Give an Inch and Take a Mile? Effects of Adding Reliable Knowledge to Heuristic Feature Tracing

Sandra Greiner
University of Southern Denmark, DK
University of Bern, CH

Alexander Schultheiß
Paderborn University, DE

Paul Maximilian Bittner
Paderborn University, DE

Thomas Thüm
Paderborn University, DE

Timo Kehrer
University of Bern, CH

Abstract

Tracing features to software artifacts is a crucial yet challenging activity for developers of variability-intensive software projects. Developers can provide feature traces either proactively in a manual and rarely semi-automated way or recover them retroactively where automated approaches mainly rely on heuristics. While proactive tracing promises high reliability as developers know which features they realize when working on them, the task is cumbersome and without immediate benefit. Conversely, automated retroactive tracing offers high automation by employing heuristics but remains unreliable and dependent on the quality of the heuristic. To exploit the benefits of proactive and retroactive tracing while mitigating their drawbacks, this paper examines how providing a minimal seed of accurate feature traces proactively (give an inch) can boost the accuracy of automated, heuristic-based retroactive tracing (take a mile). We examine how comparison-based feature location, as one representative of retroactive feature tracing, can benefit from increasing amounts of proactively provided feature mappings. For retroactive comparison-based feature tracing, we find not only that increasing amounts of proactive information can boost the overall accuracy of the tracing but also that the number of variants available for comparison affects the effectiveness of the combined tracing. As a result, our work lays the foundations to optimize the accuracy of retroactive feature tracing techniques with pinpointed proactive knowledge exploitation.

Keywords

software variability, software evolution, software product lines

ACM Reference Format:
Sandra Greiner, Alexander Schultheiß, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. 2024. Give an Inch and Take a Mile? Effects of Adding Reliable Knowledge to Heuristic Feature Tracing . In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 2–6, 2024, Dommeldange, Luxembourg*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3646548.3672593

1 Introduction

Tracing the artifacts implementing a feature is essential to productively maintain variability-intensive, software systems [11, 54]. A *feature trace* identifies all artifacts in a software project which realize the respective features. The knowledge encoded in the feature trace allows not only deriving a variant of the system, and analyzing and optimizing the implementation, but also assessing the impact of changing features in terms of affected elements. Thus, the trace stores crucial insights when maintaining variability-intensive software systems [49].

To trace features, it is essential to detect the locations implementing them in the software not only quickly and exhaustively, but also reliably [29, 53, 58]. Feature traces can be created proactively, while implementing a feature, or, retroactively, after some time to extract the information on demand [22]. Despite its importance, proactive feature tracing burdens developers with manual and rarely semi-automated documentation effort without immediate benefit during development [8, 22]. They need to actively associate the artifacts with the feature(s) they are working on. Upfront investment [10, 32], missing tool support [56], lack of flexibility [4, 48, 56], and the necessity to adapt existing workflows (e.g., by developing features in branches) [4] hinder developers to trace features proactively. Still, active developers represent a highly reliable resource for feature traces [25]; a wasted opportunity if left disregarded.

When developers require trace information *in retrospect*, for instance to fix an issue or to assess the impact of a change, they either have to recover feature traces manually or may employ automated *retroactive* feature tracing [1, 2, 13, 15, 23, 25, 33, 37, 38, 47, 60–63], techniques which are based on heuristics. If possible at all, manual retroactive feature tracing proves laborious, error-prone and, thus, of low reliability, particularly, if performed by developers who did not implement the source code originally [4, 14, 27, 30, 48, 50, 56]. Pure automated retroactive feature tracing minimizes the manual labor but at the cost of accuracy due to necessary heuristics.

While proactive traces provide a reliable resource to inform retroactive feature tracing due to their manual, on-time nature, research has not examined the entire potential of informing retroactive heuristics with proactively collected knowledge. Minimal seeds of proactively provided feature mappings (e.g., as embedded annotations) are known to benefit tasks, such as migrating single software projects into a product line [25], but research has not investigated *how* exactly proactive feature traces can benefit retroactive feature tracing; knowledge which may boost the accuracy of several retroactive methods. As a consequence, in this paper, we examine how proactive feature traces can enrich retroactive feature tracing in a controlled experiment. Particularly, we explore the research question: *How can proactive feature traces benefit the accuracy of retroactive feature traces?*

To answer this question, we conduct, to the best of our knowledge, the first controlled experiment where we study the effects of an increasing number of proactive feature traces in a retroactive feature tracing technique. Specifically, we enrich a comparison-based feature location technique [39], as one representative of *feature location* techniques [6, 13, 38, 38–40, 45] that identify features in software variants. Particularly, we examine how minimal amounts of proactive feature traces increase the accuracy of the retroactive feature tracing and how the number of variants available for comparison affects the effectiveness of combining proactive and retroactive feature tracing.

We use open-source and widely used highly configurable systems from different domains, such as the ArgoUML benchmark [3, 12, 36, 39, 42, 43] or BusyBox[1] [26, 59], as subjects. We vary the amount of available proactive feature traces to observe their effect on precision and recall, as well as the number of available variants which are essential in the comparison-based feature location. In a nutshell, our results (cf. Sec. 5.2) demonstrate a significant boosting effect for the accuracy of retroactive tracing when adding only 5% proactively provided feature traces. Furthermore, we observe that this effect is particularly high when only few variants are available to be compared. The higher the number of variants (7 may be sufficient), the less effect is added by the proactive mappings.

All in all, our work, which is available online [18], contributes the first experimentally shown understanding that only a small amount of proactive traces (give an inch) boosts the overall accuracy (take a mile) of automated retroactive feature tracing and pays off, particularly, when few variants are available. This knowledge gives rise to examine how proactive traces affect further retroactive feature tracing techniques with the eventual goal to optimally exploit proactive feature traces in retroactive feature tracing. While minimizing manual efforts, our results lay the grounds to maximize the accuracy of automatically computed feature traces.

2 Motivating Example

This section presents an example that motivates the interaction between retroactive automated feature tracing and proactively provided trace information. We illustrate it by using comparison-based feature tracing [33, 39], a technique which is particularly helpful in migrating single software projects into a software platform.

Let us assume developers Alice, Bob, and Charlie implement three variants of a Graph algorithm, as depicted in the upper part of Fig. 1. The figure demonstrates an excerpt of the Java interface Graph as implemented in the respective three software variants V1-V3. The three variants have the common features *Graph* (G) and *Edge* (E), but differ in features *Weighted* (*W*), *Directed* (*D*), and *Colored* (*C*), respectively. The features are reflected by respective methods in the interface Graph. We refer to each line of Java source code as an *implementation artifact*, similar to related work [9, 24, 33]. Some artifacts are annotated with embedded *feature mappings*. For instance, Lines 2 and 5 of V1 of the source code in the top left corner of Fig. 1, annotate the features G and E, respectively. Developers may provide embedded feature mappings when implementing their features to distinguish the following line of source code from base code which is relevant in all variants. A plethora of real-world projects uses, for instance, preprocessor statements for representing feature mappings [22, 31].

After a while, Charlie requires a Graph with weighted and directed edges. By inspecting the three variants superficially, she does not see which parts to include in her new variant. For instance, it is not clear at first glance that the method subGraph() implements the features *Directed* and *Colored*. A *feature trace* can help her not only in identifying the necessary implementation artifacts but also in estimating the complexity and, thus, the cost of implementing the variant with the corresponding features.

Despite the few available manually provided feature mappings in V1-V3, it is challenging to locate all artifacts that implement a specific feature in the software variants unambiguously. The manually provided traces help to identify a feature but are incomplete. For instance, in this example, it is unclear at first sight which artifacts of V1 implement the feature *Weighted*.

