FHistorian: Locating Features in Version Histories

Yi Li
University of Toronto
liyi@cs.toronto.edu

Chenguang Zhu
University of Toronto
czhu@cs.toronto.edu

Julia Rubin
University of British Columbia
julia.rubin@ubc.ca

Marsha Chechik
University of Toronto
chechik@cs.toronto.edu

ABSTRACT

Feature location techniques aim to locate software artifacts that implement a specific program functionality, a.k.a. a feature. In this paper, we build upon the previous work of semantic history slicing to locate features in software version histories. We leverage the information embedded in version histories for identifying changes implementing features and discovering relationships between features. The identified feature changes are fully functional and guaranteed to preserve the desired behaviors. The resulting feature relationship graph is precise and can be used to assist in understanding of the underlying connections between the features.

We evaluate the technique on a number of real-world case studies and compare our results with developer-specified feature annotations. We conclude that, when available, historical information of software changes can lead to precise identification of features in existing software artifacts.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software version control**;

KEYWORDS

Feature location, version history, feature relationship

ACM Reference format:
Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. FHistorian: Locating Features in Version Histories. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 10 pages.
https://doi.org/10.1145/3106195.3106216

1 INTRODUCTION

Feature location techniques aim to locate pieces of code that implement a specific program functionality, a.k.a. a *feature*. These techniques support developers during various maintenance tasks, e.g., locating code of a faulty feature that requires fixing, and are extensively studied in the literature [11, 29]. The techniques are based on static or dynamic program analysis, information retrieval (IR), change set analysis, or some combination of the above.

Recently, a number of techniques for identifying features in the Software Product Line (SPL) context have been proposed [2, 3, 21–23, 37, 39, 42]. Most such techniques are based on intersecting code of multiple product variants in order to identify code fragments shared by variants with a particular feature. The identified code fragments can then be attributed to that feature. These intersection-based techniques operate in a static manner and are effective when a large number of product variants are available.

Often, we cannot rely on the availability of a large number of variants. For example, consider a family of related software products realized via *cloning* (a.k.a. the "clone-and-own" approach) – a routinely used practice where developers create a new product by copying / branching an existing variant and later modifying it independently from the original [12]. Such variants are commonly maintained in a version control system, e.g., Git [14]. Their number can be relatively small, e.g., 3-10 products, while intersection-based techniques [23], are typically evaluated for tens or even hundreds of variants.

Identifying features in cloned variants is important for a variety of software development tasks. For example, developers often need to share features between variants. That becomes a challenging task as it is often unclear which commits correspond to the particular feature of interest [5, 18, 32]. Refactoring cloned variants into *single-copy* SPL representations also relies on the ability to identify and extract code that implements each feature [5, 28, 30–32].

As a step towards addressing these problems, this paper contributes a dynamic technique, called FHistorian, for locating features in software version histories. Our technique differs from other work in that it (a) traces features to historical information about their evolution, (b) leverages version histories to improve the accuracy of feature location, and (c) is efficient even if the number of available product variants is small.

Being dynamic, FHistorian relies on the availability of a test suite T_f exercising a feature of interest f; such test suites are commonly provided by developers for validating features. Starting from T_f, our technique "slices" the history to identify the commits relevant for f. It also analyses the slices produced for multiple features $f_1, \ldots, f_n$ in order to identify relationships between these features and build a feature model that represents the extracted information. The generated feature model guarantees that all product variants it describes are well-formed, as it captures runtime dependencies between features.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '17, September 25-29, 2017, Sevilla, Spain
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5221-5/17/09…$15.00
https://doi.org/10.1145/3106195.3106216

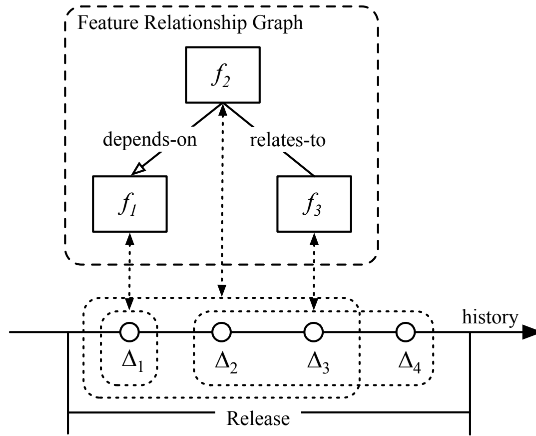


Figure 1: A schematic feature relationship graph for three features extracted from `commons-csv v1.3`.

Example. We take a history fragment from the release version 1.3 of an open source software project `commons-csv` [9] and the simplified commit history is shown in Fig. 1 as a sequence of commits $\langle \Delta_1, \Delta_2, \Delta_3, \Delta_4 \rangle$. There are three features implemented in this fragment: features "CSV-159" (f_1: add IgnoreCase option for accessing header names), "CSV-179" (f_2: add shortcut method for using first record as header to `CSVFormat`), and "CSV-180" (f_3: add `withHeader(Class? extends Enum>)` to `CSVFormat`).

FHistorian identifies a minimal set of commits required by each feature and generates a feature relation graph depicting the relationships between the features as shown in Fig. 1. The commits implementing f_1, f_2 and f_3 are $\{\Delta_1\}$, $\{\Delta_1, \Delta_2, \Delta_3\}$, and $\{\Delta_2, \Delta_3, \Delta_4\}$, respectively. Since the commit implementing f_1, namely, Δ_1, is required by f_2 to execute correctly, we say that f_2 *depends on* f_1. Similarly, since f_2 and f_3 both require commits Δ_2 and Δ_3, we say that they *relate to* each other.

The resulting feature model annotated with feature-implementing changes (or *feature changes* for short) is useful for understanding dependencies and connections between features from an evolutionary view point. Each valid product has to respect the inferred *depends-on* relationships in order to function correctly. The *relates-to* relationship indicates connections between features. They often reveal underlying hidden dependencies which are essential across the system.

These relationships indeed exist among the analyzed features. The correct behaviors of CSV-179, "using first record as headers to `CSVFormat`" requires the "IgnoreCase option" (CSV-159) being enabled to produce correct headers. Both CSV-179 and CSV-180 add new functionalities to the `CSVFormat` class and thus are connected to each other.

Contributions. Our previous work on semantic history slicing [18, 19] locates feature-implementing changes for a single feature at a time. This paper extends it in the following ways:
(1) It defines FHistorian– a dynamic approach for locating *multiple* features in version histories and building a well-formed feature model representing runtime relationships between the features; and (2) It improves the precision of the feature location technique in [18, 19] by performing hunk-level minimizations. (3) We evaluate the proposed technique on five real-world examples and show its accuracy and effectiveness.

The rest of the paper is structured as follows. Sect. 2 provides the necessary background and definitions for the rest of the paper. In Sect. 3, we introduce our history-based feature analysis technique FHistorian and describe its feature location and feature relation inference capabilities. Sect. 4 presents the evaluation of FHistorian in real-world case studies. We discuss related work in Sect. 5 and conclude in Sect. 6.

