Analysis and Propagation of Feature Revisions in Preprocessor-based Software Product Lines

Gabriela K. Michelon
ISSE and LIT Lab
Johannes Kepler University
Linz, Austria
gabriela.michelon@jku.at

Wesley K. G. Assunção
ISSE
Johannes Kepler University
Linz, Austria
wesley.assuncao@jku.at

Paul Grünbacher
ISSE
Johannes Kepler University
Linz, Austria
paul.gruenbacher@jku.at

Alexander Egyed
ISSE
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

***Abstract*—Preprocessor-based software product lines (SPLs) are used 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 releases. 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 active releases. However, there is no automated support for analyzing and propagating features in SPL releases. For instance, VCSs can only propagate changes at the commit level, but miss support at the feature level, i.e., the building blocks of SPLs. Manually analyzing and propagating a version of a feature, i.e., a feature revision, through #ifdefs is risky, time-consuming, and error-prone because a feature can be interacting with multiple features and it can be spread in multiple blocks of code across different files. We thus present a novel and tool-supported approach for the analysis and propagation of feature revisions. We evaluated our approach quantitatively by computing its correct behavior and runtime. Our approach analyzes and propagates a feature implementation in $\approx$63 seconds, with, on average, precision and recall of 99%. In total, we propagated 3,134 features in space and time between 200 pairs of releases on four real-world preprocessor-based SPLs. In addition, we qualitatively evaluated the usefulness of our tool support by conducting interviews with five experienced core developers of three popular preprocessor-based SPLs. The qualitative results confirm that our tool support is useful to speed up the analysis and propagation of feature revisions.**

***Index Terms*—variability management, feature propagation, version control system, preprocessor directives, software reuse**

I. Introduction

Preprocessor-based software product lines (SPLs) are implemented with preprocessor directives to provide a reuse-oriented platform with common and variable code from which multiple software variants are derived [1], [2], [3]. The preprocessor directives delineate features with #ifdefs, which can represent end-user functionality or are used for development purposes, such as testing, debugging, and deployment [4], [5]. SPLs are widely implemented with preprocessor directives [6], the most common mechanism used to deal with *evolution in space*, i.e., introducing or removing features of a system [7]. Preprocessor-based SPLs commonly evolve over time in addition to space, i.e., features are revised besides being introduced and deleted [7]. This *evolution in time* can result in different releases of an SPL, where different releases can contain the same feature but with different implementations [8]. The different versions of a feature are called *feature revisions* [9], [10], [11]. Currently, the most common mechanism to manage evolution of systems over time are Version Control Systems (VCSs) [11], e.g., Git. SPLs implemented with preprocessor directives integrate very well with VCSs, as the preprocessor directives are basically pieces of code [7]. However, the evolution over time of preprocessor-based SPLs in VCSs is tracked for the whole platform at the level of commits [7], [12] and there are several studies acknowledging the importance of tracking feature revisions [9], [10], [11], [13], [14], [15], [16].

Due to evolution in space and time developers have to maintain different releases/branches of an SPL. Thus, over time, it can be necessary to propagate revisions of a feature between SPL releases because of bug fixes, refactoring, and enhancements of features [10]. A recent study confirms that feature propagation is very challenging and expensive in preprocessor-based SPLs managed in VCSs, and automated support is lacking [5]. We further empirically validated with experienced developers of preprocessor-based SPLs in VCSs that the analysis and propagation of feature revisions is currently a hard manual, and time consuming task (details in Section V). Propagating an entire feature revision implementation between two SPL releases, requires laborious analyses of which feature interactions, files, and patches of code, i.e., sequences of lines of the source code belonging to a feature are differing. However, VCSs do not provide analysis and visualization of which features are interacting and affected between different releases of a preprocessor-based SPL [8]. This is a hard task because preprocessor directives make source code harder to understand and maintain due to limited separation of features, and code obfuscation [8], [17], [18], [19]. Furthermore, while VCSs support propagation of commit-level changes, they miss support for the feature level [20]. These limitations and challenges for analysis and propagation during evolution in space and time confirm the need for a novel and automated solution [8], [13].

In this work, we define a tool-supported approach for the analysis and propagation of feature revisions between releases of preprocessor-based SPLs in VCSs. We evaluated the correct behavior and runtime performance of our approach in a large dataset containing 200 target releases of four real-world preprocessor-based SPLs in VCSs. The results show that our approach takes $\approx$63 seconds on average to correctly analyze the differences of a feature's source code in multiple files, patches of code, feature interactions, and to propagate features in space and time between two SPL releases. In addition, we qualitatively evaluated the usefulness of our tool support. For the qualitative analyses, we conducted interviews with five developers of three preprocessor-based SPLs in VCSs, and presented concrete evidence of the problem of analysis and propagation of feature revisions between releases. Developers considered our tool helpful and confirmed that such automation would speed up the tasks of analysis and propagation, indicating the potential of our approach for practice. Furthermore, the developers gave recommendations on what can be improved in our tool to better support the tasks.

In summary, we make the following contributions: (i) an approach for the analysis and propagation of feature revisions in preprocessor-based SPLs managed in VCSs; (ii) a non-intrusive tool [21] that can be integrated into the existing development of preprocessor-based SPLs in VCSs to support developers in propagating features in space and time; and (iii) a dataset [22] containing information of the features propagated in 200 target releases from four real-world preprocessor-based SPLs for replication and future work.

II. Motivation

When studying the evolution of features in real-world preprocessor-based SPLs in VCSs, we found that feature revisions existing in a specific release are propagated to other releases where they do not exist or have a different implementation. Initially, it was a shred of evidence we found out by exploring the history of changes in the repositories of preprocessor-based SPLs in VCSs. Afterward, we interviewed developers of the systems analyzed (Section IV) to empirically confirm that our assumption is a real problem for developers. We now discuss in this section some challenges confirmed by developers of the systems of our two motivational examples.

