JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Gabriela Karoline Michelon, MSc

Submitted at
Institute for Software Systems Engineering

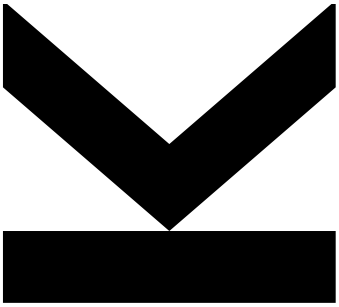
First Supervisor
Univ.-Prof. Dr. Alexander Egyed

Second Supervisor
a. Univ.-Prof. Dr. Paul Grünbacher

Co-Supervisor
Dr. Wesley Klewerton Guez Assunção

July 2022

Evolving System Families in Space and Time



Doctoral Thesis
to obtain the academic degree of
Doktorin der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

Families of software systems evolve in space, by introducing and removing features, to be available for different operating systems, and platforms, and offer functionalities to satisfy different users' requirements. In addition to evolution in space, families of software systems also evolve over time, when features are revised due to bug fixes and enhancements to keep the systems operating properly. For dealing with evolution in space and time of families of software systems, preprocessor-based systems managed in version control systems (VCSs) are the most used mechanism among existing ones. Preprocessor is one of the most widely used variability mechanisms for implementing open-source as well as industrial families of software systems within a software product line (SPL). One of the advantages of preprocessor-based SPLs is the easy creation of different product variants by selecting or deselecting different functionalities, i.e., features annotated in variation points in the source code with preprocessor directives. Moreover, preprocessor directives are pieces of text, and text can be tracked by VCSs, which makes their integration convenient.

A negative side effect of the development of a system with preprocessor directives in a VCS is that for understanding and managing features developers have to perform manual analysis of preprocessor directives within multiple files of the whole platform, which can be a complex and time-consuming task. This is because preprocessor directives obfuscate the code, which makes it harder to understand, comprehend and maintain. Moreover, VCSs do not offer proper support to retrieve and analyze the changes at the level of features annotated in the source code. In addition, often changes of a single commit touch and affect multiple features, which can be tangled and scattered over different files and variation points. Thus, determining which features were actually revised/changed is an infeasible and error-prone task without additional tool support. Further, reusing and propagating a feature implementation across different versions can be very challenging and expensive.

Even though combining other variability mechanisms, e.g., assuming certain programming paradigms as feature-oriented or aspect-oriented programming, with VCS for managing evolution in space and time, unfortunately, does not allow to comprehensively and uniformly handle variants (concurrent versions) and their revisions (sequential versions). Although variation control systems (VarCSs) have been proposed to realize a common platform based on features with revision management, they are not mature enough, such as preprocessor-based SPLs and VCS. Some of the limitations of VarCSs are that they have their own proprietary repositories, unfamiliar operations, lack of collaboration support among developers, and high complexity and demand using logical expressions to handle variants and their revisions. That is why preprocessor-based SPLs managed in VCS are still the most popular mechanisms used in practice to deal with evolution in space and time. As preprocessor-based SPLs managed in VCS are widely established and well-integrated into development processes, it is necessary to bring up new solutions addressing their existing limitations for managing variants and their versions. One of the limitations of preprocessor-based SPLs managed in VCS is that VCSs do not offer support to keep track of changes and propagation at the feature level as a feature can have different implementations in different versions of a system.

Therefore, the goal of this thesis is to overcome such limitations and challenges to deal

with evolution of families of software systems in space and time. In order to propose a novel solution, we thus first conducted an empirical study of the feature life cycle in preprocessor-based systems managed in VCSs to understand how features evolve and learn the challenges and limitations of these mechanisms. This empirical study relies on an automated approach we presented for mining repositories of preprocessor-based SPLs managed in VCSs. By mining the feature life cycle, it was possible to observe that features are often introduced, removed, and revised over time with complex implementation. This finding showed the need for better support at the level of features. Taking into account the finding from the empirical study, we posed four challenges of systems evolving in space and time. The challenges concern feature location, and feature revision location to locate different implementations of a feature at different points in time, as well as composition of new configurations based on features and feature revisions, i.e., reuse of different implementations of features in different versions of a system.

Based on these challenges, we defined and implemented a novel support that tracks feature revisions. Thus, we presented an automated feature revision location approach for tracing features' implementation to their multiple points in time. The approach is implemented in a publicly available tool and offers support for composing new products based on new combinations of features and their revisions. To support the challenge of reusing features evolving in space and time in preprocessor-based systems, this thesis also presents a novel approach implemented in a non-intrusive tool for automatic analysis and propagation of feature implementation in VCSs.

Overall, this thesis presents the findings and challenges posed concerning system evolution in space and time together with a benchmark to enable future work and comparisons of approaches addressing the challenges. The goal of presenting these challenges is to motivate researchers and tool developers to optimize and address current limitations of existing techniques and to develop more efficient mechanisms for managing systems evolving in space and time. Based on findings and challenges of systems evolving in space and time, this thesis also contributes with an automated approach for mining, analyzing and reusing feature revisions; and for feature revision location and composition of new products. Both approaches are realized in publicly available tools.

Kurzfassung

Softwaresystem-Familien entwickeln sich im Raum wenn Eigenschaften hinzugefügt oder entfernt werden um für unterschiedliche Betriebssysteme und Plattformen verfügbar zu sein. Sie stellen des Weiteren zusätzliche Funktionalitäten für unterschiedlichste Benutzer zur Verfügung. Zusätzlich zur Entwicklung im Raum findet auch eine Entwicklung über die Zeit statt wenn Eigenschaften überarbeitet werden um Fehlerbehebungen und Erweiterung eines Systems weiterhin garantieren zu können. Um die Entwicklung von Softwaresystem-Familien in Raum und Zeit gewährleisten zu können werden primär präprozessorbasierte Systeme in Verbindung mit Versionskontrollsystemen (Version Control Systems, VCSs) verwendet. Der Präprozessor ist einer der am meisten verwendete Mechanismen um Open-Sourceprojekte und Softwaresystem-Familien aus Industrie in Verbindung mit Softwareproduktlinien (SPL) zu implementieren. Ein Vorteil von präprozessorbasierten SPL ist die einfache Erstellung von unterschiedlichen Produktvarianten durch die Auswahl unterschiedlicher Funktionalitäten, d.h. Eigenschaften werden im Sourcecode an Variationspunkten mit Präprozessordirektiven annotiert. Des Weiteren ist es sehr einfach Präprozessordirektive zu integrieren, da es sich um reinen Text handelt und dieser von VCSs einfach verarbeitet werden kann.

Eine negative Auswirkung bei der Entwicklung eines System mit Präprozessordirektiven in einem VCS ist dass Entwickler Präprozessordirektive in mehreren Dateien des ganzen Systems manuell analysieren müssen. Dies kann eine sehr zeitintensive und komplexe Tätigkeit sein. Der Grund dafür ist, dass Präprozessordirektive den Sourcecode schwerer verständlich machen. Weiters bieten VCSs keine Unterstützung um Veränderungen der Eigenschaften auf Sourcecodeebene abzufragen oder zu analysieren. Zusätzlich können Änderungen eines einzelnen Commits mehrer Eigenschaften beeinflussen, die wiederum über mehrer Dateien und Variationspunkte verteilt sein können. Daher ist eine Bestimmung veränderter Eigenschaften eine undurchführbare und fehleranfällige Aufgabe ohne die Unterstützung von zusätzlichen Softwarewerzeugen. Weiter kann die Wiederverwendung und Propagierung einer Eigenschaftsimplementierung über verschiedene Versionen sehr herausfordernd und teuer sein.

Selbst das Kombinieren anderer Variabilitätsmechanismen, wie zum Beispiel Programmierparadigmen als eigenschafts- oder aspektorientierte Programmierung in Verbindung mit VCSs zur Verwaltung in Zeit und Raum anzunehmen, erlaubt leider nicht eine umfassende und gleichmäßige Verwaltung von Varianten (gleichzeitige Versionen) und deren Revisionen (sequentielle Versionen). Obwohl Variationskontrollsysteme (Variation Control System, VarCS) als gemeinsame Plattform, basierend auf Eigenschaften mit Revisionsverwaltung, vorgeschlagen wurden, sind diese leider nicht fortgeschritten genug im Gegensatz zu präprozessorbasierte SPLs mit VCSs. Einige der Einschränkungen von VarCSs sind zum Beispiel dass diese auf proprietäre Repositories basieren, unbekannte Operationen ausführen, einen Mangel an Kollaborationsunterstützung haben und eine hohe Komplexität beim Benutzen von Logikausdrücken besitzen um Varianten und Revisionen zu verwalten. Aus diesen Gründen sind präprozessorbasierte SPLs verwaltet in VCSs die populärsten Mechanismen in der Praxis um Entwicklungen in Raum und Zeit abbilden zu können. Da präprozessorbasierte SPLs verwaltet in VCSs weitgehend eingeführt und in Entwicklungsprozessen integriert wurden ist es nötig, neue Lösung vorzuschlagen um die Limitierungen

einer Verwaltung von Varianten und deren Revisionen zu beheben. Eine Limitierung von präprozessorbasierten SPLs verwaltet in VCSs ist, dass VCSs keine Unterstützung zur Verfolgung von Veränderungen und Verbreitungen auf der Eigenschaftsebene anbieten, da eine Eigenschaft mehrer Implementierungen in verschiedenen Versionen eines System besitzen kann.

Das Ziel dieser Arbeit ist daher diese Limitierungen und Herausforderungen zu bewältigen um die Veränderung von Softwaresystem-Familien in Raum und Zeit zu bewerkstelligen. Um eine neuartige Lösung vorschlagen zu können, haben wir zuerst eine empirische Studie durchgeführt. Diese untersuchte den Lebenszyklus von Eigenschaften in präprozessorbasierten Systemen verwaltet in VCSs um zu verstehen wie Eigenschaften sich verändern, und ebenso zu lernen welche Limitierungen und Herausforderungen solche Mechanismen mit sich bringen. Diese empirische Studie basiert auf einem automatisierten Vorgehen, welches wir zur Gewinnung von Repositories von präprozessorbasierten SPLs verwaltet in VCSs präsentiert haben. Durch die Gewinnung von Eigenschaftslebenszyklen war es möglich Eigenschaften über die Zeit zu beobachten, die häufig in komplexen Implementierungen hinzugefügt, entfernt und überarbeitet wurden. Diese Einsicht zeigt die Notwendigkeit einer besseren Unterstützung auf der Eigenschaftsebene. Unter Berücksichtigung der Studie haben wir vier Herausforderungen beim Entwickeln von Systemen in Raum und Zeit identifiziert. Diese Herausforderungen beinhalten, Eigenschaftsposition und Eigenschfaftsrevisionsposition um die unterschiedlichen Implementierungen einer Eigenschaft zu verschiedenen Zeitpunkten zu lokalisieren. Weiters die Komposition neuer Konfigurationen basierend auf Eigenschaften und Eigenschaftsrevisionen, daher die Wiederverwendung unterschiedlicher Implementierungen von Eigenschaften in unterschiedlichen Versionen eines Systems.

Basierend auf diesen Herausforderungen haben wir eine neue Unterstützung definiert und implementiert die es ermöglicht Eigenschaftsrevisionen nachzuverfolgen. Daher präsentieren wir in dieser Arbeit einen automatisierten Eigenschftsrevisionen-lokalisierungsansatz zur Nachverfolgung von Eigenschtsimplementationen zu deren Zeitpunkten. Dieser Ansatz wurde mittels eines öffentlich verfügbaren Werkzeugs implementiert und bietet die Möglichkeit neue Produkte zu erstellen basierend auf neuen Kombinationen von Eigenschaften und deren Revisionen. Um eine Unterstützung in präprozessorbasierten Systemen für das Wiederverwenden von Eigenschaften welche sich in Raum und Zeit ändern, präsentiert diese Arbeit außerdem einen neuen Ansatz welcher mittels eines nicht intrusiven Werkzeugs zur automatisierten Analyse und Propagierung von Eigenschfatsimplemntierungen in VCSs.

Im Allgemeinen präsentiert diese Arbeit die Erkenntnisse und Herausforderungen bezüglich Entwicklungen in Raum und Zeit in Verbindung mit einer Bewertungsmethode für zukünftige Arbeiten die versuchen diese Herausforderungen zu adressieren. Das Ziel der Präsentation dieser Herausforderung ist auch um Forscher und Werkzeugentwickler zu ermutigen aktuelle Limitierungen existierender Techniken zu adressieren und auch effizientere Mechanismen zu entwickeln um Systementwicklungen in Raum und Zeit zu verwalten. Basierend auf den Erkenntnissen und Herausforderungen von Systementwicklungen in Raum und Zeit trägt diese Arbeit ebenfalls mit einem automatisierten Ansatz zur Analyse, Wiederverwendung und Förderung von Eigenschaftsrevisionen bei, ebenso für Eigenschaftsrevisionspositionen und Produktkompositionen. Beide Ansätze wurden in öffentlich verfügbaren Werkzeugen realisiert.

Eidesstattliche Erklärung

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

Linz, July, 2022

Place, Date

Signature

Acknowledgements

I would like to thank my supervisor Prof. Dr. Alexander Egyed for giving me the opportunity to work in his institute and for supporting me in conducting scientific research. I would also like to thank Prof. Dr. Paul Grünbacher for being my second supervisor, helping me in scientific writing, and being always available for discussions. I want to give special thanks to Dr. Wesley Klewerton Assunção, who co-supervised my work and was always available and patient to advise me. My advisors certainly played an important role in my personal and professional development.

Besides my advisors, I want to thank the professors Dr. Thorsten Berger, Dr. René Mayrhofer, and Dr. Josef Küng for participating in the senate of my PhD defense. I would also like to thank each one of my wonderful co-authors for the collaborations and productive work we did together over the last few years. Finally, I want to express my deepest gratitude to my family and my husband for believing in me, for all their love and support, thank you so much!

Contents

Part I

Prologue

Chapter 1

Introduction

Throughout the life cycle of families of software systems, it is unavoidable to customize different system variants due to the diversity of functional and non-functional requirements and the need of supporting multiple platforms and operating systems across different environments [119]. In parallel, software systems have to evolve due to, e.g., scalability issues, bug fixes, or features enhancement to continuously satisfy users [121]. Therefore, software systems evolve in two dimensions: (i) *space*, when introducing or removing features of a system variant, and (ii) *time*, when existing features of a system are modified/revised over time [185].

Nowadays, software engineers and developers combine different mechanisms and tools when evolving families of software systems in space and time [8, 119]. The preprocessor variability mechanism is widely used for dealing with evolution in space in both open-source and industrial systems [74]. Among the advantages of preprocessor-based systems is the easy customization of system variants relying on features annotated in variation points guarded by `#ifdef` preprocessor directives [75]. Preprocessor-based systems enable alternating implementations and thus support different platforms and operating systems [119]. However, to deal with evolution over time, it is necessary to combine additional tool support, such as version control systems (VCSs), which have been used to keep track of the evolution history of preprocessor-based systems [26]. Challenges of combining such tools motivate the work of this thesis and are presented below.

1.1 Problem Statement and Research Goal

Preprocessor directives combined with VCSs make system understanding and maintainability harder for developers and engineers, with limited separation of concerns, error proneness, difficult comprehension, and code obfuscation [49, 118, 119, 127]. Yet, VCSs can only track changes of text and recover information either per file or for the whole system. Tracking and propagating changes at the feature level would be desirable, but VCSs are not designed to deal with preprocessor-based systems [21]. Providing such kind of support, requires to also improve the understanding on how features evolve over time, and to know how the implementation of a revision of a feature changes from one commit and from one release to another. Thus, this thesis aimed to address the following problem:

Research Problem: *How to improve the management of features in families of software systems evolving in space and time?*

The feature-level granularity can help to solve problems of, e.g., recovering feature information in single commits involving multiple files and variation points annotated with `#ifdefs` [127]. Further, recovering feature revisions can help to propagate different versions of a feature among different releases or branches of systems managed in VCS [133]. For

instance, if a particular change is required for a new hardware device acquired by a customer, reusing this new revision of the feature in other products can rapidly become cumbersome. Thus, in addition to manually analyzing which commits resulted in which feature revisions and interactions, developers may need to manually copy, paste and edit the source code to propagate changes to features in different releases [127]. Mapping feature revisions to artifacts can help to comprehend, maintain, evolve and propagate features [133]. However, the current development of preprocessor-based systems managed in VCSs does not support for mapping, visualizing, and propagating feature revisions. Mapping artifacts to feature revisions can be challenging and complex because multiple features interact with each other through preprocessor directives, including the absence and presence of features. Further, some macros defined can have arithmetic expressions besides Boolean operations that determine if a hunk of code is kept or not. In addition, there are scalability issues to be considered when mapping artifacts to feature revisions because a system can contain multiple features and revisions that increase constantly over the system life cycle [134]. For instance, the Linux kernel is maintained for more than 30 years, containing its evolution through 1,074,491 Git commits involving 15,000+ features (configuration options), and its features can have up to 3 values: “yes", “no", or “module" [153]. In addition, visualizing and propagating a feature revision implementation in space and time among different releases requires a laborious analysis because different releases contain different files, features, and interactions. Therefore, this thesis goal is defined by taking into account these limitations for properly dealing with the evolution of systems in space and time at the feature level as follows:

Research Goal: *Understanding how families of software systems evolve in space and time and providing support for the analysis, maintenance, and reuse of feature revisions.*

Based on this thesis research problem and goal, we formulate this thesis research questions (RQs) as described in the next section.

1.1.1 Research Questions

RQ1. *How often are features revised through their life cycle?* We investigate how often features are revised, introduced and removed along with evolution of preprocessor-based systems. We computed the number of newly introduced, removed, and changed features within each release of real-world preprocessor-based systems managed in VCSs.

RQ2. *What is the scope of feature revisions?* For all deltas, i.e., patches of code changed in source code files between one commit to another, we analyze how many variation points (`#ifdefs`) and lines of source code are affected for a feature. Further, we analyze how the changes are tangled with features and files. This presents what are the characteristics of the changes in features implementation over time.

RQ3. *How do feature revisions affect the complexity of feature implementation?* In this RQ, we aim to understand if the evolution of features over time gets more complex, for example, in terms of number of variation points (`#ifdefs`) and feature interactions.

RQ4. *How to improve the support to aid analysis, maintenance, and reuse of feature revisions?* The goal of this RQ, based on the knowledge of previous RQs, is to define an approach to map a feature to its different implementations over time and support the reuse of feature revisions.

Research questions 1-3 provide answers to help understanding how families of software systems evolve in space and time at the feature level, in terms of when and how features

are introduced, removed, and revised during each commit of preprocessor-based systems managed in VCSs. The research question 4 provides answer on how to support analysis, maintenance, and reuse of feature revisions in families of software systems evolving in space and time.

1.2 Contributions

This thesis contributes to expanding the knowledge on software maintenance and evolution by (i) conducting empirical studies on how preprocessor-based systems have been managed in VCS; (ii) proposing approaches to fill the gap existing in the literature for feature analysis and propagation in space and time; and (iii) realizing such approaches in a non-intrusive toolchain with a focus on directly supporting developers. More specifically, the concrete outputs and contributions of this thesis are:

C1. *Comprehension of how families of software systems evolve in space and time.* This contribution is based on empirical analyses of the feature life cycle of real-world families of software systems evolving in space and time. The empirical analyses were performed by using our developed approach that supports mining feature revisions, their implementation and change characteristics.

C2. *Support for mining, reusing, and propagating feature revisions in preprocessor-based systems managed in VCSs.* This contribution addresses families of software systems implemented specifically with preprocessor directives in VCSs, mining feature revisions from preprocessor directives and commits.

C3. *Support for locating feature revisions in families of software systems evolving in space and time and compose new products with feature revisions.* This allows to locate feature revisions in any artifact type from a set of products/variants of a family of software systems. The support is implemented with an internal data structure that can be extended with plug-ins/adapters as long as different implementation languages and kinds of artifacts are needed. This support is also designed to use the artifacts mapped to feature revisions for composing new products with feature revisions. Thus, it is a support to understand, maintain, and reuse families of software systems evolving in space and time that is not restricted to preprocessor-based systems.

C4. *Four challenges posed and a benchmark for evaluating and comparing feature (revision) location techniques with an established set of metrics and dataset.* The first and second challenge is related to evolution in space, specifically, we present a benchmark to motivate researchers and tool developers to optimize and address the limitations of existing feature location techniques and support for composing new products. The third and fourth challenge is related to evolution in space and time, where we present a benchmark to evaluate feature location techniques, i.e., feature location at multiple points in time, and support for composing new products with feature revisions.

C5. *Two non-intrusive and publicly available tools.* Our implemented tools are available to support developers to understand, maintain and reuse software systems evolving in space and time as well as for follow-up studies and future collaborative improvements of our tools.

During this PhD thesis the author was involved in several studies, which resulted in multiple papers published, already accepted for publication, or under revision/submission. Table 1.1 shows all these papers according to which dimensions of evolution each one addresses. Below, the first five papers listed are part of the core contributions of this

thesis, addressing evolution in both dimensions: space and time. The complete papers that compose the core of this thesis (Papers A-E) are available in Part II of this document.

Paper A [127] *The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time* with Wesley Klewerton Guez Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. In the Proceeding of the 20th International Conference on Generative Programming: Concepts and Experiences (GPCE 2021), pages 1–14, Chicago, IL, USA, 2021

In Paper A, we present a novel approach implemented in a tool support for mining features in preprocessor-based systems managed in VCSs. The approach mines features, interactions and their implementation characteristics, as well as the complexity of changes performed over time. We applied our approach to the entire development life cycle of four preprocessor-based systems managed in VCSs (13 to 20 years and 37,500 commits) to conduct an empirical study on how features have been evolved over the system life cycle. The information mined shows that features often change with substantial modifications and feature interactions in their implementation. The findings of our empirical analyses (see Paper A) stress the need for better support at the feature level to maintain and reuse families of software systems evolving in space and time. Regarding the study presented in Paper A, the basic ideas for the approach and evaluation were a result of discussions of all the authors. G. Michelon implemented the automated mining approach for computing metrics necessary to perform our empirical analysis. The evaluation was performed by G. Michelon. The co-authors further contributed in writing the paper, implementing parts of the approach, further discussions and proofreading the paper.

Paper B [126] *Propagating Feature Revisions in Preprocessor-based Software Product Lines* with Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. Submitted to the Conference, 2022 (omitted name for double-blind review process).

In Paper B, we present a novel approach for feature revision change analysis and propagation of preprocessor-based systems managed in VCSs. This approach is implemented in a non-intrusive tool, i.e., it enables automated feature revision change analysis and propagation without changing any aspect of the traditional development of preprocessor-

Table 1.1: Contributions for evolution in *Space* and/or *Time*.

Brief description	**Core of this thesis**	***Space***	***Time***
Empirical study of features life cycle [127]	Paper A	yes	yes
Feature revision change analysis and propagation [126]	Paper B	yes	yes
Feature revision location [134]	Paper C	yes	yes
ECSEST approach [133] extension from [134]	Paper D	yes	yes
Challenges and benchmark [131]	Paper E	yes	yes
Static feature location [129]	Additional	yes	no
Doctoral symposium [125]	Additional	yes	yes
Mining feature revisions [132]	Additional	yes	yes
Feature location for test reuse [54]	Additional	yes	no
Hybrid feature location [128]	Additional	yes	no
Spectrum-based feature location [124]	Additional	yes	no
Spectrum-based feature location [130] extension from [124]	Additional	yes	no
Journal-first [55] of [54]	Additional	yes	no
Workshop proposal VariVolution 2021 [58]	Additional	yes	yes
Workshop proposal VariVolution 2022 [61]	Additional	yes	yes
Systematic software reuse [101]	Additional	yes	no
Feature-based test traceability tool [122]	Additional	yes	yes
Code Smells detection in cloned variants [113]	Additional	yes	yes

based systems managed in VCSs. The feature revision change analysis retrieves relevant information such as which files and lines of code and feature interactions are affected in a release of a system when propagating changes to the implementation of a feature. The feature revision propagation is an automated approach that propagates the implementation of a feature from one point in time to any arbitrary one. This approach for feature revision change analysis and propagation advances in the reuse of evolution in space and time by retrieving and propagating feature revisions implementation between different releases of preprocessor-based systems managed in VCS. In this paper, we show that without our tooling support, a great effort and time are necessary when manually performing feature revision change analysis and propagation. The approach was implemented by G. Michelon. All the evaluation was performed by G. Michelon. The co-authors further contributed in writing the paper, further discussions, and proofreading the paper.

Paper C [134] *Locating Feature Revisions in Software Systems Evolving in Space and Time* with David Obermann, Lukas Linsbauer, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. In the Proceeding of the 24th International Systems and Software Product Line Conference (SPLC 2020), pages 1–10, Montreal, QC, Canada, 2021

In Paper C, we present a novel and automated feature revision location technique. To the best of our knowledge, this is the first feature location approach able to locate features in space and time. The approach is able to map features at multiple points in time, i.e., feature revisions to their implementation. This approach for feature revision location allows practitioners to reason about features with different implementation at different points in time. The mapping of a feature and its different implementations at different points in time can ease the maintenance and reuse of families of software systems evolving in space and time. The basic ideas for the approach and evaluation of Paper A were a result of discussions of all the authors. G. Michelon, L. Linsbauer and D. Obermann were responsible for implementing the proposed approach and evaluation. The co-authors further contributed to writing the paper, further discussions, and proofreading the paper.

Paper D [133] *Evolving Software System Families in Space and Time with Feature Revisions* with David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Published at the Springer International Journal Empirical Software Engineering, May 2022 (EMSE 2022), pages 1–54, 2022

In Paper D, we present a novel approach: *ECSEST* - Extraction and Composition for Systems Evolving in Space and Time. This paper presents the extension of our approach for feature revision location containing improvements for supporting C language artifacts and reuse of feature revisions to compose new products of a system. Further, in this paper, we present an enhanced analysis of feature evolution in space and time. In comparison to the previous paper (Paper C), we computed new metrics in additional systems and feature revisions. *ECSEST* offers additional support for preprocessor-based systems managed in VCS, such as the location of feature revisions implementation and combination of feature revisions from different commits. The basic ideas for the approach and evaluation were a result of discussions of all the authors. G. Michelon was responsible for implementing the necessary analysis based on C Abstract Syntax Trees (ASTs) for the feature revision location technique extracted with the Eclipse development tools CDT. G. Michelon also implemented the approach for retrieving hints for the composition of new products and the evaluation. The co-authors further contributed to implementing parts of the approach, writing the paper, further discussions, and proofreading the paper.

Paper E [131] *Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants* with David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. In the Proceeding of the 25th International Systems and Software Product Line Conference (SPLC 2021), 6 pages, Leicester, United Kingdom, 2021

In Paper E, we present four challenges concerning feature (revision) location techniques and composition of variants/products with feature/s (revisions) to motivate researchers and tool developers to develop more efficient mechanisms for managing systems evolving in space and time. In this paper, we also provide a benchmark generator and a common ground truth generated from preprocessor-based systems due to the need for introducing new, or optimizing and addressing the limitations of existing techniques. In this work, G. Michelon implemented the necessary analysis and created the benchmark and computation of metrics available. D. Obermann implemented parts of the approach to create the benchmark. The co-authors further contributed to writing the paper, further discussions, and proofreading the paper.

1.2.1 Additional Papers

G. Michelon was also involved in further work and international collaborations with researchers from different universities, countries, and industries focusing on addressing evolution in space and/or time. The additional papers published, accepted, or under revision are listed below by category.

Journal

1. **Gabriela K. Michelon**, Jabier Martinez, Bruno Sotto-Mayor, Aitor Arrieta, Wesley K. G. Assunção, Rui Abreu, Alexander Egyed: *Spectrum-based feature localization for families of systems.* Journal of Systems and Software (JSS) 2022 (Under Revision R2)

   This paper is an extension of the work presented in [124]. In this work, we focus on evolution in space and propose a solution for improving the performance of existing feature location approaches. In previous work [124], we present a Spectrum-based feature localization approach as the first step for feature-based SPL adoption, whereas in the extension, we propose approaches also for the context of families of systems, i.e., different system products resulted from opportunistic reuse. We also compare and improve some results of previous work where G. Michelon is also the main author [128, 129].

2. Stefan Fischer,**Gabriela K. Michelon**, Rudolf Ramler, Lukas Linsbauer, Alexander Egyed: *Automated test reuse for highly configurable software.* Springer International Journal Empirical Software Engineering (EMSE) 2020

   In this paper, we present an approach to automate test reuse by locating features of test variants in order to compose different test variants to help test systems with a large number of possible configurations. This is challenging in highly configurable systems because these systems' tests themselves are rarely configurable and instead built for specific configurations.

Conference

1. **Gabriela K. Michelon**, Lukas Linsbauer, Wesley K. G. Assunção, Stefan Fischer, Alexander Egyed: *A Hybrid Feature Location Technique for Re-Engineering Single*

Systems into Software Product Lines. ACM International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS) 2021

In this work, we present a hybrid feature location technique consisting of two complementary analyses: i) a dynamic analysis by runtime monitoring traces of scenarios exercising features of a system, and ii) a static analysis for refining overlapping traces between features. The goal of this work is to deal with evolution in space by automating the feature location in single systems for re-engineering into software product lines. This paper fostered discussion when presented at the VaMoS conference and resulted in a new collaboration with researchers working in the same domain of hybrid feature location techniques. That is how the collaborations that resulted in the papers [124] and [130] arose.

2. **Gabriela K. Michelon**, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, Wesley K. G. Assunção: *Spectrum-Based Feature Localization: A Case Study using ArgoUML.* ACM International Systems and Software Product Line Conference (SPLC) 2021

   In this work, we explore the use of the Spectrum-based localization technique and compare it with the state-of-the-art feature location techniques for re-engineering single systems into SPLs. We used Spectrum-based localization as a new complementary analysis for the dynamic analysis presented in the hybrid approach by Michelon et al. [128].

3. Stefan Fischer, **Gabriela K. Michelon**, Rudolf Ramler, Lukas Linsbauer, Alexander Egyed: *Automated Reuse of Test Cases for Highly Configurable Software Systems.* Software Engineering (SE) 2021

   This is a Journal-First paper of the aforementioned work [54] published in the Springer International Journal Empirical Software Engineering to increase its visibility.

4. **Gabriela K. Michelon**, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Alexander Egyed: *Mining Feature Revisions in Highly-Configurable Software Systems.* ACM International Systems and Software Product Line Conference (SPLC) 2020

   In this paper, we propose an automated approach to mine features in highly configurable software systems (HCSS) taking into account evolution in space and time. This approach contributed to future research on understanding the characteristics of HCSS and supporting developers during maintenance and evolution tasks presented in the core papers of this thesis [126, 127, 131, 133, 134].

5. **Gabriela K. Michelon**: *Evolving System Families in Space and Time.* Doctoral Symposium ACM International Systems and Software Product Line Conference (SPLC) 2020

   This is a doctoral symposium work presented to the SPL community in order to confirm this thesis goal and investigate future directions. By presenting preliminary results, G. Michelon received valuable feedback and insights on how to conduct the further steps of this thesis research.

6. **Gabriela K. Michelon**, Lukas Linsbauer, Wesley K. G. Assunção, Alexander Egyed: *Comparison-based feature location in ArgoUML variants.* ACM International Systems and Software Product Line Conference (SPLC) 2019

   This work presents a solution to the ArgoUML-SPL feature location challenge posed at SPLC [116]. The approach of the automated feature location is based on the comparison of features and their implementation. Therefore, the more variants are

available the better are the traces computed. This is the first work where G. Michelon was involved in the context of this thesis, which helped advance her understanding of system evolution and further in developing approaches that resulted in this thesis's theoretical and practical contributions.

7. Willian D. F. Mendonça, Silvia R. Vergilio, **Gabriela K. Michelon**, Alexander Egyed, Wesley K. G. Assunção: *Test2Feature: Feature-based Test Traceability Tool for Highly Configurable Software.* ACM International Systems and Software Product Line Conference (SPLC) 2022 (Accepted)

   In this paper, we propose a tool *Test2Feature* based on a static analysis for traceability of test cases to features using the source code of C/C++ annotated HCSS. The tool retrieves reports containing the source code lines that correspond to each feature, as well as the lines and features that correspond to each test case. The reports retrieved by our tool can be used to ease tasks, such as regression testing, feature management, and HCSS evolution and maintenance.

8. Luciano Marchezan, Wesley K. G. Assunção, **Gabriela K. Michelon**, Edvin Herac, Alexander Egyed: *Code Smell Analysis in Cloned Java Variants: the Apo-games Case Study.* ACM International Systems and Software Product Line Conference (SPLC) 2022 (Accepted)

   This work presents a solution for the challenge of identifying the flaws in the design and implementation of the Apo-games family, i.e., cloned variants [87]. To tackle this challenge, we applied an inconsistency and repair approach to detect and suggest solutions for code smells, which appeared repeatedly in multiple products. We could identify a considerable number of code smells resulting from clone-and-own and suggest repairs for them.

Book Chapter

1. Lukas Linsbauer, Stefan Fischer, **Gabriela K. Michelon**, Wesley K. G. Assunção, Paul Grünbacher, Roberto Erick Lopez-Herrejon and Alexander Egyed: *Systematic Software Reuse with Automated Extraction and Composition for Clone-and-Own.* Chapter - Handbook of Re-Engineering Software Intensive Systems into Software Product Lines. Springer 2021 (Accepted)

   This book chapter presents an approach for supporting flexible and intuitive clone-and-own practice for creating variants of a system with automated reuse and centralized maintenance. This is a result of successive works, in which all authors have been contributed with the basic ideas, practical implementation, and evaluation of an automated approach for extraction of feature-to-implementation traces from variants.

Workshop Organization

1. Lea Gerling, Sandra Greiner, Kristof Meixner, **Gabriela K. Michelon**: *Fourth International Workshop on Variability and Evolution of Software-Intensive Systems (VariVolution 2021).* ACM International Systems and Software Product Line Conference (SPLC) 2021

2. Sandra Greiner, Kristof Meixner, **Gabriela K. Michelon**, Philippe Collet: *Fifth International Workshop on Variability and Evolution of Software-Intensive Systems*

(VariVolution 2022). ACM International Systems and Software Product Line Conference (SPLC) 2022 (Accepted)

G. Michelon has been involved in the VariVolution workshop for several years due to the strict relation of the workshop scope and this thesis topic. Participating in this workshop certainly led to more knowledge on how to integrate or improve the management of evolution in space and time. Over the already past four editions of the VariVolution, G. Michelon participated as a listener in the first year of her PhD, a second time actively presenting work for mining features in space and time [132], and the third and fourth time (to happen) as co-organizer of the workshop. The workshop has brought together active researchers and practitioners studying software evolution and variability from different perspectives. Participating in the VariVolution had been a great opportunity for G. Michelon to exchange new ideas that certainly contributed to this thesis. Further, the fifth edition of the VariVolution will be certainly a good opportunity for G. Michelon to discuss her findings acquired over past years of research.

1.3 Outline

The thesis is organized into two parts. The first part (Part I-Prologue) contains Chapters 1-8, where the first chapter introduces the topic of software systems evolving in space and time (Chapter 1). Chapter 2 presents the motivation behind this thesis research. Chapter 3 provides the background necessary to understand concepts in this thesis, and additionally, it quickly overviews the existing state of the art in the field of this thesis. Chapter 4 presents the approach created and used to tackle this thesis research problem and answer the RQs presented. Chapter 5 evaluates the research questions posed in Chapter 1. The first part is then closed by the Chapter 6, which concludes this thesis and outlines potential future research directions. The second part of this thesis (Part II-Papers) includes five Papers A-E that represent the core of this thesis on which the author of this thesis worked on. These papers have been included as originally published/submitted, except for using a consistent layout and bibliography that do not affect the content.

Chapter 2

Motivating Examples

Preprocessor-based systems or SPLs usually have many features with annotation spread across many files and a file can have several features tangled, i.e., interacting with many others. This situation makes the SPL implementation with preprocessor directives difficult to understand and often makes maintenance, evolution, and testing activities time consuming and error-prone tasks [73]. The maintenance and evolution of these systems become more difficult when the SPL has been evolved for a while in a VCS. For instance, to determine which feature has a bug or causes other faults, developers may need to look through a large set of commits of the system development history to find where and when the bug was introduced and which features it affects. However, analyzing manually for the changes of each commit to retrieve the related features is a complex task, especially when multiple features are changed or added in a single commit [17, 76, 195]. As an example, in a single commit (77603db) from the LibSSH[1] system, 415 additions and 338 deletions were performed in 15 files. As we can see in Figure 2.1, the commit message does not reflect which features were changed, and this commit contains refactoring, cleanup debugging messages, inclusion and enhancing of features, and bug fixing. By using Git VCS, we can see for each file which lines were added or removed but not which features are involved in a change. For instance, Figure 2.2 shows multiple changes in one file, but there is no information visible about the annotated features. Developers and engineers have to manually inspect each file and identify for each delta which annotations are wrapping the changes, considering all macros, features, interactions, operations, and header files in a file.

Another situation we found when analyzing real-world systems is that often in a commit the conditional expressions of variation points are changed and not the source code itself. This can lead to misunderstanding of which features were affected by a change. For instance, the changes performed in commit 6f47401 of the LibSSH system affected 14 variation points. However, the source code of the variation points were not affected, but the conditional expression of the variation points replaced the feature `HAVE_SSH1` by the feature `WITH_SSH1`, as shown in the code snippet of Figure 2.3. Therefore, if developers try to find a specific bug by looking for the feature `HAVE_SSH1` in a version of the system with the aforementioned commit, they will be misled, since, at that time, the problem was actually located in the feature `WITH_SSH1`. The same situation can happen when new features are added to or removed from a conditional expression, resulting in changes affecting feature interactions.

Another challenge faced when evolving a preprocessor-based SPL is that developers need to consider all versions at the same time through preprocessor directives. There is no support in VCSs to track changes of features between different versions/releases of preprocessor-based SPLs. Often, changes of a feature in one release of a system have to be propagated to different releases of a system. Figure 2.4 depicts such situation, where, for example, a commit contains features introduced, revised, and removed in one release

[1] https://gitlab.com/libssh/libssh-mirror/

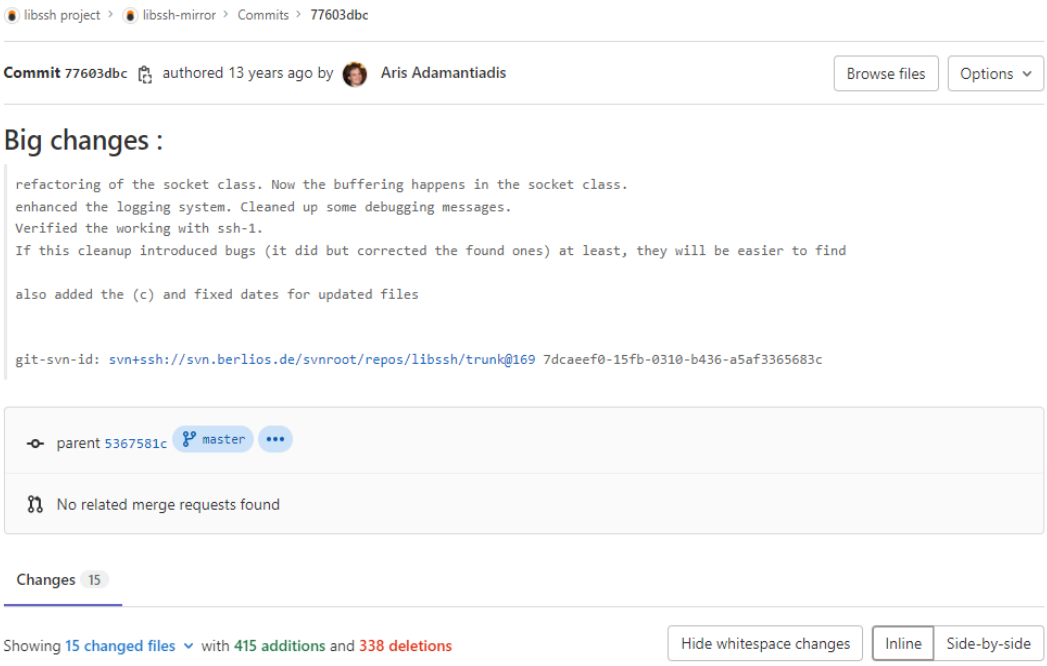


Figure 2.1: Example of a commit 77603db from the LibSSH system involving big changes in multiple files.

Figure 2.2: VCS support to visualize the changed file *session.c* in commit 77603db from the LibSSH system.

libssh/auth.c +3 -3 View file @6f474011

```
... ...  @@ -208,7 +208,7 @@ int ssh_userauth_none(SSH_SESSION *session, const char *username) {
208 208
209 209    enter_function();
210 210
211     - #ifdef HAVE_SSH1
    211 + #ifdef WITH_SSH1
212 212    if (session->version == 1) {
213 213      ssh_userauth1_none(session, username);
214 214      leave_function();
... ...  @@ -314,7 +314,7 @@ int ssh_userauth_offer_pubkey(SSH_SESSION *session, const char *username,
314 314
315 315    enter_function();
316 316
317     - #ifdef HAVE_SSH1
    317 + #ifdef WITH_SSH1
318 318    if (session->version == 1) {
319 319      ssh_userauth1_offer_pubkey(session, username, type, publickey);
320 320      leave_function();
... ...  @@ -671,7 +671,7 @@ int ssh_userauth_password(SSH_SESSION *session, const char *username,
671 671
672 672    enter_function();
673 673
674     - #ifdef HAVE_SSH1
    674 + #ifdef WITH_SSH1
675 675    if (session->version == 1) {
676 676      rc = ssh_userauth1_password(session, username, password);
677 677      leave_function();
... ...
```

Figure 2.3: Example of revising the features `HAVE_SSH1` and `WITH_SSH1` by changing conditional expressions in the file *auth.c* of commit 6f4740 from the LibSSH system.

(e.g., V4.2) of a system, and may imply in propagating these features to another releases (e.g., V3.2, V2.1, V1.4). However, there is no automated support for propagating the entire implementation of a feature and neither for visualizing the differences and interactions of features between releases or commits. Therefore, an automated approach to propagate the entire implementation between releases can aid maintenance, evolution and reuse tasks.

For instance, in the commit history of the SQLite system[2], a C-language library implementing the most widely used database engine in the world [183], the feature `SQLITE_TEST`

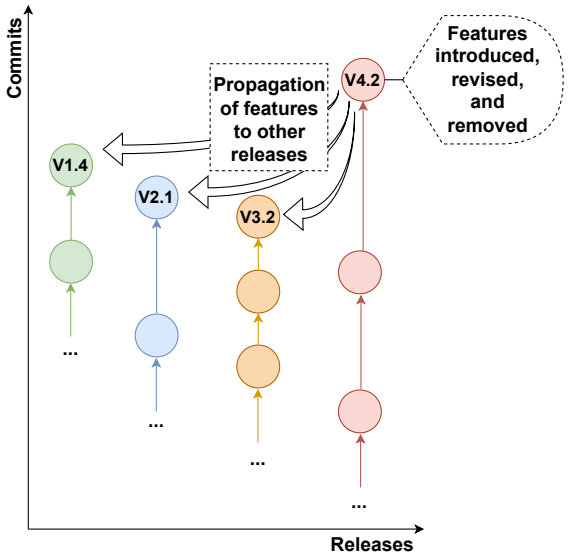


Figure 2.4: Propagation of features between releases of a system.

[2]https://www.sqlite.org/src/info/7b4583f932ff0933

was introduced to the release 3.9. The introduction of the feature `SQLITE_TEST` was propagated to other three newer releases: 3.18, 3.19, and 3.22. Another example can be seen in a pull request[3] of the Marlin SPL, an open-source firmware for 3D printers. This pull request was performed by users from the release 1.1.x. regarding the implementation of the feature `BLTouch` for version 3.0 existing in the release 2.0.x that should be added to the previous release 1.1.x. Otherwise, buying a new BL-Touch with version 3.0 would be incompatible with the release v1.1.9. The propagation of this feature involved 114 files, where 3,208 lines were added and 911 deleted through 1,100 preprocessor directives. This is an example of a software maintenance and evolution task that required mining the differences of this feature between the two releases as well as propagating and reusing the implementation of a revision of a feature. These examples highlight the need of support for mining and locating the implementation of features at multiple points in time. Such support should enable reusing of feature revisions for composition of new software system products with different combinations of feature revisions from different points in time.

[3] https://github.com/MarlinFirmware/Marlin/pull/14839

Chapter 3

Background and State of the Art

This chapter overviews the background necessary for understanding this thesis. It explains how SPL evolution is addressed in VCSs and preprocessor-based systems. This chapter also describes the basic concepts about feature location, since it is part of the main contributions of this thesis. Additionally, we also present a rough overview of the state of the art for evolution in space and time.

3.1 Software Product Line

An SPL consists of a platform with a core and variable parts represented by a set of features implemented with a variability mechanism. The features of an SPL can be systematically reused to facilitate mass customization of different products according to the customers' requirements [155], improving the productivity of software companies, reducing costs, and the human effort in developing systems [86, 128]. There are many definitions of the term *feature* in the literature [8, 22]. A feature can represent an end-user functionality or an entity used for development purposes, such as test, debug, and deployment, or serve as an abstraction for specifying, documenting, comprehending, managing, and reusing any functionality of a system [22, 86, 89]. The variability mechanisms enable users or developers enabling or disabling optional features of an SPL [8, 155]. To name some of them: feature toggles and runtime variability [120, 162]; annotative mechanisms with support of some tools for easing the comprehension of annotation-based systems [15], such as C-CLR [181], FeatureCommander [49], variation editor [94], PEoPL IDE [136] and CIDE [77]; and compositional mechanisms and modularization with feature-oriented programming [9], supported by tools such as AHEAD [18] or FeatureHouse [10], or aspect-oriented programming [78, 92], delta-oriented programming [170], or context-oriented programming [72]. Among the existing variability mechanisms, the preprocessor is the most widely used in industrial and open-source systems [8, 74, 96, 119, 137], which is presented in the next section.

3.2 Preprocessor-based Systems

The preprocessor is a mechanism to manipulate the source code of software systems with directives before compilation [8]. It allows to implement variation points by annotating feature fragments in the source code with preprocessor directives (e.g., `#ifdef`, `#if`, `#ifndef`, `#else`, and `#elif`) [86]. The annotated blocks of code with preprocessor directives involve macros in conditional expressions as shown in Figure 3.1. Line 6 shows an example of a conditional expression involving macros connected via && logical operator. This conditional expression is evaluated as `TRUE` if `MD5_DIGEST_LEN` is greater than five `AND _LIBSSH_H` is true.

Macros defining features are usually defined using command-line parameters passed to

```
1  #ifdef WITH_SERVER
2      #define MD5_DIGEST_LEN 16
3  #endif
4
5  #ifdef __cplusplus
6      #if _LIBSSH_H && MD5_DIGEST_LEN > 5
7          <code>
8      #endif
9  #endif
```

Figure 3.1: Conditional blocks of feature implementations.

the preprocessor for compilation. Although macros can be defined internally in source code using the `#define` directive, they cannot be selected directly by users. Figure 3.1 shows four different macros (`WITH_SERVER`, `MD5_DIGEST_LEN`, `__cplusplus` and `_LIBSSH_H`). In this example, the macro `MD5_DIGEST_LEN` cannot be selected by the user and is not a feature. `MD5_DIGEST_LEN` is defined in the conditional block corresponding to a conditional expression of the feature `WITH_SERVER` in Lines 1-3. This means, the macro `MD5_DIGEST_LEN` is defined when the feature `WITH_SERVER` is selected by the user. In this thesis, we refer to features as the macros that can be selected by the user and thus can be any functionality of the system visible to the end-user, as well as support for developers comprehending, implementing, and maintaining a system [88].

The preprocessor is a lightweight tool for product line adoption [8]. It eases the configuration and derivation of variants by preprocessing the source code according to the features defined by users. Then, code fragments are included or removed before compilation. Most developers are familiar with this tool, which is easy to use, learn, and adopt [8]. This is why it is the most common mechanism for implementing variability in product lines in industrial practice [8]. However, preprocessors also have been criticized by academics with the term "`#ifdef hell`" [49, 109, 118], mainly because preprocessor annotations obfuscate the source code and the system features can be scattered over many files and lines, tangled with other features [127]. This can make it hard to understand and maintain the source code [8, 132]. Next, we describe existing analysis approaches to determine where the features code is located, known as feature location or variability mining [40, 166].

3.3 Feature Location

Feature location techniques aim at locating software artifacts that implement a specific feature and feature location is one of the most important and common tasks performed by developers during software maintenance and evolution activities [40, 88, 166]. In addition, feature location is commonly used to identify and manage common and variable source code in families of software systems that emerged from an ad-hoc reuse [166]. Feature location is also an essential task for re-engineering existing system products into an SPL [12].

Existing feature location techniques use different approaches, such as textual, static, or dynamic analysis [40]. These techniques have also been combined in hybrid feature location techniques, where different techniques are used to overcome the limitations of each other [106, 128, 169]. Textual feature location is commonly based on techniques and models of information retrieval and natural language processing [29, 34, 164] that analyze naming conventions in source code artifacts, similar to a query that describes a feature. Static techniques are based on comparing common artifacts and features in relation to a set of system products/variants [129, 134] or by static analysis of control or data flow dependencies [1]. The dynamic analysis consists of running the system and invoking the feature of interest and record runtime traces of the source code that were executed [43].

The majority of existing feature location techniques can help with the evolution in

space [40, 166] but not with the evolution over time. The first feature location technique which was able to locate feature revisions, i.e., divergent implementations of features over time, was presented in our previous work [134] (see Paper C). Therefore, the terminology *feature revision* in this thesis is used to refer to a feature at a specific point in time, where the feature implementation differs from other points in time during the life cycle of a software system. The feature revision location can support evolution of features in software systems implemented in version control systems, as well as being integrated with variation control systems to support the user when making changes to a product line [98, 104, 133] (see Paper D).

3.4 Version and Variation Control Systems

VCSs are the facto tool for managing and tracking different revisions of files or the whole software system, i.e., evolution over time [8]. Each revision is identified by a unique number. VCSs mechanisms of branch and fork are used for variant management by cloning whole systems in a coarse-grained way [104]. This strategy, however, does not scale with the number of variants, leading to high maintenance efforts. A way to tackle this problem is the adoption of an SPL [12]. Currently, the vast majority of industrial SPLs are realized with preprocessor directives because of easy integration with VCSs [8, 21, 104, 119].