2 BACKGROUND

In this section, we provide background and definitions needed for the rest of the paper.

Feature and Feature Tests. While there is no universal agreement on what a feature is (and what it is not), we adopt the definition by Kang et al. [17]:

Definition 2.1. (Feature [8, 17]). A *feature* is a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained. A feature consists of a label and a short description that identifies its behavior. For conciseness, either the label or the feature description can be dropped when clear from the context.

We assume that the functionalities of features can be captured by test cases and the execution trace of a test case is deterministic. A *test case* t is a function $t : P \mapsto \mathbb{B}$ such that for a given program p, $t(p)$ is true if the test succeeds, and false otherwise. A *test suite* is a collection of unit tests that can exercise and demonstrate the functionality of interest. Let a test suite T be a set of test cases $\{t_i\}$. We write $p \models T$ if and only if a program p passes all tests in T, i.e., $\forall t \in T \cdot t(p)$.

Commit and Commit History. Let $\Delta : P \mapsto P$ be a *commit* which takes a program version p and transforms it to produce a new program version $\Delta(p)$. A commit is a collection of *hunks* [13, 18] $\{\delta_0, \ldots, \delta_n\}$, in no particular order, each representing a set of line changes with an approximate locality. Composing hunks is equivalent to applying the original commit, i.e., $\Delta = \delta_0 \circ \cdots \circ \delta_n$.

A *commit history* is a sequence of commits $H = \langle \Delta_1, \ldots, \Delta_k \rangle$. A *sub-history* is a sub-sequence of a history, i.e., a sequence derived by removing changes from H without altering the ordering. We write $H' \subseteq H$ indicating that H' is a sub-history of H, and refer to $\langle \Delta_i, \ldots, \Delta_j \rangle$ as $H_{i..j}$. We use $\mathcal{H}$ to denote the set of all sub-histories of H.

Semantics-Preserving History Slice. Consider a program $p_0 \in P$ and its n subsequent versions $p_1, \ldots, p_n$ such that they are all well-formed. Let H be the original commit history from p_0 to p_n, i.e., $H_{1..i}(p_0) = p_i$ for all integers $0 \leq i \leq n$. Let T be a set of tests passed by p_n, i.e., $p_n \models T$.

Definition 2.2. (Semantics-preserving slice [18]). A *semantics-preserving slice* of history H with respect to T, denoted by $H' \subseteq_T H$, is a sub-history of H, i.e., $H' \subseteq H$, such that $H'(p_0) \models T$.

Definition 2.3. (Minimal semantics-preserving slice) [20]. A semantics-preserving slice H^* is a *minimal* if $\forall H_{sub} \subset H^* \cdot H_{sub} \not\models T$.

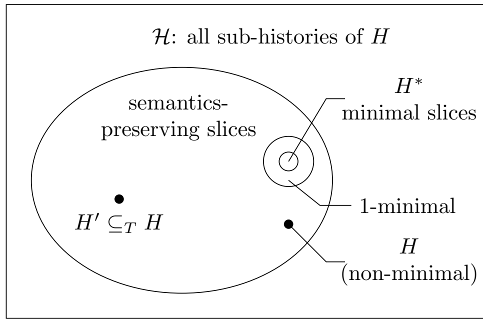


Figure 2: Relationships between various history slices.

As shown in Fig. 2, there are several special kinds of semantics-preserving slices. First, H is a semantics-preserving slice of itself, but it may not be *minimal*. Second, minimal semantic slices (H^*) are slices which are semantics-preserving and cannot be reduced further. Finally, computing minimal semantics-preserving slices is expensive [20], so we often compute an approximation known as the *1-minimal* semantic slice – a slice which cannot be further reduced by removing *any single commit*. In practice, 1-minimal slices are often minimal [19].

Slicing Software Histories. With the presence of adequate tests for a feature, and the feature development history, *semantic history slicing* [18, 19] is a technique which uses tests (slicing criteria) to identify commits in the history (i.e., a semantics-preserving slice) that contribute to the implementation of the given feature. The history slicing techniques have been successfully applied to back-porting bug fixes [18], creating self-contained and easy-to-merge pull requests [20], and transforming existing development histories [18, 25] to assist evolution understanding.

Currently, two history slicing techniques exist – CSlicer [18] and Definer [19]. The key difference between the two is that CSlicer runs the tests only once to collect test coverage information and then computes a semantics-preserving history slice that is not necessarily minimal. On contrast, Definer derives a small and precise semantic slice through the more expensive repeated test executions in a divide-and-conquer fashion that is very similar to *delta debugging* [40]. The high-level idea is to partition the input history by dropping some subset of the commits and opportunistically reduce the search space when the target tests pass on one of the partitions, until a minimal partition is reached. Definer operates on the commit-level, and the history slices produced by Definer is guaranteed to be *1-minimal* – removing any single commit from the history slice will break the desired feature behaviors.

3 OUR APPROACH

We now present FHistorian– a feature location technique based on the analysis of commit histories. Software version histories are often organized not in terms of features, but as a sequence of incremental development activities, ordered by timestamps. This usually results in a history mixed with changes to multiple features which may or may not be related to each other. Given a piece of history H which is known to implement a set of features F, and

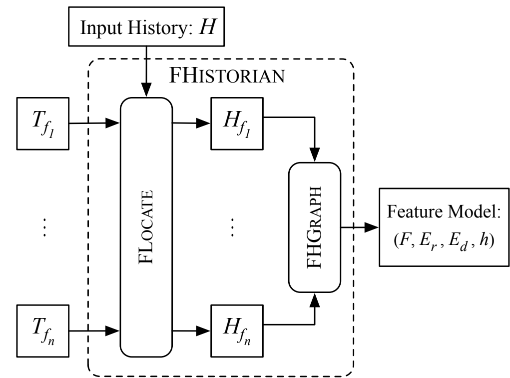


Figure 3: Overview of FHistorian architecture.

each feature $f \in F$ exercised by a test suite T_f, we would like to identify a set of relevant changes for each of the features.

Fig. 3 gives an overview of the FHistorian work flow. FHistorian is built on top of the semantic history slicer Definer. First, we recognize that semantic history slicing, as described in [18, 19] and summarized in Sec. 2, is directly applicable for dynamically locating single features, one at a time. An improved version, with hunk-level minimization, is implemented by the FLocate component shown in Fig. 3. It receives an input history H and a feature test T_f and produces a 1-minimal set of changes relevant to this feature. Then, by consolidating history information of all the target features, the FHGraph component is able to produce a feature model capturing inter-feature relationships such as the runtime dependencies between features. We describe these components below.

3.1 FLocate: History Slicing with Hunk-Minimization

The FLocate component of FHistorian is inspired by the existing history slicing technique Definer, extended with *hunk-level minimization*.

