Mining Feature Revisions in Highly-Configurable Software Systems

Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley Klewerton Guez Assunção[3], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]
[1]Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria
[2]LIT Secure and Correct Systems Lab, Johannes Kepler University Linz, Austria
[3]COTSI, Federal University of Technology - Paraná, PPGComp, Western Paraná State University, Brazil
[4]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany

ABSTRACT

Highly-Configurable Software Systems (HCSSs) support the systematic evolution of systems in space, i.e., the inclusion of new features, which then allow users to configure software products according to their needs. However, HCSSs also change over time, e.g., when adapting existing features to new hardware or platforms. In practice, HCSSs are thus developed using both version control systems (VCSs) and preprocessor directives (`#ifdefs`). However, the use of a preprocessor as variability mechanism has been criticized regarding the separation of concerns and code obfuscation, which complicates the analysis of HCSS evolution in VCSs. For instance, a single commit may contain changes of totally unrelated features, which may be scattered over many variation points (`#ifdefs`), thus making the evolution history hard to understand. This complexity often leads to error-prone changes and high costs for maintenance and evolution. In this paper, we propose an automated approach to mine HCSS features taking into account evolution in space and time. Our approach uses constraint satisfaction problem solving to mine newly introduced, removed and changed features. It finds a configuration containing the feature revisions which are needed to activate a specific program location. Furthermore, it increments the revision number of each changed feature. Thus, our approach enables to analyze when and which features often change over time, as well as their interactions, for every single commit of a HCSS. Our approach can contribute to future research on understanding the characteristics of HCSS and supporting developers during maintenance and evolution tasks.

CCS CONCEPTS

• **Software and its engineering** → **Preprocessors**; **Software product lines**; *Traceability*; *Reusability*.

KEYWORDS

system evolution, software product lines, preprocessors, feature evolution, version control systems, repository mining

SPLC '20 Companion, October 19–23, 2020, MONTREAL, QC, Canada

ACM ISBN 978-1-4503-7570-2/20/10…$15.00
https://doi.org/10.1145/3382026.3425776

ACM Reference Format:
Gabriela Karoline Michelon[1,2], David Obermann[1], Wesley Klewerton Guez Assunção[3], Lukas Linsbauer[4], Paul Grünbacher[1], Alexander Egyed[1]. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *24th ACM International Systems and Software Product Line Conference Companion (SPLC '20 Companion), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3382026.3425776

1 INTRODUCTION

A software system usually must be delivered with different configurations of features, with each feature representing a functionality of the system accessible to developers and users. To remain competitive, companies have to satisfy different customer needs of the market segment they serve. Software Product Line (SPL) engineering is a systematic approach to deal with the development of customized system products. An SPL is a set of software-intensive systems that share a common set of artifacts developed in a prescribed way to facilitate their systematic reuse [7]. The customized software products, a.k.a. variants, result from the derivation of SPL artifacts, i.e., the selection of a different set of features that are of interest to a customer. To allow customization, the features of an SPL is implemented using variability mechanisms [1].

A widely used variability mechanism in SPLs is based on annotations [26]. Annotations rely on preprocessor directives such as `#ifdef` and `#endif` which enclose blocks of variable code and enable to tailor system variants to different hardware platforms, operating systems, and application scenarios [23]. Annotation-based SPLs are often implemented as Highly-Configurable Software Systems (HCSSs) [16]. HCSSs use techniques such as feature flags, feature toggles, or feature switches, to turn on configuration options/features needed to be included in a product [8, 18, 27]. However, features also need to evolve over time. For instance, when a specific feature is adapted to a new hardware platform, then a new version of a variant is created. This evolution in time [30] is aided by some tools such as version control systems (VCSs) [28].

However, despite the benefits of managing HCSSs in VCSs, they are hardly integrated to support both evolution in space and time [21, 22]. For example, when evolving HCSSs in VCSs, developers often commit unrelated or loosely related implementations of features [13]. Then, evolving a particular feature requires to find the implementation artifacts over many `#ifdefs`, compromising code comprehension and complicating maintenance and evolution tasks [9].

In this paper, we present an automated approach[1] for mining HCSSs managed in VCSs to obtain information of the evolution of features in both space and time. For every repository commit, we mine the features that were introduced, changed, and removed. Thus, our approach enables to automatically retrieve the features that evolved in each point in time for every change in the code. The approach takes into account all subsequent lines of code and solves a Constraint Satisfaction Problem (CSP) [5, 29] to assign feature revisions to a specific changed block of code. In addition, our approach finds a configuration for every changed block of code in a Git commit that activates that specific program location, thereby easing the analysis of feature interactions [2].

