A Hybrid Feature Location Technique for Re-engineering Single Systems into Software Product Lines

Gabriela Karoline Michelon[1,2], Lukas Linsbauer[3], Wesley K. G. Assunção[4], Stefan Fischer[5], Alexander Egyed[1]
[1]Institute for Software Systems Engineering - Johannes Kepler University Linz, Austria
[2]LIT Secure and Correct Systems Lab - Johannes Kepler University Linz - Austria
[3]Institute of Software Engineering and Automotive Informatics - Technical University of Braunschweig, Germany
[4]PPGComp - Western Paraná State University, and Federal University of Technology - Paraná, Brazil
[5]Software Competence Center Hagenberg GmbH, Austria

ABSTRACT

Software product lines (SPLs) are known for improving productivity and reducing time-to-market through the systematic reuse of assets. SPLs are adopted mainly by re-engineering existing system variants. Feature location techniques (FLTs) support the re-engineering process by mapping the variants' features to their implementation. However, such FLTs do not perform well when applied to single systems. In this way, there is a lack of FLTs to aid the re-engineering process of a single system into an SPL. In this work, we present a hybrid technique that consists of two complementary types of analysis: i) a dynamic analysis by runtime monitoring traces of scenarios in which features of the system are exercised individually, and ii) a static analysis for refining overlapping traces. We evaluate our technique on three subject systems by computing the common metrics used in FL research. We thus computed Precision, Recall, and F-Score at the line- and method-level of source code. In addition to that, one of the systems has a ground truth available, which we also used for comparing results. Results show that our FLT reached an average of 68-78% precision and 72-81% recall on two systems at the line-level, and 67-65% precision and 68-48% recall at the method-level. In these systems, most of the implementation can be covered by the exercise of the features. For the largest system, our technique reached a precision of up to 99% at the line-level, 94% at the method-level, and 44% when comparing to traces. However, due to its size, it was difficult to reach high code coverage during execution, and thus the recall obtained was on average of 28% at the line-level, 25% at the method-level, and 30% when comparing to traces. The main contribution of this work is a hybrid FLT, its publicly available implementation, and a replication package for comparisons and future studies.

CCS CONCEPTS

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

VaMoS'21, February 9–11, 2021, Krems, Austria

ACM ISBN 978-1-4503-8824-5/21/02…$15.00
https://doi.org/10.1145/3442391.3442403

KEYWORDS

traceability, feature location, software reuse, runtime monitoring

ACM Reference Format:
Gabriela Karoline Michelon[1,2], Lukas Linsbauer[3], Wesley K. G. Assunção[4], Stefan Fischer[5], Alexander Egyed[1]. 2021. A Hybrid Feature Location Technique for Re-engineering Single Systems into Software Product Lines. In *15th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS'21), February 9–11, 2021, Krems, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3442391.3442403

1 INTRODUCTION

Software Product Line (SPL) is an approach to support the systematic reuse of software assets to create different products. SPLs improve the productivity of software companies, reduce costs, and human effort in developing systems. Usually, an SPL is adopted as an extractive approach, i.e., by the extraction of common assets from existing system products that were created by the opportunistic reuse of copy-and-paste [3]. Therefore, approaches for re-engineering systems into SPLs represent an important research topic, where the feature location techniques (FLTs) are essential for supporting the first step for re-engineering process [4]. The feature location process consists of identifying the implementation artifacts that belong to a feature, where a feature can be any system functionality visible to the user [17]. Furthermore, feature location is one of the most important activities in software maintenance and evolution tasks, such as documentation, configuration, addition, removal, or improvement of its functionalities [31]. In addition to that, it is necessary to perform further studies on how to support the entire process of automatic re-engineering of existing variants into an SPL, as most of the existing approaches are semi-automatic and requires extensive human effort [4].

Existing FLTs to automate the process of finding a feature's implementation are limited to coarse levels of granularity (generally the method-level) and/or to not yield good results when applied to single systems [31]. Usually, FLTs for SPL re-engineering analyze the commonalities and differences of existing system variants. However, when there is only one single variant of a system, these strategies do not provide satisfactory results [1, 9, 27]. An effective FLT for locating features of a single system can be useful when aiming to provide a larger product portfolio and serve more customers [25]. Thus, an FLT able to locate features at a fine level of granularity within a single system may facilitate the reuse of source code, and hence, the creation of new variants of the system.

From existing FLTs, static, textual, and dynamic analysis are the most common [4, 11]. The static analysis depends on textual information of a set of existing system variants as well as control or data flow dependencies [1, 27]. However, static analysis often overestimates traces of a feature and retrieve much false positive information [20, 22]. Among FLTs based on text analysis [5, 16, 29, 30, 32], information retrieval (IR) techniques are the most commonly used. However, the quality of any of the textual analysis depends on the source code naming conventions and/or the user-issued query [11]. The dynamic feature location consists of executing the system's code by exercising the subject feature behaviors and monitoring the runtime events, such as object creation/deletion, method invocation, thread creation/termination [12]. Recorded runtime traces can then be used to find parts of the corresponding feature implementation. Hence, it does not require a set of system variants to locate features, and thus, can be used for locating features of a single system [22]. However, the dynamic analysis can have a considerable overhead on a system's execution and is unable to distinguish overlapping source code between features [30]. Dynamic analysis needs appropriate test suites or the design of proper scenarios to invoke only and all traces regarding a specific feature [11, 22]. This may require prior knowledge of the implementation, and developers usually have at most a rough understanding of the system, otherwise, the feature location would not be an issue for re-engineering a system into an SPL [20]. Thus, effective FLTs for re-engineering single systems into SPLs are even more challenging [13].