VCSs are used to version the whole platform by creating different releases or branches. Every new release introduces, removes, and revises features, which can involve features for deployment, i.e., for customers, and also for development and testing, i.e., not necessarily features that provide core functionality for customers [22]. The life cycle of a release then is composed of commits over time. Every new commit introduces changes in a (set of) patch(es) of code, where changes in a (set of) patch(es) of code are represented by lines added or removed in relation to two line-oriented text files. There exists approaches for propagation of patches of code focusing on automating program repair by reusing and adapting a patch that is already available [177, 178]. However, these approaches do not relate patches of code to features, i.e., they do not take into account the features annotated in the source code. Therefore, there is no automated mapping between changes of patches of code and annotated feature(s) and interactions of a commit [126]. Current approaches do not enable propagating/reusing the entire implementation of feature revisions.

Although variation control systems (VarCSs) are promising for handling variants and evolution over time at the level of features, they are not mature tools and lack support for collaborative and distributed development [70]. Further, VarCSs have their own proprietary repository technology, besides unfamiliar operations [104]. These are some of the reasons why developers are reluctant to use VarCSs, and decide to use more popular and convenient tools, such as preprocessor-based systems managed in VCSs to deal with evolution in space and time. This thesis, therefore, focus on addressing VCSs' limitations to support SPL evolution in space and time, such as limited support for retrieving information at the feature level (commits usually involve changes in multiple files and features [17, 76, 195]), and limited support for visualization, which only presents the history at the file and line level [108].

Chapter 4

Approach

In this chapter we present an non-intrusive approach to support preprocessor-based SPLs and that can be integrated with the development of the current most used mechanisms. It is a novel support for feature revision change analysis and propagation, which reduces the time for traceability of feature revisions to implementation artifacts and help to understand, maintain and reuse preprocessor-based SPLs managed in VCSs. Nonetheless, we also present *ECSEST* (**E**xtraction and **C**omposition for **S**ystems **E**volving in **S**pace and **T**ime) approach. This is an approach for supporting software systems evolving in space and time at the feature level for any artifact type. *ECSEST* is an automated support for feature revision location that can save time and effort in aiding the maintenance and reuse of families of software systems evolving in space and time.

4.1 Support for Preprocessor-based SPLs Managed in VCSs

In order to understand how preprocessor-based SPLs evolve over time in VCSs, we created an automated approach for analyzing the evolution of features in space and time (Paper A). To investigate how often and which features are revised through their life cycle is a complex and error-prone task. Although there exists guidelines recommending to perform small and cohesive commits, this is not what usually happens in practice [17]. Often single commits involve multiple independent changes. In addition, the implementation of features with preprocessor directives is often highly complex [160]. Features implementations are scattered in many variation points and files, and interacting with multiple features. Such analysis has to take into account header files, macros in conditions, and how macros are defined within `#define` directives. Furthermore, it is necessary to treat `#else` and `#ifndef` directives when assigning changes to features to not misunderstand the system evolution [110].

We built our own approach to empirically analyze the life cycle of features because existing approaches [39, 44, 74, 96, 119, 147, 160, 191] have some limitations. For example, existing approaches do not solve constraints of complex expressions, involving Boolean, arithmetic operations, and comparisons, or they focus on computing metrics based on the programming language. Computing metrics at a low level such the abstract syntax tree (AST) is expensive in terms of computational resources and runtime performance. Further, existing approaches do not compute all the metrics needed for our analysis. For our analysis, we wanted to compute metrics at a high level of abstraction, such as the annotated blocks of code of features. We built an approach for mining the life cycle of features in the whole history of commits, overcoming existing approaches limitations. Our approach thus relies on a constraint satisfaction problem (CSP) solver to reliably identify interacting features [19], considering all corner cases of preprocessor annotations [110]. Therefore, we defined an approach to automatically computing characteristics of feature changes, feature implementation complexity, and feature interactions at multiple points in time, i.e., in every commit of a system.

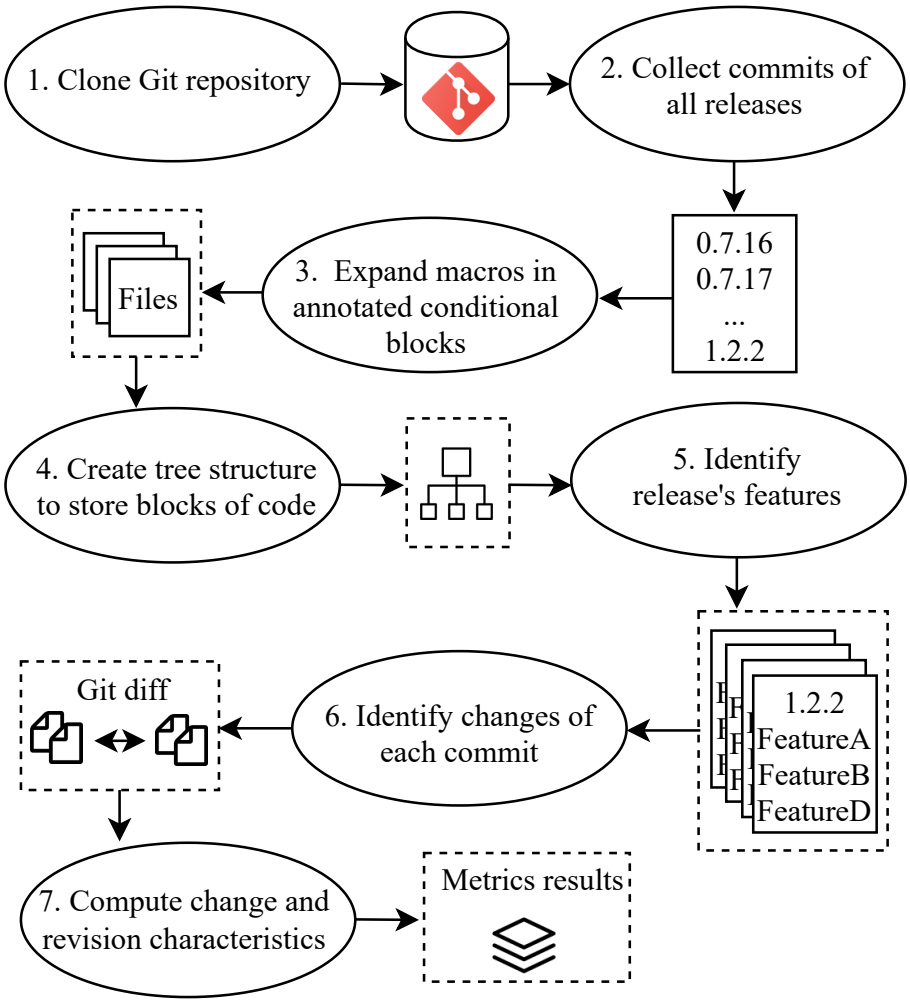


Figure 4.1: Automated approach for mining the life cycle of features.

4.1.1 Mining Feature Revisions

Figure 4.1 shows the steps performed by our approach for mining feature revisions and their implementation and change characteristics. The approach uses as input the VCS repository of a preprocessor-based SPL. Thus, the first step is to clone the repository to a local machine, where the user will run our approach for analysis. In the second step our approach collects information from all Git commits of all releases, or from a set of selected commits, if preferred by the user. The third step is a partial pre-processing of the checked-out commit files, i.e., annotated macros are expanded such that annotated conditional blocks only consist of literals. The fourth step obtains the files resulting from expanding macros in annotated conditional blocks and abstract the source code at the annotated blocks of code of features in a tree structure. The tree structure contains preprocessor directive nodes to represent the entire content of a 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. The remaining steps concern the mining of feature revisions and computation of metrics.

The fifth step identifies the features of the system by collecting all macros of conditional blocks and all `#defines` of all files for every commit of a release. Thus, our approach takes as features 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. We use the Listing 4.1 to represent a file in the first commit of a system (Commit #0) and Listing 4.2 to represent the same file

in the second commit (Commit #1). In Listing 4.1, the macros `A` and `Y` are part of the set of features because they can only be set externally by the user. The other macros `X`, `B` and `C` (non-boolean) are not considered features as they cannot be set externally, i.e., they are defined within the source code via `#define` directive.

```
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 4.1. 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     <changed code>
12 #endif
13
14 #ifndef A
15     <code>
16 #else
17      <changed code>
18 #endif
```

Listing 4.2. Commit #1.

After obtaining the features of an SPL, our approach proceed to the sixth step to identify the patches of code that has a Git diff [33], i.e., fragments of code that differ for the same file from one commit to another. With this automated approach, developers and engineers know which features changed in which commits. For example, the changes in Commit #1 (Listing 4.2) were made in Lines 11 and 17. To determine which features changed with the change in Line 11 we cannot simply assume the macros `B` and `C` as changed features, because they are defined within the source code (Lines 6-7). Our approach thus considers all the subsequent lines of the file to analyze the feature implications to all the blocks of code responsible to execute Lines 10-12. In this example, there is 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. With the implications, the approach builds constraints, which are handed to the CSP solver, which gives to the user a reliable solution that satisfies the constraint problem, i.e., which features have to be selected to execute that particular block of code. For the change in Line 11, the constraints are: $(Y \implies X) \wedge (X \implies B) \wedge (X \implies (C = 9)) \wedge (B \wedge (C > 5)) \wedge X \wedge Y$. This constraint problem is satisfied when $\texttt{Y} = T$, i.e., when feature `Y` is selected as `X`, `B` and `C` are internal macros and not features of the system. Then, the change in Line 11 is then assigned to feature `Y`. In Lines 14-18, we have corner cases with `#ifndef A` and `#else` directives. Our approach uses a heuristic that changes in negated conditions such as `#ifndef A` are considered part of the core of the system, as the code inside this block (Line 15) can only be executed when no feature is selected. Building the implications is very important, as we can see with the changes in Line 17, which are inside an `#else` conditional block that can only be executed when feature `A` is selected. Thus, the change in Line 17 is assigned to feature `A`.

The seventh step is related to the computation of metrics for change and revision characteristics (Table 4.1). These metrics are based on a literature survey [7, 50, 96, 160] of metrics useful to measure the complexity of features implementation in preprocessor-based SPLs. The change characteristics are computed for every feature that changed in a commit. The revision characteristics are computed for every block of code presented in the source

files of a commit. The revision characteristics are related to the implementation of a feature in a specific commit, and not about the changes performed in the implementation of a feature. To illustrate the metrics computed, let's continue with our example containing corner cases in Listings 4.1 and Listing 4.2. The metrics for change characteristics of Line 11 assigned to feature `Y` 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 code containing changes that affected 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).

Our illustrative example shows 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. Regarding the change in Line 17 assigned to feature `A`, the change characteristics are as follows: LOC A = 1, LOC R = 1, SD #IFs = 2 (two patches of code containing changes that affect 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 14-16 is computed as part of the feature `BASE`, rather than `A`. Then, the

Table 4.1: 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

revision characteristics of `BASE` are as follows: LOC = 1, SD #IFs = 0, SD #NIFs = 1 (`#ifndef` line 14), SD File = 1, TD = 1, ND = 0, NOTLB = 1, NONTLB = 0.

The output retrieved by our automated approach for mining feature revisions and compute change and revision characteristics enable us to observe when a specific feature has different implementations between one release to another, in terms of size, number of variation points, feature interactions, content and purpose. Therefore, by using our mining approach, we obtained results to empirically analyzing the life cycle of features, which revealed to us a challenge faced by systems evolving in space and time. As shown in our motivating examples (Chapter 2), reusing different implementations of a feature in different releases of a system often happens in preprocessor-based SPLs managed in VCSs. We thus built an approach for change analysis and propagation of feature revisions (Paper B).

4.1.2 Feature Revision Change Analysis and Propagation

Figure 4.2 shows an overview of our approach for feature revision change analysis and propagation (Paper B), which relies our mining approach already presented in this chapter. The change analysis and propagation of feature revisions start with a feature, or a set of features, needed to the user's analyses, the pair of releases where the user would like to analyze and/or propagate a feature, or a set of features. There is an optional step called (*mining releases features*), which is our aforementioned mining approach for mining which features exist in a release. This step is performed in case the developer/user does not know the features existing in a release of a system. Following, the second step, *feature revision change analysis*, is necessary to know for each delta, i.e., differences in the source code between one release to another, which change belongs to which feature revision(s) to be propagated.

The two releases' commits are referred to as *origin* O and *destination* D, and are releases in which a user wants to propagate a feature or a set of features from O to D. To compute the change analysis, firstly, it is necessary to obtain the files of the two commits O and D. Then, our approach identifies the changes between O and D, which can be propagated. A *file change* FC corresponds to a file that can be either deleted, inserted, or modified, as described next:

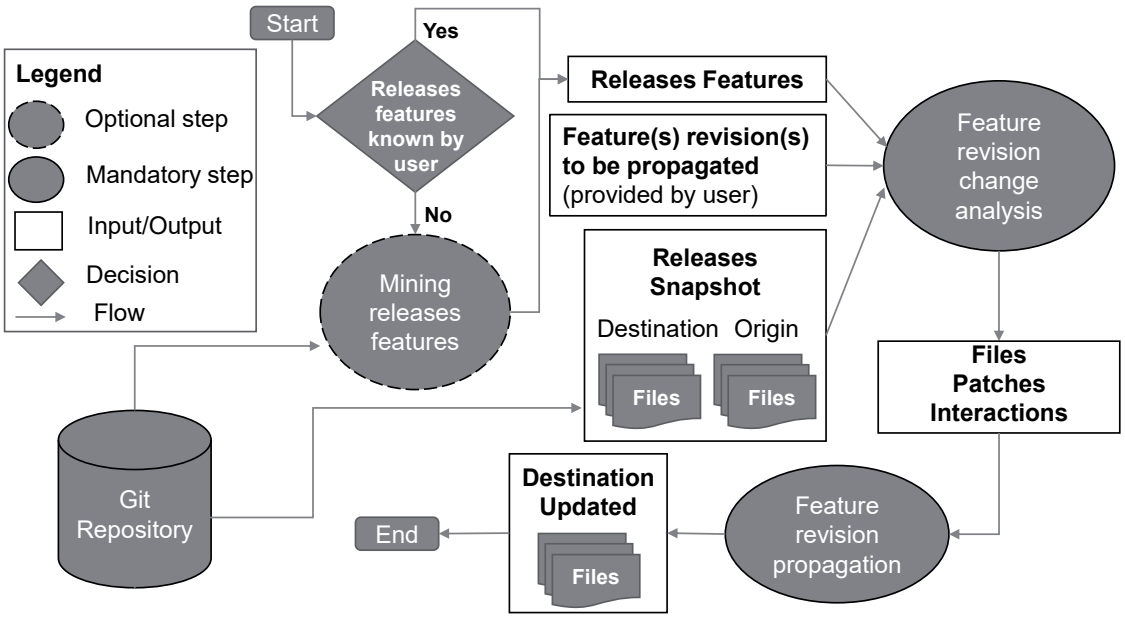


Figure 4.2: Automated approach for change analysis and propagation of feature revisions.

DELETE. Consists of a file existing in commit D that does not exist in O. The reference of this change type in FC contains the number 1 to represent the first line, and the total number of lines of the deleted file (n) to represent the last line of the deleted file.

INSERT. Consists of a file in commit O that does not exist in D. The reference of this change type in FC contains the number 1 to represent the first line, and the total number of lines of the inserted file (n) to represent the last line of the inserted file.

MODIFY. Consists of a file existing in commit O that also exists in D, but in a different state. A modified file may be affected by many deltas. Thus, to assign the reference to each of the deltas in FC, our approach uses the patches of code contained in the file's deltas. Delta is a way of storing or transmitting data in the form of differences between sequential lines rather than complete files. For each delta, our approach obtains its type (`REMOVE`, `ADD`, or `CHANGE`), as shown in our examples of Figure 4.3. An `ADD` delta (Line 2-2 in O, Figure 4.3a) contains lines added in O in comparison to D, referencing the line number of the beginning and end of the lines added in the `ADD` delta. A `REMOVE` delta contains lines removed from O to D, referencing the line number of the beginning and end of the removal (Line 2-2 in D, Figure 4.3b). A `CHANGE` delta (Figure 4.3c, Line 2-3 and Figure 4.3d, Line 2-4) contains lines changed, consecutive removals and additions, from O to D. The file change thus contains not only one, but two references. The first reference contains the beginning and end of the addition delta lines (Lines 2-2 in O, Figure 4.3c, and Lines 2-3 in O, Figure 4.3d), while the second reference contains the beginning and end of the removal delta lines (Lines 2-2 in D, Figure 4.3c, and Lines 2-2 in D, Figure 4.3d).

The output of the feature revision change analysis contains the conditional blocks of code corresponding to the deltas and their respective feature(s) to be propagated. And for each delta, the feature revision change analysis also describes the feature interactions and the features that might be affected, i.e., nested features in conditional blocks of code. A feature interaction refers to all the features that are in the set of revised features of a conditional block of code, i.e., the features of the configuration necessary to execute a particular block of code related to a change. For instance, using our running example presented in Figure 4.4, we presented three deltas modifying the code snippet adapted from the LibSSH system in Figure 4.5: The first delta (Line 1, Figure 4.5) is a `CHANGE` delta, where Line 1 in D is removed, which contains a conditional expression (`#if WITH_SERVER`) and a new line added (Line 1 in O, Figure 4.5) to substitute the conditional expression by another (`#if WITH_SERVER && WITH_SSH1`). The feature revision change analysis automates the analyses of which features were introduced, revised, and deleted. For this delta (Line 1

```
1|1   #if WITH_SERVER
  2 +        <new code>
2|3          <code>
3|4          #define MD5_DIGEST_LEN 16
4|5   #endif
```

(a) `ADD` delta.

```
1|1   #if WITH_SERVER
2|   -        <code>
3|2          #define MD5_DIGEST_LEN 16
4|3   #endif
```

(b) `REMOVE` delta.

```
1|1   #if WITH_SERVER
2|   -        <code>
 |2 +        <new code>
3|3          #define MD5_DIGEST_LEN 16
4|4   #endif
```

(c) `CHANGE` delta.

```
1|1   #if WITH_SERVER
 |2 +        <new code>
 |3 +        <new code>
2|   -        <code>
3|4          #define MD5_DIGEST_LEN 16
4|5   #endif
```

(d) `CHANGE` delta.

Figure 4.3: Examples of delta types of a modified file (FC `MODIFY`) resulting in new revisions of the feature `WITH_SERVER`.

```
1 #if WITH_SERVER
2       <code>
3       #define MD5_DIGEST_LEN 16
4 #endif
5
6 #if __cplusplus
7       #if _LIBSSH_H && MD5_DIGEST_LEN > 5
8           <code>
9       #endif
10 #endif
```

Figure 4.4: Code snippet adapted from LibSSH.

removed in D and Line 1 added in O, Figure 4.5), one feature was introduced, because after this change, the conditional block of code in Lines 1-4, Figure 4.4 has a new feature `WITH_SSH1` (assuming our system has only the features presented in Figure 4.4). The feature `WITH_SERVER` interacts with the feature `WITH_SSH1` for the changes regarding the conditional block of code in Lines 1-4, Figure 4.4. Thus, before propagating the implementation of the feature `WITH_SSH1`, the feature revision change analysis provides delta(s) of inserted, deleted, or modified files, lines added and/or removed, the features that were revised, introduced, and deleted, as well as feature interactions.

The `REMOVE` deltas in Line 6 and 10 in D (Figure 4.5) are deltas resulting in deleting the feature `__cplusplus`. In this example, propagating the deletion of the feature `__cplusplus` from the system has interaction with the code of the core of the system (`BASE`) and the feature `_LIBSSH_H` might be impacted, as it is nested with the conditional block of code respective to the `REMOVE` deltas in Line 6 and 10 in D. The `ADD` delta in Line 8 in O is the result of a revision of the feature `_LIBSSH_H`. If the feature deletion is propagated, this also means that the feature `_LIBSSH_H` does not depend anymore on the feature `__cplusplus`, but still interacts with `WITH_SERVER` because `WITH_SERVER` defines `MD5_DIGEST_LEN`.

After performing the feature change analysis, i.e., retrieving which files, patches, and feature interactions will be affected by propagating a feature revision, our approach can perform the next step. The *feature revision propagation* is based on the selection of deltas obtained in the *Feature Revision Change Analysis*. To propagate a feature revision considering differences between O and D, our approach gets the snapshot files from the Git repository of O and D releases. Propagating a feature revision implementation for an inserted file means copying the file from commit O to the resulting directory, as the file does not yet exist in commit D. Then, all files are copied from D to the resulting directory excluding the deleted and modified files of D. For modified files, our approach obtains the file from O as well as from D. Then, our approach creates a modified file line by line,

```
1|  - #if WITH_SERVER
 |1 + #if WITH_SERVER && WITH_SSH1
2          <code>
3          #define MD5_DIGEST_LEN 16
4    #endif
5
6|  - #if __cplusplus
7|6         #if _LIBSSH_H && MD5_DIGEST_LEN > 5
8|7             <code>
 |8 +           <new code>
9|9         #endif
10|  - #endif
```

Figure 4.5: Examples of deltas resulting in one feature introduced (`WITH_SSH1`), one feature revised (_LIBSSH_H) and one feature deleted (`__cplusplus`).

using a counter starting from zero and ending with the same number of lines of the file of D. Before writing a line, our approach checks whether the line is in a position of a delta, where the line must be removed or a new line must be added. In case of a removed line, the approach does not add the respective line. In case of an added line, the approach adds the respective line from O to D before continuing with the next lines of the file of D. When more lines are added in sequence, then all lines are added before continuing with the next lines of the file of D.

To illustrate the feature revision propagation of a modified file, we will continue with our running example presented in Figure 4.4. Figure 4.3 shows possible delta types in *FC* `MODIFY`: Figure 4.3a contains an `ADD` delta; Figure 4.3b contains a `REMOVE` delta; and Figures 4.3c and 4.3d contain a `CHANGE` delta, where a `CHANGE` delta can be one line/a sequence of lines removed followed by one line/a sequence of lines added (Figure 4.3c). A `CHANGE` delta can be either one line or a sequence of lines added followed by one or a sequence of lines removed (Figure 4.3d). The `ADD` delta contains one reference with one line added (Line 2 in O, Figure 4.3a) before Line 2 in D, i.e., from the previous revision of the feature `WITH_SERVER` (Figure 4.4). To propagate a feature revision from the O snapshot to D snapshot, our approach retrieves each delta for each file containing a feature revision. For the example of an `ADD` delta, the approach adds the lines of the added lines reference in O, beginning in the line number of the corresponding file in D where the delta begins. For example, the lines in Figure 4.4 are copied to the new version of the D file until there is a line number corresponding to the `ADD` delta in Figure 4.3a. As there is no line removed, the next lines existing in D are copied right after the line added. As shown in Figure 4.4, in D the line number 2 was "<code>" and now this line is in the third line of the file in D because a new line has been added in the second line number of the D file. The `REMOVE` delta contains one reference with one line removed (Line 2 in D, Figure 4.3b), which was the Line 2 of previous revision of the feature `WITH_SERVER` (Figure 4.4). The result of feature revision propagation is then the snapshot of a destination release containing the feature revision(s) propagated.

4.2 Support for Families of Software Systems Evolving in Space and Time

Features can have different implementations at multiple points in time along with system evolution. When conducting this thesis research, we observed that there was no automated approach to make the traceability of features to their different implementations at different points and time, as well as no automated approach enabling the reuse of feature revisions to create different products of a system. As we presented in Chapter 3, feature location is one of the most important and essential tasks for supporting software maintenance and evolution tasks, as well as in the re-engineering process [13, 164]. Therefore, in this thesis we also presented *ECSEST* (**E**xtraction and **C**omposition for **S**ystems **E**volving in **S**pace and **T**ime). *ECSEST* is an automated approach for feature revision location that can save time and effort in aiding the process of re-engineering of software systems into SPLs at the level of feature revisions, as well as aiding the maintenance and evolution of systems evolving in space and time (Paper D).

Figure 4.6 presents an overview of *ECSEST* for locating and composing variants/products with feature revisions. The approach is implemented in the ECCO VarCS[1], which supports the evolution of arbitrary types of artifacts [134] based on plug-in architecture [104]. Step 1 in Figure 4.6) is the feature revision location for mapping feature revisions to artifacts from existing software system products/variants. The feature revision location is an incremental process, which receives as input a product implementation and a configuration characterizing

[1] https://github.com/jku-isse/ecco

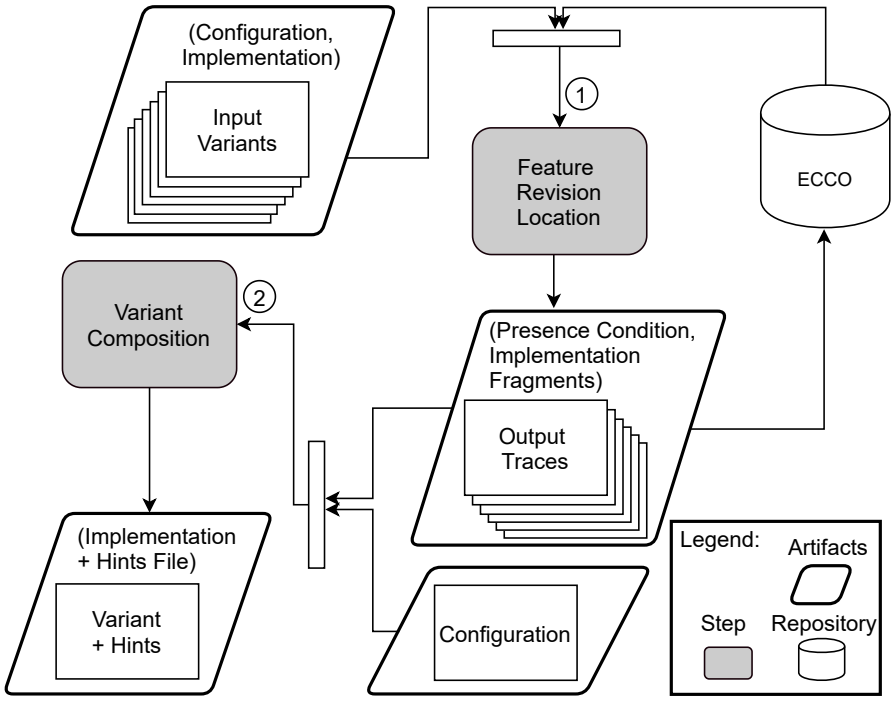


Figure 4.6: The *ECSEST* approach overview.

its features at a specific point in time. This step creates new traces and refines existing ones in the ECCO repository for every new input variant/product. Our approach for variant composition (Step 2 in Figure 4.6), requires as input a configuration provided by the user and the output traces stored in the ECCO repository created when locating feature revisions. The variant composition results in the product implementation and a file with hints to help the product completion. In the following, we give details of the processes of the feature revision location and variant composition.

4.2.1 Feature Revision Location

Our approach for feature revision location maps artifacts in common between variants to a feature called `BASE` that represents the core of a system, i.e., not related to the artifacts of features of a system. The feature revision location then analyzes in how many variants a feature revision appears, in how many variants a (set of) artifact(s) appears, and in how many variants a pair of feature revision(s) and a (set of) artifact(s) appear together. In this way, all artifacts are mapped to feature revisions. This mapping between feature revisions and artifacts consists of presence or absence of features for every artifact of a system in the form of a disjunctive normal form (DNF) formula. The literals of the DNF formula are features, i.e., a set of feature revisions. Whether a feature revision is mapped to an artifact depends on five intuitive rules that have already been proven to work properly for feature location [129]. Given two variants v_1 and v_2 of a system:

1. Common artifacts in v_1 and v_2 *likely* trace to common features.
2. Artifacts in v_1 and not v_2 *likely* trace to features that are in v_1 and not v_2, and vice versa.
3. Artifacts in v_1 and not v_2 *cannot* trace to features that are in v_2 and not v_1, and vice versa.
4. Artifacts in v_1 and not v_2 *can at most* trace to features that are in v_1, and vice versa.

5. Artifacts in v_1 and v_2 *can at most* trace to features that are in v_1 or v_2.

Regarding feature revisions, only one revision of a feature can be present in any given variant. In other words, if a feature is present in a variant, it is present in exactly one revision. Regarding artifacts, we do not consider every artifact individually, but cluster artifacts that never appear without each other in any variant and assign presence conditions to those clusters instead of every individual artifact. We use counters for mapping feature revisions to artifacts. For every feature revision, we count in how many input variants it was contained, for every artifact cluster in how many input variants it was contained, and for every pair of feature revision and artifact cluster in how many input variants both were contained together. This has the advantage that it works incrementally, i.e., new input variants can be added whenever necessary, simply by increasing the respective counters. Hence, already computed traces do not have to be recomputed when a new variant is encountered. Instead, the counters are simply increased and the existing presence conditions are trimmed by removing the feature revisions for which the above conditions do not hold anymore.

4.2.2 Variant composition

Aiming to facilitate the composition of new variants with feature revisions, we presented an approach for composing new variants with different combinations of feature revisions. Our approach for composing a variant works similar to a VCS checkout operation. It retrieves a working copy of the content from a repository. Our approach checks out a variant by joining the artifacts from a set of traces given a configuration with a set of feature revisions. Further, our approach for composing new variants retrieves hints with possible conflicts and interactions when creating a variant with a new set of feature revisions. Whether a variant has a valid configuration or not, it is not part of our approach. This thesis focus on the domain implementation and product derivation [8]. Therefore, variability models, which denote a set of choices and their dependencies for obtaining configurations [161] is out of the scope of this thesis. Nonetheless, in order to help in the composition of variants with new combination of feature revisions, the approach for composition retrieve hints, which show traces containing possible surplus and/or missing implementation artifacts of feature revisions used to compose a variant. The retrieved hints by our approach can be used by developers and engineers to analyze which artifacts may need to be added and/or removed for completing a product variant. The retrieved hints contains the trace identifier (hash code), which can be used to look in the ECCO repository which artifacts belong to a stored trace.

Chapter 5

Evaluation and Results Overview

We conducted an empirical study to understand the phenomenon of feature evolution in space and time in preprocessor-based SPLs managed in VCSs. The results of our empirical study enabled us to answer the RQ1, RQ2, and RQ3 of this thesis. The empirical study was performed with our automated support presented in Paper A analyzing the entire history evolution of four real-world preprocessor-based SPLs, covering a total of 37,500 commits from up to 20 years of development. Our analysis differs from previous work by analyzing and computing metrics of the entire feature life cycle. Existing work analyze smaller number of commits and do not compute all the set of metrics computed by our tool support [82, 95, 96, 118, 147, 152, 160, 191]. For instance, analyzing feature characteristics in every single commit of a preprocessor-based SPL enables to identify feature interactions and evolution of implementation complexity. In addition, our analysis computes metrics, such as change characteristics that make clear which features are affected in every commit of a preprocessor-based SPL.

Evaluating our RQ1, we found out that features are typically not removed after being introduced. Most of the removed features in later revisions of a system are usually features used for test purposes. Interestingly, features are revised 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 over time. Overall, our analysis of the evolution in space of the entire life cycle of a preprocessor-based SPL presented similar results compared to an analysis of few releases of the Linux kernel [82, 152]. Analyzing addition and removal of features in the variability model (KConfig files) of the Linux kernel, it was observed that evolution in space ranged between 3%-7% of the commits, which seems to be a small percentage, but in proportion to a large number of commits is still a large number of commits introducing and removing features. Yet, about 7% of the commits analyzed in few releases of the Linux kernel revised features, similar to our subject systems (up to 8% of all commits), which shows that the frequency of evolution over time affecting features in a large system such as Linux kernel is similar to smaller SPLs, such as our subject systems. Therefore, knowing which commits affect which features can reduce the effort of analyzing and verifying the commits with feature evolution in space and time.

To answer RQ2 and RQ3, we selected metrics to measure feature implementation complexity in preprocessor-based SPLs based on a literature survey [50, 160]. The metrics count the number of variation points (scattering degree), feature interactions (tangling degree), and nesting features inside variation points (nesting degree) of a feature. In addition, we also computed the number of variation points of a feature revision that are top-level branches (NOTLB), i.e., an outermost variation point, and non-top-level branches (NONTLB), i.e., a variation point enclosed by another variation point. Following Queiroz et al.'s [160] threshold for feature characteristics, in our study, some of our subject systems contained complex changes in terms of number of variation points affected (≥ 7) and feature interactions (≥ 2). By analyzing the scope of changes, we could identify that features

kept their characteristics in terms of how they are implemented, e.g., number of variation points and feature interactions, even if their implementation changed. By looking to change characteristics of nested features, i.e., features that have non-top-level variation points, we identified features being indirectly affected by changes performed in outermost variation points of other features. Therefore, mining change characteristics of a feature with our support shows when changes of a commit resulted in new variation points in additional files, or if part of the source code of a feature was removed or added in nested variation points. Answering our RQ2, commits adding, changing, or removing lines of source code is mostly related to only one feature, but sometimes it indirectly impacts the outermost variation points, namely top-level branches and dependencies to and interactions with other features. Regarding our RQ3, about the complexity of features implementation, we concluded that often features implementation become more complex over time. We observed that systems with complex implementations of their features (scattering degree ≥ 7), have their features more often revised over time. Further, our analysis shows that features scattered over many variation points tend to lead to more revisions over time than features with complex tangling, i.e., high number of feature interactions, and non-top-level branch implementation.

For RQ4, the answer was obtained by evaluations presented in Papers B, C, and D. We evaluated the quality and runtime performance of our proposed supports for mining, locating and reusing feature revisions. Our support for mining feature revisions, feature revision change analysis and propagation between releases of a preprocessor-based SPL can aid maintenance and reuse tasks at the feature level. The subject systems of our study had a high number of files and patches of code to analyze per random pair of releases, as an example, one system had on average 19 files affected per feature revision propagated and another system had up to 11,454 patches of code to be propagated between two releases. We conjecture that manually analyzing the differences between two releases and finding out feature interactions, files, and patches of code for propagating a specific feature revision is highly complex and error prone. Thus, retrieving information similar to Git VCS but at the feature level to find which files and lines of source code are differing facilitates the propagation of feature revisions and ease the reuse of feature revisions implementation. In summary, our support for mining feature revisions and changes between releases of preprocessor-based SPLs can speed up the process of feature revision change analysis and propagation. The reuse of feature revisions requires retrieving the patches of code differing between source code files of two releases as well as the feature interactions, and this is provided by our novel support in around 60 seconds.

Regarding our support for feature revision location, it as novel technique, which locate features at multiple points in time. Thus, this support addresses the limitation of existing feature location techniques of only locating features at a certain point in time. Our support for feature revision location allows to reason about feature revisions in a reasonable time, ranging between 25 and 250 seconds to map artifacts to feature revisions for each input product/variant. Our support for locating feature revisions enabled us to create new products by reusing feature revisions within 18 seconds, on average. Even in some cases manual completion can be necessary, it probably will not require extensive code additions or deletions by a developer. The completion of a product/variant can be facilitated by hints provided by our support, which presents feature interactions and possibly missing or surplus feature revisions for a specific product configuration. We show that our automated support for locating and reusing feature revisions to compose new products aids to save time and effort in maintenance and reuse of feature revisions in families of software systems evolving in space and time. Therefore, locating feature revisions supports the domain implementation and product derivation by reusing and combining artifacts that correspond to located feature revisions.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the contributions of this thesis for maintenance and reuse of families of software systems evolving in space and time, and give future research directions in this field.

6.1 Conclusion

This thesis investigated how families of software systems evolve in space and time and how to assist developers when maintaining such systems. We first conducted an analysis of the features' life cycle (Paper A). The analysis consists of how features are introduced, removed/deleted, and revised over the system development. For instance, how are the feature characteristics of preprocessor-based SPLs in terms of the number of variation points, i.e., number of `#ifdefs`, and in how many files the variation points are distributed and how many features are involved in their condition expressions. This analysis was performed over all commits of the systems analyzed, which enabled us to know how these characteristics have been evolved over time. Our findings resulted from this analysis showed that our target systems have similar evolution in space (3%-7%) when compared to the large Linux kernel system (6%) [152], which has a higher number of commits, features, and lines of source code. Still comparing our subject systems to the Linux kernel, our results confirmed that features are revised significantly less (2% to 11% of the systems' commits) than the core of the system is [83]. However, regarding the scattering degree of features implementation in our systems, we observed that features were introduced already scattered and did not remain constant over releases as shown in the evolution of some releases of the Linux kernel. Contrary to what happens to the features of the Linux kernel, in the preprocessor-based SPLs of our empirical analysis the complexity of feature implementation increases over time, which reinforces the need of support to maintain and reuse feature revisions in preprocessor-based SPLs. Therefore, in addition to the empirical analysis, we contributed with an automated support that can be used by developers and practitioners to understand and track the feature evolution of other preprocessor-based SPLs. The support can be further improved for other purposes, such as change analysis and propagation of feature revisions in preprocessor-based SPLs, as we presented in Paper B.

Throughout this thesis work, we also investigated that propagating feature revisions in the most popular mechanisms used in practice, i.e., preprocessor-based SPLs managed in VCSs, is a manual laborious and challenging task. The current VCSs do not support the analysis and propagation of revisions at the feature level. Thus, we presented a support to address such limitations and aid developers to analyze and propagating different implementations of the same feature among releases that can be in different branches (Paper B). From the practical point of view, the way our support presents information at the feature level makes easier the maintenance and evolution of the `"#ifdef hell"` [36, 49, 118]. The feature level is of paramount importance because manually recovering feature information is very

difficult, and even more when multiple files change in a commit or a sequence of commits, touching many blocks of code and involving multiple features and interactions. While VCS tracks changes and recovers information either per file or for the whole system, our tool support can recover information per feature. Thus, it can be used as complementary tool for maintenance and reuse of feature revisions in preprocessor-based SPLs managed in VCSs.

Despite the vast majority of SPLs being implemented with preprocessor directives in VCSs, there are still systems developed without a common platform. Aiming to help these systems without a common platform to deal with evolution in space and time, we also present, to the best of our knowledge, the first feature revision location technique in the literature. Our feature revision location addresses the limitation of locating features at only a single point in time (Paper C). The proposed technique allows developers to reason about variants and features at different points in time. Further, our automated technique assists developers to keep a history of modifications in features by mapping artifacts to feature revisions. The feature revision location technique can be applied to any type of artifact if the input information is compatible with its internal data structure.

Based on the acquired knowledge of the empirical study, we proposed a support that not only aids the analysis and maintenance of families of software systems evolving in space and time by locating feature revisions, but also allows the composition of new products by reusing the feature revisions located. Therefore, we presented *ECSEST* (Extraction and Composition for Systems Evolving in Space and Time) in Paper D, which is an extension of the work presented in Paper C. As further contributions of this thesis, we also presented and discussed four challenges for evolution in space and time concerning feature location and composition of new products with features and feature revisions (Paper E). In addition to the challenges, we presented a benchmark and a ground truth extractor to cover the space dimension and time dimension of software system evolution. The benchmark counts with scenarios, dataset, and tool utilities to compute metrics. By proposing a benchmark, we expect that future work related to these challenges can be evaluated with common case studies enabling reproducibility, comparison, and optimization of limitation aspects of current techniques.

6.2 Future Work

As preprocessor-based SPLs managed in VCSs are the most used mechanism to deal with evolution in space and time, it is important to provide solutions for improving their limitations. A long-term future work would be to provide a native and mature solution that is self-sufficient to provide adequate support for evolution in space and time. Short-term future work can be directed to improve our support for feature revision propagation presented in Paper B to recommend which releases might need propagation of a revision of a feature. Hence, this can save time and effort for developers to decide which features of which releases must be propagated, for example, for bug fixes or refactoring. Another future direction, is the semantic impact analysis of merging the source code of a specific feature revision to different releases. This may help to prevent possible errors and bugs that can occur from, for example, features source code containing function calls of functions not yet implemented in relation to another release.

Regarding *ECSEST* support, future improvements can be directed to optimize the runtime performance of the support for mapping and reusing feature revisions. Currently, as high is the number of feature revisions and artifacts mapped, higher is the time needed to map new feature revisions, and reuse feature revisions artifacts to compose new products. The composition of products by reusing feature revisions requires first the extraction process performed, i.e., the mapping between feature revisions and artifacts, which could be improved. Currently, the extraction process is incrementally performed for every product

containing new feature revision(s), as the process should create new mappings and update existing ones. Further efforts can be directed to improve *ECSEST* capability of composing products with feature revisions by taking into account a variability model showing the problem space of valid configurations of feature revisions. Existing studies investigating how to propose models presenting valid configurations based on the feature evolution could be integrated with *ECSEST* [48, 71].

Part II

Papers

Paper A

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

Published in the Proceedings of the 20th International Conference on Generative Programming: Concepts and Experiences (GPCE 2021), 14 pages, Chicago, IL, USA, 2021.

Authors Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher and Alexander Egyed.

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.

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 [75]. 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) [160]. Further, preprocessor directives enable alternating implementations and supporting multiple platforms and operating systems across different environments [119]. 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 [96, 119]. 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 [21, 131, 189].

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 [67, 104]. 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 [110]. Existing studies thus recommend research on recovering and tracking the evolution of features managed in version control systems (VCSs) [85, 119].

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 [19]. It also considers corner cases [110] for mining variability in space and time. Other approaches only capture features annotated in presence conditions [39, 74, 96, 160] 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 [39, 74, 96, 119, 147, 191].

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 [45, 160] 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].

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 [17]. 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 [110]. 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* [160]. 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 [110]. 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 [33]) 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 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.

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

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

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

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 [33] (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 [158], 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 [132], which reduces the computational costs of the analysis [80].

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 [33], 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 the differences between two text files in a line-oriented way as computed by a diff utils library [11]. 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 [158]. 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.

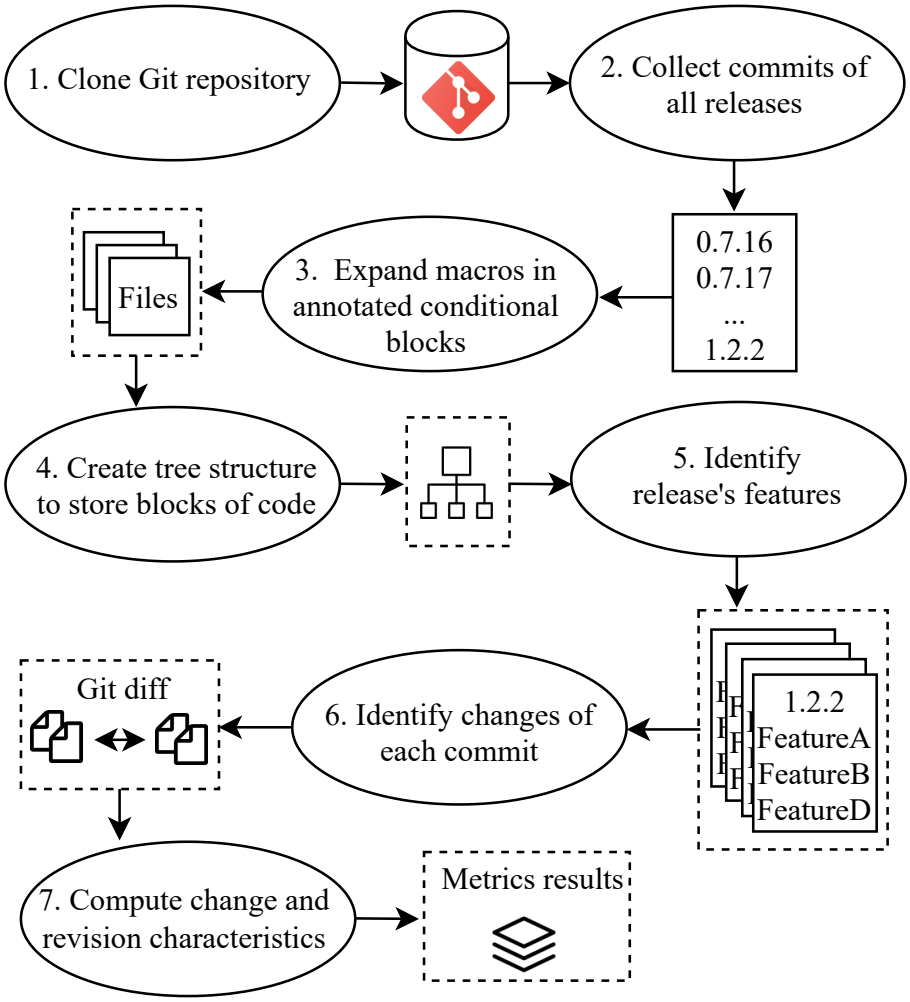


Figure 1: Automated approach for mining the life cycle of features.

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 [50, 160] 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 [7, 96].

Let us use an example containing corner cases to illustrate how we compute these metrics and why a CSP solver is needed. Listing A.1 shows a code snippet in the first commit (Commit #0) and Listing A.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 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 $\texttt{Y} = T$, 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).

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

```
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 A.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     <changed code>
12 #endif
13
14 #ifndef A
15     <code>
16 #else
17      <changed code>
18 #endif
```

Listing A.2. Commit #1.

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 A.1) when computing the metrics, which is incorrect [110]. 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 [110]. 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 [57, 64, 96, 118, 191]. 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 78^{th} release. When analyzing the removed and added features in

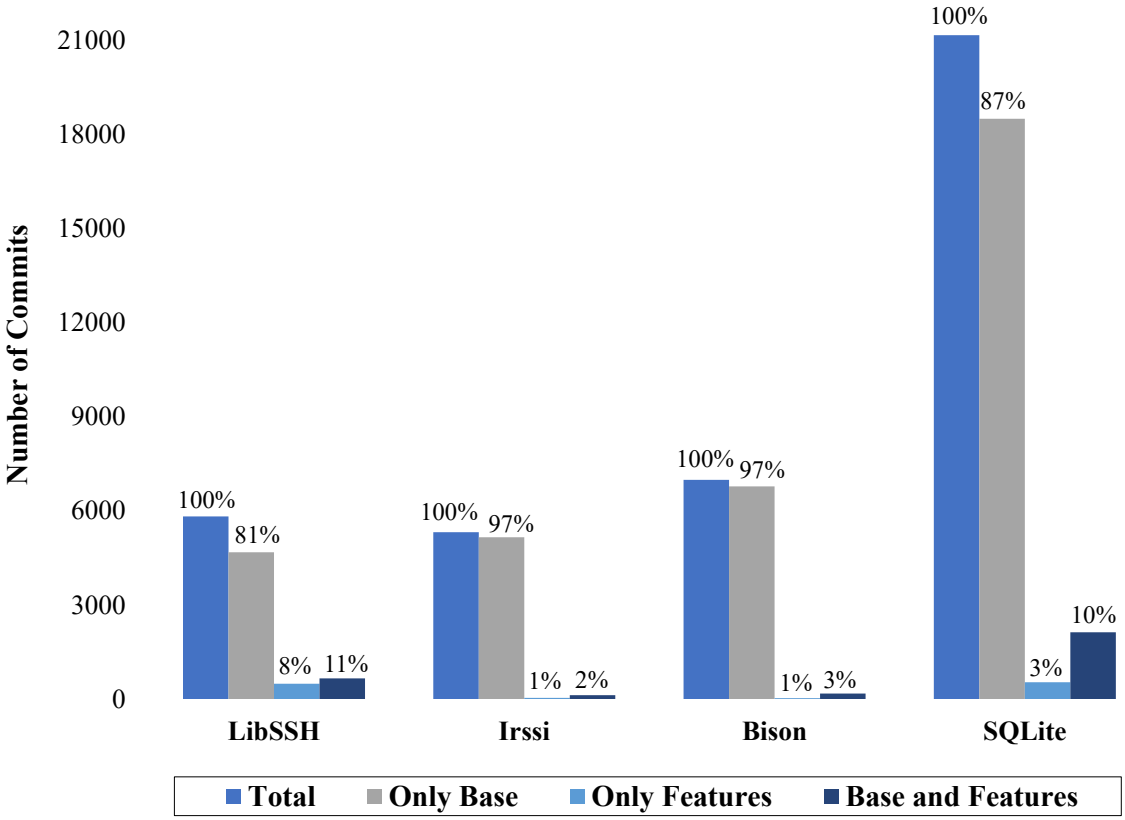


Figure 2: Number of commits based on systems' changes to feature types.

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. [22], 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 [152] 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. [82] 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 (5%) of Bison. These numbers confirm that changes over commits are tangled [67] 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 [82]. 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 [160] 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 [160]) 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

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	HAVE_CAPSICUM	HAVE_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**.

feature impact complexity in terms of SD #IFs, SD File, and TD. We applied the Spearman correlation coefficient [182], 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 [47]. 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. 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*

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

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 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 [160] 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 [149, 152], 20% of the features have high scattering, where 75% of the scattered 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

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%

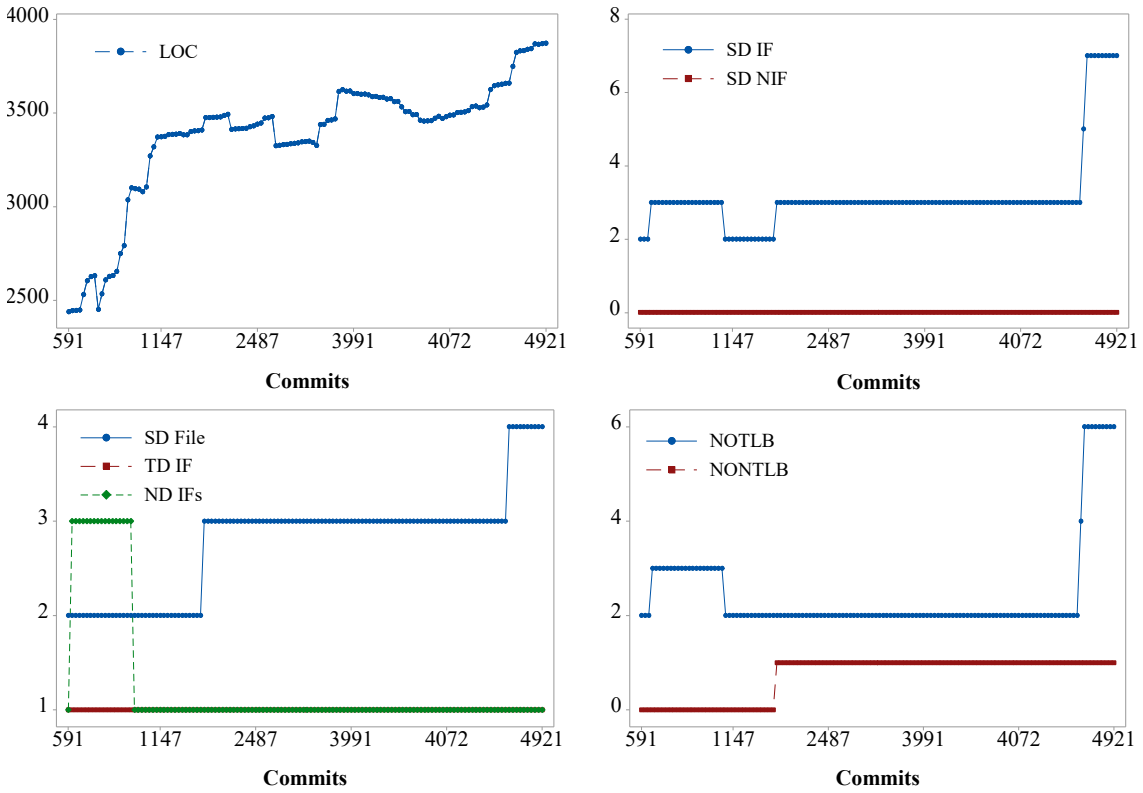
multiple times to fix bugs such as memory leaks or security vulnerabilities. Other examples are debugging and testing features that often change, adding and removing variation points for debugging problems, and testing new features and bug fixes.

