The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time

Gabriela K. Michelon[1,2], Wesley K. G. Assunção[1,3], David Obermann[1], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]
[1]Institute of Software Systems Engineering, Johannes Kepler University Linz, Austria
[2]LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria
[3]DI - Pontifical Catholic University of Rio de Janeiro and PPGComp - Western Paraná State University, Brazil
[4]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany

Abstract

Feature annotation based on preprocessor directives is the most common mechanism in Highly-Configurable Software Systems (HCSSs) to manage variability. However, it is challenging to understand, maintain, and evolve feature fragments guarded by `#ifdef` directives. Yet, despite HCSSs being implemented in Version Control Systems, the support for evolving features in space and time is still limited. To extend the knowledge on this topic, we analyze the feature life cycle in space and time. Specifically, we introduce an automated mining approach and apply it to four HCSSs, analyzing commits of their entire development life cycle (13 to 20 years and 37,500 commits). This goes beyond existing studies, which investigated only differences between specific releases or entire systems. Our results show that features undergo frequent changes, often with substantial modifications of their code. The findings of our empirical analyses stress the need for better support of system evolution in space and time at the level of features. In addition to these analyses, we contribute an automated mining approach for the analysis of system evolution at the level of features. Furthermore, we also make available our dataset to foster new studies on feature evolution in HCSSs.

CCS Concepts: • **Software and its engineering → Software product lines**; **Traceability**; **Software reverse engineering**; *Reusability*.

Keywords: mining, feature evolution, feature revision

ACM Reference Format:
Gabriela K. Michelon[1,2], Wesley K. G. Assunção[1,3], David Obermann[1], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]. 2021. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486609.3487195

GPCE '21, October 17–18, 2021, Chicago, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-9112-2/21/10...$15.00
https://doi.org/10.1145/3486609.3487195

1 Introduction

Highly-Configurable Software Systems (HCSSs) allow users to customize program behavior and create different variants by selecting configuration options that address different requirements [20]. Feature annotations rely on code fragments guarded by `#ifdef` preprocessor directives and are widely used to control such customization and implement software product lines (SPLs) [47]. Further, preprocessor directives enable alternating implementations and supporting multiple platforms and operating systems across different environments [29]. However, annotation-based code typically makes it harder to understand and maintain software features, receiving strong criticism regarding the separation of concerns, error proneness, and code obfuscation [26, 29]. Therefore, the evolution of HCSSs in *space*, i.e., features included or excluded, and in *time*, i.e., different revisions of features due to modifications of the system's implementation, is a challenging and complex task [6, 33, 53].

For managing system evolution in time, developers of HCSSs use Version Control Systems (VCSs), which keep track of changes over time. However, HCSSs implemented in VCSs do not offer proper support to retrieve and analyze the changes at the level of features as often multiple features are committed at once [18, 27]. Further, features are often tangled and scattered over different files. Thus, determining which features changed when multiple files and lines were modified in a single commit can be an infeasible and error-prone task [28]. Existing studies thus recommend research on recovering and tracking the evolution of features managed in version control systems (VCSs) [23, 29].

Based on the aforementioned, a better comprehension of the entire feature life cycle, i.e., when and how features are introduced, removed, and revised in both space and time is of paramount importance for improving the management of HCSSs in VCSs and a prerequisite for developing tools that ease maintenance and evolution. However, there is no empirical work analyzing the life cycle of features throughout all commits and for all releases of HCSSs.

Therefore, this paper provides a comprehensive analysis of the feature life cycle in space and time, revealing the challenges and limitations of current mechanisms. We provide insights regarding the complexity of feature implementations throughout the revision history. In particular, we investigate which and how features changed from one commit to another and introduce an automated approach for mining repositories of HCSSs managed in VCSs.

Our approach is the first using a constraint satisfaction problem (CSP) solver to reliably identify interacting features or features depending on the execution of other features [5]. It also considers corner cases [28] for mining variability in space and time. Other approaches only capture features annotated in presence conditions [9, 19, 26, 47] and do not solve constraints of complex expressions, involving Boolean, arithmetic operations, and comparisons. Our approach also enables mining metrics of feature changes, again going beyond previous work, which only presents metrics of feature characteristics at one point in time or for few commits [9, 19, 26, 29, 40, 54].

We applied our approach to four subject systems and analyzed their entire life cycle of active development, ranging from 13 to 20 years, altogether covering 37,500 Git commits. We adopted well-established metrics [11, 47] to analyze the variability of the systems and the implementation complexity of the features: the scattering degree counting the number of variation points; the tangling degree counting the number of features interacting in variation points; the nesting depth counting the number of variation points inside a variation point; and the number of top-level branches, i.e., the number of variation points without any outside enclosing variation points. We also analyzed the correlation of these metrics across the feature life cycle to understand, e.g., if a metric is correlated with other metrics. Adding the time dimension enabled us to find different information compared to previous work, e.g., that features scattered over many variation points lead to more revisions over time than features with complex tangling.

Our main contributions are: (1) an automated approach for analyzing the evolution of features in space and time and for computing the characteristics of changes and feature implementation complexity of HCSSs managed in VCSs[1]; (2) an empirical study of the feature life cycle, which provides findings and insights on how developers maintaining and evolving HCSSs can benefit from the mined information; and (3) a dataset covering the entire development history of four open-source systems from different domains, with a total of 37,500 commits from up to 20 years[2].

[1]https://github.com/GabrielaMichelon/git-ecco

[2]http://doi.org/10.5281/zenodo.4158191

The remainder of this paper is organized as follows: Section 2 illustrates the complexity of manually identifying changed features and their revision characteristics in HCSSs. Section 3 presents the automated mining approach for feature evolution analysis. Section 4 describes the research method of our empirical study. Section 5 presents the results of our analysis. Section 6 discusses findings and insights. Threats to validity are pointed out in Section 7. Section 8 highlights how our work differs from previous research. Section 9 presents conclusions and future work.

2 Problem Statement

Our work investigates the feature life cycle of HCSSs. Features in this context are configuration options represented by external literals, i.e., macro names never defined in the source code by a `#define` directive. Therefore, the features considered in our analysis are variation points that represent system functions as well as features needed for debugging and testing. Also, we generically use the term *ifdefs* to refer to the variation points, i.e., variability enclosed by the C preprocessor annotations in conditional blocks `#ifdef`, `#ifndef`, `#if`, `#elif`, `#else`.

Although developer guidelines recommend small and cohesive commits, developers in practice often commit multiple independent changes to VCSs at once [4]. Understanding the code of such multi-feature commits is much harder compared to the code of smaller commits representing independent changes. Thus, the analysis of which features changed and in what way is even more complex in HCSSs, as developers need to consider preprocessor directives, including `#define` and `#include`, as well as expressions in conditional blocks of code [28]. Furthermore, the implementation of features developed within preprocessor directives is often highly complex, scattered across many files, and comprising many blocks of code. Even expanding macros in conditions can make it harder to reason about problems in the code, as it may exclude parts of the source code depending on which and how macros are defined. Features can be tangled when involved in the same variation points, i.e., in the same blocks of *ifdefs*. Nonetheless, features can also be nested with other features when blocks of code guarded by *ifdefs* are enclosed by other *ifdefs* [47]. In addition, it is necessary to treat `#else` and `#ifndef` directives when assigning changes to features as ignoring such corner cases can lead to faulty assumptions about system variability [28]. As a consequence, developers are often overwhelmed when trying to understand large code changes during maintenance and evolution.

To illustrate the difficulty of manually analyzing feature changes, let us consider commit #2677[3] from release libssh-0.6.0 (i.e., Git Tag [8]) of the LibSSH system. This commit changed 22 source files of six features. As the commit message says, the developer removed the *enter_function()* and

[3]https://gitlab.com/libssh/libssh-mirror/-/commit/c64ec43

the *leave_function()* from all the 22 files, thereby affecting single lines of multiple blocks of code that belong to multiple *ifdefs*. However, the commit message does not mention the affected features. This would be difficult to analyze manually, given that the 268 additions and 495 deletions are spread over at least five features and 55 macros. Thus, such kinds of commits require an in-depth and recursive analysis of where macros are defined and which ones are responsible for activating blocks of code.