In this work, we present a semi-automated and hybrid FLT, which combines the results of a dynamic and static analysis. The first analysis consists of runtime monitoring scenarios in which system features are exercised. This provides us traces of code executed while exercising such features of a single system. With the traces, our technique can simulate variants, i.e., create artificial variants, to be used as input for the next analysis of our FLT. The second analysis consists of refining the traces by statically analyzing the common artifacts and corresponding features. This static analysis aims to filter out the source code that belongs to multiple features. We chose a hybrid technique because the dynamic execution data can contain a lot of noise and we need multiple traces from different scenarios to have variants exercising different features [2]. Then, we reduce this noise from the artificial variants by performing static analysis to filter overlapping traces from different features variants. This combination leads to promising results for locating features of single systems. Hence, our technique provides proper support for re-engineering single systems into SPLs.

For evaluating the efficiency of our technique for locating features of a single system, we applied our hybrid FLT to three Java systems, namely Sudoku, Notepad, and ArgoUML. These systems have a Graphical User Interface (GUI), which we used to perform scenarios to exercise their features. We chose those systems, as they are publicly available SPLs with known sets of features. Also, ArgoUML is a large and complex system with a ground-truth available [24]. This makes our results and experiments reproducible and allows for future comparison. After locating the features, we computed commonly used metrics Precision, Recall, and F-Score [31] for comparison of traces and source code of the system variants with the composed ones.

Our work contributes to the state of the art by improving the results of locating features of a single system [9]. Related literature argues that existing FLTs usually perform well when multiple system variants exist [26, 27]. Thus, it is important to explore how to improve feature location in single systems, and hence, ease the re-engineering process of a legacy system into an SPL. In addition to that, the implementation[1] and dataset[2] are available for reproducibility, evaluation and comparison with other techniques.

2 HYBRID FEATURE LOCATION TECHNIQUE

An overview of our technique is shown in Figure 1, composed of three steps. Steps 1 (cf. Section 2.1) and 3 (cf. Section 2.3) are designated to the dynamic and static analyses, respectively. The step in between (Step 2, cf. Section 2.2) is responsible for preparing the data obtained from the results of the dynamic analysis to compute the results of the static analysis. The basic idea of our hybrid technique is to create multiple artificial system variants from the executions of a single system. These executions can be performed manually (i.e., using the GUI) or automatically (e.g., automated tests). Each execution is intended to exercise one or more features defined in advance. The information of exercised features is used as input (cf. Figure 1) to represent the configuration of the dynamic variants. In this way, FLTs that perform well on larger sets of system variants can be applied to single systems and reach even better results than FLTs built for single systems.

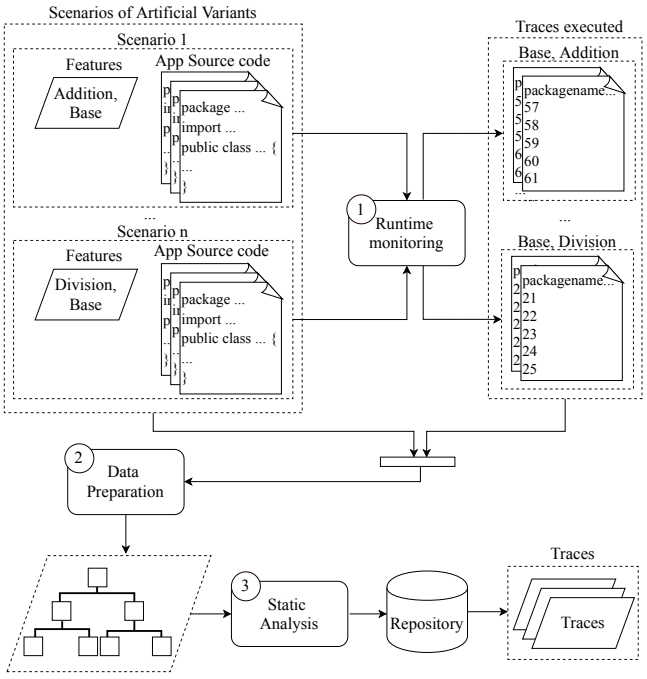

Figure 1: Hybrid feature location technique.

[1] https://github.com/GabrielaMichelon/hybridFLT
[2] https://doi.org/10.5281/zenodo.4262529

2.1 Runtime Monitoring

The dynamic analysis of our hybrid FLT starts with runtime monitoring when exercising one or more features. This monitoring allows for obtaining the features' execution traces (Step 1 in Figure 1). This step can be performed either by manually executing designed scenarios using the system GUI or automatically by executing unit tests to exercise the features. This latter requires automated unit tests being available. After all scenarios/features were executed/exercised, the output is a report of execution traces. The output consists of text files named by the package and class executed. Each file contains the number of lines executed within the source code of the respective class. This runtime monitoring can be applied for any programming language, only requiring to generate the text files with execution traces, used for the next step of our technique.

To illustrate our hybrid FLT, we use a simple calculator presented in Listing 1. The calculator system has four operations, which are the features of the system, namely *Addition*, *Subtraction*, *Multiplication* and *Division*. In the first step, we monitor the execution of scenarios that exercise the system features. Let us consider the first scenario the exercising of the feature *Addition*. When executing the calculator, the user will perform only the *Addition* operation. The lines of code executed, i.e., traced, will be the Lines 32-42 of Listing 1. Furthermore, some lines from the core of the project, i.e., common code that does not belong to a specific feature, a.k.a *Base* feature, will also be executed and traced. In this case, the Lines 1-13 from Listing 1 will be executed too. When executing other scenarios, for example for feature *Division*, our technique will get traces with the Lines 44-54 executed besides the Lines 1-10, 14, 18, and 22-25 from Listing 1.

