Locating Feature Revisions in Software Systems Evolving in Space and Time

Gabriela Karoline Michelon[1,2], David Obermann[1], Lukas Linsbauer[3],
Wesley Klewerton G. Assunção[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]Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig, Germany
[4]COTSI - Federal University of Technology of Paraná, PPGComp - Western Paraná State University, Brazil

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.

CCS CONCEPTS

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

KEYWORDS

feature location, feature revisions, variants, repository mining

ACM Reference Format:
Gabriela Karoline Michelon[1,2], David Obermann[1], Lukas Linsbauer[3], Wesley Klewerton G. Assunção[4], Paul Grünbacher[1], Alexander Egyed[1]. 2020. Locating Feature Revisions in Software Systems Evolving in Space and Time. In *24th ACM International Systems and Software Product Line Conference (SPLC '20), October 19–23, 2020, MONTREAL, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3382025.3414954

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

ACM ISBN 978-1-4503-7569-6/20/10…$15.00
https://doi.org/10.1145/3382025.3414954

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 [17]. 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 [32]. 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 [30]. HCSSs aim to satisfy the requirements of different customers and environmental restrictions such as different hardware devices [36]. 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 [4, 20, 24].

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 [2]. Unfortunately, this assumption rarely holds in practice as variants diverge both in space (different feature combinations) and time (different revisions of features) [16]. 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 [2]. However, while existing feature location techniques primarily address variability in space [8, 33], they are not as useful when features evolved over time, leading to divergent implementations [4]. 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 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 [4]. 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.

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

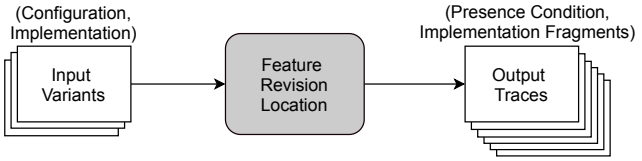


Figure 1: Feature Revision Location Overview.

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 [20, 24], 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 $\mathcal{A}_1 \wedge \neg C$ 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 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 $\mathcal{A}_1 \wedge \neg C$ it is still included in the condition.

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	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	$\mathcal{A}_1 \vee (\mathcal{A}_1 \wedge \neg C) \vee (\mathcal{A}_1 \wedge \mathcal{B}_{1\vee2})$	$\{a_1, a_2\}$
t_2	$\mathcal{B}_1 \vee (\mathcal{B}_1 \wedge \neg C) \vee (\mathcal{A}_1 \wedge \mathcal{B}_1)$	$\{a_3\}$
t_3	$\mathcal{B}_2$	$\{a_4\}$
t_4	$C_1 \vee (\neg\mathcal{A}_1 \wedge C_1) \vee (\mathcal{B}_2 \wedge C_1)$	$\{a_5, a_6, a_7\}$

3.2 Trace Computation

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

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 [25], feature location [22], or the analysis of variable systems [1, 11]. 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, $clauses(\{\mathcal{A}_1, \mathcal{B}_1, \neg C\}) = \{\mathcal{A}_1, \mathcal{B}_1, \mathcal{A}_1 \wedge \mathcal{B}_1, \mathcal{A}_1 \wedge \neg C, \mathcal{B}_1 \wedge \neg C, \mathcal{A}_1 \wedge \mathcal{B}_1 \wedge \neg C\}$. 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 [28, 31]. 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.

$$c \in C \Leftrightarrow (\forall v \in V : c \in clauses(v.F) \implies a \in v.A) \wedge (\exists v \in V : c \in clauses(v.F)) \quad (1)$$

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 [28]. 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).

$$c' \in C \Leftrightarrow (\forall v \in V : c \in clauses(v.F) \iff a \in v.A) \wedge (\exists v \in V : c \in clauses(v.F)) \quad (2)$$

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 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 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 1 (Lines 13-17).

