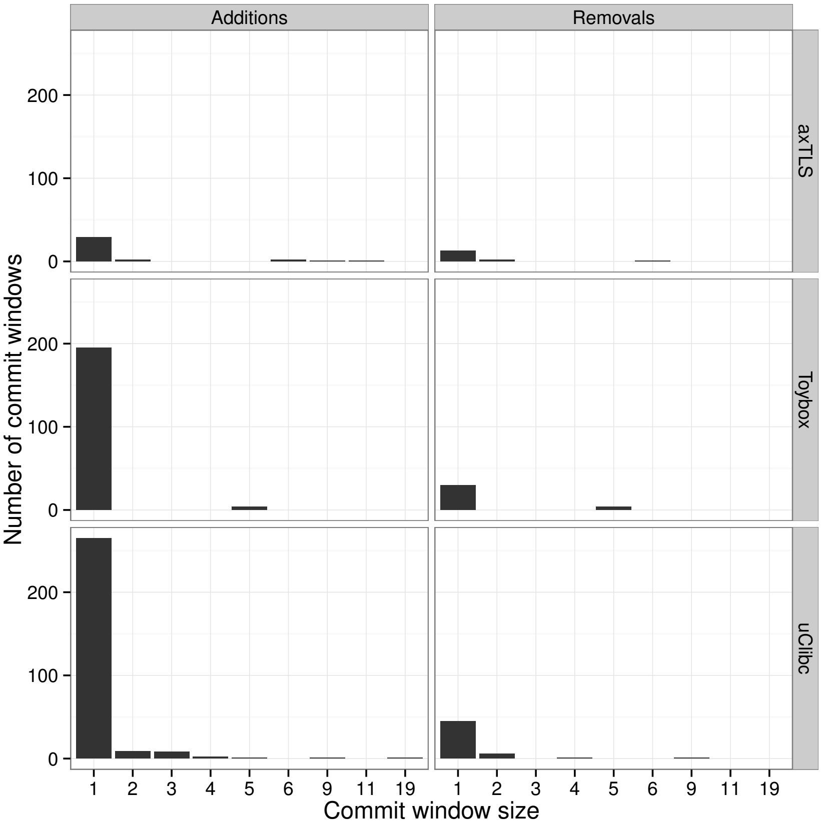


Figure 4.5: Commit window sizes of axTLS, ToyBox, and uClibc

than one; among those, most commit windows (19) have size between two and three. Large commit windows are rare; a single commit window of size 19 exists in uClibc—less than half of our 40-commit window size limit. Removals also have small commit windows, generally of size one (overall mean is 1.4).

In total, we retrieve 624 commit windows, one for each primary feature. Accounting all the commits within the target commit windows leads to a total of 771 commits—628 relate to feature additions, whereas 143 concern feature removals.

Pattern Matching. After determining the commit window of a primary feature, we attempt to derive a before and after-state of its changes, from which we set to match against the before and after states of all applicable patterns in our catalog, seeking the

Table 4.10: Number of excluded addition and removal cases

System	Excluded cases		Total
	Feature additions	Feature removals	
axTLS	0 (0 %)	0 (0 %)	0 (0 %)
Toybox	4 (2 %)	10 (29.4 %)	14 (6 %)
uClibc	20 (7 %)	1 (1.9 %)	21 (6.2 %)
Total	24 (4.6%)	11 (10.7%)	35 (5.6 %)

best fit. There are times, however, that we cannot understand the change in place, and consequently, we fail to derive a before and after states. In such a case, we exclude the given commit window from further analysis. Table 4.10 shows the number of commit windows we exclude. Exclusion ranges from 0 % (axTLS) to 29.4 % (uClibc—removals). Overall, we exclude 5.6 % (35) of all target cases; 4.6 % (24) of all additions and 10.7 % (11) of all feature removals. Thus, we are left with 589 matches to perform.

When choosing applicable matching patterns, we carefully control possible overlaps. As a driving example, consider a change where developers add a new feature (our primary source of investigation) to the variability model; at the Makefile level, developers reassign a compilation unit of an existing feature to the newly created one; developers also associate new compilation flags to the newly added feature. The latter also controls new code fragments through *ifdefs* in code. Given such a description, which pattern best applies? AVONMF, AVOCFF, AVONMCFF, or FCUTVOF?

To select a single pattern representing a change, we introduce a priority-based matching scheme. Figures 4.6 and 4.7 depicts the scheme for the additions and removal-based patterns, respectively.

Our scheme first groups applicable patterns according to the spaces that the target change affects w.r.t a primary feature. Determining such spaces allows us to select a single group of possible matching patterns. In our example, the changes that relate to the primary feature affect the variability model, the mapping, and the source code. Thus, we select (VM + M + C) as the target group, ruling out AVONMF and AVOCFF as prospective matches—such patterns belong to other distinct groups.

Each group partitions its patterns in subgroups, which we represent as stacks. Each subgroup has one or more patterns sharing similar characteristics, such as whether the primary feature under analysis is added or removed, its kind (optional, mandatory, or

Table 4.11: Correspondence between participants in the Linux kernel case study and the ones validating our catalog of patterns

Mining patterns from the Linux Git repository (Chapter 3)	Catalog validation (Chapter 4)
P_1	P_1
P_3	P_3
P_4	P_2

computed) and visibility (visible or internal), whether the primary feature has associated code elements, whether it has associated compilation units, etc. Given a target group, we select the stack whose patterns contain the highest amount of overlapping characteristics with the target change and its affected spaces. After determining a stack, we attempt to match the before and after states of the change to the corresponding states of each pattern in the stack. Determining the before and after states of a given change follows the same rationale as in the Linux kernel case. Then, matching starts from the top of the stack towards its bottom; in such traversal, we sequentially test each pattern in the stack until we find a match or we reach past the bottom. In the latter case, we report NA, meaning that no pattern applies. Otherwise, we stop as soon as we find the first match, ignoring elements in the change that could potentially match the structure of patterns lower in the stack—such elements are irrelevant to the matched pattern. The relationship between what is central to a pattern and what is ignorable is a key principle of our scheme and the associated stacks within each group. In our example, after determining (VM + M + C) as the target group, we find that its left most stack contains the patterns that our change overlaps the most. When iterating over the target stack, the states of the pattern at the top (FCUTVOF) matches the states of our example change; consequently, we do not test other patterns lower in the stack, ruling out AVONMCFF as a matching pattern.

4.2.6 Manual Identification of Patterns

Pattern identification is a manual process. As such, three participants (P_1–P_3) apply our process—all three accumulate the experience of also participating in the Linux kernel case study, which we report in Chapter 3. Table 4.11 shows the correspondence between the participants in the two studies.

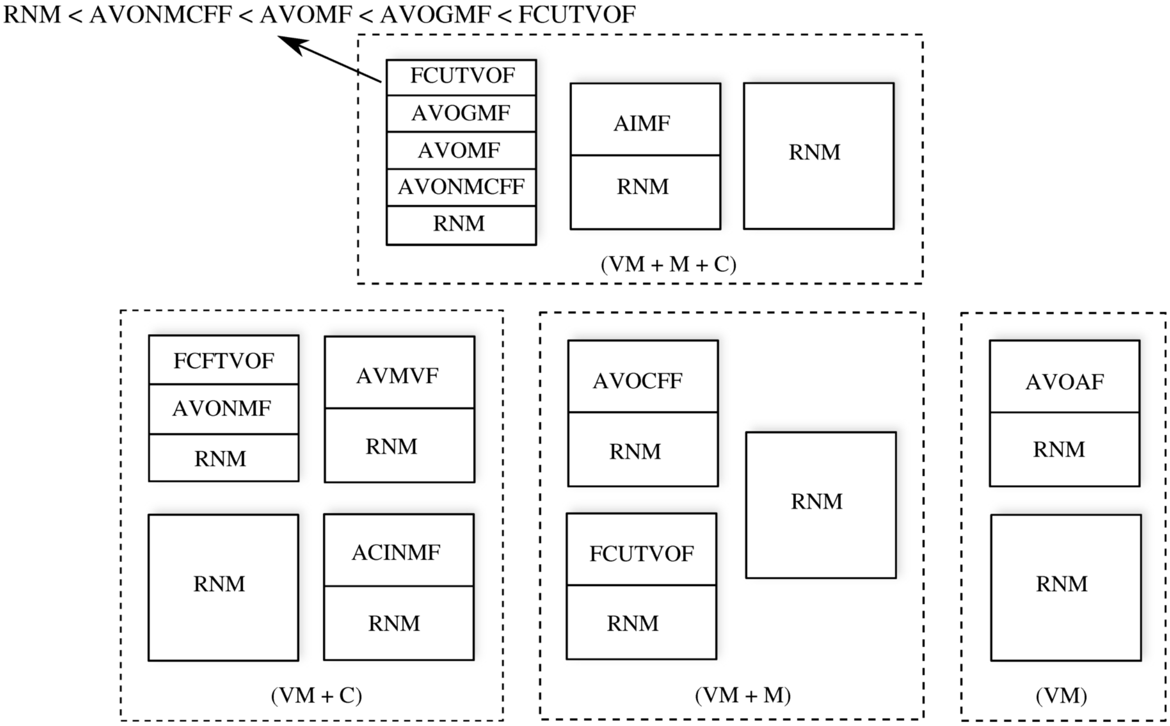


Figure 4.6: Pattern matching priorities (additions)

4.2.7 Review Process

To mitigate human error and possible subjectivity when applying our identification process, we perform reviews of all cases we analyze. Whenever one participant retrieves a commit window, matches an addition or removal case against a coevolution pattern, mark the case as NA, or exclude it, the other two participants perform a review to assure the correctness of the results. If a participant reports an inconsistency during review, a consensus phase follows. During consensus, all three participants discuss the given inconsistency, arguing in favor of the current result or proposing another one—e.g., expansion or contraction of the given commit window, an alternative matching pattern or NA, or even the exclusion of the case under analysis. After we discuss and reach a consensus of all inconsistencies that reviewers raise, the matches we report capture the common understanding of P_1, P_2, and P_3. Tables 4.12 and 4.13 show the tasks of each participant. Each participant's role follows directly from his/her availability at the time of our analyses. Participants P_2 and P_1 identify matching patterns for all feature additions and removals cases in axTLS and Toybox, respectively. In the analyses of uClibc, two participants analyze its feature

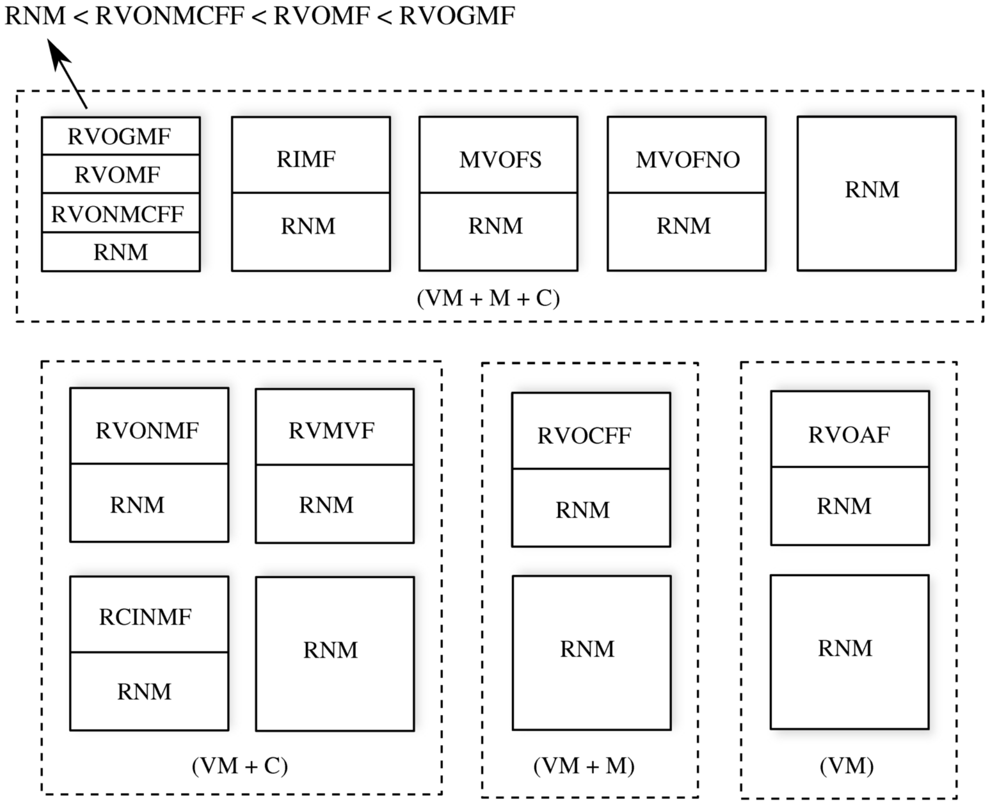


Figure 4.7: Pattern matching priorities (removals)

additions: P_2 identifies matching patterns for the first 144 cases, whereas P_1 takes over the remaining ones; for all the feature removals in uClibc, P_3 identifies matching patterns.

Review and consensus play an important role in our methodology. Next, we discuss the number of inconsistencies when retrieving commit windows and when matching patterns.

Review of Commit Windows. Although most commit windows tend to contain only the primary commit of the feature addition or removal case under analysis, determining commit windows is not free of errors. Table 4.14 shows the number of the inconsistencies we find when reviewing all commit windows. The table also presents percentage values, which we calculate by taking each cell value and dividing it by the corresponding value in Table 4.8—henceforth, all tables alike follow the same rationale. In general, our reviews find 4.3 % (27) of the commit windows to be inconsistent, which we fix during consensus. Percentage-wise, most inconsistencies relate to removal cases in axTLS and Toybox. In

Table 4.12: Activities performed by each participant in the analyses of axTLS and Toybox (I: pattern identification; R(P): review of the patterns that P identifies)

Participant	axTLS		Toybox	
	Additions (35)	Removals (16)	Additions (199)	Removals (34)
P_1	$R(P_2)$	$R(P_2)$	I	I
P_2	I	I	$R(P_1)$	$R(P_1)$
P_3	$R(P_2)$	$R(P_2)$	$R(P_1)$	$R(P_1)$

axTLS, four commit windows are incorrect; they replicate the same error, as all four share the same primary commit. The inconsistency follows from the fact that two commits add the same primary feature; while one relates to the release pair under analysis, the other does not. The latter, however, was chosen as the correct commit by the participant applying our identification process; during review, others raised awareness of such a mistake, leading to four fixes during consensus. Reviewing the results from Toybox led to an increase of four commit windows, all sharing the same primary commit. Instead of containing only the primary commit, during consensus, participants agreed to add four other commits to aid the overall understanding of the change under analysis.