2.2 Data Preparation

The second step (Step 2 in Figure 1) needs as input: (i) traces executed, which are report/text files containing the lines executed for each class from runtime monitoring scenarios; (ii) the artificial variants of each monitored scenario, which consists of the name of the feature(s) that were exercised in a specific scenario, and (iii) the system application source code files. This step consists of restructuring the source code, at the desired granularity. For our technique, we consider the granularity of the line-level of source code. The data preparation in this step consists of parsing the source code files in tree structures according to the executed lines and the system programming language. In the tree structure, presented in our previous work [27], nodes represent the desired trace granularity, e.g., lines of code, to be retrieved by the feature location. The output of this second step is one tree structure for each file that is used as input data for the third step (cf. bottom part of Figure 1).

For our illustrative example with feature *Addition*, a tree is generated in which the root is the class from Listing 1. The children of the root are: the required imports, which does not exist in our example; the method nodes, which are `addition()` and `main(String[] args)`; the field nodes, as for example `Scanner scan = new Scanner(System.in)`; nested enums or classes, which our example does not have. The two methods' children are all the lines executed when we exercised the feature *Addition*. Thus, the children of the method `addition()` are the Lines 33-42 of the Listing 1, and the children lines of the method `main(String[] args)`

```
1   public static void main(String[] args) {
2       SimpleCalculator calc = new SimpleCalculator();
3       Scanner scan = new Scanner(System.in);
4       scan.useLocale(Locale.US);
5       int x;
6       double score;
7       calc.showMenu();
8       x = scan.nextInt();
9       switch (x) {
10          case 1:
11              score = calc.addition();
12              System.out.println(score);
13              break;
14          case 2:
15              score = calc.subtraction();
16              System.out.println(score);
17              break;
18          case 3:
19              score = calc.mutliplication();
20              System.out.println(score);
21              break;
22          case 4:
23              score = calc.division();
24              System.out.println(score);
25              break;
26          default:
27              System.out.println("Wrong choice");
28              break;
29      }
30  }
31
32  public double addition() {
33      scan.useLocale(Locale.US);
34      double a, b;
35      System.out.println("Enter first value");
36      a = scan.nextDouble();
37      System.out.println("Enter second value");
38      b = scan.nextDouble();
39      double val = a + b;
40
41      return val;
42  }
43
44  public double division() {
45      scan.useLocale(Locale.US);
46      double a, b;
47      System.out.println("Enter first value");
48      a = scan.nextDouble();
49      System.out.println("Enter second value");
50      b = scan.nextDouble();
51      double val = a / b;
52
53      return val;
54  }
55
56  ...
```

Listing 1: Code snippet from class SimpleCalculator.

are the Lines 2-13 and 29 from Listing 1. Then, after restructuring the data regarding all the classes executed of a feature, we proceed with the third step (Step 3 in Figure 1), which we will explain next.

2.3 Static Analysis

In this step (Step 3 in Figure 1), we apply a static feature location technique. Steps 2 and 3 are incremental. Thus, for every artificial variant, i.e., the variants created by executing scenarios, we prepare the data (Step 2 in Figure 1) and compute new or refine existing traces (static analysis in Step 3 in Figure 1). In this step, traces are computed or refined in associations, where an association contains a condition holding positive or negative features and a tree containing the artifacts for the respective condition. The condition is represented by presence conditions. The presence condition is computed in the form of a disjunctive normal form formula, where it is composed of a conjunction of literals, i.e., the features.

As an illustrative example, the first input is related to the feature *Addition*. We thus create an association containing one or more tree

structures with nodes that represent the artifacts and assign them to a presence condition containing a positive clause, i.e., containing the features *Addition* and *Base*. As our illustrative example contains only one file, in the second input exercising feature *Division*, we will have only one artifact tree corresponding to traces of the *Division*. However, as we mentioned before, we also have lines traced for the feature *Base*. Then, when comparing the artifact tree of the existing association with the new input tree, we check for the shared/common artifacts. This comparison is performed by aligning the existing tree(s) artifacts with the input tree(s) artifacts by the Longest Common Subsequence (LCS) algorithm [10].

The traces are updated if common features or artifacts are used as input more than once. The incremental process to update existing traces consists of analyzing the commonalities and differences between the artifacts in existing associations with the new input artifacts. The analysis of which features belong to the associations and the features existing in the input is considered to refine the traces. Thus, when the same artifact is already assigned to a feature, we just update the presence condition of its association adding and/or removing positive or negative features. The uncommon artifacts between the existing and the new trees that are not assigned to any existing association will be part of a new association and the respective feature(s) used as input. In this way, common lines of source code executed between features may belong to the *Base* source code of the system. Thus, those artifacts must be assigned to a feature used to solely represent the *Base* of the system. For creating and updating traces, we use a counter to distinguish how many times a feature was used as input, in how many inputs an artifact is contained, and in how many inputs both feature(s) and artifact(s) were contained together.

For our example, when we compute traces for the second input, Lines 32-42 from Listing 1 appeared once, and part of the lines of method `main(String[])` appeared once and twice. The feature *Addition* appeared once and feature *Division* once, while feature *Base* is part of both inputs. Then, the artifacts with counter one that appeared only in feature *Addition* are unshared artifacts with *Division*, as they do not appeared in feature *Division* and are not part of *Base*, which counter is two. Besides these artifacts appeared with the feature *Addition* and *Base*, they were not in the input data used with the feature *Division* and *Base*. We thus update the existing association to hold only the artifacts that appeared once with the feature *Addition*. The association will have its presence condition updated from $Addition \wedge Base$ for $Addition \wedge Base \wedge !Division$. Then, two new associations are created, one containing the artifacts that appeared once with feature *Division* and presence condition $Base \wedge Division \wedge !Addition$. The other new association contains the remaining artifacts with the presence condition *Base* because the remaining artifacts appeared twice in variants containing *Base*, which also appeared just twice as input data.

3 EVALUATION

This section first presents the goal and the research question (RQ) of this study. Then, we describe how we evaluate our technique by showing the characteristics of the subject systems, the metrics used, and the methodology we chose for computing the metrics. Finally, we discuss the implementation aspects of our hybrid FLT.