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 [160]. We now present our fourth finding:

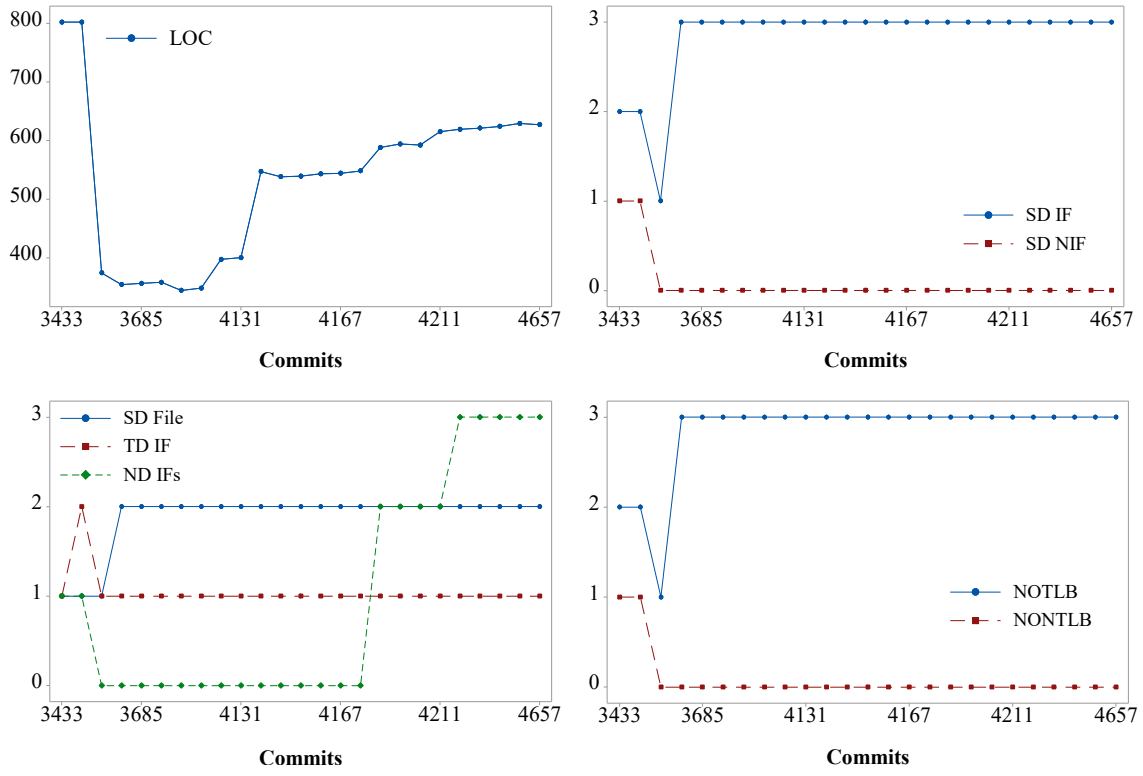
Finding 4. *Feature evolution often increases implementation complexity. We observed that systems with complex implementations of their features, i.e., scattering degree ≥ 7, have more feature revisions. Interestingly, features scattered over many variation points tend to lead to more revisions over 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 3a and 3b shows the characteristics of the hotspot feature over time for the LibSSH and Irssi systems. The LOCs of the LibSSH hotspot feature (Figure 3a) increase over time as do the 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 3b) 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



(a) LibSSH feature `WITH_SFPT`.



(b) Irssi feature `HAVE_OPENSSL`.

Figure 3: The hotspot features' evolution along their life cycle.

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.*

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

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

affected configurations when selecting these features. As pointed out by Mühlbauer et al. [139], 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 [144] because according to Henry and Ridene [66], 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.

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

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 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 44^{th} 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 [57, 64, 96, 118, 191]. Furthermore, according to Liebig et al. [96], 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. [22] features are also used for, e.g., testing, debugging and deployment. Further, previous empirical analyses [74, 96] 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 [45, 74, 96, 110, 159, 160].

Conclusion validity: Regarding the statistical analysis, we conducted a Shapiro-Wilk test [176] 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 [142, 145, 146] 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 [96, 118, 160, 191], or up to 500 commits of one specific release [95, 147]. 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. [96] 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. [110] 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. [151, 152] 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 [149], Passos et al. extended previous work [148] 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. [39] also improve Passos et al. [151] approach: FEVER automatically extracts details on changes in variability models (KConfig files), preprocessor-based source code, and mappings (Makefiles). Other studies [60, 82, 168] 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 [44] 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. [91] 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. [180] 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. [74]. 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. [186].

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 [3, 21, 125, 134]. 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 [186]. 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.

Paper B

Propagating Feature Revisions in Preprocessor-based Software Product Lines

Submitted to the Conference (Omitted for double-blind review process), 2022.

Authors Gabriela K. Michelon, Wesley K. G. Assunção, Paul Grünbacher and Alexander Egyed.

Abstract Preprocessor-based software product lines (SPLs) are used in practice to derive customized variants of a system. SPLs enable practitioners to deal with evolution in space, in which features (so-called configuration options)—annotated in source code with #ifdefs—are included, removed, and systematically reused. Inevitably, feature implementations also evolve over time, i.e., when existing features are revised. Nowadays, version control systems (VCSs) are well-integrated into SPL development processes for versioning support of the whole platform. However, evolving an SPL is complex, since developers need to work on all variants at the same time. Changes to existing features in one version, a.k.a. release of an SPL, usually developed in a branch, frequently need to be propagated to other releases. However, there is no automated support for versioning and propagating features in SPL releases managed in VCSs. For instance, the VCS can only propagate changes at the commit level. Manually mining and propagating versions of a feature, i.e., feature revisions, through #ifdefs is risky, time consuming, and error prone. In this paper, we thus present a novel approach for feature revision change analysis and propagation without changing any aspect of the traditional development of preprocessor-based SPLs managed in VCSs. The results show our approach successfully propagates 3,134 features in space and time with precision and recall of 99%, on average. These propagations involve 237,854 patches of code within 14,244 source code files in 200 pairs of releases of four real-world preprocessor-based SPLs.

1 Introduction

Preprocessor-based SPLs provide a reuse-oriented platform with common and variable code from which multiple software product variants are derived [75]. When implementing such systems, preprocessor directives have to be employed to define variation points specifying the parts of the SPL that will differ among different variants [23]. The preprocessor directives delineate features, which represent end-user functionality or are used for development purposes, such as testing, debugging, and deployment [22, 86]. Thus, preprocessor directives allow deriving variants by selecting features that address different requirements [143].

Preprocessor directives are widely used for many reasons [160], among them supporting multiple platforms and operating systems across different environments [119]. Portability problems are overcome by setting values to macros or features using preprocessor directives that enable a specific operating system or compiler. Preprocessors handle variability by guarding code fragments with preprocessor directives, like `#ifdef`, `#if`, or `#elif`, which support the definition of optional features and alternate implementations [160]. Currently, `#ifdef` directives are the most common variability mechanism to deal with *evolution over space*, i.e., introducing or removing features of a system [21].

In addition to the evolution of systems in space, feature revisions are unavoidable due to modifications of the systems' implementation, which is known as *evolution in time* [21]. Feature revisions happen because of bug fixes, refactoring, and enhancements [132]. Currently, the most common mechanism to manage evolution of systems over time are Version Control Systems (VCSs) [134], e.g., Git. SPLs implemented with preprocessor directives integrate very well with VCSs, as the preprocessor directives are basically pieces of code [21]. Hence, the preprocessor variability mechanism in combination with VCSs is widely adopted in practice to manage the development of variants (concurrent versions) and revisions (sequential versions) of an SPL [127, 134].

However, practitioners and researchers criticized the use of preprocessor directives claiming that they make source code harder to understand and maintain due to limited separation of concerns/features, error proneness, and code obfuscation [49, 118, 119, 127]. Further, recovering feature information is very difficult when multiple files change in a commit, touching many blocks of code and involving multiple features [17, 36, 76, 127, 195]. VCSs can track changes and recover information either per file or for the whole system, however, they do not recognize variation points, hindering feature analysis. Consequently, when developers need to investigate a problem happening in some variants of a preprocessor-based SPL, they have to manually analyze the preprocessor annotations within multiple files of the whole platform to understand which features are actually affected by a change [127].

Furthermore, a revision of a specific feature on a specific release of an SPL, maintained in a branch of the VCS, can be necessary to be propagated to other releases. However, with current VCS support such as Git, only commit-level changes can be propagated [28]. Yet, cherry-picking a commit is a proper solution for a few scenarios, i.e., when developers identify a pre-existing bug during the development of a release, and then create an explicit individual commit patching this bug [16]. Nonetheless, propagating an entire feature revision implementation in space and time between releases with different files and features requires laborious analysis of feature interactions to know which files and patches of code, i.e., sequences of lines are differing for a feature in different releases. A recent study confirms that feature propagation is very challenging and expensive in preprocessor-based SPLs managed in VCSs, and there is no automated support for propagating feature revisions in such reuse-oriented platform [86]. Furthermore, preprocessor-based SPLs and VCSs mechanisms do not analyse and provide visualization of which features are interacting and affected between different releases [127]. These limitations and challenges of change propagation during evolution in space and time confirm the need for a novel and automated solution [127, 131].

Recently, Michelon et al. presented a mining approach for tracking feature revisions [127] in preprocessor-based SPLs in VCSs and a feature revision location technique to map artifacts from existing variants of a system that evolved over time to feature revisions [134]. The feature revision location technique is presented to facilitate the management of system variability in space and time by the possibility of composing variants with feature revisions. However, both approaches for mining and locating feature revisions do not present automated support for propagating feature revisions in different releases of preprocessor-based SPLs in VCSs.

In this work, we present an approach for propagating feature revisions from one release to another, which can exist in different branches. Our goal is to ease the reuse of features between releases that are evolving in space and time by an automated support for analyzing and propagating feature revisions of preprocessor-based SPLs in VCSs. We evaluated our approach in a large dataset. The results show that our approach can automate inspection of `#ifdefs` in multiple files to find the affected files, patches of code, features, and interactions between releases of a system, and propagate features in space and time.

In summary, we make the following contributions: (i) an approach for change analysis and propagation at the feature level in preprocessor-based SPLs managed in VCSs; (ii) a non-intrusive tool [5] that can be integrated in the existing development of preprocessor-based SPLs in VCSs to support developers in propagating features in space and time; (iii) a dataset containing information of the 3,134 features propagated in space and time [6] in 200 target releases from four real-world preprocessor-based SPLs for replications and future work.

2 Motivation

SPL commonly evolves in space and time when features are introduced, deleted, and revised, resulting in different releases of a system with a different set of features and some features in common but with different implementations [127]. On top of that, the evolution over time of preprocessor-based SPLs in VCSs is tracked for the whole platform at coarse granularity [21, 104] and there are several studies acknowledging for the importance of tracking implementation and changes of feature revisions [2, 71, 131, 132, 134].

Analyzing the evolution of features in real-world preprocessor-based SPLs in VCSs, we found out that a feature revision existing in a specific release may be needed to be propagated in releases where it does not exist. As an example, let's use the Marlin SPL, which is an open-source firmware for 3D printers. In a pull request from Marlin[1] users were claiming that the feature `BLTouch V3.0` in release 2.0.x is not in 1.1.x, and thus nobody buying a new BL-Touch, i.e., V3.0 would be able to use the release v1.1.9. In this pull request, 114 files were changed within 3,208 lines added and 911 deleted involving around 1,100 preprocessor directives. An approach to automate the propagation, i.e., reuse of the implementation of the `BLTouch V3.0` feature revision to release v1.1.9 could ease this propagation and give the users the freedom to buy the new version of BL-Touch to use in their printers with release 1.1.x.

Another example of the utility of an automated approach for propagating feature revisions can be seen in the SQLite preprocessor-based SPL, where a feature for testing purposes is propagated over four releases[2]. This example shows that developers could also get benefit of such approach for automatically propagating feature revisions for purposes of testing or debugging the evolution of releases of an SPL. Having to manually analyze in which files and lines of code a feature of a preprocessor-based SPL is implemented and considering which features are interacting with it is a laborious, tough, and highly challenging task.

[1] https://github.com/MarlinFirmware/Marlin/pull/14839
[2] https://www.sqlite.org/src/info/7b4583f932ff0933

Therefore, we aimed to propose an approach thinking in terms of feature revisions to ease the reuse of feature (revision) implementation of preprocessor-based SPLs in VCSs. Next, we present our automated support for feature revision propagation and change analysis. The feature revision change analysis is a necessary preliminary step to feature revision propagation for retrieving relevant information such as which files and lines of code and feature interactions are affected in a release when propagating feature revisions to it.

3 Approach

Our approach comprises change analysis and propagation of feature revisions and is illustrated in Figure 1. It enables the propagation of patches of code of a specific feature revision between different releases. Before performing the feature revision change analysis and propagation, our approach needs as input the set of features existing in each release, as well as the features to be propagated between one release to another, and the snapshot of each release. Our approach has an optional step (*mining releases features*), which is based on the existing mining approach from Michelon et al. [132] for mining the set of feature revisions existing in a release in case the developer/user does not know. The *mining releases features* step retrieves for each conditional block of code, i.e., `#ifdef`, which feature(s) belong to it. Following, the second and third steps are novel and the core of our approach. The second step, *feature revision change analysis*, is necessary to know for each delta, i.e., differences in the source code between one release to another, which change belongs to which feature revision(s) to be propagated. Then, the *feature revision propagation* step is performed after knowing which files, patches, and feature interactions will be affected by propagating a feature revision. The result of this step is then the snapshot of a destination release containing the feature revision(s) propagated. We describe these steps in detail next.

3.1 Mining Releases Features (Optional Step)

The mining approach presented by Michelon et al. [132], retrieves which features were revised in which commits. For every patch of code that differs between a specific commit to

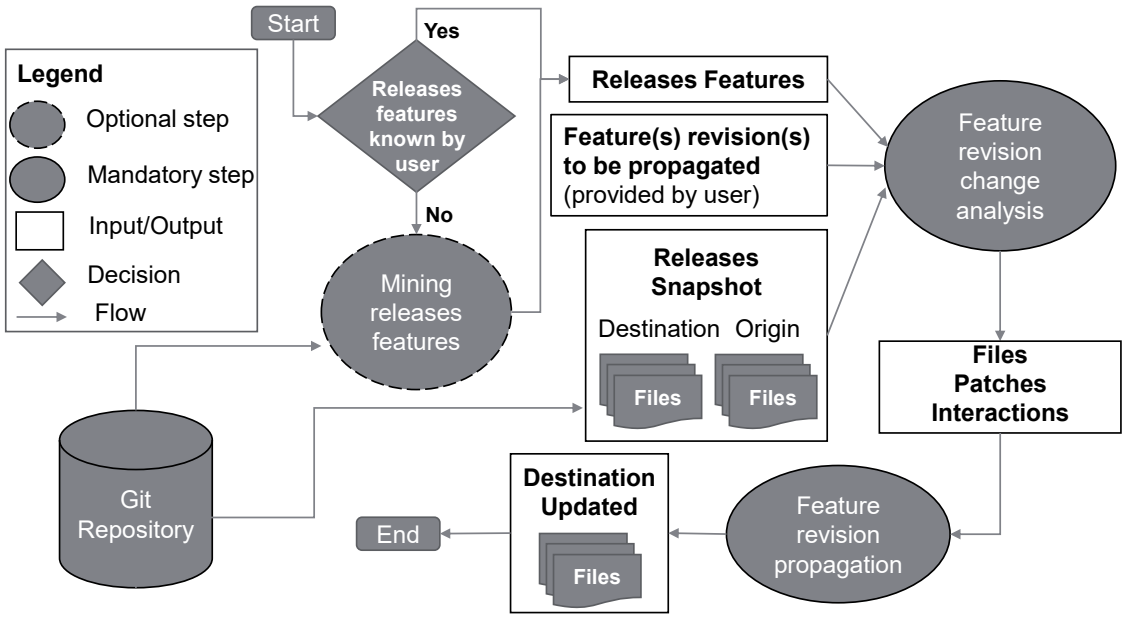


Figure 1: Approach overview for change analysis and propagation of feature revisions.

another, called by them as “changed block", the approach builds constraints including the `#ifdef` and `#define` preprocessor directives responsible for activating the changed block. A macro defined via a `#define` preprocessor directive that is wrapping a patch of code via `#ifdef`, for example, cannot be considered a feature revision as it cannot be selected directly by users. However, macros defined within the source code may influence in activating a changed block. Thus, all macros (feature revisions or not) are taken into account to build constraints influencing in activating a changed block. These constraints then are handed to a constraint satisfaction problem (CSP) solver [158], which finds a solution that contains which features must be selected in the corresponding SPL to activate a particular changed block, including all features and their interactions. The solver is necessary to automatically and reliably identify features interacting or depending on the execution of other features [19]. That is one of the reasons why we decided to build our approach for feature revision change analysis and propagation based on Michelon et al. [132].

To better illustrate the need for a CSP solver, let us use a code snippet adapted from LibSSH, as shown in Figure 2. There are four different macros `WITH_SERVER`, `MD5_DIGEST_LEN`, `__cplusplus` and `_LIBSSH_H`. In this example, the macro `MD5_DIGEST_LEN` cannot be selected by the user and is not a feature revision. `MD5_DIGEST_LEN` is defined in a block of code corresponding to a conditional expression of feature `WITH_SERVER` (Lines 1-4). This means, the macro `MD5_DIGEST_LEN` is defined when the feature `WITH_SERVER` is selected by the user. To mine the feature revision(s) of the patch of code in Line 8 of Figure 2, the following constraint is built and enable us to know the feature interactions: (`WITH_SERVER` $\implies$ `MD5_DIGEST_LEN = 16`) $\wedge$ `__cplusplus` $\wedge$ `_LIBSSH_H` $\wedge$ (`MD5_DIGEST_LEN > 5`) $\wedge$ `BASE`. The solution retrieved is then a configuration able to execute the patch of code in line 8: `WITH_SERVER = TRUE` $\wedge$ `__cplusplus = TRUE` $\wedge$ `_LIBSSH_H = TRUE`. The set of features responsible for activating the patch of code in Line 8 are `WITH_SERVER`, `__cplusplus` and `_LIBSSH_H`. Now that the features and interactions are obtained, the feature revised is computed according to the heuristic from Michelon et al. [132]. A feature is *revised* if it already has blocks of code before the commit changes and if its blocks of source code have been changed. The feature is *deleted* when no blocks of code, i.e., conditional blocks involving the feature in `#ifdefs` exist anymore after the commit changes. A feature is *introduced* when the commit changes contains at least one conditional block involving the feature in `#ifdefs`. The heuristic takes all the features retrieved in a configuration and assigns as the features revised the closest features to the changed block, which are the ones directly impacted. In our illustrative example the feature that is revised is the feature `_LIBSSH_H`, which interacts with `WITH_SERVER` and `__cplusplus`.

3.2 Feature Revision Change Analysis

This analysis starts with the computation of the differences between two arbitrary releases' commits, referred to as *origin* O and *destination* D. The commit O contains the snapshot

```
1 #if WITH_SERVER
2       <code>
3       #define MD5_DIGEST_LEN 16
4 #endif
5
6 #if __cplusplus
7       #if _LIBSSH_H && MD5_DIGEST_LEN > 5
8           <code>
9       #endif
10 #endif
```

Figure 2: Code snippet adapted from LibSSH.

of a point in time of the source code containing the feature revision to be propagated and D contains the snapshot of a point in time where the feature revision with the implementation of the commit O should be propagated. To compute the differences, firstly, the files of the two commits O and D are obtained. Secondly, the approach identifies the changes between the commits, which are the differences to be possibly propagated from O to D. A *file change* FC corresponds to a file that can be either deleted, inserted, or modified. Thirdly, the approach maps to a FC the numbers of the first and last lines removed or added for each of the patches of code that differs between the commits O and D. Each FC is described next:

DELETE. In this file change, a file existing in commit D does not exist in O. The reference of this change type in FC contains the number 1 to represent the first line, and the total number of lines of the deleted file (n) to represent the last line of the deleted file. Then, FC receives the flag "`DELETE`".

INSERT. In this file change, there is a file in commit O that does not exist in D. The reference of this file change in FC contains the number 1 to represent the first line, and the total number of lines of the inserted file (n) to represent the last line of the inserted file. Then, FC receives the flag "`INSERT`".

MODIFY. In this file change, a file existing in commit O also exists in D, but in a different state. A modified file may be affected by many deltas. Thus, to assign the reference to each of the deltas in FC, our approach uses the patches of code contained in the file's deltas. Delta is a way of storing or transmitting data in the form of differences between sequential lines rather than complete files. For each delta, our approach obtains its type (`REMOVE`, `ADD`, or `CHANGE`). Examples are presented in Figure 3. An `ADD` delta (Line 2-2 in O, Figure 3a) contains lines added in O in comparison to D, referencing the line number of the beginning and end of the lines added in the `ADD` delta. A `REMOVE` delta contains lines removed from O to D, referencing the line number of the beginning and end of the removal (Line 2-2 in D, Figure 3b).

Finally, a `CHANGE` delta (Figure 3c, Line 2-3 and Figure 3d, Line 2-4) contains lines changed, consecutive removals and additions, from O to D. The file change thus contains not only one, but two references. The first reference contains the beginning and end of the addition delta lines (Lines 2-2 in O, Figure 3c, and Lines 2-3 in O, Figure 3d), while the second reference contains the beginning and end of the removal delta lines (Lines 2-2 in D, Figure 3c, and Lines 2-2 in D, Figure 3d).

Next, our approach gets the conditional blocks of code corresponding to added, removed,

```
1|1   #if WITH_SERVER
  2 +       <new code>
2|3         <code>
3|4         #define MD5_DIGEST_LEN 16
4|5   #endif
```

(a) `ADD` delta.

```
1|1   #if WITH_SERVER
2|  -       <code>
3|2         #define MD5_DIGEST_LEN 16
4|3   #endif
```

(b) `REMOVE` delta.

```
1|1   #if WITH_SERVER
2|  -       <code>
 |2 +       <new code>
3|3         #define MD5_DIGEST_LEN 16
4|4   #endif
```

(c) `CHANGE` delta.

```
1|1   #if WITH_SERVER
 |2 +       <new code>
 |3 +       <new code>
2|  -       <code>
3|4         #define MD5_DIGEST_LEN 16
4|5   #endif
```

(d) `CHANGE` delta.

Figure 3: Examples of delta types of a modified file (FC `MODIFY`) resulting in new revisions of the feature `WITH_SERVER`.

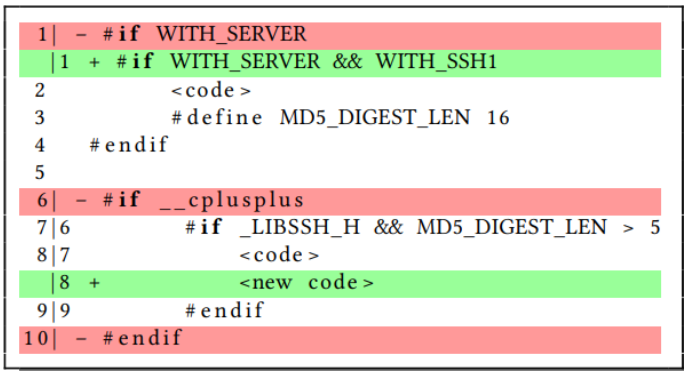


Figure 4: Examples of deltas resulting in one feature introduced (`WITH_SSH1`), one feature revised (_LIBSSH_H) and one feature deleted (`__cplusplus`).

and changed deltas respective to the feature(s) to be propagated. Thus, the output of the feature revision change analysis contains the conditional blocks of code with deltas respective to the feature(s) that can be propagated. For each delta, the feature revision change analysis also describes the feature interactions and the features that might be affected, i.e., nested features in conditional blocks of code, which is very important [96, 110, 160] and part of our novel approach. A feature interaction refers to all the features that are in the set of revised features of a conditional block of code, i.e., the features of the configuration necessary to activate the block of code related to the changes. For instance, using our running example presented in Figure 2, we presented three deltas modifying the code snippet adapted from LibSSH in Figure 4: The first delta (Line 1, Figure 4) is a `CHANGE` delta, where Line 1 in D is removed, which contains a conditional expression (`#if WITH_SERVER`) and a new line added (Line 1 in O, Figure 4) to substitute the conditional expression by another (`#if WITH_SERVER && WITH_SSH1`). The feature revision change analysis now is in charge of finding which features were introduced, revised, and deleted. For this delta (Line 1 removed in D and Line 1 added in O, Figure 4), one feature was introduced, because after this change, the conditional block of code in Lines 1-4, Figure 2 has a new feature `WITH_SSH1` (assuming our system has only the features presented in Figure 2). The feature `WITH_SERVER` interacts with the feature `WITH_SSH1` for the changes regarding the conditional block of code in Lines 1-4, Figure 2. Thus, before propagating the implementation of the feature `WITH_SSH1`, the feature revision change analysis provides delta(s) of inserted, deleted, or modified files, lines added and/or removed, the features that were revised, introduced, and deleted, as well as feature interactions.

The `REMOVE` deltas in Line 6 and 10 in D (Figure 4) are deltas resulting in deleting the feature `__cplusplus`. In this example, propagating the deletion of the feature `__cplusplus` from the system has interaction with the code of the core of the system (`BASE`) and the feature `_LIBSSH_H` might be impacted, as it is nested with the conditional block of code respective to the `REMOVE` deltas in Line 6 and 10 in D. The `ADD` delta in Line 8 in O resulted in a revision of the feature `_LIBSSH_H`. If the feature deletion is propagated, this also means that the feature `_LIBSSH_H` does not depend anymore on the feature `__cplusplus`, but still interacts with `WITH_SERVER` because `WITH_SERVER` defines `MD5_DIGEST_LEN`.

3.3 Feature Revision Propagation

The feature revision propagation is based on the selection of deltas obtained in the *Feature Revision Change Analysis*. Our approach analyzes if the selected deltas are related to inserted, deleted, and modified files. Then, it uses the files from the respective snapshots of two releases (O and D), and a directory path to check out the files of D containing the implementation updated with feature revision(s) propagated.

To propagate a feature revision considering differences between O and D, our approach gets the files from the Git repository of O and D releases' snapshot. Propagating a feature revision implementation for an inserted file means copying the file from commit O to the resulting directory, as the file does not yet exist in commit D. Then, all files are copied from D to the resulting directory excluding the deleted and modified files of D. For modified files, the approach obtains the file from O as well as from D. Then, it creates a modified file line by line, using a counter starting from zero and ending with the same number of lines of the file of D. Before writing a line, our approach checks whether the line is in a position of a delta, where the line must be removed or a new line must be added. In case of a removed line, the approach does not add the respective line. In case of an added line, the approach adds the respective line from O to D before continuing with the next lines of the file of D. When more lines are added in sequence, then all lines are added before continuing with the next lines of the file of D.

To illustrate the feature revision propagation of a modified file, we will continue with our running example presented in Figure 2. To make it easier to understand we use the conditional block of code of feature revision `WITH_SERVER` (Lines 1-4, Figure 2). Figure 3 shows possible delta types in FC `MODIFY`: Figure 3a contains an `ADD` delta; Figure 3b contains a `REMOVE` delta; and Figures 3c and 3d contain a `CHANGE` delta, where a `CHANGE` delta can be one line/a sequence of lines removed followed by one line/a sequence of lines added (Figure 3c). A `CHANGE` delta can be either one line or a sequence of lines added followed by one or a sequence of lines removed (Figure 3d).

The `ADD` delta contains one reference with one line added (Line 2 in O, Figure 3a) before Line 2 in D, i.e., from the previous revision of the feature `WITH_SERVER` (Figure 2). To propagate a feature revision from the O snapshot to D snapshot, our approach retrieves each delta for each file containing a feature revision. For the example of an `ADD` delta, the approach adds the lines of the added lines reference in O, beginning in the line number of the corresponding file in D where the delta begins. For example, the lines in Figure 2 are copied to the new version of the D file until there is a line number corresponding to the `ADD` delta in Figure 3a. As there is no line removed, the next lines existing in D are copied right after the line added. As shown in Figure 2, in D the line number 2 was "<code>" and now this line is in the third line of the file in D because a new line has been added in the second line number of the D file. The `REMOVE` delta contains one reference with one line removed (Line 2 in D, Figure 3b), which was the Line 2 of previous revision of the feature `WITH_SERVER` (Figure 2). For the `REMOVE` delta, the same procedure happens as for the `ADD` delta. However, the `REMOVE` delta has a reference for removing the content of the line number 2 existing in D, and thus the line number 2 will have the content of the line number 3 in Figure 2.

The `CHANGE` delta contains two references: in the `CHANGE` delta of Figure 3c, the first reference contains line number where addition begins and where it ends (Line 2-2 added in O), and the second reference contains line numbers where removal begins and where it ends (Line 2-2 removed in D). In the example in Figure 3c, the line number 1 does not change in D, and the second line is removed. Then the line number 2 in O is copied to the line number 2 in D. The second `CHANGE` delta presented in Figure 3d) differs from the former `CHANGE` delta because it starts with a sequence of lines added and then a line is removed between O and D. The second `CHANGE` delta contains in its first reference Lines 2-3 (added in O). Its second reference contains the beginning and end of line(s) removed (Line 2-2 in O). Therefore, the line number 1 does not change in D, and the second line in D is substituted by the second line in O. Then, instead of copying the next line existing in D (Line 3 in Figure 2), it is necessary to first add all the lines existing in the sequence of lines added before continuing writing the next lines of the file in D. Note that file D will now have the content of Lines 3 and 4 in Lines 4 and 5 after the propagation of the `CHANGE` delta in Figure 3d.

4 Evaluation

This section presents the research question (RQ), the subject systems, the methodology, and the metrics used to evaluate our approach. The evaluation is based on propagating features in space and time from one release to any another arbitrary release, which can be from a newer release to an older release and vice versa, in four open-source preprocessor-based SPLs.

With the goal of evaluating our approach to automate the reuse of different features implementation involving evolution in space and time in releases of a preprocessor-based SPL, our study was guided by the following RQ:

RQ. *How effective is the proposed approach to propagate features and feature revisions in preprocessor-based SPLs in VCSs?* Our goal is to check the correct behaviour and runtime performance of our approach for feature revision change analysis and propagation. We check whether our approach propagates the implementation of different revisions of a feature existing in two releases. In addition, we check whether our approach introduces new features and removes features not in common between two releases in preprocessor-based SPLs in Git VCS. Further, we measure propagation characteristics to estimate how much effort our approach automates in terms of analysis and propagation of conditional blocks, files, and features when propagating an entire feature revision implementation. We are aware that our approach does not avoid further developer intervention but it can automate part of this laborious task of reusing parts of a feature implementation between two releases.

4.1 Subject Systems

Our study relies on four open-source preprocessor-based SPLs managed in VCSs (Table 1). We reduced bias by choosing different application domains. Furthermore, each system has a considerable history of development and use in research [57, 64, 96, 118, 127, 191]. The systems comprise ≈40,000-174,000 lines of source code, ≈5,000-20,000 Git commits, and ≈15-22 years of development. A criterion for including these subject systems is that they were subjects in related work of feature evolution in space and time with dataset available [127]. Based on the dataset available, we thus know which features exist in one release that do not exist in any another arbitrary release. Therefore, this information helped us to check our approach's correct behaviour for propagating the implementation of feature revisions in preprocessor-based SPLs in VCSs.

4.2 Methodology

Figure 5 illustrates how we conducted the evaluation of our approach for propagating feature revisions. For each subject system, we cloned their Git repository. After, to avoid bias we randomly chose 200 different pairs of releases to be destination and origin targets. We also carefully analyzed the possible random combination of releases to obtain a significant number of commits between the releases, which should have considerable changes and thus different feature revisions. As the total number of possible pair combinations is huge and unfeasible to compute, we determined the same number of combinations for all systems,

Table 1: Subject systems and their characteristics.

System	Domain	Releases	Since	Commits	LOC	Features
LibSSH	Network library	48	2005	5,022	110,590	121
SQLite	Database system	113	2000	20,090	173,714	384
Irssi	Chat client	69	2007	5,331	85,325	57
Bison	Parser	105	2002	6,991	39,904	83

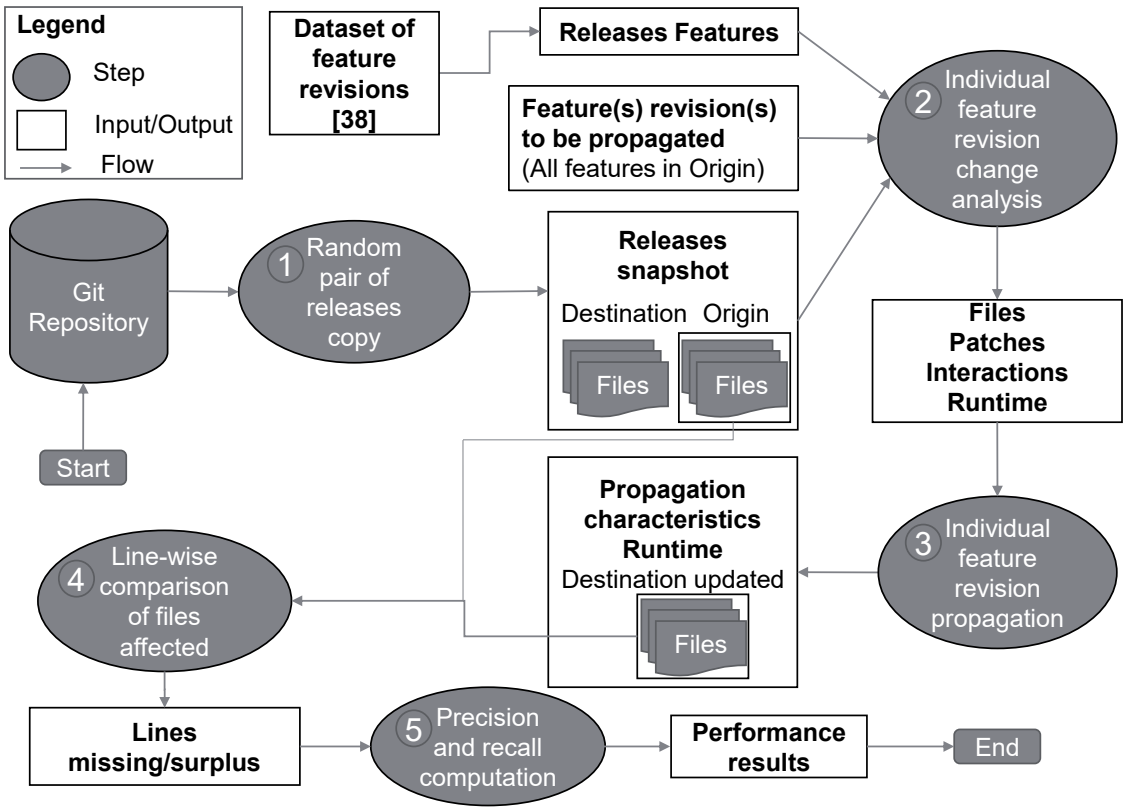


Figure 5: Evaluation methodology of our approach for feature revision propagation.

taking into account that it should not take more than 48 hours for computing all the propagations between the random pairs of releases.

For each destination and origin release, our approach copies their snapshot files (Step 1). The snapshot's files together with the set of features of each release and the features and feature revisions to be propagated are used as input for Step 2. The releases' features, as mentioned before, are available in the dataset published by related work [123]. The feature(s) revision(s) to be propagated are all the features existing in the origin release that not exist in the destination release. This means that features existing in both releases but with different implementations are the feature revisions to be propagated. The features existing in the origin release but not in the destination release are the features to be introduced in the destination release. Lastly, the features existing in destination release that do not exist in origin release are the features to be removed in destination release.

For each feature to be propagated, there is an output from Step 2, which contains *propagation characteristics* (Section 4.3). The files and lines affected per feature revision to be propagated are now used as input for Step 3. In Step 3, each feature revision is incrementally propagated, and the output of this step is the destination snapshot updated with the feature revisions propagated. The goal of propagating all the features from origin to destination is to be able to evaluate the correct behaviour for propagating the implementation of feature revisions. Therefore, after propagating all features from origin to destination we get an updated snapshot of the destination release. If our approach performed the feature revision change analysis and propagation correctly, the updated destination snapshot must be equal to or very similar to the content of origin snapshot. We also get the runtime computation of Step 3, as well as further propagation characteristics for each pair of release (Section 4.3).

Following, Step 4 consists of a line-wise comparison of files affected between the origin and destination releases with the feature revisions propagated. When there is a line or a

sequence of lines that is not supposed to be in the destination file using the corresponding origin file as baseline, this will result in false positive(s). False negative(s) occurs when a line or a sequence of lines is missing in the destination file using the corresponding origin file as baseline. Then, the false negatives (lines missing) and the false positives (lines surplus) are used as input for the last step (Step 5), which computes the remaining metrics used to answer our RQ.

4.3 Metrics

We evaluated our approach by analyzing propagation characteristics, checking the correct behaviour, and computing runtime.

Propagation characteristics. The propagation characteristics were measured by the number of features (features revised, introduced and deleted), files, patches and lines affected. Further, propagation characteristics include for each feature (revision) propagated the differences between destination and origin releases. The differences include which files are modified, deleted and inserted, as well as the patches of code. For each feature revision, we also compute the total number of conditional blocks of code, i.e., `#ifdefs` that has a delta between the pair of releases. For each delta, i.e., patch of code added, changed or removed inside a conditional block of code of a feature, there might be feature interactions, which are also computed and retrieved in the output of this step. The total number of conditional blocks of code and feature interactions are part of the output (*propagation characteristics*) and are used to quantify the change characteristics complexity (cf. [127]).

In order to check whether a system implementation includes a complex usage of `#ifdefs` Queiroz et al. [160] proposed thresholds to the use of feature annotations. There is a threshold for the number of conditional blocks of code of a feature, where, at least, 85% of the features of a system should have a number of `#ifdefs` ≤ 6, and as assumed by Michelon et al. [127], should be spread in ≤ 6 files. Another threshold presented by Queiroz et al [160], is that at least 80% of the features of a system should interact with one feature besides the core of a system. In order to estimate the complexity and effort to perform manual feature revision change analysis and propagation, we set up these thresholds as limits in relation to deltas of a feature (revision) between two releases.

The propagation characteristics metrics can be used to interpret, for example, why different pair of releases can take longer to perform feature revision change analysis and propagation by our approach, and also quantify how much manual effort would be necessary without our tooling support. These metrics are used in this work as a way to quantify how challenging can be to manually analyze source code differences regarding conditional blocks of code and propagating the source code among multiple files taking into account feature interactions using our dataset as baseline.

Correct behaviour. For checking the correct behaviour of our approach we computed precision (Equation B.1), recall (Equation B.2), and F1-score (Equation B.3) metrics [190], where the true positives (TP) are the lines of source code from the destination release updated with the propagations that match with the origin release. The false positives (FP) are the lines of source code that are surplus in matched patches of code between destination release updated with the propagations and origin release. The false negatives (FN) are the lines of source code that are missing in the matched patches of code between destination release updated with the propagations and origin release.

$$Precision = \frac{TP}{TP + FP} \tag{B.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{B.2}$$

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{B.3}$$

Runtime. We measured the runtime performance in seconds for completing the feature revision change analysis and feature revision propagation. To run the experiments, we used a laptop with an Intel® Core™ i7-8650U processor (1.9GHz, 4 cores), 16GB of RAM, and Windows 10.

5 Results

Propagation Characteristics. Table 2 shows the percentage of features containing more than six conditional blocks and files of source code affected, and more than two features interacting with a feature propagated from one release to another. According to Queiroz et al.'s [160] threshold of complexity implementation, we can see in Table 2 that 18% and 20% of the feature revisions propagated in LibSSH and Bison, respectively had patches of code propagated in more than 6 conditional blocks of code. This exceeds the limit of at most 15% of Nr. #ifdefs ≥ 6. Regarding feature interactions in the patches affected when propagating features, all systems presented more than 20% of feature revisions propagated with more than two feature interactions. These metrics allow us to conclude that for most of the feature revisions, a great effort and time would be required from engineers to manually perform feature revision change analysis and propagation without our tooling support.

Table 3 shows the total (T), mean (μ), median (η), and standard deviation (σ) for the number of feature revisions propagated over all random pairs of releases for each subject system. We can see that most of the releases contain changes to existing features of a system, and thus the number of features introduced and deleted (evolution in space) is lower than the number of features revised (evolution over time). In relation to the total number of features propagated, 87.7% were features revised, 11.3% features introduced and 1.0% removed. Therefore, 12.3% of the feature revisions propagated represented evolution in space, and 87.7% represented evolution over time affecting a total of 237,854 patches of code and 14,244 files.

Table 2: Number of conditional blocks, files and features affected per feature propagated.

System	Nr. #ifdefs ≥ 6	Nr. files ≥ 6	Nr. feature interacting ≥ 2
LibSSH	**18%**	10%	**29%**
SQLite	9%	5%	**37%**
Irssi	10%	3%	**26%**
Bison	**20%**	7%	**68%**

Table 3: Total, mean, median and standard deviation of propagation characteristics for each subject system.

	LibSSH				SQLite				Irssi				Bison			
	T	μ	η	σ	T	μ	η	σ	T	μ	η	σ	T	μ	η	σ
FP	672.0	13.4	12.0	6.3	1,367.0	27.3	21.5	28.4	278.0	5.6	5.0	3.3	817.0	16.3	15.5	9.5
FR	620.0	12.4	11.0	6.3	1,175.0	23.5	20.5	16.4	210.0	4.2	4.4	3.1	742.0	14.8	14.0	9.0
FI	52.0	1.1	1.0	0.3	166.0	3.3	1.0	12.8	64.0	1.3	1.0	1.4	72.0	1.4	1.0	3.1
FD	0.0	0.0	0.0	0.0	26.0	0.5	0.0	3.0	4.0	0.1	0.0	0.6	3.0	0.1	0.0	0.4
P	29,783.0	595.7	375.5	524.7	73,197.0	1,463.9	1,065.0	1,637.7	55,897.0	1,117.9	861.5	915.9	78,977.0	1,579.5	1,610.5	650.3
F	2,131.0	42.6	36.0	21.8	3,702.0	74.1	73.0	22.9	5,351.0	107.0	122.0	78.5	3,060.0	61.2	62.5	23.6

FP: features propagated; FR: features revised; FI: features introduced; FD: features deleted; P: patches of code; F: files of source code; T: total; μ: mean; η: median; σ: standard deviation.

Figure 6 shows additional information to Table 3, where we can see propagation characteristics for each pair of releases. For instance, for a pair of releases of SQLite, 199 feature revisions were automatically analyzed and propagated in 172 source code files resulting in 11,454 patches of code. The highest number of features propagated and files of source code affected by propagation was in SQLite mainly because it is the biggest system in terms of lines of code, files, features and Git commits (Table 1). However, although the highest number of feature revisions were propagated in SQLite, the highest number of patches of code was propagated in Bison, with on average 1,579.5 patches of code to propagate around 16 feature revisions per random pair of releases. While Irssi had the smallest number of feature revisions propagated, it had the highest number of files of source code affected by the propagations.

In summary, the random pairs of releases contain evolution in both space and time. The quantitative metrics to characterize our feature revisions propagation show that the systems have a high number of feature interactions to be analyzed when propagating feature

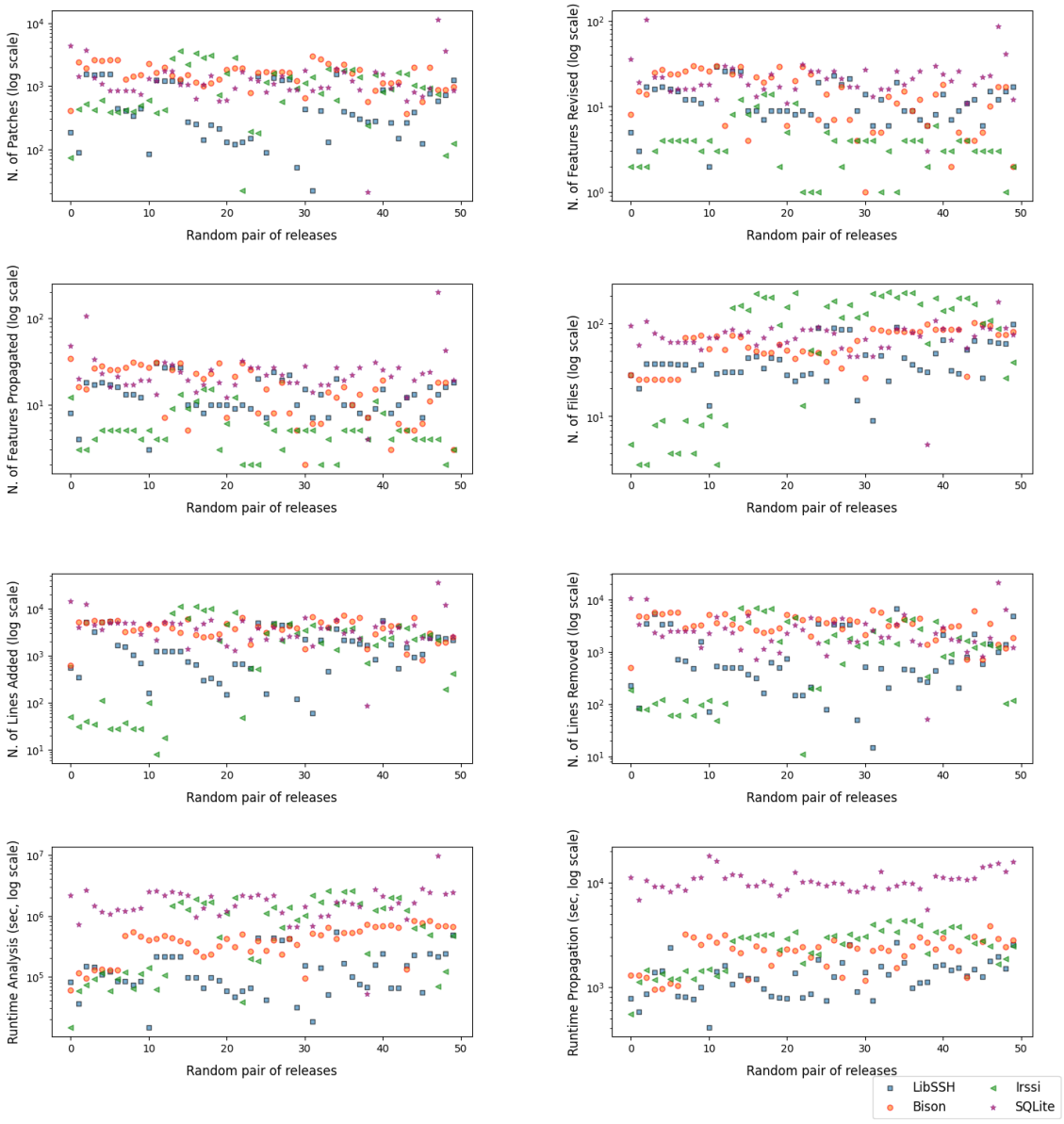


Figure 6: Number of patches of code, features revised, features propagated including features introduced and removed, number of files, lines added and removed for all features propagated between one release to any arbitrary one.

revisions. The systems had a high number of files and patches of code to analyze per random pair of releases, for example, Irssi had on average 19 files affected per feature revision propagated and SQLite had up to 11,454 patches of code to be propagated between two releases. We conjecture that manually analyzing the differences between two releases and finding out feature interactions, files, and patches of code for propagating a specific feature revision is highly complex and error prone. Thus, retrieving information similar to Git VCS but at the feature level to find which files and lines of source code are differing can facilitate the propagation of feature revisions and ease the reuse of feature revisions implementation.

Correct behaviour. Regarding the correct behaviour of our approach, we could propagate feature revisions for each target system with 99% precision, recall, and f1-score, on average, considering the 50 random pairs of releases for each system. Table 4 shows the total number of lines surplus and missing in relation to the total propagated, i.e., not considering the files of source code where no patches were propagated. We investigated why our approach did not reach 100% for the feature revision propagation. The reason is that multiple cases of "dead code" were encountered in the source code files. A dead code is a conditional block of code (`#ifdefs`) that is never included under the precondition enclosing it, which means the block is unselectable and cannot be activated under any configuration option [79]. Thus, if a presence condition of a conditional block is unsatisfiable and no solution is found by the CSP solver, it is a dead code and cannot be linked to any feature revision. Therefore, we found dead codes in some pairs of releases in our experiment during the feature revision change analysis.

Finding dead code is an important issue because it allows detecting defects and bugs in some parts of a system implementation without considering the implementation at large [81]. Dead codes are commonly encountered in preprocessor-based SPLs, for example, the Linux kernel [187]. There exist many dead variable analyses for identifying dead conditional blocks of code [81, 140, 187, 188]. However, although this is not the focus of this work we present some examples of dead codes to explain why our approach could not reach 100% precision and recall. For instance, in release `GNU_1_27`, commit hash e7ae9cf of Bison system, the file *reader.c* contains a conditional block of code with conditional expression `#if 0`. This is often used for commenting out/removing temporarily part of the source code that should not be compiled and potentially will be turned back on later. Another dead code in release `GNU_1_27` was found in file *getopt1.c*, in a conditional block with `#if !defined _LIBC && defined __GLIBC__ && __GLIBC__ >= 2` as conditional expression that is never satisfied.

In conclusion, our approach is effective for propagating feature revisions given that the surplus and missing lines were very few in relation to the total retrieved. Further, it retrieved existing dead codes that may help to find bugs and refactor the SPL.

Runtime. On average, the runtime performance for each feature revision change analysis and propagation per system is presented in Figure 7. The system that took the longest time on average for one feature revision change analysis and propagation was Irssi with 851 seconds. The runtime of feature revision change analysis and propagation varies with the

Table 4: Total number of lines surplus and missing in relation to the total propagated for all random pairs of release.

