Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Spectrum-based feature localization for families of systems[☆]

Gabriela K. Michelon[a,*], Jabier Martinez[b], Bruno Sotto-Mayor[c], Aitor Arrieta[d], Wesley K.G. Assunção[a], Rui Abreu[e], Alexander Egyed[a]

[a] Johannes Kepler University Linz, Linz, Austria
[b] Tecnalia, Basque Research and Technology Alliance (BRTA), Spain
[c] Ben Gurion University of the Negev, Be'er Sheva, Israel
[d] University of Mondragon, Mondragon, Spain
[e] FEUP and INESC-ID, Porto, Portugal

A R T I C L E I N F O

Article history:
Received 19 February 2022
Received in revised form 26 September 2022
Accepted 30 September 2022
Available online 8 October 2022

Keywords:
Feature localization
Spectrum-based localization
Clone-and-own
Software product lines

A B S T R A C T

In large code bases, locating the elements that implement concrete features of a system is challenging. This information is paramount for maintenance and evolution tasks, although not always explicitly available. In this work, motivated by the needs of locating features as a first step for feature-based Software Product Line adoption, we propose a solution for improving the performance of existing approaches. For this, relying on an automatic feature localization approach to locate features in single-systems, we propose approaches to deal with feature localization in the context of families of systems, e.g., variants created through opportunistic reuse such as clone-and-own. Our feature localization approaches are built on top of Spectrum-based feature localization (SBFL) techniques, supporting both dynamic feature localization (i.e., using execution traces as input) and static feature localization (i.e., relying on the structural decomposition of the variants' implementation). Concretely, we provide (i) a characterization of different settings for dynamic SBFL in single systems, (ii) an approach to improve accuracy of dynamic SBFL for families of systems, and (iii) an approach to use SBFL as a static feature localization technique for families of systems. The proposed approaches are evaluated using the consolidated ArgoUML SPL feature localization benchmark. The results suggest that some settings of SBFL favor precision such as using the ranking metrics Wong2, Ochiai2, or Tarantula with high threshold values, while most of the ranking metrics with low thresholds favor recall. The approach to use information from variants increase the precision of dynamic SBFL while maintaining recall even with few number of variants, namely two or three. Finally, the static SBFL approach performs equally in terms of accuracy to other state-of-the-art approaches, such as Formal Concept Analysis and Interdependent Elements.

1. Introduction

The traceability between features and implementation elements is of paramount importance for the maintenance and evolution of systems (Spanoudakis and Zisman, 2005). Feature localization (FL) is the activity to recover this traceability in those cases where it is unknown or it is highly implicit (Dit et al., 2013). When dealing with families of systems, it is even more important as some features might be optional and this traceability is mandatory to derive variants by including or excluding certain features (Assunção et al., 2017). In Software Product Line (SPL) engineering research, and more concretely in extractive approaches for SPL adoption (Krueger, 2001), this activity is considered especially useful when dealing with product variants created in a clone-and-own fashion. FL supports the migration of the family of systems to a more systematic reuse approach, such as feature-oriented SPLs (Apel et al., 2013). FL is thus key in the *detection* phase in re-engineering legacy applications into SPLs (Assunção et al., 2017). For FL, in this context, it is usually assumed that the feature names are known and, for each variant, it is known the features it includes (Fischer et al., 2014), but not which parts of the code correspond to each feature. The goal of FL is to determine the traceability between features and code.

Manual FL is considered time-consuming and error-prone (Martinez et al., 2020), thus several approaches have been proposed trying to automate this process (Rubin and Chechik, 2013b). Recently, there is interest in trying to make those approaches more comparable through the use of benchmarks (Strüber et al., 2019), but overall, the performance of automatic approaches need

☆ Editor: Raffaela Mirandola.
* Corresponding author.
E-mail address: gabriela.michelon@jku.at (G.K. Michelon).

https://doi.org/10.1016/j.jss.2022.111532

to be significantly improved for industrial uptake (Razzaq et al., 2020). In this work, we deal with FL "after the fact" instead of a solution that proactively records and maintains feature traces during development (Ji et al., 2015; Bittner et al., 2021). Unfortunately, these advanced approaches are not yet mainstream, so FL should be considered.

We present a novel FL approach for families of systems by adapting a technique from the software debugging field, namely Spectrum-based fault localization (Wong et al., 2016). This technique is based on the intuition that covered and uncovered parts of tested code can help to indicate the probability of specific lines of code having faults. Instead of using this technique to locate bugs in the code, we use it to locate features of an envisioned SPL, resulting in the Spectrum-based feature localization (SBFL) technique. While there are existing works for SBFL in single systems, to the best of our knowledge, this is the first work that explores the technique with dynamic analysis in the context of feature location in families of systems. SBFL belongs to the category of *dynamic* FL techniques as it uses execution traces, however, in this work we show its application also in *static* mode. Static FL does not use execution traces but directly the implementation elements (e.g., relevant abstract syntax tree elements such as classes or methods) and usually, for the case of families of systems, n-way comparison-based techniques for these elements.

To evaluate our proposed FL approach, we conduct a study to provide answers to the following research questions (RQs):

- ***RQ1**: Which are the best configuration alternatives for the SBFL technique in dynamic FL for single systems?* This question is relevant when dealing with monolithic systems that must be decomposed into features to create families of systems. At the same time, this question provides the basis to select a configuration that can be later used for approaches dealing with families of systems.
- ***RQ2**: To which extent can the presence of variants increase the accuracy of dynamic SBFL?* Variants created through clone-and-own are an important source of information for FL. Therefore, it is reasonable to exploit the commonality and variability of the variants to refine dynamic SBFL results. This RQ focuses on investigating how beneficial the use of variants is for SBFL.
- ***RQ3**: How can SBFL be leveraged for static n-way comparison of variants and what is its accuracy and performance?* While SBFL has been used in dynamic FL, certain SBFL concepts like the spectra have similarities with formal contexts, which are used for static FL in families of systems through Formal Concept Analysis (FCA) (e.g., Al-Msie'deen et al. (2013), Shatnawi et al. (2016)). Thus, this RQ aims to compare the novel SBFL technique results with two other state-of-the-art baseline static FL techniques.

This work extends our previous work (Michelon et al., 2021b), that focused exclusively on SBFL in single systems (RQ1). In that work, we provided the results of the spectrum-based technique for FL in the original ArgoUML version of the ArgoUML SPL FL benchmark (Martinez et al., 2018a). The ArgoUML SPL FL benchmark provides FL challenges for different scenarios, namely different number of variants and degree of similarity among them. In this work, in addition to more empirical results and analysis, we evaluate our approach considering families of systems, i.e., with the existence of variants of the system, and feature interactions to some extent.

Concretely, the contributions of this work are:

- **Dynamic SBFL in single systems**: A characterization of 33 different SBFL ranking metrics with 10 threshold values for each one in the original version of ArgoUML (RQ1). The results suggest that some configurations of the technique (i.e., combination of ranking metric and threshold) favor precision (e.g., Wong2 0.9 to 1, Ochiai2 0.7 to 1, or Tarantula 0.9 to 1) while others favor recall (e.g., Hamming 0.1 among many others with low thresholds).
- **Dynamic SBFL improvements for families of systems**: A hybrid method to use information from variants that lead to better performance, namely an increase in the precision of dynamic SBFL while maintaining recall (RQ2). The increase is visible even with few number of variants (i.e., two or three) for scenarios in the ArgoUML SPL FL benchmark.
- **Static SBFL for families of systems**: In addition to experimenting SBFL for dynamic FL, our work also presents how SBFL can be applied for static FL. We show that certain configurations of ranking metrics and thresholds of SBFL (e.g., Wong2 with 1.0) can be used for static FL of variants (RQ3). The results show that it produces the same locations as other state-of-the-art static algorithms, such as FCA (Al-Msie'deen et al., 2013; Shatnawi et al., 2016) or the Interdependent Elements (IE) algorithm (Ziadi et al., 2012).

Additional contributions related to the easy uptake of the approach, reproducibility, and ease of extension:

- **Dataset**: New execution traces were created for RQ2 and RQ3 experiments using a standard format based on the Java Code Coverage Library (JaCoCo)[1] coverage reports. This makes the approach more easily accessible compared to our previous work (Michelon et al., 2021b), where an ad-hoc pipeline to run the experiments and an ad-hoc format of the execution traces were used.
- **Tooling**: The SBFL technique is integrated in the Bottom-Up Technologies for Reuse framework (BUT4Reuse)[2] (Martinez et al., 2015) and can be used through its user interface as well as programmatically through its Java API, as we did for the experimentation pipeline related to RQ2 and RQ3.

The mentioned dataset and the experimentation pipelines of this work are publicly available.[3]

This paper is structured as follows: Section 2 motivates our work with an illustrative example, which will be used later as a running example. Section 3 presents related work. Then, Section 4 details our proposed SBFL approaches, and Section 5 presents their evaluation in the case study. Section 6 presents the results and discussion. Section 7 points the threats to validity and Section 8 concludes the paper and outlines further perspectives.

2. Motivating and running example

As a motivating and running example, we use one illustrative family of drawing application variants (Fischer et al., 2014). An imaginary company started with one product to draw lines and wipe. Then, a new customer required the capability to set the color of the lines and the removal of the wipe functionality. Product 1 was then reused in a clone-and-own fashion to tailor this new Product 2. Later, another customer wanted to include rectangles, also with color, so a new Product 3 was created from Product 2. Table 1 shows the features implemented in each variant. Therefore, the company needs to maintain three different variants for each customer, with the associated costs to propagate shared bug fixes or enhancements. They realized that

[1] https://www.jacoco.org/
[2] https://but4reuse.github.io/
[3] https://github.com/jabiercoding/spectrum_based_localization

Table 1
Draw application products and their features.

Products	Features				
	BASE	LINE	RECT	COLOR	WIPE
Product 1	✓	✓			✓
Product 2	✓	✓		✓	
Product 3	✓	✓	✓	✓	

this clone-and-own approach did not scale up for them as this variant management approach introduced significant technical debt for the maintenance and evolution of the product family as a whole (Wolfart et al., 2021).

While this is a simplified illustration of the issues, plenty of industrial cases are cataloged in similar situations with a will to extract an SPL (Martinez et al., 2017; Wolfart et al., 2021). In our drawing applications case, traceability between features and source code was implicit and it was somehow lost. Recovering this traceability can help to annotate the source code to create an annotative-based SPL and having a unique variability-rich application (e.g., with Munge (Sonatype, 2011), or Antenna (Pleumann et al., 2010)), or to extract reusable assets in a source code compositional approach (e.g., with FeatureHouse source code superimposition (Apel et al., 2009)). This transformation phase is out of the scope of this work, as we are focusing on FL for the detection phase (Assunção et al., 2017). However, the traceability is paramount to support the transformation. Based on that, developers need certain automation with accurate FL results to facilitate their task.

Recapping our illustrative example, the draw application is a family of systems with three Java programs[4] of around 300 lines of code (LoC) implemented in three to four Java files. Listing 1 shows an excerpt of the three files of Product 1. The envisioned scope of the drawing applications family will allow drawing lines or rectangles, only in black or with different colors, and an optional wipe function. The COLOR feature is clearly a cross-cutting feature as it will have interactions with LINE and RECT.

We assume that the information about which features are present in each product is known (Table 1). However, the traceability of features to the Java source code is to be recovered. For instance, in the excerpt of the Main.java file for Product 1, shown in Listing 1a, the fields in lines 3 and 4 (toolPanel and canvas) are part of the BASE feature whereas lineButton and wipeButton fields are for the LINE and WIPE features, respectively. The first two lines within the method initContentPane (10 and 11) are also for the LINE and WIPE features, respectively, whereas the rest of the method is for BASE. The last method of the excerpt belongs completely to BASE. In this small illustrative set of variants, manually performing feature localization might be straightforward. However, it becomes challenging with thousands of Java files and several features as it is the case in complex industrial settings.