Research Goal: Evaluate the efficiency of our hybrid FLT for aiding the process of re-engineering single systems into SPLs.

RQ. How effective is our technique for locating features of a single system? To answer this RQ, we designed a scenario to perform the detection analysis of the re-engineering process of a single system into an SPL. The scenario consists of a single system containing all its existing features. Then, we computed the metrics Precision, Recall, and F-Score (cf. Section 3.2) to evaluate the efficiency of our technique for locating features in single systems. The metrics were computed by comparing the traces and/or artifacts from variants for each feature of the system.

3.1 Subject Systems

We selected three public Java SPLs, widely-used for product-line analyses: Sudoku, Notepad [18], and ArgoUML [7]. These systems have often been used to evaluate and compare FLTs [1, 9, 19, 26, 27, 28]. The aforementioned studies that used ArgoUML to evaluate their FLTs only used this system as a subject. In addition to them, we use two more subject systems in this work, providing a more robust evaluation. Table 1 shows the number of lines of source code (LoC), the number of features, and tests of each subject system.

Regarding the ArgoUML system, it has an established ground truth available [24], which gives an appropriate evaluation of this system. Some of its features are diagrams, which we exercised by invoking each diagram operation manually on the GUI. However, we cannot exercise individually features that are not related to diagrams such as *Cognitive* and *Logging*, as they are features that automatically run in the background. The *Cognitive* and *Logging* features provide information to help designers to detect problems in their models. For example, they analyze the diagrams and indicate potential problems warning about parts of the project that have not been finalized and/or existing syntax errors in models. Therefore, for ArgoUML, we considered the features *Base*, *Cognitive*, and *Logging*, besides the features specific for each diagram, as input for the static analysis of each artificial variant. For the Notepad system, to execute the scenario that exercises the feature *Redo*, we also need to previously exercise the feature *Undo*. Thus, we used as input for exercising the artificial variant created by the feature *Redo*, the features *Undo*, *Redo*, and *Base*. For the Sudoku system, we used as a set of features for the scenarios executed only a specific feature exercised and *Base*. All the artificial variants and configurations used for the evaluation are provided in our dataset[3]. The ArgoUML unit tests were not annotated as the system source code of the SPL, we thus add annotations with preprocessor directives on tests to have distinction tests from ArgoUML for each feature, similar to test case generation for SPLs presented in previous work [15].

Table 1: Subject Systems

System	LoC	Features	Test Cases
Sudoku	2129	5	0
Notepad	2294	20	0
ArgoUML	319896	8	1198

[3] https://doi.org/10.5281/zenodo.4262529

3.2 Metrics

To evaluate the effectiveness of our FLT, and for allowing comparison of our technique to other ones, we used the common metrics of Precision (P), Recall (R), and F-Score (F) [23], defined as follows:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

where TP (true positives) are the correctly retrieved traces or lines of source code or methods and FP (false positives) are the traces or lines of source code or methods retrieved by our FLT that do not exist in the ground truth traces or variants.

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

where FN (false negatives) are traces or lines of source code or methods that exist in the ground truth or variant but were not retrieved by our FLT. The F-Score is the weighted average of Precision and Recall.

$$F\text{-}Score = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (3)$$

For the subject systems of our study, we compared the correctly and missed lines of code retrieved by the traces located with our FLT. Sudoku and Notepad SPLs rely on Java custom annotations to encode variability [19], which enables us to create ground truth variants (available in our dataset). We thus computed the metrics regarding the source code for each feature and the common base code by using a library for performing the comparison operations between textual data[4]. In the case of ArgoUML, we also compared traces from the benchmark available [24].

The methodology we follow for computing the evaluation metrics for every subject system is shown in Figure 2. After having applied our hybrid FLT to the set of scenarios, the computed traces are stored in a repository. We then use the computed traces to compose variants with the same configurations as were used as input (cf. top part of Figure 2). The implementation artifacts of each composed variant are then compared to the artifacts of the respective ground truth variant with the same configuration. This comparison is performed once on method-level and once on line-level of granularity to make it easier to compare our results also to FLTs that only operate on method-level. For ArgoUML not only ground truth variants but also ground truth traces are available from the ArgoUML benchmark [24] (cf. bottom part of Figure 2). In this case, we also computed metrics by comparing the computed traces with the ground truth traces.

3.3 Implementation Aspects

For monitoring traces of features exercised either manually or by tests, we used the JaCoCo free code coverage tool[5]. We chose JaCoCo because it offers support for line-level granularity, different from existing tools used in previous work [22], which supports class- and method-level granularity. Furthermore, it also allows the generation of code coverage reports readable in many formats, including HTML, CSV, and XML, showing information, such as the number and percentage of classes, methods, and lines missed.

[4]https://github.com/java-diff-utils/java-diff-utils
[5]https://www.eclemma.org/jacoco/

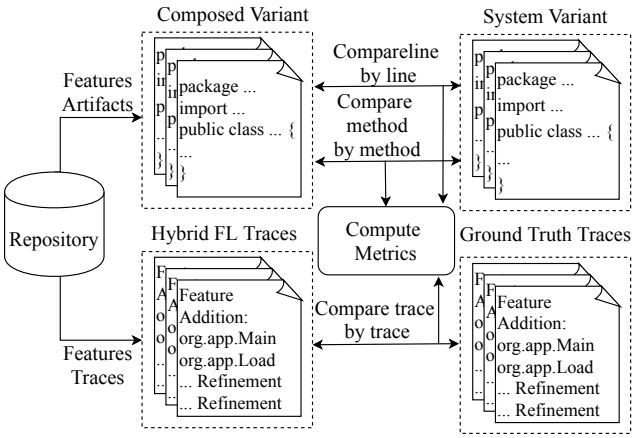

Figure 2: Methodology for computing metrics.