System	Total	Surplus	Missing
LibSSH	1,640,846	62	1,128
SQLite	7,037,363	1,270	11,488
Irssi	2,446,339	231	2,118
Bison	1,340,663	3,824	11,017

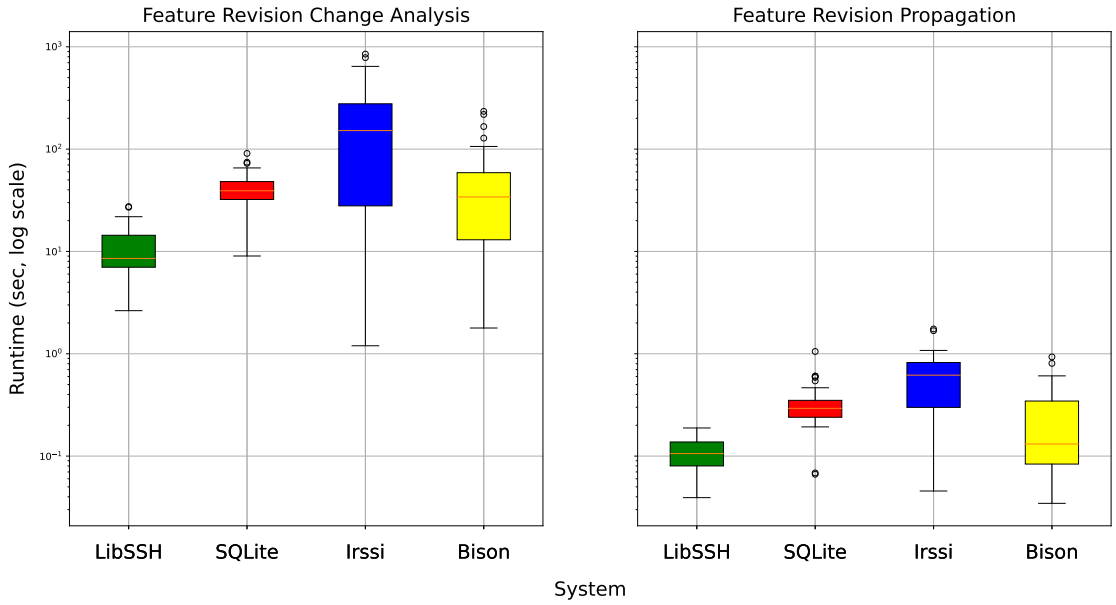


Figure 7: Average Runtime performance in seconds per feature revision change analysis and propagation for each random pair of releases.

number of feature interactions, patches, files, and lines of code involving conditional blocks of code of a feature that differ between two releases, which explains the outliers we observed in all systems. Figure 6 better illustrates the relation of the total number of patches of code, features revised, propagated, files of source code, and lines added and removed to the runtime analysis and propagation. For instance, the pair of releases denominated as "49" from SQLite is clearly the one that took the most time for the feature revision change analysis in comparison to all pairs of releases we considered for the feature revisions propagation. This is because the "49" pair of releases is also the pair with the greatest number of patches, lines of source code added and removed, and features propagated.

In summary, the runtime performance increases with the number of feature interactions and patches of code differing between two releases of a feature revision. By the results, we conclude that our automated approach for reusing feature revision implementation would not delay the SPL development. Instead, it can speed up the process of feature revision change analysis and propagation by retrieving the patches of code differing between source code files of two releases as well as the feature interactions.

Answering our RQ. *How effective is the proposed approach to propagate features and feature revisions in preprocessor-based SPLs in VCSs?* Our approach was able to propagate 3,134 feature revisions with 99% of precision, recall, and f1-score on 200 random pairs of releases of four real-world preprocessor-based SPLs involving evolution in space and time. Although our approach did not reach 100% precision and recall, the propagation behaved as intended if excluding dead codes. The runtime performance was, on average, considering the four target systems ≈63 and 0.2 seconds per feature for feature revision change analysis and propagation, respectively. We also presented that most of the feature revisions propagated were interacting with more than two features, and involving changes in multiple files, and conditional blocks of code. We thus estimate that manually feature revision change analysis and propagation would take a considerable effort and time, while our approach can automate these tasks. We are aware that our approach does not completely substitute developer intervention and as with any other common tasks of reuse, refactoring, and bug fixes, it may require further code adaptations, and tests might be run to verify that the system will behave as intended.

6 Threats to Validity

Internal Validity. We propagated feature revisions over 200 random pairs of releases to evaluate our approach, which can be a bias. However, we carefully analyzed the random combination of releases if at least one feature revision was propagated besides the core of the system, and that there was no pair of releases where the origin was released right after the destination. Thus, we also computed the number of features introduced, deleted, and revised, as well as files and patches of code affected which should have considerable deltas and thus challenging feature revisions propagation. Further, we are proposing a novel approach to propagate feature revisions and there is currently no automated mechanism to retrieve such data. Therefore, we had to validate our approach through testing and code reviews. After propagating feature revisions to a release, the SPL might contain bugs and tests have to be applied. Anyway, this is already necessary for the conventional development and evolution of software systems. Moreover, as there is no way to completely substitute tests and code review, methods and tools will be necessary to assist them. The source code [5] and data [6] are made available, and we encourage researchers to replicate our study and improve our approach for feature revision propagation.

External Validity. The selected systems can be a source of bias. However, we do not attempt to generalize the scenarios but to use them as an experience report from real-world systems to confirm the practical needs and validate our approach. Further, we included these systems as they were recently investigated in a close related work on feature evolution in space and time [127], containing information useful for a preliminary analysis of how much features have been evolved over time. Nonetheless, the subject systems encompass diverse domains and sizes and have been used also in other studies [57, 64, 96, 118, 132, 134, 191]. Further, the systems we used are available and open-source C preprocessor-based SPLs in VCSs. These SPLs have a considerable time of development and history of commits involving evolution in space and time.

Construct Validity. Our experiments rely on evaluation metrics that may affect the construct validity of the results. We use metrics to characterize the feature revisions propagated, however, most of them have already been used in prior study [127]. Another potential threat pertains to the metrics we used in order to assess the correct behaviour of our approach to propagate feature revisions. Precision and recall [190] have been widely used to assess whether the information retrieved of existing approaches to locate source code to feature(s) (revisions) is correct [34, 116, 134]. However, in our study, the calculation of precision and recall depends on our definition of true positives, false positives, and false negatives. To be able to check whether the implementation of a feature revision was successfully propagated, we considered as false positives and false negatives lines surplus or missing, respectively to correct sequences of lines matching between a pair of releases.

7 Related Work

Some tools were designed to support easier comprehension of annotation-based systems by hiding annotations [15] and/or using colors for visualization of features in the source code, such as C-CLR [181], FeatureCommander [49], variation editor [94], PEoPL IDE [136] and CIDE [77]. Although these tools can be used for better comprehension of annotated SPLs, they do not analyze changes in commits and relate them to the evolution in space and time. CheckConfigMX [27] is a tool for change-aware per-file analysis of annotated systems with preprocessor directives to reduce compilation effort. However, this tool is intended to compile only configurations impacted by one code change and the change analysis does not consider the feature level as ours.

Dintzner et al. [38] and Passos et al. [150] presented tools for mining information of

feature-evolution in a variability model, build system and source code. A Web tool called FeatureCloud [37] was presented to mine and visualize changes in #ifdef blocks of systems in Git repositories. Another related work about an empirical analysis of feature-evolution was presented by Michelon et al. [127], mentioning the challenge of dealing with changes in Git commits on preprocessor-based SPLs affecting multiple variants source code. That is why we reuse part of their approach to build up our approach. The main difference between our approach and the previous ones is that we enriched Michelon et al.'s [127] approach with a feature revision change analysis and propagation between releases. Our approach intents to support evolution in space and time of preprocessor-based SPLs at the feature level, as it is a challenge and of paramount relevance for open-source and industry systems using the C/C++ preprocessor directives for implementing variable and configurable systems [21, 71, 74, 127, 134].

Regarding propagation, there is related work of change propagation presented in [135], which presents a browser extension for GitHub synchronizing Java artifacts versions in different products forked from a core repository of a Feature-oriented SPL. There is related work in the context of propagation of patches of code proposing approaches for patch transplantation [178] and backporting [177]. Although these approaches can automatically transfer patches between different versions of preprocessor-based systems, they do not relate patches of code to features. In other words, they do not take into account the features annotated in the source code and do not focus on reusing feature (revision) but rather on automated program repair by reusing and adapting a patch that is already available. Therefore, our approach focuses on a different perspective of propagation by an automated support for change analysis and propagation of an entire feature (revision) implementation. The feature (revision) in our context must be implemented within conditional blocks (`#ifdefs`) in the source code to ease the software reuse and enable its selection by developers or users.

Gupta et al. [63] proposed an impact analysis approach allocating tokens to the changes between two versions of a system for estimating the impact of changes in the code in a semantic way. Some studies focus on the impact analysis techniques based on dependency analysis [93], analyzing the change in semantic dependencies between program entities. For instance, Zhang et al. [194] presented a change analysis for systems developed with another language than C/C++, for AspectJ programs based on AspectJ call graph construction. Also, for change impact analysis of Java systems, Chianti [165] is a tool implemented in the Eclipse environment, which uses tests execution to infer modification of the behavior of a system. Yet, compared to previous work [83] we see that they used a different change analysis of SPL evolution in Git VCS and their analysis counts the number of modified lines containing variability in code, build and model artifacts. Their approach does not focus on the lines modified per feature, and the feature interactions of a delta between releases, as our approach does.

Variation control systems have been proposed to support the evolution of preprocessor-based SPLs or a family of systems that arise from opportunistic reuse by copying, pasting, and modifying without adopting any variability mechanism. For instance, Michelon et al. [134] presented the `ECCO` variation control system as a tool support for tracking features at multiple points in time to artifacts. However, these systems have not become popular and adopted in practice, mainly, because these systems are not mature enough with limited experiences and lack of support for collaborative and distributed development [104, 184]. Further, variation control systems have their own repository and unfamiliar operations [104]. Preprocessor-based SPLs in VCSs, on the other hand, are more popular and convenient tools to deal with evolution in space and time, even though they can result in a system with a high number of feature revisions and variants. In this context, we believe our approach is a promising automated support for reusing features implementation. The main difference between our work and earlier research is that we present an approach for

feature revision change analysis of deltas from two arbitrary releases associated to a specific feature or a set of features and an automated implementation reuse at the feature level in preprocessor-based SPLs.

8 Conclusion and Future Work

We proposed a novel feature revision propagation approach to automatically and efficiently reuse different implementations of features in preprocessor-based SPLs from one release to any arbitrary one. The approach is implemented in a non-intrusive tool, which can automate manual and time-consuming propagation of features to propagate functionalities or new user's requirements.

Our approach provides (i) a feature revision change analysis between an origin and destination release used to propagate feature revisions, as for example which files and lines were modified/introduced/deleted and which features were introduced/deleted/revised and affected by features interactions when propagating a specific feature revision in space or time, and (ii) feature revision propagation advancing in the reuse of evolution in space and time merging the necessary patches of code differing between origin and destination release that affect a specific feature revision to be propagated. By conducting an experiment we evaluated our approach performance, showing that it reaches 99% precision and recall, taking on average for the four target systems ≈63 and 0.2 for performing each feature revision change analysis and propagation, respectively. Further, we also evaluated how much effort our approach can automate by showing some propagation characteristics, such as the number of conditional blocks of code, feature interactions, patches, and files of source code that were automatically analyzed and modified.

For future work, we intend to conduct studies to evaluate the usability of our tool to support developers and engineers. We believe that next future work can be directed toward extending our approach with semantic impact analysis and making our approach available as a browser extension for GitHub. We also plan for future work, an automated recommendation to update features of existing releases when a new version of the system is released to ease decision-making for propagation of feature revisions.

Acknowledgements

Omitted for double-blind review process.

Paper C

Locating Feature Revisions in Software Systems Evolving in Space and Time

Published in the Proceedings of the 24th ACM International Systems and Software Product Line Conference (SPLC 2020), 11 pages, MONTREAL, QC, Canada, 2020

Authors Gabriela K. Michelon, David Obermann, Lukas Linsbauer, Wesley K. G. Assunção, Paul Grünbacher, Alexander Egyed

Abstract Software companies encounter variability in space as variants of software systems need to be produced for different customers. At the same time, companies need to handle evolution in time because the customized variants need to be revised and kept up-to-date. This leads to a predicament in practice with many system variants significantly diverging from each other. Maintaining these variants consistently is difficult, as they diverge across space, i.e., different feature combinations, and over time, i.e., revisions of features. This work presents an automated feature revision location technique that traces feature revisions to their implementation. To assess the correctness of our technique, we used variants and revisions from three open source highly configurable software systems. In particular, we compared the original artifacts of the variants with the composed artifacts that were located by our technique. The results show that our technique can properly trace feature revisions to their implementation, reaching traces with 100% precision and 98% recall on average for the three analyzed subject systems, taking on average around 50 seconds for locating feature revisions per variant used as input.

1 Introduction

The development of large-scale software systems relies on Version Control Systems (VCSs), which offer sophisticated tool support for implementing, maintaining and evolving projects [85]. VCSs are essential for tracking the evolution of software systems over time. However, in addition to evolutionary changes, i.e., revisions, software systems are also subject to re-configuration as different combinations of features are relevant to different users. Such systems are known as Software Product Lines (SPLs), which are families of software products that share a common platform and can be distinguished by a set of features [154]. SPLs are typically realized as Highly-Configurable Software Systems (HCSSs), which use preprocessor directives, e.g., `#IFDEFs`; load-time parameters and conditional execution, e.g., simple IFs; or build systems to generate different product variants [137]. HCSSs aim to satisfy the requirements of different customers and environmental restrictions such as different hardware devices [192]. HCSSs need support to evolve over time, e.g., when fixing bugs or extending existing features, but also to evolve in space, e.g., when adding new features or configuration options. However, it has been shown that existing VCSs do not provide adequate support regarding the evolution in space [21, 99, 105].

Research in the field of HCSSs focuses mainly on solving problems related to the evolution in space. For instance, approaches for re-engineering legacy systems into SPLs typically consider that variants diverge only in terms of the different features they implement [12]. Unfortunately, this assumption rarely holds in practice as variants diverge both in space (different feature combinations) and time (different revisions of features) [71]. Assume an engineer aims to create a variant that uses older revisions of specific features. To avoid analyzing large portions of the project history the engineer needs to remember the exact point in time when the variant existed containing exactly these features in the exact revision. Despite the tools provided by the VCSs, this still remains a manual activity. In this context, techniques supporting such re-engineering and mining tasks are required.

Feature location techniques map system artifacts to features. These techniques support the understanding, maintenance, and evolution of features [12]. However, while existing feature location techniques primarily address variability in space [40, 166], they are not as useful when features evolved over time, leading to divergent implementations [21]. To overcome this limitation, this work presents an automated technique to trace feature revisions to their implementation in variants that evolved independently of each other. Our technique considers possible combinations of features and all revisions made over time.

The contributions of this work are: (i) a technique for locating feature revisions in a set of variants; (ii) an analysis of feature evolution over time in three preprocessor-based SPLs; and (iii) a replication package[1] containing the used data set, the implementation for mining ground truth variants, and the implementation of the feature revision location technique.

2 Motivation

To motivate the need of locating feature revisions, we rely on the feature `HAVE_SSH1` of LibSSH[2]. `HAVE_SSH1` was introduced in Commit c65f56ae[3], comprising five source code files and approximately 600 lines of source code. Analyzing the history of this feature, we can observe that it has eight revisions. In some commits, only small changes over time were observed, as, for example, in Commit d40f16d4[4], where the developers modified eight files, removing two source code files of `HAVE_SSH1`. On the other hand, in Commit f23685f9[5], in

[1] `https://github.com/jku-isse/SPLC2020-FeatureRevisionLocation`
[2] Analysis based on the first 50 commits of LibSSH: gitlab.com/libssh/libssh-mirror
[3] gitlab.com/libssh/libssh-mirror/-/commit/c65f56aefa50a2e2a78a0e45564526ecc921d74f
[4] gitlab.com/libssh/libssh-mirror/-/commit/d40f16d48ec1ed9670c20ffaad1005c59a689484
[5] gitlab.com/libssh/libssh-mirror/-/commit/f23685f92b91aa53546a81bf7793c38a45df15d3

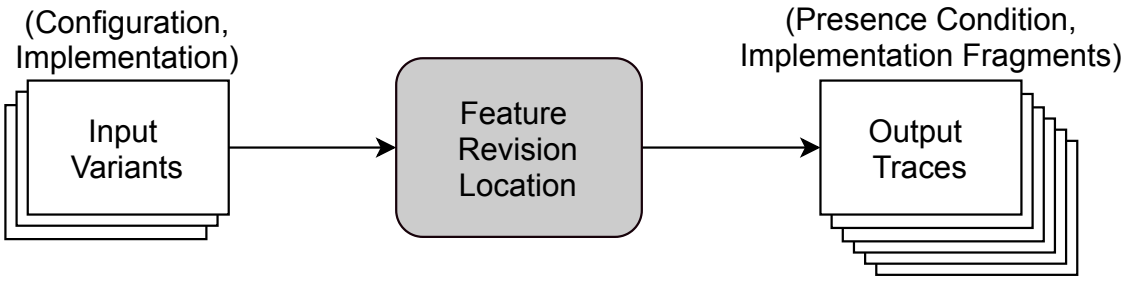


Figure 1: Feature Revision Location Overview.

addition to `HAVE_SSH1`, another feature (`DEBUG_CRYPTO`) also changed over time, and two new features (`HAVE_PTY_H` and `HAVE_STDINT_H`) were introduced, characterizing the system evolution in space. Overall, the revisions of `HAVE_SSH1` happened together with revisions of 13 other features, impacting 73 source code files. These changes were composed of 2627 additions and 1223 deletions of lines of code. Let us assume an engineer wants to recover a version of `HAVE_SSH1` at a specific point in time, for example, from Commit 5f7c84f9[6]. The engineer has to analyze 29 files, 1339 additions, and 188 deletions, which shows the complexity of dealing with variable systems evolving over time. This problem increases significantly if the system is not managed by a version control system and not already implemented as a product line based on features as in the above example. In the worst case, variants are maintained independently as clones without proper revision management.

A unified mechanism for managing system evolution in space and time at the level of features would thus significantly ease the maintenance and evolution of system variants. However, this is a challenging task as already pointed out in existing literature [21]. In this context, we stress the need of managing system variants over time at the level of feature revisions and to ease the management of features and their revisions. Therefore, we present a feature revision location technique capable of mapping implementation artifacts to a certain feature at a certain point in time.

3 Feature Revision Location

This section presents our automated feature revision location technique, which is the main contribution of this work. We first give an overview and introduce basic data structures as well as input (i.e., variants) and output (i.e., traces) of the feature revision location technique. Then, we explain the trace computation in detail. Finally, we discuss the implementation and optimizations of the technique.

3.1 Overview and Data Structures

Figure 1 shows an overview of our feature revision location technique. As *input* it receives a *set of variants*, each consisting of a configuration, i.e., a set of feature revisions, and an implementation. As *output* it computes a *set of traces*, each mapping a presence condition to implementation artifact fragments.

Consequently, we assume the following to be known for every variant: (i) the implementation; (ii) the set of features, i.e., the configuration; (iii) the revision of every feature. The last assumption is difficult to satisfy in an extractive product line adoption scenario, where clones have been maintained independently over a long period of time, as the necessary information must be retrieved first. However, in a reactive product line engineering scenario, where new variants are incorporated into the product line incrementally, this assumption can be satisfied with reasonable effort. Furthermore, variation control systems [99, 105],

[6] gitlab.com/libssh/libssh-mirror/-/commit/5f7c84f900b81e3bbff55378f8170ddf150daf9c

whose goal is to support the user when making changes to a product line, can satisfy the assumption and even profit from our feature revision location technique.

We now describe the concepts and data structures of our technique in detail.

Variants (Input). The *input* is a set of variants V. A variant $v \in V$ is a pair (F, A), where F is a set of feature revisions and A is a set of implementation artifacts. As an example consider a set of three variants $V = \{v_1, v_2, v_3\}$ shown in Table 1.

Table 1: *Input Example:* Set of Variants $V = \{v_1, v_2, v_3\}$.

Variant v_i	**Feature Revisions** $v_i.F$	**Artifacts** $v_i.A$
v_1	$\{\mathcal{A}_1, \mathcal{B}_1, \neg\mathcal{C}\}$	$\{a_1, a_2, a_3\}$
v_2	$\{\mathcal{A}_1, \mathcal{B}_2, \neg\mathcal{C}\}$	$\{a_1, a_2, a_4\}$
v_3	$\{\neg\mathcal{A}, \mathcal{B}_2, \mathcal{C}_1\}$	$\{a_4, a_5, a_6, a_7\}$

Features and Revisions. Every feature f exists in multiple revisions r, denoted as f_r, where f and r are arbitrary unique identifiers for the feature and the revision, respectively. Two variants v_1 and v_2 with the same feature f have the same revision r of feature f, i.e., feature revision f_r, if the feature is implemented in the exact same way in both variants. Absent, i.e., negated, features are not labeled with a revision. While the absence of a feature can influence the implementation of a variant, it makes no sense to label negated features with a specific revision. A feature is either present (in a specific revision) or simply absent. For example, variant v_1 in Table 1 has features $\mathcal{A}$ and $\mathcal{B}$, each in revision 1. Variant v_2 has the same features, but feature $\mathcal{B}$ is implemented differently, e.g., a bug fix might have been applied to feature $\mathcal{B}$ in variant v_2 but not in $v1$, and thus gets another revision assigned.

Implementation Artifacts. A variant's implementation consists of a set of artifacts that are organized in a hierarchical tree structure which we refer to as artifact tree. An artifact can represent a folder, a file, or any other element of a variant's implementation. For example, in the case of source code, an artifact could represent a class, a method, or a single statement. We assume that any two artifacts a_1, a_2 can be compared for equivalence $(a_1 \equiv a_2)$, as follows: two artifacts $a_1, a_2 \in A$ are equivalent $(a_1 \equiv a_2)$ if a_1 and a_2 are equal $(a_1 = a_2)$ and their parent artifacts are equivalent , i.e., their position in the artifact tree is the same.

Traces (Output). The goal of our feature revision location technique is to compute a presence condition C for every artifact a. The *output* therefore is a set of traces T. A trace $t \in T$ is a pair (C, A) that maps a set of artifacts A to a presence condition C. Table 2 presents an example solution of a set of six traces $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$ that match the set of variants V in Table 1. However, this is not a unique solution as alternative sets of traces exist that also match the set of variants V. The three variants in V are not sufficient to determine a unique set of traces. For example, the trace t_1 could also have a more restrictive condition $\mathcal{A}_1 \wedge \neg\mathcal{C}$ while trace t_2 could also have a less restrictive condition $\mathcal{A}_1$. For the three variants in set V this would make no difference. However, it would affect other variants that may potentially be created in the future. The actual output of our feature revision location technique shown in Table 3 therefore contains all clauses that satisfy the criterion for inclusion (see Equation C.1), even if initially redundant. For example, the condition in t_1 could be simplified to just $\mathcal{A}_1$. However, since the input variants were not sufficient to be certain that the actual condition cannot be $\mathcal{A}_1 \wedge \neg\mathcal{C}$ it is still included in the condition.

3.2 Trace Computation

Based on the above data structures, we now explain how the traces and presence conditions are computed.

Table 2: *Solution Example:* Set of Traces $T = \{t_1, t_2, t_3, t_4, t_5, t_6\}$.

Trace t_i	**Presence Condition** $t_i.C$	**Artifacts** $t_i.A$
t_1	$\mathcal{A}_1$	$\{a_1\}$
t_2	$\mathcal{A}_1 \wedge \mathcal{B}_{1\vee2}$	$\{a_2\}$
t_3	$\mathcal{B}_1$	$\{a_3\}$
t_4	$\mathcal{B}_2$	$\{a_4\}$
t_5	$\mathcal{C}_1$	$\{a_5, a_6\}$
t_6	$\neg\mathcal{A} \wedge \mathcal{B}_{1\vee2}$	$\{a_7\}$

Table 3: *Output Example:* Set of Traces $T = \{t_1, t_2, t_3, t_4\}$.

Trace t_i	**Presence Condition** $t_i.C$	**Artifacts** $t_i.A$
t_1	$\mathcal{A}_1 \vee (\mathcal{A}_1 \wedge \neg\mathcal{C}) \vee (\mathcal{A}_1 \wedge \mathcal{B}_{1\vee2})$	$\{a_1, a_2\}$
t_2	$\mathcal{B}_1 \vee (\mathcal{B}_1 \wedge \neg\mathcal{C}) \vee (\mathcal{A}_1 \wedge \mathcal{B}_1)$	$\{a_3\}$
t_3	$\mathcal{B}_2$	$\{a_4\}$
t_4	$\mathcal{C}_1 \vee (\neg\mathcal{A} \wedge \mathcal{C}_1) \vee (\mathcal{B}_2 \wedge \mathcal{C}_1)$	$\{a_5, a_6, a_7\}$

Presence Conditions. We compute the presence condition C for every artifact a in the form of a disjunctive normal form (DNF) formula, whose literals are features (actually a set of feature revisions as we will show). A DNF formula is a disjunction of clauses, where a clause is a conjunction of literals. We treat presence conditions as a set of such clauses. Every clause can be considered as a feature interaction, i.e., a static interaction of the features contained in the clause. This aligns with previous research in feature algebra [107], feature location [102], or the analysis of variable systems [4,53]. We denote the set of all conjunctive clauses that can be formed given a set of feature revisions $v.F$ of variant v as $clauses(v.F)$. For example, $clauses(\{\mathcal{A}_1, \mathcal{B}_1, \neg\mathcal{C}\}) = \{\mathcal{A}_1, \mathcal{B}_1, \mathcal{A}_1 \wedge \mathcal{B}_1, \mathcal{A}_1 \wedge \neg\mathcal{C}, \mathcal{B}_1 \wedge \neg\mathcal{C}, \mathcal{A}_1 \wedge \mathcal{B}_1 \wedge \neg\mathcal{C}\}$. Whether a clause c is part of a presence condition C for an artifact a depends on some fairly intuitive ideas that have already been proven to work very well for simple feature location [129, 138]. In this work we build upon these ideas and extend them to feature revisions. In the following, we first discuss the ideas based on features, ignoring revisions for the time being.

Criterion for Inclusion of Clause in Condition. For a clause c to be contained in a presence condition C of an artifact a, the artifact a must be contained in every variant $v \in V$ that contains the clause c ($c \in clauses(v.F)$) and there must be at least one variant in V that contains clause c.

$$\begin{aligned} c \in C \Leftrightarrow (\forall v \in V : c \in clauses(v.F) \implies a \in v.A) \wedge \\ (\exists v \in V : c \in clauses(v.F)) \end{aligned} \tag{C.1}$$

Criterion for Likely Clause. Our technique additionally provides a smaller and more specific set of clauses C' that is a subset of C to which the artifacts are more likely tracing than to others. This is based on our observation that, in practice, presence conditions with a logical OR between features are much less likely to occur than ones with a logical AND [129]. Therefore, a clause c' is contained in the set of likely clauses C' if all variants that have clause c' also have artifact a (inclusion criterion as above), and in addition, all variants that have artifact a also have clause c' (additional criterion).

$$\begin{aligned} c' \in C' \Leftrightarrow (\forall v \in V : c' \in clauses(v.F) \iff a \in v.A) \wedge \\ (\exists v \in V : c' \in clauses(v.F)) \end{aligned} \tag{C.2}$$

Algorithm 1 Trace Computation

```
1:  function computeTraces(V)
2:    C ← ⋃_{v∈V} clauses(v.F)
3:    A ← ⋃_{v∈V} clauses(v.A)
4:    T ← {}
5:    for each a ∈ A do
6:      C' ← {}
7:      for each c ∈ C do
8:        if (∀v ∈ V : c ∈ clauses(v.F) ⟺ a ∈ v.A) then
9:          C' ← C' ∪ {c}
10:       end if
11:     end for
12:     if C' = {} then
13:       for each c ∈ C do
14:         if (∀v ∈ V : c ∈ clauses(v.F) ⟹ a ∈ v.A) then
15:           C' ← C' ∪ {c}
16:         end if
17:       end for
18:     end if
19:     t ← (C', a)
20:     T ← T ∪ {t}
21:   end for
22:   return T
23: end function
```

Adding Revisions. Extending the previous ideas to revisions is then straightforward. Only one revision of a feature can be present in any given variant. In other words, if a feature f is present in a variant v it is present in exactly one revision r. Therefore, the set of revisions of a feature literal in a clause is the union of all revisions r of feature f that were present when the artifact a was present. Literals in clauses of a presence condition now do not refer to single features anymore, but to a set of feature revisions.

Steps for Trace Computation. Algorithm 1 shows the steps of the trace computation. It receives as *input* a set of variants V. It then computes the sets of all clauses C (Line 2) and all artifacts A (Line 3 in the input variants V. Subsequently, it computes for every artifact $a \in A$ (Line 5) a trace t with conditions C' and artifact a (Line 19) that is added to the set of traces T (Line 20) that is returned (Line 22). The set of clauses C' receives all clauses $c \in C$ that satisfy the inclusion criterion of likely clauses in Equation C.2 (Lines 7-11). If there are no such traces (Line 12) it receives all clauses $c \in C$ that satisfy the regular inclusion criterion in Equation C.1 (Lines 13-17).

3.3 Implementation and Optimizations

When applying the aforementioned concepts in practice, we perform the following optimizations:

Feature Interaction Limit. We limit the maximum size of clauses in presence conditions, i.e., the number of feature literals in a conjunction, which corresponds to the number of interacting features, to a threshold based on previous empirical research [51, 53]. This provides a major improvement to the scalability of the approach, otherwise, i.e., without a constant threshold, the number of clauses would grow exponentially with the number of features. While the threshold can be freely configured, for the evaluation presented in this paper it was set to at most three interacting features.

Negated Feature Literals. We do not label negated feature literals with a revision. While the absence of a feature can influence other features and thus have an effect on the implementation of a variant [107], it makes no sense to have a clause containing only negated features and to label negated features with a specific revision.

Artifact Clusters. We do not consider every artifact individually, but rather cluster artifacts, i.e., group artifacts together, that never appeared without each other in any variant and assign presence conditions to those clusters instead of every individual artifact. For example, artifacts a_1 and a_2 in the set of variants V in Table 1 always appear together and never without each other. We therefore group them instead of treating them individually, as shown in Table 4.

Artifact Sequence Alignment. Our technique relies on the ability to compare any two implementation artifacts for equivalence. In cases where two sibling artifacts a_1 and a_2 (i.e., artifacts with the same parent) are not unique, the order of the artifacts is important when determining equivalence. This is the case, for instance, if the same statement appears multiple times inside a method. In such cases an alignment of the artifact sequences must be performed. We adapted a Longest Common Subsequence (LCS) algorithm [35] to perform multi-sequence alignment for comparing more than two variants [51, 103], e.g., if they have the same method whose statements must be aligned.

Artifact Adapters. We keep the technique independent of the types of implementation artifacts by utilizing artifact type specific adapters that are responsible for parsing respective files and generating the generic artifact tree structure consisting of folders, files, and further file type specific artifacts. The only requirement is that artifacts can be uniquely identified and compared for equivalence.

Element Counters. We count for every clause c in how many input variants it was contained, for every artifact cluster a in how many input variants it was contained, and for every pair (c, a) of clause and artifact cluster in how many input variants both were contained together. These counters are sufficient to evaluate the above criterion for inclusion of clauses in presence conditions (see Equation C.1). This has the advantage that it works incrementally, i.e., new input variants can be added whenever necessary simply by increasing the respective counters. Hence, already computed traces do not have to be recomputed when a new variant is encountered. Instead, the counters are simply increased and the existing presence conditions trimmed, i.e., clauses removed for which the above conditions do not hold anymore.

Table 4 presents an abstract example of the counters that match the set of variants V in Table 1. The rows list the four artifact clusters with the total number of appearance in variants. The columns list (a subset of) the clauses $c_i \in \bigcup_{v \in V} \text{clauses}(v.F)$ with the total number of appearance in variants, sorted by the number of literals (i.e., interacting

Table 4: *Implementation Example:* Subset (cut off right) of Counters for Artifact Clusters (rows) and Clauses (columns).

		$\mathcal{A}$	$\mathcal{B}$		$\mathcal{C}$	$\mathcal{A} \wedge \mathcal{B}$		$\mathcal{B} \wedge \mathcal{C}$
		2	3		1	2		1
		$\mathcal{A}_1$	$\mathcal{B}_1$	$\mathcal{B}_2$	$\mathcal{C}_1$	$\mathcal{A}_1 \wedge \mathcal{B}_1$	$\mathcal{A}_1 \wedge \mathcal{B}_2$	$\mathcal{B}_2 \wedge \mathcal{C}_1$
		2	1	2	1	1	1	1
a_1, a_2	2	2	1	1	0	1	1	0
a_3	1	1	1	0	0	1	0	0
a_4	2	1	0	2	1	0	1	1
a_5, a_6, a_7	1	0	0	1	1	0	0	1

features), first in total without considering revisions, and then per revision. Each cell contains the number of times that the artifact cluster and the clause appeared together in a variant. For example, artifacts a_1 and a_2 appear in two variants. The clause $\mathcal{A}_1$ also appeared in two variants. Finally, the artifacts and the clauses appeared together also in two variants. Therefore, the criterion for likely clauses (see Equation C.2) is satisfied.

4 Evaluation

This section presents the research questions and the method adopted for evaluating our feature revision location technique. We introduce the input data set, explain the process adopted to obtain the ground truth used for comparison, and describe the metrics used to evaluate our technique.

4.1 Research Questions

The evaluation of our feature revision location technique was guided by two research questions (RQs), presented below. The next subsections describe the methodology used to answer the RQs.

RQ1. Can the proposed technique locate feature revisions from a set of existing variants? In this RQ we aim to evaluate how effective our technique is for locating feature revisions in existing variants of HCSSs obtained from VCSs.

RQ2. Can new variants be composed with feature revisions located by our technique? The goal of this RQ is to investigate if we can use artifacts, i.e., feature revisions, located by our technique from existing variants to compose new variants.

4.2 Method

The methodology followed to evaluate our feature revision location technique and answer the RQs is illustrated in Figure 2. We started by mining ground truth variants (step 1) from changes of features in HCSSs in VCSs (cf. Section 4.4). We then applied our feature revision location technique to the ground truth variants (step 2, cf. Section 3). The process of locating feature revisions was performed incrementally with the input variants. Thus, as long as we had different input variants, we used them for locating feature revisions with our technique, which continuously created new and/or refined existing traces. After having located feature revisions from all existing input variants, we used the computed traces to compose variants (step 3) by joining the artifacts of the desired configurations. Next, we compared the composed variants with the corresponding ground truth variants, i.e., containing the same configuration (step 4). The comparison of variants was performed by comparing each composed artifact with each ground truth artifact file-by-file and line-by-line. For computing differences, we used a library for performing the comparison operations between textual data[7]. Finally, we compute metrics (step 5) to quantify missing relevant information or surplus information retrieved (cf. Section 4.5).

4.3 Data Set

The evaluation of the proposed technique relies on three open source HCSSs [96] using the VCS Git (see Table 5): (i) Marlin, a variant-rich open-source embedded firmware for 3D printers[8]; (ii) SSH library, a Robot Framework test library for SSH and SFTP network

[7] `https://github.com/java-diff-utils/java-diff-utils`
[8] `https://github.com/MarlinFirmware/Marlin`

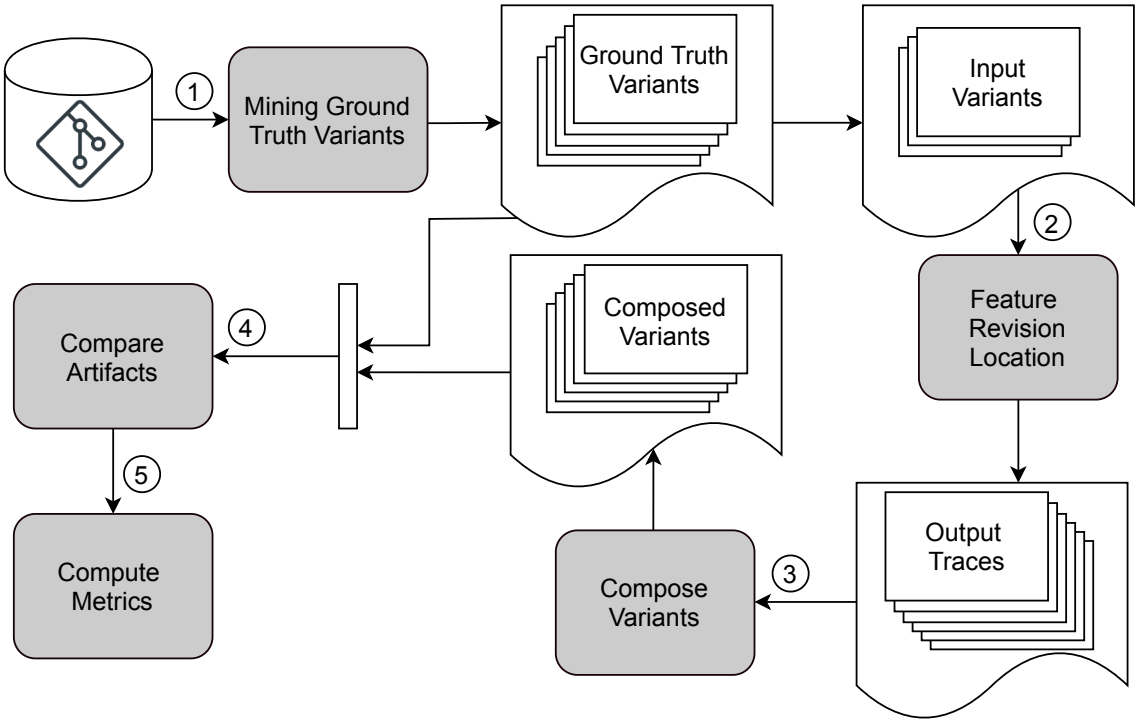


Figure 2: Methodology for evaluating our feature revision location technique.

Table 5: Overview of the subject systems.

System	Release	Since	LoC	Features	Revisions
Marlin	2.0	2011	281355	37	144
LibSSH	0.9	2005	110590	49	129
SQLite	3.7.4	2000	173714	7	55

protocols[9]; and (iii) SQLite, a library that implements an SQL database engine[10]. We tried to avoid bias by choosing three different application domains. Furthermore, each system has a considerable history of development and use in research [57, 64, 85, 88, 96, 118, 191]. Moreover, we choose systems of different sizes, which we measured by counting the total number of lines of code of their last release (excluding blank lines and comments). We used variants from the first Git commits from the main trunk ordered by date of each system to avoid bias in choosing a specific interval of commits. Despite our technique can be adopted for any number of variants, our implementation currently has scalability limitations for high number of feature revisions. Thus, we used variants mined from the first 50 commits, which give enough feature revisions to apply and evaluate the ability of our technique for locating feature revisions.

4.4 Mining Ground Truth Variants

Ground truth variants cannot come from only a single point in time. Thus, in order to have input variants that contain features at different points in time, we extract variants of a system whenever a feature evolved over time, i.e., was changed via a Git commit [132]. To explain each step of the methodology for mining ground truth, we will use the example shown in Figure 3. Let us consider that the code of the file before the change in line 12 was added in a specific point in time called T1. Later, a second commit was performed at point

[9] `https://gitlab.com/libssh/libssh-mirror`
[10] `https://github.com/sqlite/sqlite`

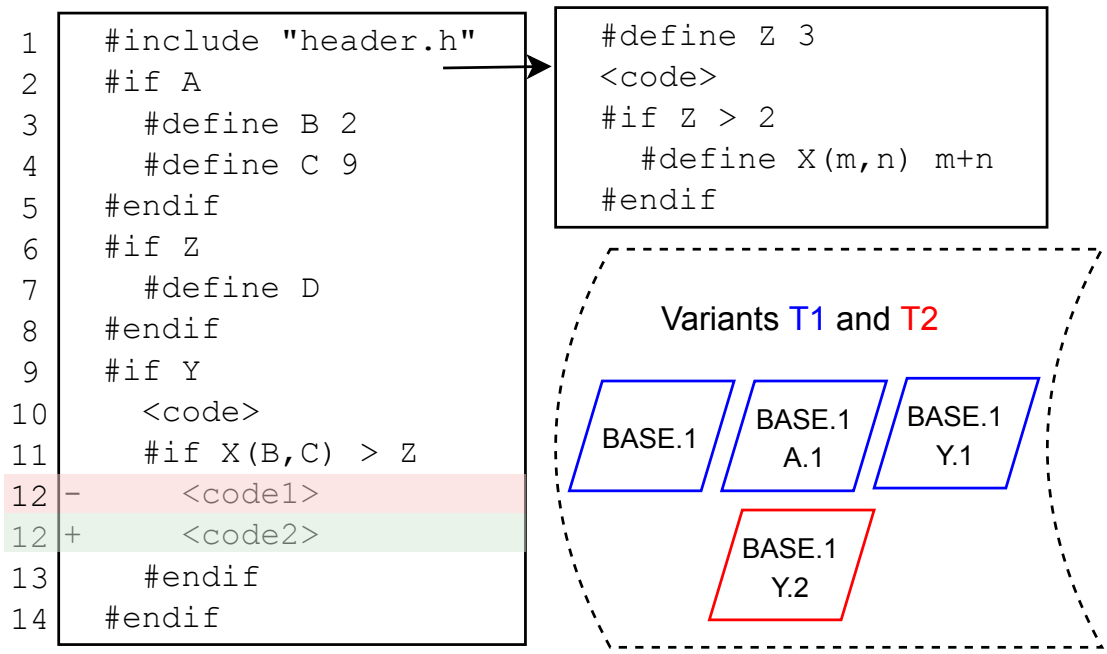


Figure 3: Mining changes over time to generate ground truth variants to evaluate our feature revision location technique.

in time T2, where the code of line 12 changed. We identify the possible features in these two points in time. In this example, three features are introduced in point T1 (`BASE`, `A`, `Y`) and one existing feature changed in point T2 (`Y` revision 2). We used this information to create our ground truth variants used as input for our technique for locating feature revisions in five steps, described next.

Identify feature literals. To identify possible features, we classify all annotated literals of the system along all Git commits analyzed. We distinguish external, internal and transient literals. External literals can only be set externally to configure variants. In Figure 3 `A` and `Y` are external literals. Internal literals are defined at some point in the code via a `#define` directive. In Figure 3 the literals `B`, `C`, `X` and `Z` are internal. In commit #1[11] of LibSSH the file `wrapper.c` contains 15 conditional blocks (`#ifdefs`), from which only four expressions contain an external feature (`DEBGU_CRYPTO`). The conditional block with feature `HAS_BLOWFISH`, for example, is an internal feature defined in the beginning of the file inside the conditional block of an external feature (`HAVE_OPENSSL_BLOWFISH_H`). We considered literals as features only if they were external in all revisions.

Resolving macros in conditions. For each analyzed Git commit, we start preprocessing the annotated code respective to macro functions (macros that can accept parameters and return values). The output of this step is the code from the specific commit with all macros in conditions resolved, i.e., the macro code is expanded to the degree where the conditions of the conditional statements only consist of literals. After expanding macros in conditions, all `#define` and `#include` statements and conditional blocks remain in the code, as they can modify the resulting code of the variants. In Figure 3 the only line that will change after processing this step is the line 11, which is replaced by `#if 2 + 9 > Z`.

Compute changes. For each Git commit n we create a tree structure with the conditional

[11] gitlab.com/libssh/libssh-mirror/-/commit/c65f56aefa50a2e2a78a0e45564526ecc921d74f

blocks to determine the differences between the actual commit and the previous *n-1*. In case of the first Git commit of the project, we consider all files inserted as the difference. From the differences we can get the tree node reflecting the changes. In case that any external feature changed or differences are in non-code files, e.g., binary, `BASE` is considered the changed feature, i.e., for every code added/removed in the body of the project that does not belong to an external feature the root feature `BASE` is considered as the changed node. In Figure 3 a new file was added at point T1 in addition to its include file. At point T2 we just have the change in line 12 of the main file. For example, in LibSSH commit #1 added 78 new files, commit #2[12] removed 8 files, while commit #3[13] comprised changes of lines added and removed in different files.

Compute configurations. Every changed node is then used to generate a variant that contains the code activated by this node. We used the Choco solver[14] to provide the first possible solution for a given constraint to activate the conditional blocks. To find a configuration for the preprocessor that activates the desired block of code, we need to obtain an assignment for all the annotated literals that are part of the condition of the block. The basic idea here is to create a set of constraints that are then handed over to a solver. Overall, the constraint we build consists of three parts, which will be explained using the example in Figure 3. Firstly, we retrieve the local condition closest to the changed code, which in the example at point T2 is: `2 + 9 > Z`. The second part is the entire condition of the desired block, which is a conjunction of all parent conditions. We obtain it by walking up the tree, starting from the changed node, which will be: `Y && (2 + 9 > Z)`. The feature implications make the third part used to create and apply a mapping of all internal literals to just external literals. In Figure 3, it can be seen that `A` defines `B=2` and `C=9`, and `BASE` defines `B=3`, which means that we can map the code block to `BASE` and `Y` and `A`. The process of traversing the tree to build the feature implications works as follows: we read until the end of the file. When a `#define` is found, we take the condition of the block which it is part of. With this information, we build an implication chain. Before handing this chain to the solver, we filter out the ones that are not necessary for the currently processed node to reduce the work for the solver. We do it by recursively analyzing the literals of the conditional expressions that define other literals. If they were defined at some point, we build their queue of implications too. For example, in Figure 3 we do not add to the queue of implications the feature Z implying `D` in line 7 as this feature does not influence to activate the changed block of code (lines 11-13). We just need the implications of features implying `Z`, `B` and `C`. In commit #1 of LibSSH, as mentioned before, the conditional block annotated with feature `HAS_BLOWFISH` is defined inside another conditional block of the external feature `HAVE_OPENSSL_BLOWFISH_H`. In the example of this block of code changed, we will have a queue of implications for feature `HAS_AES`, `HAS_BLOWFISH` and `OLD_CRYPTO` as they are defined over the file `wrapper.c`. However, we just send to the solver the queue of implications of feature `HAS_BLOWFISH` which contains the necessary condition to define this internal feature (`(HAVE_OPENSSL_BLOWFISH_H) && (BASE)`).

The constraints built by the queue of implications are then handed to the solver. If the solver finds no solution, it means that the part of code we wanted to activate is dead and there is no configuration that can activate that block. If a solution is found, we validate that all the literals that get assignments are really external by filtering out all other assigned literals. If the set of assignments is not empty at that point, we obtained a configuration for a valid variant. Before using these variants as ground truth for evaluating our technique it is essential to know what features should be marked as changed for the respective changed node and thus treated as a feature revision. We assume the features annotated closest to a change are the ones having caused it. Therefore, we get a solution using the local condition

[12] gitlab.com/libssh/libssh-mirror/-/commit/d40f16d48ec1ed9670c20ffaad1005c59a689484
[13] gitlab.com/libssh/libssh-mirror/-/commit/55846a4c7b09af2d105c7f7dfd0a43aab2f6e5a5
[14] `https://github.com/chocoteam/choco-solver`

without any implications.

In case this does not obtain any potentially changed features, meaning that there are no positive external features in the closest condition, we repeat the same process with the parent conditions until we find a positive external feature from the solution. In the worst case, the outcome is that we reach the node corresponding to `BASE`, which is trivially a positive solution. In Figure 3 one of the variants from T1 was `BASE` which is the feature assigned for the code of the header file and of lines 6-8, as it just contain internal features. Another variant at time T1 will be related to the block of code in the main file in lines 2-5, which contains feature `A` and `BASE`. Still in time T1 we have a variant with feature `Y` and `BASE` from remaining lines of code 9-14. In time T2 we just have a new variant regarding the change in line 12 which contains a new revision of the feature `Y` (the closest external feature) and the previous revision of the feature `BASE` from time T1. Regarding LibSSH, we could see in the first 50 commits analyzed that 49 features were introduced and over their changes a total of 129 revisions were identified, resulting in 129 variants.

Generate ground truth variants. After performing these previous steps, we are able to generate the ground truth variants by partially preprocessing the code. Finally, the solution found by the Choco solver for the configuration is used to retrieve the variant, which from now on is ready to be used as input for locating feature revisions. Figure 3 illustrates the variants mined with a set of feature revisions from the changes over time T1 and T2.

4.5 Metrics

The precision, recall, and F1 score measure how well information is retrieved by a system in relation to the relevant information [190]. They are commonly used to evaluate feature location techniques [34, 116, 129]. In order to assess how effective our technique is to correctly locate and not missing any relevant artifacts, we analyzed if the stored traces allow to retrieve the artifacts belonging to a specific feature revision. We applied the aforementioned metrics by comparing artifacts of feature revisions composed by the traces of our technique with the artifacts of the ground truth. We used two levels of granularity, due to the feature evolution analyzed, and the different kinds of files that existed in the subject systems (C/C++, binary and text files): (i) *file-level* comparison of two complete files by matching their content; (ii) *line-level* comparison of two code files. The precision of the file-level comparison is the percentage of correctly composed files, i.e., retrieved files with entire content matching the relevant ones. The recall measures the total percentage of entire matching of all composed files relative to the all relevant files. Regarding line-level comparison, precision is the percentage of correctly retrieved lines while recall is the percentage of matched lines retrieved relative to the total of relevant lines.

Furthermore, we also measured the runtime of our technique. The experiments were performed on a HP EliteBook laptop, with an Intel® Core™ i7-8650U processor (1.9GHz, 4 cores), 16GB of RAM, SSD storage, and Windows 10 operating system.

5 Results and Discussion

This section presents an empirical analysis of the feature evolution in space and time from the three subject systems. We present and discuss the results obtained with our feature revision location technique and answer the two RQs.