Review of Pattern Matches. The importance of reviews becomes even more evident in the matching part of our process. As Table 4.15 shows, reviews identify inconsistencies in almost a fourth of all feature addition and removal cases. Proportionally, axTLS has the highest amount of inconsistencies (57 %), followed by uClibc (26 %). Higher inconsistency percentages occur in systems with more complex structures, which often brings noise when manually analyzing commit patches. Opposed to axTLS and uClibc, Toybox displays considerably lower inconsistency percentages. To us, this follows directly from the project's architecture controlling how developers add and remove features—toys and their associated capabilities and parameters. As Figure 4.8 shows, throughout the Toybox evolution, toys are the dominant feature kind; on average, 74.5 % of all features in Toybox are toys (min = 63.1 %, max = 81.4 %). Since a toy is a modular feature whose implementation lies in a single C file in the toys subsystem, adding and removing toys becomes a matter of either adding or removing a C file. This agrees with the simplicity by design in Toybox; its head maintainer claims that the Toybox implementation should be understood by those just starting to learn the C programming language. Simplicity sure contributes

Table 4.13: Activities performed by each participant in the analyses of uClibc (I: pattern identification; R(P): review of the patterns that P identifies)

Participant	Additions		Removals
	(1–144)	(145–287)	(53)
P_1	R(P_2)	I	R(P_3)
P_2	I	R(P_1)	R(P_3)
P_3	R(P_2)	R(P_1)	I

Table 4.14: Commit window inconsistencies stemming from participants' reviews

System	Feature additions	Feature removals	Total
axTLS	2 (5.7 %)	4 (25 %)	6 (11.8 %)
Toybox	4 (2 %)	4 (11.8 %)	8 (3.4 %)
uClibc	8 (2.8 %)	5 (9.4 %)	13 (3.8 %)
Total	14 (2.7 %)	13 (12.6 %)	27 (4.3 %)

Table 4.15: Pattern identification inconsistencies stemming from participants' reviews

System	Feature additions	Feature removals	Total
axTLS	22 (62.9 %)	7 (43.8 %)	29 (56.9 %)
Toybox	2 (1 %)	4 (11.8 %)	6 (2.6 %)
uClibc	72 (25.1 %)	16 (30.2 %)	88 (25.9 %)
Total	96 (18.4 %)	27 (26.2 %)	123 (19.7 %)

to our understanding, justifying the lower inconsistency percentages in our analysis. In contrast, axTLS and uClibc do not impose how developers should add or remove features, making our analyses more subject to mistakes.

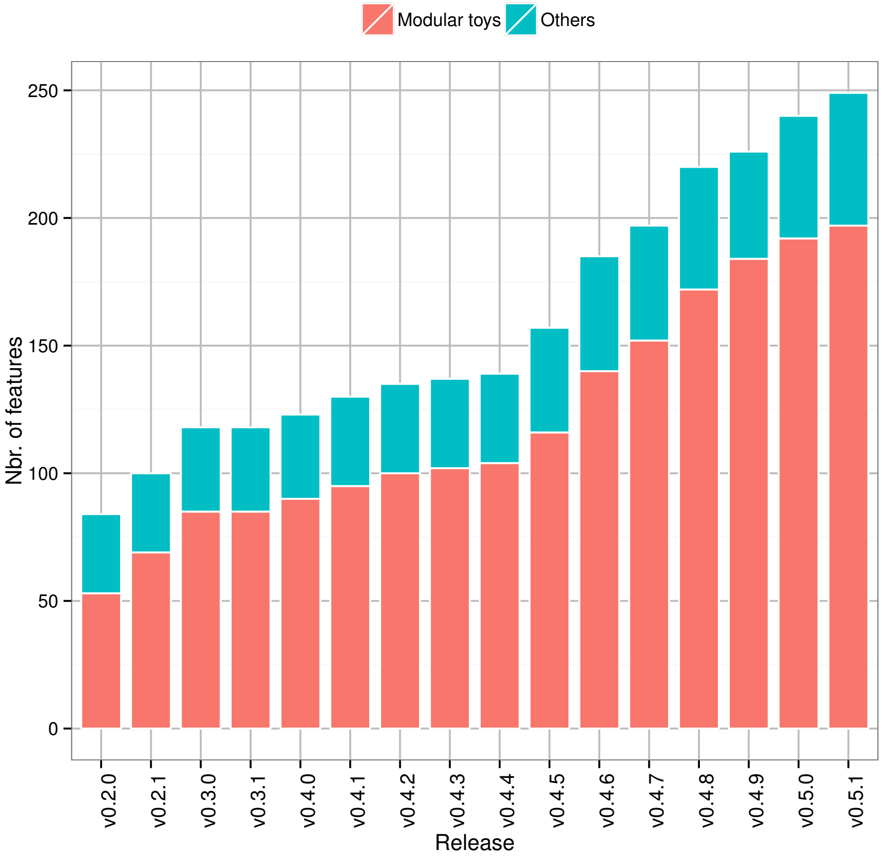


Figure 4.8: Amount of toy features against other kinds

4.2.8 Generality Inference

When we find patterns that do not conform to our generality constraints, we set to infer general patterns. Similar to the Linux kernel case, if a pattern does not satisfy either of our three generality conditions (see Section 4.1), we apply two inference rules:

IG1 There exists a general addition pattern, but its inverse removal pattern is not general. From the fact that every added feature should be eventually removed, and that such removal can be accomplished by following the opposite steps when adding the feature, we take the inverse of any general addition pattern to be an inferred general removal case if the latter has not yet been claimed to be general, nor inferred.

IG2 There exists a general removal pattern, but its inverse addition pattern is not general. From the rationale that one removes a feature only after adding it, and that such addition can be achieved by following the inverse steps of its removing pattern, we take the inverse of any general removal pattern to be an inferred general addition case if the latter has not yet been claimed to be general, nor inferred.

4.3 Results

This section presents the validation results of our catalog of variability-coevolution patterns. Next, we discuss the expressiveness of our catalog (RQ.1), followed by a discussion of its generality (RQ.2).

4.3.1 Catalog Expressiveness

RQ. 1 Is our catalog of patterns expressive enough to capture the variability coevolution induced by the addition or removal of features in the variability models of existing variability-aware sytems?

Overall, we find our catalog to be expressive. As Table 4.16 shows, our catalog covers two thirds (64 %) of all feature additions and removal cases we analyze. Toybox highly influences our expressiveness—of all the cases we match against patterns in our catalog (402), 42.5 % stem from feature additions in Toybox. In the latter case, there is a dominance of AVOMF instances, which relate to the addition of toys, the most common feature kind in Toybox (see Figure 4.8). Moreover, as Figure 4.9 shows, the feature additions and removal cases from Toybox matching a pattern in our catalog total to a handful of patterns. Nonetheless, they explain as much as 80 % of all feature addition and removal cases in the evolution history of Toybox. In contrast, the cases we analyze in axTLS and uClibc match 11 and 13 patterns, respectively. Percentage-wise, these patterns cover around 55 % of all the cases we analyze in axTLS and uClibc. Different reasons explain such differences.

- In Toybox, developers do not interact with the build system; the mapping between a toy and its implementation occurs by naming a C file after the toy it implements. Consequently, developers do not create guard features nor introduce compilation flags, which explains the absence of any patterns of the like—e.g., AVOGMF, AVOCFF, AVONMCFF, and their matching removal patterns. The opposite occurs with axTLS and uClibc.

Table 4.16: Matching cases among all additions and removals

System	Matching cases		Total
	Feature additions	Feature removals	
axTLS	18 (51.4 %)	10 (62.5 %)	28 (54.9 %)
Toybox	171 (85.9 %)	16 (47.1 %)	187 (80.3 %)
uClibc	158 (55.1 %)	29 (54.7 %)	187 (55.0 %)
Total	347 (66.6 %)	55 (53.4 %)	402 (64.4 %)

- As we show in Figure 4.8, toys are the dominant feature kind in Toybox. Moreover, Toybox's architecture does not allow developers to control where they place toys, except that it has to be under the `toys` subsystem—its location within determines the placement of a toy in the Toybox variability model. Consequently, developers do not add abstract features. Since toys are always concrete, we do not find any instances of AVOAF in the evolution history of Toybox. In contrast, axTLS and uClibc do not enforce where features should be placed, providing developers with greater freedom to change the variability model hierarchy. Thus, developers may add intermediate nodes to better structure the hierarchy of features, which explains the AVOAF instances we find in axTLS and uClibc.

- We do not find featurization pattern instances in Toybox. In axTLS, we find a single FCFTVOF instance. Among NA cases, there is one featurization in Toybox and three others in axTLS, but none fits a pattern. This contrasts with uClibc, where we find 33 FCFTVOF instances. Such results suggest that memory footprint is a concern in uClibc, while less so in the other two projects. By featurizing code fragments, uClibc developers assure that previously unconditional code snippets can only compile upon specific feature selections, decreasing the footprint of a subset of variants (those that do not activate the featurized code pieces). This agrees with the project's objective, as stated in the official website of uClibc:

> "*If you are building an embedded Linux system and you find that glibc is eating up too much space, you should consider using uClibc*"

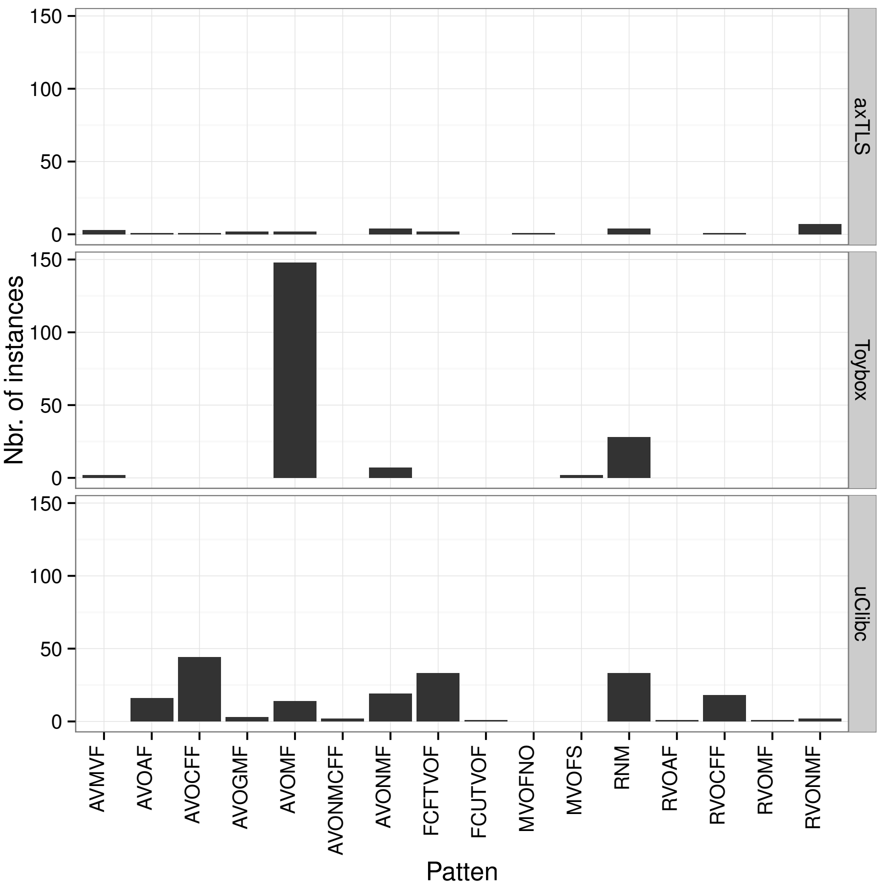


Figure 4.9: Pattern matching results

4.3.2 Pattern Generality

RQ. 2 From our catalog, what variability-coevolution patterns are general?

Combining the results of the Linux kernel case with the counting of pattern instances we identify in axTLS, Toybox, and uClibc, we check what patterns are general, i.e., patterns with three or more instances, whose application occurs in at least three systems, and by three or more distinct developers. Table 4.17 shows our results. In the table, columns 2–5, which we place under *Instances*, display the number of pattern instances in each system (Linux included), while column *Developers* shows the number of distinct developers applying each pattern. If we infer a pattern, we mark *Yes* in its *Inferred* column cell, or *No* otherwise. Similarly, the *General* column states whether a pattern satisfies our three generality conditions, in the case of non-inferred patterns, or whether we infer it otherwise.

Among our 23 patterns, nine fully satisfy our generality constraints (AVOMF, AVOGMF, AVONMF, AVOCFF, AVOAF, AVMVF, FCFTVOF, RNM, and RVONMF); five others stem from inference (RVOMF, RVOGMF, RVOCFF, RVOAF, and RVMVF)—while they do not fully comply to our generality conditions, their inverse patterns do. In total, we claim the generality of 14 patterns, which corresponds to 61 % of our initial catalog of Linux kernel coevolution patterns.

Note that we do not prove the generality of our merge patterns. In total, we find five merge cases, excluded those from Linux. From the five cases we find, we match three; the remaining two we mark as NA. This contrasts to what we observe in Linux, where we find 27 merge situations—6 fitting patterns. Since merges seek to remove redundancy between overlapping features, our results suggest that the features in each of our three target subjects have little redundancy.

4.4 Discussion

When analyzing commit windows and identifying pattern matches in the evolution history of axTLS, Toybox, and uClibc, we observe some characteristics that further contribute to a general understanding of how developers apply the patterns of our catalog. We discuss such characteristics in the following. Whenever possible, we link our findings to those of the Linux kernel case (Section 3.6).

Table 4.17: Counting of pattern instances

	Pattern	Instances				Developers	Inferred?	General?
		Linux	axTLS	Toybox	uClibc			
1	AVOMF	124	2	148	14	142	No	Yes
2	AVOGMF	11	2	0	3	13	No	Yes
3	AVONMF	32	4	7	19	38	No	Yes
4	AVOCFF	4	1	0	44	16	No	Yes
5	AVONMCFF	3	0	0	2	5	No	No
6	AVOAF	2	1	0	16	8	No	Yes
7	AVMVF	3	3	2	0	5	No	Yes
8	AIMF	12	0	0	0	11	No	No
9	ACINMF	2	0	0	0	2	No	No
10	FCFTVOF	4	2	0	33	12	No	Yes
11	FCUTVOF	10	0	0	1	4	No	No
12	RNM	29	4	28	33	26	No	Yes
13	RVOMF	22	0	0	1	18	Yes	Yes
14	RVOGMF	12	0	0	0	4	Yes	Yes
15	RVONMF	10	7	0	2	10	No	Yes
16	RVOCFF	0	1	0	18	2	Yes	Yes
17	RVONMCFF	0	0	0	0	0	No	No
18	RVOAF	6	0	0	1	7	Yes	Yes
19	RVMVF	3	0	0	0	3	Yes	Yes
20	RIMF	3	0	0	0	3	No	No
21	RCINMF	3	0	0	0	3	No	No
22	MVOFNO	3	1	0	0	4	No	No
23	MVOFS	3	0	2	0	5	No	No

4.4.1 Different Systems May Realize Patterns with Different Mechanisms

Across our analyses, we note that subjects realize some of our patterns with different mechanisms. In particular, we note differences in how developers apply AVOMF, AVONMF, and AVOGMF.

Modular Features May Lack an Explicit Mapping. Different from the other systems, Toybox does not rely on an explicit mapping between features and compilation units. Instead, developers follow specific conventions, which the build mechanism enforces when compiling target variants (e.g., a C file must be named after the toy it implements).