For runtime monitoring of features exercised on a system GUI, we used the JaCoCo Java agent. It can be used to dynamically trace a running Java program. For using the Java agent, we first generate a Java executable file of the system application and then execute it with JaCoCo for runtime monitoring. We monitored traces from the launch of the application GUI until it shuts down. While running the application, we exercised a feature to trace as much of its source code as possible, by clicking on its particular available functionalities. All the executions on the GUI were recorded for future work comparisons and are available within our dataset. For monitoring traces of features exercised by the unit tests of a system, we used the JaCoCo code coverage Maven plugin integrated into the build process to allow us to record the code coverage during testing. The feature unit tests were executed on the system variant containing all features in their source code.

Regarding the static analysis (cf. Section 2), we implemented a Java adapter for preparing the data for the FLT implemented in the ECCO tool [14, 21]. We used the FLT implemented in ECCO because it is independent of the artifacts type and reached higher precision and recall when applied to a few system variants [27]. It only requires an adapter for preparing the data in a tree structure used in the static analysis. Thus, as we only applied for Java source code artifacts, for now, we used a JavaParser library[6] to implement an adapter. The adapter is needed to parse the Java source code in a tree structure containing the required granularity. In this case, we have nodes for each class holding imports, fields, class (including nested classes), methods, and the method body as lines.

4 RESULTS AND DISCUSSION

In this section, we present and discuss the results of our study in evaluating our FLT. We present the metrics computed for each of the subject systems by comparing the lines and methods of retrieved source code with the actual lines and methods of source code of each feature variant. In the case of ArgoUML, in addition to the line- and method-level, we present results by comparing the traces retrieved with the ground truth traces from the ArgoUML benchmark [24].

[6]https://javaparser.org/

Table 2: Comparison line by line and method by method per feature exercised manually on Sudoku GUI.

	Line			Method		
Feature	**P**	**R**	**F**	**P**	**R**	**F**
States	0.78	0.33	0.46	0.70	0.31	0.43
Solver	0.81	0.89	0.84	0.77	0.82	0.79
Generator	0.67	0.77	0.72	0.66	0.75	0.70
Undo	0.59	0.84	0.70	0.62	0.79	0.70
Extended	0.58	0.76	0.66	0.60	0.74	0.67
Average	0.68	0.72	0.70	0.67	0.68	0.68

P = Precision; R = Recall; F = F-Score.

Table 3: Comparison line by line and method by method per feature exercised manually on Notepad GUI.

	Line			Method		
Feature	**P**	**R**	**F**	**P**	**R**	**F**
About	0.78	0.77	0.77	0.60	0.36	0.45
AboutMe	0.78	0.77	0.77	0.60	0.36	0.45
Copy	0.74	0.79	0.76	0.53	0.42	0.47
Cut	0.76	0.79	0.78	0.59	0.42	0.49
ExitApp	0.81	0.74	0.77	0.67	0.42	0.51
Find	0.77	0.79	0.78	0.59	0.42	0.49
FindNext	0.78	0.78	0.78	0.59	0.40	0.48
Fonts	0.88	0.82	0.85	0.84	0.53	0.65
LineNumber	0.78	0.79	0.79	0.59	0.42	0.49
LineWrap	0.76	0.77	0.77	0.61	0.44	0.51
New	0.79	0.93	0.85	0.71	0.71	0.71
Open	0.80	0.90	0.84	0.71	0.68	0.69
Paste	0.76	0.79	0.78	0.59	0.42	0.49
Print	0.79	0.81	0.80	0.73	0.53	0.62
Redo	0.83	0.81	0.82	0.79	0.52	0.63
Save	0.78	0.94	0.86	0.72	0.75	0.73
SelectAll	0.76	0.79	0.78	0.59	0.42	0.49
TimeDate	0.76	0.79	0.78	0.59	0.42	0.49
Toolbar	0.78	0.8	0.79	0.67	0.48	0.56
Undo	0.79	0.8	0.79	0.67	0.46	0.55
Average	0.78	0.81	0.80	0.65	0.48	0.55

P = Precision; R = Recall; F = F-Score.

Table 4: Comparison line by line, method by method, and trace by trace per feature on ArgoUML.

	GUI								
	Line			Method			Traces		
Feature	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
ActivityDiagram	0.81	0.30	0.44	0.81	0.21	0.34	0.05	0.24	0.08
Cognitive	0.97	0.27	0.42	0.83	0.28	0.42	0.23	0.52	0.32
CollaborationDiagram	0.81	0.29	0.42	0.79	0.23	0.35	0.04	0.19	0.06
DeploymentDiagram	0.82	0.31	0.45	0.78	0.25	0.37	0.04	0.45	0.07
Logging	0.82	0.25	0.38	0.71	0.26	0.37	0.00	0.00	0.00
SequenceDiagram	0.83	0.27	0.41	0.77	0.25	0.38	0.12	0.25	0.16
StateDiagram	0.81	0.28	0.42	0.77	0.26	0.39	0.08	0.31	0.13
UseCaseDiagram	0.82	0.29	0.42	0.73	0.26	0.39	0.08	0.47	0.13
Average	0.84	0.28	0.42	0.77	0.25	0.38	0.08	0.30	0.12
	Tests								
	Line			Method			Traces		
Feature	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
ActivityDiagram	0.99	0.05	0.09	1.00	0.03	0.07	0.24	0.07	0.11
Cognitive	0.98	0.14	0.24	0.94	0.11	0.20	0.26	0.49	0.34
CollaborationDiagram	0.99	0.05	0.09	0.98	0.04	0.08	0.29	0.14	0.19
DeploymentDiagram	0.99	0.04	0.08	0.96	0.04	0.07	0.57	0.10	0.16
Logging	0.99	0.04	0.09	0.85	0.04	0.09	0.95	0.10	0.18
SequenceDiagram	0.99	0.04	0.08	0.96	0.04	0.07	0.33	0.04	0.08
StateDiagram	0.99	0.05	0.10	0.97	0.05	0.09	0.28	0.16	0.21
UseCaseDiagram	0.99	0.05	0.09	0.88	0.05	0.09	0.62	0.20	0.30
Average	0.99	0.06	0.11	0.94	0.05	0.10	0.44	0.16	0.24

P = Precision; R = Recall; F = F-Score.