2 MOTIVATING EXAMPLES

The complexity of HCSSs implementation often makes maintenance, evolution, and testing activities time consuming and error-prone tasks. This happens mainly because source code cluttered with preprocessor directives is difficult to understand [14]. Complex systems have many features which are annotated across many files, and which often depend on or interact with other features. This makes it hard, for instance, to determine without an automated mechanism which specific feature has a bug or causes other faults.

Imagine an HCSS managed in a VCS, which has been evolved for a while. If a bug is reported by the users, the developers need to find where and when the bug was introduced and which features it affects. Developers fixing the bug may need to look through the entire VCS version history to find the commit introducing the defective code. However, manually retrieving the changes related to the desired feature is a complex task, especially when multiple features are changed or added in a single commit [4, 17, 31].

Concrete examples can be found in the commits of the LibSSH[2] system. Analyzing the version history we can see that many changes were made in a single commit (77603db), containing changes of refactoring, cleanup debugging messages, inclusion and enhancing of features, and bug fixing. In this same commit, 15 files were changed, representing a total of 415 additions and 338 deletions. To associate the different changes to specific features, firstly, we have to analyze which features really changed. Performing a manual analysis over all the files requires also to analyze each `#ifdef` block as well as `#defines` and `#includes` directives in the code to correctly assign a change to a feature. Doing this manually can become infeasible and a cumbersome when features have high degrees of scattering, tangling, and nesting [26].

It has been shown that the commit messages often do not reflect the actual changes performed [4]. For instance, the commit 6f47401 of the LibSSH system contains the code implementing the feature `HAVE_SSH1` but also changed the feature name in the `#ifdef` annotation to `WITH_SSH1`. This kind of changes can easily lead to a misunderstanding of features. For example, if a customer using a system version delivered before this name change reports a bug, and the developers try to find the 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 is actually located in the feature `WITH_SSH1`.

[1]https://github.com/GabrielaMichelon/git-ecco
[2]https://gitlab.com/libssh/libssh-mirror/

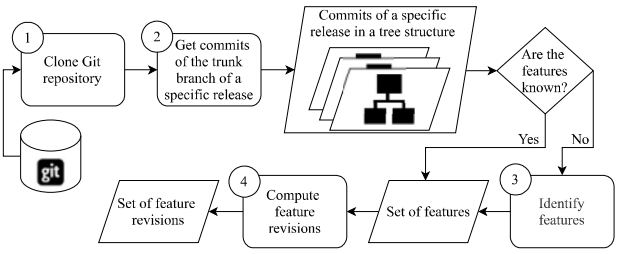

Figure 1: Approach overview.

The examples show that recovering feature implementations is a complex and costly task in HCSSs evolution, which suggests an automated mechanism to retrieve the added and removed features as well as their changes, i.e., revisions.

3 APPROACH

We present an approach for mining feature revisions of HCSSs, which are managed in VCSs. We describe its main steps, input, and output, as well as the internal representation of artifacts. From now on, we refer to feature revision as the change in a feature at a specific point in time, i.e., in a Git commit. Figure 1 presents an overview of our automated approach. Firstly, the Git repository of the HCSS is cloned (step 1) and all commits of the main branch of a specific release, i.e., a Git tag, are retrieved and represented in a tree-like structure (step 2). Since the goal of our approach is to mine feature revisions, we must know the HCSS features. If they are known in advance, they are provided as input for step 4, if not, step 3 is responsible for automatically identifying the features implemented in the HCSS by exploring the tree structure containing the files and source code of each commit of the release. In step 4, our approach performs the process of assigning feature revisions to the changes.

Below, we describe in detail how the tree-like representation of the artifacts is created and we explain the steps of identifying features (step 3) and computing feature revisions (step 4). To exemplify these activities, we use the running example presented in Listing 1.

Artifact representation. Existing tools such as TypeChef [20], SuperC [10], and KernelHaven [19] allow transforming systems that are implemented in C and annotated with preprocessor directives into an Abstract Syntax Tree (AST). In addition, TypeChef and SuperC represent the variability in the AST in the form of choice nodes. However, for our purpose, we only need nodes at the level of preprocessor directives and it would be computationally too expensive and time-consuming to analyze all commits of a system at the AST level. Thus, we decided to create our own tree structure suitable for our approach, which only needs to distinguish the preprocessor directives to easily build constraints for CSP problems and to identify the features of our subject systems, which do not have a variability model and tristate type such as the Linux Kernel.

Therefore, in our approach, the artifacts, i.e., the source code and any other files, are represented based on a tree-like structure. For this, we assume that conditional blocks wrap code that may belong to one feature, multiple features, or no feature. An example of such code blocks is presented in Listing 1. The Lines 1-4 are part