3 Automated Mining and Computation of Metrics

Figure 1 shows the steps performed by our automated mining approach to retrieve the information used for our empirical study. The mining approach computes the metrics (detailed in Section 4.2) for each feature appearing in each commit: we begin by cloning the repository of a system selected for analyses (Step 1). Then, we collect information from all Git commits of all releases, i.e., Git tags [8] (Step 2). Afterwards, we partially pre-process the checked out commit files, i.e., annotated macros are expanded such that annotated conditional blocks only consist of literals (Step 3). This allows determining the corresponding values for macros that are defined at some point in the code by a `#define` directive. These values are important to compute possible solutions via the Choco CSP Solver [45], which receives as input an expression corresponding to a specific block of code of interest to collect revision or change characteristics and a queue of implications for each particular macro that influences in some way the specific block to be executed. The files resulting from expanding macros in annotated conditional blocks are then used to create a tree structure (Step 4) to mine information about the features. The tree represents the entire contents of one commit and consists of three major nodes: conditional nodes representing a block of code that can be surrounded by `#if/#ifdef/#ifndef/#else/#elif` and its `#endif`, `#define` nodes, as well as, `#include` nodes. Our approach for computing metrics relies on a tree structure of preprocessor directive nodes [32], which reduces the computational costs of the analysis [21, 24].

Steps 5 to 7 concern the computation of metrics and the mining of information: Step 5 is necessary to obtain the features of the system. This step first collects all macros of conditional blocks and all `#defines` of all files for every commit of a release. Then, we look for the macros that have never been defined within the source code, i.e., can only be set externally by the user from the command line. In this way, we obtain the macros that can be considered features of the system. Every block is identified and for each block that has a Git diff [8], i.e., fragments of code that differ for the same file from one commit to another (Step 6), we compute metrics on the *change characteristics* (Step 7). We obtain the fragment differences with Git patches, i.e., representations of

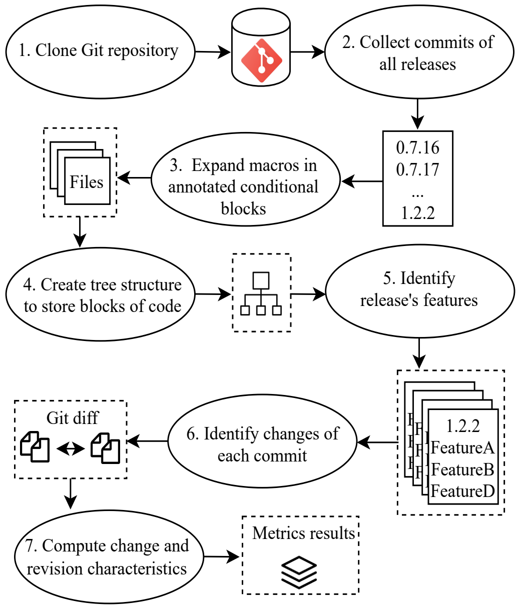

1. Clone Git repository
2. Collect commits of all releases
0.7.16
0.7.17
...
1.2.2
3. Expand macros in annotated conditional blocks
Files
4. Create tree structure to store blocks of code
5. Identify release's features
1.2.2
FeatureA
FeatureB
FeatureD
6. Identify changes of each commit
Git diff
7. Compute change and revision characteristics
Metrics results

Figure 1. Automated approach for mining the life cycle of features.

the differences between two text files in a line-oriented way as computed by a diff utils library [3]. We also compute *revision characteristics* (Step 7) for every block of code presented in the source files of a commit. The resulting metrics from our mining approach are presented in Table 1 (Section 4.2).

Our approach uses a high level of abstraction to compute metrics of features and analyzes only the annotated blocks of code and not the abstract syntax tree (AST), which makes it computationally less expensive. Thus, the runtime performance of our approach is $O(n)$, growing with the number of variation points. The Choco Solver is among the fastest constraint problem (CP) solvers available [45]. For every commit, the runtime performance for computing the metrics varies from a few seconds to a couple of minutes due to the number of features and the constraints that a block of code can involve in the CP. Table 2 shows the average time to compute the metrics in minutes per release (R_P). SQLite took longer than other systems due to the higher number of commits per release, which increases the total time needed to compute metrics per release. However, once the metrics for the entire life cycle of the system are computed, future commits can be analyzed individually to get results faster.

4 Empirical Study Design

This section explains the study goal and research questions, as well as the metrics and subject systems used in our empirical work.

4.1 Study Goal and Research Questions

Study goal: We aim to *analyze the life cycle of features, in terms of when and how they are introduced, removed, and revised in both space and time, during each commit of system development and evolution.*

RQ1: How often are features revised through their life cycle? We investigate the frequency of feature revisions along with system evolution. Also, we analyze how developers introduce and remove features in each commit, leading to system evolution in space. For each commit of the subject systems, we analyze the new and removed features, as well as the features that were revised by a change. We compute the number of newly introduced, removed, and changed features within each release.

RQ2: What is the scope of feature revisions? We analyze each patch of change of a feature in a commit, thereby assessing the impact and characteristics of the changes. In particular, we determine the characteristics of the changes of each revision of the features. We compute for each commit the number of variation points, tangled features, files, lines added and/or removed based on the Git diffs between one commit to another for each feature. We also investigate how the characteristics of the feature changes and the feature implementation evolve over time. So far, no study combines the analysis of the revision characteristics of features and the characteristics of each change to determine the impact of revisions on the implementation complexity of features along with system evolution. To answer RQ2, we compute metrics about *change characteristics* (cf. Table 1 in Section 4.2).

RQ3: How do feature revisions affect the complexity of feature implementation? We aim to understand how features evolve over time, i.e., to what extent the implementation of a revision of a feature changes from one commit and from one release to another. Answering RQ3 helps to understand how revisions of features are implemented and how their implementation complexity changes over time. To answer RQ3 we compute *revision characteristics*, such as the scattering and tangling degrees (cf. Table 1 in Section 4.2).

4.2 Metrics

We selected the metrics based on a literature survey [13, 47] on the usefulness of feature metrics for preprocessor-based systems, which regards scattering degree, tangling degree, and nesting degree as useful metrics to measure feature implementation complexity. Also, the number of nested `#ifdef` blocks can influence the understandability of feature code: when developers modify features inside a block, they have to analyze each implication for all features nested in a specific block. Thus, as an approximation of the implementation complexity of a feature, we also compute the number of variation points of a feature revision that are top-level branches (NOTLB) and non-top-level branches (NONTLB) in addition to scattering degree (SD) and nesting depth (ND). The tangling degree (TD) is a metric widely used to analyze how many features are tangled in an implementation, to estimate the complexity of modifying features [2, 26].

Table 1. Definition of metrics computed by the approach: Change characteristics concern the scope of a feature modification, while revision characteristics refer to feature complexity of a specific commit.

Change characteristics	
LOC A	Number of lines added for each feature revision's patches of change
LOC R	Number of lines removed for each feature revision's patches of change
TD	Number of feature revisions in variation points for each feature revision's patches of change
SD #IFs	Number of patches of changes affecting the variation points of a feature revision
SD File	Number of files impacted for each feature revision's patches of change
Revision characteristics	
LOC	Number of lines of code of a feature revision
SD #IFs	Number of `#ifdef`/`#elif#if` variation points of a feature revision
SD #NIFs	Number of `#ifndef`/`#else` variation points of a feature revision
SD File	Number of files with variation points of a feature revision
TD	Number of feature revisions in its `#ifdef`/`#if`/ `#ifndef`/`#elif`/ `#else` variation points of a feature revision
ND	Number of `#ifdef`/`#if`/`#ifndef`/`#elif`/`#else` variation points inside its variation point of a feature revision
NOTLB	Number of `#ifdef`/`#if`/`#ifndef`/`#elif`/`#else` variation points without any outside enclosing variation point, i.e., the number of top-level branches of a feature revision
NONTLB	Number of `#ifdef`/`#if`/`#ifndef`/`#elif`/`#else` variation points enclosed by another variation point, i.e., the number of non-top-level branches of a feature revision

Let us use an example containing corner cases to illustrate how we compute these metrics and why a CSP solver is needed. Listing 1 shows a code snippet in the first commit (Commit #0) and Listing 2 shows the code snippet of the same file, but in the second commit (Commit #1). In this illustrative example, we assume the macros `A` and `Y` as part of the set of features, which can only be set by the user. The other macros `X`, `B` and `C` (non-boolean) are defined within the source code and are not considered features as they cannot be set externally. We create logic formulas to represent feature interactions and feature implications for every block of code to know which features are responsible to activate it and to be able to compute metrics for every commit.