In practice, commits usually contain changes to many files and multiple classes and methods, organized as hunks. A *hunk* is the smallest unit of code change in language-agnostic version control systems [13]. Different hunks in the same commit are not necessarily logically related or relevant to the same feature. Considering a commit as an atomic unit does not allow us to remove many unnecessary changes for the target features.

Example 3.1. Fig. 4 shows a diagram illustrating the sources of imprecision in commit-level history slicing. The history segment H contains four commits, i.e., $H = \langle \Delta_1, \Delta_2, \Delta_3, \Delta_4 \rangle$. Each commit can be further broken into a set of hunks potentially spanning over multiple files. For instance, Δ_1 has two hunks, δ_a and δ_b, over files A and B, respectively.

The only feature-related changes in this example are δ_b and δ_e, shaded in gray. However, when performing history slicing on a commit level, we have to inevitably include unnecessary changes due to *commit dependencies* – two hunks in the same commit are *commit-dependent* on each other. For example, δ_a is included because of δ_b, and δ_f is included because of δ_e (commit bundles are depicted as dashed boxes in Fig. 4).

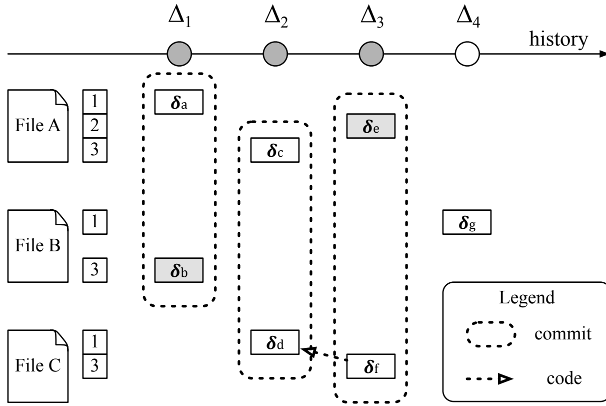


Figure 4: An illustration of the sources of imprecision in commit-level history slicing.

Unnecessary changes introduced by commit dependencies can induce further imprecisions. For example, δ_f relies on an earlier hunk δ_d in order to function correctly (shown as a dashed arrow in Fig. 4). This is known as a *code dependency* – including dependencies between a child and parent code entities, between a variable usage and definition, etc. The inclusion of δ_f therefore forces us to include δ_d as well, due to code dependency.

Thus, with commit-level history slicing, the best result achievable is a sub-history of length three: $\langle \Delta_1, \Delta_2, \Delta_3 \rangle$.

Instead of stopping at the minimal history slices at the commit-level, we zoom into individual hunks of commits, to obtain minimal *hunk slices*. This process yields a set of feature-implementing hunks which are potentially much smaller than the corresponding original commits and contain significantly fewer unrelated changes. For example, hunk slicing in Ex. 3.1 allows us to reduce the number of unnecessary changes, resulting in δ_b and δ_e, as intended. In Sect. 4, we empirically show that hunk-level minimization significantly improves the precision of FLocate in locating feature changes.

3.2 FHGraph: Inferring Feature Relationships

In addition to locating features in version histories, we also utilize the obtained feature-change information to understand the underlying relationships between features within the same history segment. In particular, we infer two types of feature relationships: *relates-to* and *depends-on*, and represent them in a feature relationship graph. The identified relationships respect well-formedness and functionalities of target features – satisfying feature dependencies is the prerequisite of producing a fully functional product variant. The produced feature model can also assist developers in recognizing interactions between software components by revealing underlying hidden connections.

Feature Relationship Graph. A *feature relationship graph* with respect to a set of features F implemented within a history H is a tuple (F, E_r, E_d, h), where (F, E_r) is an undirected graph whose nodes are features and edges are relates-to relationships. Similarly, (F, E_d) is a directed graph for depends-on relationships. $h : F \mapsto \mathcal{H}$ is a map from features to feature changes.

```
δ1 | i:int f1(){return 1;}
   | ...
δ2 | j:int f2(){return f1()+1;}
   | ...
δ3 | k:int f3(){return f1()-1;}
```
T_{f_1} : `f1()==1`, T_{f_2} : `f2()==2`, T_{f_3} : `f3()==0`

Figure 5: An example illustrating feature relationships.

Relates-To Relationship. We say that a feature f_1 *relates to* another feature f_2 if they both rely on the presence of the same set of non-empty changes $H_{(f_1,f_2)} \subseteq H$ in order to function correctly. Formally, we write $f_1 \leftrightarrow f_2$ when

$$\forall H' \subseteq H \cdot (H' \models T_{f_1} \wedge H' \models T_{f_2}) \Rightarrow H_{(f_1,f_2)} \subseteq H', (H_{(f_1,f_2)} \neq \emptyset).$$

Depends-On Relationship. Similarly, a feature f_1 *depends on* another feature f_2 if f_1 functions correctly only when f_2 does so as well. More formally, we write $f_1 \rightarrow f_2$ if

$$\forall H' \subseteq H \cdot (H' \not\models T_{f_2}) \Rightarrow (H' \not\models T_{f_1}).$$

More generally, a product variant defined by the history H_v for features $F_v = \{f_1, \ldots, f_n\}$ is semantics-preserving (i.e., $\bigwedge_{f_i \in F_v} H_v \models T_{f_i}$) only when F_v is closed under their feature dependencies. In other words, satisfying all of the dependencies is the prerequisite for the product variant to behave as expected.

Example 3.2. Fig. 5 shows an example of a history $H = \langle \delta_1, \delta_2, \delta_3 \rangle$ containing changes to three features: $F = \{f_1, f_2, f_3\}$. During the given history, three lines of code are inserted one after another – Line i, Line j and then Line k – each reflecting a feature implementation. The corresponding feature tests $\{T_{f_1}, T_{f_2}, T_{f_3}\}$ are shown at the bottom. For example, the feature f_1 is implemented as a function `f1()` which is expected to return an integer 1.

It is easy to see that f_2 depends on f_1 and f_3 depends on f_1, since both functions `f2()` and `f3()` require the definition of `f1()` introduced in the change δ_1. Likewise, we also have that f_2 is related to f_3, and their witness change $H_{(f_2,f_3)}$ is δ_1.

Discovering Feature Relationships. To infer relationships for the target features, we consolidate all the history slicing results returned by FLocate and determine pairwise feature relationships by comparing their minimal semantics-preserving slices.

For example, when the semantic slice of f_1 is subsumed by that of f_2, i.e., $H_{f_1} \subseteq H_{f_2}$, we would say f_2 depends on f_1 and also factor out H_{f_1} from the semantic slice of f_2. The resulting feature changes identified for f_2 would be $H_{f_2} \setminus H_{f_1}$. More formally, the algorithm for inferring feature relationships is based on the following theorem.