Automatic FL approaches relying on n-way comparison of the products are usually set-based, trying to distinguish features based on the intersections of the existing variants. This is illustrated in Fig. 1. For instance, we can observe how WIPE can be distinguished through the intersections, as it is the only feature that is specific to Product 1 (P1 in the figure). Following with the example in Listing 1a, we are able to locate the wipeButton field, and we know that WIPE refines the method initContent-Pane(). However, these approaches might easily fail to recover some of the features or feature interactions. For instance, BASE and LINE are always present together in the three products, then

[4] V1, V2 and V3 at https://github.com/jku-isse/dpl/tree/master/example/implementations

```
 1 public class Main extends JFrame {
 2
 3  protected JPanel toolPanel = new JPanel();
 4  protected Canvas canvas = new Canvas();
 5
 6  JButton lineButton;
 7  JButton wipeButton;
 8
 9  public void initContentPane() {
10   toolPanel.add(lineButton);
11   toolPanel.add(wipeButton);
12   contentPane.add(toolPanel, BorderLayout.WEST);
13   contentPane.add(canvas, BorderLayout.CENTER);
14  }
15
16  public void initLayout() {
17   contentPane = getContentPane();
18   contentPane.setLayout(new BorderLayout());
19   toolPanel.setLayout(new
        BoxLayout(toolPanel,BoxLayout.Y_AXIS));
20  }
21   ...
22 }
```

(a) Excerpt of Main.java content in Product 1.

```
public class Canvas extends JComponent
                implements MouseListener,
                     MouseMotionListener {
  ...
 public void mousePressedLine(MouseEvent e) {
  if (newLine == null) {
   start = new Point(e.getX(), e.getY());
   newLine = new Line (start);
   lines.add(newLine);
  }
 }
  ...
}
```

(b) Excerpt of Canvas.java content in Product 1.

```
public class Line {
  ...
 private Point startPoint, endPoint ;
  ...
}
```

(c) Excerpt of Line.java content in Product 1.

Listing 1: Example of source code of Product 1 of the drawing applications family.

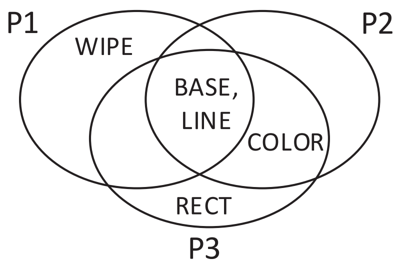


Fig. 1. Intersections of the products P1, P2 and P3.

it will not be possible, for these techniques, to directly distinguish them and locate the source code that is specific for the LINE feature. The number of products and their diversity is recognized as a sensible factor for the success of feature localization techniques in families of systems (Michelon et al., 2019; Martinez et al., 2018b).

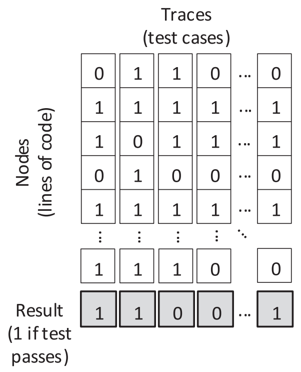


Fig. 2. Spectrum-based Fault localization.

Other existing techniques, mainly used for single systems, can be combined with these set-based techniques. This is the case of approaches based on text analysis. For instance, trying to locate the `LINE` feature based on the name and description of the feature, we can obtain that the `Line.java` class might be associated to the `LINE` feature, or the method `wipe()` in `Canvas.java` can be associated to the `WIPE` feature. In this work we do not consider text-based techniques (Razzaq et al., 2018; Thomas et al., 2013; Mahmoud and Bradshaw, 2015) as we rely on dynamic and static feature localization techniques (Dit et al., 2013; Robillard, 2008; Robillard and Murphy, 2002; Rubin and Chechik, 2012; Wilde et al., 2001; Cornelissen et al., 2009).

3. Background and related work

The following sections introduce the basic concept for our approach and describes the related work to our study.

3.1. Spectrum-based localization techniques

Spectrum-based localization (SBL) techniques are traditionally used for fault localization (Wong et al., 2016). They use a spectrum where each trace represents a test case and each node represents the lines of code of the program under analysis. As depicted in Fig. 2, SBL assigns a 1 in the spectrum when a line of code (row) is executed at least once for a particular test case (column). The last row of the spectrum describes the result of the tests; it assigns 1 when the test passes and 0 when it fails.

Several *ranking metrics* (Wong et al., 2016) have been proposed that use a spectrum to construct a ranking of suspiciousness. This ranking points developers to the lines of code that are potentially causing the issue, restricting the search to a segment of the source code that is relevant for examination or debugging. The ranking metrics are defined by formulas that use the number of executed and non-executed lines of code that are passing or failing a test. In this work, we represent the lines that were executed as e_p and e_f, and those that were not as n_p and n_f.

Wong et al. (2016) survey a broad catalog of SBL ranking metrics. A few examples of the simplest metrics are Wong1 (e_f), Wong2 ($e_f - e_p$), or Hamming ($e_f + n_p$). Moreover, well established ranking metrics are Ochiai (Abreu et al., 2007) or Tarantula (Jones et al., 2002), formulated as follows:

$$Ochiai = \frac{e_f}{\sqrt{(e_f + n_f) \times (e_f + e_p)}} \quad (1)$$

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}} \quad (2)$$

A calculated ranking metric assigns to each node a continuous normalized value between 0 (apparently completely unrelated) and 1 (apparently faulty). Therefore, the values within the range will have to be analyzed under a certain user-defined *threshold*.

Beyond fault localization, SBL has also been used in the context of mapping program features to code, known as SBFL. Wilde and Scully (Wilde and Scully, 1995) introduced the idea of using tests to map features to program components. Their approach builds program spectra based on the statement coverage of every test run and then applies a simple heuristic. The heuristic assigns a component to a feature if it is only used in the runs that use that feature. Eisenbarth et al. (2003) later proposed a set of advanced heuristics that assign six fine-grained categories to the program components based on the association level between the component and the feature, ranging from *specific* to *irrelevant*.

However, these approaches only provide a discrete range heuristics whose coverage items are categorized into a fixed set of categories. Therefore, novel approaches were introduced that apply, instead, a continuous range heuristic based on the adoption of the Tarantula (Malburg et al., 2014) and Ochiai (Perez and Abreu, 2014; Malburg et al., 2014) heuristics, which are well-known from the spectrum-based fault prediction domain. Therefore, they calculate a numerical value between 0 and 1 that describes the likelihood of the association between components and feature, overcoming the inaccuracy of a categorical association.

Perez and Abreu (2014) introduced Spectrum-based Feature Comprehension (SFC), which expanded the spectrum feature localization scope to the task of acquiring program comprehension. In particular, beyond improving feature localization using the Ochiai coefficient, they leverage information-foraging theory to improve the intuitiveness of the feature localization diagnostic report. Furthermore, they extend this approach by allowing feature localization through user participation (Perez and Abreu, 2016) and implement these techniques into a tool-set (Castro et al., 2019) that reports the program analysis to the user.

These pieces of work on SBFL are complementary to ours regarding RQ1 as we explore different settings for dynamic SBFL in single systems, but not for RQ2 and RQ3.

3.2. Dynamic analysis in hybrid techniques

This section introduces the basic concepts of dynamic analysis and related work of hybrid techniques relevant for RQ2.

Dynamic techniques started to be used for debugging, testing, and profiling, typically by analyzing program execution through interpretation (e.g., using the Virtual Machine in Java) or instrumentation (Cornelissen et al., 2009). Since 1995 feature localization received more attention, where dynamic analysis has been proposed to collect information about a program at runtime (Rubin and Chechik, 2013b). One of the first dynamic techniques was proposed by Wilde and Scully (1995). Their technique is based on the execution of different test cases of a single system by using a test coverage monitor. It was evaluated on a single system with 15,000 LoC containing test cases that exercise features. The results show high precision to identify pieces of the system that are unique to a particular feature, but show low precision in larger sets of code that are for general purposes and exist in multiple features. Their technique requires heavy user involvement, as the user has to specify the set of test cases invoking each feature.

Most of the existing dynamic techniques were used combining other techniques, such as static and information retrieval analysis (Eisenbarth et al., 2003; Koschke and Quante, 2005; Asadi et al., 2010; Rohatgi et al., 2008). On the one hand, dynamic analysis can collect precise information about the program execution but yields many false negative results. One of the reasons is because the dynamic analysis can detect only functional features (Rubin and Chechik, 2013b). On the other hand, for example, static analyses yield many false positives but can locate any type of features (Rubin and Chechik, 2013b; Koschke and Quante, 2005). Therefore, hybrid techniques combining dynamic and static analyses have been proposed to try to complement each other (Rubin and Chechik, 2013b).

Existing hybrid approaches consider only traces of a single system and at coarse levels of granularity (generally the method-level) (Michelon et al., 2021a). The hybrid approach proposed by Michelon et al. (2021a), which we refer to subsequently as Michelon et al. Hybrid (Michelon et al., 2021a) for later comparison, combines a dynamic analysis with a static analysis of the source code. The dynamic analysis considers scenarios from features manually exercised or from traces obtained from coverage tests. The static analysis obtains overlapping traces between features and thus can filter some execution traces. Their results show high precision but not high recall for the ArgoUML system. Further, they also considered as true positives the source code that is common for all features (i.e., the base of ArgoUML). A way to improve the recall of their hybrid approach would be to get rid of the common core of the system, which is part of the many false negatives. Also, different execution scenarios that retrieve the execution traces could improve the results.

Aiming to improve the performance of existing hybrid feature location techniques, we thus combined SBFL with execution traces of different variants/products of a system to investigate how beneficial is the use of variants to obtain higher precision and recall. Another aspect we take into account in RQ2 is to analyze SBFL with dynamic analysis of system variants at fine granularity level. The ArgoUML SPL FL benchmark (Martinez et al., 2018b) thus enables us to evaluate our approach and compare the results of RQ2 with the existing approach from Michelon et al. Hybrid (Michelon et al., 2021a).

3.3. Static comparison-based techniques

This section presents background information and related works relevant for RQ3.

Formal concept analysis (FCA). FCA is a mathematical method (Ganter and Wille, 1999) that analyses attributes to group elements that share common attributes. Fig. 3 shows two examples to understand the input and output of FCA for statically analyzing variants. In Fig. 3(a), we show the *formal context* that is created for our running example. This formal context captures a mapping between a set of attributes (the source code elements in our case), and objects (the product variants). With this information, FCA creates a *concept lattice* as shown in Fig. 3(b). The elements in the different concepts are disjoint. For instance, `Concept 0` are the elements that are common to all products, `Concept 3` groups the elements that are specific to Product 1, and `Concept 2` groups the elements that are common to Product 2 and Product 3, but not present in Product 1. The arrows in this concept lattice can be followed to know which are the concepts (features in our context) for a specific product. For instance, Product 1 (bottom-left in Fig. 3(b)) is based on `Concepts 3, 1, and 0`. In the same way, Product 2 is based on `Concepts 4, 1, 2, and 0`.

For the FL task at hand, each concept with at least one element is a block potentially related to a feature. FCA has been used in that direction in studies such as the ones by Al-Msie'deen et al. (2013) and Shatnawi et al. (2016). In BUT4Reuse, FCA is a technique for block identification implemented with the Galatea FCA library[5] (Falleri, 2009).