The first motivational example is from the SPL Marlin, an open-source firmware for 3D printers. In a pull request from Marlin[1] users were complaining that the feature `BLTouch V3.0` in release 2.0.x does not exist in 1.1.x, and thus nobody buying a new BL-Touch V3.0 would be able to use the release v1.1.9. The pull request propagated changes involving 114 files, 3,208 lines added and 911 deleted, and around 1,100 preprocessor directives. The second motivational example comes from the SPL SQLite, an SQL database engine, where a feature for testing purposes was propagated over four releases[2]. From these examples of real scenarios and developers' experiences, we observed two main challenges:

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

Challenge 1: Analysis of the feature interactions and differences of a feature's source code between releases. In the interviews, developers revealed that they first need to understand the nature of the feature, for example, if it is a new API function or adding a new module. It is then necessary to analyze all the source code required by a particular feature and the different lines of code respective to the feature between two releases. The analysis of the deltas, i.e., lines of code differing, can be done by the diff command of a VCS. However, developers reported that a diff between two releases might involve thousands of lines, and, for example, VCSs only show a diff within a file that changed with no support for keeping track of all the #ifdefs. One of the developers of SQLite, highly experienced with propagation of features, explicitly said that he had to propagate features multiple times over multiple releases and that he spends hours with the analyses of features through #ifdefs. Further, he mentioned that it is hard to understand feature interactions and it can become particularly complicated when combining negative features, i.e., #nifdefs. One of the Marlin developers mentioned that using Git and the integrated development environment (IDE) does not allow to see the big picture of changes related to a feature implemented in several #ifdefs and files.

Challenge 2: Propagation of a feature's source code between releases. All developers stated that there is no tool for the propagation of features, which is a manual process of copy-and-paste. Developers mentioned that although a disciplined project with one isolated change at a time allows easier propagation with the cherry-picking command from VCSs, this is still an imperfect strategy. For example, cherry-picking does not substitute the propagation of source code related to many files and related to many commits that you cannot reuse all the changes. Therefore, developers report that currently, the propagation of features is a manual, laborious, and time-consuming task.

III. Approach

Our approach comprises the analysis and propagation of feature revisions, as illustrated in Figure 1. It enables the propagation of the source code of a specific feature revision between different SPL releases. Before performing the analysis and propagation of feature revision(s), our approach needs as input the set of features existing in each release, the feature(s) to be propagated from one release to another, and the snapshot of each release, i.e., the files of their current state. In case the developer does not know the current features of each release, our approach has an optional step (Step ①, *mining releases' features* in Figure 1), for mining the set of feature revisions existing in a release. The result of Step ① is a feature-to-release mapping. To retrieve the features existing at a specific point in time, our approach retrieves for each conditional block of code wrapped by preprocessor directives, i.e., #ifdefs, which feature(s) belong to it. Step ② (*feature revision analysis* in Figure 1) 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)

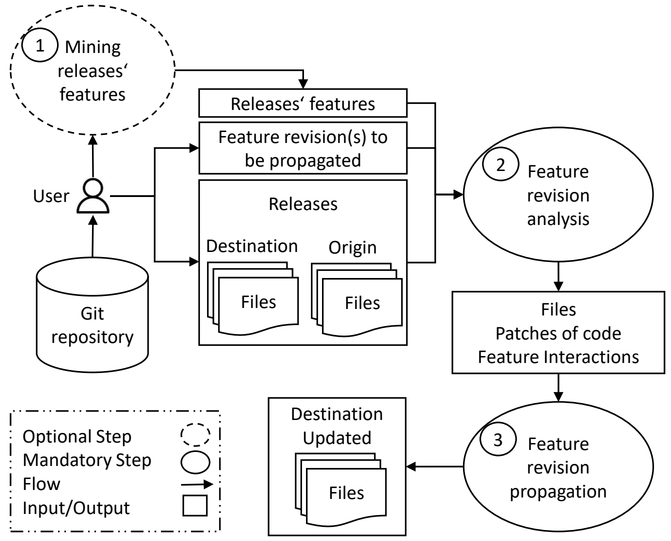
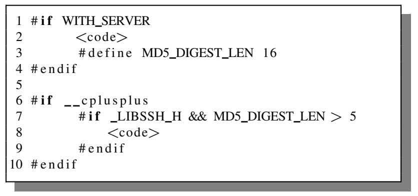
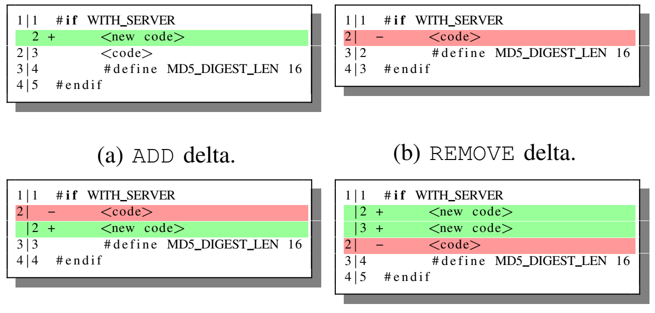


Figure 1: Approach overview for analysis and propagation of feature revisions.



```
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.



```
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.

to be propagated. Then, Step ③ (*feature revision propagation* in Figure 1) is performed after knowing which files and patches of code and feature interactions will be affected by propagating a feature revision. The result of this step are then the files of a destination release containing the feature revision(s) propagated.

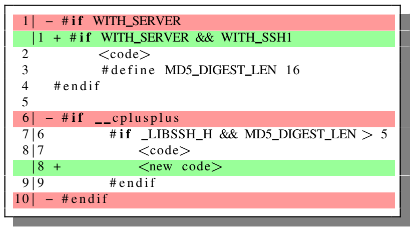
① Mining Releases' Features (Optional Step)

The mining step retrieves which features were revised in which commits of a release. For every patch of code that differs between different releases, called here 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 activating a changed block. Thus, all macros (feature revisions or not) are taken into account to build constraints influencing activating a changed block. These constraints are then handed over to a constraint satisfaction problem (CSP) solver [23], which finds a solution defining 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 interactions and dependencies between features [24].