Lines 11 and 17 changed in Commit #1. Analyzing the change in line 11, we cannot simply assume that `B` and `C` have changed, as they are not features of the system. We have to walk up the file to see which features are defining `B` and `C`. Also, we need to analyze the feature implication of the outermost block of code in case it exists. The change will

Listing 1. Commit #0.

```
1  #if Y
2      #define X
3  #endif
4
5  #if X
6      #define B
7      #define C 9
8  #endif
9
10 #if B && C > 5
11     <code>
12 #endif
13
14 #ifndef A
15     <code>
16 #else
17     <code>
18 #endif
```

Listing 2. Commit #1.

```
1  #if Y
2      #define X
3  #endif
4
5  #if X
6      #define B
7      #define C 9
8  #endif
9
10 #if B && C > 5
11     <changed code>
12 #endif
13
14 #ifndef A
15     <code>
16 #else
17      <changed code>
18 #endif
```

be assigned to the feature that can activate this block of code. Thus, walking up the file, we have a feature implication with the macro X, which defines B and sets a value greater than 5 to C. Also, the macro X is defined in another block of code, containing the feature Y in the condition expression.

The constraints built are handed to the CSP solver, which gives to us a solution that satisfies the constraint problem, i.e., which features have to be selected to activate a block of code. For the change in line 11, the constraints are defined as follows: $(Y \implies X) \wedge (X \implies B) \wedge (X \implies (C = 9)) \wedge (B \wedge (C > 5)) \wedge X \wedge Y$. This formula is satisfied when Y = T, i.e., when feature Y is selected as X, B and C are not considered as features. Then, the change is assigned to feature Y and the metrics for change characteristics are computed as follows: LOC A = 1 (one line added in Commit #1), LOC R = 1 (one line removed from Commit #0), SD #IFs = 2 (two patches of changes affecting a variation point, one line removed in Commit #0 and one line added in Commit #1), SD File = 1 (the change was in one file), TD = 2 (involving the features Y and BASE, which represents the core of the system).

With the aforementioned example, we can see that only computing feature characteristics cannot show that feature Y changed in Commit #1 as the feature characteristics remain the same as in Commit #0. This is also because the change was performed in a variation point (lines 10-12) that is impacted indirectly by this feature. The revision characteristics of feature Y are as follows: LOC = 1 (line 2), SD #IFs = 1 (variation point lines 1-3), SD #NIFs = 0, SD File = 1, TD = 1 (BASE is the only feature impacting or interacting to activate its variation point from lines 1-3), ND = 0, NOTLB = 1 (it is a top-level branch variation point as there is no other outermost block of code wrapping lines 1-3), NONTLB = 0.

Existing analyses still assign the feature constant (macro name) to the negated directives. For example, they assign feature A to the block `#ifndef A` in lines 14-16 (Listing 1) when computing the metrics, which is incorrect [28]. Another corner-case example is the code in the `#else` block (lines 16-18), which is only executed if feature A is selected. However, existing studies measuring scattering degree and tangling degree are limited to counting the macro names in source code, as shown in [28]. In our example, a change in the code of the `#ifndef` does not belong to feature A, instead, a change in the `#else` block of code does. We thus compute the change characteristics in line 17 for feature A as follows: LOC A = 1, LOC R = 1, SD #IFs = 2 (two patches of changes affecting a variation point, the line removed in Commit #0 and the line added in Commit #1), SD File = 1, TD = 2 (involving A and BASE). The revision characteristics of feature A after a change in Commit #1 remain the same as in Commit #0 as follows: LOC = 1, SD #IFs = 0, SD #NIFs = 1, SD File = 1, TD = 1, ND = 0, NOTLB = 1, NONTLB = 0. The block of code in line 14 is computed as part of the feature BASE, rather than A. Then, the feature characteristics of BASE are as follows: LOC = 2, SD #IFs = 1 (`#else` line 16), SD #NIFs = 0, SD File = 1, TD = 1, ND = 0, NOTLB = 1, NONTLB = 0.

4.3 Subject Systems

Our study investigated the evolution history, i.e., all commits of all releases of four open-source C preprocessor-based systems of different sizes and from different domains (Table 2). These systems were selected due to their substantial evolution history, their active and collaborative development, their popularity, as well as their use in previous research [14, 16, 26, 30, 54]. Table 2 presents the number of releases (N_R), the year of inception, the total number of commits (N_C), the total lines of code (LOC), the number of features (N_F) in the last release, and the runtime average (R_P) in minutes for mining and computing metrics per release, i.e., involving multiple commits of the systems.

Table 2. Overview of the subject systems.

System	Domain	N_R	Since	N_C	LOC	N_F	R_P
Bison	Parser	105	2002	6,991	39,904	83	9.6
LibSSH	Network library	48	2005	5,022	110,590	121	8.5
Irssi	Chat Client	69	2007	5,331	85,325	57	20.6
SQLite	Database system	113	2000	20,090	173,714	384	85.4

5 Results and Analysis

We present findings based on the results of our automated analysis, organized by our three research questions.

5.1 RQ1. How often are features revised through their life cycle?

The numbers of introduced and removed features for all releases of the Bison, LibSSH, and Irssi system show that features are typically hardly removed anymore after being introduced. In case of the SQLite system features were removed in 33 of 113 releases analyzed. In particular, analyzing the in-space feature evolution of SQLite, we observed that features were removed until the 78th release. When analyzing the removed and added features in more detail, however, we noticed that the features removed first were then reintroduced in commits of the same release. This happened for many features in SQLite, but mostly for features used for test purposes. This confirms the findings of Berger et al. [7], who showed for industrial systems that features are also used for testing, debugging, build, optimization, deployment, simulation, or monitoring.

Interestingly, we could see in all subject systems that features are changed while new features are introduced, showing a tendency that releases with a higher number of features evolving in space usually also have a higher number of features evolving in time. Overall, in some commits of the releases, features were removed, introduced, and changed, representing constant system evolution in space and time. In previous work [44] evaluating the Linux kernel 6% of the commits either added or removed features over the releases range analyzed (v2.6.25-v3.3). Despite the Linux kernel being much larger in terms of features (13,000) and LOC (more than 33,000 C files with over 10 million source LOC), our systems have a similar evolution in space (3%-7%).

Kröher et al. [22] show that features in the Linux kernel change significantly less often when not considering the changes in the core of the system. About 7% of all commits analyzed introduce changes to variability information, i.e., features annotated in *ifdefs*. Only up to 8% of all commits were changes affecting only the code of features. Yet, changes in both BASE and other features' code co-occur in 2% to 11% of the systems' commits. Based on the information gathered with our automated mining approach, the effort for analysis and verification can be avoided for about 80% of the commits that do not introduce changes in the variability of our subject systems. Thus, our analysis of how features evolve along the commits of the four subject systems, as observed in Figure 2, confirms their findings, which shows that the degree of evolution in time of the Linux kernel can be generalized to smaller SPLs, such as our subject systems.