Still, by comparing the variants and respective feature configurations, the information where each feature is implemented can be recovered retroactively. Charlie can try to manually identify the artifacts implementing the features in variant V1 and V2 but may introduce errors, such as missing an artifact or interpreting it the wrong way. Automated feature tracing may help but relies on heuristics which may lack reliability, because eventually it must guess potentially lost knowledge. For instance, by only considering the configurations and source code snippets, retroactive feature tracing cannot decide whether the method nodes() in each variant implements the features *Graph* and *Edge* or also the features *Weighted* or *Directed* or *Colored* [33, 39]. Thus, further reliable knowledge, such as feature mappings proactively provided by the developers, may additionally inform the automatic heuristic-based retroactive feature tracing.

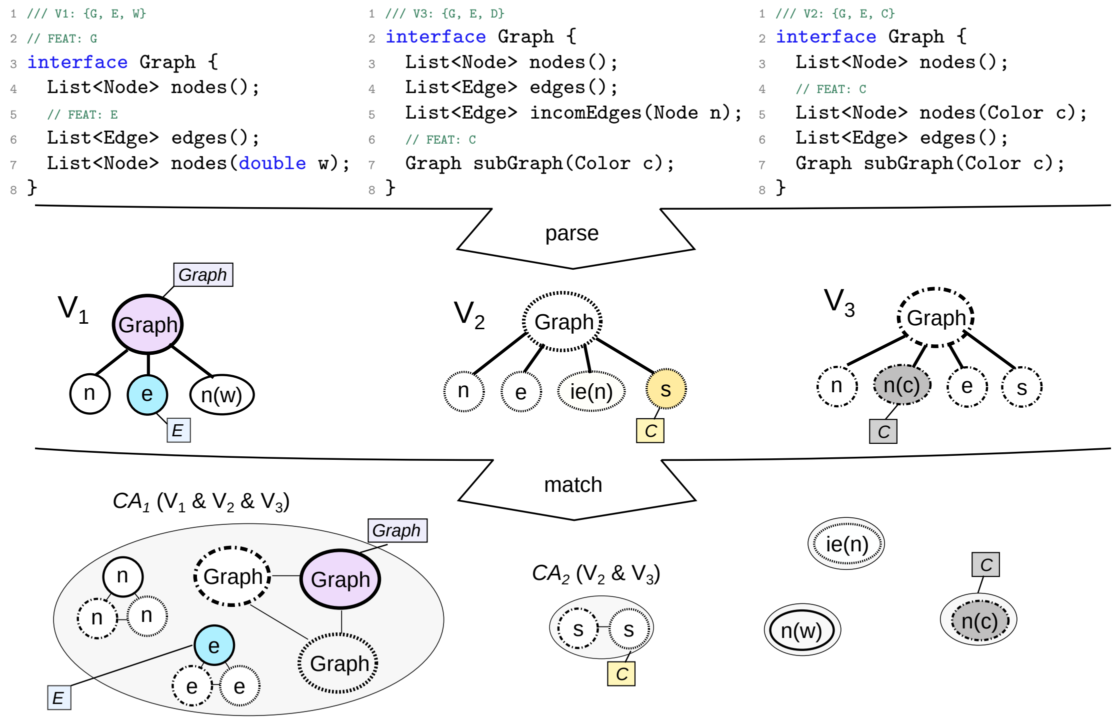
3 Background

Before introducing our experiment, this section provides background information on feature tracing techniques highlighting their pros and cons. Sec. 3.1 presents proactive feature tracing methods, followed by explaining retroactive feature tracing in Sec. 3.2. Sec. 3.3 summarizes the missing potentials of both methods in the current state of the art.

3.1 Proactive Feature Tracing

Proactive feature tracing aims at preserving feature trace knowledge while it is available in developer's memory and before it is needed [25]. Typically, proactive traces are documented manually in terms of comments, as exemplified in Fig. 1, or more structured annotations [8, 22, 57]. Informal documentation, such as comments [22], form *implicit* feature traces, useful for developers but hardly useful for automated reasoning and tools. Implicit feature traces can be given, for example as preprocessor annotations, as common in C/C++ projects (e.g., the Linux kernel), or as fine-grained feature-wise commits with respective commit messages [8, 16, 52, 55]. Conversely, formal languages or dedicated mechanisms enable *explicit* feature traces. Besides documenting

[1] https://busybox.net/about.html



```
/// V1: {G, E, W}
// FEAT: G
interface Graph {
  List<Node> nodes();
  // FEAT: E
  List<Edge> edges();
  List<Node> nodes(double w);
}
```

```
/// V3: {G, E, D}
interface Graph {
  List<Node> nodes();
  List<Edge> edges();
  List<Edge> incomEdges(Node n);
  // FEAT: C
  Graph subGraph(Color c);
}
```

```
/// V2: {G, E, C}
interface Graph {
  List<Node> nodes();
  // FEAT: C
  List<Node> nodes(Color c);
  List<Edge> edges();
  Graph subGraph(Color c);
}
```

Figure 1: Representative for retroactive feature tracing: Comparison-based feature location for three variants V_1, V_2, and V_3.

the source code, such formally stated traces can serve as a mechanism to activate or deactivate certain features.

On its downside, proactive tracing requires changing the developers' workflows by adding additional duties without immediate benefit. Thus, developers tend to neglect such manual documentation. To support developers in documenting traces on time, semi-automated techniques, such as feature trace recording [8, 20], tracing requirements [21, 34], or using AI-supported nudging [44] allow for systematic and semi-automated proactive feature tracing. For instance, in feature trace recording, feature traces are computed automatically when developers annotate their commits with information about the changed features.

3.2 Retroactive Feature Tracing

Retroactive feature tracing, also known as *feature location* or *variability mining*, aims at automatically deriving explicit feature traces. The key idea is to perform (semi-)automated analyses of the source code to recover or recommend reasonable feature traces *after* the source code was edited, hence, retroactively. Possible sources of information to recover feature traces are static [1, 25, 62] or dynamic analyses [15, 40, 60, 61], comparisons of software variants [33, 37, 46, 63], mining software repositories [2], or combinations thereof [23, 38]. Hybrid methods [28, 38] that combine static with dynamic feature tracing may increase the overall accuracy but still require a large number of variants or data to be processed.

Critically, retroactive tracing is a heuristic process because feature information has to be recovered from incomplete or imperfect information. While some retroactive tracing techniques, such as variability-mining based on type systems [25], may yield exact results, they can only cover small parts of the code base at a time. Naturally, heuristics implemented in retroactive tracing are subject to various design decisions and parameters, influencing the success of recovering traces. For instance, comparison-based feature tracing relies on the number of input variants, their configurations, and a matching operator. In particular, a set of variants with their respective configurations are compared to determine the implementation artifacts shared by certain variants. If too few variants are available, only few features can be detected at a coarse granularity.

As an example, for the three variants depicted in Fig. 1, it is impossible to distinguish artifacts implementing *Graph* from artifacts implementing *Edge* because they always appear together. Additionally, it is not possible to distinguish the contained artifacts (related with *Graph* and *Edge*) from artifacts which appear in every variant (i.e., which implement *core* features). In concrete, a subset of artifacts that exists in all three variants may be mapped onto the feature combinations: *Graph*, *Edge*, *Graph* $\wedge$ *Edge*, *Graph* $\vee$ *Edge*, or *Weighted* $\vee$ *Directed* $\vee$ *Colored*, or another combination thereof. In essence, heuristic-based feature tracing cannot determine a unique feature mapping but only sets of possible and impossible features that an artifact implements. For tasks, such as deriving a new variant, the sets of *possible* feature mappings do not suffice because it requires the concrete feature mappings for each artifact to be known. Eventually, any retroactive approach has to decide on a concrete feature mapping eventually.