Table 5: Statement coverage in ArgoUML.

		GUI		Tests	
Feature	**TS**	**CS**	**%**	**CS**	**%**
ActivityDiagram	47826	11787	24.65	3528	7.37
Cognitive	53745	*	*	7924	14.74
CollaborationDiagram	46854	11153	23.80	3475	7.42
DeploymentDiagram	47005	11807	25.12	3373	7.18
Logging	47082	*	*	3395	7.21
SequenceDiagram	48613	11044	22.72	3377	6.95
StateDiagram	48327	11304	23.40	3440	7.12
UseCaseDiagram	47396	11516	24.30	2427	5.12

TS = Total Statements; CS = Covered Statements.

The results from the computed metrics for each system's features obtained by manually exercising features on the GUI are presented in Tables 2, 3, and 4. In Tables 2 and 3 the metrics computed for Sudoku and Notepad, respectively, were based on the lines and methods of source code retrieved from the traces of our hybrid feature location. For the Sudoku system, our technique reached on average 68% of precision and 72% of recall at the line-level and 67% of precision and 68% of recall at the method-level. For the Notepad system, our technique reached on average precision of 78% and 81% of recall at the line-level and precision of 65% and recall of 48% at the method-level. For ArgoUML, presented in Table 4, our technique reached an average of 84% precision and 28% recall at the line-level, and 77% precision and 25% recall at the method-level.

Regarding the results from exercising features with ArgoUML unit tests, our hybrid FLT reached 99% of precision and 6% of recall on average. The results of features exercised either manually or with automated unit tests for ArgoUML show lower recall compared to the other two systems, which means the occurrence of many false negatives, i.e., lines of code missing in relation to the

total relevant lines of code. Table 5 presents that only ≈25% of the statements of feature variants were covered when exercising manually and on average ≈8% with unit tests. Due to ArgoUML being the most complex and large subject system of our study [8], in which more than 80% of the code belongs to the feature *Base*, it is very challenging execute all lines of its source code. Thus, when executing the scenarios for exercising each feature of the system, we could not reach the coverage of all the lines of the *Base* code. In Table 4, we also present the comparison for ArgoUML at the method-level to be able to compare with Cruz et al. [9] results using textual IR techniques. Our hybrid technique obtained higher values of precision and recall on average (77% and 25%, respectively) compared to their results (16% and 19%, respectively) when exercising manually the features on GUI. Comparing their results (at the method-level) with our results from exercising features with unit tests, we reached worse recall but almost optimal precision. However, they used five system variants to locate features, while our results used only one system variant.

In addition, using retrieved traces from the ArgoUML benchmark [24], we were also able to compare the results of our hybrid FLT to feature location from previous work [27]. In this previous work, the average precision and recall were 2% and 41%, respectively. In Table 4, we can see that our FLT reached better results of precision either exercising features manually or by tests using our hybrid FLT. Despite the recall reached being worse (30% manually on GUI, and 16% with unit tests, on average), our technique reached better precision either exercising features with unit tests and manually on GUI (44% and 8% on average, respectively). Although the existing unit tests from ArgoUML accordingly exercise the features, which explain the higher precision (line- and method-level), the existing tests from ArgoUML are not enough to cover all statements of the features and the *Base* code (cf. Table 5). This issue explains the lower recall. The feature that has higher coverage with test cases is *Cognitive* (14% of statements covered), which explains the higher recall compared to the other features. Analyzing the results of trace comparisons, for both manually on GUI and with unit tests, we can see worse results than comparisons at the line- and method-level. This is due to the code of features being annotated in an undisciplined way. Such issue requires a very high code coverage to have precise traces at the refinement level of the ArgoUML benchmark. Yet, if the feature was exercised in a way that only some of a class's methods were covered, then, the retrieved traces will contain the refinement of the covered methods. However, in some cases, an entire class is part of a feature, which leads to traces to the class without any refinement. This is an example of why some false-negative and false-positive traces are retrieved.

In summary, the three systems have some false positives in the results related to imports. Since exercising features do not retrieve imports, we considered all imports as part of all features. Another cause of false positives is due to methods that are executed when the application is initialized, which contains all features of the systems or are executed automatically when a specific feature is executed. For instance, the Notepad system has some false positive due to some lines of feature *Undo* being automatically executed when exercising the feature *Paste*. In the Sudoku system, for instance, some lines of feature *States* are executed in the constructor method of a class implemented for managing the boarder of the game. Thus, independent of which feature is executed those lines are always executed. For instance, with ArgoUML, some false-positive traces have been retrieved due to the original scenario contains some code of features *UseCaseDiagram* and *StateDiagram* being exercised by default when the application launches. Also, some traces from the runtime monitoring are missing lines of methods, even the methods are completely part of a specific feature. Depending on what is exercised, just some lines of a method will be executed. We also have some false negatives and positives in the three systems related to field declaration inside classes. Therefore, in our evaluation study, we observed that obtaining high precision and recall is challenging. Even more in a single, complex, and large system. Because executing only and all implementation of a specific feature and the *Base* code can be infeasible. We thus believe our hybrid FLT can be useful as a start point to re-engineering a single system into an SPL, as we were able to locate correctly most of the variable source code of the target systems.