Taking into account the characteristics of each subject system (cf. Table 2), we see that the number of commits, lines of code, or the number of features does not explain the number of feature changes. For example, LibSSH is smaller than SQLite but has more single commits involving changes to specific features (8% compared to 3% of SQLite) and single commits involving changes in both features and BASE code (19% compared to 13% of SQLite). When analyzing the commits involving more than one feature change, we could see that changes related to 2-5 features happened in 1,384 commits (28%) of the LibSSH system; in 3,600 commits (18%) of SQLite; in 202 commits (4%) of Irssi; and 364 commits

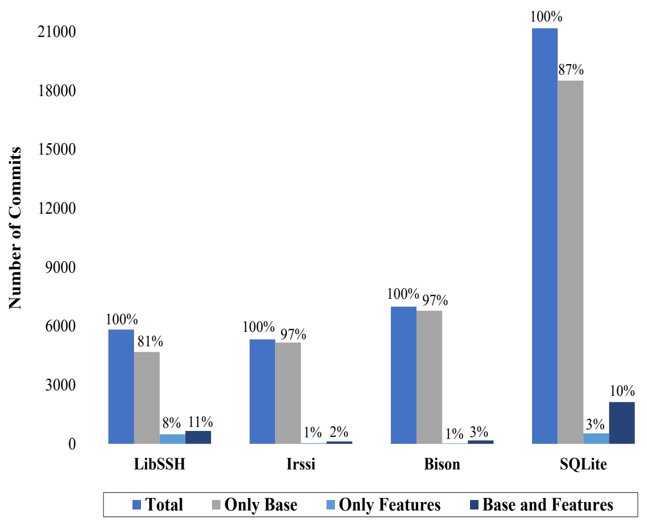


Figure 2. Number of commits based on systems' changes to feature types.

(5%) of Bison. These numbers confirm that changes over commits are tangled [18] and analyzing the evolution of features manually might be infeasible. For instance, a single commit of Bison (1f65350) changed 103 files, with 491 additions and 24,579 deletions made to 66 features due to an upgrade to the latest versions of gnulib and Automake. From the aforementioned information, we thus formulate our first finding:

Finding 1. *Features are subject to evolution in space and time, for instance, due to new and enhanced functionality, refactoring, bug fixes, or testing. Often, multiple features are introduced and changed together in a single commit. This happened over all releases of the systems. The great majority of commits contains changes to the common implementation (BASE code), which confirms the findings for the Linux kernel [22]. However, there are also many commits regarding feature evolution. In addition, we could see specific commits with a large impact, e.g., a single commit in Bison impacted 66 features and 103 files. Interestingly, we noticed that in the case of SQLite, some of the most-changed features were defined for testing purposes.*

Answering RQ1. *Our analyses show a great diversity of how developers manage and evolve HCSSs in VCS over each commit. We observed commits covering changes in both space and time. We also observed commits in which changes affect multiple features and files, sometimes including dependencies and interactions between features. Curiously, in SQLite we even observed cases of constant evolution in space, i.e., inclusion and exclusion of features, to support testing tasks in this project.*

5.2 RQ2. What is the scope of feature revisions?

Table 3 presents the change characteristics across all commits, summarizing SD `#IFs`, SD Files, and TD. Following Queiroz et al.'s [47] threshold for feature characteristics, we considered change characteristics as complex if more than 15% of the features have a scattering degree (SD `#IFs`) ≥ 7 and more than 20% have a tangling degree (TD) ≥ 3. In our study, we used a tangling degree ≥ 3 for the feature expressions impacted by a change (instead of ≥ 2 as in [47]) because we always computed TD including the `BASE` code in the feature expression. Therefore, we can infer that LibSSH changes are complex in terms of scattering `#IFs` and tangling degree. The changes in other systems are also complex in terms of tangling degree (as more than 20% of the features have a TD ≥ 3). For the scattering degree of files (SD Files) we just assumed the same classification of SD `#IFs`, but in most cases Git delta's change, i.e., the code added or removed in a system just impacted few files (≤ 6).

Analyzing correlations of change characteristics allows understanding how changes to a feature impact complexity in terms of SD `#IFs`, SD File, and TD. We applied the Spearman correlation coefficient [51], as the sample data do not follow a normal distribution and a linear behavior, with a confidence interval of 95%, and classified it according to Evans [12]. Table 4 presents the results for the Irssi system, showing very strong correlations (the highlighted cells) of the change characteristics of the hotspot features changing more often. Correlations were computed for all possible pairs of LOC A, LOC R, SD `#IFs`, SD File, and TD of Git delta changes for hotspot features. When looking at the correlations of feature `HAVE_OPENSSL` (Table 4), we can see that when more LOCs were added in a block impacting this feature, more variation points of `#IFs` were impacted. From this, we can infer that changing a feature with a complex implementation may have a big impact on system implementation and other features.

Yet, taking into account the correlations of change characteristics of the feature `HAVE_OPENSSL`, we can also see that besides SD `#IFs` and LOC A also SD `#IFs` and LOC R are strongly correlated. When a change involves many SD `#IFs`, it not only means that code was added or changed, but also that lines were removed from its implementation. While in the case of feature `SSL_get_server_tmp_key` there is a strong correlation between LOC R and SD `#IFs`, there is a moderate negative correlation between SD `#IFs` and LOC A.

Table 3. Change characteristics in all commits of the system.

System	LibSSH	Irssi	Bison	SQLite
SD #IFs ≥ 7	19%	9%	13%	14%
TD ≥ 3	22%	29%	56%	38%
SD Files ≥ 7	1%	1%	1%	1%

Table 4. Correlation of change characteristics in Irssi features.

Feature	SSL	CAPSICUM	OPENSSL
LOC A/LOC R	-0.62	-0.42	0.54
LOC A/SD #IFs	-0.58	-0.08	0.80
LOC A/SD File	-0.66	0.06	-0.04
LOC A/TD	*	*	*
LOC R/SD #IFs	0.98	0.64	0.82
LOC R/SD File	0.71	0.42	0.32
LOC R/TD	*	*	*
SD #IFs/SD File	0.70	0.93	0.07
SD #IFs/TD	*	*	*
SD File/TD	*	*	*

SSL = **SSL_get_server_tmp_key**; **CAPSICUM** = **HAVE_CAPSICUM**; **OPENSSL** = **HAVE_OPENSSL**.

This shows that the Git deltas of a change across this feature's life cycle often affect the feature by removing existing lines instead of adding new lines to it.

We analyzed the commits for the points in time where the feature `SSL_get_server_tmp_key` changed. The strong correlation between changed SD #IFs and LOC R can be seen in commit 322625b[4], where the SD affected were deleted, and the removed lines were only the lines of the preprocessor directives, meaning that the code of this feature was moved to the `BASE`, affecting three files. Furthermore, when analyzing the characteristics of this feature, we see that it was introduced with an SD of four, while the SD was one in the last release of the system. The change characteristics of this feature indicate that it was not changed after the 62nd release of the Irssi system, and its revision characteristics also remain unchanged until the last release. Existing research on feature evolution does not evaluate such characteristics of changes. By doing that, we could observe that the code may change even if the feature revision characteristics do not change, and thus a feature of a specific commit does not have the same implementation as in another commit. Based on the change characteristics, we now present our second finding:

Finding 2. *Analyzing both feature changes and revision characteristics, i.e., implementation complexity, is important for understanding the feature life cycle. Interestingly, only knowing revision characteristics of features, e.g., the number of variation points or the nesting depth, is not enough to reveal the features and how they were changed in some cases. By analyzing the change characteristics, we could identify that features kept their characteristics even if their implementation changed.*

According to the number of changes, the feature `HAVE_OPENSSL` changed more often than other features of the Irssi system. We analyzed the commit messages of all

[4]https://github.com/irssi/commit/322625b

changes of this feature to understand the reasons for the changes, as the revision characteristics of this feature show that it is not interacting with other features (TD = 2) and it is in the variation points' top-level branches with no nested features. Thus, this feature has no interactions and there are no features implemented in its variation points. Hence, we know that when this feature changed, it was not due to changes in other features that could have affected the feature `HAVE_OPENSSL`. It is interesting for developers to know that changing existing variation points of a feature does not affect other features. We now present our third finding:

Finding 3. *Features are indirectly affected by changes in other features, mainly because of the nesting degree of features. For example, features of top-level branches, i.e., outermost variation points, in some cases have increasing numbers of lines of code due to nesting features that changed their code implementation.*

Answering RQ2. *Our study investigated the scope of feature revisions from many perspectives, e.g., regarding the number of variation points and files affected in a feature by a single commit and the kind of changes usually performed in a specific feature over time. Mining the change characteristics of a feature shows if the changes of a commit resulted in new variation points in additional files, or if code was removed or added in nested features. This information can be obtained by mining both the revision characteristics of features and the change characteristics of each revision along the feature life cycle. Although adding, changing, or removing lines of code is most of the time related to only one feature, sometimes it indirectly impacts the outermost variation points, namely top-level branches and dependencies to and interactions with other features.*

5.3 RQ3. How do feature revisions affect the complexity of feature implementation?

Table 5 shows that the implementation complexity of the features, according to the Queiroz et al.'s [47] threshold, is high when more than 15% of a system's features have an SD ≥ 7, which means that features are spread over many variation points. In their findings, feature scattering is highly skewed, which is what we could also observe in our subject systems. This is due to the scattering degrees reaching extreme values for large systems, such as the Linux kernel (max. SD = 2,698). In our subject systems, high SD #IFs was found for SQLite (max. SD #IFs = 239) and Bison (max. SD #IFs = 130).