3.3 Consequence

One of the main problems of proactive feature tracing lies in its manual nature which causes developers to neglect it. If neglected, it is hardly possible to reliably annotate an entire code base retroactively. Knowledge about feature mappings of old source code and the respective developers that implemented it may not be available anymore. On top, non-code artifacts, such as requirement documents or code documentation, may be outdated, such that they do not offer a reliable source of feature knowledge. As the code has to be analyzed and understood in retrospect, manually creating feature traces for old code may be as cumbersome as writing the code anew. Retroactive feature tracing promises to automate the process but lacks reliability due to necessarily employed heuristics. The accuracy depends on various factors and an understanding of how to benefit from proactive knowledge and exploit it in retroactive tracing is missing so far; crucial knowledge to optimize the accuracy and reliability of automated retroactive feature tracing and to minimize the manual burden for developers.

4 Boosted Retroactive Feature Tracing

As a consequence of the missing accuracy of automated retroactive feature tracing techniques, we examine how the reliable information of proactively provided feature traces can increase the accuracy of retroactive methods. First, we present the concept of integrating proactive feature traces into a retroactive feature tracing technique, which we refer to as *boosted algorithm*. Second, we exemplify the method with a comparison-based feature location technique and illustrate its original and adapted behavior; i.e., without and with regarding proactive feature traces.

Conceptual Approach. Retroactive feature tracing computes feature mappings (i.e., a Boolean expression over a set of possible features for a software artifact) by using either static or dynamic analysis [6, 12]. In our case study, we *extend* a static comparison-based retroactive feature tracing algorithm [39]; as one representative of feature location techniques which eventually compute feature traces. The key idea is to perform the original algorithm and to integrate proactive knowledge whenever possible. The chosen algorithm [39] compares software variants to identify sets of possible and impossible features for each of their artifacts, based on the commonalities and differences between variants. We boost this comparison-based algorithm with the *propagation* of known mappings among matched artifacts in the variants.

Alg. 1 demonstrates the resulting boosted algorithm. It receives sets of variants parsed in artifacts trees A (e.g., the trees V1-V3 in Fig. 1) and a proactively provided feature trace FT_p. Based on this information, it computes a complete feature trace FT. The original algorithm computes only *matches* in form of co-occurring artifacts (cf., l. 6) and determines the feature trace retroactively (cf., l. 8). Our extension, highlighted in red color, considers the additional input FT_p to propagate its feature mappings among the co-occurring artifacts before computing the feature trace heuristically. To illustrate the boosted comparison-based feature tracing, the following two paragraphs apply the original and the boosted algorithm to our motivating example (cf., Sec. 2).

Original Algorithm. Let us apply the original, purely retroactive, feature tracing algorithm in a simplified form to the variants of

Algorithm 1 Boosted comparison-based feature tracing.

```
1: PROCEDURE traceFeatureBoosted($A$, $FT_p$, $FT$)
2: in $A$ {set of artifact trees of distinct variants}
3: in $FT_p$ {set of proactively provided mappings}
4: out $FT$ {set of complete feature mappings}
5:
6: var $matches \leftarrow$ computeMatches($A$)
7: $FT' \leftarrow$ propagateExisting($FT_p$, $matches$)
8: $FT \leftarrow$ computeRetroactively($FT'$, $matches$, $A$)
```

the example in Fig. 1. First, the algorithm parses the source code of the three input variants into *artifact trees*. For the sake of simplicity, we employ a simple parser that transforms the class and its methods into a tree of individual artifact nodes. The algorithm compares the resulting trees and computes matching artifacts among the variants. For the sake of the example, we assume a name-based matching. Next, the algorithm aggregates artifacts into sets of *co-occurring artifacts (CA)*. Artifacts are co-occurring if they appear in exactly the same subset of variants: if an artifact is part of a variant, all its co-occurring artifacts must also be part of that variant. In Fig. 1, the artifacts defining the class Graph (G) and the methods nodes (n) and edges (e) reside in one set of co-occurring artifacts CA_1, the two methods subGraph() (s) in another set CA_2. The remaining artifacts form individual sets.

Lastly, the algorithm computes feature mappings for co-occurring artifacts. By building intersections and unions of the configurations of the variants involved in each artifact set, it computes sets of possible and impossible features. Based on the feature sets, it needs to determine a feature mapping. For instance, the set of possible features for CA_1 the algorithm includes *Graph*, *Edge*, *Weighted*, *Directed*, and *Colored*. At minimum, the algorithm may combine the features *Graph* and *Edge* as they appear in each variant[2]. If a set comprises more than one feature, the algorithm needs to select a Boolean expression over them heuristically without guarantee of choosing the *accurate* one for the artifact. Thus, the overall accuracy of the algorithm depends on the correctness of the sets of possible features as well as on determining an accurate mapping. The originally published algorithm [39] assumes the largest conjunction in terms of the number of common features.

Boosted Algorithm. Our extension of this comparison-based feature location algorithm integrates proactively provided feature mappings. First, it iterates the proactive feature trace to annotate the nodes of the artifact trees. For instance, the class Graph in variant V_1, the method e of V_1, the method s() of V_2, and the method n(c) of V_3 are proactively annotated with feature mappings. In Fig. 1, we attach these feature mappings to the respective artifact nodes.

The boosted algorithm exploits the proactively provided feature mappings which are added to the nodes to *propagate* them to matched artifacts without annotation, in two steps. First, during the matching, it propagates them to the matched node if there is no mapping yet. Second, it propagates a mapping if it is not contradicting. For instance, in CA_2 the mapping is propagated to the method of V_3 whereas in CA_1 the mapping E may be provided to the matched nodes but not to the remaining artifacts in CA_2 as the mapping Graph is contradicting. In general, if an expression over

[2] *Weighted*, *Directed*, or *Colored*, may be involved, too.

features f is mapped onto an artifact a_i in a variant v_i, f can be propagated to any artifact a_k in variants v_k, $k \neq i$, which matches a_i. In this way, proactive knowledge can be exploited to reduce the number of heuristically determined feature traces.

5 Controlled Experiment

To answer our overall research question of how proactive feature traces influence the accuracy of retroactive tracing and to guide our experiment, we define two central research questions:

RQ1 *How does the **number of proactively collected feature traces** affect retroactive feature trace recovery?*

RQ2 *How does the **number of available variants** affect the effectiveness of combining proactive and retroactive feature tracing?*

With RQ1, we evaluate the effectiveness of combining proactive and retroactive feature tracing. We add increasing percentages of proactively collected traces to the retroactive tracing and evaluate how the accuracy of retroactive feature tracing changes depending on this percentage in terms of precision and recall. In the worst case, the retroactive feature tracing cannot utilize the proactive traces; i.e., it does not improve the accuracy of the tracing. In an ideal case, the retroactive tracing can leverage proactive traces and thereby significantly increase its accuracy.

As previous research has shown, the number of available variants significantly influences comparison-based feature location [39, 40]: The more variants available, the better the tracing accuracy. If a high number of variants is available for comparison, the effect of adding proactive feature traces may decrease. In RQ2, we explore how the number of compared variants affects the effectiveness of combining proactive and retroactive tracing.

5.1 Study Setup

We now describe the setup of our experiment to study the effects of proactive feature tracing employed in retroactive feature tracing. We implemented a prototype combining the retroactive, comparison-based feature location with proactive feature mappings and examined it in 5 subject systems. The implementation of the prototype, evaluation, and all results are available online [18].

Subject Systems. Table 1 presents our experimental subjects. We selected five (highly-)configurable systems from different domains, of varying size (KLOC), and of varying number of variables (features). For instance, ArgoUML-SPL[3] comprises eight optional features and 120 KLOC of Java source code. We chose ArgoUML-SPL because it serves as benchmark for feature location techniques [3, 12, 36, 39, 42, 43]. OpenVPN, Vim, BusyBox, and Marlin are open-source highly-configurable systems, primarily implemented in C and C++. These four subjects all comprise large code bases, but differ greatly in variability in terms of the number of unique variables in the conditions of their variation points (i.e., conditional C preprocessor annotations), ranging from a few hundred to about 10,000 variables.