[5] https://github.com/jrfaller/galatea

Interdependent elements (IE). IE is a block identification technique from a set of variants proposed by Ziadi et al. (2012). This technique is based on calculating *interdependent elements*, which create element groups that are feature candidates. IE are defined as follows: given a set A of artifacts that we want to compare, two elements (of artifacts of A) e_1 and e_2 are *interdependent* if and only if they belong to exactly the same artifacts of A. Therefore, e_1 and e_2 are interdependent if the two conditions in Eq. (3) are fulfilled.

$$\begin{aligned} &\exists a \in A \quad e_1 \in a \wedge e_2 \in a \\ &\forall a \in A \quad e_1 \in a \Leftrightarrow e_2 \in a \end{aligned} \tag{3}$$

The consequence of these rules is that the intersections of the variants are distinguished. For example, in our running example, similarly to FCA, we will have one block with the elements that are common to all products, three blocks for the elements that are specific to each product, and, for instance, another block grouping the elements that are common to Product 2 and Product 3, but not present in Product 1. This technique was integrated in ExtractorPL (Ziadi et al., 2014), and later in BUT4Reuse (Martinez et al., 2015) as a block identification technique. It has been also integrated for the specific case of feature candidates identification in Cyber–Physical Production Systems variants (Meixner et al., 2020).

Strict feature-specific (SFS). Both FCA and IE identify blocks that are potential features. This way, they are useful for feature identification when the feature list is not completely known beforehand. Notice that both FCA and IE work at the level of the variants, but no information about the features is embedded. Then, these blocks should be manually analyzed. However, to fully automate FL, a final step is needed to assign features to these blocks. SFS is a basic feature assignation approach that takes as input a set of known features and blocks, and apply the following two rules. A feature is located in a block when:

- The block *always* appears in the artifacts that implements this feature and,
- The block *never* appears in any artifact that does not implement this feature.

For example, if we take the block `Concept 2` from Fig. 3(b), and we want to check if it corresponds to the `COLOR` feature, we find that it always appears in the products where `COLOR` is present (Product 2 and Product 3), and never where it was not there (Product 1). So this block is assigned to `COLOR` using SFS. But if we take the same block `Concept 2` to check if it corresponds to `RECT`, we have that it always appears in the products where `RECT` is present (Product 3), but it also appears in Product 2 which has not the `RECT` feature. So the second rule is not satisfied. These principles are similar to locating distinguishing features using diff sets (Rubin and Chechik, 2012).

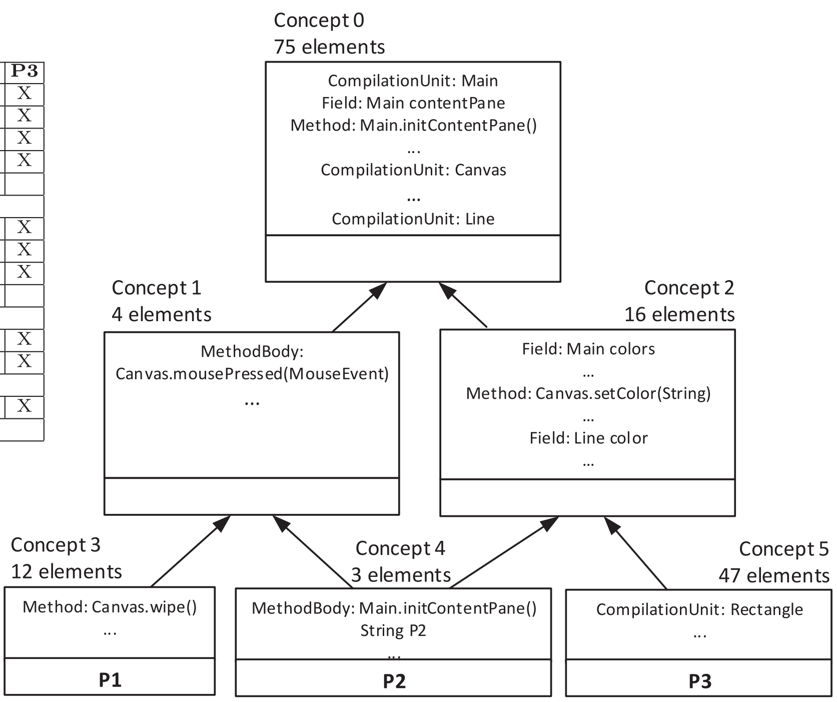
In this work we refer to FCA+SFS and IE+SFS as the use of the two different block identification approaches (FCA and IE respectively) followed by the SFS method to assign features to blocks. Regarding tool support, block identification and feature localization are activities that can be seamlessly chained in BUT4Reuse (Martinez et al., 2015).

4. Spectrum-based feature localization approach

This section explains our proposed FL techniques. Section 4.1 presents the abstraction mechanisms of the implementation elements to be located. Then, Section 4.2 presents the dynamic SBFL of single systems and Section 4.3 its enhancement for families of systems. Finally, Section 4.4 presents the static SBFL approach.

	P1	P2	P3
CompilationUnit: Main	X	X	X
Field: Main contentPane	X	X	X
Field: Main colors		X	X
Method: Main.initContentPane()	X	X	X
MethodBody: Main.initContentPane() String P2		X	
...		...	
CompilationUnit: Canvas	X	X	X
MethodBody: Canvas.mousePressed(MouseEvent)	X	X	X
Method: Canvas.setColor(String)		X	X
Method: Canvas.wipe()	X		
...		...	
CompilationUnit: Line	X	X	X
Field: Line color		X	X
...		...	
CompilationUnit: Rectangle			X
...		...	

(a) Formal context: 157 attributes and 3 objects.



(b) Concept lattice obtained from the formal context through FCA.

Fig. 3. Formal Concept Analysis (FCA) illustrated through the draw application running example.

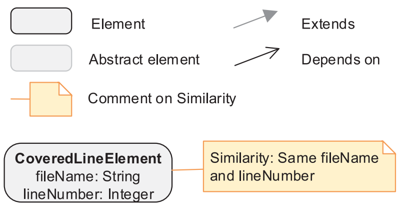
4.1. Elements to be located

Fig. 4 shows the abstractions used in the implementation elements to be located in this work. In Fig. 4(a), the *Covered Line Element* represents an executed line of code, which is defined in terms of the file name that was executed (e.g., `Canvas.java`) and the concrete line number within that file. JaCoCo coverage reports can be exported to XML (examples in our dataset[6]) that can be later adapted to this abstract representation as a set of *Covered Line Elements*. For Product 1 of our running example (Section 2), if we launch the JaCoCo coverage tool, draw a line, and close the drawing application, the JaCoCo XML coverage report will return 82 *Covered Line Elements*.

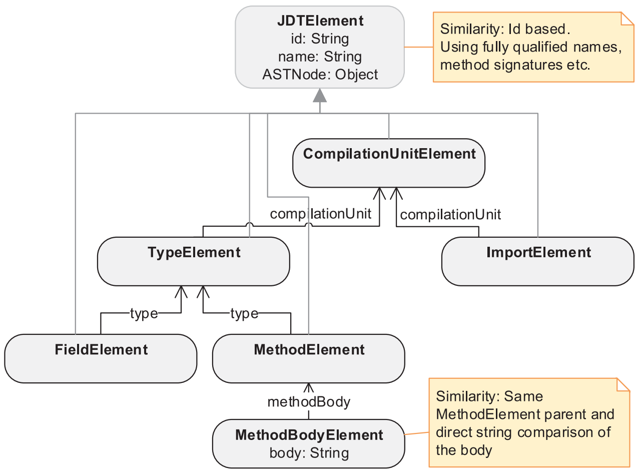
Fig. 4(b) shows elements of abstract syntax trees (ASTs) of source code, which is Java in our case. This abstraction was already used in Fig. 3 to represent the elements in the draw application. The *Compilation Unit* corresponds to Java files. A Java file has Types (usually one) and it can have several Imports. A Type has Fields and Methods, and each Method has a Method Body. The abstractions from *Covered Line Elements* to Java source code syntax tree are implemented as adapters in BUT4Reuse (Martinez et al., 2015). BUT4Reuse together with the adapters allow to seamlessly chain different block identification and FL approaches. The Java adapter was already available, but the JaCoCo XML coverage reports adapter is a contribution of this work and currently available in the tool. Fig. 4(b) shows an excerpt of the concepts of the Java adapter of BUT4Reuse, which is based on JDT[7] (Eclipse Java Development Tools). To be able to count the elements retrieved, we then parsed the XML report files with our implementation.[2] For example, when abstracting the elements of the source code of Product 1, 157 of these types of elements were retrieved with our implemented parser.

6 https://github.com/jabiercoding/spectrum_based_localization/tree/master/plugins/spectrum/execTraces

7 https://www.eclipse.org/jdt/



(a) Source code execution traces in JaCoCo XML coverage reports.



(b) Elements of the Abstract syntax tree structure used to parse the *Covered Line Elements* to Java source code.

Fig. 4. Abstraction of the implementation elements to be located.

In our scenario of families of systems, a similarity metric is needed to know when an element is the same as another element in a different variant. Similarity in software engineering (i.e., the matching strategy among elements) is a difficult problem. We rely on id-based and signature-based similarity, providing a discrete value (boolean). For the source code elements, the "ids" are created through fully qualified names and method signatures. And for the Method Body, we compare the method signature and the text of the body of the method. It is a design decision to not consider statement logic/semantic inside the method. This is challenging in terms of similarity calculation if comparing different source code variants, and we consider that the granularity level of FL at method-level is already sufficient for our context.

A *Covered Line Element* will be the same as another if the other refers to exactly the same file name and line number. However, this only holds for comparing execution traces of single systems. In the case of variants, the files can be changed, e.g., a method of a feature is not in a file because this variant does not have this feature. Thus, this can change the line numbers of the rest of the code in that file. To compare *Covered Line Elements* between variants, in this work, we used our implemented parser to transform *Covered Line Elements* to Java source code elements. This transformer allows knowing if covered elements refer to the same Method or Field initialization, for example.

4.2. SBL for single systems as a dynamic FL technique (RQ1)

Reference to the running example. In dynamic FL, we should exercise a specific feature manually or through a test case, and get the execution traces through source code coverage analysis tools such as JaCoCo. As mentioned before, for FL in Product 1 of our running example, we can exercise LINE by launching the coverage tool and drawing a line returning 82 executed lines. And then, to exercise WIPE, we can launch the coverage tool and we click on the wipe button. This returns 55 executed lines. In large systems (Dit et al., 2013; Couto et al., 2011; Salman et al., 2013; Revelle et al., 2010; Eaddy et al., 2008), full coverage of a feature might not be feasible. However, the intuition is to use these traces to locate feature source code that is highly related to a feature. For these two traces, it is reasonable to think that their intersection (shared covered lines) corresponds to BASE (50 lines) while the others might be specific to LINE (32 lines) and WIPE (5 lines), respectively. Listing 2 shows these concrete lines for WIPE obtained by running our FL implementation to show that the result is precise when validated by a domain expert or when compared with a ground-truth.

```
In Main.java
 public void actionPerformed(ActionEvent e) {
  canvas.wipe();
 }

In Canvas.java
 public void wipe() {
  this.lines.clear();
  this.repaint();
 }
}
```

Listing 2: Excerpt of Main.java content in Product 1.

For Product 1 we do not have automatic test cases, but if they exist and we know which feature is being exercised per test case, this can also be used as feature exercises. Once we have feature execution traces, SBFL can be applied.

Approach. The application of SBFL in single systems was the main contribution of our previous work (Michelon et al., 2021b). We use the SBL for FL using a Spectrum like the one shown in Fig. 5. Compared to fault localization, our traces are executions of the program purposely exercising a given feature (i.e., manual

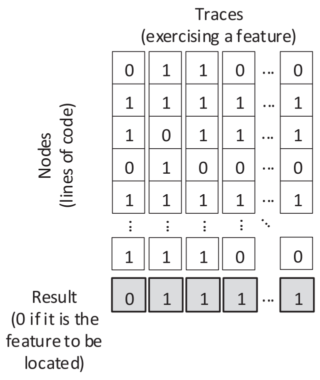


Fig. 5. Dynamic SBFL for single systems.

executions or tests that we know are related to a given feature). The result will be 0 for the traces that belong to the feature that we want to locate, i.e., as if it was the "faulty" code to locate. The ranking obtained when applying one of the ranking metrics will be based on the suspiciousness of each line of code belonging to a feature under analysis.