In terms of SD `#IFs` less than 15% of features have a SD ≥ 7 for Bison and Irssi, which indicates comparably simple implementations in general. SQLite and LibSSH have a complex implementation because for more than 15% of their features SD `#IFs` are ≥ 7. Comparing to Linux kernel [42, 44], 20% of the features have high scattering, where 75% of the scattered

Table 5. Revision characteristics over the features' life cycle

	LibSSH	Irssi	Bison	SQLite
SD #IFs ≥ 7	24%	5%	10%	16%
SD #NIFs ≥ 7	4%	9%	6%	6%
SD Files ≥ 7	16%	3%	3%	9%
TD ≥ 3	16%	7%	31%	18%
ND ≥ 2	12%	7%	8%	7%
NOTLBs	58%	67%	39%	51%
NONTLBs	42%	33%	61%	49%

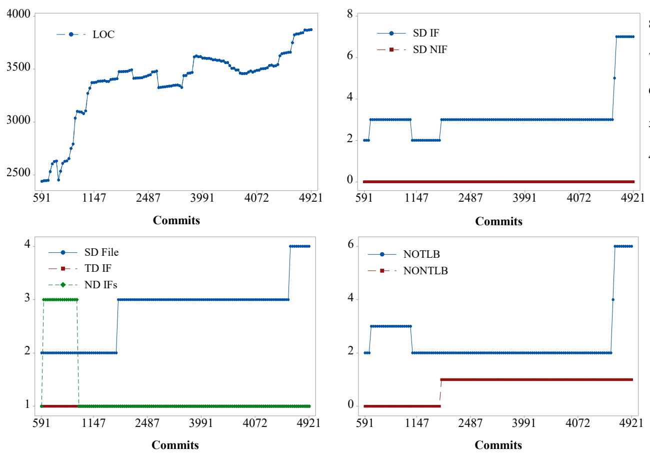
features have a SD ≥ 8, and are mainly features used for infrastructure and platform purposes, thus showing complex implementation.

Regarding the feature characteristics, we observed different evolution in comparison to the Linux kernel because features in our systems were introduced already scattered and did not remain constant over releases. We found that this happens for features with high impact on the system. For example, in LibSSH, the feature `WITH_SERVER` was changed multiple times to fix bugs such as memory leaks or security vulnerabilities. Other examples are debugging and testing features that often change, adding and removing variation points for debugging problems, and testing new features and bug fixes.

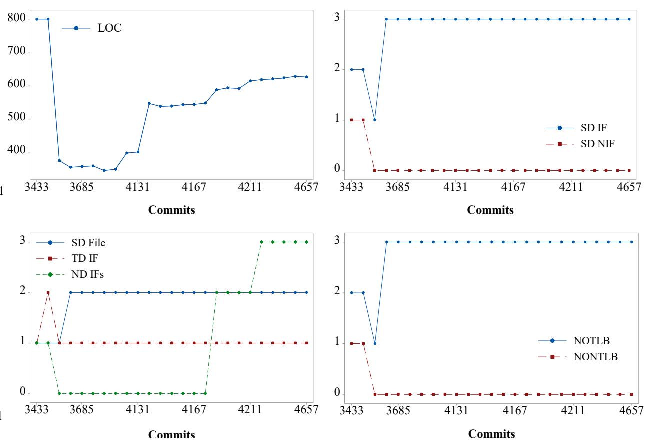
In terms of SD File, LibSSH also has more than 15% of the implementation of features with SD File ≥ 7. Bison is considered a complex system in terms of TD because it is the only system for which TD ≥ 3 for more than 20% of the features. The high TD means that the implementation of features is dependent on other features, and changing a tangled feature might affect these other features. ND ≥ 2 holds for at least 7% of features for all systems analyzed. For Bison, the NOTLBs is about 40% of the features' variation points, meaning more than 60% of features implementation are a non-top-level branch. Higher NONTLBs make it difficult to retrieve a feature's code for maintenance, and also the changes of features can have an impact on other features. Thus, only the SQLite system is below Queiroz et al.'s thresholds of complex implementation [47]. We now present our fourth finding:

Finding 4. *Feature evolution often increases implementation complexity. We observed that systems with complex implementations of their features, i.e., scattering degree ≥ 7, have more feature revisions. Interestingly, features scattered over many variation points tend to lead to more revisions over time than features with complex tangling and non-top-level branch implementation.*

Figure 3 allows analyzing hotspot features which most often change over time for all commits. Figure 3(a) and 3(b) shows the characteristics of the hotspot feature over time for the LibSSH and Irssi systems. The LOCs of the LibSSH hotspot feature (Figure 3(a)) increase over time as do the



(a) WITH_SFPT



(b) HAVE_OPENSSL

Figure 3. The hotspot features' evolution along their life cycle: (a) LibSSH feature `WITH_SFPT`; (b) Irssi feature `HAVE_OPENSSL`.

numbers of SD `#IFs` and File. In addition, we can see that its ND is ≥ 1, meaning that when a change to this feature increases its LOC, the new lines of code can be part of another feature that is inside of one of its variation points. Then, to understand this feature's evolution, it is necessary to determine within the commits at what point its characteristics changed, thereby finding out if the changes affected only the feature itself or one of its nested features. We can also note by looking at the LibSSH hotspot feature that the number of changes is not related to a high number of SD `#IFs` because even with lower SD `#IFs` it changed many times. However, these changes can be highly correlated with the ND of a feature. When a variation point that is inside an outermost variation point changes, the outermost feature indirectly changes. Thus, developers need to take care not only of scattered features, but also of features with higher nesting degrees.

The TD of the hotspot features is one, meaning that their condition expression does not contain other features. The ND is more complex for the hotspot feature of Irssi, reaching three conditional blocks inside its variation points. The NOTLBs from the Irssi system (Figure 3(b)) increased when the NONTLB decreased, which can indicate that the non-top level variation points were removed or transferred to newer top-level variation points of this feature. Yet, when looking at the LOCs of the hotspot feature of Irssi, we can see that it continuously increases, while its SD `#IFs`, SD `#NIFs`, SD File, and TD remain unchanged, meaning that feature complexity did not increase during evolution. However, its ND increased, i.e., variation points of other features were introduced inside this feature. Evolving this feature may thus have an impact on the features enclosed by its variation point. It is important to analyze all these revision characteristics to understand the impact of changes.

We found that the characteristics of the feature `HAVE_OPENSSL` increased in terms of ND in release *0.8.16*, in which the feature `HAVE_DANE` was introduced (commit[5]) inside its variation points. In another commit[6], the ND of `HAVE_OPENSSL` increased again because another feature was introduced inside its variation points (`SSL_CTRL_SET_TLSEXT_HOSTNAME`). The increase of the LOC of the feature `HAVE_OPENSSL` in this commit was due to the lines of code of the new feature introduced. This can make developers aware that the feature `HAVE_OPENSSL` evolved over time and became responsible for activating other features. Thus, we can easily notice the new features and their dependencies on the feature `HAVE_OPENSSL`.