We deliberately use the term *variable* instead of feature, because we refer to the number of unique variables that occur in C preprocessor annotations. Not all of these variables may represent features in the sense of a product line [5]. We consider variables for practical reasons: Extracting "real" features would require analyzing the problem space which is a challenging process with many uncertainties and would require considerable technical investment to cover all experimental subjects. Still, *all variables* directly influence the static variability of the code base, and they constitute a superset of the features implemented in the solution space.

Table 1: Subject systems.

Name	#Variables	KLOC	Domain
ArgoUML	8	307	Modeling Tool
OpenVPN	475	125	Security and Networking
Vim	1,536	498	Text Editor
BusyBox	1,959	255	Embedded Systems
Marlin	10,529	504	3D Printer Firmware

Experiment Design. In our experiment, the available proactive features and available variants represent the independent variables which affect the dependent variable of accuracy (which we take in the following as synonym for precision and recall). To answer **RQ1**, we simulate variants with increasing numbers of proactively provided feature mappings. We increase the percentage of known mappings in steps of 5 % from 0 % (i.e., no proactive tracing) to 25 % and distribute the respective percentage of mappings uniformly among each sampled variant in each step.

To answer **RQ2**, we consider 3, 5, and 7 variants for each of the percentages mentioned above. On the one hand, these numbers may represent typical clone-and-own scenarios. On the other hand, we observed diminishing effects for 7 variants so that we would not expect relevant effects of proactive traces for a higher number of compared variants. To reduce the bias of the random aspects of our experiment, each combination of percentage and variant count is repeated 30 times and the results are averaged[4].

Tooling. To conduct our experiment, we employed the benchmark generation tool VEVOS [51] to analyze the variability of each subject system. VEVOS analyzes variability with the help of DiffDetective [7], a library which constructs variability-aware differences [9] that distinguish changes to preprocessor annotations from changes to source code. By using the resulting variability-aware diffs, VEVOS incrementally determines the feature mappings and presence conditions of all lines of source code across a version history. Moreover, VEVOS determines the unique variables of the extracted presence conditions, which we treat as the optional and independent features of the subjects. We analyzed the latest revision per subject system at the moment of conducting our study.

Based on the extracted variables, VEVOS allows us to uniformly sample variant configurations and generate the source code of the variants. Unfortunately, in preliminary experiments, we found that comparison-based feature tracing faces severe scalability issues if many features are involved. The required runtime and memory grows exponentially with the number of features. Due to this limitation, we are forced to sample a random subset of 10 features for each variant sample; the configurations of the variants are sampled from these 10 features. Furthermore, we limit the minimum number

[3] https://github.com/marcusvnac/argouml-spl

[4] Due to runtime and space limitations, it was not feasible to execute a higher number of repetitions.

of features of each variant to 5, so that there is a higher chance that variants have at least some features in common. In total, our experiments consider the traceability of up to a few hundred unique features per subject in total. Features that are not part of the sample are deselected and their source code is excluded from the variants.

To assess the accuracy of the computed feature traces, we compare the feature trace in form of feature mappings to the source code with the ground truth provided by VEVOS for each variant. Furthermore, we use the ground truth to simulate the proactive feature traces by randomly sampling partial traces as known traces. Specifically, we randomly select a desired percentage of code lines that implement a feature (or feature interaction) and treat them as proactively traced to this feature. For example, if exactly one feature A is implemented by 100 lines of code, trace sampling of 5% will randomly select 5 of these lines and provide their mapping as proactive trace to inform the retroactive tracing.

Measurement. We measure the accuracy of the computed mappings as the agreement between a computed feature mapping and the ground truth mapping: Given a variant's configuration, both feature mappings have to agree on whether to include the respective artifact in the variant. If they agree, the computed mapping is considered correct. In this way, we determine the set of true and false positives and negatives, respectively, which allow us to compute accuracy, precision, recall and F1-Score. True positives are computed mappings which include the artifact in a variant if the ground truth also includes it. True negatives correctly exclude an artifact from a variant. False positives are computed mappings which include the traced artifact in a variant while the ground truth excludes them and vice versa for false negatives.

5.2 Results

Fig. 2 and Fig. 3 depict the resulting precision and recall of conducting our experiment with each subject system[5]. Each row shows the results for a specific subject system and each column shows the results for a specific number of variants, 3 left, 5 center, and 7 right. Each plot presents the precision (Fig. 2) or recall (Fig. 3) for increasing percentages of proactive traces, from 0% through 25%. Please note, that each leftmost box plot depicts the accuracy of the pure retroactive tracing without additional proactive knowledge.

*5.2.1 RQ1: How does the **number of proactively collected feature traces** affect retroactive feature trace recovery?* To answer RQ1, we inspect the curves outlined by the box plots in the individual sub-figures of Fig. 2 and Fig. 3; we focus on one column at a time to fixate the number of variants.

In Fig. 2, we observe overall high tracing precision for all subject systems. For ArgoUML, the precision is close to the maximum of 1 and can barely be improved even by considering proactive traces. For the remaining subjects, the precision shows higher deviation, but overall is still high (mainly >0.8, when adding 5% of proactive traces). In Fig. 3, we can observe similar effects for the recall across all subjects, although the recall is more volatile than the precision, as visible through the wider deviation in the box plots.

Both figures highlight that proactive traces visibly boost retroactive feature tracing. The observed boosting effect is most noticeable when adding the first few percent of proactively traced features and appears to diminish with an increasing number of proactively traced features. If 5% of traces have been traced proactively for 3 variants, the precision and recall of boosted retroactive tracing improves by 10 to 20 percent on average, depending on the subject system. Conversely, little improvement in precision and recall is achieved when comparing the precision and recall of boosting with 20% and 25% proactive traces.

Only small amounts of proactively collected traces suffice to show a visible boosting effect, improving the precision and recall of retroactive tracing by 10 to 20 percentage points. This effect diminishes with increasing amounts of proactive tracing.

*5.2.2 RQ2: How does the **number of available variants** affect the effectiveness of combining proactive and retroactive feature tracing?* To answer RQ2, we compare the columns in Fig. 2 and Fig. 3 to draw conclusions from increasing numbers of variants.

Fig. 2 and Fig. 3 confirm that the number of variants affects the precision and recall of comparison-based feature tracing. We observe that the deviation in precision and recall decreases with an increasing number of variants. We only observe slight differences in the average precision with an increasing number of variants. On the other hand, the average recall is affected by an increasing number of variants, showing improvements of 10 to 20 percent for two additional variants depending on the subject system. More interestingly, increasing the number of variants also positively affects the boosted tracing with proactively collected traces. The deviation of precision and recall decreases if proactive traces are available for a higher number of variants. This effect is especially strong for the recall of OpenVPN. Without proactive traces, the recall may even drop to 0, but 5% of proactive traces for 7 variants boost the minimum observed recall to about 0.58.

We observe a clear boosting effect if proactive traces are available, regardless of the number of variants. More variants enable higher tracing accuracy and a more reliable boosting effect.

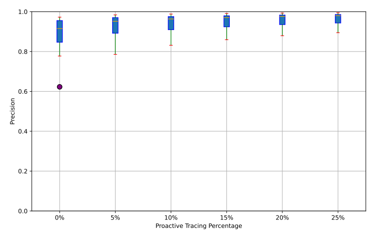
6 Discussion

Based on our results, this section discusses observations that we made, related work, and further potentials of adding proactive traces to retroactive tracing.

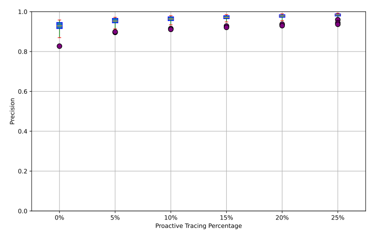
Observations. Our results provide valuable insights over previously reported results for comparison-based feature location and retroactive feature tracing methods in general.