4.3. Enhancement of dynamic SBFL for families of systems (RQ2)

Reference to the running example. The presence of variants can help us to reason on dynamic SBFL results. For instance, in Listing 1a, the method initContentPane() is an initialization method. Therefore, it is executed for all manual execution traces we do in Product 1. SBFL in single systems will most probably suspect that it is a BASE method. However, it is not complete for BASE as it also has statements related to WIPE and LINE. By looking at Product 2 or Product 3, that does not have WIPE, we will notice it, as the statement toolPanel.add(wipeButton) is missing for this method.

Approach. Once we have the SBFL results for a single system (e.g., a reference variant we use for testing) we transform the *Covered Line Elements* to the JDT Elements that they refer to. This transformer was explained in Section 4.1. Then, from the resulting set of JDT Elements for each feature, the approach is based on:

- Removing all elements in common with variants that do not contain the feature. The intuition is that an element of a feature cannot appear in a variant without that feature.
- Removing all elements that are not present in a variant that contains the feature. The reasoning is that an element of a feature must appear in all variants with that feature.

This will discard, from the results, several elements that are likely not part of the feature. As an illustrative example, imagine that we have variants A, B, and C. Both variants A and B contain feature F1 but variant C does not have F1. Variant A has elements e1, e2 and e3, Variant B has elements e1 and e2, and Variant C has element e1. The results for SBFL in the reference variant A returned that F1 is located in e1, e2, and e3. This is the input for our enhanced technique with the two rules. The first rule will look into variant C (the one without F1) and discard e1 as it appears in C. After that, the second rule will look into the variants with F1, and when reaching variant B, it will be noticed that e3 is not present in this variant. Therefore, e3 will be discarded as well because it does not appear in B that has F1. The refined results for SBFL will be then just e2.

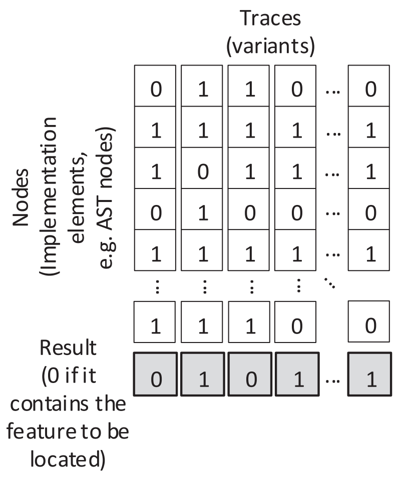


Fig. 6. Static SBFL for families of systems.

4.4. SBFL for families of systems as a comparison-based technique (RQ3)

Reference to the running example. The usage of comparison-based techniques for the running example was already illustrated and discussed in Section 3.3. The running example of a draw application contains four features besides the core of the application (`BASE`): `LINE`, `RECT`, `COLOR`, and `WIPE`. The comparison-based technique is able to map program features to code when comparing the common and variable code among different features products. In this way, a program element existing in a variant containing `LINE` and not present in any other feature variant means that the program element belongs to `LINE`.

Approach. One of the contributions of this work is to show that it is possible to reuse the SBL technique for n-way system comparison. Fig. 6 presents how the spectrum is encoded to achieve it. In comparison to Fig. 5, following the fault localization analogy, the traces are the variants and the nodes are the implementation elements, in this case JDT Elements, and not lines of code. Therefore, the result is 0 in the case that the variant contains the feature (as if the fault was there following the analogy), while in Fig. 5 the result is 0 if it is the feature to be located.

Now, if we create the spectrum and we consider the ranking metric Wong2 ($e_f - e_p$) with the maximum 1.0 as threshold, we will be encoding the same principles in a SBL way. As further explanation, e_f is rewarding the elements that appear in the variants with the target feature, while $-e_p$ is penalizing the elements that appear in variants without the target feature. Using the 1.0 as threshold will take only the elements where it is always this case.

5. Evaluation setup

Fig. 7 presents the design of the evaluation of the proposed approach of this work. Each RQ has its own experiment protocol, i.e., input assets, approach for feature localization, and evaluation, which we explain in detail in the following subsections. Overall, RQ1 evaluates a dynamic approach combined with SBFL for locating features in single systems. The RQ2 focuses on FL for a family of systems. For answering RQ2, we refine dynamic analysis results with the commonality and differences of execution traces between features of variants to afterward apply the SBFL. In RQ3, we also focus on FL for a family of systems, but instead of applying dynamic approach combined with SBFL, we only applied a comparison-based static SBFL.

5.1. Case study

ArgoUML is a Java-based open-source tool for modeling software systems in the Unified Modeling Language (UML). The ArgoUML SPL FL benchmark (Martinez et al., 2018a) provides a feature location ground-truth for eight optional features within the ArgoUML source code. Six features are related to different UML diagrams and two features are cross-cutting features related to Logging support, and Cognitive support, which analyzes the diagrams and provides critics about how to improve them. The feature location ground-truth was created based on manual annotations in the source code using the original ArgoUML (Couto et al., 2011). From a total of 148 KLoC in ArgoUML, each feature contains from 1,579 to 16,319 LoC (Martinez et al., 2018a). The manual FL took around 4 months per feature and 0.7 months per feature-specific KLoC (Martinez et al., 2020). The duration in this case study, and the error-proneness of the process (Martinez et al., 2020), is also a motivation to investigate automatic FL.

Considering the created ArgoUML SPL and a set of variants ranging from one with all optional features selected (i.e., the Original ArgoUML) to one with all the optional features disabled, Couto et al. (2011) reported metrics regarding object-oriented complexity and SPL complexity. Regarding all the metrics reported, the number of packages can vary between 55 and 81, while the number of Java types can vary between 1,243 and 1,666. The scattering degree (i.e., the number of feature annotations) for the two cross-cutting features Logging and Cognitive support is 1,287 and 319 annotations, respectively, while for the other features can range between 64 and 167.

The benchmark is suitable for FL in families of systems, as it provides 15 predefined scenarios in that regard. The scenarios are useful to check how sensible FL techniques are with respect to the number of variants or the similarity among them. The Original ArgoUML scenario can be used for FL in single systems. There are 10 scenarios with different numbers of variants (2 to 10, 50 and 100). These 10 pre-defined scenarios were created by selecting randomly different possible configurations of the eight features, with the condition that all the features appear in the scenario. Then there are three scenarios that have special properties: The PairWise scenario contains nine variants with full pair-wise feature coverage, the Traditional scenario of 10 variants is one set of configurations that has been used in previous literature (e.g., Al-Msie'deen et al. (2013)), and the scenario All that contains all possible variants, namely 256 different product configurations.

The benchmark provides its own granularity level for the expected FL results. It is based on traces to Java types, methods, and what is referred to as Refinement of methods or types. For instance, a method might not completely correspond to a feature, but some statements within it might be related to this feature. Considering all features, there are traces related to 439 complete types, 44 complete methods, 388 type refinements, and 871 method refinements.

Taking the ArgoUML SPL FL benchmark, several FL techniques have been proposed to locate features[8] statically in sets of variants (Michelon et al., 2019; Müller and Eisenecker, 2019; Mortara et al., 2020; Cruz et al., 2019) as well as a hybrid (static and dynamic) FL approach (Michelon et al., 2021a). Our approach is novel compared to these previous works, especially with regard to RQ2. Finally, for RQ3, the objective is to check whether SBFL is suitable for static analysis as well.

Unfortunately, other benchmarks were not possible to use. The Eclipse FL Bench (Martinez et al., 2018b) is for locating features at the level of components (i.e., Eclipse plugins) and not for the source code within these components. Both the Linux

[8] https://variability-challenges.github.io/2018/ArgoUMLSPL

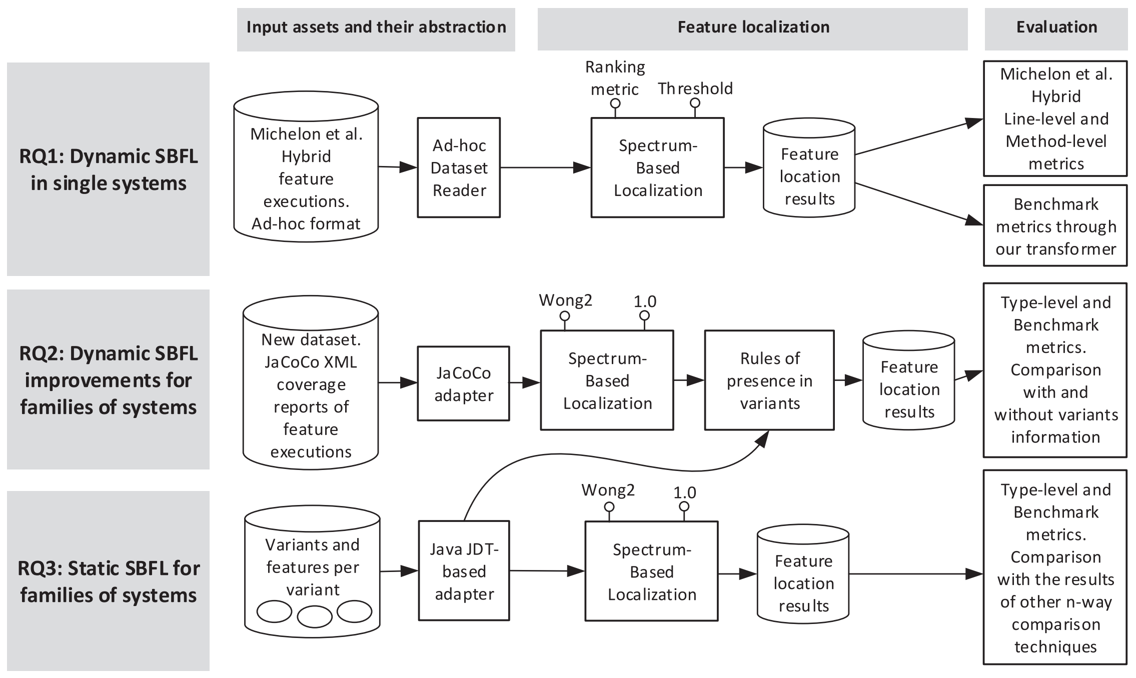

Input assets and their abstraction
Feature localization
Evaluation
RQ1: Dynamic SBFL in single systems
Michelon et al. Hybrid feature executions. Ad-hoc format
Ad-hoc Dataset Reader
Ranking metric
Threshold
Spectrum-Based Localization
Feature location results
Michelon et al. Hybrid Line-level and Method-level metrics
Benchmark metrics through our transformer
RQ2: Dynamic SBFL improvements for families of systems
New dataset. JaCoCo XML coverage reports of feature executions
JaCoCo adapter
Wong2
1.0
Spectrum-Based Localization
Rules of presence in variants
Feature location results
Type-level and Benchmark metrics. Comparison with and without variants information
RQ3: Static SBFL for families of systems
Variants and features per variant
Java JDT-based adapter
Wong2
1.0
Spectrum-Based Localization
Feature location results
Type-level and Benchmark metrics. Comparison with the results of other n-way comparison techniques

Fig. 7. Illustration of the approach to respond to the research questions.

kernel (Xing et al., 2013) and the Marlin (Krüger et al., 2019) code were programmed in C. Our implementation is developed for Java programs, thus, different adaptations and dependencies would be required in order to use other benchmarks.