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
```

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 [9, 11]. 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 [25], 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 [7] to perform multi-sequence alignment for comparing more than two variants [9, 23], 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 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 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 2) is satisfied.

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

		$\mathcal{A}$	$\mathcal{B}$		C	$\mathcal{A} \wedge \mathcal{B}$		$\mathcal{B} \wedge C$
		2	3		1	2		1
		$\mathcal{A}_1$	$\mathcal{B}_1$	$\mathcal{B}_2$	C_1	$\mathcal{A}_1 \wedge \mathcal{B}_1$	$\mathcal{A}_1 \wedge \mathcal{B}_2$	$\mathcal{B}_2 \wedge C_1$
		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

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

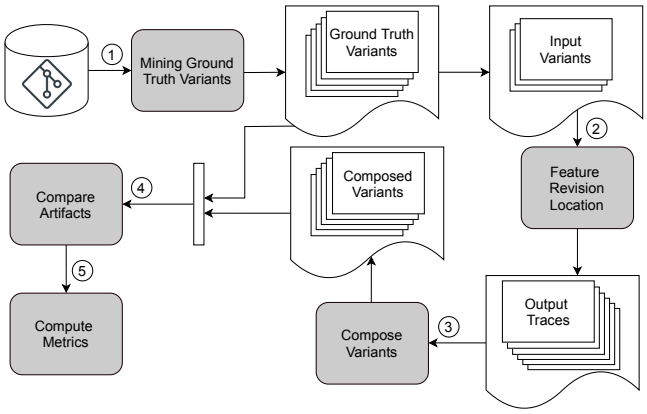


Figure 2: Methodology for evaluating our feature revision location technique.

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

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 [19] 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 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 [13, 14, 17–19, 27, 35]. 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 [29]. 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 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`

[7] https://github.com/java-diff-utils/java-diff-utils
[8] https://github.com/MarlinFirmware/Marlin
[9] https://gitlab.com/libssh/libssh-mirror
[10] https://github.com/sqlite/sqlite

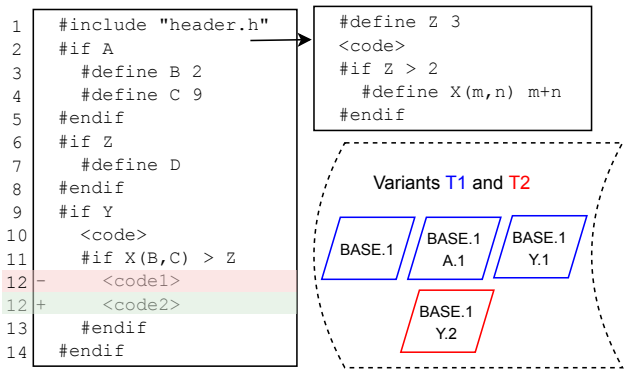


Figure 3: Mining changes over time to generate ground truth variants to evaluate our feature revision location technique.

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 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.

[11] gitlab.com/libssh/libssh-mirror/-/commit/c65f56aefa50a2e2a78a0e45564526ecc921d74f
[12] gitlab.com/libssh/libssh-mirror/-/commit/d40f16d48ec1ed9670c20ffaad1005c59a689484
[13] gitlab.com/libssh/libssh-mirror/-/commit/55846a4c7b09af2d105c7f7dfd0a43aab2f6e5a5

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 without any implications.

In case this does not obtain any potentially changed features, meaning that there are no positive external features in the closest

[14] https://github.com/chocoteam/choco-solver

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 [34]. They are commonly used to evaluate feature location techniques [6, 26, 28]. 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.

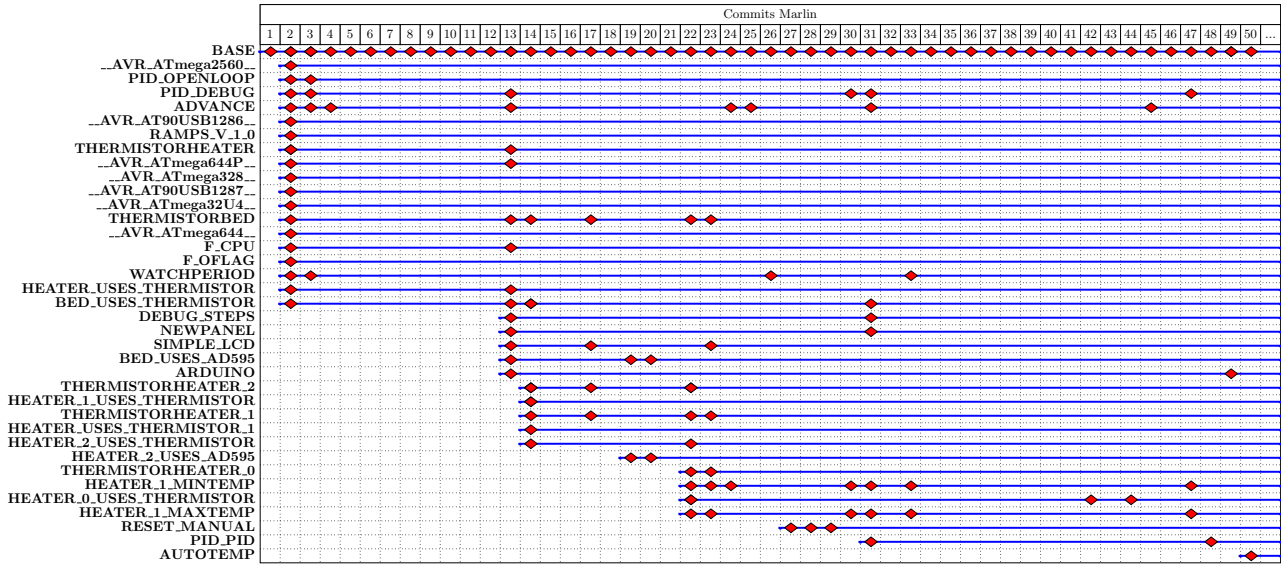
5 RESULTS AND DISCUSSION

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.

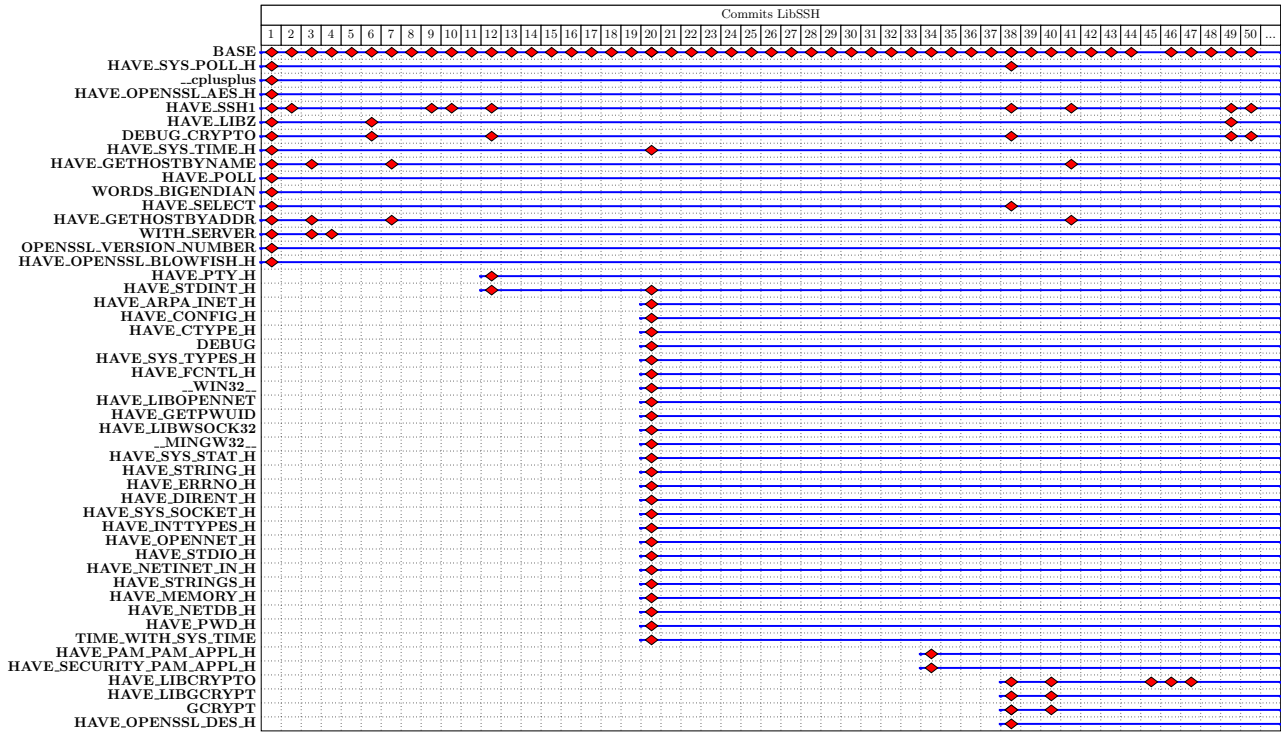
Analysis of feature evolution in space and time in subject systems. We present in Figure 4 the evolution in space and time, i.e., the features introduced/changed in the first 50 Git commits of three systems. The blue line in a row represents the time between the inclusion of a feature in the Git repository 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 4(a). 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 4(b) 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 4(c), 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 4(a) 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 [15] 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.