RQ: How effective is our technique for locating features of a single system? *From the results, we can see that our technique for locating features might not be effective in large systems as in small ones, mainly when most of the lines of source code belong to common implementations, i.e. the feature Base. In our study, this leads to many false negatives in ArgoUML. However, we could obtain higher precision for each feature of all the subject systems, varying from 58% to 81% for Sudoku, 74%-88% for Notepad, and 98%-99% for ArgoUML (at the line-level). Furthermore, in comparison with the results of existing techniques, our hybrid FLT reached better results on average. Hence, we can argue that our technique performs better for supporting the re-engineering of a single system into an SPL when compared to previous work [9, 27].*

5 THREATS TO VALIDITY

Regarding the systems we chose to evaluate our FLT, they are public systems and used in previous works. This enables further work comparisons, which is currently difficult from existing published works. According to Razzaq et al. [31], only 27% of the existing FLTs up to 2015 could be reproduced from the published material, being difficult to make a cross-comparison between FLTs. Furthermore, the three systems have different sizes and particularities that can show our technique efficiency to locate features in different systems domain and sizes. Yet, one of the systems we chose is an open-source and complex system with a benchmark available and there are not many studies of SPLs proposing benchmarks. Usually, ground-truths are coarse-grained, i.e., at the level of files, and for more realistic scenarios, the granularity must be at least at the method-level [31]. Thus, the ArgoUML benchmark has an appropriate granularity, as it is at the level of source code statements instead of the method-level [24]. Furthermore, previous work already published used only the ArgoUML benchmark as a subject system to evaluate their technique [1, 9, 27, 28]. Besides ArgoUML being one of the most often systems used to evaluate FLTs, it is also extensively used in the extractive SPL community research [3, 31]. In addition to this, using the ArgoUML system for feature location is challenging as it has several cross-cutting features, i.e., a set of

source code elements shared by more than one feature. Thereby, using the ArgoUML system, makes our work challenging and comparable with already existing ones with a very often used system for evaluation of FLTs, which is very important to take into account. Because the performance of FLTs depends not only on the FLT itself but also on the system/benchmark characteristics.

6 RELATED WORK

Feature location has become increasingly popular since the pioneered work by Wilde and Scully [34]. Since then, numerous dynamic analysis techniques have been proposed in the literature to aid program comprehension [6]. Wilde and Scully [34] developed the Software Reconnaissance tool, which works analyzing the execution traces of test cases by a coverage analyzer. The method can help to find components to a particular feature, however, it does not retrieve all the components that belong to a feature.

Static FLTs have proven to be effective and reached higher values of precision and recall as shown in our previous work [27]. However, worse results were obtained when a few variants exist, as this static technique is based on the comparison of commonalities and differences of source code of system variants. Hence, this efficiency is influenced by the number of variants used to locate features. Another static feature location applied in ArgoUML SPL is presented by AL-Msie'deen et al [1], which relies on the identification of the implementation of features among object-oriented elements of the source code. They applied the technique to a collection of 10 ArgoUML variants. Even, their results show that all of the features were identified, they did not determine precisely each feature implementation.

A popular choice for feature location is the dynamic analysis because it can map the execution of feature code to traces [11]. However, this kind of analysis has also some limitations as the traces collection may overhead the system's execution, and some executions may not invoke all of the code that is relevant to the feature and/or can invoke code of features running in the background. Hence, some feature implementation will not be located, and/or irrelevant traces will be wrongly retrieved. Eisenberg and Volder [12] used JUnit tests to collect traces. Their tool first collects the execution trace, generates the calls, and then uses heuristics to rank methods in relation to what extent a method is relevant to a feature. Their technique works well when the input quality is high, which depends on the coverage of the test suite with respect to the feature under consideration; and on the partitioning of the test suite that is under the developer's control. Their branch coverage was about 86-90% and line coverage ranges from 66-80%. However, their technique is more coarse-grained than ours, considering up the method-level. Yet, Eisenberg and Volder's evaluation is not based on traditional metrics [31]. Walkinshaw et al. [33] technique is based on call graphs and showed promising results. However, this technique requires the identification of methods that have a key role in the execution of a particular feature code, which requires the system documentation, or developer knowledge of the system.

Regarding textual techniques, Cruz et al. [9] evaluated three different strategies based on textual IR also using the ArgoUML benchmark [24]. Their results show the Latent Semantic Indexing strategy (reach an average of precision and recall of $\approx$ 16% and $\approx$ 19%, respectively) is better than Paragraph Vectors and Latent Dirichlet Allocation. However, they explicitly show and mention that the obtained values from the metrics are not high, and they only consider the method-level of granularity. One of the causes is because textual IR techniques require a high number of terms related to the features, which is not always the case as with ArgoUML scenarios, and are challenging scenarios for this kind of strategy. They also mention the need of proposing innovative FLTs as extractive SPL adoption. They suggest the use of hybrid techniques as a future direction to obtain better results.

7 CONCLUSIONS AND FUTURE WORK

We believe our technique can help developers to save time and effort when migrating a single system to an SPL, mainly in smaller systems. For larger systems, even most of the base code may not be retrieved, we believe it can help to find the variable code, i.e., most of the code of the system's features. In this work, we also show that existing FLTs are limited to retrieve traces at coarse-levels of granularity, mainly when just a single system is used for locating features. With our hybrid technique for dynamically tracing features and refining them with static analysis, we could reach higher precision and recall in the smaller systems and higher precision in the largest system. To get rid of part of the wrong traces we would need an expert user of the system to execute the scenarios to obtain the traces more complete and precise as possible. Our technique depends on the proper monitoring of traces to obtain higher recall and eliminating traces that belong to the common code to guarantee higher precision. We thus can infer that the results can be improved if the exercise of features is optimized.

By this work, we also want to aware the research community of the need of making available more ground truth with a fine-level of granularity. Besides it, it is essential that future FLTs provide evaluation using the common metrics, such as precision and recall because many FLTs exist, but they use different metrics to evaluate their technique. This diversity of evaluation limits their comparison, and then, it becomes a hard task for the developers to choose which technique is more suitable for applying the software engineering tasks. We would like to reinforce that future work considers using systems with ground truth available, common metrics, and a technique able to locate features at the statement-level and able to reach higher results with only a single system.