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 that 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 enables 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 that is 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 is `WITH_SERVER`, `__cplusplus`, and `_LIBSSH_H`. After the features and interactions are obtained, the revised feature is computed according to the heuristic from Michelon et al. [10]. A feature is *revised* when it has blocks of code before the commit changes where its blocks of source code have been changed. The feature is *deleted* when no conditional blocks, i.e., #ifdefs involving the feature, exist anymore after a commit. A feature is *introduced* when a commit 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 revised features 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`.

② Feature Revision Analysis

This step starts with the computation of the differences



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

between two arbitrary releases, i.e., between the last commit of each arbitrary release, referred to as *origin* O and *destination* D. The commit O is related to the snapshot of a point in time of the source code containing the feature revision to be propagated and D is related to 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. A 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 (Line 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 (Line 2-2 in D, Figure 3c, and Line 2-2 in D, Figure 3d).

Next, our approach gets the conditional blocks of code corresponding to added, removed, and changed deltas respective to the feature(s) to be propagated. Thus, the output of the feature revision analysis contains the conditional blocks of code with deltas respective to the feature(s) that can be propagated. For each delta, the feature revision analysis also describes the feature interactions and the features that might be affected, i.e., nested features in conditional blocks of code, which is essential for preprocessing the source code [1], [6], [25]. For instance, using our running example (Figure 2), in Figure 4, we present three deltas modifying the code snippet adapted from LibSSH: The first delta (Line 1, Figure 4) is a `CHANGE` delta, where Line 1 in D is removed, which contains `#if WITH_SERVER` conditional expression and a new line added (Line 1 in O, Figure 4) to substitute this conditional expression by another conditional expression `#if WITH_SERVER && WITH_SSH1`. The feature revision analysis finds which features were introduced, revised, and deleted. For this delta (Line 1 removed in D and Line 1 added in O, Figure 4), `WITH_SSH1` feature was introduced. 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 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 Lines 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 an interaction with the code of the core of the system (`BASE`). The feature `_LIBSSH_H` might also 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 *Feature Revision Propagation*

The feature revision propagation is based on the selection of deltas obtained by the *Feature Revision 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 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 the 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 line number 2 was "<code>" and now it is in the third line of the file in D because a new line content has been added to the second line of the D file. The `REMOVE` delta contains one reference with one line removed (Line 2 in D, Figure 3b), which was Line 2 of the 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 line number 2 existing in D, and thus line number 2 will have the content of line number 3 in Figure 2.

The `CHANGE` delta contains two references: in the `CHANGE` delta of Figure 3c, the first reference contains the 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, line number 1 does not change in D, and the second line is removed. Then line number 2 in O is copied to 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 the line(s) removed (Line 2-2 in O). Therefore, line number 1 does not change in D, and the second line in D is substituted by the second line in O. Lastly, 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.

IV. Evaluation

We performed a quantitative evaluation of the correctness and runtime performance of our approach and a qualitative evaluation of the usefulness of our tool. Specifically, our study was guided by the following research questions (RQs):

RQ1. How effective is the proposed approach for analysis and propagation of feature revisions in preprocessor-based SPLs in VCSs? Our goal was to check the correct behaviour and runtime performance of our approach for analysis and propagation of feature revisions. We checked whether our approach correctly propagates the implementation of feature revisions between two releases. We also checked whether our approach introduces new features and removes features, not in common between two SPL releases.

RQ2. What is the perception of developers regarding the usefulness of our tool for the analysis and propagation of feature revisions? Our approach and tool support was defined based on existing practical problems. However, to better understand the impact of our tool support, we wanted to understand how developers perceive the usefulness of our tool for the analysis and propagation of feature revisions annotated in preprocessor-based SPLs.

A. Quantitative Analyses (RQ1)

Establishing the ground truth. Figure 5 illustrates how we evaluated our approach for analysis and propagation of 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 infeasible to compute, we determined the same number of combinations for all systems and limited the total runtime to no longer than 48 hours for computing all the propagations.

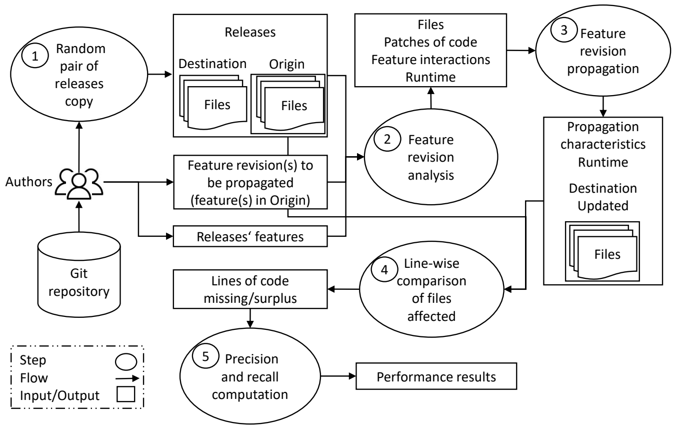

Random pair of releases copy
Releases
Destination
Origin
Files
Files
Files
Patches of code
Feature interactions
Runtime
Feature revision propagation
Propagation characteristics
Runtime
Destination Updated
Files
Feature revision analysis
Authors
Feature revision(s) to be propagated (feature(s) in Origin)
Releases' features
Git repository
Lines of code missing/surplus
Line-wise comparison of files affected
Precision and recall computation
Performance results
Step
Flow
Input/Output

Figure 5: Evaluation methodology of the quantitative analyses of our approach.

For each destination and origin, our approach copies their snapshot files (Step 1). As input for Step 2, the set of features of each release (mined by our approach in advance) and the feature revision(s) to be propagated are also required. The feature(s) revision(s) to be propagated are features with different implementations in the origin and destination releases. Some feature revisions can either exist in both releases or not. For example, a feature revision existing in the origin release but not in the destination release is a feature revision to be introduced in the destination release. Lastly, a feature revision existing in the destination release that does not exist in the origin release is a feature revision to be removed in the destination release. We performed propagations of only a single feature revision at a time, as well as multiple feature revisions.