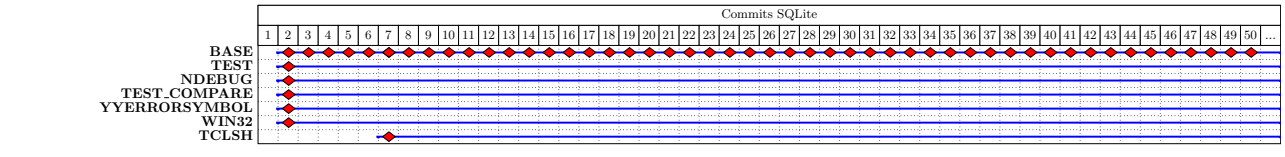
The false negative lines in Marlin were about 3,138 from a total of 1,460,533 relevant lines over 144 compared variants (min: 0, max: 163, mean: 21.79 per variant). From false negative lines, 774 are comment lines. Those false negative lines were caused by incorrect traces since Git commit #31, which 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



(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.

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

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

since Git commit #10. In SQLite, three lines were false negatives on composed variants corresponding to the Git 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`.

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

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

Listing 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 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.

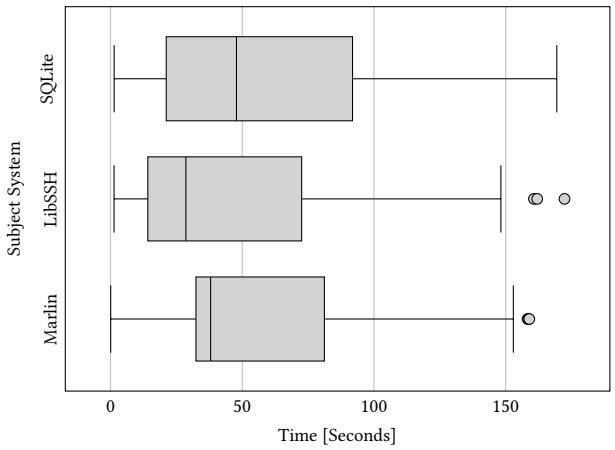


Figure 5: Runtime of feature revision location per variant.

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 [9, 10, 12, 21].

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. [5] 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 [4]. 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 [20, 24].

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 [3]. However, the feature revision concept is still untreated among feature location techniques in the literature [3, 6, 8, 28, 31, 33]. As suggested by Hinterreiter et al. [16], 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.

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 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.

REFERENCES

[1] Florian Angerer, Andreas Grimmer, Herbert Prähofer, and Paul Grünbacher. 2019. Change Impact Analysis for Maintenance and Evolution of Variable Software Systems. *Automated Software Engineering* 26 (June 2019), 417–461. Issue 2. https://doi.org/10.1007/s10515-019-00253-7

[2] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empirical Software Engineering* 22, 6 (01 Dec 2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[3] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. 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* (Florence, Italy) *(SPLC 2014)*. ACM, New York, USA, 52–59. https://doi.org/10.1145/2647908.2655967

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

[5] Reidar Conradi and Bernhard Westfechtel. 1998. Version Models for Software Configuration Management. *Comput. Surveys* 30, 2 (June 1998), 232–282. https://doi.org/10.1145/280277.280280

[6] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In *13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS 2019)*. ACM, New York, USA, Article 16, 10 pages. https://doi.org/10.1145/3302333.3302343