Theorem 3.3. *Suppose the minimal semantics-preserving slices for f_1 and f_2 are both unique in H, denoted by H_{f_1} and H_{f_2}, respectively. We have $(f_1 \leftrightarrow f_2) \Leftrightarrow (H_{f_1} \cap H_{f_2}) \neq \emptyset$, and $(f_2 \rightarrow f_1) \Leftrightarrow H_{f_1} \subseteq H_{f_2}$.*

The key for proving this theorem lies in realizing that if the minimal semantics-preserving slice H_f for a feature f is unique, then H_f is essential for passing the feature test T_f, i.e., $\forall H' \subseteq H \cdot H' \models T_f \Rightarrow H_f \subseteq H'$. Hence, $H_{f_1} \cap H_{f_2}$ serves as the witness for $f_1 \leftrightarrow f_2$, i.e., $H_{(f_1,f_2)} = H_{f_1} \cap H_{f_2}$. Similarly, when $H_{f_1} \subseteq H_{f_2}$, H_{f_1} is essential for both f_1 and f_2.

```
1:  procedure FHGraph(F, H)
2:      E_r, E_d, h ← ∅, ∅, ∅
3:      for f ∈ F do
4:          H_f ← FLocate(H, T_f)                ▷ get minimal slices
5:          h(f) ← H_f
6:      end for
7:      for (f_1, f_2) ∈ F × F s.t. f_1 ≠ f_2 do
8:          if H_f1 ∩ H_f2 = ∅ then continue
9:          if H_f1 ⊆ H_f2 then
10:             E_d ← E_d ∪ (f_2 → f_1)          ▷ depends-on
11:             h(f_2) ← H_f2 \ H_f1             ▷ factor out H_f1 from H_f2
12:         else if H_f2 ⊈ H_f1 then
13:             E_r ← E_r ∪ (f_1 ↔ f_2)          ▷ relates-to
14:         end if
15:     end for
16:     return (F, E_r, E_d, h)
17: end procedure
```

Figure 6: An algorithm for constructing a history-based feature relationship graph.

With the assumption of unique minimal semantics-preserving slices, an algorithm for constructing feature relationship graph is given in Fig. 6. The procedure FHGraph receives a set of features F and the history H implementing them. It updates two edge sets, E_r and E_d, for the relates-to and depends-on relationships, respectively. It also maintains a map h which stores the final feature location results for each feature.

First, we store the minimal semantics-preserving slices returned by FLocate for all features (Line 4). The algorithm then iterates through all feature pairs (Line 7 – 15). For each pair of minimal slices H_{f_1} and H_{f_2}, if H_{f_1} is subsumed by H_{f_2}, then we create a depends-on edge $f_2 \rightarrow f_1$ and factor out H_{f_1} from H_{f_2} and store it into $h(f_2)$ (Line 10 and 11). If there is no depends-on relationship between f_1 and f_2, then a relates-to edge is constructed instead (Line 13). The procedure returns a feature relationship graph (F, E_r, E_d, h) when it terminates.

4 EVALUATION

In this section, we present the empirical evaluation of our approach on real-world software systems. Our goal is to have a better understanding of the capability of our history-based feature analysis technique by answering the following research questions:
RQ1: How accurate is the feature location performed by FHistorian? **RQ2:** How accurate are the feature relationships inferred by FHistorian?

We implemented FHistorian on top of Definer [19], and used the interactive mode of the Git add command to automatically split commits into hunks. We also generated feature relation graphs represented using the DOT graph format. Our prototype implementation is available at bitbucket.org/liyistc/gitslice.

4.1 Subjects

To effectively evaluate FHistorian, we need access to project source code, test cases, and commit histories to run feature analysis, as well as adequate feature annotations to determine its effectiveness. In particular, to perform feature location within a history segment, FHistorian requires a predefined set of features, known to be implemented in the history period, along with test cases.

Table 1: Experimental subjects.

Project & Release	#C	#F	LOC	#Issue	Features	
					#New	#Tested
`commons-csv v1.3`	79	28	2353	12	7	4
`commons-compress v1.13`	148	144	6650	13	7	6
`commons-io v1.4`	140	140	8607	24	18	9
`commons-io v2.2`	136	182	7328	24	15	7
`commons-lang v3.4`	262	146	8817	63	17	10

To find suitable evaluation subjects, we looked for complete histories between two software releases and referred to release notes to determine the newly implemented feature set. We selected experimental subjects from a combination of recently published history analysis datasets [19, 41] which are well-documented and organized, and ended up with five releases accompanied by comprehensive release notes and feature annotations. All selected software projects use JIRA [16] as their issue tracking system. In JIRA, each feature has a unique developer-assigned ID, referred to as the "issue key", which is associated with an issue report recording detailed information about the feature. A JIRA issue key is a string with the format "ABC-123", where "ABC" stands for the name of the project containing this feature, and "123" is a unique ID. Developers label commits with issue keys to indicate the purpose of the changes, which enables us to determine which feature models the developers had in mind.

The set of features implemented during the release histories was determined from the release notes. In all the release notes that we analyzed, newly implemented functionalities are organized as issues including new features, tasks, bug fixes, improvements, etc. For the purpose of our experiments, we focused on releases that had at least four implemented and tested features. The resulting subjects are summarized in Table 1. Each row represents a history segment for a particular release. Column "Project & Release" designates the name of the project from which the releases are chosen, followed by their version number. Columns "#C", "#F", "LOC" represent the number of commits, the number of files modified, and the number of lines of code changed during the the release histories, respectively. Column "#Issue" represents the number of all issues reported. Column "#New" and "#Tested" represents the number of all new features and those with associated tests cases. For example, the second row of Table 1 shows that during the development of version 1.13 of the project `commons-compress`, there were 148 commits, 144 files were modified and 6650 lines of code have been changed. Developers created 13 issues, seven of which were documented as new features; among them, six were accompanied by test cases capturing the expected feature behaviors.

4.2 RQ1: Precision of Feature Location

Methodology. We compared the feature location results of FHistorian with developers' feature annotations found in the commit logs. For each new feature found in the release notes, we used the assigned JIRA issue key to map the feature with the commits considered as the feature implementations by the developers. We then ran FHistorian on the whole feature set, taking feature relations

into consideration, to compute relevant changes for each feature. Finally, we repeated the same experiments with hunk-level minimization disabled to decide the effectiveness of our optimizations on commit-level history slicing.

Results. The studies were conducted on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. It took on average 4,062 seconds for DEFINER to obtain a 1-minimal semantic slice for each feature. The hunk minimization on the resulting slice took on average 3,740 seconds per feature.

Table 2 lists the feature location results of FHISTORIAN, comparing them with the developers' feature annotations. Each row in the table shows results for a particular feature, identified by the feature key. Column "Releases" lists the release histories being analyzed. Columns "#Labeled" and "#Found" show the number of commits labeled by the developers and identified by FHISTORIAN, respectively. We also list the differences between their results in the last two columns – column "#FN" shows the number of commits labeled by developers but not found by us and vice versa for column "#FP". For instance, the developers annotated one commit for feature "CSV-159" and FHISTORIAN found the same commit. However, three commits were annotated by the developers and FHISTORIAN found one of them with six extra commits and missed the other two.