5.1.1. Evaluation metrics

Similar to other FL techniques, we used the standard *precision*, *recall* and *f1* metrics to assess performance of the proposed approaches. To obtain these metrics, we need to extract the True Positives (TP), False Positives (FP) and False Negatives (FN). We apply such metrics at different granularity levels depending on the experiment. In global, they are line, method, type and benchmark traces level. A TP refers to a trace which has been appropriately located at its corresponding feature (hit). An FP refers to a trace which has been located in a feature, when it should have not (false alarm). An FN refers to a trace which has not been located in a feature when it should have (miss). The equations for the metrics are obtained as follows:

$$Precision = \frac{TP}{TP + FP} \tag{4}$$

$$Recall = \frac{TP}{TP + FN} \tag{5}$$

$$F1 = \frac{2 \times (precision \times recall)}{precision + recall} \tag{6}$$

For type and benchmark metrics, we used the ground-truth provided by the ArgoUML SPL FL benchmark. Type-level ground-truth can be obtained in a straightforward way from the benchmark ground-truth. This is because for the Type-level it is only needed to get the unique Types that appear for each feature, independently if the trace refers to the whole type, method, or refinement of type or method. However, to compute the metrics at the line-level (i.e., part of the evaluation of RQ1), we used a diff library for patches of code.[9] This library was used for performing the comparison operations between textual data. Concretely, for each type (i.e., Java class), the library compares (line by line) the text from the located lines of the feature, with the text of the available ground-truth. In this case, the ground-truth can be found in the scenarios available in the ArgoUML SPL FL benchmark, i.e., for each feature, there is a variant that contains only its source code plus the base source code. This textual comparison has been already used in previous work on feature localization (Michelon et al., 2021a, 2020, 2022). For the ground-truth at the method-level (also as part of RQ1), it was considered a method signature as part of a feature if a feature has at least one line included inside the method body.

6. Results and analysis

By adopting the evaluation setup presented in the previous section (Fig. 7), we conducted the study to evaluate the proposed approach. In the following, we present the experimental protocol and the results in order to answer each of the three RQs.

6.1. RQ1: Dynamic SBFL in single systems

6.1.1. Experimental protocol

Exercising the features. We reused execution traces from Michelon et al. Hybrid (Michelon et al., 2021a). These traces were made publicly available[10] in files with data about the lines of code that were executed per feature. Two scenarios were analyzed: (i) manual exercises and (ii) exercises from existing tests that are related to the features. Videos of the manual feature executions can be watched as part of the supplementary material of that work (Michelon et al., 2021a), and the test cases (1,198 in total) originate from Fischer et al. (2020). By reusing the feature execution files, we are able to compare the current results with the ones reported by Michelon et al. Hybrid (Michelon et al., 2021a) approach. Creating our own manual executions or using different test cases might introduce a significant bias. This is because of the nature of dynamic approaches, which results are highly sensitive to this input.

9 https://github.com/java-diff-utils/java-diff-utils

10 Dataset with feature execution traces: https://doi.org/10.5281/zenodo.5035177

Table 2
Characteristics of the feature exercises.

Feature	**Manual**			
	ELoC	FSLoC	FLoC (Martinez et al., 2018a)	EFLoC (Michelon et al., 2021a)
ActivityDiagram	20,185	1,139	2,282	29% (34%)
CollaborationDiagram	19,000	1,028	1,579	34% (37%)
DeploymentDiagram	19,980	1,590	3,147	41% (41%)
SequenceDiagram	18,537	1,562	5,379	30% (30%)
StateDiagram	19,534	1,288	3,917	36% (36%)
UsecaseDiagram	19,320	1,312	2,712	36% (36%)
Average	*19,426*	*1,320*	*2,169*	*34% (36%)*
Feature	**Tests**			
	ELoC	FSLoC	FLoC (Martinez et al., 2018a)	EFLoC (Michelon et al., 2021a)
ActivityDiagram	3,091	78	2,282	1% (2%)
CollaborationDiagram	3,095	42	1,579	2% (2%)
DeploymentDiagram	2,965	22	3,147	≈0% (0%)
SequenceDiagram	2,984	5	5,379	≈0% (0%)
StateDiagram	3,542	482	3,917	6% (6%)
UsecaseDiagram	3,061	64	2,712	1% (1%)
Logging	3,009	940	2,159	3% (8%)
Cognitive	9,119	6,071	16,319	14% (14%)
Average	*3,858*	*963*	*4,687*	*3% (4%)*

Table 2 presents a characterization of these feature exercises, where: ELoC (Exercise LoC) is the total number of unique LoC for the exercise of each feature. FSLoC (Fully-specific LoC) is the number of LoC from ELoC that are part of the feature and never appear in the exercise of other features. FLoC (Feature LoC) is the total number of LoC of the feature as per the benchmark data (Martinez et al., 2018a). Notice that each FSLoC is not necessarily a FLoC as it might be just a "coincidence" that, in the execution traces, no other feature executed that line. EFLoC (Exercise Feature LoC) is the percentage of the total number of LoC of the feature exercised from the execution traces reused from Michelon et al. Hybrid (Michelon et al., 2021a). EFLoC is the ratio of FLoC to the execution traces of the corresponding feature scenario, and also in parentheses, the ratio considering traces of all feature scenarios. By looking at EFLoC and FLoC metrics, it is possible to see how many lines of code of a feature were executed. This means that the results of the dynamic approach with SBFL using manual execution traces is favorable for higher recall in comparison to the test execution traces. However, the results are limited to high recall, because, on average, around 36% of the FLoC were executed for the manual executions and 4% for the tests. Therefore, improving the ratio of EFLoC and FLoC would improve the recall of the FL.

Feature localization. As mentioned before, two aspects are key for SBFL techniques: (i) the metric used to rank and (ii) the threshold to decide when a line of code is suspicious enough to be considered part of a feature. The threshold is needed as the benchmark does not consider probability as part of the metrics computation. In this work, we report the results using several combinations of ranking metrics and threshold values. Concretely, 33 ranking metrics[11] with 10 threshold values for each one (0.1, 0.2, …, 0.9, 1.0) were used. We refer in this paper to the results by using the name of the ranking metric followed by a threshold, e.g., Wong1 1.0, where Wong1 is a ranking metric and 1.0 is a threshold. The goal is not to overfit the technique to this dataset by using those 330 combinations, but to provide an overview of what can be the results of SBFL. We implemented the SBFL technique using the Stardust Java library.[12]

Comparison. In Michelon et al. Hybrid (Michelon et al., 2021a), metrics based on line- and method-level were used due to existing feature location techniques being limited to method-level or do not yield satisfactory results when applied to single systems. Similarly to the feature traces, we reused the code to calculate the line- and method-level metrics used in Michelon et al. Hybrid (Michelon et al., 2021a) to compare the two FL techniques in equal conditions. In addition, Michelon et al. Hybrid (Michelon et al., 2021a) computed the benchmark metrics (i.e., similar to statement-level). However, the execution traces granularity is at the line-level and then the results are also based on lines of code. Thus, it is not straightforward how to transform them to the convention established by the benchmark (Martinez et al., 2018a). As mentioned in Section 5.1, this convention mixes type-level, method-level, and refinements of types and methods (e.g., a "Refinement" tag in a method indicates that some lines in the method correspond to a feature but not the whole method).

We implemented our own transformer (i.e., from line of code to benchmark trace) with a simple implementation consisting of always adding a type-level localization when there is at least one line in the type. For instance, if we have one or several lines in a Java type with qualified name `package.myClass`, it will be transformed to a single benchmark trace to `package.myClass`. By using this naive approach, which does not consider methods or Refinement tags at type- and method-level, we assume a certain loss in recall. Other more sophisticated approaches have been explored, leading to worse results, and thus a better transformer for this specific benchmark will have to be part of further work. Nonetheless, the results are positive.

6.1.2. RQ1 – results

Comparison with the Hybrid approach. The static analysis implemented in the ECCO tool (Fischer et al., 2014) was used in Michelon et al. (2021a) for refining overlapping traces from the execution traces. In Michelon et al. (2021a), the FL is intended for re-engineering single systems into SPLs for creating variants, which should contain the base plus a set of feature-specific source code. Thus, line- and method-level metrics reported by Michelon et al. Hybrid (Michelon et al., 2021a) consider true positives, not only the feature-specific lines and methods but also the source code that is common for all features, i.e., the base of ArgoUML. For the SBL techniques, including the source code of the base in the precision and recall metrics is not potentially beneficial. This is because the standard ranking metrics are not designed

[11] The 33 ranking metrics (Naish et al., 2011) used in this study are: Ample, Anderberg, Arithmetic Mean, Cohen, Dice, Euclid, Fleiss, Geometric Mean, Goodman, Hamann, Hamming, HarmonicMean, Jaccard, Kulczynski1, Kulczynski2, M1, M2, Ochiai, Ochiai2, Overlap, Rogers Tanimoto, Rogot 1, Rogot 2, RussellRao, Scott, SimpleMatching, Sokal, SorensenDice, Tarantula, Wong1, Wong2, Wong3, and Zoltar.

[12] https://github.com/FaKeller/stardust commit 0071fe3 on Jan. 27th, 2016.

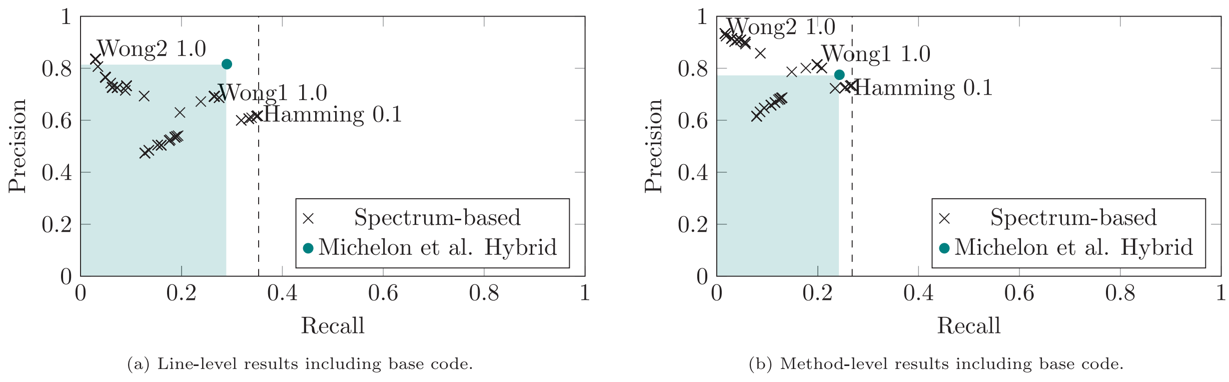


Fig. 8. Manual: Each point of "Spectrum-based" is a different ranking metric and threshold. Comparison with Michelon et al. Hybrid 2021 (Michelon et al., 2021a) using their precision and recall metrics that considered as true positives the retrieved source code from the feature-specific plus the base source code.

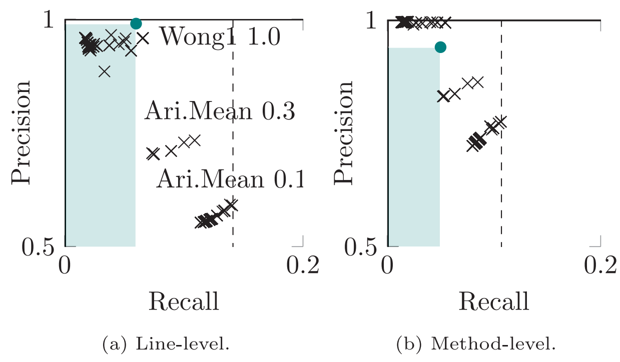


Fig. 9. Similar to Fig. 8 but using tests feature exercises.

for identifying the base plus the target feature, but only the target feature. However, the results are also competitive in this comparison.