The revision characteristics show if new revisions of features have more dependencies or interactions with other features. If the number of variation points increases over time and the ND of the feature `HAVE_OPENSSL` does not change, it can be assumed that these features have new variation points outside the feature `HAVE_OPENSSL` and that they can have interactions with other features if their TD ≥ 3. Yet, they can be responsible for activating the code of other features if their ND ≥ 1, or can be dependent on other features if their NONTLB increases instead of their NOTLB. Our last finding thus is:

Finding 5. *Dependencies and/or interactions were constantly introduced, changed, and removed along the life cycle of the features. By analyzing the revision characteristics, i.e., SD, TD, ND, and NOTLB, we observed that the complexity of feature implementations increased. We found cases in which a feature that evolved over time had many changes covering different files and affecting multiple features depending on and interacting with each other.*

[5]https://github.com/irssi/commit/d826896
[6]https://github.com/irssi/commit/28aaa65

Answering RQ3. *The evolution of feature complexity during system evolution varies in terms of changes and their characteristics. Many changes indirectly affect other features, and impacted multiple variation points, leading to the scattering of feature implementations. Usually, features with more revisions are the ones with a higher number of variation points that have high importance in the system, and thus need constant bug fixes and enhancements. Changes increasing the number of variation points usually lead to more revisions over time. Another characteristic directly affecting the variability and complexity of features is the inclusion, change, or exclusion of dependencies and interactions between features that are commonly scattered among many different files.*

6 Discussion

Based on our results we now present lessons learned and discuss the practical usefulness of our mining approach.

Developers need awareness of HCSSs evolution in space and time. Regarding RQ1, our analysis shows that in most commits, the code of the BASE feature changed. However, we still observed that many features changed across releases, e.g., in the LibSSH system, 1,143 commits were changes to features from a total of 5,022 commits, covering 45 of 48 releases of the system. It would be difficult to manually detect the changed features for such a high number of commits. Hence, it would be hard to determine the most likely affected configurations when selecting these features. As pointed out by Mühlbauer et al. [35], it is infeasible to test each and every commit and configuration when uncovering and fixing performance deficiencies and related problems caused by certain revisions. Our approach therefore suggests which commits led to specific feature changes. This can reduce the number of variants developers need to check.

Therefore, mining how frequently features are revised can make developers aware in which releases features actually changed. Our approach retrieves which features changed across all commits, which eases the selection of commits to be analyzed when investigating a specific feature. In the LibSSH system, for example, the release *libssh-0.6.0* contains the highest number (185) of commits changing features. For example, the LibSSH hotspot feature WITH_SFPT changed in 14 commits of this release. The feature WITH_SFPT from LibSSH changed 174 times during its life cycle, so knowing which commits to analyze can be extremely helpful.

Features co-evolve in single commits. Our approach determines which features evolve together, e.g., the feature WITH_SFPT co-evolved 61 times with 21 other features (not considering the feature BASE). This information reveals that the feature WITH_SERVER most frequently co-changed with WITH_SFPT. Yet, looking at the release *libssh-0.6.0* shows that the feature WITH_SERVER is the one that changed most often. It was involved in 71 of the 262 commits of this release. Knowing which features often co-evolve can also help to migrate a monolithic system to microservices [37] because according to Henry and Ridene [17], a good strategy is to migrate features that frequently change and that can bring more value when being managed separately.

VCSs lack mechanisms to deal with the evolution of HCSSs. In a particular commit[7], the features HAVE_LIBCRYPTO or HAVE_LIBGCRYPT require that the feature WITH_SSH1 is included. The information retrieved, such as ND and NOTLB across revisions shows cases when the number of top-level branches is reduced, while the number of non-top-level branches increased in a commit. This helps to investigate if new feature interactions and dependencies are still consistent with the feature model of a system. Further, the feature history helps in finding out which commit first introduced a feature and which commits are part of its further evolution, to determine the developers that originally developed or maintained a feature. For example, in the aforementioned commits, we could observe that the same developer moved the features HAVE_LIBCRYPTO and HAVE_LIBGCRYPT in and out of the feature WITH_SSH1. Future maintenance and evolution tasks involving these features could thus be assigned to this developer.

Keep track of change characteristics over time makes clear which features are affected. The information mined regarding RQ2 shows that one changed feature usually impacts other features in 20% of the commits of a system. For example, three features changed in the commit d6829d0[8] of the LibSSH system (release *libssh-0.6.0*). The mined information shows that the features HAVE_LIBCRYPTO and HAVE_LIBGCRYPT remained constant in terms of their LOC, SD #IF, SD #NIF, SD File, TD, and ND but changed in terms of NOTLB and NONTLB. When analyzing the source code of these commits, we could see that this happened because one block of code of each feature was transferred to the feature WITH_SSH1. Hence, the feature characteristics of WITH_SSH1 show that in this commit 79 new lines were added, which already existed in the system in variation points of the features HAVE_LIBCRYPTO and HAVE_LIBGCRYPT. Thus, these two features were just moved into a variation point of the feature WITH_SSH1, being dependent on the selection of WITH_SSH1. This information may help, for instance, to find a bug when any of these features does not work as expected.

Analyzing feature characteristics in every single commit can bring further information about bugs of feature interactions, test priorities, and implementation complexity. In the aforementioned example, the features HAVE_LIBCRYPTO and HAVE_LIBGCRYPT did not change in

[7] https://gitlab.com/libssh/libssh-mirror/-/commit/bf72440
[8] https://gitlab.com/libssh/libssh-mirror/-/commit/d6829d0

terms of their implementation code but were transferred to a variation point of another feature (`WITH_SSH1`). This impacted the implementation characteristics such as the ND and NOTLB of the three features. Thus, by looking at the revision characteristics of the feature `WITH_SSH1`, a developer can find out that the features were moved out again from the feature `WITH_SSH1` 130 commits later[7], when the LOC of the feature `WITH_SSH1` decreased. Therefore, it is important to know the characteristics of a feature in every single commit of a system. This makes it easier to determine in multi-feature commits, if the implementation of features changed, or just their SD, TD, ND, and NOTLB values. Thus, the information mined by our automated approach may help to find feature interaction bugs.

Yet, regarding the example about the hotspot feature of LibSSH, the revision characteristics show why the feature `WITH_SERVER` co-evolved often with the feature `WITH_SFPT`. The feature `WITH_SERVER` has an SD `#IFS` = 2, and one of its variation points is nested with an outermost variation point of the feature `WITH_SFPT`. When analyzing the life cycle evolution of the feature `WITH_SFPT`, we see that its SD `#IFS` raised above six in the 44th release (*libssh-0.9.0*). The mined information shows the features which often change, features which influence other features, features which usually were changed with more features, or features which the number of variation points increased over time. This information all together indicates that future maintenance and evolution of the `WITH_SFPT` feature may be difficult.

7 Threats to Validity

Internal validity: The selection of software systems can lead to biased results. To minimize this threat, we analyzed multiple systems from diverse domains and of different sizes, with varying numbers of releases, and commits, which have been used already in previous work [14, 16, 26, 30, 54]. Furthermore, according to Liebig et al. [26], the complexity of feature annotations is independent of the size of the software system. Regarding the features included in our analysis, we considered as features only the macros that can be set externally by the command line, which includes features other than functional ones. According to Berger et al. [7] features are also used for, e.g., testing, debugging and deployment. Further, previous empirical analyses [19, 26] have considered all macros of a system as features. Although the non-functional features considered in our analysis comprise only 1-5% of the features, we believe that analyzing their life cycle is also important for evolution tasks.

External validity: The systems we analyzed use C preprocessor directives to encode variability. It is uncertain whether our results can be generalized to other variability mechanisms. However, HCSSs are widely used to deal with evolution in space. Regarding the metrics, we tried to avoid varying definitions, and different ways of measurement to not limit the applicability of our work. Specifically, we used common metrics from previous research [11, 19, 26, 28, 46, 47].

Conclusion validity: Regarding the statistical analysis, we conducted a Shapiro-Wilk test [49] to check for the normal distribution of collected data to use the most suitable correlation coefficient.

8 Related Work

In this section, we discuss how our work differs from related work of evolution of C/C++ preprocessor-based SPLs in VCSs. We exclude previous work on feature-oriented SPL and feature model evolution [36, 38, 39] as our automated mining approach is intended to support the analysis of HCSSs at the level of annotated features in source code.