For 15 out of 36 features, FHISTORIAN's results match perfectly with the developers' annotations. To understand the differences in the rest of the cases, we analyzed all of FHISTORIAN's the false positives and false negatives.

False Positives. FHISTORIAN includes not only conceptually essential changes but also peripheral changes to guarantee the executability of its produced feature models. When committing and labeling feature changes, developers often overlooked preexisting changes which support the compilation and execution of the features. For example, FHISTORIAN considered commit `f8e09945` as necessary for the feature COMPRESS-369 but it is not labeled by the developers. We inspected the commit, finding it to be a bug fix updating the configuration file to use a newer Java JDK. The target feature code cannot be compiled or executed when ignoring this commit, and thus it is essential.

The second reason why FHISTORIAN detects more commits is that it also takes *hunk dependencies* [18] specific to text-based version control systems into consideration. That is, FHISTORIAN includes additional commits providing a necessary context for the application of the essential commits. We verified that all the feature changes found by FHISTORIAN were minimal, i.e., they could not be further reduced and yet pass the feature tests. Thus, all commits found by FHISTORIAN but not labeled by the developers were required for the correct execution of feature tests.

False Negatives. On the other hand, FHISTORIAN occasionally missed commits labeled by the developers (28% missed). We manually inspected each missed commit and summarize the most common reasons below.

Some commits missed by FHISTORIAN contained only changes which did not affect feature execution. For example, developers occasionally created separate commits that updated the release note file documenting addition of a new feature, and then labeled them as part of the feature. There were also commits labeled as feature implementations which only updated Javadocs or performed refactoring. FHISTORIAN also missed commits which performed minor optimizations which were labeled as part of the feature but the associated feature tests were not updated to capture the modified behaviors.

Table 2: Feature location results of FHISTORIAN compared with developer annotations.

Releases	Feature Keys	#Labeled	#Found	#FN	#FP
`csv v1.3`	CSV-159	1	1	0	0
	CSV-175	3	6	2	5
	CSV-179	1	8	0	7
	CSV-180	1	8	0	7
`compress v1.13`	COMPRESS-327	10	17	2	9
	COMPRESS-368	6	5	3	2
	COMPRESS-369	2	5	1	4
	COMPRESS-373	1	7	0	6
	COMPRESS-374	1	4	0	3
	COMPRESS-375	2	1	1	0
`io v1.4`	IO-126	1	1	0	0
	IO-129	2	1	1	0
	IO-130	1	1	0	0
	IO-135	2	1	1	0
	IO-138	1	1	0	0
	IO-144	1	1	0	0
	IO-145	1	1	0	0
	IO-148	4	1	3	0
	IO-153	1	1	0	0
`io v2.2`	IO-173	1	1	0	0
	IO-275	1	1	0	0
	IO-288	3	1	2	0
	IO-290	1	1	0	0
	IO-291	3	2	1	0
	IO-297	1	1	0	0
	IO-305	1	1	0	0
`lang v3.4`	LANG-536	1	3	0	2
	LANG-883	1	1	0	0
	LANG-993	1	1	0	0
	LANG-999	1	1	0	0
	LANG-1015	1	1	0	0
	LANG-1021	1	2	0	1
	LANG-1033	1	1	0	0
	LANG-1080	1	1	0	0
	LANG-1082	1	1	0	0
	LANG-1093	2	1	1	0

Effectiveness of Hunk-level Minimization. The comparison of the feature location precision results of FHISTORIAN with and without the hunk-level minimization (i.e., staying at the commit level) are shown in Fig. 7. On average, hunk-level minimization improved FHISTORIAN's precision 3.47 times. In particular, for releases `commons-io v1.4` and `commons-io v2.2`, hunk-level minimization yielded a 16X improvement in precision (from 5.21% to 81.8% and from 6.32% to 100%, respectively).

Answer to RQ1. FHISTORIAN precisely identifies commits that implement a specific feature and required for the feature execution. Its feature location results coincide with the developer annotations with regard to essential feature implementation changes, while they differ from the annotations when considering changes that do not affect the test execution. The commit-level optimizations for

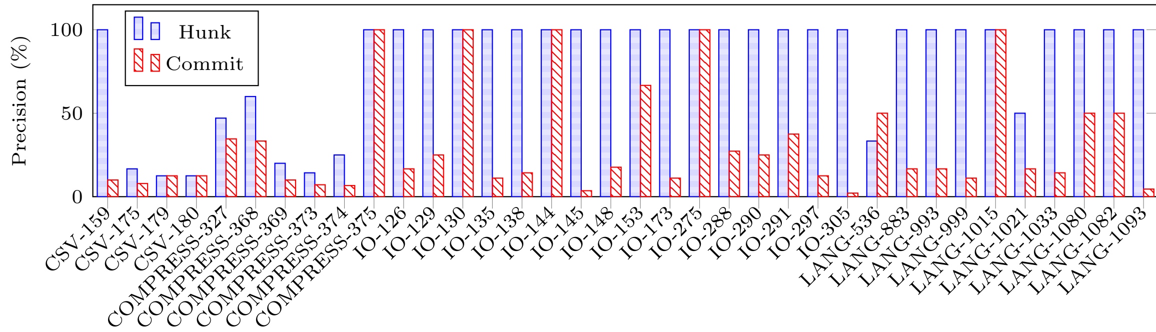


Figure 7: Precision of FHistorian's feature location with and without hunk-level minimization.

history slicing are effective in improving the precision of feature location.

4.3 RQ2: Accuracy of Feature Relation

Methodology. In this experiment, we evaluated the accuracy of feature relationships inferred by FHistorian through code inspections and qualitative studies of feature documentation. For each subject release, we ran FHistorian with the same feature set used in answering RQ1 to generate a feature relationship graph. We then used additional feature annotations extracted from release notes to refine the feature relationships. Finally, we analyzed each of the inferred feature relationships and verified it either by internal code inspection or external evidence such as log messages and issue links on JIRA issue pages.

Results. To simplify the discussion, we categorize the five releases studied into two groups. In one group, namely, `commons-io v1.4 v2.2` and `commons-lang v3.4`, the feature relationships are relatively sparse and only depends-on relationships are observed. In the other group, namely, `commons-csv v1.3` and `commons-compress v1.13`, the feature relationships are more complex and the relates-to relationships between tested features often reveal "hidden" untested features. We show that with the additional knowledge about the hidden features, the relates-to relationships can be refined and reflect more accurate relationships among analyzed features.

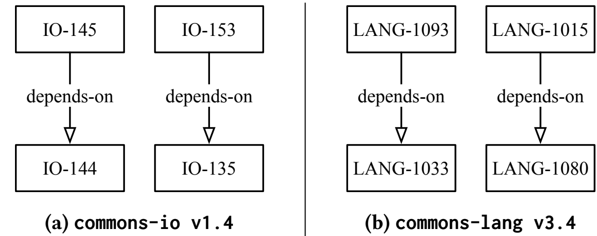
Case Study 1: `commons-io` and `commons-lang`. No feature relationships were observed for release v2.2 of `commons-io`. All analyzed features in this release can be independently executed using non-overlapping commits.