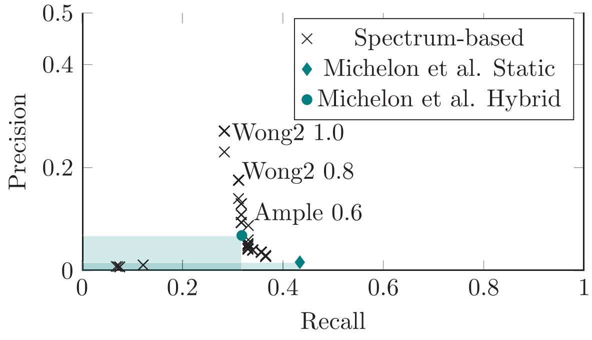
Figs. 8 and 9 present the results of the different ranking metrics for the manual and test execution traces using the line- and method-level metrics as defined in Michelon et al. (2021a). The vertical dashed lines show the boundaries regarding recall. Even if we include all the source code executed through the input execution traces, these execution traces did not include all the feature source code and base code. For the sake of illustration, we selected and included specific ranking labels in Figs. 8 and 9 to show examples of those with high precision, recall, and balance between them. The labels correspond to the closest points to the label.

In Fig. 8, the average of the six diagram features from manual execution from Michelon et al. Hybrid (Michelon et al., 2021a) approach mostly dominates at line-level (the dominated area has a light background). However, ranking metrics in the extremes of precision (e.g., Wong2 1.0) and recall (e.g., Hamming 0.1) are also part of the Pareto front (i.e., non-dominated solutions). At the method-level, SBFL techniques get closer to Michelon et al. (2021a) even if they were not specifically designed to locate the base source code. Fig. 9 shows the results for the tests execution traces for the eight features. At line-level, the results by Michelon et al. Hybrid (Michelon et al., 2021a) dominates in terms of precision, but several SBFL techniques are close to it, such as Wong1 1.0, which also has a higher recall. The highest precision for SBFL is obtained with Wong3 1.0. At the method-level, we have solutions that slightly outperform Michelon et al. Hybrid (Michelon et al., 2021a) both in precision and recall, such as Wong1 1.0, Ochiai 0.1 to 0.3, or Tarantula 0.1 to 0.5.

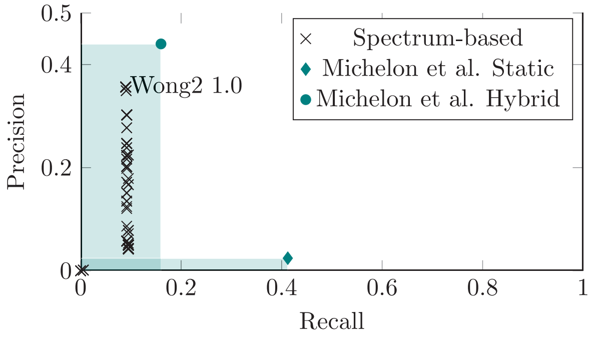
In summary, on the one hand, it can be appreciated that SBFL techniques outperform the approach from Michelon et al. Hybrid (Michelon et al., 2021a) when considering the method-level metrics. On the other hand, when considering line-level metrics, the approach from Michelon et al. Hybrid (Michelon et al., 2021a) shows a slightly higher precision, although the difference is not high. The approach from Michelon et al. Hybrid (Michelon et al., 2021a) retrieves higher precision at the line-level because it considers also the lines of the base source code as true positives, which our technique does not. At the method-level, the base source code does not make such a difference because lines belonging to the base in a method body are not considered. A method signature is part of a feature when at least one line is within the method body. Thus, fewer false positives are retrieved by our SBFL technique, which gets closer to the approach from Michelon et al. Hybrid (Michelon et al., 2021a). Some results of our approach show higher recall when compared to the approach from Michelon et al. Hybrid (Michelon et al., 2021a). Specifically, the results present higher precision scores when applying SBFL with the Wong2 metric, whereas the ArithmeticMean or Hamming metrics provide higher recall scores.

Spectrum-based localization results. Fig. 10 shows the results with our own transformer to benchmark convention for the manual and test exercises. For comparison, we include the results reported by Michelon et al. Hybrid (Michelon et al., 2021a) using the benchmark convention. In addition, we include Michelon et al. Static (Michelon et al., 2019) results, which is a static approach reasoning on variants overlap. The latter is successful in scenarios with an increasing number of variants and it also considers feature interactions. From this work, we include only the results in the benchmark scenario with only one variant and using the average of the considered features.

In Fig. 10(a), the results with the highest precision are obtained through different ranking metrics (Wong2 0.9 to 1.0, Ochiai 0.8 to 1.0, Ochiai2 0.7 to 1.0, Tarantula 0.9 to 1.0, Hamming 0.9 to 1.0, Euclid 1.0, ArithmeticMean 0.9 to 1.0, HarmonicMean 0.9 to 1.0, Anderberg 0.4 to 1.0, among others) and with the same values as with the manual exercises. Those are also the ones that obtained the highest F1, i.e., HarmonicMean of the precision and recall. For recall, not surprisingly, the highest values are obtained with low thresholds with different techniques (e.g., Wong2 0.1, Hamming 0.1, ArithmeticMean 0.1 to 0.4, among others). We can observe how Michelon et al. Static (Michelon et al., 2019) is in the Pareto front because of its recall value. Michelon et al. Hybrid (Michelon et al., 2021a) got lower values in both precision and recall compared to SBFL ranking metrics and thresholds (Ample 0.5 to 0.6). Fig. 10(b) shows the results for the testing exercises. In this case, Michelon et al. Hybrid (Michelon et al., 2021a) dominates the SBFL technique in both dimensions. The ones closer to Michelon et al. Hybrid (Michelon et al., 2021a) are Wong2, Tarantula, Hamming, ArithmeticMean 0.9 to 1.0, Anderberg 0.4 to 1.0, Ochiai 0.8 to 1.0, and Ochiai2 0.7 to 1.0.



(a) Manual: Benchmark results for the six features.



(b) Tests: Benchmark results for the eight features.

Fig. 10. Feature exercises: using manual (six diagram features) and tests (all eight features) traces.

As a summary, for the manual exercises, techniques based on our approach provide higher precision than previously proposed by Michelon et al. Static (Michelon et al., 2019) and Michelon et al. Hybrid (Michelon et al., 2021a) approaches. Specifically, the Wong2 0.9 to 1.0 show the highest precision, while being competitive in the recall too. However, the approach from Michelon et al. Static (Michelon et al., 2019) shows an overall higher recall, but precision was lower. The reason why the approach from Michelon et al. Static (Michelon et al., 2019) results in fewer false negatives is because it considers all the source code and not only the source code from execution traces as our approach does. In our case, the execution traces do not have good coverage of the source code.

On the other hand, when using the tests, the approach from Michelon et al. Hybrid (Michelon et al., 2021a) showed both higher precision and recall than ours. The reason is again because of considering as true positives also the lines belonging to the base source code, besides the features source code. The approach from Michelon et al. Static (Michelon et al., 2019) also outperforms our approach in terms of recall, although not in precision.

Regarding the performance, we computed the average of 30 runs using a laptop model Latitude 5480, Intel(R) Core(TM) i5-7300U processor (2.60 GHz), running the Windows 10 operating system. The Dataset Reader takes around 18 s as it needs to parse all the execution traces' files. For the creation of the spectrum, the computation of the ranks using the ranking metric for the six features, under the defined threshold 0.1 for all the runs in the performed experiment, took less than a second or around a second. Thus, scalability regarding performance does not seem to be a problem for SBFL techniques. The runtime performance of the static approach used in Michelon et al. Hybrid (Michelon et al., 2021a), after having the execution traces, took on average ≈19 s per feature. We cannot compare the runtime of our approach with Michelon et al. Hybrid (Michelon et al., 2021a) because the laptop model is different, but both performed in a reasonable time. As a drawback, we should remember the time needed to exercise the features, a required step in any dynamic FL technique, which can add several minutes for preparation and obtaining the execution traces.

6.2. RQ2: Improving dynamic SBFL for families of systems

6.2.1. Experimental protocol

Exercising the features. We created a new JaCoCo-based dataset to provide a more standard format for replication and to ease building on top of this work. Table 3 shows the number of executed lines per feature. For the manual exercises, we replicated the same procedure. As it was not exactly the same manual interaction, we ended up with around 4,000 more lines of code per feature compared with Table 2. Overall, for the manual exercises, the number of covered lines are quite similar for each feature. However, for the tests, the feature `Cognitive` seems to be more largely exercised. For the tests, we created JUnit test suites according to the mapping of features and tests from Fischer et al. (2020). Table 3 shows also the number of tests per feature. We can observe that `Cognitive` consists of 251 tests which is a significantly larger number than the available tests for other features.

Table 3
Characteristics of the feature exercises for the JaCoCo dataset.

Feature	ELoC		Number of Tests
	Manual	Tests	
ActivityDiagram	23,693	5,560	14
CollaborationDiagram	23,177	4,575	3
DeploymentDiagram	23,636	3,909	1
SequenceDiagram	22,119	4,350	7
StateDiagram	24,481	5,333	13
UsecaseDiagram	23,219	4,841	20
Cognitive	–	22,407	251
Average	*23,387*	*7,282*	*44*

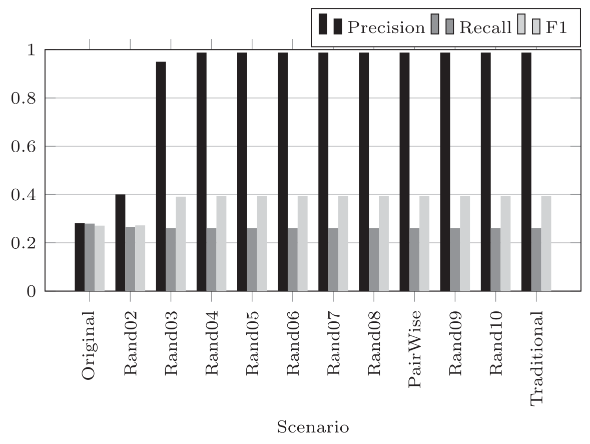
We consider that showing the results for all the ranking metrics and threshold combinations is no longer needed. We decided to leverage the findings in RQ1 to select one of the combinations with high precision (e.g., Wong2 0.9 to 1.0, Ochiai2 0.7 to 1.0, or Tarantula 0.9 to 1.0) so we will show the results for Wong2 1.0. If our hypothesis is correct, the method described in Section 4.3 to refine SBFL results, will allow increasing precision. This way, selecting one that we know that will have relatively high precision, will be the worst case to show increase in precision.

For the experimentation of RQ2 we use 12 scenarios of the benchmark, and for both RQ2 and RQ3, the performance metrics are obtained with a MacBook Pro 2020, 2 GHz Quad-Core Intel Core i5, 16 GB 3733 MHz LPDDR4X.

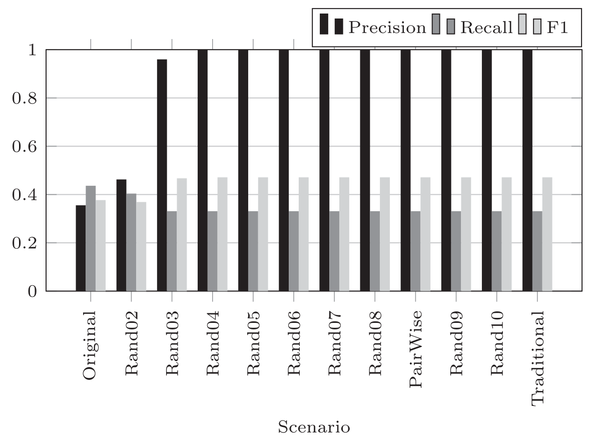
6.2.2. RQ2 – results

Figs. 11 and 12 present the results for the manual and test execution traces, respectively. They show precision, recall, and F1 for both the benchmark metrics and type-level metrics. In this RQ2, we assume that variants exist, but we included the Original scenario (single system) to show the initial results of the SBFL technique in single systems with the new JaCoCo execution traces dataset.