Analysis of feature evolution in space and time in subject systems. We present in Figure 4 the evolution in space and time, i.e., the features introduced/changed in the first 50 Git commits of three systems. The blue line in a row represents the time between the inclusion of a feature in the Git repository and the last revision of that feature. The first red diamond in a blue line represents the inclusion of a feature and the other ones along the lines

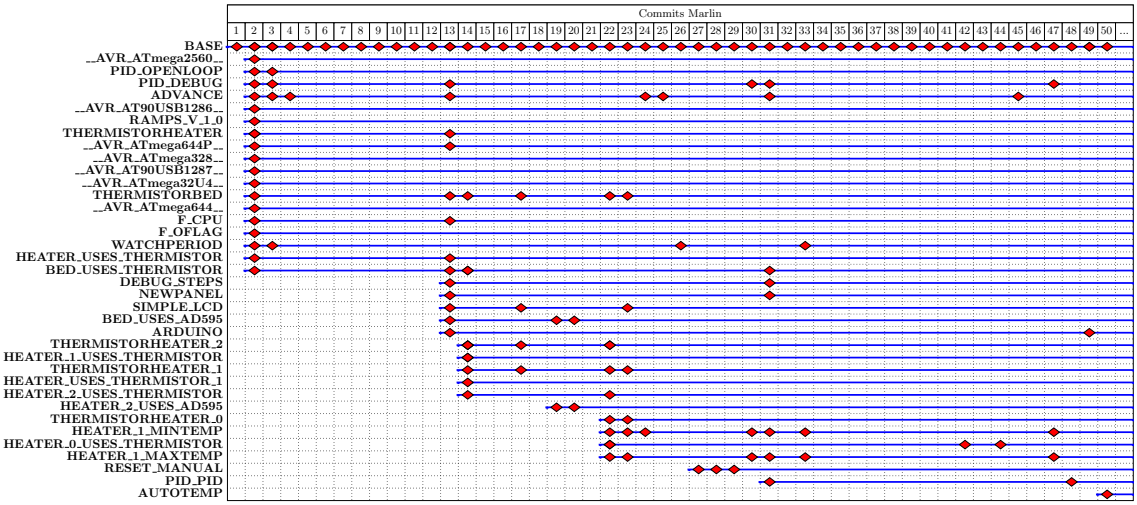
are feature revisions. First, analyzing system evolution in space, we can see the feature evolution of the Marlin system in Figure 4a. After the product started with Git commit #1 with just `BASE`, the second Git commit introduced 18 new features. Furthermore, additional new features were included in the following commits: #13 (+5), #14 (+5), #19 (+1), #22 (+4), #27 (+1), #31 (+1), and #50 (+1). For the LibSSH system shown in Figure 4b the initial version started in Git commit #1 with 16 features. Then, there were four evolution-in-space changes (Git commits #12, #20, #34, and #38) including 33 new features. In case of the SQLite system shown in Figure 4c, the first commit had no files, so just in the second commit we have feature code introduced. Along the commits analyzed, commit #2 had six features introduced of the total of seven existing until Git commit #50. The remaining feature appeared in Git commit #7. For every feature revision along the 50 Git commits, there was no evolution in time because changes were introduced only in `BASE` code. Thus, over few Git commits, we can also see that many features change over time besides `BASE`. By looking to Figure 4a we can observe a great density of feature revisions in 20 Git commits (between #13 and #33), adding 93 feature revisions, which represent 65% of all revisions. The evolution in time by feature revision can have much impact in HCSSs product configurations. For example, the commit #16 of Marlin changed nine different features and commit #38 in LibSSH included four new features and changed five other ones. This evolution in time and space makes engineering tasks complex. Suppose an engineer needs to recover an older version of feature `ADVANCE` before commit #31, keeping the change of other features. This would require great effort and would be error prone, since other current variants of Marlin system could be still using the newer version of that feature. Considering these three subject systems, we see different magnitudes of software systems' variability in time.

Locating feature revisions with our technique. The results related to the quality [65] of our technique for locating feature revisions are shown in Table 6. The precision for the three subject systems was 100% at the file and line level of granularity. Recall values are almost all optimal (retrieving 95% up to 99% of the total relevant artifacts). The values of F1, which consider both precision and recall, are between 97% and 99%, what shows that our technique is reliable to locate feature revisions by a given set of variants in different space configurations and in many points in time.

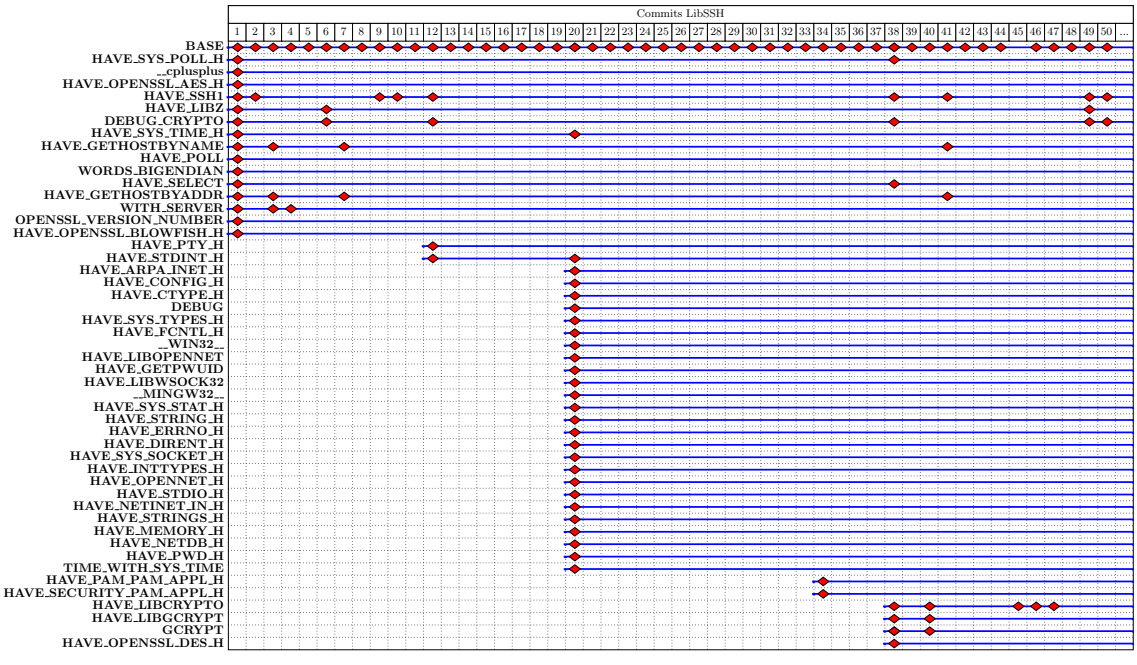
The false negative lines in Marlin were about 3,138 from a total of 1,460,533 relevant lines over 144 compared variants (min: 0, max: 163, mean: 21.79 per variant). From false negative lines, 774 are comment lines. Those false negative lines were caused by incorrect traces since Git commit #31, which were part of feature revisions evolved up to the last Git commit analyzed. In LibSSH, 1,470 lines were missing over all 129 composed variants (min: 0, max: 106, mean: 13.02 per variant) of a total of 4,344,801 relevant lines. Those false negative lines were incorrect traces of artifacts and feature revisions since Git commit #10. In SQLite, three lines were false negatives on composed variants corresponding to the Git

Table 6: Average precision, recall and F1 metrics of composed artifacts per system.

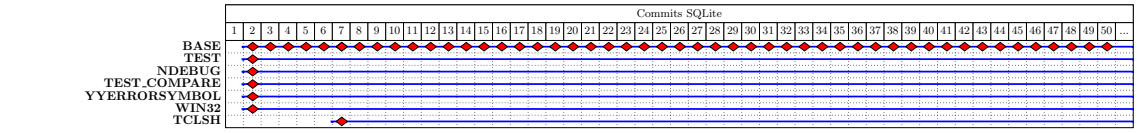
Subject System	Granularity	Precision	Recall	F1
Marlin	Files	1.00	0.95	0.97
	Lines	1.00	0.99	0.99
LibSSH	Files	1.00	0.99	0.99
	Lines	1.00	0.99	0.99
SQLite	Files	1.00	0.99	0.99
	Lines	1.00	0.99	0.99
All	Files	1.00	0.98	0.99
	Lines	1.00	0.99	0.99



(a) Marlin.



(b) LibSSH.



(c) SQLite.

Figure 4: Marlin, LibSSH and SQLite evolution in space and time on the first 50 commits on Git VCS.

commit #37, resulting in 39 false negative lines over all 55 composed variants (min: 0, max: 3, mean: 0.71 per variant) among a total of 727,387 relevant lines. The false negatives were caused by incorrect alignments performed by the LCS algorithm. An adapter with a more fine-grained tree structure for the specific programming language may contribute to a more precise alignment and higher precision and recall.

RQ1. Can the proposed technique locate feature revisions from a set of existing variants? *Our technique proved to be effective for locating feature revisions in the used data set, with high values for the measures of precision, recall and F1. The proposed technique correctly located the artifacts in an automated way, which can help developers to easily perform this task and save time.*

Composing variants with new configurations of existing feature revisions. Table 7 shows the values of precision, recall, and F1 from comparison of artifacts (file and line levels) of the ground truth and our composed variants. Our technique retrieves artifacts with precision of 100% and recall of 93%-99% at file-level granularity. At line-level granularity, the average precision is 100% for SQLite and 99% for the other two. Recall is 99% for the three subject systems. All values for F1 are greater than 96%, almost the same as the minimum F1 achieved when comparing the artifacts of composed variants with the ones from the input variants (97%).

For Marlin, within a total of 496769 relevant lines, 1782 were false negative lines and from these, 394 are just comment lines. In the case of LibSSH, 684 lines were false negatives, five of which are comment lines, from a total of 1661585 relevant lines. In case of SQLite, 39 lines were missing in the composed files from a total of 727897 relevant lines of ground truth files. The false negative lines were missed due to some wrong traces caused by incorrect alignments of lines with the LCS algorithm. False positive lines were composed in variants from Marlin (four lines added) and LibSSH (19 lines added) systems. The false positive lines in the composed variants are caused by feature interactions in the configurations, which we randomly chose without considering whether a selected feature excludes parts of code that can be in other features when preprocessing ground truth variants. This can result in an invalid configuration, where the preprocessed variant as ground truth is missing artifacts. As an example, the random variant generated in Git commit #12 of LibSSH, which contains the features `HAVE_SSH1`, `DEBUG_CRYPTO`, `HAVE_PTY_H` and `BASE`.

Listing C.1 shows that when preprocessing a variant with feature `HAVE_SSH1` defined, the ground truth variant will contain Line 2 and not Line 4. Only when this feature is not defined Line 4 will be present in the variant. Our feature revision location technique then mapped the artifact from Line 2 to presence conditions containing feature `HAVE_SSH1` and Line 4 to presence conditions containing `BASE` and other features from the respective point

Table 7: Average precision, recall and F1 metrics of composed artifacts for random configurations per system.

Subject System	Granularity	Precision	Recall	F1
Marlin	Files	1.00	0.93	0.96
	Lines	0.99	0.99	0.99
LibSSH	Files	1.00	0.95	0.98
	Lines	0.99	0.99	0.99
SQLite	Files	1.00	0.99	0.99
	Lines	1.00	0.99	0.99
All	Files	1.00	0.96	0.98
	Lines	0.99	0.99	0.99

```
1 #ifdef HAVE_SSH1
2     option->ssh1allowed=1;
3 #else
4     option->ssh1allowed=0;
5 #endif
```

Listing C.1. code snippet from LibSSH, file options.c.

in time. Thus, our composed variant with the combination of features will contain artifacts of both `#ifdefs` and `#else` blocks that are missing in the ground truth variant.

RQ2. Can new variants be composed with feature revisions located by our technique? *The traces computed by our technique proof useful for creating new variants with random configurations, still achieving high values for the measures of precision, recall and F1. Our feature revision location technique can therefore support tasks such as extractive SPLE.*

Performance of our feature revision location technique. The performance of our technique is shown in Figure 5. It took around 50 seconds on average, and even in the worst cases not longer than 200 seconds. As expected, the runtime for locating feature revisions increases with the number of feature revisions because the number of artifacts and features is greater to refine the traces. As Marlin and LibSSH subject systems have more feature revisions to be located, they presented outliers that were probably caused by the garbage collector unexpectedly slowing down the application. Although SQLite has fewer feature revisions, the size of artifacts that have been changed is such as big as in the other systems. Thus, independently of the number of feature revisions, the size of artifacts can slow the process to refine traces.

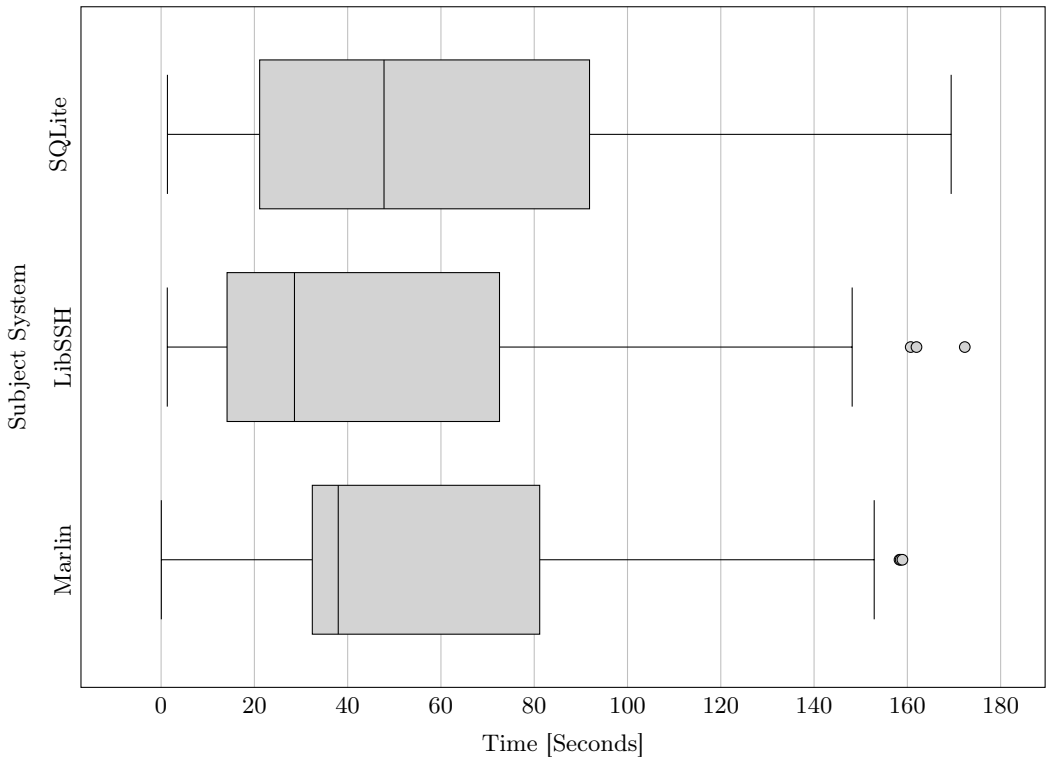


Figure 5: Runtime of feature revision location per variant.

6 Threats to Validity

The threats to *construct validity* are related to the study setup. Firstly, the scenarios used to validate our technique contain changes to features, but we did not have data on the type of evolution, e.g, performance improvement, new hardware support, bug fixing, etc. Secondly, the methodology chosen to evaluate our technique was based on variants in space and time created by a configuration of features in specific changes of annotated code in the Git commits we analyzed. This was necessary since there was no ground truth we could use. To mitigate this threat, we generated new variants with different configurations of feature revisions (not used as input) randomly chosen for the points in time analyzed.

A threat to *internal validity* is the limitation of the underlying tools that could have affected our results. We used our own developed tool to compose variants. However, our developed tool is available for further comparison and was widely used in previous works where it successfully composed variants [51, 52, 56, 97].

A threat to *external validity* is the generalization of the results. Our evaluation was conducted with a subset of commits from three Git projects. The three selected target systems are from different domains and have different sizes with different behaviors in terms of how often their features change along the Git commits, so we believe that the results cover diverse enough scenarios.

7 Related Work

The idea of creating software versions started when important dimensions of evolution such as revisions and variants were introduced. Conradi et al. [32] defined revisions as a software versions that evolve along the time dimension. In this context, our feature revision location technique can help to get the feature variability along the time dimension. However, the evolution of variable software systems is still a challenge. While product line engineering requires new tools and processes, VCSs do not scale with the number of variants and require the consolidation of cloned variants into a product line. Moreover, evolving a product line is more complex than evolving single variants [21]. Indeed, VCSs and the annotation-based preprocessors are still widely used as a variability mechanism for handling a high number of revisions and variants. To improve on the current situation, variation control systems, a special kind of VCSs with a focus on variant management rather than revision management, have been developed [99, 105].

In order to support the extractive adoption of SPLs by reusing existing variants as the basis for the core assets several feature location techniques have been proposed [13]. However, the feature revision concept is still untreated among feature location techniques in the literature [13, 34, 40, 129, 138, 166]. As suggested by Hinterreiter et al. [71], maintaining revisions of individual features may help to understand the evolution history of a variant and capture ongoing changes. Thus, with our automated technique, developers can re-engineer a system for feature-oriented development and manage evolution in the time dimension by means of feature revisions.

8 Conclusions and Future Work

To the best of our knowledge, existing feature location techniques are limited to one certain point in time. Due to this limitation, this paper highlights the importance of feature location in both space and time and introduce a new automated feature revision location technique, allowing practitioners reason about variants in different points of time. Our results showed that our feature revision location technique can locate the features' artifacts with a precision of 100% at file-level and line-level granularity and a recall of, at least, 95% at file-level and 99% at line-level granularity. Regarding the performance of our feature

revision location technique, we reported that it took on average 50 seconds to trace artifacts to feature revisions for each input variant. Even if manual completion is necessary, it will not require extensive code additions or deletions by a developer. Thus, our automated technique can aid developers re-engineering software systems into SPLs at the level of feature revisions, thereby saving time and effort. Hence, facilitating the management of system variability in space and time by the possibility of composing variants with feature revisions.

We hope with our results to inspire researchers and tool builders to work with feature revisions, treating feature evolution in space and time and encourage them to improve our technique, which can be compared with common metrics and available ground truth used in our work. As future work, we want to conduct more experiments with industrial systems and from different domains and consider other programming languages such as Java. Furthermore, we will improve the scalability of our technique implementation for dealing with the growth of feature revisions. Then, we will seek on how to provide an independent mechanism for enabling the management of variants with any combination of feature revisions.

Acknowledgment

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; CNPq, grant no. 408356/2018-9 and FAPPR, grant no. 51435. The support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

Paper D

Evolving Software System Families in Space and Time with Feature Revisions

Published at the Springer International Journal Empirical Software Engineering (EMSE), 54 pages, 2022

Authors Gabriela K. Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon and Alexander Egyed

Abstract Software companies commonly develop and maintain variants of systems, with different feature combinations for different customers. Thus, they must cope with variability in space. Software companies further must cope with variability in time, when updating system variants by revising existing software features. Inevitably, variants evolve orthogonally along these two dimensions, resulting in challenges for software maintenance. Our work addresses this challenge with *ECSEST* (Extraction and Composition for Systems Evolving in Space and Time), an approach for locating feature revisions and composing variants with different feature revisions. We evaluated *ECSEST* using feature revisions and variants from six highly configurable open source systems. To assess the correctness of our approach, we compared the artifacts of input variants with the artifacts from the corresponding composed variants based on the implementation of the extracted features. The extracted traces allowed composing variants with 99-100% precision, as well as with 97-99% average recall. Regarding the composition of variants with new configurations, our approach can combine different feature revisions with 99% precision and recall on average. Additionally, our approach retrieves hints when composing new configurations, which are useful to find artifacts that may have to be added or removed for completing a product. The hints help to understand possible feature interactions or dependencies. The average time to locate feature revisions ranged from 25 to 250 seconds, whereas the average time for composing a variant was 18 seconds. Therefore, our experiments demonstrate that *ECSEST* is feasible and effective.

1 Introduction

Software system families can evolve in two dimensions: (i) evolution *in space*, when new product variants need to be customized, and (ii) evolution *in time*, when features of existing individual product variants need to be modified over time [185]. Throughout the life cycle of software systems, the creation of different product variants is unavoidable due to the diversity of functional and non-functional customer requirements in different market segments, usage scenarios, and platforms, leading to evolution in space. In the time dimension, each product variant continuously evolves, resulting in multiple revisions of the variant. Evolution in this dimension is the result of customers requiring enhancements, bug fixes, or scalability issues requiring implementation changes [121].

A software product line (SPL) can encompass evolution in space by systematically managing and tailoring many variants of a software system. An SPL consists of a platform with a set of defined and managed features, each implementing functionality and behavior visible to the end-user [154]. The SPL approach initially requires high upfront investment compared to traditional single system development, but in the long run pays off with high numbers of features and product variants [185]. SPLs also evolve over time due to bug fixes, refactoring, or enhancing modifications of features or the code of existing variants [71, 132].

Features of product variants derived from an SPL can be modified to quickly meet customer demands. However, if modifications are not propagated to the SPL platform, they can hardly be reused in other products. For instance, if a particular change is required for a new hardware device acquired by a customer, reusing this new version of the feature in other variants can rapidly become cumbersome. As a result, software companies not only have to deal with different product variants but also with different revisions of product variants and even different revisions of features over time. Analogous to sequential versions of software systems, i.e., revisions [104], we adopt the concept of feature revisions as implementation modifications over time that lead to new feature versions. Despite this practical need of different feature versions, there is currently no straightforward solution for the challenging problem of integrating the management and evolution of system families in space and time [21].

Nowadays, software engineers use distinct approaches and tools to deal with these dimensions of evolution. An example is the combination of an SPL with a version control system (VCS) providing capabilities to track changes [31]. However, developers need to combine additional mechanisms and tools when evolving system families in space and time. Variability mechanisms are often specific for realizing variability in certain types of artifacts, e.g., AspectJ for Java or preprocessor annotations for text files [104]. A well-known example is the Linux kernel, which combines several variability management techniques [30] to provide an integrated software platform keeping variability information consistent across different types of artifacts [21]. Linux relies on a variant-aware build system [24], an interactive configurator tool [179], and a variability model representation [25]. In addition to its build rules encoded in Makefiles, the Linux kernel is implemented with preprocessor directives to customize its features and to control the compilation of entire source files or fragments [150]. Still, the system is hosted in a VCS to keep track of changes over time[1].

Most existing mechanisms for variability management have a strong impact on software development, as they require adding annotations to the source code, or assume certain programming paradigms (such as feature-oriented or aspect-oriented programming). These variability mechanisms require manual placement of variation points and their concrete realization is usually specific for a certain type of artifact (e.g., AspectJ for Java) [104]. The vast majority of industrial SPLs are realized with preprocessor directives [8, 119]. However, they do not support revision management, which is usually handled by VCSs to preserve the evolution history [26]. Although preprocessor directives can be used in combination

[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux

with VCSs, they have received strong criticism regarding the separation of concerns, error proneness, and code obfuscation [119]. Furthermore, preprocessor directives are limited to managing system variability of textual files. System families, however, rarely consist of only a single type of artifact [99]. Furthermore, while VCSs support *evolution in time* at the file or directory level, they do not provide adequate support for *evolution in space* as shown recently [21, 85, 99, 104]. Thus, the current combination of tools in practice, unfortunately, does not allow to comprehensively and uniformly handle variants and revisions [71, 104, 127].

Variation control systems (VarCSs) have been proposed to address these challenges, as discussed in a recent survey [104]. A VarCS supports system development based on features, reducing the complexity of changing variants and easing the maintenance and evolution of revisions by alleviating developers from manually editing variation points and integrating the changes [104]. For instance, the VarCS ECCO[2] supports the evolution of arbitrary types of artifacts [134] based on plug-in architecture [104]. However, developers are still concerned to replace popular and mature mechanisms, such as the combination of SPLs and VCS, with VarCSs providing proprietary repositories and unfamiliar operations. That is why annotation-based preprocessors combined with VCS are still the most popular variability mechanism [104].

Based on the limitations and needs for properly dealing with the evolution of systems in space and time at the level of features, this paper extends previous work [134] on feature revision location implemented in ECCO to recover information of the system evolution over time at the feature level. We present and evaluate *ECSEST* (**E**xtraction and **C**omposition for **S**ystems **E**volving in **S**pace and **T**ime), an approach for aiding system evolution in space and time by locating feature revisions and composing variants with different combinations of features and their revisions. *ECSEST* thus not only supports the analysis of software systems evolving in space and time by locating feature revisions, but also allows the composition of new products by using the located feature revisions.

As a novelty to support the composition of new products, *ECSEST* introduces a strategy to provide hints about feature interactions and possibly missing or surplus feature revisions for a configuration. This eases system development and the composition of new products based on new combinations of features and their revisions. Hence, *ECSEST* can aid the evolution of software families at both levels of domain engineering and application engineering [8]. For example, it supports the domain implementation and product derivation by reusing and combining artifacts that correspond to feature revisions.

Specifically, in this paper we extend our previous work [134] as follows:

- *Composition of product variants based on feature revisions:* We present an approach for composing variants and provide further details of the feature revision location. We include an illustrative example to show how our approach supports system evolution in space and time and how it is implemented in ECCO[2]. Additionally, we further extended our approach with hints showing possible conflicts and interactions between feature revisions when composing new configurations.

- *Support for C language artifacts:* We developed a new adapter, i.e., a new ECCO plug-in, using a fine-grained tree structure to perform feature revision location, while our previous work [134] was using a text plug-in. Our new plug-in improves the analysis of artifact equivalence when computing traces by analyzing differences in the abstract syntax tree (AST) of feature revisions. Thus, it also enables the evaluation of the evolution of C source code. Although our approach is independent of artifact type, the new plug-in makes our approach easily adoptable in practice, given the high number of SPLs implemented in C.

[2] https://github.com/jku-isse/ecco

- *Evaluation with additional systems and more feature revisions:* We extended our empirical evaluation by applying our approach to more systems from different domains. We now locate feature revisions and compose new configurations with the located feature revisions from six systems. Our analysis includes more points in time, leading to more feature revisions and more product variants. We thus mined more Git commits of more C preprocessor-based systems and included more system variants in our replication package[3].

- *Enhanced analysis:* To evaluate the approach of this extended work, we now compute the runtime performance of composing variants with feature revisions besides precision, recall, and extraction time. Further, we computed new metrics for the hints retrieved when composing new products. These metrics indicate conflicts and interactions when composing product variants with different combinations of feature revisions. We computed further metrics allowing an in-depth analysis of feature evolution at the implementation level. These metrics count the different AST nodes used by our plug-in to store the artifacts of feature revisions.

The remainder of this paper is structured as follows. Section 2 presents background information. Section 3 discusses the motivation for our research and the problems we aim to address. Section 4 explains our *ECSEST* approach. Section 5 presents the research questions of our evaluation, as well as the methodology, subject systems, the metrics used in our experiments, and relevant implementation aspects. Section 6 summarizes and discusses the results. Section 7 discusses threats to validity and Section 8 presents related research. Finally, Section 9 concludes the paper and outlines future work.

2 Background

Variability in space and time is the consequence of collectively developing and maintaining families of software systems [171]. Software systems need to evolve due to developer mistakes and unpredictable future requirements. Variability in space means that different co-existing functional assets of a software system exist at a specific point in time. Variability in time means that different revisions of a functionality, i.e., an asset or a set of assets, exist at different points in time [154]. This section discusses existing concepts, approaches, techniques, and tools to deal with evolution in space and/or time.

Software Product Lines. SPLs provide systematic reuse of assets through the development of a common core and a set of features satisfying customer needs of a particular market segment [30]. The main advantages of systematic reuse in SPLs are the reduction of development costs and the time-to-market, and an increase in software quality [117, 154]. However, adopting an SPL strategy requires expensive up-front investment [114] for defining the SPL scope and product portfolio offered by a software company [156]. Further, it is necessary to define which set of features and which set of domain artifacts can be reused for product composition.

A product, therefore, is composed of a set of artifacts that realize the set of features constituting a valid software system [46]. A new product variant is needed when no existing variant implements the requested features, or when some of the features were modified to update existing variants. Thus, the composed product variant will contain the same features, but not the same implementation as before [59]. In this context, feature location techniques can ease evolution and maintenance tasks [20].

Feature Location. Features are the building blocks of SPLs and are defined as a user-visible functionality of the system [8]. Feature location aims at finding the artifacts responsible

[3] http://doi.org/10.5281/zenodo.4555199

for implementing specific system functionalities. Additionally, feature location is used during the incremental change process to determine where a change should be done in the code and to find the affected code. Thus, feature location techniques have been used for maintenance and evolution tasks, as well as for analyzing the impact of changes [40]. Feature location has received significant attention in the research community and many (semi-)automated techniques have been proposed [13, 34, 166], which can be classified into four categories: (i) dynamic feature location techniques examine the system during runtime and retrieve feature information through execution traces of constructed scenarios, where the feature to be located is exercised [34]; (ii) static feature location techniques rely on the source code structure to find feature code [40]; (iii) textual feature location techniques use textual analysis to find feature code, such as information retrieval and natural language processing analysis [34]; and (iv) hybrid feature location techniques combine several strategies [40, 124, 128]. Our approach adopts static analysis to compare the artifacts and feature revisions of existing system variants (see Section 4).

Version and Variation Control. VCSs have been used to manage system evolution in time. A version control system, a.k.a revision control system, tracks incremental versions (or revisions) of files and directories over time [31]. VCSs allow the implementation of systems in a collaborative way, i.e., a system can be developed in parallel by multiple developers who can later explore the change history. Parallel development of software features is commonly handled in VCSs either with branching mechanisms or optimistic methods, such as copy-modify-merge, workspaces, and transactions [111]. However, as explained, developers not only have to maintain and evolve revisions of a software system but also need to maintain different products of an SPL [32]. Therefore, the evolution of software systems can be characterized with a two-dimensional view with variants incrementing along one axis and revisions incrementing through time along the other axis [111].

Cloning variants via branching or forking mechanisms of VCSs offers only limited support for system families because variations are not managed at a fine-granular level based on features or a similar concept [104]. As a consequence, multiple products need to be maintained, which leads to high maintenance efforts [171]. Even branching models [111] for feature development require developers to manually edit variation points and to manually integrate changes. Further, as the feature is developed in isolation, it is no longer available once merged into the system branch. Hence, branching mechanisms do not offer adequate support for understanding and maintaining the evolution in space and time of system variants.

VCSs and SPLs are thus widely used in combination to support variability and evolution [21]. Annotation-based SPLs combined with VCSs allow customizing different products with preprocessor directives. However, with this variability mechanism features can be delimited only in text files, and an SPL rarely consists of only a single type of artifact [104]. Furthermore, VCSs enable the versioning of the whole platform and can recover and keep track of changes of lines and files but not at the level of features [21]. Therefore, maintenance and evolution tasks require manual analysis of tangled features in multiple files and blocks of code. This is a cumbersome and complex task since preprocessor directives are error-prone and hamper code comprehension [119, 127].

Some VarCSs offer support for feature development of software systems over time by transactionally editing and automatically integrating the features back into the variant-rich system [104]. In a survey, Linsbauer et al. [104] identified six VarCSs that can offer visible operations, such as externalization, modification, and internalization in a transactional way. Three VarCSs support revisions of the whole system, while only ECCO supports feature revisions [134]. It also provides support for different programming language and artifact types and can thus version any kind of artifact. Thus, ECCO has capabilities to aid the maintenance and evolution of a software system family at the level of features with no extra costs. Previous studies [51, 52, 56, 128, 129, 134] already showed satisfactory evaluation

results using ECCO in the context of SPLs and software system evolution.

3 Motivation

In an SPL the evolution in time is more complex than evolving single variants. For instance, developers need to consider all variants at the same time when using preprocessor directives [121]. Although annotation-based SPLs can be versioned in VCSs, the evolution in time is tracked for the whole platform at coarse granularity [21, 104, 127, 131]. Even if systematic reuse is realized by annotation-based SPLs in VCSs, manual analysis and propagation of changes of feature revisions in different releases of a system are highly challenging. An example can be seen in SQLite[4], a C-language library implementing the most widely used database engine in the world. The feature `SQLITE_TEST` was modified for the release *branch-3.9*[5]. The same set of changes, i.e., the feature revision `SQLITE_TEST` committed in the release *branch-3.9*, had to be propagated to three newer releases: *branch-3.18*, *branch-3.19*, and *branch-3.22*. This example confirms that features have different implementations, with different behaviors at different points in time, which is of interest for developers combining different revisions of a feature in existing configurations.

We illustrate the challenges of evolving the source code of a real system implemented with preprocessor directives in Git VCS. For that, we consider LibSSH[6], a C multiplatform library implementing the SSHv2 protocol on the client and server-side. We used our mining tool [131, 132] to retrieve LibSSH's feature revisions from all 5022 commits, covering 48 releases and around 16 years of development. The analysis shows significant system evolution in both space and time. Over the system life cycle, 511 features were introduced, 302 were changed at least once, representing a total of 6242 feature revisions, also including the changes of the system core as feature revisions. Dealing with all releases of a system and considering the huge configuration space with feature revisions leads to complex maintenance tasks. Usually, multiple features change in a single commit, and commit messages not always reflect the changes performed [67]. Finding which parts of the source code of specific annotated features are causing problems and should be changed requires deep developer's knowledge, in particular if other developers do not comprehend earlier design decisions [89, 141], mainly when many developers are involved in open-source projects.

Regarding feature evolution, we present an analysis of the features with the highest number of revisions in the LibSSH system. The feature `WITH_SERVER` changed in 21 releases, resulting in a total number of 241 feature revisions. In this case, 21 system releases contain this feature, i.e., each possible configuration including this feature could have 241 different implementations, if we consider all commits of its development. However, even if a developer has to analyze a smaller number of feature revisions, the code has to be manually analyzed and retrieved. We analyzed some of the commits changing `WITH_SERVER`: usually multiple files and lines of source code changed in a single commit and also between releases of the system, thus resulting in different implementations of the feature at different points in time. Some of the changes represent bug fixes, e.g., commits 3b8c4dc7[7] and 9546b20d[8], some represent new system functionality, e.g., commits b9b7174d[9] and 9b2eefe6[10], implemented by this feature, some represent deletions of functionality of this feature, e.g., commits 55846a4c[11] and 636432e4[12]. Furthermore, we analyzed how many files and lines were part

[4] https://www.sqlite.org/index.html
[5] https://www.sqlite.org/src/info/7b4583f932ff0933
[6] Analysis based on all commits of all releases of LibSSH: https://gitlab.com/libssh
[7] https://gitlab.com/libssh/commit/3b8c4dc
[8] https://gitlab.com/libssh/commit/9546b20
[9] https://gitlab.com/libssh/commit/b9b7174
[10] https://gitlab.com/libssh/commit/9b2eefe
[11] https://gitlab.com/libssh/commit/55846a4
[12] https://gitlab.com/libssh/commit/636432e

of the feature implementation in the first and last releases of the system. The feature was first added in two source code files and comprised 119 lines, while it covered 30 source code files and 4734 lines in the last release.

Now, suppose a software company has a product with a configuration containing a couple of features and needs to use another specific revision of the feature `WITH_SERVER`. If the SPL is implemented in a VCS, a developer may have to rely on documentation describing the kinds of changes performed in different commits of the feature `WITH_SERVER`, which is usually not available. Thus, the developer would have to manually select the feature code, then, copy and paste it to specific configuration releases. To retrieve another feature, for instance the feature `HAVE_SSH1` at a specific point in time from commit 5f7c84f9[13], the developer would need to analyze 29 files, 1339 additions, and 188 deletions. Thus, this manual task is very time-consuming, even more so if multiple features are committed at a single point in time, which is common in practice: for instance, the revisions of the feature `HAVE_SSH1` in the first 50 commits happened together with revisions of 13 other features impacting 73 source code files. Therefore, in this context, *ECSEST* aids maintenance and evolution tasks in annotation-based SPLs in VCSs by retrieving information of the system evolution in space and time at the feature granularity [131].

4 ECSEST Approach

We now present details of *ECSEST* (**E**xtraction and **C**omposition for **S**ystems **E**volving in **S**pace and **T**ime), our approach supporting software systems evolving in space and time. Figure 1 presents an overview of *ECSEST*. We first outline the feature revision location for extraction in Section 4.2 (Step 1 in Figure 1), i.e., to map feature revisions to artifacts from existing software system variants. Locating feature revisions is an incremental process, which receives as input a product implementation and a configuration characterizing its features at a specific point in time. This step creates new traces and refines existing ones in the ECCO repository for every new input variant. We explain our approach for variant composition in Section 4.3 (Step 2 in Figure 1), which requires as input a configuration provided by the user and the output traces stored in the ECCO repository created when locating feature revisions. The variant composition results in the product implementation and a file with hints to help the product completion. In the following, we give details of the data structures and processes of the feature revision location and variant composition.

4.1 Data Structures

Variants (Input). A variant $v \in V$ is a pair (F, A), where F is a set of feature revisions and A is a set of implementation artifacts.

Features and Revisions. Every feature f exists in multiple revisions r, denoted as f_r, where f and r are arbitrary unique identifiers for the feature and the revision, respectively. Two variants v_1 and v_2 with the same feature f have the same revision r of feature f, i.e., feature revision f_r, if the feature is implemented in the exact same way in both variants.

Implementation Artifacts. A variant's implementation consists of a set of artifacts that are organized in a hierarchical tree structure, which we refer to as artifact tree. An artifact can represent a folder, a file, or any other element of a system's implementation. For example, in case of C source code, an artifact could represent a file, a field, a function, a block or a line of code inside a function, a header, or a define statement. We assume that any two artifacts a_1, a_2 can be compared for equivalence ($a_1 \equiv a_2$) as follows: two artifacts $a_1, a_2 \in A$ are equivalent ($a_1 \equiv a_2$) if a_1 and a_2 are equal, e.g., textually equal in the case

[13] https://gitlab.com/libssh/commit/5f7c84f

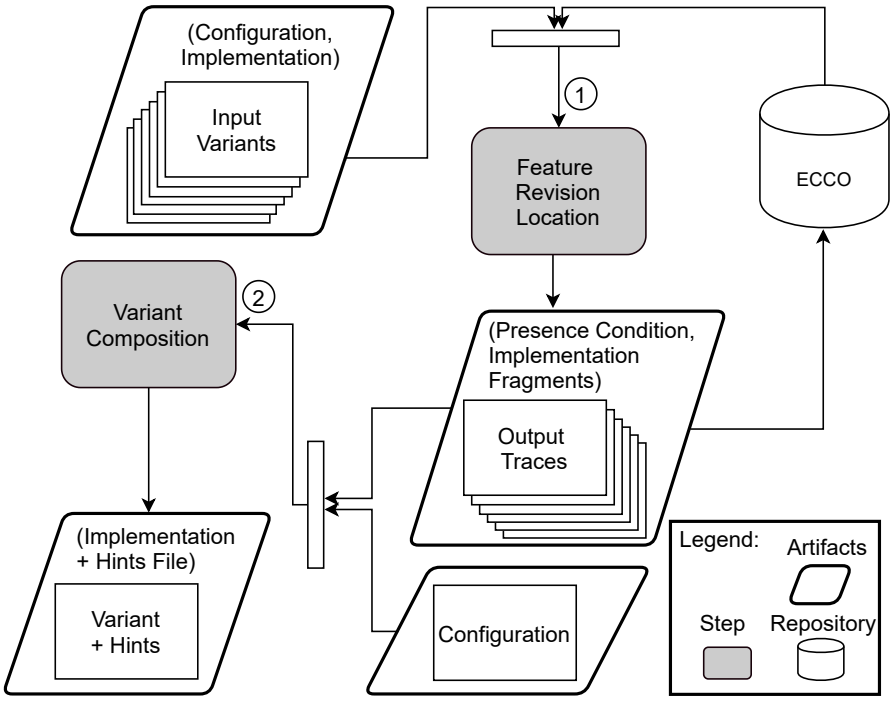


Figure 1: The *ECSEST* approach overview.

of programming artifacts ($a_1 = a_2$) and their parent artifacts are equivalent, i.e., their position in the artifact tree is the same. Thus, for programming artifacts, we compare if two nodes contain the same text-based artifact and if they have the same parent nodes. Here the same rule applies, parent nodes are equivalent if they are syntactically equal.

Traces (Output). The goal of our approach for feature revision location is to compute a presence condition C for every artifact a. The *output* therefore is a set of traces T. A trace $t \in T$ is a pair (C, A) that maps a set of artifacts A to a presence condition C. The traces can abstract where a feature is implemented, e.g., in which files and lines; as well as abstract feature interactions, i.e., artifacts that always appear together when specific feature revisions are present in a configuration. Furthermore, traces show which artifacts are common between feature revisions.

4.2 Feature Revision Location

For extraction of the evolution in space and time, the first step of our approach locates feature revisions (see Figure 1). The *input* of the step is a *set of variants* V, with each variant v consisting of a configuration, i.e., a *set of feature revisions* F, and an implementation, i.e., a set of *artifacts* A. This is an incremental step where the existing traces T (output) stored in a repository are refined for every new input variant. A trace t consists of a presence condition C for a (set of) artifact(s). The input, as we explain in Figure 3, for instance, can be a partial configuration of a variant, containing the set of feature revisions that changed in a specific commit. The commits of a system in a VCS thus represent points in time of new revisions of features. As *output*, our approach retrieves a *set of traces* T', each mapping the implementation artifact fragments to a presence condition. Every artifact is then mapped to a (set of) feature revision(s). This is necessary for composing the variants later on, i.e., when joining all artifacts in a product, the configuration must include the feature revisions containing the artifacts of the required core and functionalities.

ECSEST is independent of the artifact type by using a common structure for data in the location process. Thus, our approach can locate feature revisions in any artifact type if

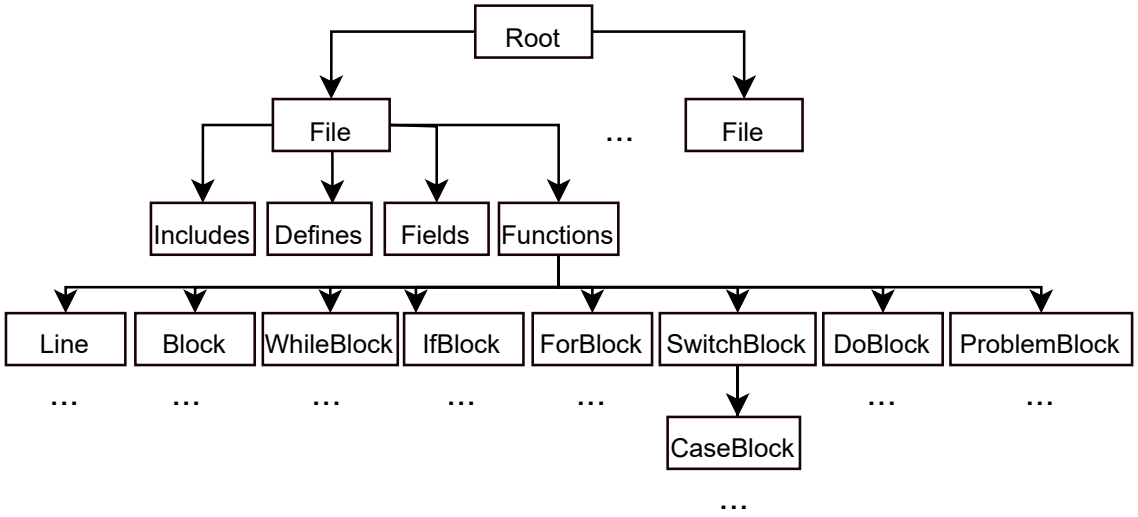


Figure 2: Tree structure of the new ECCO plug-in for parsing C source code.

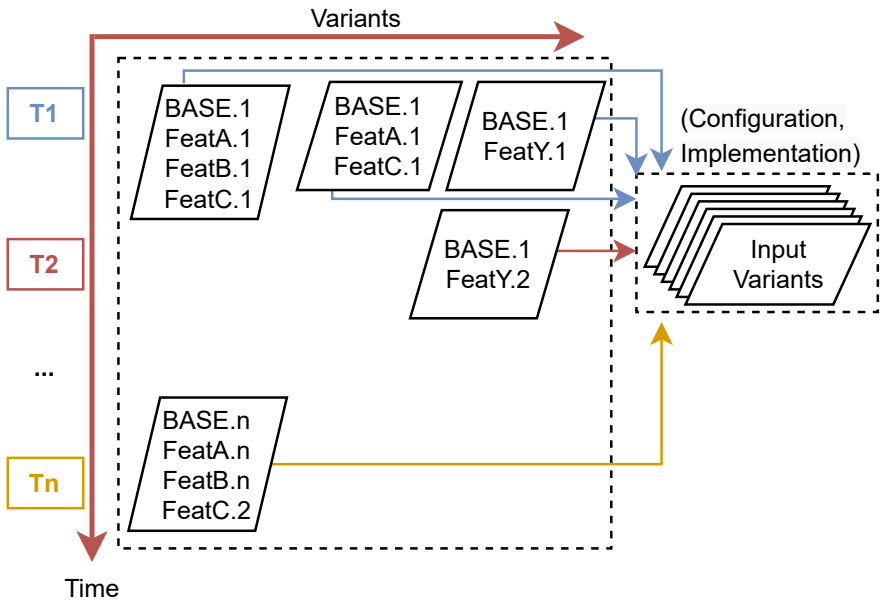


Figure 3: Input variants of our approach for the two dimensions of variability analysis.

provided the input for our internal data structure. The approach can be extended with plug-ins (adapters) as long as different implementation languages and kinds of artifacts can be represented in a tree structure. In Figure 2 we show the tree structure of the new plug-in implemented for parsing C source code artifacts. The tree structure adopted here is due to the type of artifacts of our ground truth variants used to evaluate our approach. For example, our approach already supports plug-ins for Java, text, UML models, PNG images, and LilyPond music artifacts, as shown in previous work [62, 128, 129, 134].

Figure 3 shows the analysis of two dimensions: space and time of existing system variants for obtaining input variants necessary for the feature revision location process. For characterizing different points in time, i.e., when features were changed, numbers are added incrementally to the features' name. Figure 3 depicts such a situation: at a specific point in time T1, the software system was developed from scratch. For T1 we know that the features of the system were in their first revision and thus assigned the revision number *1*. At the second point in time T2, we see a change of a specific feature (`FeatY`), which already existed at T1. Thus, the revision number of the feature was incremented to *2*.

In Figure 3, we can also see that each variant contains a feature called `BASE`, which represents the common code of the variants and represents the core of an SPL, i.e., parts of the system not related to features of the SPL. However, the core of the system is subject to frequent changes, and thus knowing the versions of the common code is also important for managing and evolving the system artifacts. Therefore, the core of the system is also mapped to a feature revision, which can have any name, but is represented here as the feature `BASE`. The feature revision location then analyzes in how many variants a feature revision appears, in how many variants a (set of) artifact(s) appears, and in how many variants a pair of feature revision(s) and a (set of) artifact(s) appear together. In this way, all artifacts are mapped to feature revisions.

Trace Computation

Based on the aforementioned data structures, we now explain how the traces and presence conditions are computed based on the running example shown in Table 1. This example was extracted from a code snippet from file *connect.c* of the commit c65f56ae[14] (Listing D.1). Listing D.2 shows the code of a variant v_1 containing features `BASE` and `HAVE_POLL` (Lines 1, 2, 4, 8, 9, 11, 12 and 23 from Listing D.1) at point T1. Listing D.3 shows the code of a variant v_2 containing `BASE` and absence of the features `HAVE_SELECT` and `HAVE_POLL` (Lines 1, 2, 6, 8, 9, 20, 21 and 23 from Listing D.1) at point T1. Listing D.4 shows the code of a variant v_3 containing the features `BASE` and `HAVE_SELECT` (Lines 1, 2, 6, 8, 9, 14-18 and 23 from Listing D.1 at point T1). Listing D.5 shows the code of a variant v_4 containing `BASE` at point T1 and `HAVE_SELECT` at point T2, where the Line 15 from Listing D.1 changed.

```
1  #include <stdio.h>
2  int ssh_fd_poll(SSH_SESSION *session){
3  #ifdef HAVE_POLL
4      struct pollfd fdset;
5  #else
6      struct timeval sometime;
7  #endif
8      if(session->data_to_read)
9          return(session->data_to_read);
10 #ifdef HAVE_POLL
11     fdset.fd=session->fd;
12     (...)
13 #elif HAVE_SELECT
14     (...)
15     if (select(session->fd+1,&descriptor,NULL,NULL,&sometime)<0) {
16       ssh_set_error(NULL,SSH_FATAL,"select: %s",strerror(errno));
17       return -1;
18     }
19 #else
20 #error This system does not have poll() or select()
21   return 0;
22 #endif
23 }
```

Listing D.1. Code snippet from file *connect.c* from LibSSH in commit c65f56ae.

[14] https://gitlab.com/libssh/commit/c65f56a

Table 1: *Input:* Set of variants $V = \{v_1, v_2, v_3, v_4\}$ and their respective feature revisions $vi.F$.

Variant v_i	**Feature Revisions** $v_i.F$	**Artifacts** $v_i.A$
v_1	$\{\texttt{HAVE\_POLL}_1, \texttt{BASE}_1\}$	Listing D.2
v_2	$\{\texttt{BASE}_1\}$	Listing D.3
v_3	$\{\texttt{HAVE\_SELECT}_1, \texttt{BASE}_1\}$	Listing D.4
v_4	$\{\texttt{HAVE\_SELECT}_2, \texttt{BASE}_1\}$	Listing D.5

```
#include <stdio.h>
int ssh_fd_poll(SSH_SESSION *session){
    struct pollfd fdset;
    if(session->data_to_read)
        return(session->data_to_read);
    fdset.fd=session->fd;
    (...)
}
```

Listing D.2. Variant v_1: `BASE` and `HAVE_POLL` at point T1.

```
#include <stdio.h>
int ssh_fd_poll(SSH_SESSION *session){
    struct timeval sometime;
    if(session->data_to_read)
        return(session->data_to_read);
    #error This system does not have poll() or select()
  return 0;
}
```

Listing D.3. Variant v_2: `BASE` at point T1.

```
#include <stdio.h>
int ssh_fd_poll(SSH_SESSION *session){
    struct timeval sometime;
    if(session->data_to_read)
        return(session->data_to_read);
    (...)
    if(select(session->fd+1,&descriptor,NULL,NULL,&sometime)<0){
      ssh_set_error(NULL,SSH_FATAL,"select: %s", strerror(errno));
      return -1;
    }
}
```

Listing D.4. Variant v_3: `BASE` and `HAVE_SELECT` at point T1.

```
#include <stdio.h>
int ssh_fd_poll(SSH_SESSION *session){
    struct timeval sometime;
    if(session->data_to_read)
        return(session->data_to_read);
    (...)
     if(select(session->fd + 1, &rdes,&wdes,&edes, &sometime)<0) {
      ssh_set_error(NULL,SSH_FATAL,"select: %s", strerror(errno));
      return -1;
    }
}
```

Listing D.5. Variant v_4: `BASE` at point T1 and `HAVE_SELECT` at point T2.