For each feature revision to be propagated, there is an output from Step 2, which contains *propagation characteristics*, i.e., files and patches of code, and feature interactions regarding the differences of a feature revision between two releases (detail and results are available in our online appendix [26]). The files and lines of code affected per feature revision to be propagated are now used as input for Step 3. In this step, each feature revision is incrementally propagated, and the output is the destination files of code updated with the feature revisions propagated. Therefore, after propagating feature revisions from origin to destination, we get updated files of code of the destination release. If our approach performed the analysis and propagation of feature revisions correctly, the updated files of code in the destination release must be equal to or very similar to the content of files of code of the origin release. We also get the runtime computation of Step 3 as well as further propagation characteristics for each pair of releases.

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 are not supposed to be in the destination file using the corresponding origin file as a 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 a baseline. Then, the false negatives (lines missing) and the false positives (surplus lines) are used as input for the last step (Step 5), which computes the precision and recall used to answer RQ1.

Table I: 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

Subject Systems. The quantitative analyses rely on four open-source preprocessor-based SPLs managed in the Git VCS (Table I). We reduced bias by choosing different application domains. Furthermore, each system has a considerable history of development and use in research [1], [8], [19], [27], [28], [29]. 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 a dataset of their evolution available [8]. Based on the dataset available, we thus know which features exist in one release that does not exist in any other arbitrary release. Therefore, this information helped us to check our approach's correct behaviour for analysis and propagation of the implementation of feature revisions.

Correct behaviour. For checking the correct behaviour of our approach we computed precision, recall, and F1-score [30].

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

B. *Qualitative Analyses (RQ2)*

To evaluate the usefulness of our tool support for analysis and propagation of feature revisions in preprocessor-based SPLs, we first contacted open-source developers who contributed to or maintained source code of the preprocessor-based SPLs used in our study. The goal of this first contact by email was to get in touch with them and to learn more about their experience. Developers with industry experience were then invited for follow-up interviews of 30 minutes. Interviews were conducted to find out whether our tool support would help with the analysis and propagation of feature revisions. We also wanted to obtain insights on their current development challenges and promising areas for future investigation.

Participant selection. We emailed developers if they performed at least 10 commits to one of the investigated preprocessor-based SPLs (Marlin, SQLite, Bison, Irssi, and LibSSH), and we could find a public email address or website for contacting them. In the end, we sent emails to 103 developers of these systems. 13 developers replied to us out of which 8 had sufficient experience to be invited to the interviews. Five developers from Marlin, SQLite, and Bison were available and accepted our invitation. During the fifth interview, we already had reached saturation [31], [32] in the sense that we only got marginal additional insights. The developers' profile and experience is available in our online appendix [26].

Interview. We conducted each interview in the same way, following a semi-structured fashion divided into four phases, inspired by Zhou et al. [33]. The interviews with developers were conducted after we executed a pilot interview to detect and resolve misunderstandings and structuring problems.

- *Opening and introduction:* We started each interview by asking the participants whether they consent to video/audio recording based on a consent form sent in advance to them. Right after we started recording and asked the interviewees to watch an introductory video[3] briefly explaining our research topic and the general purpose of the interview.
- *Understanding how features are propagated between releases by developers:* We first asked participants to describe how they would perform analysis and propagation of a feature between two releases of preprocessor-based SPLs. Subsequently, we pursued two questions: Q1. What is your step-by-step process to analyze and propagate a feature between two releases (do you use any tool, or is it a fully manual process)? Q2. How do you ensure that the propagation is correct?
- *Understanding how helpful is our tool support and what can be improved:* Before asking the developers about the usefulness of our tool, they watched a video[4] with a demo of our tool, showing an example illustrating its capabilities and demonstrating what developers can automate. Afterwards, we asked them four questions: Q1. Would the tool help in maintenance and reuse tasks relying on the analysis and propagation of features? Q2. Would the tool help even if you would have to compile and run tests to ensure that the propagation is correct? Q3. Can you remember a development situation in which the tool could have helped? Q4. Would you use the tool when propagation is needed?
- *Closing:* We concluded the interview by asking two last questions: Q1. What do you suggest for improving the tool? Q2. Are there other problems you regard as more relevant?

Analysis. After the interview, we transcribed all the video and audio recordings. We employed Grounded Theory (GT) procedures [32] to conduct an in-depth qualitative analysis of the data. Thus, first, we performed open coding to associate codes of developers' utterances with categories. Afterward, we related the codes through axial coding, i.e., the codes were merged and grouped into more abstract categories. Then, we proceed with selective coding where we select one central aspect of data as a core category or final category, which is the GT analysis to answer RQ2.

[3]https://youtu.be/TX9HOtir4Ws
[4]https://youtu.be/KmMzRM8Ft3g

V. Results and Discussion

This section presents the results for our RQs. At the end of each subsection, we present the answer for each RQ.

A. Approach Efficiency (RQ1)

This section presents the quantitative results, i.e., the approach performance in terms of *correct behavior* and *runtime*. Our dataset [22] contains propagations of 3,134 features and from the total number of features propagated, 87.7% were revised features, 11.3% were introduced features, and 1.0% were removed features. 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. Our online appendix [26] presents more detail and the *Propagation Characteristics*, as well as an analysis of the complexity of the propagations performed.

Correct behaviour. Regarding the correct behaviour, our approach could propagate feature revisions for each target system with 99% precision, recall, and F1-score, on average, considering 200 random pairs of releases. Table II shows the total number of lines of surplus code and missing code concerning 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% of precision, recall, and F1-score 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 [34]. Thus, if a presence condition of a conditional block is unsatisfiable and no solution is found by the CSP solver, it is dead code and cannot be linked to any feature revision.