There are two depends-on edges detected in release v1.4 of `commons-io`: from IO-153 to IO-135 and from IO-145 to IO-144 (see Fig. 8a, where feature nodes without relationships are omitted).

Similarly, Fig. 8b illustrates the relationship between the features in release v3.4 of `commons-lang`. This example also has two depends-on edges: from LANG-1093 to LANG-1033, and from LANG-1015 to LANG-1080.

To verify the feature relationship found by FHistorian, we report evidence observed from log messages and source code.

(1) IO-144 → IO-145, IO-153 → IO-135, LANG-1093 → LANG-1033. The three depends-on edges were verified by inspecting contents of the commits. For example, feature IO-144 is implemented by a



(a) `commons-io v1.4` (b) `commons-lang v3.4`

Figure 8: Feature relationships for `commons-io` and `commons-lang`.

single commit `db3e834e` with the commit message "add a compare method". We inspected the code changes in this commit and found that it creates a special "checkCompareTo()" method for comparing strings with the ability to adjust case sensitivity. The commit message of IO-145 (`55dfa6eb`) is "add new package of file comparator implementations". This commit creates and modifies multiple file comparator classes. Three of these, namely, "ExtensionFileComparator", "NameFileComparator", and "PathFileComparator", rely on the "checkCompareTo()" method for file name comparison. This observation confirms that feature IO-145 depends on IO-144 for its implementation, which in turn supports the identified feature relationship. The other two edges are similar – one feature provides a utility function which is then used by the other feature, thus creating a feature dependency between the two.

(2) LANG-1015 → LANG-1080. FHistorian's results indicate that feature LANG-1015 is implemented by two commits, one of which labeled as LANG-1015 and the other as LANG-1080. By inspecting the commits, we concluded that LANG-1015 depends on LANG-1080 due to the hunk dependency. Patching the former without the latter using Git would result in a conflict because the two commits modify adjacent lines in the same file. This dependency can of course be ignored if a language-aware version control system (e.g., SemanticMerge [34]) is used instead.

Case study 2: `commons-csv` and `commons-compress`. In this case study, we first show the feature models produced by FHistorian using tested features. We then show how additional feature annotations extracted from release notes and log messages can be used to refine these feature models (e.g., with explicating "hidden features").

(1) commons-csv. Fig. 9a (top) illustrates the feature relationship graph originally obtained by FHistorian in the release v1.3 of `commons-csv`. In this example, there are three depends-on edges, indicating that CSV-175 → CSV-159, CSV-179 → CSV-159, and CSV-180 → CSV-159, respectively. There are also three relates-to edges, showing that CSV-175, CSV-179, and CSV-180 are related to each other.

To obtain a more precise relationship between these features, we took into consideration bug fix annotations in the release notes and discovered that feature CSV-175's member commits can be separated into two groups, with one implementing the feature's functionality, and the other, labeled as CSV-169, contributing to a bug-fix. The bug-fix does not belong to CSV-175 technically. But since there is no test case associated with it, FHistorian could not distinguish the bug-fix commit from the feature-implementing commit.

Using this obtained information, we created a new feature node, CSV-169, to represent the "hidden" bug-fix (in dashed box). We use node CSV-175' to represent the rest of the commits that originally belonged to CSV-175. With the updated feature set, FHistorian computed a newly refined relationship graph shown in Fig. 9a (bot). Since the commits of CSV-175' are fully subsumed by both CSV-179 and CSV-180, Two original relates-to relationships, CSV-179 ↔ CSV-175 and CSV-180 ↔ CSV-175 can be refined into stronger depends-on relationships, namely, CSV-179 → CSV-175' and CSV-180 → CSV-175'.

The refined feature relationships better reflect the actual situation. Some evidences confirming these relationships can be observed from the source code and commit messages. For example, CSV-179 and CSV-180 both rely on the "ignoreHeaderCase" option, created by CSV-159, to generate CSV file headers. This verified the inferred depends-on relationships CSV-179 → CSV-159 and CSV-180 → CSV-159.

(2) commons-compress v1.13. Fig. 9b (top) illustrates the feature relationships in release v1.13 of `commons-compress`. From FHistorian's result, we can determine that features COMPRESS-327, COMPRESS-368, COMPRESS-369, COMPRESS-373, and COMPRESS-374 all relate to each other. However, the underlying reasons for these relationships were unclear without additional knowledge about the software project.

We inspected the relevant commits, aiming to build a more precise feature relationship graph. We discovered that all features rely on a shared commit, `7e35f57`, labeled as COMPRESS-327. This shared commit upgrades a basic component – it re-implements the output stream of Zip format using a new class named "SeekableByteChannel", replacing an old class "RandomAccessFile". This upgraded component is widely used as a basis for many other functionalities related to stream compressors. Therefore, this commit affects all the five features mentioned earlier.

Another commit `f8e09945` is also shared among features. It contributes to the implementation of four features: COMPRESS-327, 368, 369, and 373. Upon further investigation, we determined that it is another bug-fix commit, labeled by the developers as COMPRESS-360, which updates the project's minimum JDK version requirement from 1.6 to 1.7. The change was made in the project configuration file and without this change, the other four features failed to compile.

We separated the shared commits from COMPRESS-327 to create individual nodes for them in the feature relationship graph ("Seekable" and "COMPRESS-360" in dashed boxes). As a result, the refined feature relationship graph is shown in Fig. 9b (bot). The new graph reveals multiple new feature relationships. First, it now shows that COMPRESS-327, COMPRESS-368, COMPRESS-369, and COMPRESS-373 all depend on the hidden nodes "Seekable" and "COMPRESS-360". In addition, COMPRESS-374 also depends on "Seekable". The original relates-to edges can be trivially inferred from the current graph: two nodes depending on the same node are automatically connected by related-to. We omit those edges in Fig. 9b (bot).

We now discuss evidences in support of the found feature relationships. We found direct evidence provided by the developers for the edge COMPRESS-368 ↔ COMPRESS-369. Developers explicitly labeled the relationship between COMPRESS-368 and COMPRESS-369 with a JIRA issue link, "is related to COMPRESS-369 (allow archiver extensions through a standard JRE ServiceLoader)" on the issue description page of COMPRESS-368.

Answer to RQ2. Feature relationships inferred by FHistorian are accurate. The depends-on relationships reflect runtime dependencies between features. They are essential for ensuring well-formedness and correct execution of the product variants constructed from the target features. The relates-to relationships are useful in revealing underlying connections between features and can be further refined into stronger depends-on relationships using additional project expertise such as issue tracking tickets, developer conversations, log messages, etc.