Presence Conditions. We compute the presence condition C for every artifact a in the form of a disjunctive normal form (DNF) formula, whose literals are features, i.e., a set of feature revisions as we will show. A DNF formula is a disjunction of clauses, where a clause is a conjunction of literals. We treat presence conditions as a set of such clauses. Every clause can be considered a feature interaction, i.e., a static interaction of the features contained in the clause. This aligns with previous research in feature algebra [107], feature location [102], and the analysis of variable systems [4, 53]. We denote the set of all conjunctive clauses that can be formed given a set of feature revisions $v.F$ of variant v as $clauses(v.F)$.

Whether a clause c is part of a presence condition C for an artifact a depends on five intuitive rules that have already been proven to work properly for feature location [129]. Given two variants v_1 and v_2 of a system:

1. Common artifacts in v_1 and v_2 *likely* trace to common features.
2. Artifacts in v_1 and not v_2 *likely* trace to features that are in v_1 and not v_2, and vice versa.
3. Artifacts in v_1 and not v_2 *cannot* trace to features that are in v_2 and not v_1, and vice versa.
4. Artifacts in v_1 and not v_2 *can at most* trace to features that are in v_1, and vice versa.
5. Artifacts in v_1 and v_2 *can at most* trace to features that are in v_1 or v_2.

In our work, we build upon these rules and extend them to feature revisions. In the following, we first discuss the rules based on features, ignoring revisions for the time being. We now describe the criterion and two resulting equations based on the aforementioned five rules for including a clause in a presence condition, which composes the traces between artifacts and feature revisions.

Criterion for the Inclusion of a Clause in a Condition. For a clause c to be contained in a presence condition C of an artifact a, the artifact a must be contained in every variant $v \in V$ that contains the clause c ($c \in clauses(v.F)$) and there must be at least one variant in V that contains clause c.

$$\begin{aligned} c \in C \Leftrightarrow (\forall v \in V : c \in clauses(v.F) \implies a \in v.A) \wedge \\ (\exists v \in V : c \in clauses(v.F)) \end{aligned} \tag{D.1}$$

Criterion for Likely Clause. Our approach additionally provides a smaller and more specific set of clauses C', that is a subset of C, to which the artifacts are more likely tracing than to others. This is based on our observation that, in practice, presence conditions with a logical OR between features are much less likely to occur than conditions with a logical AND [129]. Therefore, a clause c' is contained in the set of likely clauses C' if all variants that have clause c' also have artifact a (inclusion criterion Equation D.1). In addition, all variants that have artifact a also have clause c' (additional criterion).

$$\begin{aligned} c' \in C' \Leftrightarrow (\forall v \in V : c' \in clauses(v.F) \iff a \in v.A) \wedge \\ (\exists v \in V : c' \in clauses(v.F)) \end{aligned} \tag{D.2}$$

Adding Revisions. Extending the previous ideas to revisions is then straightforward. Only one revision of a feature can be present in any given variant. In other words, if a feature f is present in a variant v, it is present in exactly one revision r. Therefore, the set of revisions of a feature literal in a clause is the union of all revisions r of feature f that were present when the artifact a was present. Literals in clauses of a presence condition now do not refer to single features anymore but to a set of feature revisions.

Steps for Trace Computation. Algorithm 2 shows the steps of the trace computation. This algorithm receives as *input* a set of variants V and computes the sets of all clauses C (Line 2) and all artifacts A (Line 3) in the input variants V. Subsequently, it computes for every artifact $a \in A$ (Line 5) a trace t with conditions C' and artifact a (Line 19) that is added to the set of traces T (Line 20) that is returned (Line 22). The set of clauses C' receives all clauses $c \in C$ that satisfy the inclusion criterion of likely clauses in Equation D.2 (Lines 7-11). If there are no such traces (Line 12) it receives all clauses $c \in C$ that satisfy the regular inclusion criterion in Equation D.1 (Lines 13-17).

Algorithm 2 Trace Computation

```
1:  function computeTraces(V)
2:    C ← ⋃_{v∈V} clauses(v.F)
3:    A ← ⋃_{v∈V} clauses(v.A)
4:    T ← {}
5:    for each a ∈ A do
6:      C' ← {}
7:      for each c ∈ C do
8:        if (∀v ∈ V : c ∈ clauses(v.F) ⟺ a ∈ v.A) then
9:          C' ← C' ∪ {c}
10:       end if
11:     end for
12:     if C' = {} then
13:       for each c ∈ C do
14:         if (∀v ∈ V : c ∈ clauses(v.F) ⟹ a ∈ v.A) then
15:           C' ← C' ∪ {c}
16:         end if
17:       end for
18:     end if
19:     t ← (C', a)
20:     T ← T ∪ {t}
21:   end for
22:   return T
23: end function
```

Now, to better understand the definitions, let us recall the running example (Listing D.1). The computation of traces consists of computing new, or updating existing, ones after the artifacts alignment for equivalence. The variants v_3 and v_4 have equivalent artifacts under the function `ssh_fd_poll(SSH_SESSION *session)`. When the variant is used as input, the comparison for artifacts equivalence is performed between the Lines from Listings D.4 and D.5. Then, as feature revision location is an incremental process, the existing traces in the repository from the equivalent artifacts have to be updated. The updated traces thus include this new feature revision in the clauses of the traces containing the equivalent artifacts. In the incremental feature revision location, after input variant v_4, the old artifact in Line 7 from variant v_3 is traced solely to the feature revision $\texttt{HAVE\_SELECT}_1$. Also, a new trace is created for the new code in Line 7 from Listing D.5 to the new feature revision $\texttt{HAVE\_SELECT}_2$.

Mapping feature revisions to artifacts can be challenging when a set of variants is not sufficient to determine a unique set of traces. We thus have to consider more restrictive traces by adding negated features in presence conditions to represent artifacts of a variant that do not appear when specific features are present. A feature absent in a configuration can influence the implementation of a variant [107].

Our approach uses presence conditions to map artifacts and feature revisions with negated

features when a specific artifact only exists in a variant with a specific feature absent in the configuration. We use logical negation (¬) to express an absent feature. Despite a variant configuration contains a feature either present in a specific revision or simply absent, our approach can trace artifacts with presence conditions containing positive and negated features. The final set of clauses $clauses(v.F)$ contains all positive features and negated features. Negated features in presence conditions are not labeled with a revision, which indicates that a specific artifact only appears in a variant when the feature negated in the presence condition is not present in the variant configuration. On the other hand, presence conditions containing positive features indicate that an artifact is present in a variant if at least one of the clauses is satisfied with the set of feature revisions of a variant configuration. For the last assumption, it does not matter if features of the other clauses of a presence condition are absent in the configuration.

In our example, a developer does not have to indicate the absence of the features `HAVE_SELECT` and `HAVE_POLL` with a logical negation (¬) in an input configuration of a variant (such as variant v_2), as including `BASE` in the configuration is sufficient. However, when preprocessing the variant, our approach computes traces for the specific artifacts that do not belong to the feature `BASE`, i.e., the core of the system, but are part of a variant when a specific feature is not part of the configuration. For example, in an `#if` and `#else` conditional compilation, the `#else` block artifacts of a system will be part of a variant only when a feature of the `#if` is absent in the configuration, similar to an `#if !(Feature)`. However, including `BASE` in the configuration cannot guarantee that the `#else` part will be in the variant as the feature from the `#if` part also has to be absent in the configuration. This is why only adding positive feature revisions in clauses is not sufficient for creating traces to artifacts that are part of a variant only when specific features are present and specific features are absent. In such cases, different possible traces can affect the variants created in the future.

The output of the feature revision location for our running example shown in Table 2 contains all clauses that satisfy the criterion for inclusion, even if initially redundant. For example, the condition in t_1 could be simplified to just $\texttt{HAVE\_POLL}_1$. However, since the input variants were not sufficient to be certain that the actual condition cannot be, $\texttt{HAVE\_POLL}_1 \wedge \neg\texttt{HAVE\_SELECT}$ it is still included in the condition.

Optimization Aspects. We do not consider every artifact individually, but cluster artifacts that never appear without each other in any variant and assign presence conditions to those clusters instead of every individual artifact. For example, since the artifacts from Lines 1-2,

Table 2: *Output:* Set of Traces $T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$.

Trace t_i	**Presence Condition** $t_i.C$	**Artifacts** $t_i.A$
t_1	$\texttt{F3}_1 \vee (\texttt{F3}_1 \wedge \texttt{F1}_1) \vee (\texttt{F3}_1 \wedge \neg\texttt{F2}) \vee (\texttt{F3}_1 \wedge \texttt{F1}_1 \wedge \neg\texttt{F2})$	Lines 4,11-12 (LD.1)
t_2	$\texttt{F2}_1 \vee (\texttt{F2}_1 \wedge \neg\texttt{F3}) \vee (\texttt{F2}_1 \wedge \texttt{F1}_1) \vee (\texttt{F2}_1 \wedge \texttt{F1}_1 \wedge \neg\texttt{F3})$	Line 15 (LD.1)
t_3	$\texttt{F1}_1 \wedge \neg\texttt{F2} \wedge \neg\texttt{F3}$	Lines 20-21 (LD.1)
t_4	$\texttt{F2}_{1\vee2} \vee (\texttt{F2}_{1\vee2} \wedge \neg\texttt{F3}) \vee (\texttt{F2}_{1\vee2} \wedge \texttt{F1}_1) \vee (\texttt{F2}_{1\vee2} \wedge \texttt{F1}_1 \wedge \neg\texttt{F3})$	Lines 14,16-18 (LD.1)
t_5	$\texttt{F1}_1$	Lines 1-2,8-9,23 (LD.1)
t_6	$\texttt{F1}_1 \wedge \neg\texttt{F3}$	Line 6 (LD.1)
t_7	$\texttt{F2}_2 \vee (\texttt{F2}_2 \wedge \neg\texttt{F3}) \vee (\texttt{F2}_2 \wedge \texttt{F1}_1) \vee (\texttt{F2}_2 \wedge \texttt{F1}_1 \wedge \neg\texttt{F3})$	Line 7 (LD.5)

L = Listing; `BASE` = $F1$; `HAVE_SELECT` = $F2$; `HAVE_POLL` = $F3$.

8-9 and 23 in our running example in Listing D.1 always appear together and never without each other, they are grouped in one artifact cluster instead of treating them individually.

We use counters to evaluate the above criterion for inclusion of clauses (Equation D.1) in presence conditions. For every clause c, we count in how many input variants it was contained, for every artifact cluster a in how many input variants it was contained, and for every pair (c, a) of clause and artifact cluster in how many input variants both were contained together. This has the advantage that it works incrementally, i.e., new input variants can be added whenever necessary, simply by increasing the respective counters. Hence, already computed traces do not have to be recomputed when a new variant is encountered. Instead, the counters are simply increased and the existing presence conditions are trimmed by removing the clauses for which the above conditions do not hold anymore.

Table 3 presents the counters of our running example that match the set of variants V in Table 1. The rows list the nine artifact clusters with the total number of appearances in variants. The columns list (a subset of) the clauses $c_i \in \bigcup_{v \in V} \text{clauses}(v.F)$ with the total number of appearance in variants, sorted by the number of literals, i.e., interacting features first in total without considering revisions, and then per revision. Each cell contains the number of times that the artifact cluster and the clause appear together in a variant. For example, artifacts from Lines 1, 2, 8, 9 and 23 in Listing D.1 appear in four variants. The clause *F1*, which represents the feature revision `BASE`, also appears in four variants. Finally, the artifacts and the clause appear together also in four variants. Therefore, the criterion for likely clauses (Equation D.2) is satisfied. The cells in Table 3 highlighted with gray color indicate pairs of feature revision(s) and artifact(s) that always appear together in input variant(s).

The correct presence conditions shown in Table 2 can only be created with the additional criterion for *likely* clause (Equation D.2). This shows why trivial, i.e., less restrictive presence conditions are not complete and can result in different traces. For example, trace t_1 has a presence condition containing a disjunction of four clauses: ($\texttt{HAVE\_POLL}_1$) $\vee$ ($\texttt{HAVE\_POLL}_1 \wedge \neg$ `HAVE_SELECT`) $\vee$ (`BASE` $\wedge$ $\texttt{HAVE\_POLL}_1$)$\vee$ (`BASE` $\wedge$ $\texttt{HAVE\_POLL}_1 \wedge \neg$ `HAVE_SELECT`). The first clause is obtained by the criterion for likely clause because artifacts from Lines 4,11-12 in Listing D.1 are contained in every variant that contains the feature $\texttt{HAVE\_POLL}_1$ and all variants that have artifacts from Lines 4,11-12 in Listing D.1 also have the feature $\texttt{HAVE\_POLL}_1$. The second clause is also created with the criterion for likely clause, where all variants that have artifacts from Lines 4,11-12 in Listing D.1 also have feature $\texttt{BASE}_1$. Although artifacts from Lines 4,11-12 are not from $\texttt{BASE}_1$, $\texttt{BASE}_1$ was present in the variants

Table 3: *Implementation Example:* Subset (cut off right) of Counters for Artifact Clusters (rows) and Clauses (columns).

		$F1$	$F2$		$F3$	$F1 \wedge F2$		$F1 \wedge F3$
		4	2		1	2		1
		$F1_1$	$F2_1$	$F2_2$	$F3_1$	$F1_1 \wedge F2_1$	$F1_1 \wedge F2_2$	$F1_1 \wedge F3_1$
		4	1	1	1	1	1	1
Lines 1-2,8-9,23 (L1)	4	4	1	1	1	1	1	1
Lines 4,11-12 (L1)	1	1	0	0	1	0	0	1
Line 6 (L1)	3	3	1	1	0	1	1	0
Line 7 (L5)	1	1	0	1	0	0	1	0
Line 15 (L1)	1	1	1	0	0	1	0	0
Lines 14,16-18 (L1)	2	2	1	1	0	1	1	0
Lines 20-21 (L1)	1	1	0	0	0	0	0	0

L = Listing; `BASE` = $F1$; `HAVE_SELECT` = $F2$; `HAVE_POLL` = $F3$.

containing these artifacts. That is what we show in the counter table for storing this information (Table 3), while the third clause is created because all absent features for a specific artifact are negated in our approach. Thus the third clause ($\texttt{HAVE\_POLL}_1 \wedge \neg$ HAVE_SELECT) has the feature HAVE_SELECT negated because this feature does not appear in variants with the artifacts from Lines 4,11-12 in Listing D.1. The fourth clause is the most restrictive condition, combining the previous clauses.

Analyzing the second row of Table 3 shows that the artifacts from Lines 4, 11-12 are present in one variant, where $\texttt{BASE}_1$ is present. However, $\texttt{BASE}_1$ is present in four variants. When looking at the other columns of the second row of the Table 3, we can see that the feature revision $\texttt{HAVE\_POLL}_1$ is also present in one variant, only in the one containing the artifacts from Lines 4, 11-12 from Listing D.1. Therefore, we know the feature revision $\texttt{HAVE\_POLL}_1$ must be traced to Line 4 and our final presence condition contains the feature revision $\texttt{BASE}_1$, as Line 4 appears only once and in a variant containing also the feature $\texttt{BASE}_1$ in its configuration. Thus, the first clause is less restrictive and the fourth clause is the most restrictive condition of the presence condition of the trace t_1. If another input variant would exist with only the feature revision $\texttt{HAVE\_POLL}_1$, and containing Lines 4, 11-12 from Listing D.1, the trace t_1 would be refined and its presence condition would be ($\texttt{HAVE\_POLL}_1$) $\vee$ ($\texttt{HAVE_POLL}_1 \wedge \neg$HAVE_SELECT) $\vee$ ($\texttt{HAVE_POLL}_1 \wedge \neg$BASE) $\vee$ ($\texttt{HAVE_POLL}_1 \wedge \neg$HAVE_SELECT $\wedge \neg$BASE). Having a clause in a presence condition with $\neg$BASE does not mean that this feature must not exist in the configuration of a variant containing the respective trace artifact, but that BASE was absent in a variant where the respective trace artifact appears and was thus not mapped to the artifact. Hence, BASE would not be a mandatory feature for having these artifacts in a variant.

4.3 Variant Composition

For a given configuration containing a set of feature revisions, we compute a checkout operation, similar to the checkout in a VCS [32]. The checkout operation retrieves a working copy of the content from a repository. Thus, the checkout operation joins the artifacts of the feature revisions from a repository in order to compose a system variant.

Composition. We compose a variant v from a set of traces T given a configuration F (set of feature revisions). First, the set of traces T' selected from the set of all traces T:

$$T' = \{t \mid t \in T \wedge clauses(v.F) \cap t.C \neq 0\} \tag{D.3}$$

The final resulting variant v is then given as $v = (F, A)$, where A is the set of artifacts:

$$A = \bigcup_{t \in T'} t.A \tag{D.4}$$

The developer is responsible for selecting a valid configuration to compose a valid variant. We do not consider variability models, which define a set of choices and their dependencies for obtaining configurations [161], but rather focus on feature revision location and variant composition. The composition thus generates a product variant and a file with hints of the traces containing possible surplus and/or missing clauses used to compose the variant. With these hints, developers can analyze which artifacts may need to be added and/or removed for completing the product variant. The file with hints contains the trace identifier (hash code), which can be used to look in the ECCO repository which artifacts belong to a stored trace.

As an example, consider selecting feature revisions $\texttt{HAVE\_POLL}_1$, $\texttt{HAVE\_SELECT}_1$ and $\texttt{BASE}_1$ to compose a variant. The traces t_1, t_2, t_4, t_5 (Table 2) corresponding to these feature revisions will be retrieved from the repository and their artifacts will be joined in order to create a variant. These traces are considered in the set of traces T' because at least one

clause of the presence condition is satisfied (clauses are split with the $\vee$ logical operator). For example, t_1 contains a clause with $F3_1$, which represents the feature `HAVE_POLL` and is one of the features of the configuration. However, this combination of feature revisions has feature interactions, as we can see in Listing D.1. Then, when composing the variant we also get the hints, which we will explain next.

Computation of Hints. To provide the artifacts for a variant, we retrieve the existing traces with at least one clause from the disjunction of clauses in a presence condition containing the feature revision(s) of the configuration. Traces containing clauses with negated features that are in the configuration are not considered in the set of traces T' selected to compose a variant, i.e., the artifacts of a trace will not be included in the variant, while every other trace with the positive feature(s) will. If a configuration contains a feature revision that does not exist in the repository, the composed variant will contain only the artifacts of other existing feature revisions in the configuration. Then, with the composed variant, a hint will be retrieved with *Missing Clauses* because no traces exist for unknown feature revisions in the repository. Also, a missing trace can be retrieved if a feature interaction of a configuration is missing in the existing set of traces because no trace containing the needed implementation yet exists. Therefore, the set of potential *Missing Clauses* H^- for a composed variant v with a configuration F is:

$$H^- = clauses(v.F) \setminus \bigcup_{t \in T'} t.C \tag{D.5}$$

From the set of of selected clauses for composition, we can determine one or more *Surplus Clauses* H^+ as follows:

$$H^+ = \bigcup_{t \in T'} t.C \setminus clauses(v.F) \tag{D.6}$$

From a trace containing multiple clauses, when a clause of a trace contains a feature revision that should be part of a variant and some clauses of the trace contain feature revisions that are not present in the configuration, our approach issues a hint on surplus clauses. This means that all artifacts of the trace were added as artifacts of the variant. However, not all clauses contained in the trace contain only the feature revisions of the configuration, which can result in potential surplus artifacts in the variant from the other feature revisions and show possible feature interactions and/or dependencies due to the artifacts in common. As explained before, our approach computes a presence condition for a (set of) artifact(s) containing a disjunction of clauses. The trace is added to compose a product variant for every presence condition containing a feature revision in one of its clauses, unless there is at least one feature existing in the configuration that is negated in the less restrictive clause of the presence condition. In this case, the trace is not added for composing a variant, which can have missing artifacts.

The hints retrieved by our approach will inform the surplus clauses followed by the trace identifier that can have artifacts surplus and should not be in the variant. The hints retrieved by the new configuration (`HAVE_SELECT`$_2$, `HAVE_POLL`$_1$, `BASE`$_1$) to compose a variant, used as example, contains some of the clauses of the trace *t1* and *t4* as *Surplus Clauses*. Trace *t1* is selected to compose the variant because it contains a clause with `HAVE_POLL`$_1$ and another with `BASE`$_1 \wedge$ `HAVE_POLL`$_1$ from the existing ones in the presence condition of *t1*. However, two clauses are *Surplus Clauses*: (`HAVE_POLL`$_1 \wedge \neg$`HAVE_SELECT`) and (`BASE`$_1 \wedge$ `HAVE_POLL`$_1 \wedge \neg$`HAVE_SELECT`), as they correspond to artifacts that never appear together within the artifacts of all existing revisions of the feature `HAVE_SELECT`. The same happens with trace *t4*, which results in hints for possible surplus artifacts, because some of the artifacts of the input variant containing the feature revision `HAVE_SELECT`$_2$ did not appear with some artifacts of the input variant containing the feature revision `HAVE_POLL`$_1$. The

hints can help developers that need to manually remove part of the artifacts of a specific trace from the composed variant, if the new combination of feature revisions has conflicts when used together.

5 Evaluation

We now present the research questions and the methodology adopted for evaluating the *ECSEST* approach. This evaluation covers both feature revision location in variants of software systems evolving in space and time as well as SPL evolution by reusing the located feature revisions for automatically composing new variants. We introduce the input dataset, i.e., the characteristics of our subject systems. Then, we explain the process adopted to obtain a ground truth dataset used for evaluating *ECSEST*'s efficiency for locating feature revisions and composing variants. Finally, we describe the metrics used to evaluate *ECSEST*.

5.1 Research Questions

The evaluation of *ECSEST* was guided by five research questions (RQs):

RQ1. To what extent do features evolve in space and time? This RQ investigates how features evolve in space and time in real systems to show the practical relevance and implications of our approach for evolving software systems in space and time at the level of feature revisions.

RQ2. To what extent does *ECSEST* support feature revision location of existing variants that evolved in space and time? We evaluate how effective *ECSEST* is for locating feature revisions in existing families of software systems that evolved over time.

RQ3. How effective is *ECSEST* for composing new variants with feature revisions? We investigate if *ECSEST* can effectively compose new variants by joining artifacts traced to feature revisions with our feature revision location approach from a set of system variants that have evolved in space and time.

RQ4. How useful are the hints suggested by *ECSEST* for completing new variants and finding feature interactions when creating new configurations? We estimate how helpful the hints provided by *ECSEST* are for the manual completion of new variants.

RQ5. What is *ECSEST*'s performance for extracting feature revisions and composing variants? This RQ answers the execution time of our approach for performing feature revision location per variant and for composing a variant.

5.2 Method

Figure 4 illustrates the methodology we followed to evaluate *ECSEST*. We investigated both the feature revision location and the composition of variants with feature revisions. We started by mining ground truth variants (Step 1) from feature revisions in preprocessor-based SPLs in VCSs (cf. Section 5.4). We then applied our feature revision location approach to input product variants obtained from the ground truth generation (Step 2). The input variants are the ones generated from existing configurations, and the remaining ground truth variants are the ones generated with new configurations, which we used later on to compare the composed variants with new configurations. For variants with new configurations, we randomly choose a set of feature revisions existing for each point in time. The step of locating feature revisions was performed incrementally with the input variants.

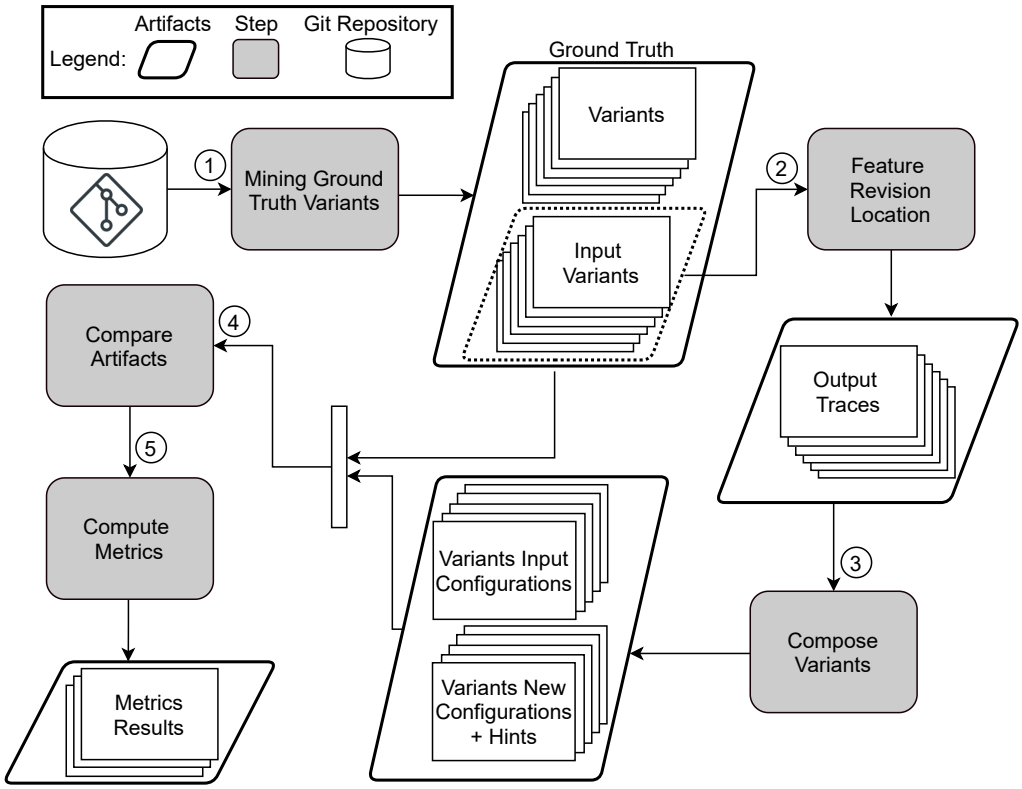


Figure 4: Methodology for evaluating *ECSEST* to support software systems evolving in space and time.

Thus, as long as we had different input variants, we used them for locating feature revisions with *ECSEST*, which continuously created new and/or refined existing traces.

After locating the feature revisions from all existing input variants, we used the computed traces to compose variants with existing configurations and with a new combination of feature revisions (Step 3) by joining the artifacts of the desired feature revisions. Next, we compared the composed variants with the corresponding ground truth variants containing the same configuration (Step 4). The comparison of variants was performed by comparing each composed artifact with each ground truth artifact both file-by-file and line-by-line (cf. Section 5.5). To compute differences of the artifacts of input and composed variants, we implemented a Java program for performing the comparison operations between textual data using a Java diff library[15]. Finally, we computed metrics (Step 5) to quantify missing relevant information or surplus information retrieved in relation to the variants composed from existing configurations (cf. Section 5.5). We also computed metrics for the hints retrieved when composing new configurations of possible surplus or missing source code of feature revisions in new configurations of variants composed.

5.3 Dataset

The evaluation of the proposed approach relies on six open source preprocessor-based SPLs [96] using the VCS Git. Table 4 presents details of the SPLs: (i) Marlin, a variant-rich open-source embedded firmware for 3D printers[16]; (ii) LibSSH, a multiplatform C library implementing the SSHv2 protocol on client and server side[17]; (iii) SQLite, a library implementing an SQL database engine[18]; (iv) Irssi, an internet relay chat client program

[15]https://github.com/java-diff-utils/java-diff-utils
[16]https://github.com/MarlinFirmware/Marlin
[17]https://gitlab.com/libssh/libssh-mirror
[18]https://github.com/sqlite/sqlite

Table 4: Overview of the subject systems.

System	Since	LoC	Commits	Features	Feature Revisions	Input Variants
Marlin	2011	281355	52	151	106	191
LibSSH	2005	110590	400	44	538	577
SQLite	2000	173714	337	36	388	424
Irssi	2007	85325	400	30	414	441
Bison	2002	39904	240	134	272	310
Curl	1999	22490	350	84	422	485

for Linux[19]; (v) Bison, a general-purpose parser generator[20]; and (vi) Curl, a command-line tool for transferring data specified with URL syntax. We try to reduce bias by choosing different application domains. Furthermore, each system has a considerable history of development and use in research [57, 64, 85, 88, 96, 118, 191]. Moreover, we choose systems of different sizes, which we measured by counting the total number of lines of code of their last release (excluding blank lines and comments). We used variants from the first Git commits from the main trunk ordered by the date of each system to avoid bias in choosing a specific interval of commits.

The number of variants we mined (last column in Table 4) is the largest one that we could use as input for each subject system given the memory limitation of the used Java Virtual Machine (JVM) to store and manipulate data. Specifications of the machine used to run the experiments are given in Section 5.5. The number of variants used as input is influenced by the number of artifacts of a system and the degree of artifacts evolution, which determines how many traces and feature revisions have to be stored and manipulated. Therefore, for our evaluation we used input variants from a large number of commits and of more than one release for some of the systems. This is a considerable extension to our previous work [134], since we now apply *ECSEST* to more systems, covering more Git commits and many more variants with different features at different points in time for each system. The variability thus comes from the Git commits. The changes vary from lines in a file to multiple files affected. Some commits introduce new feature revisions, some commits change existing feature revisions, while some commits introduce new feature revisions and change existing ones in parallel. This mining process is presented in Section 6.1.

5.4 Mining Ground Truth Variants from Evolution in Space and Time

Our evaluation needs ground truth variants containing feature revisions, i.e., variants that contain features with different implementations at different points in time. We thus extracted variants of preprocessor-based SPLs in VCSs whenever a feature evolved in time, i.e., was changed via a Git commit [132]. Figure 5 illustrates the time dimension (Git commits) on the y-axis and the space dimension representing the multiple features that originated from multiple variants, which are in the x-axis. The different colors of the edges represent different points in time of features. For every change in a Git commit, we mined feature revisions that were then used to preprocess the variants. Finally, we used the resulting variants as ground truth to represent software systems with different features' artifacts at different points in time.

Although our approach can locate feature revisions in any type of artifact of system variants, even without a variability mechanism, we choose preprocessor-based SPLs in VCSs as input variants because they are widely used to deal with system evolution in space and time [21]. Therefore, every time a feature changes in a Git commit we generate

[19] https://github.com/irssi/irssi
[20] https://github.com/akimd/bison

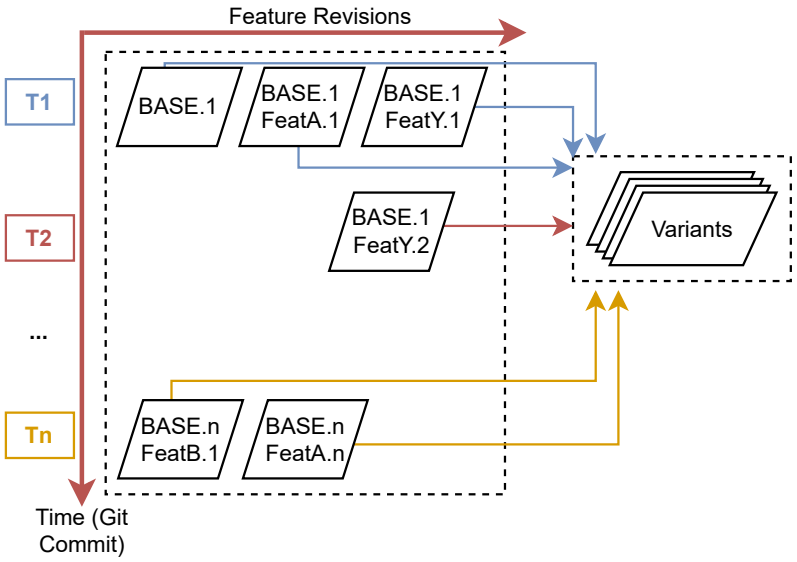


Figure 5: The changes of Git commits represent the evolution of features in time in VCSs. They resulted in a set of feature revisions, which are used to create ground truth variants for the *ECSEST* evaluation.

variants containing a new feature revision for simulating our incremental step of locating feature revisions whenever a feature has a new implementation. In summary, our approach for generating the ground truth consists of getting changes from one commit to another for a set of Git commits. This approach can be computationally expensive but is well suited for precisely locating feature revisions. To cover all changes, a set of configurations is determined by a constraint satisfaction problem (CSP) solver. For each configuration composed of external features, we preprocess the version of the system in the specific commit, which results in an input variant for evaluating *ECSEST*. Next, we explain this process in detail.

We use the example in Figure 6, which contains a corner case with a feature interaction and a feature implication to explain the methodology for mining the ground truth variants. Let us consider the code of the file *main.c* presented in Figure 6 before performing the change in Line 12 at the point in time called T1. Then, changes of a second commit (point in time T2) can be seen in Line 12 of the file *main.c* in Figure 6. We identify the possible features in these two points in time. In this example, three features are introduced in point T1 (`BASE`, `A`, `Y`) and one existing feature changed in point T2 (`Y` revision 2). Based on that, the mining process is as follows.

Identifying feature literals. As our target systems do not have a variability model available, we used the following strategy to identify possible features. We first classified all feature literals, i.e, macros annotated to characterize features of the system along all Git commits analyzed. For this, we distinguished external, internal, and transient feature literals. External feature literals can only be set externally to configure variants from the compiler command line. In Figure 6, the feature literals `A` and `Y` are external. Internal feature literals are defined at some point in the code via a `#define` directive. Thus, we can see in Figure 6, that the feature literals `B` and `C` defined in *main.c* as well as the feature literals `X` and `Z` defined in *header.h* are internal.

We considered feature literals as system features only if they were external in all Git commits analyzed. We cannot ensure that all identified external feature literals are actually features of the system. However, according to Berger et al. [22], features are also used for testing and debugging purposes. In addition, our approach enables the manual setting of system features if the set of features is known. Our ground truth generator approach is

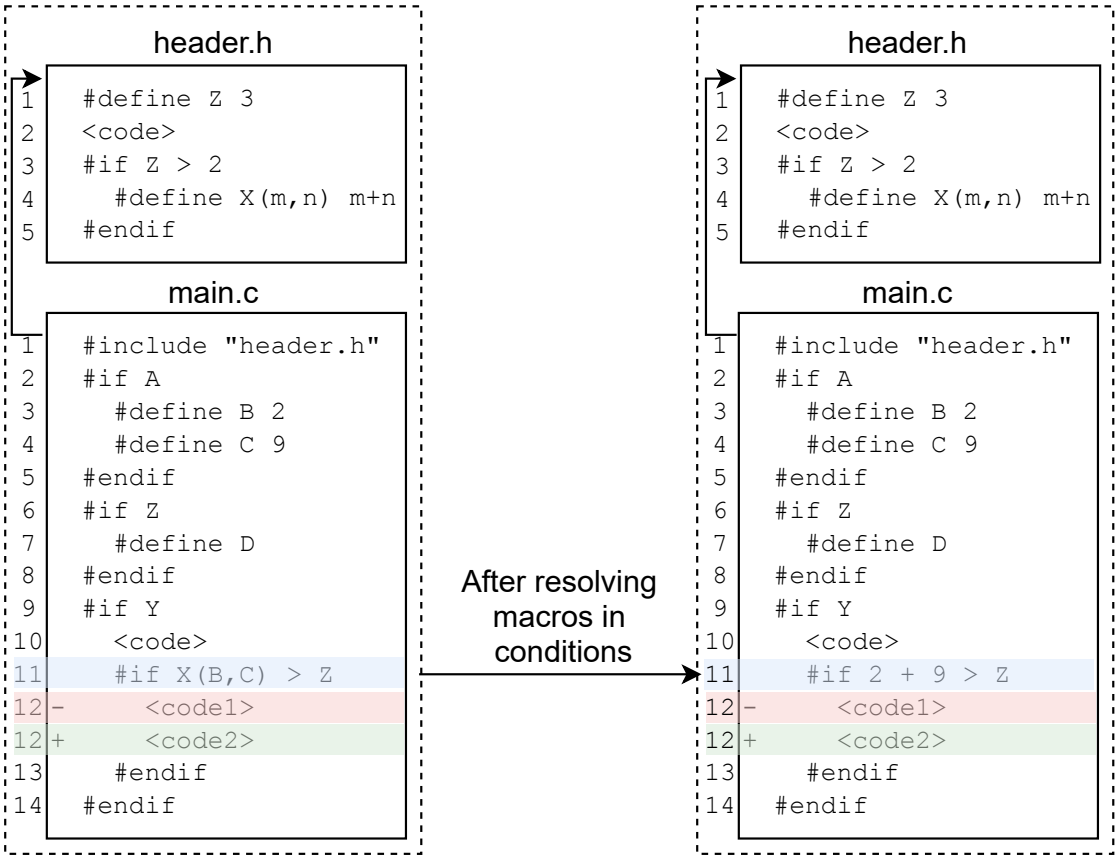


Figure 6: Mining feature revisions from changes in time in preprocessor-based SPLs.

limited to systems that do not consider dependencies in Kconfig and Makefiles such as the Linux Kernel system [132].

Resolving macros in conditions. For each analyzed Git commit, we started preprocessing the annotated code to find macros that can accept parameters and return values. The output of this step is the code from the specific commit with all macros in conditions resolved, i.e., the macro code is expanded to the degree where the conditions of the conditional statements only consist of feature literals. This step is necessary because we need only macros and their values in the expressions of conditional blocks to correctly collect all possible features from conditions. Obtaining these values from expressions and functions is important to build up the constraints and to retrieve a possible solution via a CSP Solver. After expanding macros in conditions, all `#define` and `#include` statements and conditional blocks remained in the code, as they can modify the resulting code of the variants. On the right of Figure 6, we see that the highlighted Line 11 is the only one that changed after this step replacing `#if X(B,C) > Z` with `#if 2 + 9 > Z`.

Computing changes. For each Git commit n we created a tree structure for representing variability in source code, as shown in Figure 7. Files at a certain point in time are represented either by *SourceNodes* or *BinaryNodes*. The *SourceNodes* contain child nodes each with the content of a source code file, e.g., `.c/.cpp`. A *SourceNode* has as a root node `BASE` that emulates the feature `BASE`, which contains *ConditionalNodes* as much as needed to represent each `#ifdef` in a file. *DefineNodes* represent the location in a file of `#define` and `#undef` preprocessor statements, while *IncludeNodes* represent the `#include` preprocessor statements in a file. The tree nodes are used to determine the differences

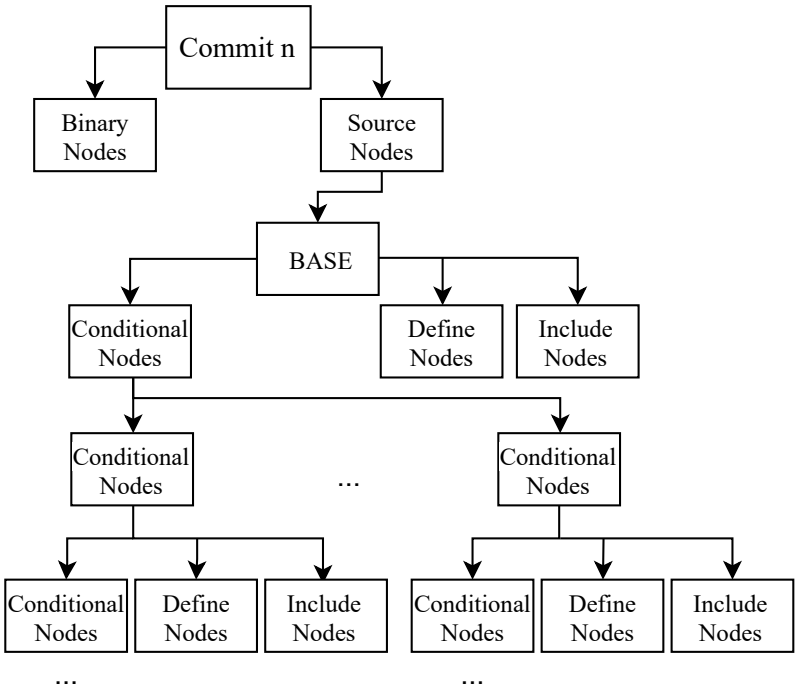


Figure 7: Structure for computing Git commit differences to analyze changes in annotated blocks of code.

between an actual commit and its previous one according to Git-diff[21]. The adopted tree structure has a higher level of abstraction, i.e., for every annotated block, a child stores its content in its respective node category, e.g., conditional nodes, define nodes, and include nodes. This makes our mining process computationally less expensive. We adopted the changes at Git-diff granularity, i.e., files and lines, to be able to easily inspect the correctness of the generated ground truth variants according to changes of features annotated in Git VCS.

The choice of inspecting changes from consecutive commits was to avoid bias in choosing specific commits to generate variants as any change results in new feature revision(s) as input for our approach to generate variants. Therefore, in case of the first Git commit of the project, we consider all files inserted as the difference. From the differences, we can get the tree node reflecting the changes. In case any external feature changed or differences are detected in non-code files, e.g., binary, `BASE` is considered the changed feature, i.e., for every code added/removed in the body of the project that does not belong to an external feature the root feature, i.e., `BASE` is considered as the changed node. Figure 6 shows two files, the header file (on top of the figure indicated by an arrow) and the file containing 14 lines on the bottom of the figure. At point T1 we have these two files, and at point T2 the main file (the file on the bottom of the Figure 6) has been changed in Line 12.

Computing configurations. Every changed node was then used to generate a variant containing the code activated by this node. We used the Choco solver[22] library to provide the first possible solution for a given constraint to activate the conditional blocks. To find a configuration for the preprocessor that activates the desired block of code, we obtain an assignment for all the annotated feature literals that are part of the condition of the block. We then create a set of constraints that are handed over to a solver. The constraints we build consist of three parts, which will be explained using the example in Figure 6. Firstly, we retrieve the local condition, i.e., the condition of the closest conditional block to the

[21]https://git-scm.com/docs/git-diff
[22]https://github.com/chocoteam/choco-solver

changed code. As mentioned before, point T1 is the code of the file *main.c* before the change in Line 12, and point in time T2 is when the change was performed in the code of Line 12 of the file *main.c* (Figure 6). Thus, the logic formula of the local condition in the example at point T2 is: $2 + 9 > Z$. The second part is the global condition of the desired block, which is a conjunction of all parent conditions, i.e., all conditional blocks wrapping the closest conditional block. We obtain it by walking up the tree, starting from the changed node, which in our example results in a global condition with logic formula: $Y \wedge (2 + 9 > Z)$.

The feature implications make the third part used to create and apply a mapping of all internal feature literals to just external feature literals. We thus traverse the tree to build the feature implications. For example, in Figure 6, we can be seen that `A` defines `B=2` (Line 3, *main.c*) and `C=9` (Line 4, *main.c*), and `BASE` defines `Z=3` as there is no conditional block wrapping Line 1 in the file *header.h*. Thus, `BASE` implies *header.h* and the features that activate the code block that changed (Line 12) are `X`, `B`, `C`, and `Z`, which are defined by features `A` and `BASE`. Still, when walking up the file we see that there is an outermost code block with a condition expression involving the feature `Y` (Lines 9-14), which wraps the changed block. The feature implications are mathematically defined as follows: $(A \implies (B = 2)) \wedge (A \implies (C = 9)) \wedge (BASE \implies (Z = 3))$. The conjunction of all these parts, local and global condition and implications, are the logic expression to the problem constraint that can be handed to the solver: $(A \implies (B = 2)) \wedge (A \implies (C = 9)) \wedge (BASE \implies (Z = 3)) \wedge Y \wedge (2 + 9 > Z)$. The solution assigned that satisfies this formula is then: $BASE = TRUE \wedge Y = TRUE \wedge A = TRUE$. We thus know that these features must be selected to include the changed block of code in a variant.

If the solver finds no solution, the part of code we want to activate is dead as no configuration can activate it. If a solution can be found, we validate that all feature literals with assignments are external. If the set of assignments are not empty at this point, we obtain a configuration for mining a variant. Before using these variants as ground truth for evaluating *ECSEST*, it was essential to know what features should be marked as changed for the respective changed node and thus be treated as a feature revision. We assumed the features annotated closest to a change as the ones that caused it. Therefore, we got a solution using only the local condition without any implications. In cases where a local condition contains more than one feature to activate a particular changed block of code, nothing affects the ground truth generator approach because the constraint is built considering all the features of the conditional block. Then, the CSP solution is retrieved according to the constraints and can assign the change to more than one feature. Therefore, depending on the feature interactions in more complex conditional expressions comprising several features, it might happen that a changed block of code is assigned to more than one feature revision.

In case the solution did not retrieve any potentially changed feature, meaning that there were no positive external features in the closest condition, we repeated the same process with the parent conditions until we find a positive external feature from the solution. In the worst case, we reached the node corresponding to `BASE`, which is trivially a positive solution.

Generating ground truth variants. After these previous steps, we generated the ground truth variants by partially preprocessing the code. Finally, the solution found by the Choco solver for the configuration was used to retrieve the variant, which could be used as input for locating feature revisions. Figure 5 illustrates the variants mined with a set of feature revisions from the changes in time T1 and T2.

5.5 Metrics

We present and discuss the metrics we used for the evaluation of our approach. We first computed metrics characterizing system evolution in space and time in real systems to show the need of such approach at the level of feature revisions. Thus, we computed feature revision characteristics showing the feature evolution over time, related to their source code artifacts. We continued by computing the metrics for evaluating the ECSEST approach. Furthermore, we computed metrics to evaluate *ECSEST* for composing new variants with a new combination of feature revisions. Additionally, we also measured runtime metrics to evaluate the performance for locating feature revisions and composing variants.

The *Feature Evolution Metrics* are computed to show the number of new features introduced and the number of features that were changed over the life cycle of a system. Thus, they indicate the feature evolution of the ground truth variants used in our experiments.

- *FeaturesIntroduced*. Number of new features introduced over the Git commits analyzed.
- *FeaturesChanged*. Number of features changed over the Git commits analyzed.

The *Feature Revision Metrics* are computed to characterize the differences of the source code of different revisions of a feature in terms of AST nodes. These metrics represent the variability existing in the ground truth variants used to evaluate our approach. We thus count the number of AST nodes used to represent the feature revisions artifacts by our adapter for C language artifacts. Each of the following metrics counts the number of a specific AST node within the source code of a feature revision.

- *Header*. Number of *header files*.
- *Define*. Number of *defines*.
- *Field*. Number of *field/struct declarations*.
- *Function*. Number of *functions*.
- *If*. Number of *if conditions*.
- *For*. Number of *for loops*.
- *Do*. Number of *do loops*.
- *Switch*. Number of *switch conditions*.
- *Case*. Number of *case statements*.
- *While*. Number of *while loops*.
- *Problem*. Number of *problem blocks* not recognized in the C AST.

Feature Revision Location Metrics. Precision, recall, and F1-score measure how well information is retrieved by a system in relation to the relevant information [190]. They are commonly used to evaluate feature location techniques [34, 116, 129]. In order to assess the effectiveness of *ECSEST* to correctly locate and not miss any relevant artifacts, we analyzed if the stored traces allow retrieving the artifacts belonging to a specific feature revision. We applied the aforementioned metrics by comparing artifacts of feature revisions composed by the traces of *ECSEST* with the artifacts of the ground truth (see Section 5.4). We used two levels of granularity, due to the feature evolution analyzed, and the different kinds of files that existed in the subject systems (C, binary and text files): *file-level* comparison of

two complete files by matching their content; *line-level* comparison of two code files. As the C files from the input variants used for the feature revision location consist of source code after resolving preprocessor directives, the composed variants also contain the C source code files with preprocessor directives resolved. Thus, the comparison is performed on the C source code files after resolving preprocessor directives.

Precision of the file-level comparison is the percentage of correctly composed files, i.e., retrieved files with entire content matching the relevant ones. Recall measures the total percentage of entire matching of all composed files relative to all relevant files. Regarding line-level comparison, precision is the percentage of correctly retrieved lines, while recall is the percentage of matched lines retrieved relative to the total of relevant lines.

- *PrecisionFileLevel*. The percentage of correctly retrieved files in relation to the total retrieved.
- *RecallFileLevel*. The percentage of correctly retrieved files in relation to the total ground truth ones.
- *F1ScoreFileLevel*. The percentage of the weighted average of Precision and Recall at the file level.
- *PrecisionLineLevel*. The percentage of correctly retrieved lines in relation to the total retrieved.
- *RecallLineLevel*. The percentage of correctly retrieved lines in relation to the total ground truth ones.
- *F1ScoreLineLevel*. The percentage of the weighted average of Precision and Recall at the line level.

Hint Metrics. To estimate the usefulness of the hints to complete new variants we used the *ArtifactsRatio* indicating for how many new variants with hints it might be necessary to add and/or remove artifacts. Measuring the *InteractionsRatio* shows the ratio of variants with hints that say there is no trace with this new combination. This can help to analyze feature interactions as two specific feature revisions that never appeared together often cannot co-exist in the same configuration. The *ArtifactsRatio* is used to present how helpful our hints can be for showing possible feature interactions when composing a product with a new combination of feature revisions never used before. Thus, we evaluate if hints with surplus/missing artifacts are the result of possible feature interactions or an invalid configuration. Therefore, the correctness of the approach for composing variants is measured by precision and recall from comparing artifacts of a composed variant with the corresponding ground truth variant.

- *ArtifactsRatio*. The percentage of the number of new variants composed with hints that have artifacts missing/surplus in relation to the total of new variants with hints.
- *InteractionsRatio*. The percentage of the number of new variants composed with hints that have feature interactions and retrieved missing/surplus artifacts in relation to the total number of new variants.

As mentioned in Section 4, the invalid configurations used to compose a variant can retrieve invalid variants due to feature interactions. However, our approach is designed to trace artifacts to feature revisions and use them to compose variants. Thus, our evaluation aims to quantify the correctness of the traces computed and the feasibility of using our approach for composing new variants or new product revisions with feature revisions containing updated implementation. In this way, if artifacts are retrieved correctly, valid

configurations will result in valid variants. Our evaluation does not focus on analyzing if valid configurations are created but on whether our approach can correctly locate feature revisions and compose variants given a configuration with feature revisions. Despite already providing some hints, we need to improve our approach to help users to compose valid and consistent configurations with the evolution over time. We plan to improve our approach in future work to analyze the evolution of dependencies and interactions in the source code of feature revisions, as shown by Feichtinger et al. [48].

Performance Metrics. To run the experiments we used a machine with an Intel® Core™ i7-6700U processor (3.4GHz, 4 cores), 32GB of RAM, SSD storage, and the Windows 10 operating system. Thus, under this capacity circumstances we measured the approach performance:

- *ExtractionTime*. The time in seconds for locating feature revisions, i.e., for extraction of mappings of feature revisions to artifacts from a variant.
- *CompositionTime*. The time in seconds for composing a new variant, i.e., the time needed to retrieve traces from a set of feature revisions and compose their artifacts in order to generate a variant.

Regarding the performance metrics, we are interested only in evaluating *ECSEST* and not the process of mining the ground truth. The mining process was just necessary to create the ground truth and input variants for the evaluation. We thus evaluate our approach supporting the evolution of annotation-based SPLs in VCSs by locating feature revisions and composing variants.