Finding dead code is an important issue because it allows the detection of defects and bugs in some parts of a system implementation without considering the implementation at large [35]. Dead codes are commonly encountered in preprocessor-based SPLs, for example, the Linux kernel [36]. There exist many dead variable analyses for identifying dead conditional blocks of code [35], [36], [37], [38]. 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 the release `GNU_1_27`, commit hash e7ae9cf[5] of Bison system,

Table II: Total number of lines of surplus code and missing code 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

[5]https://github.com/bison/e7ae9cf

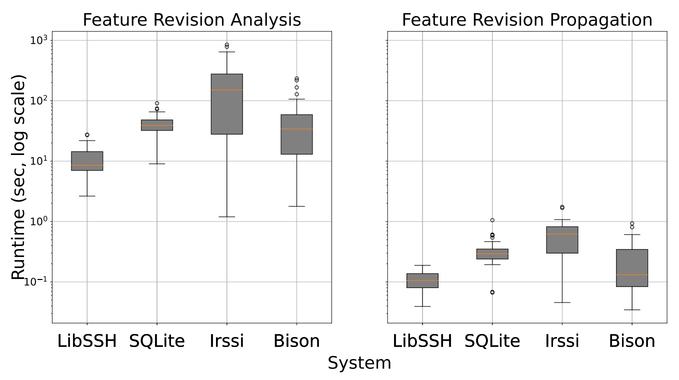


Figure 6: Average Runtime performance in seconds for analysis and propagation of a feature revision for each random pair of releases.

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.

Runtime. The runtime performance for each feature revision analysis and propagation per system is presented in Figure 6. The system that took the most time on average for analysis and propagation per feature revision was Irssi with 850.5 seconds in total, with 848.75 seconds for the analysis and 1.75 seconds for propagation. The runtime of analysis and propagation of a feature revision varies with the number of feature interactions, patches, files, and lines of code that differ between two releases, which explains the outliers we observed in all systems. All these metrics are available [26].

Answering RQ1. *How effective is the proposed approach for analysis and propagation of 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, which is a problem in the SPLs analyzed, rather than in our approach. Further, our approach retrieved existing dead codes that may help to find bugs and refactor SPLs. The runtime performance was, on average, considering the four target systems ≈63 and 0.2 seconds for the analysis and propagation of a feature revision, respectively. Most of the feature revisions propagated were interacting with more than two feature revisions, and involved changes in multiple files, and conditional blocks of code. We thus estimate that manually performing analysis and propagation of feature revisions would take considerable effort and time, while our approach can automate these tasks.

B. Tool Support Usefulness (RQ2)

After applying GT (online appendix [26]), we derived conclusions and lessons learned, which were fundamental to answering RQ2:

Need to propagate features between releases. This is not a scenario for all systems, according to one of the developers. On the other hand, for example, another developer mentioned that *"Maintaining different releases is necessary due to customers that have, for example, a particular contract or hardware, and make customers locked in a particular release."* For instance, the original 1.1.x release (from Marlin) is still maintained as it is used by the initial targets, i.e., Arduino boards. Propagation is also necessary between releases because propagation of features is performed by different developers than the original developers: *"Sometimes people ask us to propagate features and we are like 'no we do not have time to do that but you are free to do it yourself' [...] sometimes people will actually try to do that on their own [...]. To make it would be nice if things were easier for them so they could maintain some version that they are using because some companies are stuck on a particular version because of policy."*

Challenges in propagation of features in preprocessor-based SPLs in VCSs. Developers agreed that propagating a feature is difficult and hard to automate. This is because there is no mapping between features and segments of code, i.e., VCSs do not keep track of #ifdefs. Further, extracting feature dependencies in preprocessor directives is difficult. Developers mentioned right in the beginning of the interview, before even knowing about our tool, that: *"The mapping between features and the segments of code will make propagation easier."* and: *"Tooling in C/C++ is poor. New tools are most welcome."* Developers confirmed existing studies [8], [17], [18] that the use of preprocessor directives is painful: *"I also know how painful it is to deal with the CPP and I am sorry for you dealing with the CPP. It is really a mess."* Developers always try to ensure that the propagation is performed correctly by running tests. However, depending on the system this is not enough, for instance, for embedded systems: *"Well, in the case of Marlin, for the propagation of BLTouch feature, it would be very nice to have access to the hardware itself so that I can test the code running."*

Feedback on our tool support. Developers mentioned that the tool meets its purpose and they would certainly try it when propagation is needed: *"It is great. I am amazed, I actually wrote a tool that does some similar thing with #ifdefs and so I am impressed because I know what is required to make that work. That is impressive."* Another positive feedback about our tool support is related to its interface: *"I think it is really helpful since it has a clear interface."* Furthermore, our tool support can reduce the possibility of leaving necessary code out and making mistakes: *"That [the tool] would reduce the need to do manual double checking of all the source code changes, you can just run your tool to look at the overall diff, make sure the diff looks reasonable, and run the tests.*

It is going to speed up things dramatically." Still, regarding the visualization of differences of a feature between releases, another developer said: *"With your tool, it seems like you see the bigger picture and then you can 'grab and go' into the details."* Also according to one of the developers, our tool support can be applicable for code reviews: *"That is really nice. That would help I think especially in code reviews, so we are reviewing a piece of code, you are missing a lot of the information and what you are looking for is the bigger picture because you do not really want to go into like this much detail, right? You do not want to run the code base in your head basically to determine whether it works. I think that would help certainly."* Developers reported that our tool can automate part of a laborious task of reusing parts of a feature implementation between two releases and thus save hours of work: *"This would save several hours of work for this [analysis and propagation]. I mean, this automation is great because it will be a big time saver."*

Limitations and improvements for our tool support. Our tool supports developers to a great extent, but developers still must double-check the deltas and decide which patches of code to propagate. This was acknowledged by one interviewee: *"It is your part to decide which lines to be removed and what kind of code should we move to the old branch [release]."* Another developer said: *"A lot of human knowledge is required. I like the idea of potentially automating it, but I think it is going to be very difficult automating it 100%."* Therefore, even when using our tool to automatically show and merge the differences of a feature, there are still manual adaptations of the changes that may need to be performed by developers. Suggestions are proposed by developers to extend our tool support with semantic analysis and a view of a dependency graph to know what and how features are interrelated. For example, we can enhance our approach with a directed dependency graph based on the semantically annotated abstract syntax tree of the source code [39], and a graphical interface with a graph view for navigation on the graph nodes [40]. Regarding the tool accessibility, developers suggested integrating it into GitHub and also making it available for the command line.