If-Statements are an Alternative to Ifdefs. In Toybox, we also note a difference in how developers write annotations. Instead of *ifdefs*, developers favor if-statements.[6] Figure 4.10 provides an example from the LSPCI feature. In the implementation of the do_lspci function, the if-statement at line 5 conditions the following block (lines 6–7) to the presence of LSPCI_TEXT. Conditional compilation works as follows. In Toybox, each Boolean feature that is not selected during configuration results in a `#define` setting a corresponding feature macro identifier to zero, or one otherwise (Toybox does not have tristate features). Pre-processing then replaces feature macro identifiers with their corresponding value. All if-conditions replacing *ifdefs* have a special format—they are either a reference to a single feature identifier or they are a conjunction of an expression over feature macro names with another Boolean expression (as in the example). After pre-processing selection statements of the kind, the compiler partially evaluates their conditions. Those which turn into zero or conjunct with it necessarily guard dead code, which the compiler optimizer removes from the final binary. In our example, if LSPCI_TEXT is not in the final configuration, its corresponding macro name CFG_LSPCI_TEXT is set to zero. Thus, pre-processing the condition at line 5 results in

```
0 && (TT.numeric != 1)
```

Since the partial evaluation of the above expression results in false,[7] the compiler optimizer removes the entire block that our example condition enframes.

We clearly see advantages of replacing *ifdefs* with if-statements:

- The use of if-statements do not impose special variability-aware parsing to check the underlying code. This is particularly useful for performing automated refactorings.

[6] Given such a preference, when calculating the scattering degree of a feature in Toybox, we account both the number of *ifdefs* and if-statements referring to the given feature. For all other systems, we account only *ifdefs*.

[7] In C, the literal 0 in a Boolean expression has the same semantics as the false Boolean literal.

```
toys/pending/lspci.c

int do_lspci(struct dirtree *new)
  ...
  if (CFG_LSPCI_TEXT && (TT.numeric != 1)) {
     res = find_in_db(bufs->vendor, bufs->device, TT.db,
                      bufs->vname, bufs->devname);
  }
  ...
  return 0 ;
}
```

Figure 4.10: Selection statement as a replacement for `#ifdef`

- If-statements allow the compiler to identify syntactic and semantic errors in annotated fragments. The same fragments, if guarded by *ifdefs*, could hide errors that would only be reported when compiling specific variants.
- Using if-statements forces developers from writing *ifdefs* at the level of statements and expressions, which are undisciplined. At the cost of possibly introducing duplicate code, developers can easily rewrite undisciplined annotations in any control-flow sequence as if-statements. If duplicate code arises, developers may refactor it (e.g., by extracting a function), increasing the overall implementation quality.

While if-statements have clear advantages over *ifdefs*, they are not universally applicable. Rather, they can only annotate statements that are part of a control-flow, which is not the case with elements at the global and type levels. For the latter, *ifdefs* are a good-fit.

Different Makefile Constructs Support AVOGMF. In case of AVOGMF instances, axTLS and uClibc rely on similar, but different mechanisms than those we observe in Linux.

At the Makefile level, axTLS developers use `ifdef` conditionals over feature names, guarding recursive calls to make to process specific target directories. In the latter case, developers set the -C flag, instructing make to change to a target directory before further processing (see Figure 4.11 for an example).

In uClibc, developers also use Makefile conditionals, but rely on Makefile variables to store target directories. As an example, consider the addition of the UCLIBC_HAS_WORDEXP

```

 RELEASE=axTLS-$(VERSION)

 # standard version
 target:
         $(MAKE) -C crypto
         $(MAKE) -C ssl
 ...
+ifdef CONFIG_AXTLSWRAP
+        $(MAKE) -C axtlswrap
+endif
```

Figure 4.11: AVOGMF instance using a Makefile conditional (axTLS)

```
1  ...
2 +ifeq ($(strip $(UCLIBC_HAS_WORDEXP)),y)
3 +DIRS += wordexp
4 +endif
```

Figure 4.12: AVOGMF instance using a Makefile conditionals and a directory variable (uClibc)

feature.[8] As Figure 4.12 shows, when coevolving the Makefile, developers add an ifeq-condition at line 2. Its evaluation first trims the value of UCLIBC_HAS_WORDEXP, comparing it with y. If the comparison holds, make appends wordexp to DIRS, which stores target directories to proceed with compilation. Different from the solution in axTLS, setting DIRS in our example works as a lazy approach, as it does not immediately cause make to change the current directory.

4.4.2 Feature Modularity Directly Relates to a Project's Notion of Feature

In our analyses of axTLS, Toybox, and uClibc, we note a great disparity in their pattern frequencies. In particular, we note that axTLS and uClibc do not favor the modular addition of features in the same intensity as Toybox does. As it happens in the Linux kernel case, the majority of features in Toybox denote coarse-grain functionality units (toys), which map to entire C files. Thus, 74 % (148) of the feature additions in Toybox

[8] See commit: 94e3b336

match the AVOMF pattern. In contrast, a third (96) of the additions in uClibc fall under three patterns that do not relate to modular extensions: AVOCFF (44), FCFTVOF (33), and AVONMF (19). In axTLS, the percentage is 26, with a count of 9 instances: AVONMF (4), AVMVF (3), FCFTVOF (2). The percentage of AVOMF instances in the latter two systems is 5 % (14) and 6 % (2), respectively. Adding the count of AVOGMF does not change the prevalence of the dominant patterns.

One could argue that such disparity may result from our manual analyses of commit patches, followed by manually matching them against patterns in our catalog. When doing so, we could have incorrectly labelled authentic AVOMF instances in axTLS and uClibc as NA or excluded them from further analyses, potentially contributing to the differences we observe. Such explanation, however, is unlikely to hold, as we consider AVOMF among the easiest patterns to identify. We are confident that we are not labeling valid AVOMF instances as NA nor excluding them. Instead, we seek a better hypothesis.

Opposed to Toybox, axTLS, and uClibc are libraries. Thus, their main unit of extension consists of adding functions. Adding the latter, however, does not favor modularity, as AVOMF is not dominant in both systems. Wee see three competing explanations:

1. Assuming an alignment between functions and features, files implement more than a single feature (function). Thus, when developers introduce new features adding code, they place the latter inside existing files, or add them together with other new features inside new compilation units.

2. If there is not an alignment between features and functions, features serve the purpose of controlling the behaviour of functions of the target library, such as (de-)activating specific capabilities, setting their parameters, or by affecting the compilation process as a whole.

3. A mix of the previous two cases.

In either case, features in axTLS and uClibc will necessarily differ from those in Toybox in terms of the elements they control (e.g., controlling functions versus entire C files), varying purposes (e.g., changing compilation flags versus switching on/off functionality chunks), or both.

Following the above rationale, lower percentages of modular-related patterns is not necessarily a bad smell. While certainly favoring overall evolution, a high frequency of modular-based patterns is directly affected by the system's notion of a feature and its primary purpose in configuring target variants. If, at the implementation level, developers

Table 4.18: AVOMF instances introducing extensions

System	Instances	Extensions (total)
axTLS	2 (100 %)	9
Toybox	1 (0.7 %)	15
uClibc	5 (35.7 %)	13
Total	8 (4.9 %)	37

use features for configuring the internals of compilation units, as we note in axTLS and uClibc, the amount of non-modular extensions, along with the level that developers add them, seem more relevant as implementation quality criteria than the number of modular-based features per se.

4.4.3 There Appears to Exist a Balance Between the Number of AVOMF Instances Adding Extensions and the Scattering Degree Values of such Features

When inspecting extensions of AVOMF features across header and C implementation files, we note that all the instances in axTLS add extensions; in uClibc, the percentage is lower, but it is not negligible—36 %. Table 4.18 details the results. Altogether, both systems oppose to what we observe in Toybox and in the Linux kernel. Rather than a coincidence, the alignments between Linux+Toybox and axTLS+uClibc are likely to follow from a dominant notion of how these systems perceive features and how they set to utilize them (see Section 4.4.2).

In the combined evolution of axTLS, Toybox, and uClibc, most AVOMF instances adding extensions concern scattered features, i.e., those with $SD > 1$ (see Table 4.19). Moreover, as Table 4.20 shows, scattering degrees tend to be lower in uClibc and axTLS than in Toybox. Such results suggest a balancing force. While Toybox has a single scattered feature, its scattering degree is as high as 15. axTLS and uClibc, in contrast, have more instances adding extensions (7 in total), but their *SD*-values are considerably lower; the median value in axTLS is 4.5, whereas in uClibc it is one.

Table 4.19: (Non-)scattered cases among AVOMF instances introducing extensions

System	*SD* = 1	*SD* > 1
axTLS	0 (0 %)	2 (100 %)
Toybox	0 (0 %)	1 (100 %)
uClibc	3 (60 %)	2 (40 %)
Total	3 (37.5 %)	5 (62.5 %)

Table 4.20: *SD*-values of AVOMF instances introducing extensions

System	*SD*-values		
	Min	Max	Median
axTLS	2	7	4.5
Toybox	15	15	15
uClibc	1	8	1

4.4.4 Most AVONMF Instances Relate to Scattered Features, but with Low SD.

Given AVONMF instances, our three target subjects add a total of 117 extensions: 6 in axTLS, 12 in Toybox, and 99 in uClibc—see Table 4.21.[9] Similar to what we observe in the Linux kernel, 70 % of the AVONMF instances in the combined evolution of axTLS, Toybox, and uClibc concern scattered features (see Table 4.22). Also, the scattering degrees of AVONMF instances are low; as Table 4.23 shows, uClibc has a median *SD* of three, the highest among our three subjects. In Toybox, typical values range between 1–2, whereas in axTLS, a typical AVONMF feature introduces a single extension. The distributions of *SD*-values in axTLS and Toybox are not skewed, as their median value approximates to the mean. In contrast, the mean *SD*-value (5.5) in uClibc is greater than the median, causing right skewness. This is due to five features whose *SD*-values range from 9 to 15.

[9] When analyzing the number of extensions of AVONMF instances, we do not account one instance in uClibc, as its implementation is in Assembly. Thus, as Table 4.21 shows, the total number of uClibc instances sums to 18 (95 % of all the AVONMF features in uClibc). Counting extensions only in C header and implementation files makes our analyses consistent throughout the thesis, as such files types are the most common artifact type across all the subjects we analyze.

Table 4.21: Extensions among AVONMF instances

System	Instances	Extensions (total)
axTLS	4 (100 %)	6
Toybox	7 (100 %)	12
uClibc	18 (95 %)*	99
Total	29 (97 %)	117

Table 4.22: (Non-)scattered cases among AVONMF instances

System	*SD* = 1	*SD* > 1
axTLS	2 (50 %)	2 (50 %)
Toybox	4 (57.1 %)	3 (42.9 %)
uClibc	2 (11.1 %)	16 (88.9 %)
Total	8 (30 %)	21 (70 %)

Table 4.23: *SD*-values of AVONMF instances

System	*SD*-values		
	Min	Max	Median
axTLS	1	2	1.5
Toybox	1	3	1
uClibc	1	15	3

4.4.5 Annotations Tend to be Disciplined

In the analyses of the two most frequent patterns in the Linux kernel, we observe that the extensions of AVOMF and AVONMF are disciplined, i.e., they align with syntactic units of the host programming language.

We note the same behaviour across the evolution of axTLS, Toybox, and uClibc. Tables 4.24 and 4.25 show the level at which developers introduce extensions when adding AVOMF and AVONMF features, respectively. When calculating the percentage values in both

Table 4.24: Syntactic level of extensions among AVOMF instances introducing *ifdefs*

System	Granularity of extensions						
	Global	Function	Type	Block	Statement	Expression	Signature
axTLS	4 (44.5 %)	2 (22.2 %)	2 (22.2 %)	0 (0 %)	1 (11.1 %)	0 (0 %)	0 (0 %)
Toybox	2 (13.3 %)	1 (6.7 %)	0 (0 %)	12 (80 %)	0 (0 %)	0 (0 %)	0 (0 %)
uClibc	4 (30.8 %)	5 (38.4 %)	0 (0 %)	1 (7.7 %)	3 (23.1 %)	0 (0 %)	0 (0 %)
Total	10 (27 %)	8 (21.6 %)	2 (5.4 %)	13 (35.2 %)	4 (10.8 %)	0 (0 %)	0 (0 %)

Table 4.25: Syntactic level of AVONMF extensions

System	Level of extensions						
	Global	Function	Type	Block	Statement	Expression	Signature
axTLS	1 (16.7 %)	2 (33.3 %)	1 (16.7 %)	0 (0 %)	2 (33.3 %)	0 (0 %)	0 (0 %)
Toybox	3 (25 %)	4 (33.3 %)	0 (0 %)	5 (41.7 %)	0 (0 %)	0 (0 %)	0 (0 %)
uClibc	26 (26.3 %)	35 (35.4 %)	3 (3 %)	12 (12.1 %)	12 (12.1 %)	11 (11.1 %)	0 (%)
Total	30 (25.7 %)	41 (35 %)	4 (3.4 %)	17 (14.5 %)	14 (12 %)	11 (9.4 %)	0 (0 %)

tables, we take the number of the extensions of the corresponding pattern in a given system as the target denominator; in the case of total percentages (last row), the denominator is relative to the total number of extensions that instances of the given pattern introduce.

Different from the Linux kernel, we do not find global extensions as the dominant kind; rather, we find most annotations of AVOMF instances occurring at the block-level, whereas those of AVONMF occur at the function-level. Nonetheless, annotations at the global and type levels, which are suitable to be modularized using alternative techniques (e.g., by using aspects [2]), represent 32 % of the extensions in AVOMF; in AVONMF, the percentage is 29 %. Altogether, disciplined annotations are prevalent, capturing 89 % (33) and 79 % (92) of the extensions in AVOMF and AVONMF, respectively. Interestingly, we do not find a single extension at the level of function signatures. Although absent from axTLS and Toybox, undisciplined annotations occur in uClibc—23 % of the annotations in its AVOMF and AVONMF instances are not disciplined.

We also find the annotations of FCFTVOF, the most common pattern in uClibc, to be disciplined, following a similar distribution to the extensions in AVOMF and AVONMF.

4.5 Threats to Validity

Construct Validity. Our generality operationalization imposes a construct validity to the identification of general patterns. We argue, however, that our operationalization simply builds on top of the *Rule of Three*, a commonly accepted operationalization in pattern analysis. The extra constraint we impose in comparison to the operationalization in Chapter 3 is consistent to the level at which we set to identify general patterns, i.e., within a single target domain (the systems software domain). Others seeking to validate our patterns across different domains would apply yet another constraint layer, requiring the application of patterns across at least three distinct domains.

All the construct validity threats of our Linux kernel case study also apply to our catalog validation. We do not re-state such threats. Rather, readers should refer to Section 3.7.

External Validity. While our validation increases the external validity of part of our catalog, we cannot claim that the general patterns we identify extrapolate to domains different from the systems software. In fact, we do not seek such extrapolation, nor claim it.

Reliability Validity. Manually performing the steps or our methodology, allied with human subjectivity when matching patterns, imposes a reliability threat to our results. To mitigate it, we document our methodology accordingly, providing clear steps on how to define commit windows, along with a priority-based scheme to guide the matching of commit windows to a single pattern representing a change. Our results suggest a low reliability threat when identifying commit windows, as we find inconsistencies in only 4.3 % of all the cases we analyze. Pattern matching, in contrast, presents a higher reliability threat, as we find inconsistencies in almost 20 % of all the matches we perform. We mitigate the error-proneness of our matching process by performing reviews among participants' results, in addition to consensus meetings.

4.6 Conclusion

The validation of our catalog suggest that our patterns are expressive (RQ.1), at least, in the context of our three chosen targets (axTLS, Toybox, and uClibc). Our validation shows that our patterns explain 64 % (402) of all the additions and removals in our scope of analysis (624 cases). Moreover, our validation identifies 14 general patterns within the systems software domain (RQ.2), including patterns relative to the addition of features from completely new elements, featurization of existing code, renaming, and retirement situations. Currently, however, we cannot prove the generality of merge-related patterns.