In the mentioned figures, we can observe how for the Random02 scenario, our approach already increases precision without losing recall. The Random02 scenario contains two variants where one is the Original and the other one is the Original without one feature. Then we observe a significant increase of

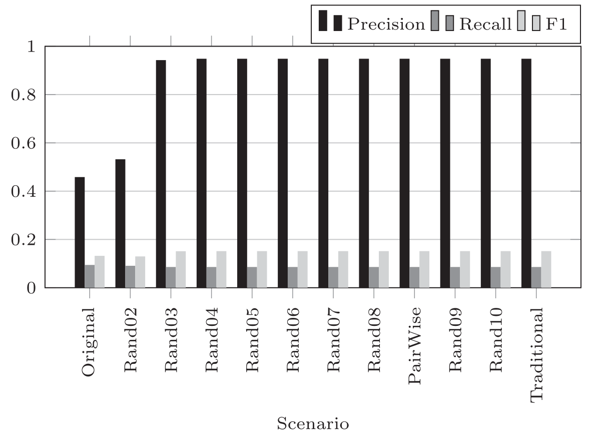


(a) Benchmark metrics.

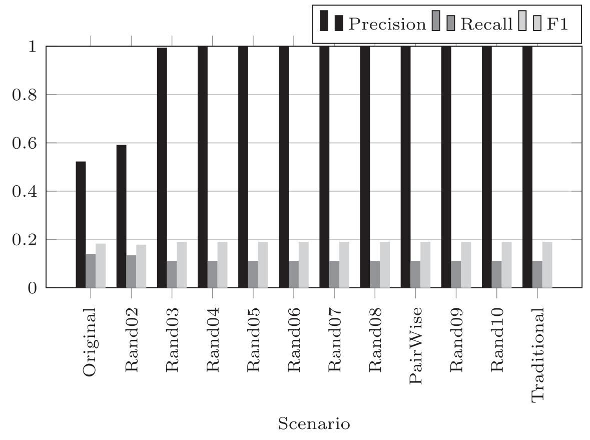


(b) Type-level metrics.

Fig. 11. Results of the dynamic SBFL with manual traces.



(a) Benchmark metrics.



(b) Type-level metrics.

Fig. 12. Results of the dynamic SBFL with test traces.

precision with three variants (Random03). Then, a smaller increase with four variants, and finally it gets constant independently of the scenario. These results confirm that the proposed approach for enhancing dynamic SBFL results for families of systems is sound. Thus, the proposed approach should be used when variants are available.

Figs. 13(a) and 13(b) show the time measurements. We separated the time from the dynamic SBFL technique (few seconds for all scenarios) and the relatively time-consuming rules. No optimization was performed as in our context, where FL is normally done once, the time performance is not highly relevant.

6.3. RQ3: Static SBFL for families of systems

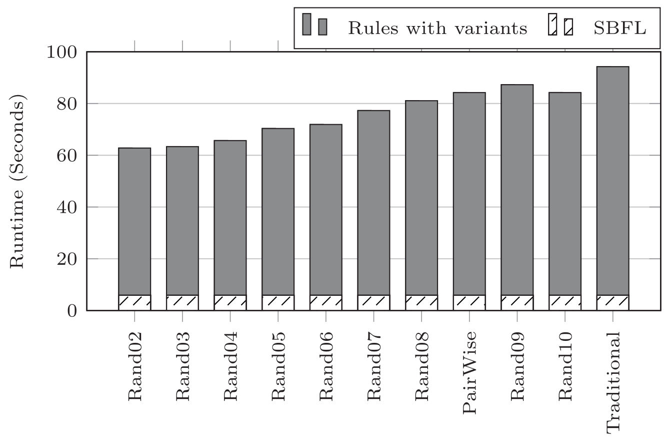
6.3.1. Experimental protocol

We use Wong2 1.0, as explained in Section 4.4, and 11 scenarios from the benchmark. For the list of features, we automatically add interaction features for each pair of features present in each scenario. This expansion of the feature list considering feature interactions is relevant in the SFS step (see Section 3.3) for assigning the identified blocks not only to features, but also to feature interactions. With this method, available in BUT4Reuse, the pairwise feature interactions that are considered are determined by the scenario. Thus, feature interactions that are present in at least one variant of a respective scenario are added, and interactions that are not present in the variants of the respective scenario are not added to the feature list. The intuition is that, for instance, if features A and B are never present in any variant of the scenario, it is not worthy to try to locate the implementation elements exclusively related to the interaction of A and B, as they will not be present in any of the variants.

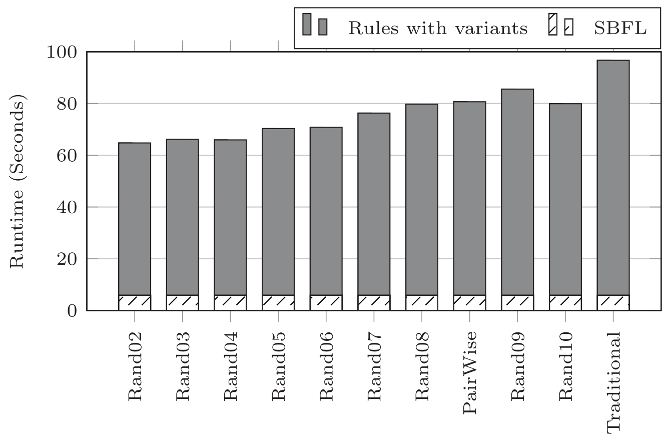
6.3.2. RQ3 – results

Fig. 14 shows the results for the benchmark metrics while using the static SBFL approach presented in Section 4.4. We can observe that the results of our static SBFL technique are the same as FCA+SFS and IE+SFS for all the scenarios. Only in four scenarios we found variations of very small order that we did not find a clear reason for. Anyhow, the four variations were positive for the static SBFL approach.

As presented in Section 4.1, the n-way matching used in our experiments is id-based and signature based. Identical n-way matching strategies were used for our SBFL technique, FCA+SFS and IE+SFS. Certainly, more sophisticated n-way matching strategies have been proposed in the literature, such as multi-dimensional search trees (Schultheiß et al., 2021) or algorithms that incrementally process variants' subsets (Rubin and Chechik, 2013a). Despite that investigating the effect of different n-way matching strategies is an interesting direction, it was out of the scope of this work. Further, these advanced approaches cannot show all their benefits with ArgoUML SPL FL benchmark, as variants of each scenario did not evolve with refactoring or other type of change.

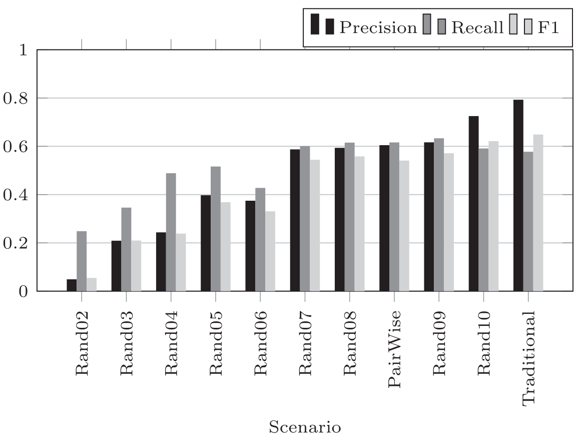


(a) Manual traces.

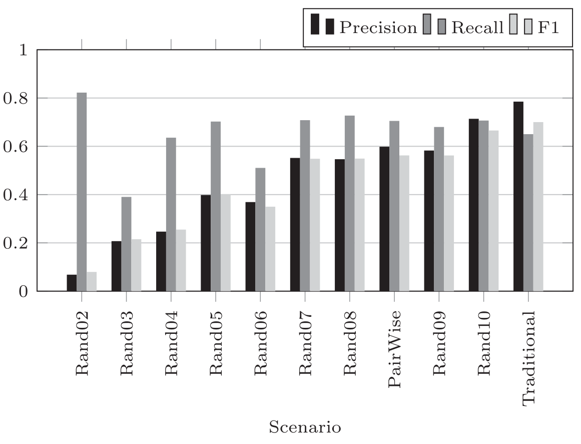


(b) Test traces.

Fig. 13. Measured average execution time of 10 runs for the dynamic SBFL.

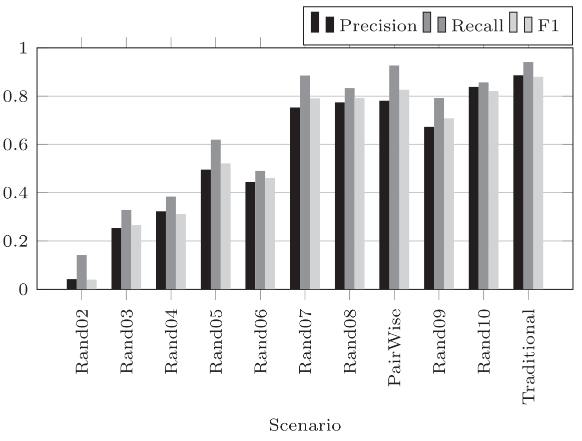


(a) Benchmark metrics.

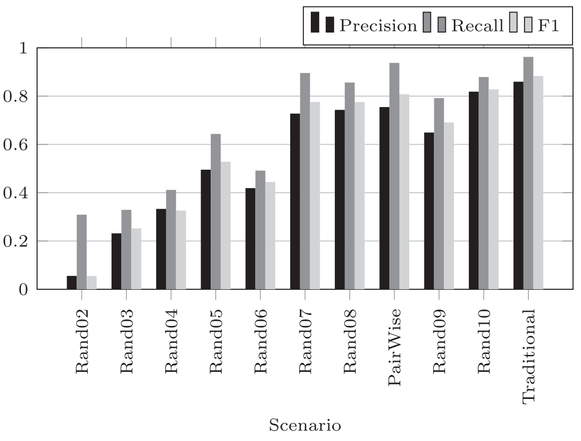


(b) Type-level metrics.

Fig. 14. Results of the static SBFL which are the same as FCA+SFS and IE+SFS.



(a) Benchmark metrics.



(b) Type-level metrics.

Fig. 15. ECCO results using its comparison-based feature localization technique (Michelon et al., 2019).

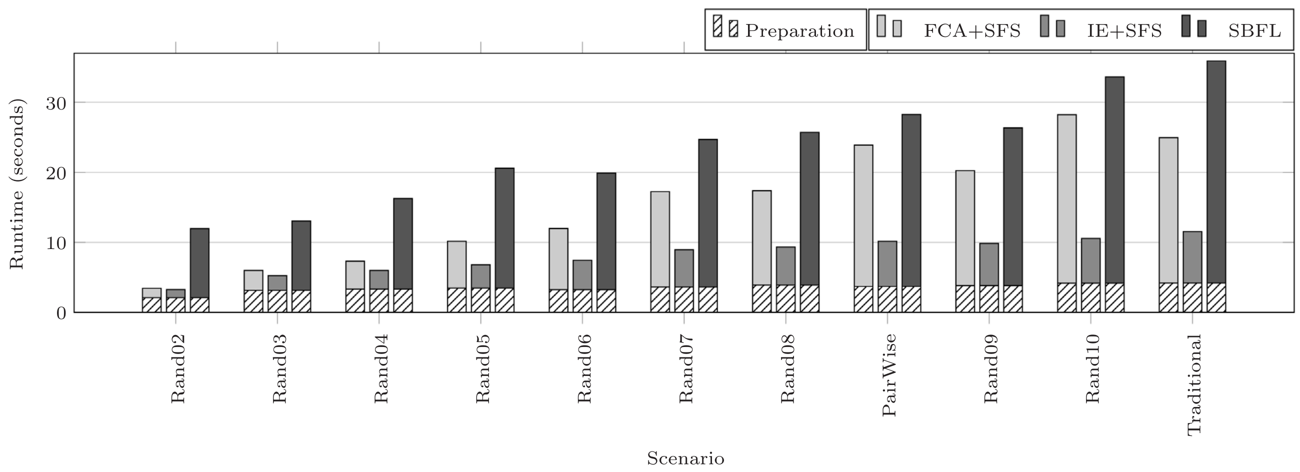


Fig. 16. Measured average execution time of 10 runs for FCA+SFS, IE+SFS, and SBFL. It includes the preparation and execution time for each technique.