[7] Sebastian Deorowicz, Agnieszka Debudaj-Grabysz, and Adam Gudyś. 2014. Kalign-LCS — A More Accurate and Faster Variant of Kalign2 Algorithm for the Multiple Sequence Alignment Problem. In *Man-Machine Interactions 3*, Dr. Aleksandra Gruca, Tadeusz Czachórski, and Stanisław Kozielski (Eds.). Springer International Publishing, Cham, 495–502.

[8] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* 25, 1 (2013), 53–95. https://doi.org/10.1002/smr.567

[9] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *30th IEEE International Conference on Software Maintenance and Evolution* (Victoria, BC, Canada) *(ICSME 2014)*. IEEE, New York, USA, 391–400. https://doi.org/10.1109/ICSME.2014.61

[10] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *37th IEEE International Conference on Software Engineering* (Florence, Italy) *(ICSE 2015)*, Vol. 2. IEEE, New York, USA, 665–668. https://doi.org/10.1109/ICSE.2015.218

[11] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. A Source Level Empirical Study of Features and Their Interactions in Variable Software. In *16th International Working Conference on Source Code Analysis and Manipulation* (Raleigh, USA) *(SCAM 2016)*. IEEE, New York, USA, 197–206.

[12] Stefan Fischer, Rudolf Ramler, Lukas Linsbauer, and Alexander Egyed. 2019. Automating test reuse for highly configurable software. In *23rd International Systems and Software Product Line Conference* (Paris, France) *(SPLC 2019)*. ACM, Paris, France, 1–11. https://doi.org/10.1145/3336294.3336305

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

[14] Huong Ha and Hongyu Zhang. 2019. Performance-Influence Model for Highly Configurable Software with Fourier Learning and Lasso Regression. In *35th International Conference on Software Maintenance and Evolution* (Cleveland, OH, USA) *(ICSME 2019)*. IEEE, New York, USA, 470–480. https://doi.org/10.1109/ICSME.2019.00080

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

[16] Daniel Hinterreiter, Michael Nieke, Lukas Linsbauer, Christoph Seidl, Herbert Prähofer, and Paul Grünbacher. 2019. Harmonized Temporal Feature Modeling to Uniformly Perform, Track, Analyze, and Replay Software Product Line Evolution. In *18th International Conference on Generative Programming: Concepts & Experiences* (Athens, Greece) *(GPCE 2019)*. ACM, New York, USA, 115–128. https://doi.org/10.1145/3357765.3359515

[17] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my feature and what is it about? A case study on recovering feature facets. *Journal of Systems and Software* 152 (2019), 239–253. https://doi.org/10.1016/j.jss.2019.01.057

[18] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. 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* (Madrid, Spain) *(VAMOS 2018)*. ACM, New York, USA, 105–112. https://doi.org/10.1145/3168365.3168371

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

[20] Lukas Linsbauer, Thorsten Berger, and Paul Grünbacher. 2017. A classification of variation control systems. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017*, Matthew Flatt and Sebastian Erdweg (Eds.). ACM, New York, USA, 49–62. https://doi.org/10.1145/3136040.3136054