Analyzing different pattern instances from distinct target systems reveal some interesting facts. Among others, we note that Toybox developers tend to replace *ifdef* annotations with if-statements, allowing the compiler to detect syntactic and semantic errors across all the branches of code annotations within a control-flow sequence. We also confirm that developers tend to write code annotations in a disciplined manner; moreover, we find features to have a low scattering degree.

Chapter 5

Towards Better Understanding Feature Scattering: A Longitudinal Analysis of Linux Kernel Device-Driver Features

The preliminary findings of Chapter 3 suggest that feature scattering is recurrent in Linux kernel evolution, but it is kept under control—our most frequent addition patterns suggest that scattering degrees are low and that scattering is predominantly localized. In such settings, scattering is not necessarily bad; if kept low, developers may benefit from it when extending the kernel in unforeseen ways, or when circumventing the dominant decomposition of the C language. In general, however, practical scattering limits, as found across the evolution of long-lived software systems, are yet unknown. Thus, further investigation of feature scattering across the Linux kernel evolution may shed some light into such lack of knowledge.

This chapter investigates the empirical limits of feature scattering in the Linux kernel evolution, in addition to investigating the influence of particular feature kinds on scattering degree values and scattering location. Specifically, we perform a longitudinal analysis of the scattering of device-driver features, the most common feature kind in Linux, studying their evolution in almost eight years of the kernel's 20-year history. Our goal is not to extract universal scattering limits, nor general feature kinds. Rather, we seek to provide evidence that scattering scales up to the limits seen in the Linux kernel evolution, and report features in the operating system domain that may influence scattering. Knowing practical limits and influencing features, whilst restricted to a particular system, is a requirement towards a formulation of a future general scattering theory.

Chapter Organization. Section 5.1 presents our four research questions. Section 5.2 outlines our methodological steps. By applying our methodology, we collect and analyze related data, presenting results in Section 5.3. We then argue about possible threats to validity in Section 5.4, concluding the chapter in Section 5.5.

5.1 Research Questions

As a further assessment of the results of our exploratory analysis in Chapter 3, we aim at better understanding the evolution of feature scattering. As such, we perform a longitudinal analyses of the scattering of Linux kernel device-driver features, guiding our investigation by four research questions. In the following, we enumerate and briefly discuss each research question.

RQ. 1. How does the growth of scattered features differ from non-scattered ones?

We analyze the relative and absolute growths of scattered and non-scattered features, comparing the evolution of the two kinds—for instance, to understand whether the proportion of scattered features is increasing, decreasing, or stable over time.

RQ. 2. How does the growth of locally scattered features differ from globally scattered ones?

We analyze the relative and absolute growths of features that are (i) scattered within their containing subsystem (local scattering), and (ii) of those that are scattered across at least another subsystem different from its containing one (global scattering). We compare both growth rates and aim at understanding how scattering is related to the kernel's architecture.

RQ. 3. How does the extent of feature code scattering evolve over time?

We analyze the scattering degree values, aiming at understanding their underlying distribution and possible thresholds. We also want to assess how the scattering degree relates to local and global scattering.

RQ. 4. What are possible factors influencing scattering of feature code?

We formulate and test hypotheses about possible factors influencing scattering. To this end, we collect and classify a random sample of features, identifying possible characteristics affecting where a feature is scattered across (local versus global scattering) or leading to higher scattering degrees.

5.2 Methodology

This section describes our methodology for collecting supporting data for our longitudinal study.

5.2.1 Scoping

To answer our research questions, we concentrate on device-driver features, that is, features defined in the driver subsystem of the kernel. This decision relies on the evidence showing that the Linux kernel evolution is mainly driven by the evolution of its device drivers. Such evidence includes the fact that most patterns adding features concern device drivers (see Chapter 3), as well as other results showing device drivers as the most active part of the Linux kernel evolution [53, 61, 70, 91].

Setting the scope to features in the driver subsystem requires us to distinguish them from features of other subsystems. Next, we explain how we perform such distinction.

5.2.2 Identifying Driver Features

To distinguish driver features from features of other subsystems, we perform as in Section 3.5.1. Using Hartman's mapping, we assign source files to a single subsystem. We then consider the subsystem of a feature's declaring Kconfig file as the feature's subsystem. It is worth noting that some features in the driver subsystem, although very few ($0.65 \pm 0.46\,\%$), are also declared in other subsystem(s)—e.g., inside core. As we cannot decide the subsystem of such features, we exclude them from our analysis.

After filtering the unique features in each kernel subsystem, we confirm that the Linux kernel is actually driven by the evolution of driver features. As Figure 5.1 shows, the driver subsystem is not only the largest in number of features, but also the fastest growing.

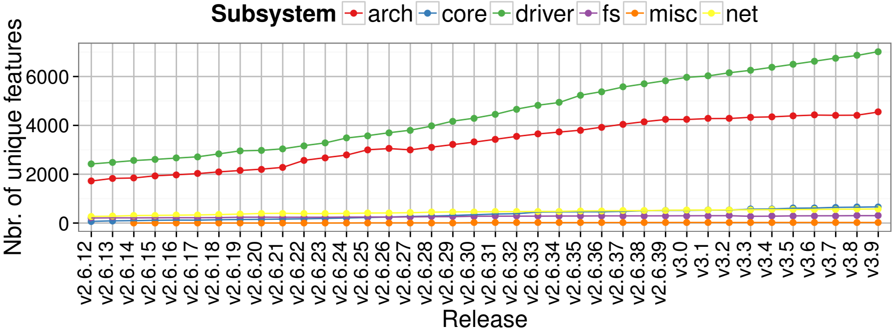


Figure 5.1: Feature distribution per subsystem

5.2.3 Data Collection

Our data analyses is based on longitudinal data from the v2.6.12–v3.9 release range. Figure 5.2 depicts our data-collection procedure. With a cloned repository of the Linux kernel source code in place, we query the kernel's source management system (Git) to list all release tags. From the listing, we filter the stable release identifiers (step 1). We then check out each stable release (step 2), setting the repository to a particular release snapshot. In the checked-out release, we list all C implementation and header files therein and clean them by removing empty lines and comments, and by transforming multilines[1] into single ones (step 3). We also eliminate strings in the source code as a means to facilitate pattern matching when mining feature references across the code (step 4). Finally, we collect metadata of each identified reference (step 5), including the name of the file in which a feature is referred, the line in which the reference occurs, and the associated *ifdef* pre-processor directive. We then store all feature references and their associated metadata in a relational database. For any given feature reference, there exists an associated file record in the database, which in turn links to a kernel subsystem in a given stable release.

All steps in our process are fully automated and are currently supported by extensions made to our pattern analysis toolset.

[1] Multilines end with the '\' character. They spread many physical lines, but the C compiler interprets them as a single one.

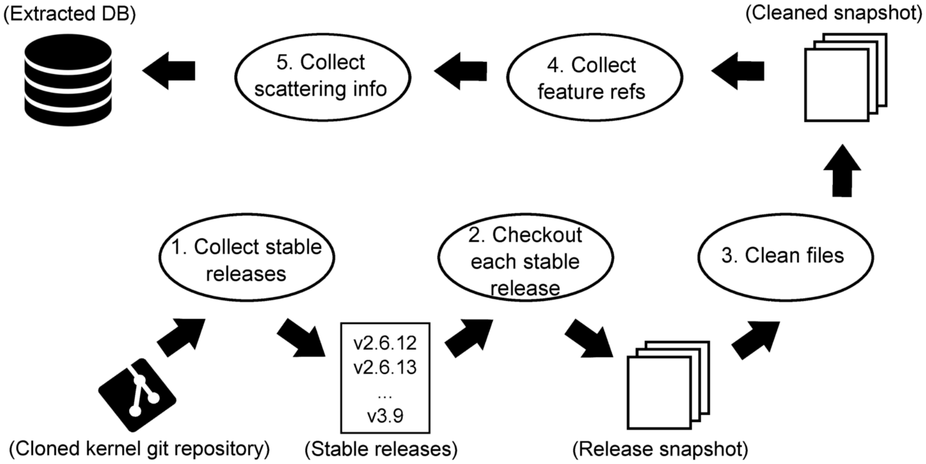


Figure 5.2: Data extraction process

5.2.4 Data Analysis

To answer our research questions, we issue SQL queries through the R statistical environment, which we connect to our database [32, 63]. Then, we plot the results and perform different statistical analyses according to each research question (we defer details to Section 5.3).

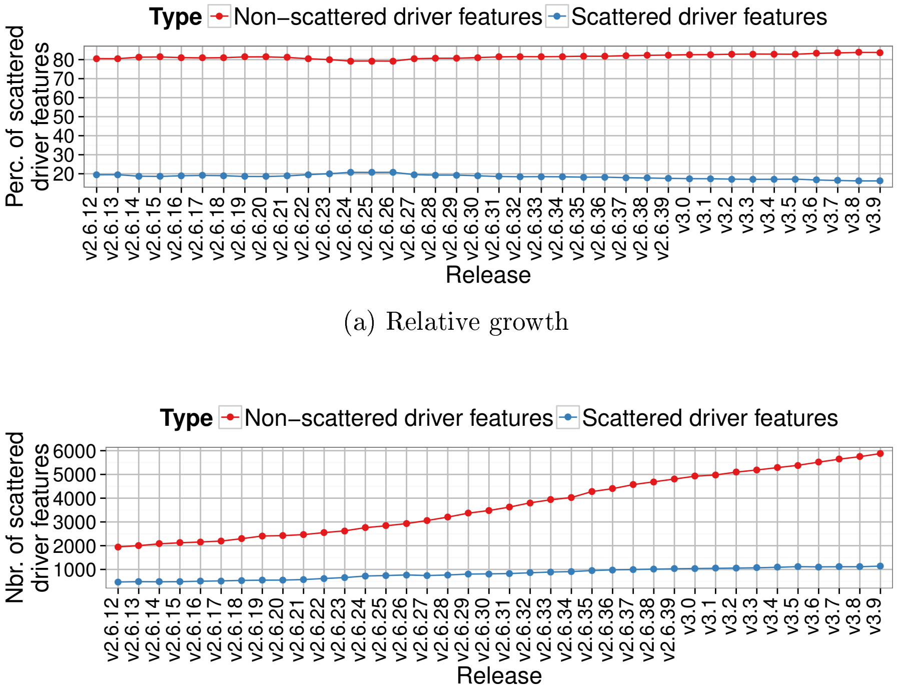
In our analysis, we measure the scattering degree (*SD*) of a driver feature *ft* in terms of its scattering degree at each implementation and header C file *f* in a set of target subsystems *S*, i.e.:

$$SD(ft, S) = \sum_{s \in S} \sum_{f \in s} SDF(ft, f) \tag{5.1}$$

where *SDF*(*ft*, *f*) is the number of *ifdefs* (`#if`, `#ifdef`, `#ifndef`, `#elif`) in *f* referring to *ft*. This is an alternative, yet equivalent, equation to our earlier *SD* definition in Chapter 2. As before, we restrict scattered features to those having an *SD*-value of at least two.

5.3 Results

This section reports the results of four respective research questions, aiming at understanding the evolution of scattered versus non-scattered features (RQ 1), of scattering within and across subsystem boundaries (RQ 2), of scattering degrees (RQ 3), and possible causes of the observed scattering and scattering degrees (RQ 4).



(a) Relative growth

(b) Absolute growth

Figure 5.3: Growth of (non-)scattered driver features

5.3.1 Scattered versus Non-Scattered Features

RQ. 1. How does the growth of scattered features differ from non-scattered ones?

To answer this question, we plot the proportion of scattered driver features in each kernel release, along with their absolute number. In both cases, we compare the growth rate of scattered driver features with the evolution of non-scattered ones. Figure 5.3 displays both plots, with summary statistics provided in the Tables 5.1 and 5.2. When applying Eq. 5.1 to identify scattered features, we take S as the union of all subsystems in the Linux kernel.

On average, 18 ± 1.2 % of driver features are scattered in any given release, with a maximum of 21 % and a minimum of 16 %. The average proportion is stable over time, although a decreasing trend starts from release v2.6.26. In absolute terms, the number of scattered driver features grows by 2.5 ± 2.4 % between each pair of consecutive stable

Table 5.1: Summary statistics of (non-)scattered driver features (relative)

Type	Min	Max	Mean	Diff	Slope
Scattered	16.23 %	20.79 %	18.44 ± 1.2 %	-3.22 %	-0.08
Non-scattered	79.21 %	83.77 %	81.56 ± 1.2 %	3.22 %	0.08

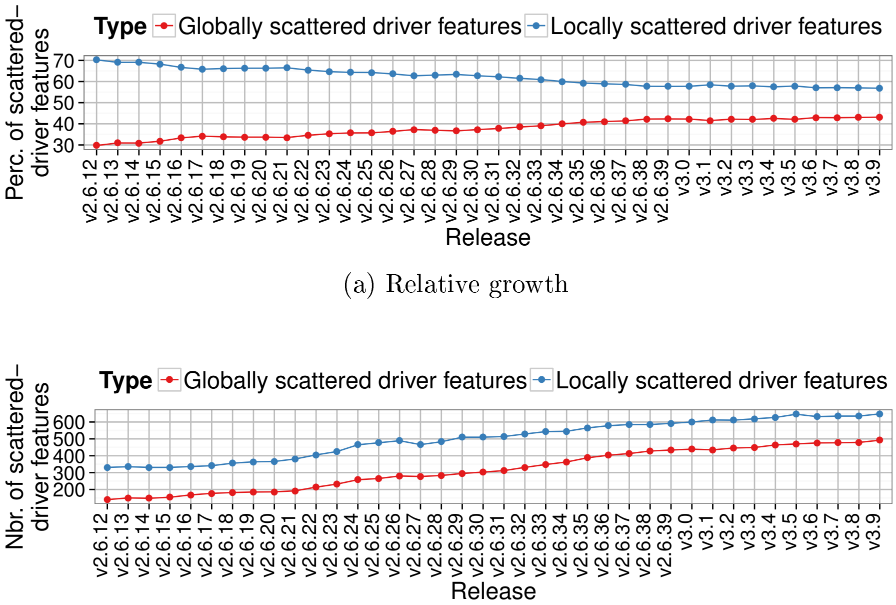
Table 5.2: Summary statistics of (non-)scattered driver features (absolute)

Type	Min	Max	Mean	Diff	Slope
Scattered	471	1,140	819.45 ± 228.88	142.04 %	20.4
Non-scattered	1,949	5,880	3,702.87 ± 1,280.49	201.69 %	114.36

releases. Since the first release under analysis (v2.6.12), the number of scattered driver features has grown by 142 %, as given by the *Diff* statistic.[2] In release v3.9, the kernel has over 1,000 scattered driver features. The latter, however, grows almost six times slower when compared to non-scattered driver features, as given by the ratio of their regression line slope coefficients. Moreover, the absolute growth of scattered driver features is not monotonic, with three small periods of decrease: v2.6.13–v2.6.14, v2.6.26–v2.6.27, and v3.5–v3.6.

The data confirms our earlier finding in Chapter 3, showing that the kernel architecture allows most driver features to be incorporated without causing any scattering. Some driver features, however, do not fit well into this architectural model and are thus scattered across the source code. Moreover, the proportion of scattered driver features is nearly constant, which may indicate that it is an evolution parameter actively controlled throughout the kernel evolution.