Answering RQ2. *What is the perception of developers regarding the usefulness of our tool for the analysis and propagation of feature revisions?* All interviewed developers agreed that the tool would be useful and save hours of manual work. Further, developers reported that the tool has a clear interface that helps to see the big picture of all the differences between two releases associated with a respective feature, which would also help in code reviews. In summary, the tool was evaluated by the developers as a novel and a much-needed support, which provides automation and is a big time saver. All the developers affirmed that they would use the tool support whenever they would need to propagate features.

VI. 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 to make sure that at least one feature revision was propagated, and that there was no pair of releases where the origin was released right after the destination. Further, we are proposing a novel approach to propagate feature revisions and there is currently no other approach for comparison. To alleviate this we conducted a study with developers. After propagating feature revisions to a release, the SPL might contain bugs and tests should be run. 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 [21], data [22], and supplementary material [26] are made available, and we encourage researchers to replicate our study and improve our approach for analysis and propagation of feature revisions.

External Validity. The selected systems can be a source of bias. However, we do not attempt to generalize the scenarios but 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 closely related work on feature evolution in space and time [8], containing information useful for a preliminary analysis of how much features have evolved in VCSs. Nonetheless, the subject systems encompass diverse domains and sizes and have been used also in other studies [1], [10], [11], [19], [27], [28], [29]. Further, the systems we used are C/C++ preprocessor-based SPLs available in VCSs. These SPLs have a considerable history of commits involving evolution in space and time.

Construct Validity. A potential threat pertains to the metrics used to assess the correct behaviour of our approach. Precision and recall [30] have been widely used to assess whether the information retrieved from existing approaches to locate source code to feature(s) (revisions) is correct [11], [41], [42]. In our study, the calculation of metrics 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 false positives and false negatives lines of code surplus or missing, in relation to the correct sequences of lines of code matching between a pair of releases. Regarding the qualitative analyses, possible distortions when interpreting the data using GT procedures pose a threat. To mitigate this, the GT coding and categories were discussed by all the authors until reaching a consensus, and we provide all the interview data in our online appendix [26]. Regarding reaching data saturation to stop further data collection, i.e., interviews, we followed factors presented in [31], [32]. The reliability of the interview data can be another threat to construct validity, which we mitigate with a diversity of participants' experience, degree, and geographical location (information available in our online appendix [26]). Further, we were able to interview two core developers of two of the three systems analyzed, where one of the developers did more than 70% of the total number of commits, and the other developer has been contributing for more than eleven years, with significant experience in propagating features.

VII. Related Work

Some tools were designed to support easier comprehension of annotation-based systems by hiding annotations [43] or using colors for visualization of features in the source code, such as C-CLR [44], FeatureCommander [17], variation editor [45], PEoPL IDE [46] and CIDE [47]. Although these tools can be used for better comprehension of annotated SPLs, they do not analyze changes in commits and relate them to the feature evolution in space and time. Dintzner et al. [48] and Passos et al. [49] presented tools for mining information of feature-evolution in a variability model, build system, and source code. A Web tool called FeatureCloud [50] proposed to mine and visualize changes in #ifdef blocks of systems in Git repositories. Another related work is an empirical analysis of feature-evolution presented by Michelon et al. [8]: the main difference of our approach is that we analyze and propagate feature revisions between releases. Michelon et al. [8] focus on tracking the history of feature changes. Our approach intents to give support to evolution in space and time of preprocessor-based SPLs at the feature level.

Regarding propagation, Montalvillo & Díaz [51] presented a browser extension for GitHub synchronizing Java artifacts versions in different products forked from a core repository of a Feature-oriented SPL. In the context of propagation/transplantation of patches of code [52] and backporting [53] there exist approaches for automating program repair. However, these approaches automatically transfer patches between different versions of preprocessor-based systems without taking into account features annotated in the source code.

Gupta et al. [54] 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 impact analysis techniques based on dependency analysis [55], analyzing the change in semantic dependencies between program entities. For instance, Zhang et al. [56] presented a change analysis for systems developed with AspectJ. Also, for change impact analysis, Chianti [57] is a tool implemented in the Eclipse environment, which uses test execution to infer modification of the behavior of a Java system. Yet, compared to previous work [58] 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 for C/C++ preprocessor-based SPLs.

Variation control systems [12] 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. [11] 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 they are not mature enough with limited support for collaborative and distributed development [12], [59].

There is related work combining the pros of clone-and-own and SPL as well as synchronizing cloned variants to support a more agile development with cloning and quickly prototyping new variants [60], [61]. Therefore, the goal is to reduce the gap between clone-and-own and SPL, which differs from our work, which offers tool support for the analysis and propagation of feature revisions between releases of preprocessor-based SPLs. The variant synchronization and variation control systems can be promising solutions for the main problem of evolution in space and time. However, we focus on solving the current challenges of preprocessor-based SPLs in VCSs, which are currently more popular and well-known by developers.

VIII. Conclusion and Future Work

We proposed a novel approach implemented in a non-intrusive tool to automatically and efficiently reuse different implementations of features in preprocessor-based SPLs from one release to another arbitrary one. Our approach provides (i) feature revision analysis between an origin and a destination release, retrieving, for example, which files and lines were modified/introduced/deleted and which feature revisions were introduced/deleted/revised and affected by feature 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 lines of code differing between origin and destination releases that affect a specific feature revision to be propagated. We evaluated our approach performance, showing that it reaches 99% precision and recall, taking on average for the four target systems $\approx$63 and 0.2 seconds for performing analysis and propagation of a feature revision, respectively.

We also evaluated our tool support by interviewing developers of preprocessor-based SPLs, who confirmed that the analysis and propagation of features are challenging tasks if performed manually. Overall, the developers' feedback confirmed the usefulness of our tool and gave us future directions toward extending our approach. For example, these included a semantic impact analysis with a graph view of the source code mapped to features, as well as making our approach available as both a browser extension for Git VCSs and a command line tool.