4.4 Threats to Validity

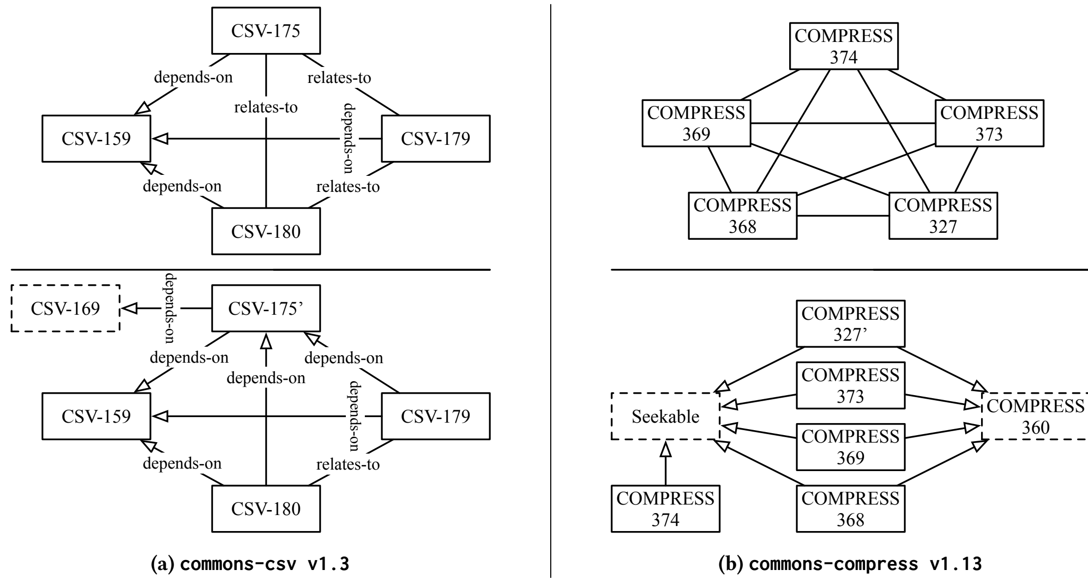
While we selected different projects and attempted to cover different scenarios, our results may not be sufficiently representative. Furthermore, the projects that we selected for evaluation have complete change logs and release notes. While our feature location technique produces encouraging results on our experimental subjects, it is not always applicable to projects that are not well-managed. In the absence of documentation of the release histories and the corresponding feature information, expert insights are required for FHistorian to achieve comparable good results.

Due to the absence of adequate feature documentation, it is not always possible to verify the feature relations obtained by FHistorian rigorously with developers' conceptual models. For example, we cannot be certain whether all of the relationships between the features have been generated. Our results were therefore confirmed by multiple indirect evidences such as commit messages, contents of code changes, etc.

5 RELATED WORK

We discuss related work in four areas given below.

Dynamic Feature Location. Dynamic feature location techniques rely on program execution for identifying source code that corresponds to a feature of interest. More than 10 such techniques are reviewed in [11, 29]. The earliest dynamic feature location technique which also became a foundation of future approaches is



(a) `commons-csv v1.3` (b) `commons-compress v1.13`

Figure 9: Feature relationships for `commons-csv` and `commons-compress`.

software reconnaissance [38]. It compares execution traces obtained by exercising the feature of interest to those obtained when the feature is inactive. The technique runs a set of test cases that invoke each feature and extracts components (code statements or methods) executed by each test case. For each feature, it then identifies the (1) indispensably involved components – executed by all test cases of the feature, (2) potentially involved components – executed by at least one test case of the feature, and (3) uniquely involved components – executed by at least one test case of the feature and not executed by any test case of the other features. It also extracts the set common components executed for all features.

Several later approaches extended this work by involving static code analysis, information retrieval, and other techniques to further prune the feature execution traces and improve the accuracy for feature detection, allowing them to operate on a single trace rather than on multiple traces corresponding to multiple features [24, 27]. FHistorian also relies on the presence of test cases to perform feature location. Yet, it detects features in change histories rather than in the "final" version of the program, thus assisting in tasks such as porting features and their histories across multiple branches in version control systems. It also uses information about change histories to improve the accuracy of feature location and does not require tests of multiple features in order to operate.

Feature Location in Version Histories. In CVSSearch [7], a feature is specified as a text query. The technique uses CVS diff to examine changes between subsequent commits and associates each line of code changed in a commit with its corresponding commit message. It then retrieves all lines that match the input query, i.e., either the line itself or its associated message containing at least one word from the query. Unlike CVSSearch, our technique does not rely on textual similarity but rather extracts executable feature implementations.

Feature Location for SPLs. Each of the existing feature location techniques can be used for detecting features in products of a product line by treating these products as singular independent entities. Yet, several techniques that consider commonalities and differences in SPL products have recently emerged [2, 3, 21–23, 37, 39, 42]. Most such techniques are based on intersecting code of multiple product variants in order to identify code fragments shared by variants with a particular feature. For example, Xue et al. [39] use information about version differencing to further improve the accuracy of information retrieval in multiple products. Linsbauer et al. [22, 23]

present a technique for deriving the traceability between features and code in product variants by matching code overlaps and feature overlaps. Moreover, this technique also identifies code that depends on the combination of features present in a product variant thus dealing with feature dependencies and interactions. While the above interaction-based techniques operate statically and are effective when a large number of product variants are available, our approach is dynamic and does not rely on the presence of a large set of variants to be effective. Moreover, it is also able to distinguish between features that always appear together in all product variants – a clear limitation of the intersection-based techniques.

Extracting Feature Models. Several approaches focus on extracting constraints between features of multiple variants [1, 6, 15, 26, 33, 35] or on building a desirable feature model when such constraints are given [4, 10, 36]. For example, Assunção et al. [6] extend their intersection-based feature location technique [22] with an approach to identify dependencies between features by looking at shared / exclusive code fragments. The above approaches mostly consider product and feature combinations, without inspecting semantic dependencies between code artifacts. Moreover, they rely on the availability of multiple product variants while FHistorian does not make such an assumption and is also able to identify dependencies between features of a single variant.

Nadi et al. [26] focus on inferring constraints between features implemented in C by statically identifying potential preprocessor, parser, linker, and type errors. Instead, our approach is dynamic and precisely identifies all runtime dependencies between the executed features.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented a dynamic feature location technique FHistorian. The technique works by analyzing version histories, taking into account feature release information and developer-committed tests demonstrating the new feature, to precisely extract feature-related changes. It also produces models representing runtime relationships between features. Our cases studies on multiple features of five real-world software projects show that FHistorian can locate features effectively.

In the future, we aim to combine FHistorian with information retrieval techniques for extracting expert feature knowledge from historical artifacts such as developer conversations, log messages, and documentation. With more complete feature information, our technique can produce significantly better feature models.

REFERENCES

[1] R. Al-msie'deen, M. Huchard, A. D. Seriai, C. Urtado, S. Vauttier, and A. Al-Khlifat. 2014. Concept Lattices: A Representation Space to Structure Software Variability. In *Proc. of Int. Conf. on Information and Communication Systems (ICICS'14)*. 1–6.