5.6 Implementation Aspects

We implemented *ECSEST* on top of the VarCS ECCO[2] and performed some optimizations to implement the concepts of our approach presented in Section 4.

Feature Interaction Limit. We limited the maximum size of clauses in presence conditions, i.e., the number of feature literals in a conjunction, which corresponds to the number of interacting features, to a threshold based on previous empirical research [51, 53]. This provides a major improvement to the scalability of the approach, as otherwise the number of clauses would grow exponentially with the number of features.

The threshold can be freely configured, however, for the evaluation presented in this paper it was set up to three interacting features, which strikes a reasonable balance between computational effort and quality of results [51, 53]. Considering higher-order feature interactions would yield only very little additional gain while significantly increasing cost, similar to t-wise interaction testing of product lines. Using a threshold of feature interactions limits higher orders of feature interactions in a clause in the set of clauses of a trace.

Artifact Sequence Alignment. The artifact equivalence is performed by an adaptation of the Longest Common Subsequence (LCS) algorithm [35] to perform multi-sequence alignment for comparing more than two variants [51, 103], e.g., if they have the same method whose statements must be aligned.

Artifact Adapters. We keep the approach independent of the types of implementation artifacts by utilizing artifact type specific adapters that are responsible for parsing respective files and generating the generic artifact tree structure consisting of folders, files, and further file type-specific artifacts. The only requirement is that artifacts can be uniquely identified and compared for equivalence. In this work, we used the Eclipse CDT[23], i.e., a C/C++ Development Tooling for implementing the adapter for parsing our target systems artifacts.

[23] https://www.eclipse.org/cdt/

The approach itself is implemented with the Java programming language and the data storage and manipulation depend on the JVM memory. The more nodes have to be created to store uncommon artifacts, the higher is the memory consumption. This is why we use fine-grained parsing to store the artifacts. With this new plug-in, compared to our previous work [134], we can locate more feature revisions because of the reuse of the tree structure nodes for storage and manipulation of traces and feature revisions in the repository.

6 Results and Discussion

This section discusses the results of our empirical analysis of the feature evolution in space and time from the six subject systems. Based on the results and analysis, we provide answers for the five posed RQs.

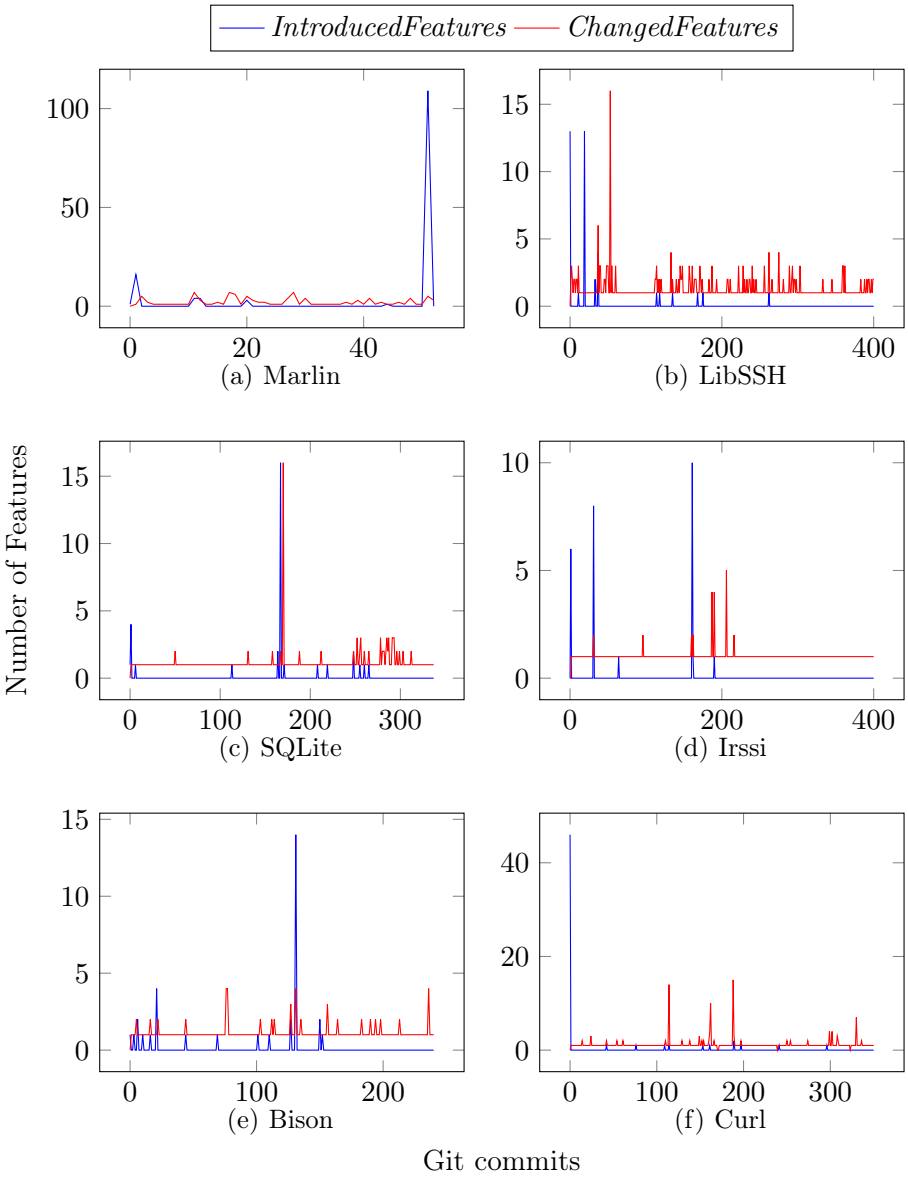


Figure 8: Relation of the number of features introduced and changed over the Git commits analyzed for each system.

6.1 Feature evolution in space and time

Figure 8 summarizes the evolution in space and time, i.e., the number of features introduced and changed over the range of Git commits for each of the six systems. The blue line represents the evolution in space, i.e., the number of new features introduced, while the red line represents the evolution in time, i.e., the number of revisions of already existing features. First, regarding evolution in space, Figure 8(a) shows the feature evolution of the Marlin system. After the product started with Git commit #1 with just feature `BASE`, the second commit introduced 16 new features. Then, later in Git commit 51, 109 new features were introduced. Furthermore, additional new features were included in four Git commits. For the LibSSH system, shown in Figure 8(b), the initial version started in Git commit #1 with 13 features. Then, there were ten changes affecting the evolution in space in the 400+ commits analyzed resulting in a total of 39 new features. In case of SQLite, shown in Figure 8(c), after the first Git commit introducing only the feature `BASE`, four features were added in the second commit. Along the commits analyzed, within 11 commits 33 new features introduced. Regarding the evolution over time, there were 29 Git commits with feature revisions. In the Irssi system (Figure 8(d)), six features were added in Git commit #2, eight in Git commit #32 and 10 features in Git commit #162. In three other commits evolving the system in space, only one feature was introduced. Over time, usually, one feature changed, with exceptions in eight Git commits, ranging from two up to five features introduced in the same commit. Regarding Bison, shown in Figure 8(e), features were introduced in 14 Git commits. The evolution over time resulted in 270 feature revisions over the 241 commits analyzed, ranging from one to four revisions per commit. In the Curl system (Figure 8(f)), 46 features were introduced in the first Git commit of the project. The next evolution in space happened in 10 Git commits. The evolution over the 350 Git commits resulted in 422 feature revisions, with the highest number of revisions (15) in a single commit happening in Git commit #189.

From the analysis of system evolution over time of these six systems, we observed that many features change over time besides the feature that represents the core of the system, i.e., the feature `BASE`. For Marlin, 22 different features changed in the Git commits analyzed. In the LibSSH and Curl systems, 30 features evolved over their Git commits analyzed. For the SQLite system, 24 features changed, and for the commits analyzed in the Irssi and Bison systems, 12 and 13 different features changed, respectively.

The evolution over time by feature revision can strongly impact product configurations of configurable software systems. For example, LibSSH had six features changed and four introduced in Git commit #38. This evolution in space and time in a single commit makes engineering tasks complex. Suppose an engineer needs to recover an older version of a specific feature introduced before commit #38, keeping the change of other features. This would require great effort and would be error-prone since other current variants of the LibSSH system could be still using the newer version of that feature. Considering these six subject systems with different domains, we can see that features have been introduced and changed frequently during their development, which would benefit from a mechanism to handle feature revisions, such as the *ECSEST* approach.

Regarding feature evolution, we considered not only functional features because according to Berger et al. [22], in the industrial systems features are also needed for testing, debugging, build, optimization, deployment, simulation, or monitoring. These atypical features can be introduced for the optimization of non-functional aspects. Therefore, features such as `YYDEBUG` from the Curl system (shown in Figure 9(f)), indirectly realize customer requirements. An interesting example of a feature revision that have to be reused in previous revisions of the SQLite system can be seen in the feature `SQLITE_TEST`[24], which evolved meaning that its change had to be applied in four releases of the system: *branch-3.9*,

[24] https://www.sqlite.org/src/info/7b4583f932ff0933

branch-3.18, *branch-3.19* and *branch-3.22*.

In Figure 9, we show the evolution of the C artifacts in the AST of the feature revisions that most often changed over the Git commits analyzed for each target system without considering the `BASE` feature. The number of AST nodes of the feature `HAVE_SSH1` from the LibSSH system (Figure 9(b)) has constant changes, but the number of fields increased significantly in its second revision. This is also the case for the feature `SQLITE_TEST` from SQLite and the feature `MSDOS` from the Bison system in revision 6 (Figures 9(c) and (e)), and for the feature `ADVANCE` from Marlin in revision 4 (Figure 9(a)). The evolution over time of the feature `SQLITE_TEST` from the SQLite system (Figure 9(c)) shows an increasing number of AST nodes up to its sixth revision. After that, the AST nodes remain constant in terms of numbers per node type, and in case of field nodes, the number decreases in the twelfth revision of the feature `SQLITE_TEST`. The feature `__GNUC__` from the Irssi system (Figure 9(d)) has not been changed regarding the number of AST nodes over the three revisions, with the exception of the number of problem statements/blocks and defines AST

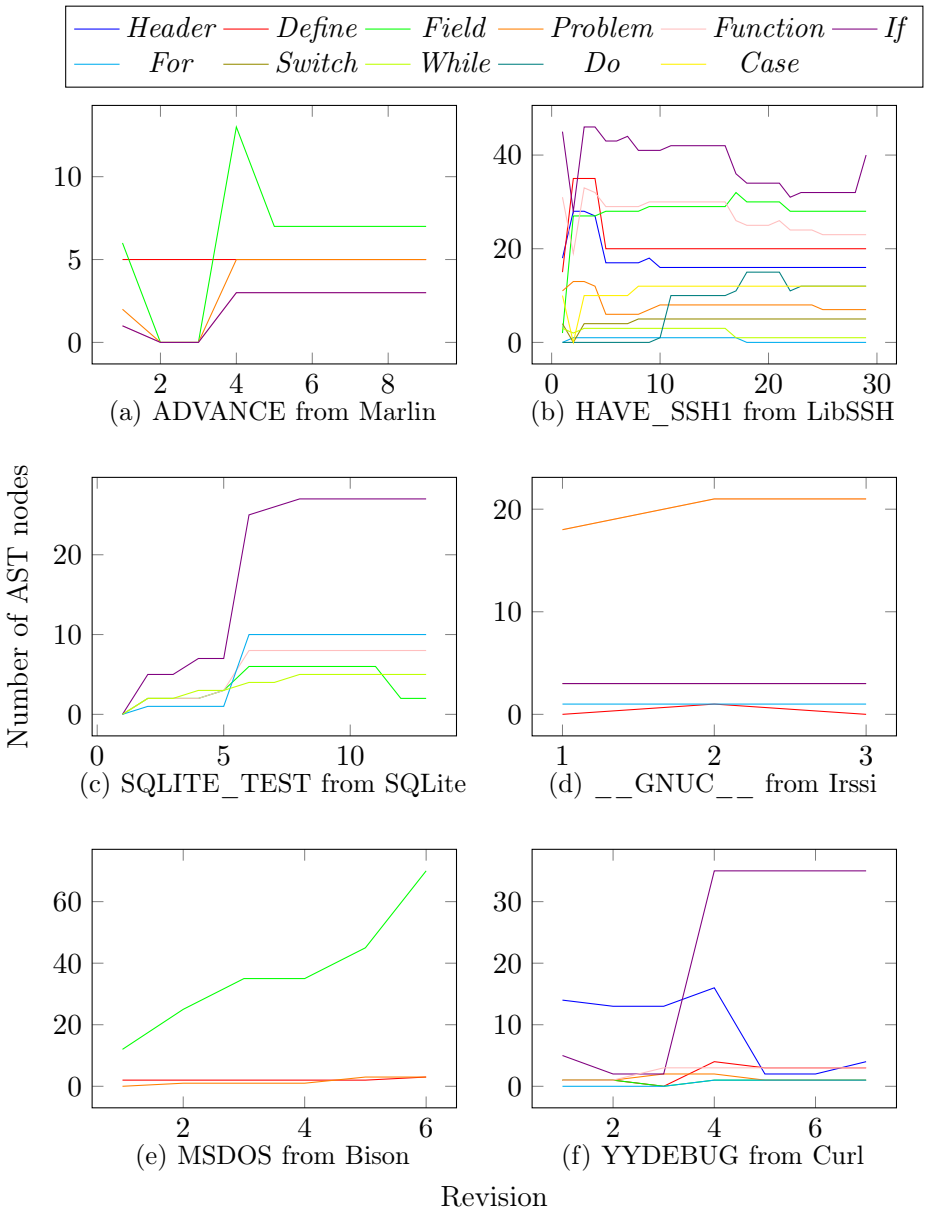


Figure 9: Evolution over time of the number of AST nodes from feature revisions of each system.

nodes, which increased during the second revision. For example, for the feature `YYDEBUG` from the Curl system (Figure 9(f)) the number of header files, define, field, if, and for AST nodes increased in its fourth revision. In the fifth revision, for example, 14 header files were removed from its implementation.

Knowing which commits have new feature revisions can make it easier for developers to find a specific revision of a feature. The chart on Figure 9(a), for example, shows that the fourth revision of `ADVANCE` from the Marlin system substantially changed compared to its predecessor. We analyzed the Git commit 094afe7[25] and saw from the commit message that a merge was performed. A developer added 12 new files, changed eight, and removed two files that affected the source code of the feature `ADVANCE`, and hence, the behavior of how the movement of the printer is done with linear acceleration. For every new revision of this feature, the movement is affected. In the ninth revision (Git commit 65934ee[26]) many changes were performed in the planner source code, which influences the buffers movement commands and manages the acceleration profile plan.

We now use the feature `YYDEBUG` from the Curl system as an example. The revisions of this feature are from five different releases of the system. This feature is used for debugging purposes and contains many *fprintf* calls to print the debug messages in a file. Thus, depending on the revision selected, different debug messages are printed. If developers want to use the revision of this feature to get more debug messages and combine it with features of another release, it can be supported with *ECSEST* instead of manually retrieving the Git commits of the feature revision and release. Besides that, developers will need to work manually copy, paste and modify the code of the release with the code of the desired feature revision.

With the analysis of feature revisions and their number of AST nodes over time in Figure 8, we can see that feature evolution happens over time, because the source code changes cover more than single lines and also affect header files, defines, fields, conditions, and loops. Thus, if a developer wants to use, for example, an older revision of a specific feature with the previous revisions of other features, *ECSEST* eases the process of combining features with different revisions to obtain their different source code, thus producing different system behaviors.

RQ1. To what extent do features evolve in space and time? *We could see in the mined ground truth variants that multiple features are substantially changed and introduced in single commits, showing the need for our approach support. By analyzing the AST nodes of the feature revisions with ECSEST, we could also see that the results of the feature evolution are meaningful.*

6.2 Locating feature revisions.

The results of *ECSEST* for locating feature revisions are shown in Table 5. The precision for the six subject systems was 100% at file level and 99% at line level, except for the Bison system, which was 100% at line level too. The recall values ranged from 92% up to 99% at file level and were 99% at line level for all systems. The values of F1, which consider both precision and recall, are between 96% and 99% at file level and 99% at line level, showing that *ECSEST* reliably locates feature revisions by a given set of variants in different space configurations and in many points in time. Overall, when computing the average of all systems shows that precision and recall stay above 97% at file level and 99% at line level.

Although in this work we developed an adapter more fine-grained for the specific syntax of the C programming language, there are still issues: some deleted lines are shown in the example from a code snippet in Listing D.6, with comments after a source code statement. Our adapter was not developed to capture this kind of comment split into multiple lines.

[25] https://github.com/Marlin/commit/094afe7c1065d5663628b389f27687a5f465abb8
[26] https://github.com/Marlin/commit/65934eee9c6ae792c708bc1cea9996c8a5df67f5

Table 5: Average precision, recall and F1-score metrics of composed artifacts per system.

Subject System	Granularity	Precision	Recall	F1-Score
Marlin	*FileLevel*	1.00	0.95	0.98
	LineLevel	0.99	0.99	0.99
LibSSH	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	0.99	0.99
SQLite	*FileLevel*	1.00	0.97	0.98
	LineLevel	0.99	0.99	0.99
Bison	*FileLevel*	1.00	0.99	0.99
	LineLevel	1.00	0.99	0.99
Curl	*FileLevel*	1.00	0.92	0.96
	LineLevel	0.99	0.99	0.99
Irssi	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	0.99	0.99
All	*FileLevel*	1.00	0.97	0.99
	LineLevel	0.99	0.99	0.99

It thus ignores Lines 2 and 3, which are false negatives in the composed variant when comparing it with the input variant. False positives are due to some lines that are split into multiple lines by our adapter when reading the source code. For example, Listing D.7 shows a *Do Block* followed by the *If Block* and followed by the *While Block* at the same line. The parser gets the statement and adds the *Do Block* in a new line, the *If Block* in another line, and the *While Block* in a third line due to our tree structure for parsing the artifacts in specific types of AST nodes (Figure 2). Therefore, the source code retrieved is correct but retrieved in more lines. Therefore, in the end, if we look for the total amount of lines of all variants of the systems we could easily get a higher number of lines in these specific cases as explained in the Listings D.6 and D.7.

```
1 str = buf;     /* Default buffer to use when we write the
2                    buffer, it may be changed in the flow below
3                    before the actual storing is done. */
```

Listing D.6. Code snippet from Curl, file download.c.

```
1 do { if (...) {...} } while (0);
```

Listing D.7. Code snippet from LibSSH, file client.c.

There were only 350 false negative lines and about 300 false positive lines in the Marlin system from a total of 692,001 relevant lines across all 191 compared variants. In the LibSSH system, 39 lines were missing and 151 were surplus over all 577 composed variants of a total of 8,191,428 relevant lines. In the SQLite system, 358 lines were false positives and 152 were false negative lines in all 424 composed variants from a total of 4,187,636 relevant lines. In the Irssi system, 725 were inserted lines and 789 were missing lines from a total of 7,549,177 relevant lines. In the Bison system, there were no inserted lines and only three missing lines from 1,799,181 relevant lines. In the Curl system, there were 241 inserted lines and 5,163 deleted lines from a total of 4,556,535 relevant lines. Despite not having 100% of precision and recall, as explained before, the few false positives and false negatives resulted from comment lines ignored when parsing the C source code files or different alignments, which did not change the code semantically. Furthermore, compared to a traditional VCS, evolution is tracked at the level AST nodes of feature revisions, not at the level of text lines or entire files.

RQ2. To what extent does *ECSEST* support feature revision location of existing variants that evolved in space and time? *ECSEST proved to be effective for locating feature revisions in the used dataset, with high values for the measures of precision, recall, and F1-score. The approach correctly located the artifacts in an automated way, hence, it may support developers to locate feature revisions and understand their implementation in systems evolving in space and time, even in cases with many feature revisions.*

6.3 Composing variants with new configurations of existing feature revisions

Table 6 shows the precision, recall, and F1-score from the comparison of artifacts (file and line levels) of the ground truth and our composed variants according to the random configurations generated for the Git commits analyzed. *ECSEST* retrieves artifacts with 100% precision and 92%-99% recall at file-level granularity. At line-level granularity, the average precision and recall are 99% for all systems, with an exception for the recall of the system Bison that is 100%. All values for F1 are greater than 96% at file level, and as well as the F1 achieved at line level with the F1 achieved from the input variants, all systems have 99% F1 from random configurations.

For the Marlin system, within a total of 133,161 relevant lines, 64 were inserted and 67 were missing lines. For the LibSSH system, 1002 were inserted lines and 97 lines were deleted lines, from a total of 5,433,889 relevant lines. For the SQLite system, zero were false negative lines and 369 were false positive lines in the composed variants with random configurations from a total of 3,375,242 relevant lines of ground truth random variants. The false positive lines in the composed variants are caused by feature interactions in the chosen configurations, which we randomly chose without considering whether a selected feature excludes parts of code that can be in other features when preprocessing ground truth variants. Therefore, when the random combination of feature revisions resulted in an invalid configuration, the ground truth variant cannot be correctly preprocessed, and thus, has missing artifacts. An example of an invalid random configuration generated in our evaluation is the random variant generated in Git commit #12 of LibSSH, which contains the features `HAVE_SSH1`, `DEBUG_CRYPTO`, `HAVE_PTY_H` and `BASE`.

Table 6: Average *Precision*, *Recall* and *F1 − Score* metrics of composed artifacts for random configurations per system at *FileLevel* and *LineLevel*.

Subject System	Granularity	Precision	Recall	F1-Score
Marlin	*FileLevel*	1.00	0.96	0.98
	LineLevel	0.99	0.99	0.99
LibSSH	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	0.99	0.99
SQLite	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	0.99	0.99
Bison	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	1.00	0.99
Curl	*FileLevel*	1.00	0.92	0.96
	LineLevel	0.99	0.99	0.99
Irssi	*FileLevel*	1.00	0.99	0.99
	LineLevel	0.99	0.99	0.99
All	*FileLevel*	1.00	0.97	0.99
	LineLevel	0.99	0.99	0.99

```
1 #ifdef HAVE_SSH1
2     option->ssh1allowed=1;
3 #else
4     option->ssh1allowed=0;
5 #endif
```

Listing D.8. Code snippet from LibSSH, file options.c.

Listing D.8 shows that when preprocessing a variant with feature `HAVE_SSH1` defined, the ground truth variant will contain Line 2 and not Line 4. Only when this feature is not defined Line 4 will be present in the variant. Our feature revision location approach correctly mapped the artifact from Line 2 to presence conditions containing feature `HAVE_SSH1` and Line 4 to presence conditions containing `BASE` and other features from the respective point in time. However, the ground truth variant does not contain artifacts of both `#ifdefs` and `#else` blocks, hence, not matching with the composed variant. Curl random variants were composed with 208 inserted lines and 3,787 deleted lines among a total of 3,266,773 relevant lines. The randomly composed variants from the Bison system retrieved 54 inserted lines and no deleted lines from a total of 1,412,709 relevant lines. From a total of 6,972,575 relevant lines in the Irssi system, 786 lines were inserted in the randomly composed variants and 1,017 lines were deleted.

We did not test the approach's capability for combining feature revisions from different points in time due to limitations of our ground truth generator. However, the efficient feature revision location assures that feature revisions are correctly traced. In addition, our results of precision and recall for composing new variants only presented lower values when invalid configurations were used due to feature interactions. We did not evaluate if valid configurations can be retrieved, but if our approach can correctly compose variants with the located feature revisions. Our focus is on supporting the feature evolution of annotation-based SPLs in VCSs as variability models of our target systems were not available.

RQ3. How effective is *ECSEST* for composing new variants with feature revisions? *The traces computed by ECSEST proved to be useful for creating new variants with random configurations, still achieving high values for the measures of precision, recall, and F1-score. ECSEST can be used to support tasks to compose new variants from an SPL with a set of feature revisions.*

As explained above and in Section 4, false negatives and false positives can be retrieved in the variants depending on how features are annotated in the ground truth variants and which feature revisions are combined when composing a new configuration that did not exist so far. However, the false positives and negatives can be identified easier with the hints retrieved by *ECSEST* when composing a new variant. It can happen that in some composed variants, no artifacts are missing/surplus and the hints file can either have no hints retrieved or can present hints even there are no feature revision interactions. In the last case, the hints are retrieved because some of the feature revisions of the configuration never appeared together in the input variants.

Table 7 shows the *ArtifactsRatio* indicating that the retrieved hints are useful for all systems with exception of the hints retrieved for the Bison system. For Bison, only a few false positives were retrieved from the new variants, which are almost all false positives caused by the AST nodes used to store the source code in a tree structure. Thus, the false positives in the Bison system do not stem from the algorithm for locating feature revisions, and most of them were not retrieved due to feature interactions, as we see when comparing which artifacts were surplus in relation to the ground truth.

Analyzing the *InteractionsRatio*, the hints with missing traces were most useful for

Table 7: Hints Metrics.

Subject System	ArtifactsRatio	InteractionsRatio
Marlin	88%	90%
LibSSH	38%	88%
SQLite	75%	37%
Bison	2%	66%
Curl	100%	100%
Irssi	60%	83%
All	60.5%	77%

finding artifacts surplus/missing for Marlin (90%) but not as useful for SQLite (37%). In SQLite, this means that despite having new configurations with feature revisions never used together previously, most of the new configurations can be combined and might not have feature interactions. For the Bison system, 66% of the hints with missing traces really pointed to new configurations with surplus artifacts, which means some of them have been useful to find that some features should not be used together.

Although hints were obtained for Marlin, SQLite, and Irssi, they do not reflect actual missing or surplus artifacts. The missing or surplus artifacts were retrieved due to the differences found when parsing the C source code. For example, the SQLite system has 77 surplus lines due to feature interactions used in the random configurations from the total false positive lines retrieved. In the Curl system hints have been more useful for finding surplus/missing artifacts (100% *ArtifactsRatio*) and alerting for missing/surplus artifacts because all new variants have feature interactions due to feature revisions never used together.

RQ4. How useful are the hints suggested by *ECSEST* for completing new variants and finding feature interactions when creating new configurations?
The retrieved hints support the completion of a variant by showing clauses of the presence conditions used to compose a configuration that has feature revisions not included in the configuration. Furthermore, it makes aware of possible interactions of feature revisions, when combining feature revisions that were never used together in a configuration.

6.4 Performance of *ECSEST* to locate feature revisions and compose variants

The performance of *ECSEST* to extract features from systems evolving in space and time (*ExtractionTime*) is shown in Figure 10. The least time a variant took for extraction of feature revisions is the minimum value on the left side of each system box plot. While the highest time for extracting feature revisions of a variant can be seen in the maximum value, excluding outliers, on the right side of each box plot. On average, the analysis took around 83 seconds for Bison, 250 seconds for Curl, 25 seconds for Irssi, 249 seconds for LibSSH, 88 seconds for Marlin, and 212 seconds for SQLite. In the worst case, it took around 15 minutes for the Curl and LibSSH systems, which are the systems with the highest number of variants in relation to the other systems.

As expected, the runtime for locating feature revisions increases with the number of feature revisions and artifacts because the number of artifacts and features is greater to refine the traces. Thus, the time to create new and update traces, increases for every new input variant. For the Marlin system, the outliers (represented in Figure 10 by circles) were caused by Git commit #52, because of the huge number of features introduced (56) and the necessary refinement of the traces of `BASE` for every input variant. In addition, it takes a long time for every new input variant to extract what is new and update from what

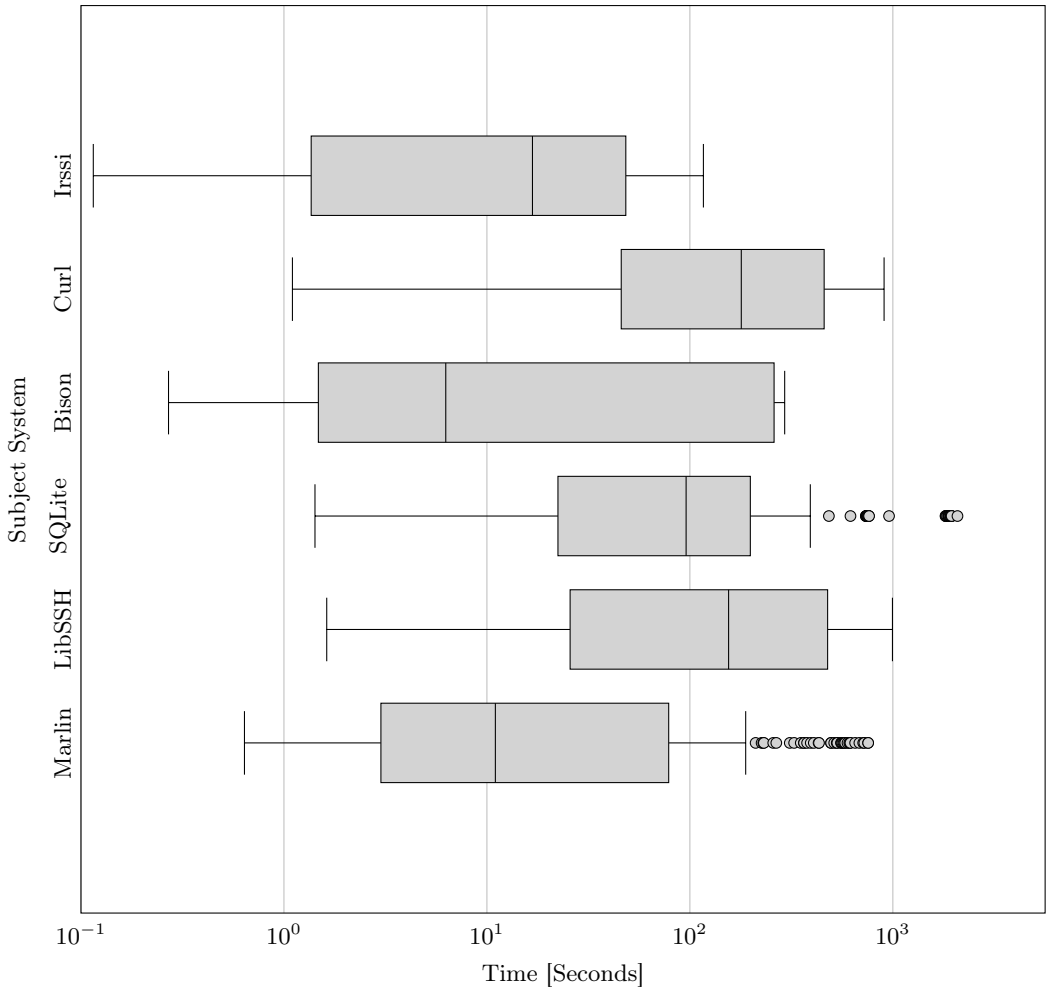


Figure 10: *ExtractionTime* for feature revision location per variant.

is already in the repository. For SQLite the longer extraction time compared to Irssi is probably caused by the huge number of artifacts that had to be compared.

Thus, independently of the number of feature revisions, the size of artifacts can impact the time to refine traces. Another thing that impacts the time to refine traces is the complexity of the tree structure nodes used to store the artifacts in the repository. It is our implementation limitation, and as many more different artifacts from one commit to another in the input variants, many more tree nodes are created to store this information in a tree structure. In a real scenario, developers may limit the number of commits and variants to extract feature revisions, using only the desired ones for new combination of feature revisions. Further, despite developers may need to wait, using the *ECSEST* approach does not require developers' time and effort, which they can use in parallel to complete other higher-level tasks and decision making.

The *CompositionTime* for composing a new variant, i.e., joining the artifacts from traces of a set of feature revisions, is presented in Figure 11. Similar to box plots in Figure 10, the least time to compose a variant with feature revisions is the minimum value on the left side of each system box plot, while the highest time for composing a variant with feature revisions can be seen in the maximum value, excluding outliers, on the right side of each box plot. For the system with the best runtime performance, it took around two seconds on average per variant. For the system with the worst average, it took around 45 seconds per variant. Some of the outliers (represented in Figure 11 by circles) in the systems are due to the warm-up effect of the JVM. After the warm-up effect, the time remains constant

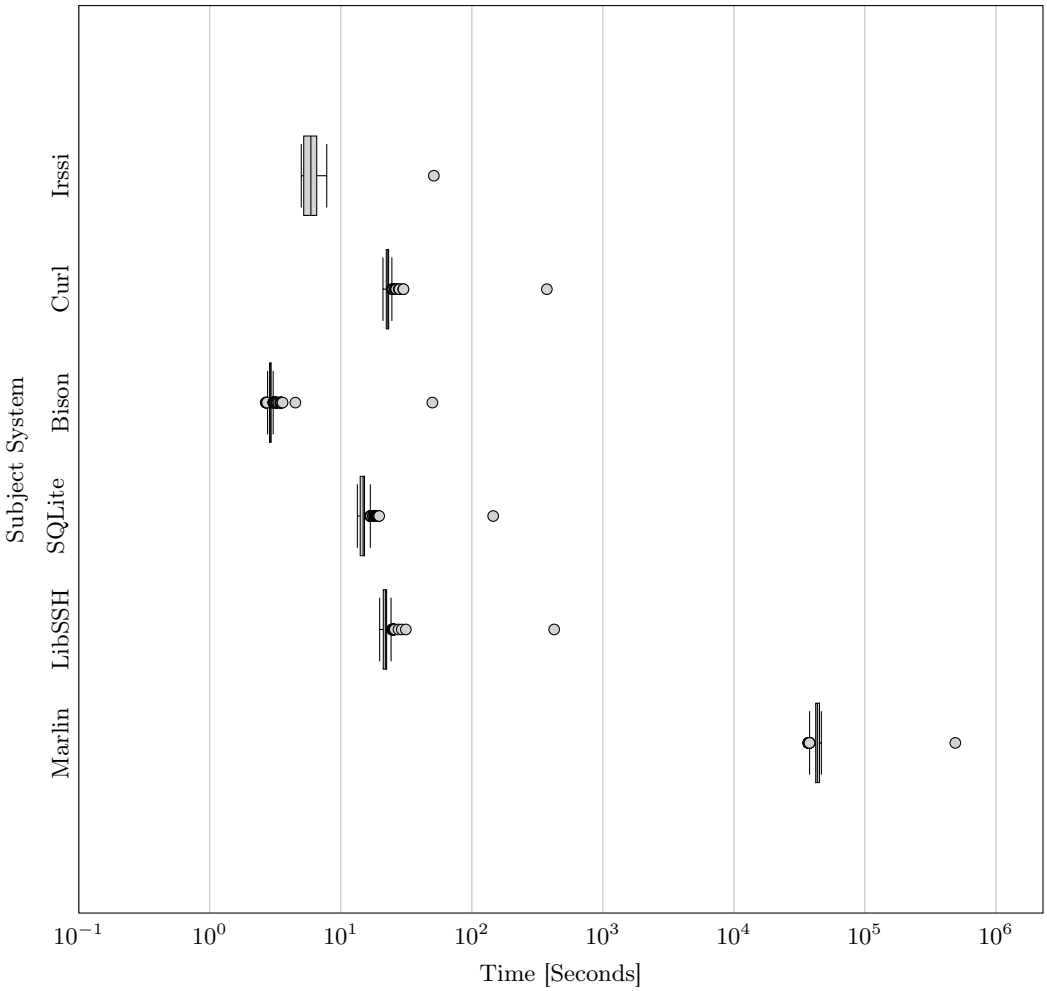


Figure 11: *CompositionTime* for composing each variant.

to compose variants.

For the LibSSH and Curl systems, we had some outliers for which it took up to seven minutes to compose a variant (Figure 11). When comparing the time between systems, we see that it is higher in repositories containing more traces and feature revisions because many clauses from many traces need to be analyzed to join the artifacts into a variant. However, the number of artifacts is also a factor that influences the composition time. For example, the Bison system has the smallest number of artifacts compared to the other systems (see Table 5) and also the smallest time to compose variants. However, the Curl system is smaller than the Irssi system in terms of artifacts, but took more time to compose variants because it has more feature revisions, hence, more input variants, which results in a higher number of traces. Thus, more time is needed to compose a variant with high numbers of feature revisions and artifacts.

Regarding the composition time, the composition requires that the extraction process was already performed. Despite the extraction time, only new variants with new feature revisions have to be analyzed in our incremental process of locating feature revisions, i.e., the approach uses existing variants and refines existing and creates new traces only when needed. Optimizations of the runtime performance can be performed in the implementation aspects of the approach to store the artifacts and the counter table for mapping feature revisions to artifacts. Still, the composition time in the worst case only took seven minutes, while the approach could save significant effort of manually copying and pasting artifacts for composing variants or for propagating changes of features.

RQ5. What is *ECSEST*'s performance for extracting feature revisions and composing variants? *The automated approach to locate feature revisions and composing variants with ECSEST may save time in recovering feature information and maintenance and evolution tasks. The average time for extraction for our target systems ranged from 25 to 250 seconds per variant, whereas the average time for composing a variant ranged from two to 45 seconds.*

7 Threats to Validity

We discuss the threats to the validity of our evaluation using the taxonomy of Wohlin et al. [193]. We also describe how we mitigated possible threats.

The threats to *construct validity* are related to the study setup. Firstly, the scenarios used to validate our approach contain changes to features, but we did not have data on the actual type of evolution, e.g, performance improvement, new hardware support, bug fixing, etc. However, the Git commit hashes of every variant and revisions of the features are available in our dataset for future replications and deeper analysis. Secondly, the methodology chosen to evaluate our approach was based on variants in space and time created by a configuration of features in specific changes of annotated code in the Git commits we analyzed. This was necessary since there is no ground truth available with variants containing feature revisions. To mitigate this threat and to demonstrate the efficiency of our approach, we generated new variants with different configurations of feature revisions, which were not used as input and randomly chosen for the points in time we analyzed.

Another constructive threat can be related to the correctness of our mining ground truth approach. To mitigate this threat, we manually inspected the ground truth variants generated for the first 50 Git commits of each target system. Yet, regarding the sufficient variability of the ground truth variants, we used variants from Git commits containing introduced and changed features. Furthermore, we also presented some results of how representative the feature evolution of the mined ground truth variants is for the real systems we used.

A further threat to construct validity is the new combinations of feature revisions, as we do not ensure that they are type-safe. However, as mentioned, our approach is intended for tracing feature revisions to artifacts and using the revisions to compose variants. Thus it is the user's responsibility to select valid configurations. Our aim in this work is to analyze if the mapping between feature revisions and artifacts is performed correctly, and if the variant composition approach works. Hence any valid configuration will be correctly composed when every feature revision is correctly traced and the variant composition results in the expected artifacts. Furthermore, work by Feichtinger et al. [48], presents an approach to inform engineers about possible inconsistencies between code-level dependencies to feature models. Thus, we can improve our approach in future work by using a similar analysis to the variant composition.

A threat to *internal validity* is the limitation of the underlying tools that could have affected our results. We implemented our approach in ECCO, in which source code is available and was used in previous works [51, 52, 56, 97, 129] that shows its efficiency to extract features and compose variants. We also used our developed adapter to parse the C source code and to write it back when composing variants. Although we did not compile the resulting code of the composed variants, we validated the correct composition of artifacts with smaller examples and subsets of the dataset. Furthermore, our implementation is also available for further comparison and to reduce possible bias in our results.

A threat to *external validity* is related to our findings be generalized beyond the cases we considered. Despite we conducted our evaluation with only six systems, these systems are from different domains and have different sizes with different behaviors in terms of how

often their features change along the Git commits. Furthermore, the range of Git commits analyzed for each system varies, which makes our analysis valid for systems with a higher number of feature revisions. To mitigate bias in relation to the number of Git commits we used for each system, we performed a triage specifically for each system, allowing us to define how far we could go with the memory limitations of our machine used to run the experiments. We thus believe that our results cover diverse enough scenarios and our approach can support a large number of feature revisions.

From the perspective of *conclusion validity*, a threat can be related to the metrics we used to evaluate our approach effectiveness. However, precision, recall, and F1-score are efficient to measure how correct is the information retrieved [190]. Furthermore, they are commonly used to evaluate feature location techniques [34, 116, 128, 129], and hence, can make easier the comparison of our results.

8 Related Work

Many existing solutions for evolution in space consider only variability in space, such as SPL, and require integration with a VCS to manage the evolution of software systems in time [21]. Thus, there is a lack of dedicated and mature tools supporting system evolution in both dimensions for tracing and developing features over time [3]. Considering both dimensions, Ananieva et al. [3] presented a conceptual model, which proposes a unified terminology for tools managing variability in space and time. The conceptual model aims at clarifying communication between researchers and developers for understanding and comparing existing tools, and for preventing duplicated tool development.

Regarding existing tools for software system evolution in space and time, ECCO was presented by Fischer et al. [51] as an extraction and composition tool for re-engineering cloned system variants into SPL. After mapping all existing features to artifacts, developers can select the desired set of features to compose a new product variant and also provide hints for manual completion of which software artifacts would need adaptation. Then, ECCO was built upon the checkout/commit workflow for distributed software development [100]. It evolved to extract variability information from system variants computing traces of not only single features but also feature interactions and absence of features, non-unique traces, and dependencies between traces [103]. ECCO has also been used in a large-scale industrial case study [68]. In this work, we present a significant extension to the tool, with a feature revision location [134] and composition of variants to support system evolution and comprehension of feature evolution.

Superimposition of Models (SuperMod) [172] is a tool for evolving model and implementation of SPLs in a conceptual framework for integrating revision and variation control of model-driven software projects [173]. Similar to VCSs, SuperMod is based on a workspace repository, where the user can store and edit the files of source code and modeling of all revisions and variants of an SPL. Thus, the workspace is populated with the feature model and the model artifacts belonging to a revision of the SPL. The tool allows artifacts and feature models to co-evolve. The SPL is evolved in iterations via commit/checkout operations, similarly to VCSs. SuperMod allows collaborative development, where each user works with a local repository and can copy and update the central remote repository. One of the tool limitations is that there is no possibility of working with different artifacts than model and text files. Furthermore, there are some adoption barriers to migrating the SPLs from external tools.

DeltaEcore [175] is a tool for capturing variability in space by automatically defining delta languages for a variety of languages relevant for SPLs and software ecosystems (SECOs). These delta languages can be textual, graphical, or use any other representation, and are required for software systems that consist of multiple artifacts, such as design models, source code, configuration files, or documentation. Thus, the tool is able to derive syntax

and semantics for custom delta languages from a specific source language's meta-model. Furthermore, the tool supports editing, parsing, and interpreting a generated delta language, which can be integrated into a mechanism for composing variants of an SPL or SECO. Despite the tool supports variability model based on a Hyper Feature Model (HFM) [174], which depicts the dependencies and incompatibilities of features version ranges of SPLs, it is limited to the evolution of system families in time, because it does not support to evolve artifacts and/or features.

A recent survey by Linsbauer et al. [104] describes existing VarCSs and their essential differences, the challenges, and insights for the future generation of tools to support the development of system families evolving in space and time. Among the challenges mentioned, the externalization expression of which functionalities (variable artifacts) are part of a variant can become cognitively complex to handle because a high number of revisions and variants can exist and a million thousand features. For instance, the Linux kernel has 15,000+ features (configuration options), which some of them can have 3 values: "yes", "no", or "module", with an estimated number of $3^{15,000}$ possible variants of Linux [153]. It is still unclear when and for what developers would have advantages by using a VarCS in such a context. Thus, it is important to conduct studies on what characteristics a VarCS should have to help to deal with systems evolving in space and time. This includes the types of artifacts VarCSs should allow to create and manipulate, the kind of operations they should support, and features ensuring usability to deal with the cognitive complexity involved. In this direction, studies are needed to investigate the ECCO VarCS capabilities in more detail. For example, evaluating if its characteristics and operations are useful and can be performed efficiently to support the evolution of system families. Our study shows ECCO's current utility and suggests for improvements, which can serve as a basis for new studies and the development of tools for software system variability and evolution.

An approach to support the composition of new variants based on opportunistic reuse, namely clone-and-own methodology, is presented by Ghabach et al [59]. It supports mappings between features and artifacts in an automated and incremental way. The paper also discusses possible scenarios, constraints and cost estimation for operations to compose new variants using clone-and-own. The scenarios are given by three hints: (i) clone and retain, when developers clone an artifact and can retain it as it is, without modifying its implementation; (ii) clone and remove, when developers may clone an artifact instance, and have to remove from it the implementation artifacts that are not required by the configuration; and (iii) extract and add, when developers extract from product artifact implementation of feature required by the configuration and add it to a cloned new variant under composition. The mapping of features and composition hints and cost estimation is defined by means of correlations, indicating the coexistence of a feature and an artifact, or a feature and an artifact. Thus, this approach, similar to ours for locating features from existing system variants, is independent of the artifacts types. However, *ECSEST* is able to locate features at different points in time. Regarding the composition of new variants, *ECSEST* can also use existing cloned variants to compose new products. In addition, *ECSEST* can easily retrieve the variant in an automated way by informing the set of feature revisions desired in the configuration. Our hints can help developers to determine if the variants generated in an automated way have possible remaining or missing artifacts. The results from our feature revision location technique [134] show that our approach maps features to their artifacts with high precision and recall, which means that less effort is needed to tailor a variant, as fewer removals and additions are necessary when composing variants with new configurations.

But4Reuse [114] is an approach for migrating software variants into an SPL by constructing its feature model. Also, a unified, generic, and extensible framework is proposed to create benchmarks of feature location techniques and enable users, developers, and researchers to analyze and compare different techniques [115]. However, the approach does not permit

an incremental evolution of the SPL, and the feature location approach is not able to locate feature revisions. Furthermore, although it is not the focus of our work, we also present a mining tool to generate ground truth variants. Then, ground truth variants can be generated with our mining tool and the ones already used in our work are available too. Also, future work on locating feature revisions and re-engineering software system variants with multiple revisions can benefit of our ground truth generator.

9 Conclusions and Future Work

Existing feature location techniques are limited to analyzing specific system snapshots at one certain point in time. To address this limitation, this paper demonstrated the importance of feature location in both space and time and introduced an automated approach for feature revision location, which allows to reason about features in different points of time and supports software systems evolving in space and time. The results show that our approach can locate the features' artifacts with a precision of 100% at file-level and $\geq$ 99% at line-level granularity, as well as a recall of 97% at file-level and 99% at line-level granularity. The incorrect information retrieved is due to the different syntactic structures of the semantic of a source code. Regarding the performance of our feature revision location approach, we reported that it took on average in the worst case 250 seconds and in the best case 25 seconds to trace artifacts to feature revisions for each input variant.

For composing a new variant, our approach took around 18 seconds on average of all systems to compose a variant. Even if manual completion is necessary, it will not require extensive code additions or deletions by a developer. The hints provided by our approach make it easier to find possible artifacts to be added or removed based on the presenting missing and surplus clauses containing the feature revisions and traces with conflicts and/or that do not exist in the repository. Thus, our automated approach can aid developers to evolve and maintain software systems at the level of feature revisions, thereby saving time and effort. Hence, it facilitates the management of system variability in space and time by composing variants with feature revisions easily and in a reasonable time. It also supports combining feature revisions that never were combined previously. Therefore, *ECSEST* provides additional functionalities than commit messages in Git VCS, such as location of feature revisions and combination of feature revisions from different commits.

We hope that our results will inspire researchers and tool builders to work with feature revisions to treat feature evolution in space and time, and will also encourage them to address current VarCSs limitations and/or improve other existing variability tools combined with a strategy for dealing with the evolution in time. We encourage future work comparisons with our work to reuse *ECSEST* approach's strength, or fulfill remaining gaps, and improve its weaknesses/limitations by the use of common metrics, such as precision and recall, and by the dataset available[27] containing the ground truth used.

As future work, we want to conduct more experiments with industrial systems and from different domains, considering other programming languages such as Java, and other different artifact types. We also want to evaluate *ECSEST* for managing clones in product line engineering with feature revisions, using operations such as a Git VCS pull and push, but using a distributed ECCO repository for feature revisions to aid the implementation of system variants with feature revisions [69]. In addition, we plan to improve our approach for dealing with evolution of dependencies and interactions in the source code of feature revisions, similar to Feichtinger et al. [48], to automatically check for inconsistencies between feature revisions and their implementation when composing new configurations. Concluding, our future biggest goal is to provide an independent mechanism for enabling the management of variants with any combination of feature revisions.

[27] http://doi.org/10.5281/zenodo.4555199

Acknowledgment

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; the Brazilian National Council for Scientific and Technological Development (CNPq), grant no. 408356/2018-9; and Carlos Chagas Filho Foundation for Supporting Research in the State of Rio de Janeiro (FAPERJ), program PDR-10 Fellowship, 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. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant RGPIN-2017-0542, and also has been supported by the competence centers program COMET of the Austrian Research Promotion Agency (FFG), grant no. 865891.

Paper E

Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants

Published in the Proceedings of the 25th ACM International Systems and Software Product Line Conference (SPLC 2021), 6 pages, Leicester, United Kingdom, 2021.

Authors Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher and Alexander Egyed.

Abstract Software companies need to provide a large set of features satisfying functional and non-functional requirements of diverse customers, thereby leading to *variability in space*. Feature location techniques have been proposed to support software maintenance and evolution in space. However, so far only one feature location technique also analyses the *evolution in time* of system variants, which is required for feature enhancements and bug fixing. Specifically, existing tools for managing a set of systems over time do not offer proper support for keeping track of feature revisions, updating existing variants, and creating new product configurations based on feature revisions. This paper presents four challenges concerning such capabilities for feature (revision) location and composition of new product configurations based on feature/s (revisions). We also provide a benchmark containing a ground truth and support for computing metrics. We hope that this will motivate researchers to provide and evaluate tool-supported approaches aiming at managing systems evolving in space and time. Further, we do not limit the evaluation of techniques to only this benchmark: we introduce and provide instructions on how to use a benchmark extractor for generating ground truth data for other systems. We expect that the feature (revision) location techniques maximize information retrieval in terms of precision, recall, and F-score, while keeping execution time and memory consumption low.

1 Introduction

Software companies have to tailor and maintain variants of software systems co-existing simultaneously to serve different customers and new requirements. Variants of a system are composed of variable assets related to different features that realize the variability of a system [14]. The system variants reflect different configurations and have been described as *variability in space* [174,189]. *Variability in time*, on the other hand, results from the need of modifying variants due to enhancements, for example, to address new customer requirements or changes to the environment, such as alternative hardware or the optimization of non-functional properties [189]. Thus, over the system life cycle, the introduction of new features in existing variants, a.k.a *evolution in space* can be required. Further, features can be subject to failures or unwanted behaviors and bug fixes have to be done, introducing new revisions of features, which is referred to as *evolution in time* [174]. Furthermore, the evolution in time results in variant revisions, which are sequential versions of a variant, containing different artifacts for the same configuration, i.e., a set of features [21, 104].