Acknowledgment

The research reported in this paper has been funded by BMK, BMDW, and the State of Upper Austria in the frame of SCCH, part of the COMET Programme managed by FFG; the LIT Secure and Correct System Lab funded by the State of Upper Austria; and the Austrian Science Fund (FWF), grant no. P31989.

References

[1] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze, "An analysis of the variability in forty preprocessor-based software product lines," in *32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 105–114.

[2] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand, and S. Soares, "On the impact of feature dependencies when maintaining preprocessor-based software product lines," in *10th ACM International Conference on Generative Programming and Component Engineering*, ser. GPCE '11. New York, NY, USA: ACM, 2011, p. 23–32.

[3] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, "PrefFinder: Getting the right preference in configurable software systems," in *29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, p. 151–162.

[4] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature?: a qualitative study of features in industrial software product lines," in *19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, D. C. Schmidt, Ed. New York, NY, USA: ACM, 2015, pp. 16–25.

[5] J. Krüger and T. Berger, "An empirical analysis of the costs of clone- and platform-oriented software reuse," in *28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 432–444.

[6] R. Queiroz, L. Passos, T. M. Valente, C. Hunsen, S. Apel, and K. Czarnecki, "The shape of feature code: an analysis of twenty C-preprocessor-based systems," *Software and Systems Modeling (SoSyM)*, vol. 16, pp. 77–96, 2017.

[7] T. Berger, M. Chechik, T. Kehrer, and M. Wimmer, "Software evolution in time and space: Unifying version and variability management (dagstuhl seminar 19191)," *Dagstuhl Reports*, vol. 9, no. 5, pp. 1–30, 2019.

[8] G. K. Michelon, W. K. G. Assunção, D. Obermann, L. Linsbauer, P. Grünbacher, and A. 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*, ser. GPCE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1–10.

[9] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed, "Evolving software system families in space and time with feature revisions," *Empir. Softw. Eng.*, vol. 27, no. 5, p. 54, 2022.

[10] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, and A. Egyed, "Mining feature revisions in highly-configurable software systems," in *24th ACM International Systems and Software Product Line Conference - Volume B*, ser. SPLC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 74–78.

[11] G. K. Michelon, D. Obermann, L. Linsbauer, W. K. G. Assunção, P. Grünbacher, and A. Egyed, "Locating feature revisions in software systems evolving in space and time," in *24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, ser. SPLC '20. New York, NY, USA: Association for Computing Machinery, 2020.

[12] L. Linsbauer, F. Schwägerl, T. Berger, and P. Grünbacher, "Concepts of variation control systems," *J. Syst. Softw.*, vol. 171, p. 110796, 2021.

[13] G. K. Michelon, D. Obermann, W. K. G. Assunção, L. Linsbauer, P. Grünbacher, and A. Egyed, "Managing systems evolving in space and time: Four challenges for maintenance, evolution and composition of variants," in *25th ACM International Systems and Software Product Line Conference - Volume A*, ser. SPLC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 75–80.

[14] D. Hinterreiter, M. Nieke, L. Linsbauer, C. Seidl, H. Prähofer, and P. Grünbacher, "Harmonized temporal feature modeling to uniformly perform, track, analyze, and replay software product line evolution," in *18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 115–128.

[15] S. Ananieva, S. Greiner, J. Krueger, L. Linsbauer, S. Gruener, T. Kehrer, T. Kuehn, C. Seidl, and R. Reussner, "Unified operations for variability in space and time," in *16th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '22. New York, NY, USA: Association for Computing Machinery, 2022.

[16] W. D. F. Mendonça, S. R. Vergilio, G. K. Michelon, A. Egyed, and W. K. G. Assunção, "Test2feature: Feature-based test traceability tool for highly configurable software," in *26th ACM International Systems and Software Product Line Conference - Volume B*, ser. SPLC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 62–65.

[17] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Softw. Engg.*, vol. 18, no. 4, p. 699–745, Aug. 2013.

[18] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, ser. LIPIcs, J. T. Boyland, Ed., vol. 37. Prague, Czech Republic: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[19] 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*, vol. 44, no. 5, pp. 453–469, May 2018.

[20] P. Bunyakiati and C. Phipathananunth, "Cherry-picking of code commits in long-running, multi-release software," in *2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 994–998.

[21] G. Michelon. (2022) Tool for analysis and propagation of feature revisions in preprocessor-based SPLs. Johannes Kepler University Linz. [Online]. Available: https://github.com/GabrielaMichelon/git-ecco/tree/changepropagation

[22] G. K. Michelon, W. K. G. Assunção, P. Grünbacher, and A. Egyed. (2022) Dataset. Johannes Kepler University Linz. [Online]. Available: https://figshare.com/s/f508d59fd07632ab3f39

[23] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Solver Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. [Online]. Available: http://www.choco-solver.org

[24] D. Benavides, S. Segura, P. T. Martín-Arroyo, and A. R. 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)*, ser. Lecture Notes in Computer Science, vol. 4143. Berlin, Heidelberg: Springer, 2005, pp. 399–408.

[25] K. Ludwig, J. Krüger, and T. Leich, "Covert and phantom features in annotations: Do they impact variability analysis?" in *23rd International Systems and Software Product Line Conference - Volume A*, ser. SPLC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 218–230. [Online]. Available: https://doi.org/10.1145/3336294.3336296

[26] "Online appendix," https://sites.google.com/view/analysis-and-propagation.

[27] A. Gargantini, J. Petke, M. Radavelli, and P. Vavassori, "Validation of constraints among configuration parameters using search-based combinatorial interaction testing," in *Search Based Software Engineering*, F. Sarro and K. Deb, Eds. Cham: Springer International Publishing, 2016, pp. 49–63.

[28] 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*, ser. ICSME '19. San Francisco, CA, USA: IEEE Press, Sep. 2019, pp. 470–480.