First, by considering highly-configurable subject systems beyond ArgoUML, we observed that the precision of retroactive tracing is heavily influenced by the complexity of the variability in terms of feature interactions and feature negations. The precision of comparison-based tracing is almost perfect, if the code implementing a specific feature can be traced to a specific subset of features (e.g., $\{A\}$, or $\{A, B, C\}$). This is the case for ArgoUML which only involves "positive" annotations for implementation artifacts; a likely result from being re-engineered from software variants where

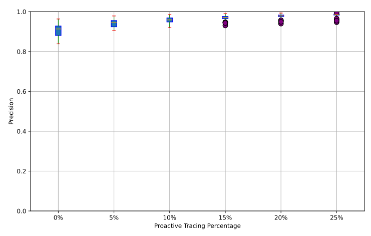
[5]The plots for accuracy and F1-score are available online: https://github.com/VariantSync/trace-boosting-eval/tree/main/reported-results/plots



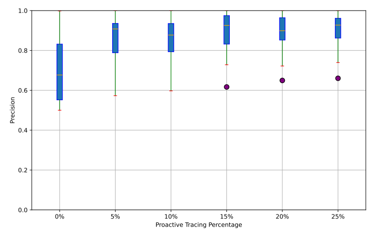
(a) ArgoUML 3 Variants



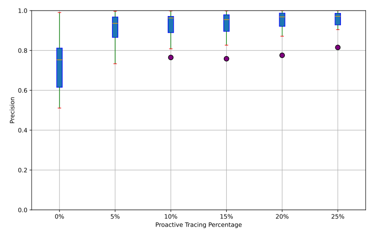
(b) ArgoUML 5 Variants



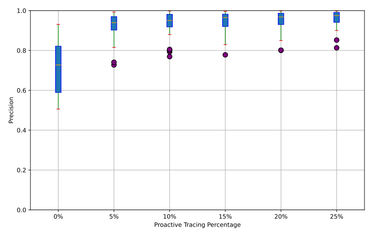
(c) ArgoUML 7 Variants



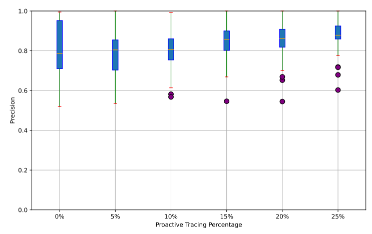
(d) OpenVPN 3 Variants



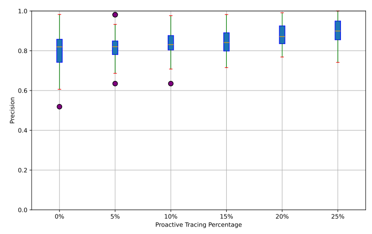
(e) OpenVPN 5 Variants



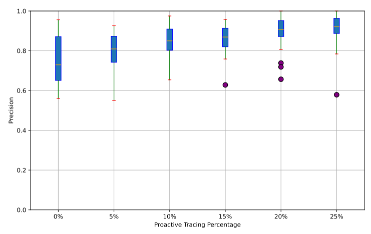
(f) OpenVPN 7 Variants



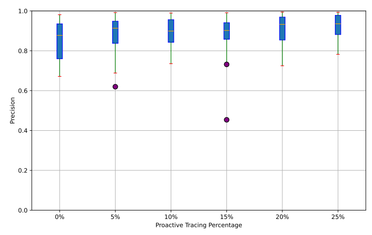
(g) Vim 3 Variants



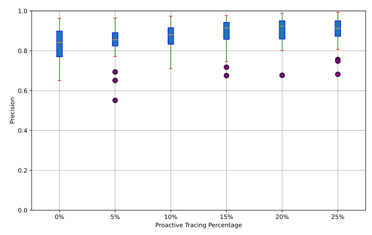
(h) Vim 5 Variants



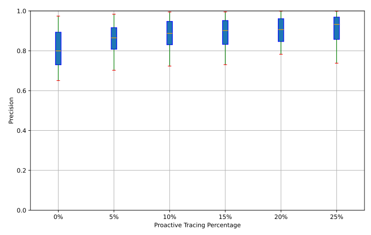
(i) Vim 7 Variants



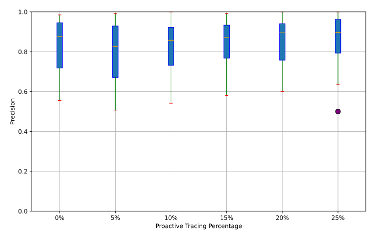
(j) BusyBox 3 Variants



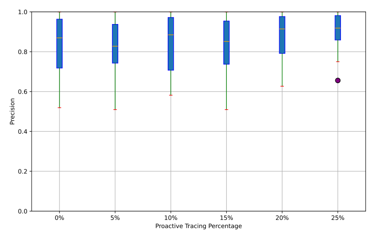
(k) BusyBox 5 Variants



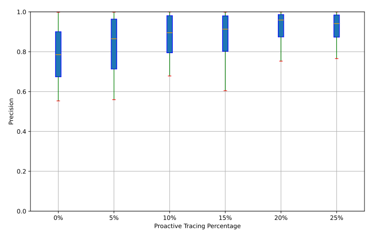
(l) BusyBox 7 Variants



(m) Marlin 3 Variants

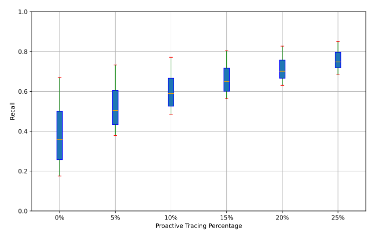


(n) Marlin 5 Variants

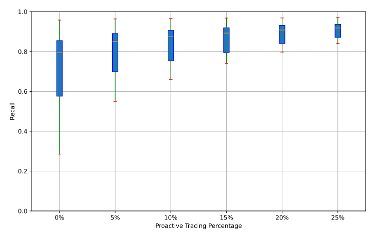


(o) Marlin 7 Variants

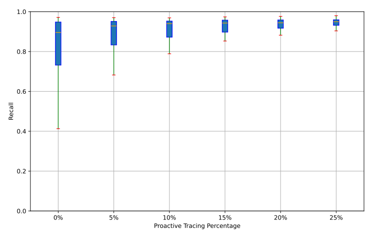
Figure 2: *Precision* of each examined subject for 3, 5, and 7 sampled variants (from left to right).