[2] The percentage difference (*Diff*) of two non-percentage values x_2 and x_1 is $100 \times (x_2 - x_1)/x_1$. If x_2 and x_1 are percentages, the *Diff*-value is simply $x_2 - x_1$. When calculating *Diff* for a given metric (e.g., number of scattered driver features), we take x_2 to be the metric value at the last inspected kernel release (v3.9), whereas x_1 is the metric value for the first release (v2.6.12).



(a) Relative growth

(b) Absolute growth

Figure 5.4: Growth of locally and globally scattered driver features

5.3.2 Local versus Global Scattering

With the next research question, we investigate to what extent the scattering of driver features is local and to what extent it is global. A globally scattered driver feature has at least one associated *ifdef* in an implementation or header C file that is not in the driver subsystem. In the case of a locally scattered driver feature, referring *ifdefs* occur only in source files in the driver subsystem. Ideally, most scattering should be local, contributing to internal cohesion, while decreasing coupling.

> **RQ. 2.** How does the growth of locally scattered features differ from globally scattered ones?

The growth of locally scattered driver features varies along the Linux kernel evolution. Nonetheless, it dominates the growth of globally scattered driver features, both proportionally and in absolute numbers. Figure 5.4 shows the corresponding plots, with summary statistics provided in Tables 5.3 and 5.4.

Table 5.3: Summary statistics of locally and globally scattered driver features (relative)

Type	Min	Max	Mean	Diff	Slope
Local	56.84 %	70.28 %	62.13 ± 4.12 %	-13.43 %	-0.36
Global	29.72 %	43.16 %	37.87 ± 4.12 %	13.43 %	0.36

Table 5.4: Summary statistics of locally and globally scattered driver features (absolute)

Type	Min	Max	Mean	Diff	Slope
Local	331	648	500.08 ± 110.89	95.77 %	9.82
Global	140	492	319.37 ± 118.53	251.43 %	10.58

In release v2.6.12, the proportion of locally scattered driver features is 70 %—the highest across all releases. Immediately after, the proportion follows a steady decrease, which stabilizes around 57 % from v2.6.38 onwards. In the latest release (v3.9), the percentage of locally scattered features is 56.8 % (648 absolute). The stabilization of local scattering causes a stabilization of globally scattered driver features at 43 %. Before stabilization, however, the graph depicts an increasing trend. In absolute terms, the number of globally scattered driver features grows at a faster rate than locally scattered ones, as given by their corresponding slope coefficients. Consequently, their relative difference decreases over time, resulting in the funnel shape of Figure 5.4a.

The relative and absolute dominance of local scattering contributes to internal cohesion within the driver subsystem. We conjecture that it eases maintenance, as local scattering requires less synchronization across subsystems. Nonetheless, it is interesting to see that the gap between the proportions of locally and globally scattered features is consistently decreasing, with a growing proportion of globally scattered driver features. Consequently, there is an increasing dependency from other subsystems to driver features. Although the latter may indicate an evolution decay, it does not seem to hinder the Linux kernel growth. As Section 3.2 shows, the kernel grows at a similar pace between each pair of consecutive releases. Thus, we interpret the stabilization of the proportion of globally scattered driver features as an effort to control its preceding growth trend. Hence, 43 % seems a current upper limit kept by Linux kernel developers.

5.3.3 Scattering Degrees

RQ. 3. How does the extent of feature code scattering evolve over time?

To answer this question, we plot the scattering degrees (*SD*) of all scattered driver features at each kernel release. When measuring *SD* (see Eq. 5.1), we take the target set of subsystems (S) as the union of all subsystems in the kernel. The boxplot in Figure 5.5, which we adjust for skewness [68], shows that 50 % of all scattered driver features have a low scattering degree, with $SD \leq 4$ across all stable releases. Above the 50 % of the distribution, however, the scattering of features considerably increases. In the third quartile (up to 75 % of the distribution), *SD*-values practically double, lying between seven and eight. In the remaining 25 %, the highest *SD*-values that are not outliers range from 34 to 55, as indicated by the top whiskers. In this range, the average *SD*-value is 44 ± 5.3. Above the top whiskers, outliers (shown as dots) have high *SD*-values, with a minimum of 35 and a maximum of 377 (median of 63). As the kernel evolves, outliers grow in absolute numbers as well as relatively. Figure 5.6 displays the corresponding graphs, with summary statistics provided in Table 5.5. In absolute numbers, outliers display a 500 % increase, with as little as 7 features in release v2.6.12 and 42 in v3.9. Relatively, however, the *Diff* between the first and last release is only 2.2 %.

The analysis of the *SD*-values of scattered driver features indicates a skewed distribution. In the kernel's evolution, 75 % of *SD*-values are small (4) to medium (8). A dispersion, however, occurs in the remaining 25 % (values 34–377), pushing the distribution tail to the right. Consequently, the distribution is skewed to the right, increasing the difference between a typical *SD*-value (4) and the mean (8). In such settings, the mean is a not robust statistic. Instead, practical scattering limits should be relative (e.g., 75 % of the features should have $SD \leq 8$), rather than a single value to which all features would adhere to.

To ascertain the observed skewness, while summarizing how unevenly *SD*-values are distributed among scattered driver features, we calculate the Gini coefficient [138] for each kernel release. The Gini coefficient is a popular summary statistic in economics, measuring the inequality of wealth (e.g., the value of a software metric, such as *SD*) among the individuals (e.g., features) of a population. Its value is in the range of zero and one; zero means a perfect equality, where all individuals have the same wealth. A high value, in contrast, denotes an uneven distribution.

Figure 5.7 shows the evolution of the Gini coefficient in the Linux kernel evolution. The coefficient follows a decreasing trend in the first 12 releases, meaning that *SD* is more evenly distributed. From release v2.6.23 onwards, an increasing trend can be observed, indicating

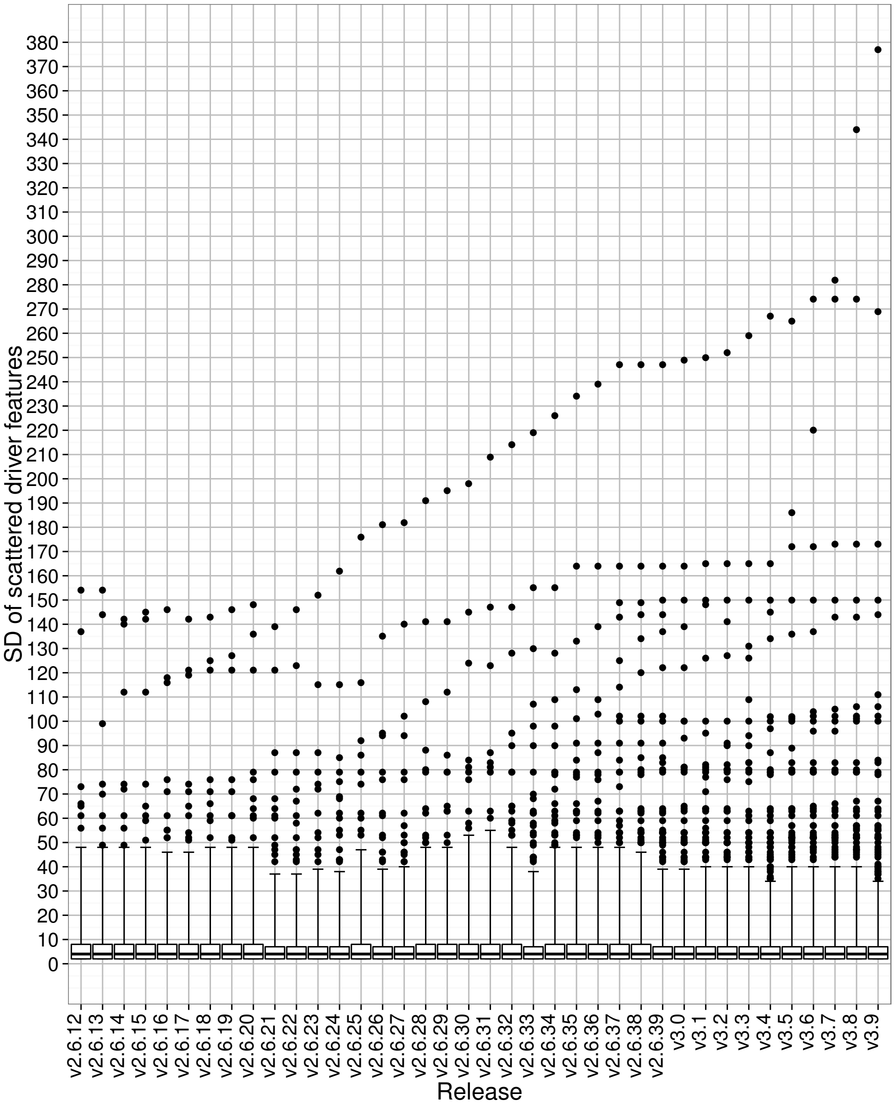
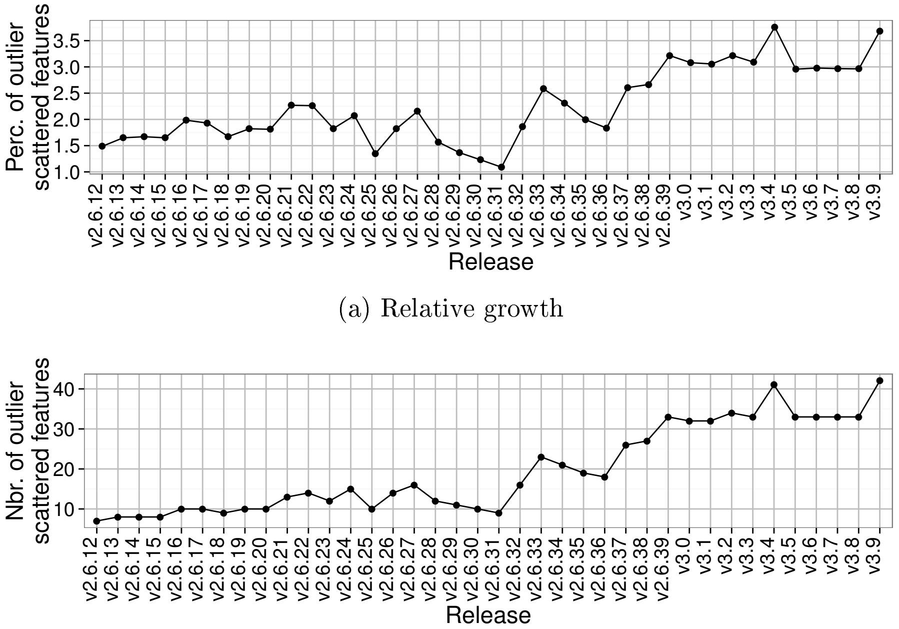


Figure 5.5: *SD*-values of scattered driver features

that *SD* is more concentrated towards a particular set of features. The absolute difference between the coefficients in v2.6.23 and v3.9, however, is only 0.06, which indicates that *SD* distribution does not vary considerably. At all times, the Gini coefficient is closer to one than to zero, confirming the observed right-skewness.

Finally, we partition the *SD* distribution into globally and locally scattered driver



(a) Relative growth

(b) Absolute growth

Figure 5.6: Growth of outlier scattered features

Table 5.5: Relative and absolute growth of scattered outlier features

Type	Min	Max	Mean	Diff	Slope
Relative	1.09 %	3.76 %	2.25 ± 0.71%	2.2 %	0.05
Absolute	7	42	19.61 ± 10.84	500 %	0.89

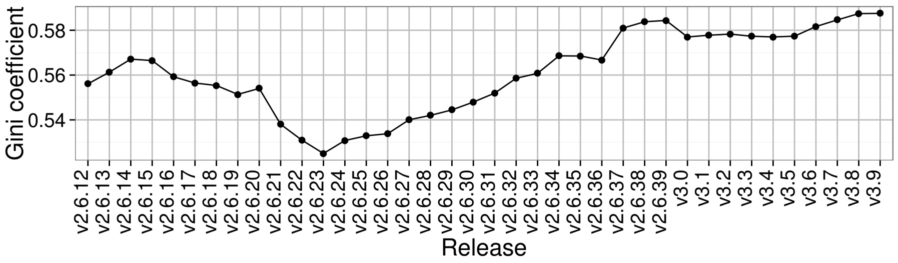


Figure 5.7: Evolution of the Gini coefficient of the *SD*-value of scattered driver features

Table 5.6: Summary statistics of the average *SD*-value of scattered driver features

Type	Min	Quartile 1	Median	Quartile 3	Max
Local	2	2	3	6	84.13
Global	2	2.14	4	7.49	199.5

features. For each feature, we take the average of all its *SD*-values, accounting each release containing the feature. We then compare the distributions of the averages in each partition. As Table 5.6 shows, starting from the median, globally scattered features have higher average *SD*-values. Thus, globally scattered driver features do not only affect more subsystems, but also tend to have higher prospective maintenance costs, given that more locations in the code base might have to be maintained.

5.3.4 Causes of Scattering

RQ. 4. What are possible factors influencing scattering of feature code?

To answer this research question, we investigate whether specific kinds of features exist that by their nature affect where a feature is scattered across (local versus global scattering) or lead to higher scattering degrees. The characteristics we test are the result from past experience and observations when manually analyzing and classifying features in the Linux kernel and other systems [108].

The first kind of features we observe relates to so-called *platform* devices. As opposed to *hotplugging* devices, these cannot be discovered by the CPU. An experienced kernel developer explains:[3]

> "*Happily, we now live in the days of busses like PCI which have discoverability built into them; any device sitting on a PCI bus can tell the system what sort of device it is and where its resources are. [...] Alas, life is not so simple; there are plenty of devices which are still not discoverable by the CPU. [...] So the kernel still needs to provide ways to be told about the hardware that is actually present. 'Platform devices' have long been used in this role in the kernel.*"

[3] Written by Jonathan Corbet, the main author of the *Linux Device Drivers* book [34]. See `http://lwn.net/Articles/448499/`

In the kernel, a platform driver is any driver that instantiates a platform_driver C structure. Since platform devices cannot be discovered by the CPU, the kernel cannot automatically load their corresponding LKMs, as in hotplugging. Instead, board-specific code [73] instantiates which devices to support for a target CPU, and with which drivers. However, developers do not instantiate all possible platform devices when porting Linux to a particular CPU, as only some will be present at all times. In the face of such hardware variability, it is intuitive to assume that developers will be more prone to introducing extensions outside the driver subsystem (e.g., in the arch subsystem, which contains CPU-dependent code), conditioning them on the presence of specific platform devices and their associated drivers and capabilities. For non-platform driver features, the opposite should occur; through hotplugging, devices should be discovered at runtime, triggering the automatic loading of required LKMs.

The second kind of features concerns domain abstractions, which provide a core *infrastructure* from which concrete drivers are built. These abstractions do not bind to a specific vendor, but rather represent a generic set of devices and driver-related capabilities. Examples include generic buses (e.g., USB, PCI, and ACPI), drivers declaring specific device classes (a type of a device, such as an audio or network device),[4] and hardware-description frameworks (e.g., OpenFirmware).[5] Since these features denote abstractions in the operating-system domain, we assume that they should have a higher likelihood of being scattered in comparison to non-infrastructure features. In such cases, extensions in code would check for specific generic functionality and related capabilities, allowing features to react accordingly.