[29] T. Vale and E. S. Almeida, "Experimenting with information retrieval methods in the recovery of feature-code SPL traces," *Empirical Software Engineering*, vol. 24, no. 3, p. 1328–1368, Jun. 2019.

[30] K. M. Ting, *Precision and Recall*. Boston, MA: Springer US, 2010.

[31] K. Aldiabat and C.-L. Le Navenec, "Data saturation: The mysterious step in grounded theory methodology," *Qualitative Report*, vol. 23, pp. 245–261, 01 2018.

[32] A. Strauss, J. Corbin, and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998.

[33] S. Zhou, S. Stănciulescu, O. Leßenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, May 2018, p. 105–116.

[34] J. Kim, D. Batory, and D. Dig, "Refactoring java software product lines," in *21st International Systems and Software Product Line Conference - Volume A*, ser. SPLC '17. New York, NY, USA: ACM, 2017, p. 59–68.

[35] C. Kröher, M. Flöter, L. Gerling, and K. Schmid, "Incremental software product line verification - A performance analysis with dead variable code," *Empir. Softw. Eng.*, vol. 27, no. 3, p. 68, 2022.

[36] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem," in *6th Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, p. 47–60.

[37] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake, "A classification and survey of analysis strategies for software product lines," *ACM Comput. Surv.*, vol. 47, no. 1, jun 2014.

[38] S. Nadi and R. C. Holt, "Mining kbuild to detect variability anomalies in linux," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, T. Mens, A. Cleve, and R. Ferenc, Eds. USA: IEEE Computer Society, 2012, pp. 107–116.

[39] M. Hamza, R. J. Walker, and M. Elaasar, "Ciahelper: Towards change impact analysis in delta-oriented software product lines," in *22nd International Systems and Software Product Line Conference - Volume 1*, ser. SPLC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 31–42.

[40] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 989–1000.

[41] D. Cruz, E. Figueiredo, and J. 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*, ser. VAMOS 2019. New York, USA: ACM, 2019, pp. 16:1–16:10.

[42] J. Martinez, T. Ziadi, M. Papadakis, T. F. Bissyandé, J. Klein, and Y. le Traon, "Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants," *Information and Software Technology*, vol. 104, pp. 46 – 59, 2018.

[43] D. Atkins, T. Ball, T. 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*, vol. 28, no. 7, pp. 625–637, 2002.

[44] N. Singh, C. Gibbs, and Y. Coady, "C-clr: A tool for navigating highly configurable system software," in *6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ser. ACP4IS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 9–es.

[45] D. Le, E. Walkingshaw, and M. Erwig, "#ifdef confirmed harmful: Promoting understandable software variation," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011*, G. Costagliola, A. J. Ko, A. Cypher, J. Nichols, C. Scaffidi, C. Kelleher, and B. A. Myers, Eds. San Francisco, CA, USA: IEEE, 2011, pp. 143–150.

[46] M. Mukelabai, B. Behringer, M. Fey, J. Palz, J. Krüger, and T. Berger, "Multi-view editing of software product lines with peopl," in *40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 81–84.

[47] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code," in *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops)*, S. Thiel and K. Pohl, Eds. Francisco, Clifornia, USA: Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 303–312.

[48] N. Dintzner, A. van Deursen, and M. Pinzger, "Fever: Extracting feature-oriented changes from commits," in *13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 85–96.

[49] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo, "Coevolution of variability models and related software artifacts," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1744–1793, May 2015.

[50] O. Díaz, R. Medeiros, and L. Montalvillo, "Change analysis of #if-def blocks with ¡i¿featurecloud¡/i¿," in *23rd International Systems and Software Product Line Conference, SPLC 2019, Volume B, Paris, France, September 9-13, 2019*, ser. SPLC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 17–20.

[51] L. Montalvillo and O. Díaz, "Tuning github for spl development: Branching models & repository operations for product engineers," in *19th International Conference on Software Product Line*, ser. SPLC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 111–120.

[52] R. S. Shariffdeen, S. H. Tan, M. Gao, and A. Roychoudhury, "Automated patch transplantation," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 1, dec 2021.

[53] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," in *30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. New York, NY, USA: ACM, 2021, pp. 633–645.

[54] C. Gupta, M. Srivastav, and V. Gupta, "Software change impact analysis: An approach to differentiate type of change to minimise regression test selection," *Int. J. Comput. Appl. Technol.*, vol. 51, no. 4, p. 366–375, Jul. 2015.

[55] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," in *25th International Conference on Software Engineering*, ser. ICSE '03. USA: IEEE Computer Society, 2003, p. 308–318.

[56] S. Zhang, Z. Gu, Y. Lin, and J. Zhao, "Change impact analysis for aspectj programs," in *2008 IEEE International Conference on Software Maintenance*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 87–96.

[57] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," *SIGPLAN Not.*, vol. 39, no. 10, p. 432–448, Oct. 2004.

[58] C. Kröher, L. Gerling, and K. Schmid, "Identifying the intensity of variability changes in software product line evolution," in *Proceeedings of the 22nd International Systems and Software Product Line Conference - Volume 1, SPLC 2018, Gothenburg, Sweden, September 10-14, 2018*, T. Berger, P. Borba, G. Botterweck, T. Männistö, D. Benavides, S. Nadi, T. Kehrer, R. Rabiser, C. Elsner, and M. Mukelabai, Eds. New York, NY, USA: ACM, 2018, pp. 54–64.

[59] S. Stănciulescu, T. Berger, E. Walkingshaw, and A. 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*. San Francisco, CA, USA: IEEE Computer Society, 2016, pp. 323–333.

[60] W. Mahmood, D. Strüber, T. Berger, R. Lämmel, and M. Mukelabai, "Seamless variability management with the virtual platform," in *43rd International Conference on Software Engineering*, ser. ICSE '21. IEEE Press, 2021, p. 1658–1670.

[61] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer, "Synchronizing software variants with variantsync," in *20th International Systems and Software Product Line Conference*, ser. SPLC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 329–332.