[21] L. Linsbauer, S. Fischer, R. E. Lopez-Herrejon, and A. Egyed. 2015. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *8th International Symposium on Software and Systems Traceability* (Florence, Italy) *(SST 2015)*. IEEE, New York, USA, 57–60. https://doi.org/10.1109/SST.2015.16

[22] Lukas Linsbauer, E. Roberto Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability between Features and Code in Product Variants. In *17th International Software Product Line Conference* (Tokyo, Japan) *(SPLC 2013)*. ACM, New York, USA, 131–140. https://doi.org/10.1145/2491627.2491630

[23] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Software and Systems Modeling* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

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

[25] Jia Liu, Don Batory, and Christian Lengauer. 2006. Feature Oriented Refactoring of Legacy Applications. In *28th International Conference on Software Engineering* (Shanghai, China) *(ICSE 2006)*. ACM, New York, USA, 112–121. https://doi.org/10.1145/1134285.1134303

[26] Jabier Martinez, Tewfik Ziadi, Mike Papadakis, Tegawendé F. Bissyandé, Jacques Klein, and Yves le Traon. 2018. Feature location benchmark for extractive software product line adoption research using realistic and synthetic Eclipse variants. *Information and Software Technology* 104 (2018), 46 – 59. https://doi.org/10.1016/j.infsof.2018.07.005

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

[28] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. 2019. Comparison-based feature location in ArgoUML variants. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC 2019)*. ACM, New York, USA, 17:1–17:5. https://doi.org/10.1145/3336294.3342360

[29] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2020. Mining Feature Revisions in Highly-Configurable Software Systems. In *Proceedings of the 24th International Systems and Software Product Line Conference, SPLC 2020, Volume B, Montréal, Canada, October 19-23, 2020*. ACM, New York, USA, 1–5. https://doi.org/10.1145/3382026.3425776

[30] Mukelabai Mukelabai, Damir Nešiundefined, Salome Maro, Thorsten Berger, and Jan-Philipp Steghöfer. 2018. 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* (Montpellier, France) *(ASE 2018)*. ACM, New York, USA, 155–166. https://doi.org/10.1145/3238147.3238201

[31] Richard Müller and Ulrich Eisenecker. 2019. A Graph-Based Feature Location Approach Using Set Theory. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC 2019)*. ACM, New York, , USA, 88–92. https://doi.org/10.1145/3336294.3342358

[32] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag, Berlin, Heidelberg.

[33] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering, Product Lines, Languages, and Conceptual Models*. Springer Berlin Heidelberg, Heidelberg, DE, 29–58. https://doi.org/10.1007/978-3-642-36654-3_2

[34] Kai Ming Ting. 2010. *Precision and Recall.* Springer US, Boston, MA. 781–781 pages. https://doi.org/10.1007/978-0-387-30164-8_652

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

[36] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-condition Simplification in Highly Configurable Systems. In *37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE 2015)*. IEEE, New York, USA, 178–188.