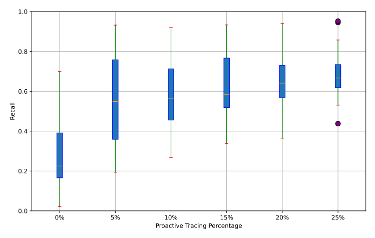
(a) ArgoUML 3 Variants



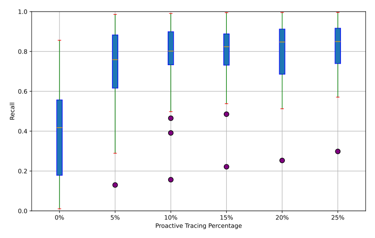
(b) ArgoUML 5 Variants



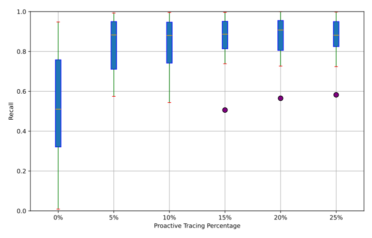
(c) ArgoUML 7 Variants



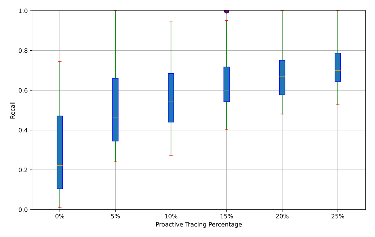
(d) OpenVPN 3 Variants



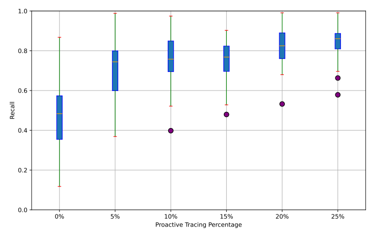
(e) OpenVPN 5 Variants



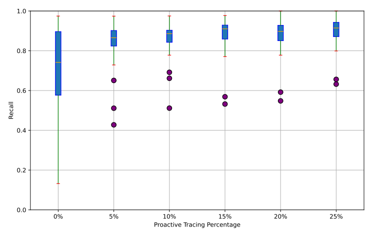
(f) OpenVPN 7 Variants



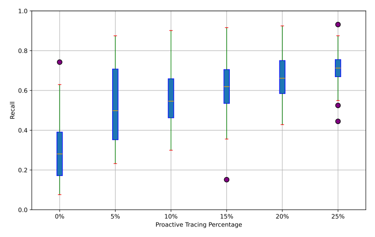
(g) Vim 3 Variants



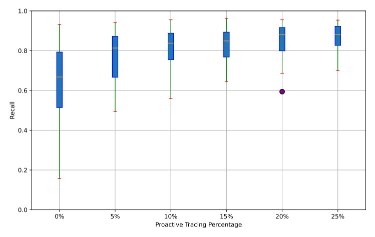
(h) Vim 5 Variants



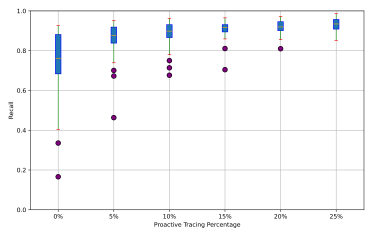
(i) Vim 7 Variants



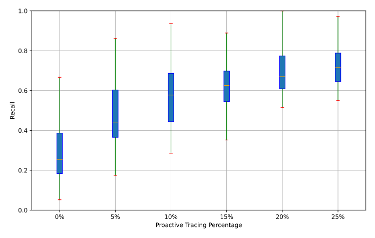
(j) BusyBox 3 Variants



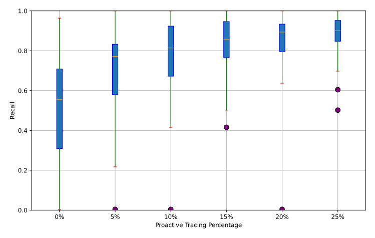
(k) BusyBox 5 Variants



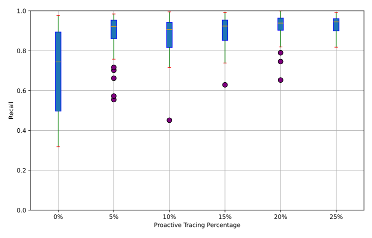
(l) BusyBox 7 Variants



(m) Marlin 3 Variants



(n) Marlin 5 Variants



(o) Marlin 7 Variants

Figure 3: *Recall* of each examined subject for 3, 5, and 7 sampled variants (from left to right).

functionality was added but not removed. However, comparison-based tracing struggles to determine that code may be required if a specific feature is not present. In subject systems, such as BusyBox or Marlin, several implementation artifacts may be removed from the code base for certain configurations through feature mappings (i.e., by stating "not FEAT E" instead of including the source code optionally "FEAT E"). Typically, comparison-based feature tracing cannot recognize these "negative features", resulting in reduced precision. In our experiment, negated features and complex feature expressions may be provided as proactive traces. Propagating those clearly improves precision.

Second, by considering randomly sampled variants in our experiment, our results demonstrate that the selection of variants influences the tracing accuracy considerably, as can be seen by the deviation in the results. The alignment of the variants (i.e., their commonalities and differences) influences the results strongly because comparison-based feature tracing slices the code into artifact subsets depending on the variants in which the code was found. If the slicing cannot distinguish features, the tracing accuracy will suffer. For example, comparison-based feature tracing will be highly accurate if each variant implements exactly one of the traced features. Vice versa, it will be very inaccurate if one variant implements almost all features, or if there is a large number of features implemented by all variants, because those features can then no longer be distinguished from one another.

Third, our results indicate that it is not necessary to collect a high number of proactive traces to benefit from boosted feature tracing. In contrast, proactive tracing pays off immediately. The diminishing boosting effects for higher amounts of proactive traces (<10%), might be caused by proactive traces overlapping in our experiment. Different proactive traces can yield the same boosting effect by propagating the same trace information to similar artifacts. In light of these results, future research may investigate how the distribution of proactively collected traces affects the boosting. This might yield insights into the most effective strategies for proactive tracing with boosted tracing in mind.

Lastly, the results indicate that the number of variants may influence the accuracy more than the amount of proactive traces. The more distinct information, as potentially available in diverse variants, the higher the accuracy without adding proactive knowledge. Still, new variants have to be created in the first place. Only adding a new feature to a copied variant does not provide a boosting effect as it does not provide new feature information about the existing variants. A new variant will only increase accuracy of comparison-based tracing, if it implements a different subset of features than implemented by the remaining variants. Creating such a variant requires a feature trace so that distinct features can be removed from the copied variant. Thus, creating new variants may only improve comparison-based tracing, if at least partial feature traces are available.

Related Work. Hybrid feature tracing techniques [12, 38, 40] combine static and dynamic retroactive feature tracing to achieve higher accuracy. Despite their combinations, all of these approaches still rely *only on retroactive* heuristic-based feature tracing [38]. As stated in the literature [22] and stressed in our experimental results, pure retroactive feature tracing lacks reliability. To gain certainty, developers still have to check the resulting feature traces manually.

Conversely, boosted feature tracing shall exploit minimal upfront investments, ideally provided in everyday workflows, to automatically compute accurate feature traces retroactively for entire projects. In fact, this effect was observed in variability mining yet without systematically exploring its disruptive potentials [25]. The authors do not examine why and which proactive seeds caused the boosting effect, an understanding which we plan to gain for feature tracing to be adopted in practice.

Potentials of Proactive Traces. In our case study, we propagate proactive feature traces while integrating a new variant and inside a set of co-occurring artifacts as computed by the retroactive feature location algorithm and explained in the example depicted in Fig. 1. While proactive feature mappings represent a highly reliable resource, we need heuristics to integrate them. In our experimental subjects, frequently not only one node but several of them are annotated (potentially differently) in the same set of co-occurring artifacts. This leaves room to employ different heuristics again. We decided to only regard and propagate the mapping to all nodes in the set if the available proactive mappings were exactly the same. Further possibilities to handle distinct proactive feature traces in a co-occurring artifact set include at minimum

(1) using only the first found manual mapping and propagating it to all remaining artifact nodes
(2) in case of contradictions,
    (a) splitting the set of co-occurring sets into those containing only the same proactive mapping and propagating them.
    (b) combine them in a logical OR-expression and propagate them.
    (c) enriching the mappings with quantities and taking the most frequent one.
    (d) enriching the mappings with confidential values to address uncertainties.

The first alternative is computationally cheaper but at the cost and risk of overwriting different already provided mappings. The second alternatives particularly deal with how to handle conflicting mappings. Such a situation occurs, for instance, in our example (cf., Fig. 1) in CA_1: The set of co-occurring artifacts involves two different proactively provided feature mappings. Resolutions may either require further changes to the algorithm (i.e., Strategy 2a which may split the association set and Strategy 2c which needs to count occurrences), involve the risk to lose accuracy (i.e., Strategy 2b may compute more false positives), or require further information from the proactively provided feature traces (Strategy 2d). Therefore, we examined only the safest version in terms of reliability but plan to compare them and further strategies to propagate annotations [17, 19] in the future.