Our analysis goes beyond previous work by analyzing and computing metrics of the entire feature life cycle. Existing work computing metrics of features annotated with C/C++ preprocessor directives used more subject systems, however, the total number of commits analyzed was much smaller than in our analysis: one commit per system [26, 30, 47, 54], or up to 500 commits of one specific release [25, 40]. Our analysis on the other hand covers all commits of the systems and from all releases of the system, overall covering up to 20 years of development and 37,500 commits. Liebig et al. [26] mentioned that they did not analyze all commits as this would be an expensive analysis, however, they pointed out that mining data of a software system over time could raise interesting insights for developers. This is demonstrated in our work. In addition, our automated approach not only computes the revision characteristics of the implementation complexity of features but also the change characteristics affecting the revisions. It also considers the corner cases mentioned by Ludwig et al. [28] and the metrics are computed with a CSP solver that can be easily interpreted by humans. Therefore, our work may help to gain interesting insights for every system's version adaptation and evolution over all commits.

Passos et al. [43, 44] explored the variability evolution by capturing the addition and removal of features in the Linux kernel. They manually analyzed commits where features were introduced or removed in the variability model (KConfig files), implementation (source code with *ifdefs*), and mappings (Makefile). They did not consider changes of features. In [42], Passos et al. extended previous work [41] and improved their analysis and discussion of feature scattering in the Linux kernel by adding a survey and interview with developers. Their goal was to analyze only the scattering of features and which ones were usually scattered. Their work does not contain an approach for computing metrics by analyzing *ifdefs* conditions, as ours does with CSP solvers. For measuring the scattering of features, they identify *ifdefs* that refer to the corresponding feature. For this, their approach is based on the declaration of features in the variability model and the syntactic reference of features in code.

Dintzner et al. [9] also improve Passos et al. [43] approach: FEVER automatically extracts details on changes in variability models (KConfig files), preprocessor-based source code, and mappings (Makefiles). Other studies [15, 22, 48] also used the FEVER approach for analyzing the variability of the Linux kernel development, for characterizing and propose safe and partially safe evolution scenarios in SPL. Despite many studies analyzing the evolution of the Linux kernel, their results cannot be generalized for other SPLs, because other systems (including ours) do not use "tristate" features, the features are not defined in the Kconfig file (kernel's variability model), and they do not map features to source files in Makefiles. Furthermore, the authors of FEVER also suggested future improvements and a more precise approach that can capture the exact presence condition of assets, rather than the main features participating in that condition.

MetricHaven presented in [10] computes up to 42,000 metrics for variability-aware of SPLs. However, their focus is to compute metrics based on the programming language, and thus, it is more expensive in terms of computational resources and runtime performance. Their approach takes more than 11 hours to compute the metrics of only one version of the Linux Kernel when running on one thread. Kästner et al. [24] presented the TypeChef approach to computing detailed information of the AST of the source code besides the annotated variability, which requires much more effort than our approach. Furthermore, it uses a SAT solver, which only covers blocks of code with Boolean values. Undertaker from Sincero et al. [50] is similar to our approach and focuses on only parsing annotated blocks of code to extract variability information of product line artifacts. It also uses SAT solver and only analyzes dead blocks of code, i.e., conditional blocks that cannot be executed under any possible input configuration. Therefore, their work does not compute metrics and only analyzes the Linux kernel.

Our approach goes beyond existing ones, as it can deal with the expensive computational process of solving constraints to assign features to changes using constraint satisfaction problems. Furthermore, our automated analysis is more detailed and for each commit of every release of a system, also segmenting the *ifdefs* analysis by features. This allows distinguishing between external literals (macros never defined within the source code and likely the features of the system) and internal literals (macros defined by `#define` directives) in relation to previous work from Hunsen et al. [19]. They considered all literals/macros of *ifdefs* to capture the entire system's variability, while our work analyzes other systems than the Linux kernel. Our approach also automatically computes an analysis of the number of lines that affected variability. In addition, our approach computes the characteristics of the feature complexity and each change in every commit of the system.

Therefore, our approach focuses on a higher level of abstraction to compute metrics for features parsing only preprocessor annotated blocks of code with a CSP solver involving Boolean as well as numeric values in the range of integers or double in arithmetic operations and comparisons. In this way, our approach is faster and computationally less expensive for analyzing all versions of all commits of a system in computing metrics for change and feature characteristics. We thus believe our approach can also be used to compute more feature metrics and can be used to several further research opportunities, such as combining with heuristics to analyze commit messages aiming to result in more reliable metrics and more accurate predictions of defects than previous work from Strüder et al. [52].

9 Conclusions and Future Work

This work presents an analysis of the features' life cycle by investigating the frequency of changes, their characteristics, and the impact of changes on the complexity of feature implementations. We introduced an automated approach for mining characteristics of changes and the implementation complexity of feature revisions to track the feature evolution. The results show that a specific feature from one commit of one release can be very different from another commit of another release, not only in terms of size, SD, and TD but also in terms of its content and purpose. Our analysis and findings also show the complexity of evolving software systems in space and time combining HCSSs with VCSs and stress the need for better support at the level of features [1, 6, 31, 34]. In future work, we will investigate granularity-levels of changes of every commit and connect the information on feature evolution with bug reports and bug fixes [52]. Further, we aim to provide a tool for data visualization to make the result analysis understandable for developers and engineers.

Acknowledgments

This research was funded by the LIT Secure and Correct Systems Lab; the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; FAPERJ PDR-10 program, grant no. 202073/2020. The support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

References

[1] Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. 2019. Towards a conceptual model for unifying variability in space and time. In *23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019.* ACM, New York, NY, USA, 67:1–67:5. https://doi.org/10.1145/3307630.3342412

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-oriented software product lines.* Springer, New York, NY, USA.

[3] Google Code Archive. 2021. *Java-diff-utils.* Google Code Archive. https://java-diff-utils.github.io/java-diff-utils/

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. 2015. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15).* IEEE Press, San Francisco, CA, USA, 134–144. https://doi.org/10.1109/ICSE.2015.35

[5] David Benavides, Sergio Segura, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. 2005. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *Generative and Transformational Techniques in Software Engineering, International Summer School, (GTTSE '05)* (Braga, Portugal) *(Lecture Notes in Computer Science, Vol. 4143).* Springer, Berlin, Heidelberg, 399–408. https://doi.org/10.1007/11877028_16

[6] Thorsten Berger, Marsha Chechik, Timo Kehrer, and Manuel Wimmer. 2019. Software Evolution in Time and Space: Unifying Version and Variability Management (Dagstuhl Seminar 19191). *Dagstuhl Reports* 9, 5 (2019), 1–30. https://doi.org/10.4230/DagRep.9.5.1

[7] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *19th International Systems and Software Product Line Conference (SPLC '15).* ACM, Nashville, USA, 1–10. https://doi.org/10.1145/2791060.2791108

[8] Git Software Freedom Conservancy. 2020. *Documentation.* Git Software Freedom Conservancy. Retrieved Apr. 22, 2021 from https://git-scm.com/doc

[9] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2017. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering* 23, 2 (Nov. 2017), 905–952. https://doi.org/10.1007/s10664-017-9557-6

[10] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 2020. Fast Static Analyses of Software Product Lines: An Example with More than 42,000 Metrics. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems* (Magdeburg, Germany) *(VAMOS '20).* ACM, New York, NY, USA, Article 8, 9 pages. https://doi.org/10.1145/3377024.3377031

[11] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. 2019. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology* 106 (2019), 1–30. https://doi.org/10.1016/j.infsof.2018.08.015

[12] James D. Evans (Ed.). 1996. *Straightforward statistics for the behavioral sciences.* Thomson Brooks, Vol. 1. Cole Publishing Co, California, USA.

[13] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. 2016. Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel. In *20th International Systems and Software Product Line Conference* (Beijing, China) *(SPLC '16).* ACM, New York, NY, USA, 65–73. https://doi.org/10.1145/2934466.2934467

[14] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. 2016. Validation of Constraints Among Configuration Parameters Using Search-Based Combinatorial Interaction Testing. In *Search Based Software Engineering* (Raleigh, NC, USA), Federica Sarro and Kalyanmoy Deb (Eds.). Springer International Publishing, Cham, 49–63.