Saving time and effort are the main purpose of using FLTs [4, 26]. Thus, future work can be directed to analyze the effort for executing scenarios, and evaluate our hybrid FLT of how much time and effort it would be still required by developers to complete the product after applying the technique.

ACKNOWLEDGMENTS

This research was funded by the LIT Secure and Correct Systems Lab; the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184; CNPq, grant no. 408356/2018-9 and FAPPR, grant no. 51435.

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 *Safe and Secure Software Reuse*, John Favaro and Maurizio Morisio (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 302–307.

[2] Giuliano Antoniol and Yann-Gael Gueheneuc. 2006. Feature Identification: An Epidemiological Metaphor. *IEEE Transactions on Software Engineering* 32, 9 (2006), 627–641. https://doi.org/10.1109/TSE.2006.88

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

[4] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2* (Florence, Italy) *(SPLC '14)*. Association for Computing Machinery, New York, NY, USA, 52–59. https://doi.org/10.1145/2647908.2655967

[5] Brendan Cleary, Chris Exton, Jim Buckley, and Michael English. 2009. An Empirical Analysis of Information Retrieval Based Concept Location Techniques in Software Comprehension. *Empirical Software Engineering* 14, 1 (Feb. 2009), 93–130. https://doi.org/10.1007/s10664-008-9095-3

[6] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. 2009. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[7] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. IEEE Computer Society, USA, 191–200.

[8] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. 2011. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *15th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, USA, 191–200. https://doi.org/10.1109/csmr.2011.25

[9] Daniel Cruz, Eduardo Figueiredo, and Jabier Martinez. 2019. A Literature Review and Comparison of Three Feature Location Techniques Using ArgoUML-SPL. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems* (Leuven, Belgium) *(VAMOS '19)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/3302333.3302343

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

[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. https://doi.org/10.1002/smr.567

[12] A.D. Eisenberg and K. De Volder. 2005. Dynamic feature traces: finding features in unfamiliar code. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE, USA, 1–10. https://doi.org/10.1109/icsm.2005.42

[13] Wolfram Fenske, Thomas Thüm, and Gunter Saake. 2014. A Taxonomy of Software Product Line Reengineering. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems* (Sophia Antipolis, France) *(VaMoS '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2556624.2556643

[14] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, USA, 391–400. https://doi.org/10.1109/ICSME.2014.61

[15] Stefan Fischer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2018. Towards a Fault-Detection Benchmark for Evaluating Software Product Line Testing Approaches. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 2034–2041. https://doi.org/10.1145/3167132.3167350

[16] E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Automatically capturing source code context of NL-queries for software maintenance and reuse. In *31st International Conference on Software Engineering* (Vancouver, Canada). IEEE Computer Society, USA, 232–242. https://doi.org/10.1109/ICSE.2009.5070524

[17] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) feasibility study*. Technical Report. CMU/SEI-90-TR-21, SEI, CMU. 148 pages.

[18] Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. 2010. Reducing Configurations to Monitor in a Software Product Line. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 285–299.

[19] Jongwook Kim, Don Batory, and Danny Dig. 2017. Refactoring Java Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A* (Sevilla, Spain) *(SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/3106195.3106201

[20] 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)*. Association for Computing Machinery, New York, NY, USA, 86–95. https://doi.org/10.1145/1101908.1101923

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

[22] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. 2007. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 234–243. https://doi.org/10.1145/1321631.1321667

[23] Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Cambridge university press, Cambridge, England.

[24] 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)*. Association for Computing Machinery, New York, NY, USA, 257–263. https://doi.org/10.1145/3233027.3236402

[25] Jabier Martinez, Daniele Wolfart, Wesley K. G. Assunção, and Eduardo Figueiredo. 2020. Insights on Software Product Line Extraction Processes: ArgoUML to ArgoUML-SPL Revisited. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A* (Montreal, Quebec, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/3382025.3414971

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

[27] Gabriela Karoline Michelon, Lukas Linsbauer, Wesley K. G. Assunção, and Alexander Egyed. 2019. Comparison-Based Feature Location in ArgoUML Variants. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 93–97. https://doi.org/10.1145/3336294.3342360

[28] Richard Müller and Ulrich Eisenecker. 2019. A Graph-Based Feature Location Approach Using Set Theory. In *23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 88–92. https://doi.org/10.1145/3336294.3342358

[29] Maksym Petrenko, Václav Rajlich, and Radu Vanciu. 2008. Partial Domain Comprehension in Software Evolution and Maintenance. In *16th IEEE International Conference on Program Comprehension (ICPC '08)*. IEEE Computer Society, USA, 13–22. https://doi.org/10.1109/ICPC.2008.14

[30] 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 Transactions on Software Engineering* 33, 6 (June 2007), 420–432. https://doi.org/10.1109/TSE.2007.1016

[31] Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. 2019. The State of Empirical Evaluation in Static Feature Location. *ACM Transactions on Software Engineering and Methodology* 28, 1 (Feb. 2019), 1–58. https://doi.org/10.1145/3280988

[32] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2007. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (Vancouver, British Columbia, Canada) *(AOSD '07)*. Association for Computing Machinery, New York, NY, USA, 212–224. https://doi.org/10.1145/1218563.1218587

[33] Neil Walkinshaw, Marc Roper, and Murray Wood. 2007. Feature Location and Extraction using Landmarks and Barriers. In *2007 IEEE International Conference on Software Maintenance*. IEEE, USA, 54–63. https://doi.org/10.1109/icsm.2007.4362618

[34] Norman Wilde and Michael C. Scully. 1995. Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice* 7, 1 (Jan. 1995), 49–62. https://doi.org/10.1002/smr.4360070105