Another potential may involve refining retroactively computed feature traces. Let us consider the Graph variants of Fig. 1 again. For the method subGraph(), the retroactive feature tracing may prefer a feature mapping that combines the features *Graph* and *Edge* (e.g., $Graph \wedge Edge$), as the variants V_2 and V_3 share them but may discard the distinct features *Directed* and *Colored*. A boosted feature tracing can detect a conflict between the proactively provided mapping *Colored* in V_3 (cf., Line 6 of Fig. 1), and the retroactively preferred mapping $Graph \wedge Edge$ and specifically include *Colored* in the heuristically determined mapping, for instance resulting in a mapping $\mathit{Graph} \land \mathit{Edge} \land (\mathit{Directed} \lor \mathit{Colored})$.

Practicality. Retroactive feature tracing offers the unique possibility to recover feature trace information without any additional burden to the developers at first sight. Thus, it may be questionable why and how to collect proactive traces to enrich these techniques from a practical viewpoint.

Firstly, due to its heuristic nature, retroactive feature tracing may still require manual inspection. Developers still have to skim the results of such analyses. In contrast, a small amount of proactive feature traces may be acquired fairly easily. In annotated software, such as C/C++ projects enriched with preprocessor annotations, proactive traces can be extracted out-of-the-box based on the source code annotations. In other types of software projects, code comments, commit messages, or semi-automated proactive tracing, may provide the reliable sources for proactive traces. As a consequence, a minimal seed of proactive traces can be assumed to exist in any software project. Our work aims at optimally exploiting those already spent manual efforts. In fact, we can confirm that about 5% of proactively provided traces can essentially boost the accuracy of retroactive tracing.

Vision. Supported by our results that found a clear boosting effect in several scenarios, we plan to assess and exploit the effect of optimizing the combination of proactive and retroactive feature tracing. The assessment may involve a qualitative analysis of properties between subjects (or code clones in single systems) as well as further subjects (e.g., provided in the ESPLA catalog [35]) to scrutinize how proactive mappings can achieve the best result on retroactive tracing. Given the resulting understanding *which inch to give to take a mile*, we envision to build optimized feature tracing tools which allow developers of highly configurable software to easily assess the complexity of a feature and the impacts of modifying them through highly accurate feature traces. In a broad perspective, this knowledge helps to optimally exploit manually provided information to optimize automated feature techniques, for instance, to train AI models that suggest feature traces retroactively with a minimal but optimal amount of data.

7 Threats to Validity

Internal threats to validity may affect the quality of the results. Errors may have been accidentally introduced in our implementation of the comparison-based feature location algorithm, in the ground truth and variant generation or in the truth tables. To diminish these threats, at minimum a second author reviewed the implementation of another author independently, and vice versa.

External threats to validity concern the generalizability of the results. In contrast to previous work [39], we examined not only ArgoUML as subject but also other configurable, open-source software systems with different types of implicit feature mappings, and with active source code development, thereby also meeting other domains, such as security and networking, text editors, and embedded systems. Furthermore, we did not employ fixed sets of previously aligned variants as proposed in a challenge paper [41], but uniformly sampled the variants. So far, we only examined 30 samples due to computational restrictions, particularly due to the exponential cost of considering additional features in comparison-based tracing. Still, by repeating our experiment with different variant samples, we examined 90 different sets of variants for each subjects system. Thereby, we evaluated the tracability of up to 900 unique features (i.e., preprocessor variables) per subject. This gives confidence in producing similar results when replicating the experiment with different subjects.

Conclusion and construct threats may have been introduced by the selection of mappings to consider in our evaluation, and by the selection of features and mappings to be distributed proactively. We tried to minimize the threats of selecting features and distributed mappings by sampling them in a uniform random way through shuffling the selected features and the selected mappings from the ground truth that are employed as proactive traces. While losing insights on the alignment of the variants, this way we minimize the threat to select only variants that are aligned to produce only positive or negative results.

8 Conclusion

In this paper, we conducted the first in depth exploration of how proactive traces affect the accuracy of automated retroactive feature tracing. We find that only 5% of added proactive knowledge can boost the accuracy of the retroactive tracing and that the number of available variants diminishes this effect. These results are highly promising to minimize the manual efforts by boosting the automated retroactive feature trace recovery. The gained knowledge may also be exploited to train machine learning approaches on accurate and minimal data. Overall, the results demonstrate that only little implicit existing feature knowledge can be used on demand to inform retroactive feature tracing.

In a broader context, we envision the fusion of heuristic retroactive feature tracing with little reliable input from proactive feature tracing. Exploring proactive tracing to boost a retroactive one represents an entirely new research direction which combines two branches of research considered alternative to each other. The promising results of our controlled experiment let us thrive for examining the effects of proactive traces qualitatively in further retroactive techniques; knowledge which eventually allows us optimizing the reliability of automated retroactive feature tracing and, eventually, to optimally support the maintenance of highly configurable software systems and software product lines.

Acknowledgments

We thank Judi Abdullah, Luisa Gerlach, and Angelina Jellinek for contributing to initial versions of the software we used to conduct the experiments. Particularly, we thank Dan Plyukhin for his patience when testing the artifact on MacOS and Benjamin Moosherr doing the same on Windows. This work is partially supported by the German Research Foundation within the project *VariantSync* (TH 2387/1-1 and KE 2267/1-1).

References

[1] 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. Int'l Conf. on Software Reuse (ICSR)*, John M. Favaro and Maurizio Morisio (Eds.), Vol. 7925. Springer, 302–307.

[2] Nasir Ali, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2013. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Trans. on Software Engineering (TSE)* 39, 5 (2013), 725–741.

[3] Sofia Ananieva, Thomas Kühn, and Ralf Reussner. 2022. Preserving Consistency of Interrelated Models during View-Based Evolution of Variable Systems. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Auckland, New Zealand) *(GPCE 2022)*. ACM, New York, NY, USA, 148–163. https://doi.org/10.1145/3564719.3568685

[4] 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. Int'l Conf. on Software Engineering (ICSE)*. ACM, 532–535.

[5] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[6] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *Proc. Int'l Workshop on Reverse Variability Engineering (REVE)*. ACM, 52–59.

[7] Paul Maximilian Bittner, Alexander Schultheiß, Benjamin Moosherr, Timo Kehrer, and Thomas Thüm. 2024. Variability-Aware Differencing with DiffDetective. In *Proc. Int'l Conference on the Foundations of Software Engineering (FSE)*. ACM, New York, NY, USA. To appear.

[8] Paul Maximilian Bittner, Alexander Schultheiß, Thomas Thüm, Timo Kehrer, Jeffrey M. Young, and Lukas Linsbauer. 2021. Feature Trace Recording. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 1007–1020.

[9] Paul Maximilian Bittner, Christof Tinnes, Alexander Schultheiß, Sören Viegener, Timo Kehrer, and Thomas Thüm. 2022. Classifying Edits to Variability in Source Code. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 196–208.

[10] Goetz Botterweck and Andreas Pleuss. 2014. Evolution of Software Product Lines. In *Evolving Software Systems*. Springer, 265–295.

[11] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE 2014* (Hyderabad, India), James D. Herbsleb and Matthew B. Dwyer (Eds.). ACM, 55–69. https://doi.org/10.1145/2593882.2593891

[12] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 16, 10 pages.

[13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *J. Software: Evolution and Process* 25, 1 (2013), 53–95.

[14] 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. Europ. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 25–34.

[15] Andrew David Eisenberg and Kris De Volder. 2005. Dynamic Feature Traces: Finding Features in Unfamiliar Code. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 337–346.

[16] Stefan Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. The ECCO Tool: Extraction and Composition for Clone-and-Own. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 665–668.