Next, we investigate whether the kind of a feature affects which subsystem the feature's code is scattered across (location) or its scattering degree.

Influence on Scattering Location

We test the effect of being a platform feature on scattering location by first collecting a random sample of 10 % of all scattered driver features (population size is 1,700). We then manually classify sample features as either platform or not. A platform-driver feature is either a platform driver (i.e, it has at least one compilation unit instantiating a platform_driver structure) or it is a capability of a container platform-driver feature. For further details on the classification process, see our online appendix [1].

[4] https://www.kernel.org/pub/linux/kernel/people/mochel/doc/text/class.txt

[5] http://www.openfirmware.info/

With the classified sample, we then perform the χ^2 statistical test at a significance level of 0.05. Our hypotheses are:

> *Null hypothesis* (H_0): *being a platform-driver feature has no effect on scattering location*
>
> *Alternative hypothesis* (H_1): *being a platform-driver feature has an effect on scattering location*

Table 5.7 shows the contingency table we use in the test, along with the resulting χ^2 statistic and p-value.

We find strong evidence ($p = 1.933 \times 10^{-5} < 0.05$) of a dependency between being a platform-driver feature and scattering location. Thus, we can reject the null hypothesis in favor of the alternative one. In fact, the analysis of Table 5.7 indicates that platform-driver features are 2.5 times more likely to be globally scattered than non-platform ones. Conversely, a non-platform driver feature is 1.8 times more likely to be locally scattered. In summary, the test confirms our initial understanding: when facing non-discoverable devices, developers are more likely to introduce *ifdefs* outside the driver subsystem. For non-platform devices, the scattering of their driver code is likely local. As Figure 5.8 shows, most globally scattered platform-driver features in our sample are scattered across the arch subsystem, either only in arch, or in both arch and driver (in the figure, 'either' is captured by the '+' sign, whereas 'and' is denoted by '&'). This evidences a tight relationship between the arch subsystem and platform-driver features; since platform devices are not discoverable by the CPU, supporting the drivers of some of such devices requires scattering CPU-dependent code, which is mostly found inside the arch subsystem.

To check the influence of infrastructure features, we perform similar steps as in the previous test. Using the same sample set of scattered features, we classify them as either infrastructure or not. We test the following hypotheses:

> *Null hypothesis* (H_0): *being an infrastructure-driver feature has no effect on scattering location*
>
> *Alternative hypothesis* (H_1): *being an infrastructure-driver feature has an effect on scattering location*

Here, we do not have a strong evidence suggesting that being an infrastructure-driver feature has an effect on scattering location. As Table 5.8 shows, running the χ^2 test results in a p-value greater than the chosen significance level. Thus, we fail to reject the null hypothesis.

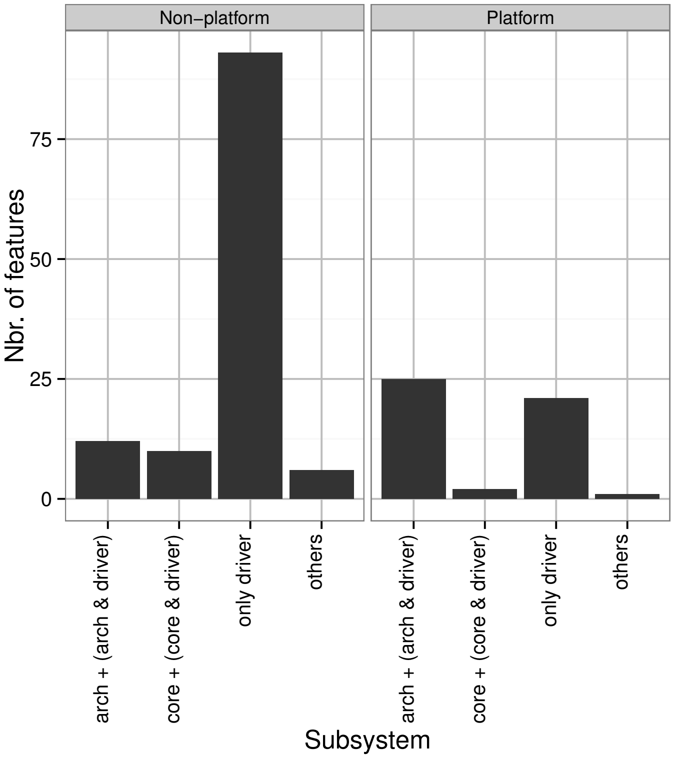


Figure 5.8: Scattering location of sampled (non-)platform drivers

Influence on Scattering Degree

In order to verify the influence of being a platform or an infrastructure-driver feature on scattering degree, we initially calculate the average SD-value of each sample feature across all releases containing it.

To check the influence of being a platform-driver feature, we split the calculated average SD-values into those that concern platform-driver features, and those that do not. We then perform a one-tailed Mann-Whitney-Wilcoxon rank sum statistical test to assess whether platform-driver features systematically yield higher average SD-values in comparison to non-platform driver features.[6] Our hypotheses are:

[6] Systematically here means that the probability of having an average SD greater than a value X among platform-driver features is greater than the probability of having an average $SD > X$ among non-platform ones [98].

Table 5.7: Relationship between being a platform-driver feature and scattering location

$\chi^2 = 18.26, p = 1.933 \times 10^{-5}$

	Is locally scattered?		
Is platform?	No	Yes	Total
No	28	93	121
Yes	28	21	49
Total	56	114	170

Table 5.8: Relationship between being an infrastructure-driver feature and scattering location

$\chi^2 = 2.43, p = 0.1194$

	Is locally scattered?		
Is infrastructure?	No	Yes	Total
No	44	100	144
Yes	12	14	26
Total	56	114	170

> *Null hypothesis* (H_0): *there is no difference in the distribution of average SD-values of platform and non-platform driver features*
>
> *Alternative hypothesis* (H_1): *average SD-values are systematically higher in platform-driver features*

We do not find convincing evidence that average SD-values are systematically higher in platform-driver features. With a 0.05 significance level, we are unable to reject the null hypothesis. We also test whether platform-driver features influence the average SD-value of non-infrastructure features, as we do not classify platform-driver features as infrastructure. As before, we do not find convincing evidence ($p = 0.8496$).

There seems to be also no significant influence of being an infrastructure-driver feature on scattering degree. Running the one-tailed Mann-Whitney-Wilcoxon rank sum statistical test to compare the average SD-values of infrastructure with non-infrastructure features only supports the null hypothesis.

To better understand these results, we place sample features in one of three groups according to the scattering degree limits reported in RQ 3: low (average $SD \leq 4$), medium ($4 <$ average $SD \leq 8$), or high (average $SD > 8$). The resulting plot in Figure 5.9, which compares infrastructure and non-infrastructure features, shows that both feature kinds have a similar proportional contribution to each scattering degree level bin, yielding similar outcome probabilities. The same occurs between platform and non-platform driver features.

In the case of the outliers observed in RQ 3 (see Figure 5.5), however, we do find some influence of being an infrastructure-driver feature on extremely high SD-values. By ranking the average SD-value of each outlier feature, we plot the histogram of the number of features with an average SD-value matching each rank, partitioning the feature set into infrastructure and non-infrastructure outliers. As Figure 5.10 shows, infrastructure-driver features are the most scattered features among outliers, with 9 out of the 15 most scattered driver features in the kernel evolution. Considering the total number of outliers, however, infrastructure-driver features are out-performed by non-infrastructure ones. Consequently, most outliers are not infrastructure-related, but are rather narrow in purpose (e.g., target a specific bus-type or particular hardware manufacturer). Among those (Figure 5.11), we find that most relate to platform-driver features bound to specific system-on-a-chip devices, such as serial link devices, general input/output (GPIO) capabilities, and video support.

5.4 Threats to Validity

Next, we discuss the possible threats to the validity of our analyses and results.

External Validity. The largest threat to external validity is that our data are based on one case study only. 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.

To investigate whether two specific kinds of features (platform and infrastructure features) impact scattering degrees and lead to global scattering, we perform hypothesis testing based on a sample of 170 scattered features (population size is 1,700), given that it requires manual classification of features. This sampling is justified, and we rely on standard p-value limits to test hypotheses. Recall that the investigation of outliers does not rely on sampling, but on classifying the whole population (54 features).

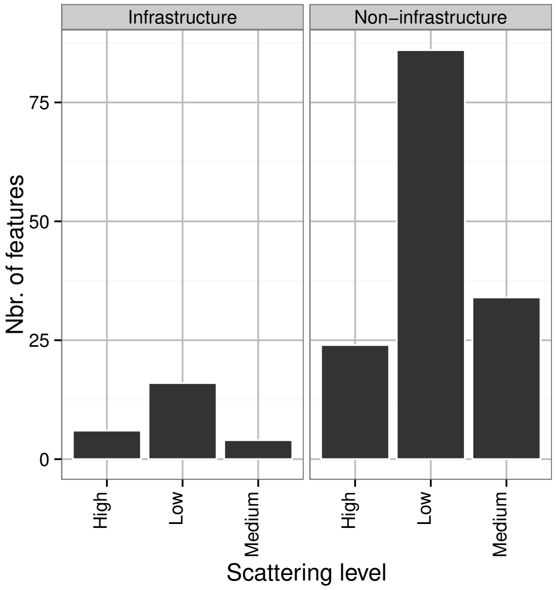


Figure 5.9: Scattering degree levels in (non-)infrastructure driver features

Finally, our analysis of code scattering relies on pre-processor directives. However, variability in the Linux kernel also affects entire files, as their compilation is controlled by specific features. Thus, we show a partial, yet valid, view of the true story. Of course, our results should be complemented by studying code scattering on the more coarse-grained source file level. Using this information, such as from previous attempts to analyze the Linux kernel's build system, is left as valuable future work.

Internal Validity. There is always the risk that bugs in our custom-made tools and scripts impact results. To mitigate this threat, we perform extensive code reviews, including the author of this thesis and an external collaborator. In total, we expend almost 16 hours of code reviews. Additionally, we also use a test suite with over 70 test cases.

For all analyses, we exclude features that we cannot uniquely map only to the driver subsystem. This limitation, however, has no further impact on our results, as only very few driver features (0.65 ± 0.46 % per kernel release) are declared across different subsystems. We also exclude references to features that occur in strings in the code, assuming that such references have no impact on maintenance, as opposed to the code parts controlled by pre-processor-directives, which we analyze.

Construct Validity. To measure scattering of feature code, we rely on a very simple metric (*SD*). Since it measures the parts related to feature code as specified by the original

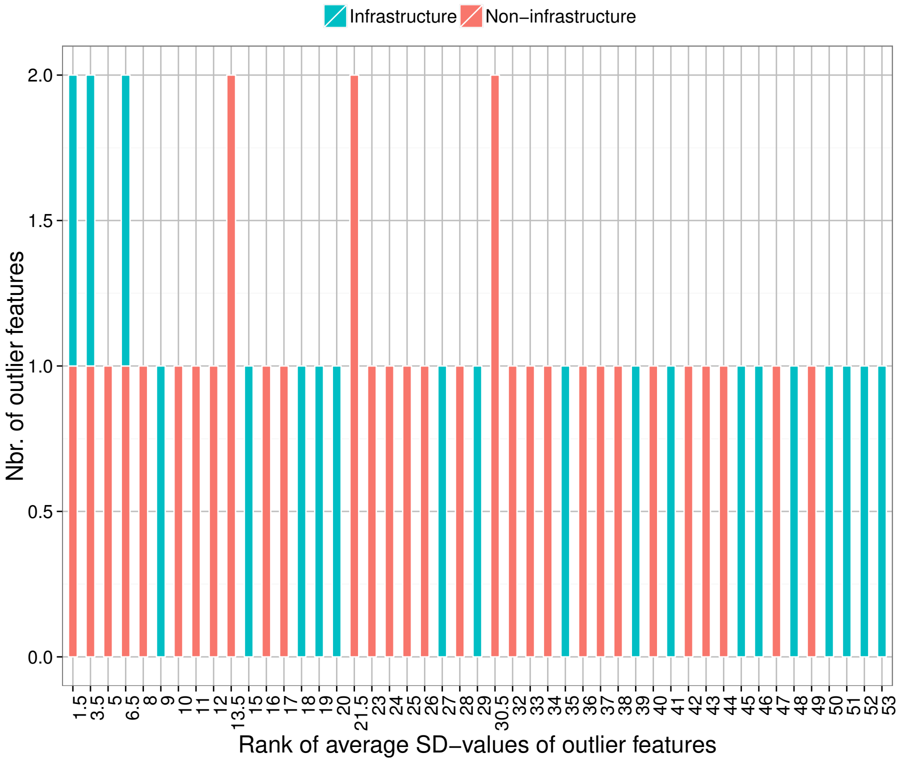


Figure 5.10: Ranking of outlier features

developers (using pre-processor directives), it is also a very valid measurement of scattering. In fact, the ability to rely on this information is a major advantage over previous studies, which need to recover the mapping of features (or concerns, see Chapter 2) to code.

However, it is not completely clear how these syntactic code extensions, which aim at realizing variability, relate to semantic code extensions, that is, units of functionality from a domain-oriented view. Understanding this relationship constitutes an interesting future research question.

5.5 Conclusion

We analyze almost eight years of evolution history of device-driver features in the Linux kernel, focusing on the feature scattering evolution. Our goal is not to investigate limita-

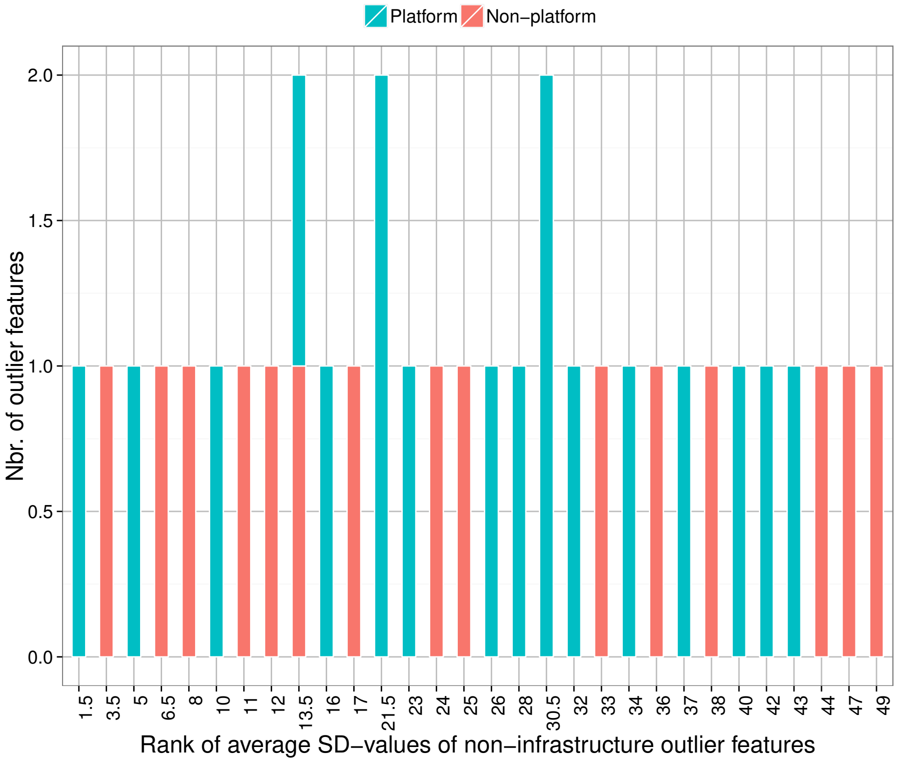


Figure 5.11: Ranking of features among non-infrastructure outliers

tions or maximum degrees of scattering, but to find empirical evidence that scattering can be handled to the extents we can find in one of the largest feature-based software systems in existence today.