[15] Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. 2019. Characterizing Safe and Partially Safe Evolution Scenarios in Product Lines: An Empirical Study. In *13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS '19).* ACM, New York, NY, USA, Article 15, 9 pages. https://doi.org/10.1145/3302333.3302346

[16] H. Ha and H. Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *35th International Conference on Software Maintenance and Evolution (ICSME '19).* IEEE Press, San Francisco, CA, USA, 470–480. https://doi.org/10.1109/ICSME.2019.00080

[17] Alexis Henry and Youssef Ridene. 2020. *Migrating to Microservices.* Springer International Publishing, Cham, 45–72. https://doi.org/10.1007/978-3-030-31646-4_3

[18] Kim Herzig and Andreas Zeller. 2013. The Impact of Tangled Code Changes. In *10th Working Conference on Mining Software Repositories (MSR '13).* IEEE Press, San Francisco, CA, USA, 121–130. https://doi.org/10.1007/s10664-015-9376-6

[19] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Lessenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (April 2016), 449–482. https://doi.org/10.1007/s10664-015-9360-1

[20] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. 2014. PrefFinder: Getting the Right Preference in Configurable Software Systems. In *29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14).* ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/2642937.2643009

[21] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. 2018. KernelHaven: An Open Infrastructure for Product Line Analysis. In *22nd International Systems and Software Product Line Conference - Volume 2* (Gothenburg, Sweden) *(SPLC '18).* ACM, New York, NY, USA, 5–10. https://doi.org/10.1145/3236405.3236410

[22] Christian Kröher, Lea Gerling, and Klaus Schmid. 2018. Identifying the Intensity of Variability Changes in Software Product Line Evolution. In *22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18).* ACM, New York, NY, USA, 54–64. https://doi.org/10.1145/3233027.3233032

[23] Jacob Krüger. 2019. Tackling Knowledge Needs during Software Evolution. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE '19).* ACM, New York, NY, USA, 1244–1246. https://doi.org/10.1145/3338906.3342505

[24] Christian Kästner. 2013. *TypeChef.* Christian Kästner. Retrieved Apr. 22, 2021 from https://ckaestne.github.io/TypeChef/

[25] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2018. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering* 44, 2 (2018), 182–201. https://doi.org/10.1109/TSE.2017.2664824

[26] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE'10).* ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1806799.1806819

[27] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. 2021. Concepts of variation control systems. *J. Syst. Softw.* 171 (2021), 110796. https://doi.org/10.1016/j.jss.2020.110796

[28] Kai Ludwig, Jacob Krüger, and Thomas Leich. 2019. Covert and Phantom Features in Annotations: Do They Impact Variability Analysis?. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19).* ACM, New York, NY, USA, 218–230. https://doi.org/10.1145/3336294.3336296

[29] Flávio Medeiros, Christian Kaestner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP '15, Vol. 37).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Prague, CZE, 495–518. https://doi.org/10.4230/DARTS.1.1.7

[30] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. 2018. Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 453–469. https://doi.org/10.1109/TSE.2017.2688333

[31] Gabriela Karoline Michelon. 2020. Evolving System Families in Space and Time. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020*, Rafael Capilla, Philippe Collet, Paul Gazzillo, Jacob Krüger, Roberto Erick Lopez-Herrejon, Sarah Nadi, Gilles Perrouin, Iris Reinhartz-Berger, Julia Rubin, and Ina Schaefer (Eds.). ACM, New York, NY, USA, 104–111. https://doi.org/10.1145/3382026.3431252

[32] Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume B*. ACM, New York, NY, USA, 74–78. https://doi.org/10.1145/3382026.3425776

[33] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. Managing systems evolving in space and time: four challenges for maintenance, evolution and composition of variants. In *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021*, Mohammad Mousavi and Pierre-Yves Schobbens (Eds.). ACM, New York, NY, USA, 75–80. https://doi.org/10.1145/3461001.3461660

[34] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating feature revisions in software systems evolving in space and time. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, New York, NY, USA, 14:1–14:11. https://doi.org/10.1145/3382025.3414954

[35] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. 2020. Identifying Software Performance Changes Across Variants and Versions. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3324884.3416573

[36] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. 2011. Investigating the Safe Evolution of Software Product Lines. In *10th ACM International Conference on Generative Programming and Component Engineering* (Portland, Oregon, USA) *(GPCE '11)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/2047862.2047869

[37] Sam Newman. 2015. *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc., Farnham, UK.

[38] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. 2018. Anomaly Analyses for Feature-Model Evolution. In *17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Boston, MA, USA) *(GPCE '18)*. ACM, New York, NY, USA, 188–201. https://doi.org/10.1145/3278122.3278123

[39] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-Oriented Software Evolution. In *Seventh International Workshop on Variability Modelling of Software-Intensive Systems* (Pisa, Italy) *(VaMoS '13)*. ACM, New York, NY, USA, Article 17, 8 pages. https://doi.org/10.1145/2430502.2430526

[40] Leonardo Passos, Jianmei Guo, Leopoldo Teixeira, Krzysztof Czarnecki, Andrzej Wąsowski, and Paulo Borba. 2013. Coevolution of Variability Models and Related Artifacts: A Case Study from the Linux Kernel. In *17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC '13)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/2491627.2491628

[41] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *14th International Conference on Modularity* (Fort Collins, CO, USA) *(MODULARITY 2015)*. ACM, New York, NY, USA, 81–92. https://doi.org/10.1145/2724525.2724575

[42] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. 2021. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* 47, 1 (2021), 146–164. https://doi.org/10.1109/TSE.2018.2884911

[43] Leonardo Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wąsowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. 2016. Coevolution of Variability Models and Related Software Artifacts. *Empirical Softw. Engg.* 21, 4 (Aug. 2016), 1744–1793. https://doi.org/10.1007/s10664-015-9364-x

[44] Leonardo Teixeira Passos. 2016. *Towards a Better Understanding of Variability Evolution.* Ph.D. Dissertation. University of Waterloo, Ontario, Canada. http://hdl.handle.net/10012/10529

[45] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. 2016. *Choco Solver Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. http://www.choco-solver.org

[46] Rodrigo Queiroz, Leonardo Passos, Marco Tulio Valente, Sven Apel, and Krzysztof Czarnecki. 2014. Does Feature Scattering Follow Power-Law Distributions? An Investigation of Five Pre-Processor-Based Systems. In *6th International Workshop on Feature-Oriented Software Development* (Västerås, Sweden) *(FOSD '14)*. ACM, New York, NY, USA, 23–29. https://doi.org/10.1145/2660190.2662114

[47] Rodrigo Queiroz, Leonardo Passos, Tulio Marco Valente, Claus Hunsen, Sven Apel, and Krzysztof Czarnecki. 2017. The shape of feature code: an analysis of twenty C-preprocessor-based systems. *Software and Systems Modeling (SoSyM)* 16 (2017), 77–96. https://doi.org/10.1007/s10270-015-0483-z

[48] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. 2019. Partially safe evolution of software product lines. *Journal of Systems and Software* 155 (2019), 17–42. https://doi.org/10.1016/j.jss.2019.04.051

[49] S. S. Shapiro and M. B. Wilk. 1965. An analysis of variance test for normality (complete samples). *Biometrika* 52, 3-4 (dec 1965), 591–611. https://doi.org/10.1093/biomet/52.3-4.591

[50] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *9th International Conference on Generative Programming and Component Engineering* (Eindhoven, The Netherlands) *(GPCE '10)*. ACM, New York, NY, USA, 33–42. https://doi.org/10.1145/1868294.1868300

[51] C. Spearman. 1904. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology* 15, 1 (1904), 72–101.

[52] Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. 2020. Feature-Oriented Defect Prediction. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. ACM, New York, NY, USA, Article 21, 12 pages. https://doi.org/10.1145/3382025.3414960

[53] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *23rd International Systems and Software Product Line Conference* (Paris, France) *(SPLC '19)*. ACM, New York, NY, USA, 57–64. https://doi.org/10.1145/3307630.3342414

[54] Tassio Vale and Eduardo Santana Almeida. 2019. Experimenting with Information Retrieval Methods in the Recovery of Feature-Code SPL Traces. *Empirical Software Engineering* 24, 3 (June 2019), 1328–1368. https://doi.org/10.1007/s10664-018-9652-3