The aforementioned scenarios lead to many system variants that need to be managed and evolved in parallel. This highly increases the workload of developers. Furthermore, keeping system variability consistent across different types of artifacts manually is an error-prone task [104]. Software product line (SPL) approaches have been adopted by engineers for systematic variability management and reuse of the core assets and features' artifacts, thereby accelerating the production of variants and reducing the effort and costs for maintaining and creating products [84]. Regarding the transition of existing systems to an SPL, feature location is the first and one of the most essential tasks of the re-engineering process to migrate a family of existing system variants into an SPL [12].

Despite feature location and SPLs cover the *space* dimension, they do not address the *time* dimension [21]. SPLs by themselves do not provide proper management of evolution in time and engineers have to adopt additional mechanisms and tools. SPLs are frequently managed in version control systems (VCSs), which track changes of a system over time [31]. However, current VCSs have support for managing the versions of variants but not for managing versions of features. Some pieces of work point out the need for an SPL to have a portfolio to reflect potential versions of a feature, i.e., the feature revisions that co-exist, which can be reused for creating different variants [21, 104, 134].

Existing feature location techniques can locate features of a system [40, 128, 157] or a set of systems [1, 129], but only at one point in time. Although there are some feature location techniques considering the *space* dimension, they have limitations as presented in our previous work [128]. Still, regarding the *time* dimension, there is only one feature revision location technique able to retrieve traces of feature revisions [134], which is in the early stages of development, with sub-optimal results, and limitations in terms of the number of feature revisions that can be located.

We thus stress the need of introducing new, or improving existing, feature (revision) location techniques by describing four challenges to be solved by the research community and tool developers. These challenges are concerned with locating features at one point in time as well as at multiple points in time (Section 2). Yet, the proposed feature (revision) location techniques support engineers in creating new configurations based on the traced features and their revisions. By proposing these challenges, we aim to motivate researchers and tool developers to optimize and address the limitations of existing techniques and to develop more efficient mechanisms for managing systems evolving in space and time.

Evaluating solutions for the challenges with a common benchmark can enable future work comparisons [128]. For this purpose, we contribute with both a benchmark and a ground truth extractor[1] for evaluating these techniques. Thus, the benchmark contains:

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

(i) a ground truth dataset[2] with variants from three C open-source systems evolving in space and time with their respective configurations at one point and multiple points in time; (ii) tool utilities[1] to evaluate the efficiency of feature (revision) location techniques that compute automatically three metrics: precision, recall, and f-score.

The remainder of this paper is structured as follows. Section 2 presents the motivation and challenges of this paper. Section 3 provides detailed information on the benchmark. We discuss the scenarios and metrics for evaluating solutions and briefly explain the ground truth extractor. Section 4 concludes the paper.

2 The Challenges

To describe our challenges, we now present the background of feature (revision) location techniques. Then we explain the importance of these techniques, their current limitation, and observed complexity, or lack of studies, which makes the challenges interesting.

2.1 Feature Location

A feature can be a functional or non-functional requirement that represents a software system's functionality [40]. Let's use the Marlin system, open-source firmware for 3D printers, as an example. It has features for linear acceleration, control of the temperature to melt the filament or buzzer sounds for warning signals [88]. Some of the features are optional, i.e., not all products of a system have to include them. These optional features are thus units of variability responsible for changing the system's functionality and behavior.

Marlin is an annotated SPL, however, according to a study from Krüger et al. [88], not all optional features are used in variation points, e.g., `#ifdef` preprocessor directives. During maintenance and evolution, developers need the complete locations of features, which can be outside the annotations. This requires manual work that could be automated by feature location techniques. Furthermore, feature location techniques are helpful not only for the software maintenance and evolution tasks as well as for the re-engineering process of cloned software systems into SPLs [12, 163].

There already exists a large number of feature location techniques, which use, e.g, textual, static, or dynamic analyses, or combinations thereof [40, 128]. Despite many feature location techniques available, results can be compromised by the number of existing software systems when using a comparison-based static analysis for re-engineering existing software systems into SPLs [129], for example. Yet, the quality results of textual analysis are highly-dependent on index terms and queries, while the results from the dynamic analysis are very sensitive to how scenarios and features are executed [128]. Nonetheless, feature location techniques have different evaluations, metrics, and ground truth data sets, making it difficult for practitioners to decide which one is most appropriate for them [163]. Further, some of the work proposing feature location techniques cannot be reproduced because of not available material, which makes it difficult to compare existing feature location techniques with improvements addressing their limitations [128, 163].

Therefore, more common benchmarking frameworks for evaluating feature location techniques have been suggested [116]. Currently, there is a benchmark proposed by Martinez et al. [116] based on the ArgoUML system, which is implemented in Java. However, differences in source code entities between different languages have a strong impact on feature location [167]. Yet, there are few benchmarks available that can be used to apply feature (revision) location to C-preprocessor-based systems. A benchmark for C software systems would ease the proposal and evaluation of feature (revision) location techniques because C is widely used for realizing SPLs with preprocessor directives [119]. We thus now present our first challenge:

[2] http://doi.org/10.5281/zenodo.4586774

Challenge 1: Feature location at one point in time. We aim to motivate researchers and tool developers to evaluate existing or new feature location techniques based on systems developed in C or C++, using a common benchmark enabling the studies' reproducibility and comparison.

We also want to motivate the development of approaches for automation of the reuse of features for composing new configurations. We thus present our second challenge:

Challenge 2: Composition of new product configurations with a set of features. We evaluate if the proposed feature location approaches for C/C++ systems can be used to compose new configurations with the traces retrieved to simplify and accelerate the composition of not yet existing variants of a system.

2.2 Feature Revision Location

A feature revision represents the change of the implementation artifacts associated with a feature at a specific point in time [71, 132]. Previous studies [21, 71, 132, 134] stress the need to manage system variants over time at the level of feature revisions. Even if a software system already manages its features as an SPL, maintenance and evolution will introduce changes, affecting the implementation of the system's features, which may become inconsistent across the existing variants. This makes changes increasingly hard to understand and propagate to variants at any time a feature has to be revised [42]. In the literature and practice, there is no unified mechanism to deal with the evolution of systems in space and time [21]. While we presented a feature revision location technique for software systems evolving in space and time in our previous work [134], there are some limitations that need yet to be addressed. For instance, a higher number of feature revisions could be traced with less memory consumption and higher effectiveness in retrieving information. We thus present the third challenge to motivate researchers and tool developers to improve our feature revision location technique or propose new ones overcoming current limitations.

Challenge 3: Feature revision location at multiple points in time. We expect solutions with feature revision location techniques to automate the process of mapping implementation artifacts to feature revisions for every existing different implementation of a feature at multiple points in time.

Aiming to motivate better and unified mechanisms and tools for system evolution in space and time, we present our fourth challenge. It is intended to use the feature revision location technique solution from the third challenge as an extractive approach [12] for re-engineering existing variants' versions by systematically reusing feature revisions. By raising initial solutions for systematic reuse in software systems evolving in space and time to the level of features, developers and engineers can benefit not only when propagating bug fixes and refactoring but also when creating new configurations with different behaviors of the same feature.

Challenge 4: Composition of new product configurations with a set of feature revisions. We expect solutions that can automate the reuse of existing feature revisions of a system in order to compose different configurations with the different implementations of features at different points in time.

3 Benchmark

Our benchmark can be used for evaluating and comparing feature (revision) location techniques for the C programming language with an established set of metrics[1] and dataset[2]

from available open-source systems.

3.1 Subject Systems

Our benchmark is composed of preprocessed SPLs implemented in combination with version control systems, which keep a history of the changes over time and enable us to generate ground truth variants with features from multiple points in time. The systems are LibSSH, Irssi, and Marlin. These systems have been used in previous studies [57, 64, 90, 96, 118, 132, 134], and are managed in Git repositories. We thus believe they are representative target systems to be used to evaluate feature (revision) location techniques. The LibSSH[3] system is a multi-platform C library implementing the SSHv2 protocol on the client- and server-side. This project was initiated in 2005 and now has around 5000 commits in the master branch. The Marlin[4] system is a variant-rich open-source embedded firmware for 3D printers created in 2011 and with currently around 15000 commits. The Irssi[5] system is an internet relay chat client for Linux with around 6000 commits since 1999.

3.2 Evaluation Scenarios

Variants with Features. The scenarios for evaluating solutions for *Challenge 1* are from variants containing a set of features from one release, i.e., the state of the system after the last commit of a release in the repository. We designed 13 scenarios (Table 1) of each system with a specific number of variants, where scenarios 1-10 have 1-10 input configurations and scenario 11 has 100, scenario 12 has 200, and scenario 13 has 300 input configurations.

For *Challenge 2*, we make available 50 new configurations that do not exist in any one of the scenarios to evaluate the solutions.

Table 1: Scenarios to feature revision location.

		Number of Features		
Scenario	Number of Variants	LibSSH	Marlin	Irssi
1	1	65	41	26
2	2	91	56	32
3	3	99	61	37
4	4	102	65	40
5	5	104	65	40
6	6	104	67	40
7	7	104	67	41
8	8	104	67	41
9	9	104	67	41
10	10	104	67	41
11	100	104	67	41
12	200	104	67	41
13	300	104	67	41

Variants with Feature Revisions. The scenarios for evaluating solutions for *Challenge 3* are from variants containing a set of feature revisions from 400 points in time, i.e., from the first 400 Git commits of the master branch. We designed nine scenarios (Table 2) for each system according to a specific number of Git commits, by varying the number of variants for each system. For both all systems, we present scenarios that consist of locating feature

[3]https://gitlab.com/libssh/libssh-mirror
[4]https://github.com/MarlinFirmware/Marlin
[5]https://github.com/irssi/irssi

Table 2: Scenarios for feature revision location.

		LibSSH			Marlin			Irssi		
S	C	V	F	R	V	F	R	V	F	R
1	1	14	14	0	1	1	0	7	7	0
2	5	22	14	8	19	13	6	11	7	4
3	10	32	14	18	24	13	11	16	7	9
4	15	40	15	25	37	16	20	21	7	14
5	50	111	34	77	81	16	64	65	15	50
6	100	182	34	148	333	130	179	84	16	101
7	200	322	39	283	463	135	303	170	28	209
8	300	458	40	418	579	139	413	341	28	314
9	400	575	40	536	683	142	514	441	28	414

S = Scenario; C = Number of Git commits; V = Number of variants; F = Number of features; R = Number of feature revisions.

revisions from 1 point in time to up 400 points in time. Furthermore, we make available an additional scenario for the LibSSH system with a set of 6730 variants, resulting in 6596 feature revisions from 103 features, thereby covering the entire evolution of the master branch.

For *Challenge 4*, we make available one new configuration for each point in time where solution proponents can combine different feature revisions for all the scenarios presented in this work.

3.3 Format of the Proposed Solutions

The ground truth is composed of a set of variants and no traces, which allows for multiple valid traces. Then, the solutions for all challenges have to show as result the variants composed with the mappings of each feature (revision) to its artifacts that are part of a configuration, i.e., the artifacts that form the input and new product configurations of the ground truth. Additionally, we expect the result files from the metrics computed (see Section 3.4).

3.4 Metrics

In this section, we present the metrics suggested to evaluate the feature (revision) location technique regarding its quality (correctness) and performance (scalability). The correctness must be computed based on the comparison between the ground truth variants with corresponding retrieved ones obtained after performing the feature (revision) location.

Correctness. To evaluate the effectiveness of the feature revision location technique, i.e., the quality of its search results, we adopt efficient and frequently used information retrieval metrics precision (P) and recall (R) [112, 163] (cf. Equations E.1 and E.2). Furthermore, we also adopt the F-score (F), i.e., the harmonic average between precision and recall, as a single value equally balancing precision and recall (cf. Equation E.3) and assessing the techniques' effectiveness [41].

We used two levels of granularity due to the granularity of the ground truth extraction. The variants can be obtained from lines that have been added/removed/changed in C source code, binary, or text files from Git commits. Therefore, the metrics can be computed at the granularity of file-level and line-level: The file-level comparison checks if two complete files (ground truth and retrieved) match their content; while the line-level analysis compares every line of source code of two files (ground truth and retrieved).

Precision (Equation E.1) is the relation of the true positives (TP), i.e., the correctly retrieved files, which entire content matches, and false positives (FP), i.e., the files that their entire content does not match. At the line-level, TP are the lines of source code that match related to the lines of source code retrieved by the technique that does not exist in the ground truth variants.

$$Precision = \frac{TP}{TP + FP} \tag{E.1}$$

Recall (Equation E.2) is the relation of the false negatives (FN), i.e., the files or lines of source code that exist in the ground truth variant but were not retrieved by the technique.

$$Recall = \frac{TP}{TP + FN} \tag{E.2}$$

The F-score (Equation E.3) is the harmonic average of precision and recall.

$$F\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{E.3}$$

Instructions on how to use our tool utils for computing automatically these metrics are available in our Git repository[1].

Scalability. To evaluate the scalability of the feature (revision) location technique, we expect proponents to compute runtime and memory consumption, reporting the specification of the infrastructure used to run the proposed solutions. These metrics can help to compare and improve techniques regarding the time complexity and space complexity, i.e., how much time a technique takes to locate features (revisions) for each variant and how much memory is necessary to locate features and their revisions for a specific number of variants.

3.5 Ground Truth Extractor

The ground truth extractor and instructions on how to use it are available in our Git repository[1]. We now explain how our extractor mines features and feature revisions to generate a ground truth. Our explanation relies on a small running example shown in Listing E.1, where we use the first Git commit of the Marlin system.

We first need the set of features of a system defined to be able to preprocess variants. Then, we make available the possibility of setting up manually the set of existing features of a system in our extractor or computing features automatically based on our approach to identifying features.

Identifying features. From a specific range of Git commits, we analyze all macros used in preprocessor directives. The macros used in the *#ifdefs* directives are candidates to be part of the features of the system. We also analyze the macros used in *#define* directives, which we discard from being features of the system. Therefore, the macros considered features are the ones that have never been used in *#define* directives in the range of Git commits analyzed.

In the case of our running example (Listing E.1), the possible feature candidates are the macros `CONFIGURATION_H`, `ADVANCE`, `MOTHERBOARD`, and `__AVR_ATmega644P__`. In the next analysis, we look for *define* directives and eliminate the macros `CONFIGURATION_H` and `MOTHERBOARD`. Thus, the set of features we consider is composed of features `ADVANCE`, `__AVR_ATmega644P__` and `BASE`. The feature called `BASE` is the feature containing the core of the system. The `BASE` can be represented by the files that are not source code files, and all code of conditional blocks, i.e., *#ifdefs* with macros that are not part of the set of features, for example, the conditional block from Lines 1-9 in Listing E.1. Now we have the set of features to preprocess the variants or to start the process of mining feature revisions.

```
6   #ifndef CONFIGURATION_H
7       #define CONFIGURATION_H
8       #define MOTHERBOARD 5
9       #ifdef ADVANCE
10          #define EXTRUDER_ADVANCE_K 0.02
11      #endif
12  #endif
13
14  #if MOTHERBOARD == 1
15      #ifndef __AVR_ATmega644P__
16          #error
17      #endif
18  #endif
```

Listing E.1. Code snippet adapted from file Configurations.h from the first Git commit 750f6c3 of the Marlin system.

Mining feature revisions. Mining feature revisions consists of finding features, which were affected by changes to their implementation in some Git commits, with lines added, changed, or removed at specific points in time. For this analysis, we consider all conditional blocks, all *#define* directives, and also the blocks and directives from the top of the file including recursively all the ones in header files, i.e., files used in the *#include* directives. We create then a set of constraints to represent the conditions that must be satisfied to execute a specific line of source code (see [132, 134]). For example, Line 3 in Listing E.1 will be executed if the macro `CONFIGURATION_H` is not defined. In this example, we then know that `CONFIGURATION_H` is not a feature and the conditional block of the macro `CONFIGURATION_H` belongs to the `BASE`.

However, in more complex cases, let us suppose the macro `MOTHERBOARD` is not defined on Line 3 in Listing E.1 and there is another file containing a conditional block with a feature that is defining a value 1 for the macro `MOTHERBOARD`. Thus, the conditional block from Line 9-13 would belong to that specific feature defining `MOTHERBOARD`, instead of belonging to the `BASE` feature. Yet, another example is the macro `MOTHERBOARD` defined in two locations: on Line 3 in Listing E.1 and in another file as we mentioned. We thus consider the conditional block of the macro `MOTHERBOARD` as part of the closest feature, which in this example is `BASE`. We use the closest feature because the `MOTHERBOARD` value would have already been replaced by the value on Line 3, which is part of the feature `BASE` before preprocessing the block of Lines 9-13 in Listing E.1. Finally, when preprocessing the source code, the lines of the conditional block of the macro `MOTHERBOARD` would be executed from the code of the feature `BASE`.

A feature revision then is a feature that is introduced or changed when comparing one point in time to another. We thus get a range of Git commits and compare the first commit with the second, the second commit with the third, and so on. The comparison consists of analyzing for each line of source code added, removed, or changed between two Git commits, which features are part of it. The feature(s) is/are then selected to preprocess a variant, which contains the artifacts of at least the feature `BASE` and possible other feature(s). Thus, from the features used to preprocess a variant, only the closest feature will have an increment in its revision, i.e., in the number that proceeds the name of the feature, which represents kind of a new version of a feature revision.

Taking into account that all the lines from Listing E.1 were added, we have three variants representing the changes of this point in time. One is a variant containing the feature revision `BASE.1` with the number 1 as it is the first revision of the feature `BASE`. The preprocessed result comprises Lines 2, 3, and 5 from Listing E.1. A second variant contains the feature revisions `BASE.1` and `ADVANCE.1`. The result from preprocessing comprises Lines 2 and 3 from Listing E.1. The third variant containing the feature revisions `BASE.1` and `__AVR_ATmega644P__` has then Lines 2, 3 and 11 from Listing E.1. This same process

repeats for every change over all artifact files of a system for every Git commits of a selected range. More details of the approach we used to create variants with feature revisions are shown in our previous work [132, 134].

We used the ChocoSolver[6] to implement our benchmark extractor and to automate the analysis of building correctly the set of constraints and getting correct solutions, i.e, the features to be selected or excluded to execute a specific line of source code. We chose this solver because it enables us to get basic arithmetic operations and comparisons of numeric values in the range of integer or double despite basic logic operations and Boolean values, which would not be possible with an SAT solver, for example.

4 Conclusion

We presented four challenges relevant for feature (revision) location techniques to motivate the proposal of solutions for better mechanisms and tools to support the evolution of systems in space and time. We made available a benchmark for comparing future work for supporting reproducibility. It contains a dataset and tool utilities for computing metrics. The dataset comprises a set of variants and their configurations to be used as input for the techniques and a set of variants and their configurations to be used as new configurations. This allows to evaluate if the resulting traces can be used to compose variants with a new set of features and feature revisions.

The ground truth of features at one point in time was generated by preprocessing SPLs with our benchmark extractor, as well as the ground truth of feature revisions from multiple points in time. The ground truth extractor for generating variants with feature revisions was also used in our previous studies [132, 134], which mines previously the feature revisions of a range of Git commits from the SPLs in Git version control systems. Thus, although the benchmark comprises three systems, our ground truth extractor can be used to generate ground truth data sets and variants for any point in time.

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; CNPq, grant no. 408356/2018-9; FAPPR, grant no. 51435; and 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.

[6] `https://choco-solver.org/`

Bibliography

[1] Ra'Fat AL-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. Feature location in a collection of software product variants using formal concept analysis. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, pages 302–307, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[2] Sofia Ananieva. *Consistent Management of Variability in Space and Time*, page 7–12. Association for Computing Machinery, New York, NY, USA, 2021.

[3] Sofia Ananieva, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, S. Ramesh, Andreas Burger, Gabriele Taentzer, and Bernhard Westfechtel. 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*, pages 67:1–67:5, New York, NY, USA, 2019. ACM.

[4] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. Change impact analysis for maintenance and evolution of variable software systems. *Automated Software Engineering*, 26:417–461, June 2019.

[5] Anonymous. Change analysis, visualization and propagation tool. `https://figshare.com/s/d848ab8528065f5e66e1`, 2022.

[6] Anonymous. Dataset. `https://figshare.com/s/f508d59fd07632ab3f39`, 2022.

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

[8] Sven Apel, Don Batory, Christian Kstner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company, Incorporated, 2013.

[9] Sven Apel and Christian Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, July 2009.

[10] Sven Apel, Christian Kästner, and Christian Lengauer. FEATUREHOUSE: language-independent, automated software composition. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 221–231. IEEE, 2009.

[11] Google Code Archive. Java-diff-utils. `https://java-diff-utils.github.io/java-diff-utils/`, 2021.

[12] Wesley K. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. Reengineering legacy applications into software product lines: A systematic mapping. *Empirical Softw. Engg.*, 22(6):2972–3016, dec 2017.

[13] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. Feature location for software product line migration: A mapping study. In *18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, SPLC 2014, pages 52–59, New York, USA, 2014. ACM.

[14] Wesley K.G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. Automatic extraction of product line architecture and feature models from uml class diagram variants. *Information and Software Technology*, 117:106198, 2020.

[15] D.L. Atkins, T. Ball, T.L. Graves, and A. Mockus. Using version control data to evaluate the impact of software tools: a case study of the version editor. *IEEE Transactions on Software Engineering*, 28(7):625–637, 2002.

[16] Atlassian-Bitbucket. Git cherry pick). `https://www.atlassian.com/git/tutorials/cherry-pick`, 2021.

[17] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 134–144, San Francisco, CA, USA, 2015. IEEE Press.

[18] Don Batory. Feature-oriented programming and the ahead tool suite. In *26th International Conference on Software Engineering*, ICSE '04, page 702–703, USA, 2004. IEEE Computer Society.

[19] David Benavides, Sergio Segura, Pablo Trinidad Martín-Arroyo, and Antonio Ruiz Cortés. Using java CSP solvers in the automated analyses of feature models. In *Generative and Transformational Techniques in Software Engineering, International Summer School, (GTTSE '05)*, volume 4143 of *Lecture Notes in Computer Science*, pages 399–408, Berlin, Heidelberg, 2005. Springer.

[20] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Conference on The Future of Software Engineering*, ICSE '00, page 73–87, New York, NY, USA, 2000. ACM.

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

[22] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 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, pages 1–10, Nashville, USA, 2015. ACM.

[23] Thorsten Berger, Rolf-Helge Pfeiffer, Reinhard Tartler, Steffen Dienst, Krzysztof Czarnecki, Andrzej Wasowski, and Steven She. Variability mechanisms in software ecosystems. *Information and Software Technology*, 56(11):1520–1535, nov 2014.

[24] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. Feature-to-code mapping in two large product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 98–499, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[25] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Trans. Software Eng.*, 39(12):1611–1640, 2013.

[26] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. The state of adoption and the challenges of systematic variability management in industry. *Empir. Softw. Eng.*, 25(3):1755–1797, 2020.

[27] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, Leopoldo Teixeira, and Sabrina Souto. A change-aware per-file analysis to compile configurable systems with #ifdefs. *Computer Languages, Systems & Structures*, 54:427–450, 2018.

[28] Panuchart Bunyakiati and Chadarat Phipathananunth. Cherry-picking of code commits in long-running, multi-release software. In *2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE '17, page 994–998, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Software Engineering*, 14(1):93–130, February 2009.

[30] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, Boston, MA, 2002.

[31] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2002.

[32] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[33] Git Software Freedom Conservancy. Documentation. `https://git-scm.com/doc`, 2020.

[34] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. A literature review and comparison of three feature location techniques using argouml-spl. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '19, New York, NY, USA, 2019. ACM.

[35] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Adam Gudyś. Kalign-lcs — a more accurate and faster variant of kalign2 algorithm for the multiple sequence alignment problem. In Dr. Aleksandra Gruca, Tadeusz Czachórski, and Stanisław Kozielski, editors, *Man-Machine Interactions 3*, pages 495–502, Cham, 2014. Springer International Publishing.

[36] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In Yann-Gaël Guéhéneuc, Bram Adams, and Alexander Serebrenik, editors, *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 341–350, San Francisco, CA, USA, 2015. IEEE Computer Society.

[37] Oscar Díaz, Raul Medeiros, and Leticia Montalvillo. Change analysis of #if-def blocks with featurecloud. In *23rd International Systems and Software Product Line Conference - Volume B*, SPLC '19, pages 17–20, New York, NY, USA, 2019. ACM.

[38] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. Fever: Extracting feature-oriented changes from commits. In *13th International Conference on Mining Software Repositories*, MSR '16, page 85–96, New York, NY, USA, 2016. Association for Computing Machinery.

[39] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *Empirical Software Engineering*, 23(2):905–952, November 2017.

[40] 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, 2013.

[41] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, July 2008.

[42] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[43] A.D. Eisenberg and K. De Volder. Dynamic feature traces: finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 1–10, USA, 2005. IEEE.

[44] Sascha El-Sharkawy, Adam Krafczyk, and Klaus Schmid. 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*, VAMOS '20, New York, NY, USA, 2020. ACM.

[45] Sascha El-Sharkawy, Nozomi Yamagishi-Eichler, and Klaus Schmid. Metrics for analyzing variability and its implementation in software product lines: A systematic literature review. *Information and Software Technology*, 106:1–30, 2019.

[46] Jacky Estublier. Software configuration management: A roadmap. In *Conference on The Future of Software Engineering*, ICSE '00, page 279–289, New York, NY, USA, 2000. ACM.

[47] James D. Evans, editor. *Straightforward statistics for the behavioral sciences*, volume 1 of *Thomson Brooks*. Cole Publishing Co, California, USA, 1996.

[48] Kevin Feichtinger, Daniel Hinterreiter, Lukas Linsbauer, Herbert Prähofer, and Paul Grünbacher. Guiding feature model evolution by lifting code-level dependencies. *Journal of Computer Languages*, 63:101034, 2021.

[49] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Softw. Engg.*, 18(4):699–745, August 2013.

[50] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the Linux kernel. In *20th International Systems and Software Product Line Conference*, SPLC '16, pages 65–73, New York, NY, USA, 2016. ACM.

[51] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *30th IEEE International Conference on Software Maintenance and Evolution*, ICSME 2014, pages 391–400, New York, USA, Sep. 2014. IEEE.

[52] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. The ecco tool: Extraction and composition for clone-and-own. In *37th IEEE International Conference on Software Engineering*, volume 2 of *ICSE 2015*, pages 665–668, New York, USA, 2015. IEEE.

[53] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. A source level empirical study of features and their interactions in variable software. In *16th International Working Conference on Source Code Analysis and Manipulation*, SCAM 2016, pages 197–206, New York, USA, 2016. IEEE.

[54] Stefan Fischer, Gabriela Karoline Michelon, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. Automated test reuse for highly configurable software. *Empir. Softw. Eng.*, 25(6):5295–5332, 2020.

[55] Stefan Fischer, Gabriela Karoline Michelon, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. Automated reuse of test cases for highly configurable software systems. In Anne Koziolek, Ina Schaefer, and Christoph Seidl, editors, *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*, volume P-310 of *LNI*, pages 39–40. Gesellschaft für Informatik e.V., 2021.

[56] Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. Automating test reuse for highly configurable software. In *23rd International Systems and Software Product Line Conference*, SPLC 2019, pages 1–11, Paris, France, 2019. ACM.

[57] Angelo Gargantini, Justyna Petke, Marco Radavelli, and Paolo Vavassori. Validation of constraints among configuration parameters using search-based combinatorial interaction testing. In Federica Sarro and Kalyanmoy Deb, editors, *Search Based Software Engineering*, pages 49–63, Cham, 2016. Springer International Publishing.

[58] Lea Gerling, Sandra Greiner, Kristof Meixner, and Gabriela Karoline Michelon. Fourth international workshop on variability and evolution of software-intensive systems (varivolution 2021). In Mohammad Mousavi and Pierre-Yves Schobbens, editors, *SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A*, page 204. ACM, 2021.

[59] Eddy Ghabach, Mireille Blay-Fornarino, Franjieh El Khoury, and Badih Baz. Clone-and-own software product derivation based on developer preferences and cost estimation. In *12th International Conference on Research Challenges in Information Science*, pages 1–6. IEEE, 2018.

[60] Karine Gomes, Leopoldo Teixeira, Thayonara Alves, Márcio Ribeiro, and Rohit Gheyi. Characterizing safe and partially safe evolution scenarios in product lines: An empirical study. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '19, New York, NY, USA, 2019. ACM.

[61] Sandra Greiner, Kristof Meixner, Gabriela Karoline Michelon, and Philippe Collet. Fifth international workshop on variability and evolution of software-intensive systems (varivolution 2022). In *SPLC '22: 25th ACM International Systems and Software Product Line Conference, Graz, Austria, September 12-16, 2022, accepted*. ACM, 2022.

[62] Paul Grünbacher, Rudolf Hanl, , and Lukas Linsbauer. Using music features for managing revisions and variants in music notation software. In Rama Gottfried, Georg Hajdu, Jacob Sello, Alessandro Anatrini, and John MacCallum, editors, *International Conference on Technologies for Music Notation and Representation*, TENOR'20/21, pages 212–220, Hamburg, Germany, 2021. Hamburg University for Music and Theater.

[63] Chetna Gupta, Maneesha Srivastav, and Varun Gupta. Software change impact analysis: An approach to differentiate type of change to minimise regression test selection. *Int. J. Comput. Appl. Technol.*, 51(4):366–375, July 2015.

[64] H. Ha and H. Zhang. Performance-influence model for highly configurable software with fourier learning and lasso regression. In *35th International Conference on Software Maintenance and Evolution*, ICSME '19, pages 470–480, San Francisco, CA, USA, Sep. 2019. IEEE Press.

[65] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering*, 32(1):4–19, 2006.

[66] Alexis Henry and Youssef Ridene. *Migrating to Microservices*, pages 45–72. Springer International Publishing, Cham, 2020.

[67] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *10th Working Conference on Mining Software Repositories*, MSR '13, page 121–130, San Francisco, CA, USA, 2013. IEEE Press.

[68] Daniel Hinterreiter, Lukas Linsbauer, Kevin Feichtinger, Herbert Prähofer, and Paul Grünbacher. Supporting feature-oriented evolution in industrial automation product lines. *Concurrent Engineering: Research and Applications*, 28:265–279, 2020.

[69] Daniel Hinterreiter, Lukas Linsbauer, Paul Grünbacher, and Herbert Prähofer. Feature-oriented clone and pull for distributed development and evolution. In *14th International Conference on the Quality of Information and Communications Technology*, QUATIC '21, 2021.

[70] Daniel Hinterreiter, Lukas Linsbauer, Florian Reisinger, Herbert Prähofer, Paul Grünbacher, and Alexander Egyed. Feature-oriented evolution of automation software systems in industrial software ecosystems. In *23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2018)*, pages 107–114, Torino, Italy, 2018.

[71] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution. In *18th International Conference on Generative Programming: Concepts & Experiences*, GPCE '19, pages 115–128, Athens, Greece, 2019. ACM.

[72] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008.

[73] Wanja Hofer, Christoph Elsner, Frank Blendinger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Leviathan: Spl support on filesystem level. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 491–491, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[74] 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, April 2016.

[75] Dongpu Jin, Myra B. Cohen, Xiao Qu, and Brian Robinson. PrefFinder: Getting the right preference in configurable software systems. In *29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 151–162, New York, NY, USA, 2014. ACM.

[76] Christian Kastner, Alexander Dreiling, and Klaus Ostermann. Variability mining: Consistent semi-automatic detection of product-line features. *IEEE Trans. Softw. Eng.*, 40(1):67–82, January 2014.

[77] Christian Kästner, Salvador Trujillo, and Sven Apel. Visualizing software product line variabilities in source code. In Steffen Thiel and Klaus Pohl, editors, *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12,*

2008, Proceedings. Second Volume (Workshops), pages 303–312, Francisco, Clifornia, USA, 2008. Lero Int. Science Centre, University of Limerick, Ireland.

[78] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[79] Jongwook Kim, Don Batory, and Danny Dig. Refactoring java software product lines. In *21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, page 59–68, New York, NY, USA, 2017. ACM.

[80] Christian Kröher, Sascha El-Sharkawy, and Klaus Schmid. Kernelhaven: An open infrastructure for product line analysis. In *22nd International Systems and Software Product Line Conference - Volume 2*, SPLC '18, page 5–10, New York, NY, USA, 2018. ACM.

[81] Christian Kröher, Moritz Flöter, Lea Gerling, and Klaus Schmid. Incremental software product line verification - A performance analysis with dead variable code. *Empir. Softw. Eng.*, 27(3):68, 2022.

[82] Christian Kröher, Lea Gerling, and Klaus Schmid. Identifying the intensity of variability changes in software product line evolution. In *22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, page 54–64, New York, NY, USA, 2018. ACM.

[83] Christian Kröher, Lea Gerling, and Klaus Schmid. Identifying the intensity of variability changes in software product line evolution. In Thorsten Berger, Paulo Borba, Goetz Botterweck, Tomi Männistö, David Benavides, Sarah Nadi, Timo Kehrer, Rick Rabiser, Christoph Elsner, and Mukelabai Mukelabai, editors, *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, pages 54–64, New York, NY, USA, 2018. ACM.

[84] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[85] Jacob Krüger. Tackling knowledge needs during software evolution. In *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '19, page 1244–1246, New York, NY, USA, 2019. ACM.

[86] Jacob Krüger and Thorsten Berger. An empirical analysis of the costs of clone- and platform-oriented software reuse. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 432–444, New York, NY, USA, 2020. Association for Computing Machinery.

[87] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. Apo-games: A case study for reverse engineering variability from cloned java variants. In *22nd International Systems and Software Product Line Conference - Volume 1*, SPLC '18, page 251–256, New York, NY, USA, 2018. ACM.

[88] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: A case study of marlin. In *12th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS '18, page 105–112, New York, NY, USA, 2018. ACM.

[89] Jacob Krüger. *Understanding the re-engineering of variant-rich systems: an empirical work on economics, knowledge, traceability, and practices*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, 2021.

[90] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. Where is my feature and what is it about? a case study on recovering feature facets. *Journal of Systems and Software*, 152:239–253, 2019.

[91] Christian Kästner. Typechef. `https://ckaestne.github.io/TypeChef/`, 2013.

[92] Ramnivas Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning Publications Co., USA, 2nd edition, 2009.

[93] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *25th International Conference on Software Engineering*, ICSE '03, page 308–318, USA, 2003. IEEE Computer Society.

[94] Duc Le, Eric Walkingshaw, and Martin Erwig. #ifdef confirmed harmful: Promoting understandable software variation. In Gennaro Costagliola, Amy J. Ko, Allen Cypher, Jeffrey Nichols, Christopher Scaffidi, Caitlin Kelleher, and Brad A. Myers, editors, *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, pages 143–150, San Francisco, CA, USA, 2011. IEEE.

[95] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. *IEEE Transactions on Software Engineering*, 44(2):182–201, 2018.

[96] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE'10, pages 105–114, New York, NY, USA, 2010. ACM.

[97] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. Using traceability for incremental construction and evolution of software product portfolios. In *8th International Symposium on Software and Systems Traceability*, SST 2015, pages 57–60, New York, USA, 2015. IEEE.

[98] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A classification of variation control systems. In Matthew Flatt and Sebastian Erdweg, editors, *16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 49–62, New York, NY, USA, 2017. ACM.

[99] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. A classification of variation control systems. In Matthew Flatt and Sebastian Erdweg, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, pages 49–62, New York, USA, 2017. ACM.

[100] Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. A variability aware configuration management and revision control platform. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *38th International Conference on Software Engineering*, ICSE '16, pages 803–806. ACM, 2016.

[101] Lukas Linsbauer, Stefan Fischer, Gabriela K. Michelon, Wesley K. G. Assunção, Paul Grünbacher, Roberto Erick Lopez-Herrejon, and Alexander Egyed. *Systematic Software*

Reuse with Automated Extraction and Composition for Clone-and-Own, pages 369–393. Springer International Publishing, Cham, 2022. Accepted.

[102] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. Recovering traceability between features and code in product variants. In *17th International Software Product Line Conference*, SPLC 2013, page 131–140, New York, USA, 2013. ACM.

[103] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. Variability extraction and modeling for product variants. *Software and Systems Modeling*, 16(4):1179–1199, 2017.

[104] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. Concepts of variation control systems. *J. Syst. Softw.*, 171:110796, 2021.

[105] Lukas Linsbauer, Felix Schwägerl, Thorsten Berger, and Paul Grünbacher. Concepts of variation control systems. *Journal of Systems and Software*, 171:110796, 2021.

[106] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 234–243, New York, NY, USA, 2007. ACM.

[107] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *28th International Conference on Software Engineering*, ICSE 2006, page 112–121, New York, USA, 2006. ACM.

[108] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, Inc., 2012.

[109] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the ecos kernel. In Yolande Berbers and Willy Zwaenepoel, editors, *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, pages 191–204. ACM, 2006.

[110] Kai Ludwig, Jacob Krüger, and Thomas Leich. Covert and phantom features in annotations: Do they impact variability analysis? In *23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, page 218–230, New York, NY, USA, 2019. ACM.

[111] Stephen A. MacKay. The state of the art in concurrent, distributed configuration management. In *Selected Papers from the ICSE SCM-4 and SCM-5 Workshops, on Software Configuration Management*, page 180–193, Berlin, Heidelberg, 1995. Springer-Verlag.

[112] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. *Introduction to information retrieval*. Cambridge university press, Cambridge, England, 2008.

[113] Luciano Marchezan, Wesley K. G. Assunção, Gabriela K. Michelon, Edvin Herac, and Alexander Egyed. Code smell analysis in cloned java variants: the apo-games case study. In *Under Submission*, 26th ACM International Systems and Software Product Line Conference, page 1–5, New York, NY, USA, 2022. ACM.

[114] Jabier Martinez. *Mining Software Artefact Variants for Product Line Migration and Analysis*. PhD thesis, Pierre and Marie Curie University, France, 2016.

[115] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-up adoption of software product lines: A generic and extensible approach. In *19th International Conference on Software Product Line*, SPLC '15, page 101–110, New York, NY, USA, 2015. ACM.

[116] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants. *Information and Software Technology*, 104:46 – 59, 2018.

[117] James McGovern, Scott W. Ambler, Michael E. Stevens, James Linn, Elias K. Jo, and Vikas Sharan. *The Practical Guide to Enterprise Architecture*. Prentice Hall PTR, USA, 2003.

[118] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, May 2018.

[119] Flávio Medeiros, Christian Kaestner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The love/hate relationship with the C preprocessor: An interview study. In *29th European Conference on Object-Oriented Programming*, volume 37 of *ECOOP '15*, pages 495–518, Prague, CZE, 2015. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[120] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. Exploring differences and commonalities between feature flags and configuration options. In *42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, page 233–242, New York, NY, USA, 2020. ACM.

[121] Jean Melo, Claus Brabrand, and Andrzej Wasowski. How does the degree of variability affect bug finding? In *38th International Conference on Software Engineering*, ICSE '16, page 679–690, New York, NY, USA, 2016. ACM.

[122] Willian D. F. Mendonça, Silvia R. Vergilio, Gabriela K. Michelon, Alexander Egyed, and Wesley K. G. Assunção. Test2feature: Feature-based test traceability tool for highly configurable software. In *Under Submission*, 26th ACM International Systems and Software Product Line Conference, page 1–4, New York, NY, USA, 2022. ACM.

[123] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time, April 2021.

[124] Gabriela K. Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley K. G. Assunção. *Spectrum-Based Feature Localization: A Case Study Using ArgoUML*, page 126–130. SPLC '21. ACM, New York, NY, USA, 2021.

[125] Gabriela Karoline Michelon. Evolving system families in space and time. In Rafael Capilla, Philippe Collet, Paul Gazzillo, Jacob Krüger, Roberto Erick Lopez-Herrejon, Sarah Nadi, Gilles Perrouin, Iris Reinhartz-Berger, Julia Rubin, and Ina Schaefer, editors, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020*, pages 104–111, New York, NY, USA, 2020. ACM.

[126] Gabriela Karoline Michelon, Wesley Klewerton Guez Assunção, Paul Grünbacher, and Alexander Egyed. Propagating feature revisions in preprocessor-based software product lines. In *Under Submission*, Conference 2022, page 1–12, New York, NY, USA, 2022. ACM.

[127] Gabriela Karoline Michelon, Wesley Klewerton Guez Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. The life cycle of features in highly-configurable software systems evolving in space and time. In *20th International Conference on Generative Programming: Concepts & Experiences*, GPCE 2021, page 1–10, New York, NY, USA, 2021. Association for Computing Machinery.

[128] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, Stefan Fischer, and Alexander Egyed. A hybrid feature location technique for re-engineering single systems into software product lines. In Paul Grünbacher, Christoph Seidl, Deepak Dhungana, and Helena Lovasz-Bukvova, editors, *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, VaMoS '21, pages 11:1–11:9. ACM, 2021.

[129] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. Comparison-based feature location in argouml variants. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, page 93–97, New York, NY, USA, 2019. Association for Computing Machinery.

[130] Gabriela Karoline Michelon, Jabier Martinez, Bruno Sotto-Mayor, Aitor Arrieta, Wesley Klewerton Guez Assunção, Rui Abreu, and Alexander Egyed. Spectrum-based feature localization for families of systems. *Journal of Systems and Software*, 2022. Under Revision.

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

[132] Gabriela Karoline Michelon, David Obermann, Wesley Klewerton Guez Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 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*, pages 74–78, New York, NY, USA, 2020. ACM.

[133] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Evolving software system families in space and time with feature revisions. *Empirical Software Engineering*, 27(5):1–54, May 2022.

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

[135] Leticia Montalvillo and Oscar Díaz. Tuning github for spl development: Branching models & repository operations for product engineers. In *19th International Conference on Software Product Line*, SPLC '15, page 111–120, New York, NY, USA, 2015. Association for Computing Machinery.

[136] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. Multi-view editing of software product lines with peopl. In *40th*

International Conference on Software Engineering: Companion Proceeedings, ICSE '18, page 81–84, New York, NY, USA, 2018. Association for Computing Machinery.

[137] Mukelabai Mukelabai, Damir Nešiundefined, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. Tackling combinatorial explosion: A study of industrial needs and practices for analyzing highly configurable systems. In *33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 155–166, New York, USA, 2018. ACM.

[138] Richard Müller and Ulrich Eisenecker. A graph-based feature location approach using set theory. In *23rd International Systems and Software Product Line Conference - Volume A*, SPLC 2019, page 88–92, New York, , USA, 2019. ACM.

[139] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. Identifying software performance changes across variants and versions. In *35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 1–12, New York, NY, USA, 2020. ACM.

[140] Sarah Nadi and Richard C. Holt. Mining kbuild to detect variability anomalies in linux. In Tom Mens, Anthony Cleve, and Rudolf Ferenc, editors, *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, pages 107–116, USA, 2012. IEEE Computer Society.

[141] Mathieu Nassif and Martin P. Robillard. Revisiting turnover-induced knowledge loss in software projects. In *2017 IEEE International Conference on Software Maintenance and Evolution*, ICSME '17, pages 261–272. IEEE Computer Society, 2017.

[142] Laís Neves, Leopoldo Teixeira, Demóstenes Sena, Vander Alves, Uirá Kulesza, and Paulo Borba. Investigating the safe evolution of software product lines. In *10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, page 33–42, New York, NY, USA, 2011. ACM.

[143] Damir Nešić, Jacob Krüger, undefinedtefan Stănciulescu, and Thorsten Berger. Principles of feature modeling. In *2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '19, page 62–73, New York, NY, USA, 2019. Association for Computing Machinery.

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

[145] Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. Anomaly analyses for feature-model evolution. In *17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '18, page 188–201, New York, NY, USA, 2018. ACM.

[146] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-oriented software evolution. In *Seventh International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '13, New York, NY, USA, 2013. ACM.

[147] 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*, SPLC '13, page 91–100, New York, NY, USA, 2013. ACM.

[148] Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. Feature scattering in the large: A longitudinal study of linux kernel device drivers. In *14th International Conference on Modularity*, MODULARITY 2015, page 81–92, New York, NY, USA, 2015. ACM.

[149] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Alejandro Padilla. A study of feature scattering in the linux kernel. *IEEE Transactions on Software Engineering*, 47(1):146–164, 2021.

[150] 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. *Empirical Software Engineering*, 21(4):1744–1793, May 2015.

[151] 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. *Empirical Softw. Engg.*, 21(4):1744–1793, August 2016.

[152] Leonardo Teixeira Passos. *Towards a Better Understanding of Variability Evolution.* PhD thesis, University of Waterloo, Ontario, Canada, 2016.

[153] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. Sampling effect on performance prediction of configurable systems: A case study. In José Nelson Amaral, Anne Koziolek, Catia Trubiani, and Alexandru Iosup, editors, *International Conference on Performance Engineering*, ICPE '20, pages 277–288. ACM, 2020.

[154] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, Berlin, Heidelberg, 2005.

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

[156] Klaus Pohl and Andreas Metzger. *Software Product Lines*, pages 185–201. Springer International Publishing, Cham, 2018.

[157] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, June 2007.

[158] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Solver Documentation.* TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[159] 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 *6th International Workshop on Feature-Oriented Software Development*, FOSD '14, page 23–29, New York, NY, USA, 2014. ACM.

[160] Rodrigo Queiroz, Leonardo Passos, Tulio Marco 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 (SoSyM)*, 16:77–96, 2017.

[161] Rick Rabiser, Paul Grünbacher, and Martin Lehofer. A qualitative study on user guidance capabilities in product configuration tools. In *International Conference on Automated Software Engineering*, ASE '12, pages 110–119. ACM, 2012.

[162] Md Tajmilur Rahman, Louis-Philippe Querel, Peter C. Rigby, and Bram Adams. Feature toggles: Practitioner practices and a case study. In *13th International Conference on Mining Software Repositories*, MSR '16, page 201–211, New York, NY, USA, 2016. ACM.

[163] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. The state of empirical evaluation in static feature location. *ACM Trans. Softw. Eng. Methodol.*, 28(1), December 2018.

[164] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. The state of empirical evaluation in static feature location. *ACM Transactions on Software Engineering and Methodology*, 28(1):1–58, February 2019.

[165] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, October 2004.

[166] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*, pages 29–58. Springer Berlin Heidelberg, Heidelberg, DE, 2013.

[167] Ripon K. Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E. Perry. On the effectiveness of information retrieval based bug localization for C programs. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 161–170, New York, USA, 2014. IEEE Computer Society.

[168] Gabriela Sampaio, Paulo Borba, and Leopoldo Teixeira. Partially safe evolution of software product lines. *Journal of Systems and Software*, 155:17–42, 2019.

[169] Trevor Savage, Meghan Revelle, and Denys Poshyvanyk. Flat³: Feature location and textual tracing tool. In *32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, page 255–258, New York, NY, USA, 2010. ACM.

[170] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2010.

[171] Felix Schwägerl. *Version Control and Product Lines in Model-Driven Software Engineering*. PhD thesis, University of Bayreuth, Germany, 2018.

[172] Felix Schwägerl and Bernhard Westfechtel. Supermod: tool support for collaborative filtered model-driven software product line engineering. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *31st International Conference on Automated Software Engineering*, ASE '16, pages 822–827. ACM, 2016.

[173] Felix Schwägerl and Bernhard Westfechtel. Integrated revision and variation control for evolving model-driven software product lines. *Software and Systems Modeling*, 18(6):3373–3420, February 2019.

[174] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Capturing variability in space and time with hyper feature models. In *8th International Workshop on Variability Modelling of Software-Intensive Systems*, VAMOS 2014, pages 6:1–6:8, New York, USA, 2013. ACM.

[175] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. Deltaecore – A model-based delta language generation framework. In Hans-Georg Fill, Dimitris Karagiannis, and Ulrich Reimer, editors, *Modellierung 2014*, volume P-225 of *LNI*, pages 81–96. GI, 2014.

[176] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, dec 1965.

[177] Ridwan Shariffdeen, Xiang Gao, Gregory J. Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. Automated patch backporting in linux (experience paper). In Cristian Cadar and Xiangyu Zhang, editors, *30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, pages 633–645, New York, NY, USA, 2021. ACM.

[178] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. Automated patch transplantation. *ACM Trans. Softw. Eng. Methodol.*, 30(1), dec 2021.

[179] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is The Linux Kernel a Software Product Line? In Frank van der Linden and Björn Lundell, editors, *International Workshop on Open Source Software and Product Lines*, SPLC-OSSPL '07, Kyoto, Japan, 2007.

[180] Julio Sincero, Reinhard Tartler, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Efficient extraction and analysis of preprocessor-based variability. In *9th International Conference on Generative Programming and Component Engineering*, GPCE '10, page 33–42, New York, NY, USA, 2010. ACM.

[181] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-clr: A tool for navigating highly configurable system software. In *6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '07, page 9–es, New York, NY, USA, 2007. Association for Computing Machinery.

[182] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[183] SQLite. What is sqlite? `https://sqlite.org/index.html`, 2022.

[184] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wasowski. Concepts, operations, and feasibility of a projection-based variation control system. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 323–333, San Francisco, CA, USA, 2016. IEEE Computer Society.

[185] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. Facing the truth: Benchmarking the techniques for the evolution of variant-rich systems. In *23rd International Systems and Software Product Line Conference - Volume A*, SPLC '19, page 177–188, New York, NY, USA, 2019. ACM.

[186] Stefan Strüder, Mukelabai Mukelabai, Daniel Strüber, and Thorsten Berger. Feature-oriented defect prediction. In *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA, 2020. ACM.

[187] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *6th Conference on Computer Systems*, EuroSys '11, page 47–60, New York, NY, USA, 2011. ACM.

[188] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1), jun 2014.

[189] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. Towards efficient analysis of variation in time and space. In *23rd International Systems and Software Product Line Conference*, SPLC '19, page 57–64, New York, NY, USA, 2019. ACM.

[190] Kai Ming Ting. *Precision and Recall*. Springer US, Boston, MA, 2010.

[191] Tassio Vale and Eduardo Santana Almeida. Experimenting with information retrieval methods in the recovery of feature-code SPL traces. *Empirical Software Engineering*, 24(3):1328–1368, June 2019.

[192] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. Presence-condition simplification in highly configurable systems. In *37th International Conference on Software Engineering - Volume 1*, ICSE 2015, pages 178–188, New York, USA, 2015. IEEE.

[193] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, USA, 2000.

[194] Sai Zhang, Zhongxian Gu, Yu Lin, and Jianjun Zhao. Change impact analysis for aspectj programs. In *2008 IEEE International Conference on Software Maintenance*, pages 87–96, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[195] Shurui Zhou, Stefan Stanciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kästner. Identifying features in forks. In *40th International Conference on Software Engineering*, ICSE '18, page 105–116, New York, NY, USA, May 2018. ACM.