We learn that the majority of driver features (82 %) can actually be introduced without any scattering (RQ 1). Classic modularity mechanisms, as employed by the Linux kernel software architecture, seem to suffice. Yet, the absolute number of scattered driver features is still higher than our expectations. Proportionally, however, the amount of scattered features remains nearly constant throughout the kernel evolution. Whether such a limit is actively maintained by developers remains an interesting open question.

We also find that scattering is not limited to subsystem boundaries (RQ 2). While most driver features are in fact only implemented in the driver subsystem, a significant proportion (43 %) of features has extensions in other subsystems. This proportion, however, is stable in the last third of releases.

The implementation of the majority (75 %) of scattered driver features is scattered across a moderate number of four to eight locations in code (RQ 3). Moreover, the median is low and constant across the entire evolution ($SD = 4$). Yet, the distribution is skewed to the right, with outliers having scattering degrees up to 377. Thus, the arithmetic mean is not a reliable threshold to monitor the evolution of feature scattering. Outliers, however, are limited in number, accounting for less than 4 % of the total number of features in the kernel; however, their absolute counting and magnitude grow with the system.

We identify and analyze two kinds of features that are prone to scattering (RQ 4). So-called *infrastructure* features account for 9 out of the 15 most highly scattered outliers in the scattering distribution of driver features, affecting many parts of the code. So-called *platform* features in the Linux kernel are more frequently scattered across subsystem boundaries, but do not necessarily have higher scattering degrees. The cases where platform-driver features affect scattering degree occur within non-infrastructure outlier features, where platform-driver features account for most of the outliers in that group. While the scattering of platform features across subsystem boundaries can be potentially avoided, the necessary generalization of code and abstraction layers might be too expensive or difficult to be achieved in practice, due to hardware detection limitations. Thus, scattering using pre-processor directives is a natural mechanism in this context, yet facing a potential maintenance trade-off.

Chapter 6

Conclusion

6.1 Summary

We build this thesis on the assumption that the existing poor variability tool support follows from lacking a through understanding of how variability evolution happens in real-world settings. Thus, as a research goal, we seek to advance such understanding.

We perform our research in two major steps. First, we perform an exploratory analysis of a large, complex, and real-world variability-aware system: the Linux kernel. In a second step, we further assess the results of our exploratory analysis by considering three other variability-aware systems in the same domain of the Linux kernel (the systems software domain), in addition to the Linux kernel itself.

In our exploratory analysis, we sample a portion of the Linux kernel commit history, analyzing the variability evolution induced by adding or removing features from the Linux kernel variability model, tracking how different artifacts coevolve as a result. We report our findings as a catalog of 23 patterns covering five main evolution scenarios: addition of new features from completely new elements (9), renaming of features (1), merge cases (2), featurization (2), and feature retirement situations (9). For each pattern, we crosscheck specific properties of its instances against evidence from existing literature, in addition to documenting trends in how developers employ our patterns. Among others, we find that whenever developers add extensions in code, they do it in a disciplined manner. Moreover, contradicting best practices of software development, we find feature scattering to be recurrent in the Linux kernel development.

Following our exploratory research analysis, we seek the validation of our Linux kernel coevolution patterns in the context of three other subjects in the systems software domain, in addition to assessing the extents of feature scattering across an entire snapshot of the Linux kernel repository.

To validate our catalog, we analyze over 20 years of the combined evolution of three target subjects: axTLS, Toybox, and uClibc. Overall, we find that our patterns can express two thirds of all addition and feature removal cases in the evolution of our three chosen systems. Furthermore, we prove the generality of 14 patterns within our catalog. Generality, in this case, restricts to the systems software domain.

In the assessment of feature scattering limits, we analyze almost eight years of Linux kernel development. Our analysis scopes to device-driver features, which stand as the most frequent kind across the entire Linux kernel evolution. Among others, we show that, while scattered features are regularly added, their proportion is lower than non-scattered ones, indicating that the Linux kernel architecture allows most features to be integrated in a modular manner. The median scattering degree of features is constant and low, but the scattering-degree distribution is heavily skewed. Thus, using the arithmetic mean is not a reliable threshold to monitor the evolution of feature scattering. When investigating influencing factors, we find that platform-driver features are 2.5 times more likely to be scattered across architectural (subsystem) boundaries when compared to non-platform ones. Their use illustrates a maintenance-performance trade-off in creating architectures as for Linux kernel device drivers.

6.2 Future Work (Research Agenda)

From our findings, we define directions for future work. We do it by presenting it as a research agenda for ourselves, as well as for other researchers.

Other Variability Evolution Cases and External Practices. Since our patterns cover only a small fraction of the entire evolution history of all the subjects we analyze, future research shall investigate which other kinds of changes exist outside our scope of analysis, subsequently mining new patterns. This includes analyzing changes not triggered by adding or removing features in the variability model, as well as those that occur within a single space or spanning multiple ones. Outside the systems software domain, it is fair to assume that some of our patterns apply to other systems, as long as they are organized in terms of a variability model, a mapping, and an implementation space with code annotations. Nonetheless, it is currently unknown which patterns of our catalog occur outside the systems software domain. Future research shall address such direction.

Also, as our patterns are an indirect observation of what developers do, we aim to conduct interviews with the developers of our subjects to get further insights on how they coevolve variability models and related artifacts. Additionally, we seek to assess the existence of maintenance practices corroborating our patterns.

Further Assessment of Non-General Patterns While we claim the generality of 14 patterns in our catalog, there are nine others for which we do not. Future research shall investigate their generality.

Pattern-Based Feature Traceability. Our patterns provide a starting point for creating new feature traceability heuristics in systems that follow a similar structure as found in the Linux kernel (variability model, mapping, and C code with pre-processor annotations). Although existing feature localization techniques [28, 40, 49, 95, 124, 137, 146] can relate code artifacts (or fragments of them) to features of the system, enabling the vertical traceability between features and code, evolution imposes a *temporal traceability* among features; to trace a feature from a given point to another back in the evolution history or forward in time, one must account for changes that occur together with the variability model. Otherwise, incorrect traces might be reported. To the best of our knowledge, we are unaware of any existing technique that performs such a holistic analysis. In this case, our patterns can serve as a starting point for researching pattern-based traceability heuristics. For example, as reported in our two merge patterns (MVOFS and MVOFNO), the removal of a feature and its implementation artifacts, together with aiding the implementation of another feature with the capabilities of the removed one, is likely to characterize a merge between the two features.

Alternatively, evolution patterns can be incorporated in the evolution process of variability-aware systems. Once cataloged (e.g., following our methodology), patterns can be associated with each new commit, either manually (e.g., by stating such relation in commit log messages), or automatically. In the latter case, research shall investigate how to detect whether patches conform to specific patterns. In fact, our research is already fostering such direction—see [43]. Associating patterns and commit patches are likely to improve developers' productivity when revisiting a past change and reduce misinterpretations when analyzing its structure.

Variability Evolution Algebra and New Product-Line Theories. A natural follow-up of our work is the decomposition of our patterns into a set of small, but composable operators to transform the variability model, mapping, and code. The application of a pattern, in turn, shall become an application of a sequence of operators. Such operators could be formalized as an algebra for evolving systems whose structure is similar to the one in the Linux kernel. Building on top of the evolution algebra, new theories could also be devised, accounting not only feature refinement (as in [22]), but also retirement situations.

Identify Preferred Mechanisms Supporting Coevolution Patterns. When studying axTLS, Toybox, and uClibc, we note that developers realize some of our patterns using different mechanisms. Future research shall investigate whether there is a general preference towards any specific mechanism, or whether preferences exist within particular communities. Such information is useful for tool builders seeking to automate some of our patterns.

IDE Support. After devising an algebra for variability evolution and the identification of preferred mechanisms for realizing our patterns, research shall seek the automation of patterns in our catalog. For instance, by supporting a variability evolution algebra as a transformational engine, integrated development environments (IDEs) could automate the application of existing patterns (e.g., AVOMF, FCFTVOF, RNM, etc), while requiring user assistance for others (e.g., merges).

Confirmatory Studies for Better Understanding Feature Scattering. As future work, we aim at studying scattering in a confirmatory manner, running interviews (or surveys) with kernel contributors and device-driver developers. Specifically, we want to uncover whether kernel developers consciously manage feature scattering and whether the limits we find are enforced in practice. We also do not know whether the observed scattering evolution is a model for other systems. Obtaining a more general picture requires further case studies.

Another direction concerns the effect of scattering on actual maintenance effort. For instance, are modules with highly scattered features harder to maintain, to what extent, and are they more error-prone, and how? What is the effect of feature-ownership on maintenance effort, especially when features are highly scattered? A single developer maintaining a feature is likely most efficient, but given a high number of features and a distributed development model, it is unrealistic. Thus, finding an optimal organizational structure in a project such as the Linux kernel is a difficult problem—solving it requires further empirical measurements.

Another future work direction concerns the investigation of how scattering could be reduced with alternative solutions, either by using better languages or designs. In either

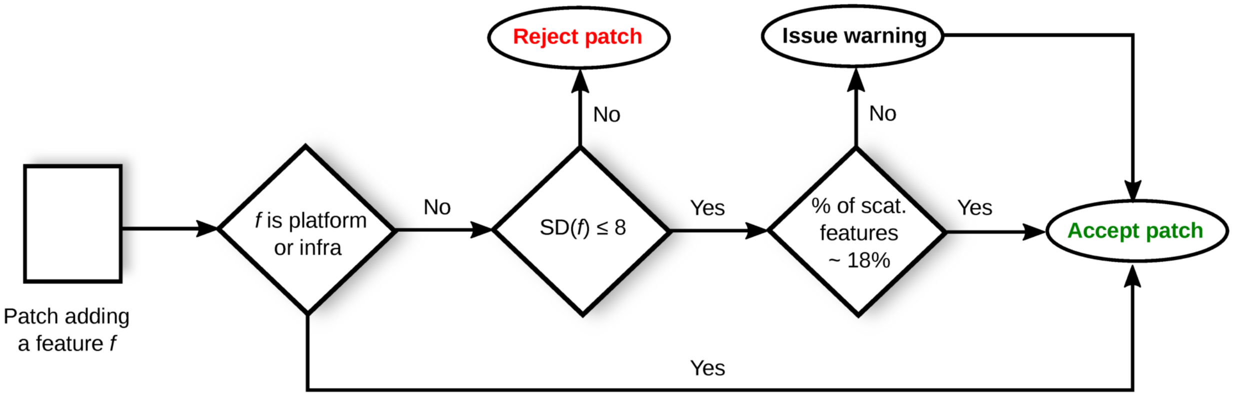


Figure 6.1: Patch validation

case, we need to know from developers whether such alternatives are suitable to their development context—if not, the reasons for not adopting them.

Enforcing Scattering Limits One way to ensure low scattering is by documenting and enforcing guidelines on how to write *ifdef* annotations. As we show in Chapter 3, guidelines do exist in the Linux kernel. However, it is currently unknown whether the same guidelines apply to or are enforced by other projects. Future work shall investigate this. As for the scattering limits we report, those could be used by the Linux kernel community as a means to validate patches—for instance, as outlined in Figure 6.1. Such application depends on whether Linux kernel developers agree with the limits arising from the system's evolution. Additionally, one could investigate whether developers tend to reject patches adding features with higher scattering degrees than those with lower *SD*-values.

References

[1] Online Appendix. `http://lpassos.bitbucket.org/thesis/`.

[2] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can We Refactor Conditional Compilation into Aspects? In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 243–254. ACM, 2009.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[4] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 201–210. ACM, 2006.

[5] MichałAntkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible Product Line Engineering with a Virtual Platform. In *Proceedings of the International Conference on Software Engineering*, pages 532–535. ACM, 2014.

[6] Sven Apel. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology*, 9(1):117–142, 2010.

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

[8] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008.

[9] Muhammad Ali Babar, Lianping Chen, and Forrest Shull. Managing Variability in Software Product Lines. *IEEE Software*, 27(3):89–91, 94, 2010.

[10] L. Baresi, S. Guinea, and L. Pasquale. Service-Oriented Dynamic Software Product Lines. *IEEE Computer*, 45(10):42–48, 2012.

[11] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Conference on Software Product Lines*, pages 7–20. Springer, 2005.

[12] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-wise Refinement. In *Proceedings of the International Conference on Software Engineering*, pages 187–197. IEEE, 2003.

[13] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software*, 11(5):89–94, 1994.

[14] Don S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An Extensible Database Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.

[15] Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software. In *Proceedings of the Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 397–412. ACM, 2006.

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

[17] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wąsowski. Variability Modeling in Industry: Practices, Benefits, and Challenges. In *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 302–319. Springer, 2014.

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

[19] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wąsowski. Feature-to-Code Mapping in Two Large Product Lines. In *Software Product*

Lines: Going Beyond, volume 6287 of *Lecture Notes in Computer Science*, pages 498–499. Springer, 2010.

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

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

[22] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. A Theory of Software Product Line Refinement. *Theoretical Computer Science*, 455(0):2–30, 2012.

[23] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[24] Robert Bowdidge. Performance Trade-offs Implementing Refactoring Support for Objective-C. In *Workshop on Refactoring Tools*, 2009.

[25] Magiel Bruntink, Maja D'Hondt, Arie van Deursen, and Tom Tourwé. Simple Crosscutting Concerns do not Exist—Analysing Variability in Large-Scale Idioms-Based Implementations. In *Proceedings of the International Conference on Aspect-Oriented Development Engineering*, pages 199–211. ACM, 2007.

[26] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. Discovering Faults in Idiom-based Exception Handling. In *Proceedings of the International Conference on Software Engineering*, pages 242–251. ACM, 2006.

[27] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014.

[28] Kunrong Chen and Václav Rajlich. Case Study of Feature Location Using Dependence Graph, after 10 Years. In *Proceedings of the International Conference on Program Comprehension*, pages 1–3. IEEE, 2010.

[29] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability Management in Software Product Lines: A Systematic Review. In *Proceedings of the International Software Product Line Conference*, pages 81–90. Carnegie Mellon University, 2009.

[30] Lianping Chen and MuhammadAli Babar. Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges. In *Proceedings*

of the International Conference on Software Product Lines: Going Beyond, pages 166–180. Springer, 2010.

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

[32] Joe Conway, Dirk Eddelbuettel, Tomoaki Nishiyama, Sameer Kumar, and Tiffin Neil. *Package 'RPostgreSQL'*, 2013. R package version 0.4.

[33] Jonathan Corbet, Greg Kroah-Hartman, and Amanda 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-2015`, 2015. Last seen: June 6, 2016.

[34] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition.* O'Reilly, 2005.

[35] James R. Cordy. Submodel Pattern Extraction for Simulink Models. In *Proceedings of the International Software Product Line Conference*, pages 7–10. ACM, 2013.

[36] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 422–437. Springer, 2005.

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

[38] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10:7–29, 2005.

[39] Krzysztof Czarnecki, Simon Helsen, and Eisenecker Ulrich. Staged Configuration through Specialization and Multilevel Configuration of Feature Models. *Software Process: Improvement and Practice*, 10:143–169, 2005.