[17] Sandra Greiner, Michael Nieke, and Christoph Seidl. 2022. Towards Trace-Based Synchronization of Variability Annotations in Evolving Model-Driven Product Lines. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)* (Florence, Italy). ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/3510466.3510470

[18] Sandra Greiner, Alexander Schultheiss, Paul Maximilian Bittner, Thomas Thüm, and Timo Kehrer. 2024. *Replication Package: Give an Inch and Take a Mile? Effects of Adding Reliable Knowledge to Heuristic Feature Tracing*. https://doi.org/10.5281/zenodo.12620493

[19] Sandra Greiner and Bernhard Westfechtel. 2019. On Determining Variability Annotations In Partially Annotated Models. In *Proc. Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* (Leuven, Belgium). ACM, New York, NY, USA, Article 17. https://doi.org/10.1145/3302333.3302341

[20] Florian Heidenreich, Jan Kopcsek, and Christian Wende. 2008. FeatureMapper: Mapping Features to Models. In *Companion Int'l Conf. on Software Engineering (ICSEC)*. ACM, 943–944. Informal demonstration paper.

[21] Philipp Heisig, Jan-Philipp Steghöfer, Christopher Brink, and Sabine Sachweh. 2019. A Generic Traceability Metamodel for Enabling Unified End-to-End Traceability in Software Product Lines. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. ACM, New York, NY, USA, 2344–2353. https://doi.org/10.1145/3297280.3297510

[22] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability With Embedded Annotations. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 61–70.

[23] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. 2013. Integrating Conceptual and Logical Couplings for Change Impact Analysis in Software. *Empirical Software Engineering (EMSE)* 18, 5 (2013), 933–969.

[24] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2008. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 311–320.

[25] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semiautomatic Detection of Product-Line Features. *IEEE Trans. on Software Engineering (TSE)* 40, 1 (2014), 67–82.

[26] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. 2012. A Variability-Aware Module System. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 773–792.

[27] Benjamin Klatt, Martin Küster, and Klaus Krogmann. 2013. A Graph-Based Analysis Concept to Derive a Variation Point Design From Product Copies. In *Proc. Int'l Workshop on Reverse Variability Engineering (REVE)*. 1–8.

[28] Rainer Koschke and Jochen Quante. 2005. On dynamic feature location. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 86–95.

[29] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems - Foundations and Applications*. Auerbach Publications / Taylor & Francis, 153–172.

[30] Raúl Lapeña, Manuel Ballarin, and Carlos Cetina. 2016. Towards Clone-and-Own Support: Locating Relevant Methods in Legacy Products. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 194–203.

[31] 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 *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 105–114.

[32] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2015. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *Proc. Int'l Symposium on Software and Systems Traceability (SST)*. IEEE, 57–60.

[33] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability Extraction and Modeling for Product Variants. *Software and System Modeling (SoSyM)* 16, 4 (2017), 1179–1199.

[34] Salome Maro, Jan-Philipp Steghöfer, Paolo Bozzelli, and Henry Muccini. 2022. TracIMo: a traceability introduction methodology and its evaluation in an Agile development team. *Requirments Engineering* 27, 1 (2022), 53–81. https://doi.org/10.1007/s00766-021-00361-5

[35] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 38–41.

[36] Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. 2018. Feature Location Benchmark with ArgoUML SPL. In *Proceedings of the 22nd International Systems and Software Product Line Conference - Volume 1* (Gothenburg, Sweden) *(SPLC '18)*. ACM, New York, NY, USA, 257–263. https://doi.org/10.1145/3233027.3236402

[37] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110.

[38] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K.G. Assunção, Stefan Fischer, and Alexander Egyed. 2021. A Hybrid Feature Location Technique for Re-Engineering Single Systems Into Software Product Lines. In *Proc. Int'l Working Conf. on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Article 11, 9 pages.

[39] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. 2019. Comparison-Based Feature Location in ArgoUML Variants. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 93–97.

[40] Gabriela Karoline Michelon, Jabier Martinez, Bruno Sotto- Mayor, Aitor Arrieta, Wesley K. G. Assunção, Rui Abreu, and Alexander Egyed. 2023. Spectrum-based feature localization for families of systems. *Journal of Systems and Software* 195 (2023), 111532. https://doi.org/10.1016/j.jss.2022.111532

[41] Gabriela Karoline Michelon, David Obermann, Wesley K. G. Assunção, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. Managing Systems Evolving in Space and Time: Four Challenges for Maintenance, Evolution and Composition of Variants. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 75–80.

[42] Gabriela K. Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley K. G. Assunção. 2021. Spectrum-Based Feature Localization: A Case Study Using ArgoUML. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. ACM, New York, NY, USA, 126–130. https://doi.org/10.1145/3461001.3473065

[43] Rodrigo André Ferreira Moreira, Wesley K. G. Assunç ão, Jabier Martinez, and Eduardo Figueiredo. 2022. Open-source software product line extraction processes: the ArgoUML-SPL and Phaser cases. *Empirical Software Engineering* 27, 4

(2022), 85. https://doi.org/10.1007/s10664-021-10104-3

[44] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. 2023. FeatRacer: Locating Features Through Assisted Traceability. *IEEE TSE* 49, 12 (2023), 5060–5083. https://doi.org/10.1109/TSE.2023.3324719

[45] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Trans. on Software Engineering (TSE)* 33, 6 (2007), 420–432.

[46] Kamil Rosiak and Ina Schaefer. 2023. The e4CompareFramework: Annotation-based Software Product-Line Extraction. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 34–38.

[47] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Languages, and Conceptual Models*, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin (Eds.). Springer, 29–58.

[48] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *Proc. Int'l Systems and Software Product Line Conf. (SPLC)*. ACM, 101–110.

[49] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. 2023. SEAL: Integrating Program Analysis and Repository Mining. *ACM Transactions on Software Engineering and Methodology* 32, 5, Article 121 (jul 2023), 34 pages. https://doi.org/10.1145/3585008

[50] Thomas Schmorleiz and Ralf Lämmel. 2014. Similarity Management via History Annotation. In *Proc. Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE)*. Dipartimento di Informatica Università degli Studi dell'Aquila, L'Aquila, Italy, 45–48.

[51] Alexander Schultheiß, Paul Maximilian Bittner, Sascha El-Sharkawy, Thomas Thüm, and Timo Kehrer. 2022. Simulating the Evolution of Clone-and-Own Projects With VEVOS. In *Proc. Int'l Conf. on Evaluation Assessment in Software Engineering (EASE)*. ACM, 231–236.

[52] Felix Schwägerl and Bernhard Westfechtel. 2016. SuperMod: Tool Support for Collaborative Filtered Model-Driven Software Product Line Engineering. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*. ACM, 822–827.

[53] Janet Siegmund. 2016. Program Comprehension: Past, Present, and Future. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–20.

[54] George Spanoudakis and Andrea Zisman. 2005. Software traceability: a roadmap. In *Handbook of software engineering and knowledge engineering: vol 3: recent advances*. World Scientific, 395–428.

[55] Stefan Stănciulescu, Thorsten Berger, Eric Walkingshaw, and Andrzej Wąsowski. 2016. Concepts, Operations, and Feasibility of a Projection-Based Variation Control System. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 323–333.

[56] Stefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME)*. IEEE, 151–160.

[57] Matús Sulír, Milan Nosál, and Jaroslav Porubän. 2018. Recording Concerns in Source Code Using Annotations. *Computing Research Repository (CoRR)* abs/1808.03576 (2018).

[58] Anneliese von Mayrhauser, A. Marie Vans, and Adele E. Howe. 1997. Program Understanding Behaviour During Enhancement of Large-Scale Software. *J. Software: Evolution and Process* 9, 5 (1997), 299–327.

[59] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. 2018. Variability-Aware Static Analysis at Scale: An Empirical Study. *ACM TOSEM* 27, 4 (2018), 18:1–18:33. https://doi.org/10.1145/3280986

[60] Neil Walkinshaw, Marc Roper, and Murray Wood. 2007. Feature Location and Extraction using Landmarks and Barriers. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE, 54–63.

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

[62] David Wille, Sandro Schulze, Christoph Seidl, and Ina Schaefer. 2016. Custom-Tailored Variability Mining for Block-Based Languages. In *Proc. Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 271–282.

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