[2] Ra'Fat Al-Msie'deen, Abdelhak-Djamel Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. 2013. Mining Features from the Object-oriented Source Code of Software Variants by Combining Lexical and Structural Similarity. In *Proc. of Int. Conf. on Information Reuse & Integration (IRI'13)*. 586–593.

[3] Ra'Fat Al-Msie'deen, Abdelhak Seriai, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Hamzeh Eyal Salman. 2013. Feature Location in a Collection of Software Product Variants Using Formal Concept Analysis. In *Proc. of Int. Conf. on Software Reuse (ICSR'13)*, Vol. 7925. 302–307.

[4] Nele Andersen, Krzysztof Czarnecki, Steven She, and Andrzej Wąsowski. 2012. Efficient Synthesis of Feature Models. In *Proc. of Int. Software Product Line Conf. (SPLC'12)*. 106–115.

[5] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, Stefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. 2014. Flexible Product Line Engineering with a Virtual Platform. In *Proc. of Int. Conf. on Software Engineering (ICSE Companion 2014)*. 532–535.

[6] Wesley K.G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2015. Extracting Variability-Safe Feature Models from Source Code Dependencies in System Variants. In *Proc. of 2015 Annual Conf. on Genetic and Evolutionary Computation (GECCO'15)*. 1303–1310.

[7] Annie Chen, Eric Chou, Joshua Wong, Andrew Y. Yao, Qing Zhang, Shao Zhang, and Amir Michail. 2001. CVSSearch: Searching through Source Code using CVS Comments. In *Proc. of Int. Conf. on Software Maintenance (ICSM'01)*.

[8] Kunrong Chen and Václav Rajlich. 2000. Case Study of Feature Location Using Dependence Graph. In *Proc. of Int. Workshop on Program Comprehension (IWPC'00)*. 241–247.

[9] Commons-CSV 2017. The Apache Commons CSV Library. (2017). https://commons.apache.org/proper/commons-csv

[10] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *Proc. of Int. Software Product Line Conf. (SPLC'07)*. 23–34.

[11] 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.

[12] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR'13)*. 25–34.

[13] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. 2009. Empirical Evaluation of Hunk Metrics As Bug Predictors. In *Proc. of Int. Conf. on Software Process and Product Measurement (IWSM '09 /Mensura '09)*. 242–254.

[14] Git 2017. Git Version Control System. (2017). https://git-scm.com

[15] Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2011. Reverse Engineering Feature Models from Programs' Feature Sets. In *Proc. of Working Conf. on Reverse Engineering (WCRE'11)*. 308–312.

[16] JIRA 2017. JIRA Software. (2017). https://www.atlassian.com/software/jira

[17] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. 1998. FORM: A Feature-oriented Reuse Method with Domain-specific Reference Architectures. *Annals of Software Engineering* 5, 1 (1998), 143–168.

[18] Yi Li, Julia Rubin, and Marsha Chechik. 2015. Semantic Slicing of Software Version Histories. In *Proc. of 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'15)*. 686–696.

[19] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2016. Precise Semantic History Slicing through Dynamic Delta Refinement. In *Proc. of 31st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'16)*. 495–506.

[20] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. Semantic Slicing of Software Version Histories. *IEEE Trans. on Software Engineering* (2017).

[21] Lukas Linsbauer, Florian Angerer, Paul Grünbacher, Daniela Lettner, Herbert Prähofer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2014. Recovering Feature-to-Code Mappings in Mixed-Variability Software Systems. In *Proc. of Int. Conf. on Software Maintenance and Evolution (ICSME'14)*. 426–430.

[22] Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2013. Recovering Traceability Between Features and Code in Product Variants. In *Proc. of Int. Software Product Line Conf. (SPLC'13)*. 131–140.

[23] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2016. Variability Extraction and Modeling for Product Variants. *Software & Systems Modeling* (2016), 1–21.

[24] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. 2007. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proc. of Int. Conf. on Automated Software Engineering (ASE'07)*.

[25] Kivanç Muşlu, Luke Swart, Yuriy Brun, and Michael D. Ernst. 2015. Development History Granularity Transformations. In *Proc. of 30th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE'15)*. 697–702.

[26] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proc. of Int. Conf. on Software Engineering (ICSE'14)*.

[27] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. 2010. Using Data Fusion and Web Mining to Support Feature Location in Software. In *Proc. of Int. Conf. on Program Comprehension (ICPC'10)*. 14–23.

[28] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *Proc. of Int. Conf. on Software Engineering (ICSE'13)*. 1233–1236.

[29] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Conceptual Models, and Languages.*, Iris Reinhartz-Berger et al. (Ed.). Springer, 29–58.

[30] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: a Framework and Experience. In *Proc. of Int. Software Product Line Conf. (SPLC'13)*. 101–110.

[31] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned Product Variants: From Ad-Hoc to Managed Software Product Lines. *Journal on Software Tools for Technology Transfer* 17, 5 (2015), 627–646.

[32] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *Proc. of Int. Software Product Line Conf. (SPLC'12)*. 156–160.

[33] Uwe Ryssel, Joern Ploennigs, and Klaus Kabitzsch. 2011. Extraction of Feature Models from Formal Contexts. In *Proc. of Int. Conf. on Software Product Lines (SPLC) Workshop, Vol. 2*. 4:1–4:8.

[34] SemanticMerge 2016. The Diff and Merge Tool that Understands Your Code – SemanticMerge. (2016). https://www.semanticmerge.com

[35] Anas Shatnawi, Abdelhak Seriai, and Houari Sahraoui. 2014. Recovering Architectural Variability of a Family of Product Variants. In *Proc. of Int. Conf. on Software Reuse (ICSR'15)*. 17–33.

[36] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *Proc. of Int. Conf. on Software Engineering (ICSE'11)*. 461–470.

[37] Ryosuke Tsuchiya, Tadahisa Kato, Hironori Washizaki, Masumi Kawakami, Yoshiaki Fukazawa, and Kentaro Yoshimura. 2013. Recovering Traceability Links between Requirements and Source Code in the Same Series of Software Products. In *Proc. of Int. Software Product Line Conf. (SPLC'13)*. 121–130.

[38] Norman Wilde and Michael C. Scully. 1995. Software Reconnaissance: Mapping Program Features to Code. *J. of Software Maintenance* 7 (1995), 49–62. Issue 1.

[39] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *Proc. of Working Conf. on Reverse Engineering (WCRE'12)*. 145–154.

[40] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[41] Chenguang Zhu, Yi Li, Julia Rubin, and Marsha Chechik. 2017. A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories. In *Proc. of 14th Int. Conf. on Mining Software Repositories (MSR'17)*. 523–526.

[42] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature Identification from the Source Code of Product Variants. In *Proc. of European Conf. on Software Maintenance and Reengineering (CSMR'12)*. 417–422.