[40] Jean-Christophe Deprez and Arun Lakhotia. A Formalism to Automate Mapping from Program Features to Code. In *Proceedings of the International Workshop on Program Comprehension*, pages 69–78. IEEE, 2000.

[41] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the International Software Product Line Conference*, pages 21–30. ACM, 2012.

[42] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikshat, and Daniel Lohmann. Understanding Linux Feature Distribution. In *Proceedings of the Workshop on Modularity in Systems Software*, pages 15–20. ACM, 2012.

[43] Nicolas Dintzner. Safe Evolution Patterns for Software Product Lines. In *Proceedings of the International Conference on Software Engineering - Volume 2*, pages 875–878. IEEE, 2015.

[44] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. Analysing the Linux Kernel Feature Model Changes using FMDiff. *Software and Systems Modeling*, pages 1–22, 2015.

[45] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

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

[47] Marc Eaddy, Alfred Aho, and Gail C. Murphy. Identifying, Assigning, and Quantifying Crosscutting Concerns. In *Proceedings of the International Workshop on Assessment of Contemporary Modularization Techniques*, pages 2–7. IEEE, 2007.

[48] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[49] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[50] Michael D. Ernst, Greg J. Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[51] J. M. Favre. Understanding in the Large. In *Proceedings of the International Workshop on Program Comprehension*, pages 29–38. IEEE, 1997.

[52] Jean-Marie Favre. Preprocessors from an Abstract Point of View. In *Proceedings of the International Conference on Software Maintenance*, pages 329–339. IEEE, 1996.

[53] Dror G. Feitelson. Perpetual Development: a Model of the Linux Kernel Life Cycle. *Journal of Systems and Software*, 85(4):859–875, 2012.

[54] Eduardo Figueiredo, Bruno Carreiro da Silva, Cláudio Sant'Anna, Alessandro F. Garcia, Jon Whittle, and Daltro José Nunes. Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *Proceedings of the International Conference on Program Comprehension*, pages 138–147. IEEE, 2009.

[55] Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, Thiago T. Bartolomei, Walter Cazzola, and Alessandro Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE, 2008.

[56] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

[57] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[59] Alejandra Garrido and Ralph Johnson. Analyzing Multiple Configurations of a C Program. In *Proceedings of the International Conference on Software Maintenance*, pages 379–388. IEEE, 2005.

[60] Paul Gazzillo and Robert Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 323–334. ACM, 2012.

[61] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142. IEEE, 2000.

[62] Emanuel Graf, Guido Zgraggen, and Peter Sommerlad. Refactoring Support for the C++ Development Tooling. In *OOPSLA Companion*, pages 781–782. ACM, 2007.

[63] G. Grothendieck. *Package 'sqldf'*, 2014. R package version 0.4-7.1.

[64] J. Guo, Y. Wang, P. Trinidad, and D. Benavides. Consistency Maintenance for Evolving Feature Models. *Expert Systems and Applications*, 39(5):4987–4998, 2012.

[65] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. In *Proceedings of the Working Conference on Software Architecture*, pages 45–54. IEEE, 2001.

[66] William A. Hetrick, Charles W. Krueger, and Joseph G. Moore. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. In *Proceedings of Symposium on Object-oriented Programming Systems, Languages, and Applications*, pages 798–804. ACM, 2006.

[67] Hannes Holdschick. Challenges in the Evolution of Model-Based Software Product Lines in the Automotive Domain. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 70–73. ACM, 2012.

[68] M. Hubert and E. Vandervieren. An Adjusted Boxplot for Skewed Distributions. *Computational Statistics & Data Analysis*, 52(12):5186–5201, 2008.

[69] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Lessenich, Martin Becker, and Sven Apel. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*, 21(2):449–482, 2016.

[70] Clemente Izurieta and James Bieman. The Evolution of FreeBSD and Linux. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 204–211. ACM, 2006.

[71] Mikolas Janota and Joseph Kiniry. Reasoning About Feature Models in Higher-Order Logic. In *Proceedings of the International Software Product Line Conference*, pages 13–22. IEEE, 2007.

[72] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. Minimally Invasive Migration to Software Product Lines. In *Proceedings of the International Software Product Line Conference*, pages 203–211. IEEE, 2007.

[73] M. Tim Jones. Anatomy of the Linux Kernel. *IBM Developer Works*, 2007.

[74] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute, 1990.

[75] Christian Kästner and Sven Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 9 2009.

[76] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference*, pages 223–232. IEEE, 2007.

[77] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 311–320. ACM, 2008.

[78] Christian Kästner, Sven Apel, and Klaus Ostermann. The Road to Feature Modularity? In *Proceedings of the International Software Product Line Conference*, pages 5:1–5:8. ACM, 2011.

[79] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 805–824. ACM, 2011.

[80] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A Variability-aware Module System. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, pages 773–792. ACM, 2012.

[81] Kbuild. The Kernel Build Infrastructure. `http://www.kernel.org/doc/Documentation/kbuild`. Last seen: June 6, 2016.

[82] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

[83] Chang Hwan Peter Kim and Krzysztof Czarnecki. Synchronizing Cardinality-Based Feature Models and Their Specializations. In *European Conference on Model Driven*

Architecture Foundations and Applications, volume 3748 of *Lecture Notes in Computer Science*, pages 331–348. Springer, 2005.

[84] G. Krone, M.; Snelting. On the Inference of Configuration Structures from Source Code. In *6th International Conference on Software Engineering*, pages 49–57. IEEE, 1994.

[85] Partha Kuchana. *Software Architecture Design Patterns in Java.* Auerbach Publications, 2004.

[86] Duc Le, E. Walkingshaw, and M. Erwig. #ifdef Confirmed Harmful: Promoting Understandable Software Variation. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 143–150. IEEE, 2011.

[87] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the International Conference on Software Engineering*, pages 105–114. ACM, 2010.

[88] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. Morpheus: Variability-aware Refactoring in the Wild. In *Proceedings of the International Conference on Software Engineering*, pages 380–391. IEEE, 2015.

[89] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proceedings of the International Conference on Aspect-oriented Software Development*, pages 191–202. ACM, 2011.

[90] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013.

[91] Rafael Lotufo, Steven She, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. Evolution of the Linux Kernel Variability Model. In *Proceedings of the International Conference on Software Product Lines: Going Beyond*, pages 136–150. Springer, 2010.

[92] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power Laws in Software. *Transactions on Software Engineering and Methodologies*, 18(1):2:1–2:26, 2008.

[93] Robert Love. *Linux Kernel Development.* Addison Wesley, 3rd edition, 2010.

[94] Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the International Conference on Software Product Lines*, pages 176–187. Springer, 2002.

[95] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static Techniques for Concept Location in Object-Oriented Code. In *Proceedings of the International Workshop on Program Comprehension*, pages 33–42. IEEE, 2005.

[96] Anthony Massa. *Embedded Software Development with eCos.* Prentice Hall, 2002.

[97] Kim Mens and Tom Tourwé. Evolution Issues in Aspect-Oriented Programming. In *Software Evolution*, pages 203–232. Springer, 2008.

[98] David S. Moore, George P. McCabe, and Bruce Craig. *Introduction to the Practice of Statistics.* W. H. Freeman, 6th edition, 2009.

[99] Freddy Munoz, Benoit Baudry, Romain Delamare, and Yves Le Traon. Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study. In *Proceedings of the International Conference on Software Maintenance*, pages 137–146, 2009.

[100] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the International Conference on Software Engineering*, pages 140–151. ACM, 2014.

[101] Sarah Nadi and Ric Holt. Mining Kbuild to Detect Variability Anomalies in Linux. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 107–116. IEEE, 2012.

[102] Sarah Nadi and Ric Holt. The Linux Kernel: A Case Study of Build System Variability. *Journal of Software: Evolution and Process*, 2013.

[103] Colin J. Neill and Philip A. Laplante. *Antipatterns: Identification, Refactoring, and Management.* CRC Press, 2005.

[104] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulezsa, and Paulo Borba. Investigating the Safe Evolution of Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 33–42. ACM, 2011.

[105] P. Oliveira, M.T. Valente, and F. Paim Lima. Extracting Relative Thresholds for Source Code Metrics. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pages 254–263. IEEE, 2014.

[106] Yoann Padioleau. Parsing C/C++ Code Without Pre-processing. In *Proceedings of the International Conference on Compiler Construction*, pages 109–125. Springer, 2009.

[107] Leonardo Passos, Krzysztof Czarnecki, and Andrzej Wąsowski. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM, 2012.

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

[109] Andreas Pleuss, Goetz Botterweck, Deepak Dhungana, Andreas Polzer, and Stefan Kowalewski. Model-Driven Support for Product Line Evolution on Feature Level. *Journal of Systems and Software*, 85(10):2261–2274, 2012.

[110] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Sven Apel, and Krzysztof Czarnecki. Does Feature Scattering Follow Power-law Distributions?: An Investigation of Five Pre-processor-based Systems. In *Proceedings of the International Workshop on Feature-Oriented Software Development*, pages 23–29. ACM, 2014.

[111] Rodrigo Queiroz, Leonardo Passos, Marco Túlio Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems. *Software and Systems Modeling*, pages 1–20, 2015. First Online.

[112] Martin P. Robillard and Gail C. Murphy. Representing Concerns in Source Code. *Transactions on Software Engineering Methodologies*, 16(1), 2007.

[113] Marko Rosenmüller, Sven Apel, Thomas Leich, and Gunter Saake. Tailor-made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering*, 68(12):1493–1512, 2009.

[114] Marko Rosenmüller, Norbert Siegmund, Horst Schirmeier, Julio Sincero, Sven Apel, Thomas Leich, Olaf Spinczyk, and Gunter Saake. FAME-DBMS: Tailor-made Data

Management Solutions for Embedded Systems. In *Proceedings of the EDBT Workshop on Software Engineering for Tailor-made Data Management*, pages 1–6. ACM, 2008.

[115] Julia Rubin and Marsha Chechik. A Framework for Managing Cloned Product Variants. In *Proceedings of the International Conference on Software Engineering*, pages 1233–1236. IEEE, 2013.

[116] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing Cloned Variants: A Framework and Experience. In *Proceedings of the International Software Product Line Conference*, pages 101–110. ACM, 2013.

[117] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing Forked Product Variants. In *Proceedings of the International Software Product Line Conference*, pages 156–160. ACM, 2012.

[118] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley & Sons, 1st edition, 2012.

[119] Sandro Schulze, Oliver Richers, and Ina Schaefer. Refactoring Delta-oriented Software Product Lines. In *Proceedings of the Annual International Conference on Aspect-Oriented Software Development*, pages 73–84. ACM, 2013.

[120] Markus Schumacher, Eduardo Fernandez, Duane Hybertson, and Frank Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[121] Christoph Seidl, Florian Heidenreich, and Uwe Aßmann. Coevolution of Models and Feature Mapping in Software Product Lines. In *Proceedings of the International Software Product Line Conference*, pages 76–85. ACM, 2012.

[122] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[123] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. The Variability Model of the Linux Kernel. In *Proceedings of the International Workshop on Variability Modelling of Software-intensive Systems*, pages 45–51. Universität Duisburg-Essen, 2010.

[124] Sharon Simmons, Dennis Edwards, Norman Wilde, Josh Homan, and Michael Groble. Industrial Tools for the Feature Location Problem: an Exploratory Study. *Journal of Software Maintenance and Evolution*, 18(6):457–474, 2006.

[125] Henry Spencer and Geoff Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Security Symposium*. Usenix Association, 1992.

[126] Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Software Product Lines*, volume 3154, pages 34–50. Springer, 2004.

[127] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the European Software Engineering Conference*, pages 166–175. ACM, 2005.

[128] Jing Sun, Hongyu Zhang, and Hai Wang. Formal Semantics and Verification for Feature Modeling. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 303–312. IEEE, 2005.

[129] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the International Conference on Software Engineering*, pages 650–660. IEEE, 2012.

[130] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 4th edition, 2002.

[131] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119. ACM, 1999.

[132] Reinhard Tartler, Julio Sincero, Christian Dietrich, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Revealing and Repairing Configuration Inconsistencies in Large Scale System Software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551, 2012.

[133] Thomas Thüm, Don Batory, and Christian Kästner. Reasoning About Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering*, pages 254–264. IEEE, 2009.

[134] Thomas Thüm, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference*, pages 191–200. IEEE, 2011.

[135] Jenifer Tidwell. *Designing Interfaces.* O'Reilly, 2010.

[136] Christian Tischer, Andreas Muller, Thomas Mandl, and Ralph Krause. Experiences from a Large Scale Software Product Line Merger in the Automotive Domain. In *Proceedings of the International Software Product Line Conference*, pages 267–276. IEEE, 2011.

[137] Marco Tulio Valente, Virgilio Borges, and Leonardo Passos. A Semi-Automatic Approach for Extracting Software Product Lines. *IEEE Transactions on Software Engineering*, 38(4):737–754, 2012.

[138] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative Analysis of Evolving Software Systems Using the Gini Coefficient. In *Proceedings of the International Conference on Software Maintenance*, pages 179–188. IEEE, 2013.

[139] Bart Veer and John Dallaway. The eCos Component Write's Guide. `http://ecos.sourceware.org/ecos/docs-3.0/pdf/ecos-3.0-cdl-guide-a4.pdf`, 2001. Last seen: June 6, 2016.

[140] Sreekrishnan Venkateswaran. *Essential Linux Device Drivers.* Prentice Hall, 1st edition, 2008.

[141] Marian Vittek. Refactoring Browser with Preprocessor. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 101–. IEEE, 2003.

[142] Bo Wang, Leonardo Passos, Yingfei Xiong, Krzysztof Czarnecki, Haiyan Zhao, and Wei Zhang. SmartFixer: Fixing Software Configurations Based on Dynamic Priorities. In *Proceedings of the International Software Product Line Conference*, pages 82–90. ACM, 2013.

[143] Bo Wang, Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Wei Zhang, and Hong Mei. A Dynamic-Priority Based Approach to Fixing Inconsistent Feature Models. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 181–195. Springer, 2010.

[144] R. Wheeldon and S. Counsell. Power Law Distributions in Class Relationships. In *Proceedings of the International Workshop on Source Code Analysis and Manipulation*, pages 45–54. ACM, 2003.

[145] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proceedings of the International Software Product Line Conference*, pages 225–234. IEEE, 2008.

[146] Norman Wilde and Michael C. Scully. Software Reconnaissance: Mapping Program Features to Code. *Journal of Software Maintenance*, 7(1):49–62, 1995.

[147] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering.* Springer, 2012.

[148] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the USENIX Security Symposium*, pages 17–31. USENIX Association, 2002.

[149] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. Generating Range Fixes for Software Configuration. In *Proceedings of the International Conference on Software Engineering*, pages 58–68. IEEE, 2012.

[150] Haihan Yin, Christoph Bockisch, and Mehmet Aksit. A Fine-grained Debugger for Aspect-oriented Programming. In *Proceedings of the Annual International Conference on Aspect-oriented Software Development*, pages 59–70. ACM, 2012.

[151] Shurui Zhou, Jafar Al-Kofahi, Tien N. Nguyen, Christian Kästner, and Sarah Nadi. Extracting Configuration Knowledge from Build Files with Symbolic Analysis. In *Proceedings of the International Workshop on Release Engineering*, pages 20–23. IEEE, 2015.