Fig. 15 shows the results of an advanced comparison-based approach proposed for the benchmark (Michelon et al., 2019) based on the ECCO tool. We can observe that the results shown in Fig. 14 are relatively similar to the ECCO approach until the scenario Random06. Fig. 14(b) shows the results at Java types granularity. In this graph, we can observe higher recall until Random06, meaning that the involved Java files for the features are better retrieved. However, after Random06, ECCO clearly outperforms our approach, being also a static one. The reason can be related to the finer granularity that ECCO does of the AST nodes, being able to calculate similarity at statement level, or the way it incrementally compares the variants.

Fig. 16 shows the time measurements for the static SBFL approach as well as for FCA+SFS and IE+SFS. We separate each one in Preparation time and FL time. Preparation in BUT4Reuse consists in adapting the variants, i.e., parsing the source code and creating the JDT Elements abstraction. This is common to all FL techniques. Using the concurrent adaptation option of BUT4Reuse we noticed better times for this preparation phase. The adaptation in parallel of the variants makes that the time for preparation is relatively similar for all the scenarios, independently of the number of variants. We can observe that IE+SFS outperforms the time performance of both FCA+SFS and our static SBFL. We implemented some optimizations, such as generating the spectra only once for locating all the different features. However, there is potential for further optimizations and other SBL libraries can be tested to replace the one we use now. Contrary to both our static SBFL technique and FCA+SFS, IE+SFS does not use any external library as it is implemented directly in BUT4Reuse. Thus, this can also explain its runtime measures. An analysis of the order of complexity of the different algorithms is out of the scope of this paper. Our objective was to empirically show results in terms of accuracy and time performance.

Based on the results of this section, we can see that SBFL can be used not only for dynamic FL, but also for static FL without detriment to accuracy compared to state-of-the-art approaches. Our technique takes longer time, but it is still feasible for practical use. Nonetheless, we recall that our main goal in RQ3 was to evaluate the flexibility of SBFL techniques.

7. Threats to validity

An *external validity* threat in our evaluation refers to the generalizability of the results. We only used one case study to evaluate our approach, the ArgoUML SPL FL benchmark (Martinez et al., 2018a). Other case studies exist, such as the Linux kernel (Xing et al., 2013) and Marlin (Krüger et al., 2019). Both of them are programmed in C. As we mentioned in Section 5.1, our implementation has dependencies with different tools for Java (e.g., JaCoCo), therefore, not being possible to use with these case studies. However, our case study is widely used by the community (Martinez et al., 2017). Moreover, as mentioned before, the tool contains a total of eight optional features, and diverse scenarios. The scenarios provide certain level of diversity in the set of variants. Further, the scenarios can be considered different case studies, as we evaluated each of them separately. Each feature contains from 1,579 to 16,319 lines of code (LoC). The total number of LoC of the variants range between 110 and 148 KLoC (Martinez et al., 2018a). Thus, the complexity of the case study was high compared to other illustrative systems that could have been considered (e.g., the mentioned Draw Product Line or Sudoku and Notepad (Michelon et al., 2021a)). Adding any of those illustrative systems will not have added value to our contribution as a manual FL is straightforward in those cases. The availability of benchmarks is a recognized concern in the SPL field (Strüber et al., 2019) and very recent efforts are trying to automatize their creation (Schultheiß et al., 2022). In our case, the system also requires having a graphical user interface to exercise the features, or feature-based tests, which presents also challenging requirements. While we consider that our techniques are sound, the threat to generalization will need to be considered in future works.

An *internal validity* threat refers to source code of our experimentation pipeline that might have implementation issues. We mitigated this by being three researchers the ones inspecting the source code and comparing our results to allow future reproducibility. Another internal threat to validity is related to the matching algorithm we used to compute the metrics at the line-level. To mitigate this, we used a diff library, which has been already used in previous works on feature location (Michelon et al., 2021a, 2020, 2022), and its implementation is available.

8. Conclusions

Feature location is a relevant activity for the detection phase in re-engineering variants into SPLs. We explored the use of SBFL in the context of families of systems. Our results obtained in several scenarios of an established feature localization benchmark show that SBFL in families of systems increases precision even with only a few variants. Thus, the presence of variants can make the dynamic SBFL results more precise. Regarding some settings of SBFL, using high thresholds in our study favors precision with the metrics Wong2, Ochiai2, or Tarantula. Conversely, using most of the SBFL metrics with low thresholds favors recall. We also confirmed that SBFL can be used in a static way to compare variants providing equal, or at least competitive, results as those in the state of the art. We expect that our novel approaches, and the obtained empirical results, could position SBFL as an alternative to be considered in FL for SPL adoption tasks.

As further work, the proposed static and dynamic SBFL techniques could be combined with text-based techniques. Other case studies will also be desirable for the generalization of the findings.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data is available and shared with a link in the paper as footnotes.

Acknowledgments

This research was funded by the LIT Secure and Correct Systems Lab, Spain; the Austrian Science Fund (FWF), grant no. P31989; Pro2Future, a COMET K1-Centre of the Austrian Research Promotion Agency (FFG), grant no. 854184. Aitor Arrieta is part of the Software and Systems Engineering research group of Mondragon Unibertsitatea (IT1519-22), supported by the Department of Education, Universities and Research of the Basque Country, Spain.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. In: TAICPART-MUTATION '07, IEEE, USA, pp. 89–98. http://dx.doi.org/10.1109/TAIC.PART.2007.13.

Al-Msie'deen, R., Seriai, A., Huchard, M., Urtado, C., Vauttier, S., Salman, H.E., 2013. Feature location in a collection of software product variants using formal concept analysis. In: Favaro, J.M., Morisio, M. (Eds.), Safe and Secure Software Reuse - 13th International Conference on Software Reuse. ICSR '13, Pisa, Italy, June 18-20. Proceedings, In: Lecture Notes in Computer Science, vol.7925, Springer, pp. 302–307. http://dx.doi.org/10.1007/978-3-642-38977-1_22.

Apel, S., Batory, D.S., Kästner, C., Saake, G., 2013. Feature-Oriented Software Product Lines - Concepts and Implementation. Springer, New York, USA.

Apel, S., Kastner, C., Lengauer, C., 2009. FEATUREHOUSE: Language-independent, automated software composition. In: 31st International Conference on Software Engineering. ICSE '09, pp. 221–231. http://dx.doi.org/10.1109/ICSE.2009.5070523.

Asadi, F., Penta, M.D., Antoniol, G., Guéhéneuc, Y.-G., 2010. A heuristic-based approach to identify concepts in execution traces. In: 14th European Conference on Software Maintenance and Reengineering. IEEE, pp. 1–10. http://dx.doi.org/10.1109/csmr.2010.17.

Assunção, W.K.G., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A., 2017. Reengineering legacy applications into software product lines: a systematic mapping. Empir. Softw. Eng. 22 (6), 2972–3016. http://dx.doi.org/10.1007/s10664-017-9499-z.

Bittner, P.M., Schultheiß, A., Thüm, T., Kehrer, T., Young, J.M., Linsbauer, L., 2021. Feature trace recording. In: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE '21:, ACM, pp. 1007–1020. http://dx.doi.org/10.18420/se2022-ws-002.

Castro, B., Perez, A., Abreu, R., 2019. Pangolin: an SFL-based toolset for feature localization. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE '19, IEEE, New York, USA, pp. 1130–1133.

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., Koschke, R., 2009. A systematic survey of program comprehension through dynamic analysis. IEEE Trans. Softw. Eng. 35 (5), 684–702. http://dx.doi.org/10.1109/TSE.2009.28.

Couto, M.V., Valente, M.T., Figueiredo, E., 2011. Extracting software product lines: A case study using conditional compilation. In: 15th European Conference on Software Maintenance and Reengineering. CSMR '11, 1-4 March 2011, Oldenburg, Germany, IEEE, New York, USA, pp. 191–200. http://dx.doi.org/10.1109/CSMR.2011.25.

Cruz, D., Figueiredo, E., Martinez, J., 2019. A literature review and comparison of three feature location techniques using argouml-SPL. In: 13th International Workshop on Variability Modelling of Software-Intensive Systems. VAMOS '19, Leuven, Belgium, February 06-08, 2019, ACM, New York, USA, pp. 16:1–16:10. http://dx.doi.org/10.1145/3302333.3302343.

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2013. Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process. 25 (1), 53–95. http://dx.doi.org/10.1002/smr.567.

Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V., 2008. Do crosscutting concerns cause defects? IEEE Trans. Software Eng. 34 (4), 497–515. http://dx.doi.org/10.1109/TSE.2008.36.

Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. IEEE Trans. Softw. Eng. 29 (3), 210–224. http://dx.doi.org/10.1109/TSE.2003.1183929.

Falleri, J.-R., 2009. Automatic Refactoring and Alignment of Class Models (Contributions à l'IDM : reconstruction et alignement de modèles de classes),Vol. 1 (Ph.D. thesis). Université de Lille, p. 217, http://www.theses.fr/2009MON20103 https://github.com/jrfaller/galatea.

Fischer, S., Linsbauer, L., Lopez-Herrejon, R.E., Egyed, A., 2014. Enhancing clone-and-own with systematic reuse for developing software variants. In: 30th IEEE International Conference on Software Maintenance and Evolution. ICSME '14, Victoria, BC, Canada, September 29 - October 3, 2014, IEEE, New York, USA, pp. 391–400. http://dx.doi.org/10.1109/ICSME.2014.61.

Fischer, S., Michelon, G.K., Ramler, R., Linsbauer, L., Egyed, A., 2020. Automated test reuse for highly configurable software. Empir. Softw. Eng. 25 (6), 5295–5332. http://dx.doi.org/10.1007/s10664-020-09884-x.

Ganter, B., Wille, R., 1999. Formal Concept Analysis - Mathematical Foundations. Springer.

Ji, W., Berger, T., Antkiewicz, M., Czarnecki, K., 2015. Maintaining feature traceability with embedded annotations. In: Schmidt, D.C. (Ed.), 19th International Conference on Software Product Line. SPLC '15, Nashville, TN, USA, July 20-24, 2015, ACM, pp. 61–70. http://dx.doi.org/10.1145/2791060.2791107.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: 24th International Conference on Software Engineering. ICSE '02, Association for Computing Machinery, New York, NY, USA, pp. 467–477. http://dx.doi.org/10.1145/581339.581397.

Koschke, R., Quante, J., 2005. On dynamic feature location. In: 20th IEEE/ACM International Conference on Automated Software Engineering. ASE '05, Association for Computing Machinery, New York, NY, USA, pp. 86–95. http://dx.doi.org/10.1145/1101908.1101923.

Krueger, C.W., 2001. Easing the transition to software mass customization. In: Software Product-Family Engineering, 4th Int. Workshop. PFE '01, Bilbao, Spain, October 3-5, 2001, Revised Papers, In: Lecture Notes in Computer Science, vol.2290, Springer, New York, USA, pp. 282–293. http://dx.doi.org/10.1007/3-540-47833-7_25.

Krüger, J., Mukelabai, M., Gu, W., Shen, H., Hebig, R., Berger, T., 2019. Where is my feature and what is it about? a case study on recovering feature facets. J. Syst. Softw. 152, 239–253. http://dx.doi.org/10.1016/j.jss.2019.01.057.

Mahmoud, A., Bradshaw, G., 2015. Estimating semantic relatedness in source code. ACM Trans. Softw. Eng. Methodol. 25 (1), http://dx.doi.org/10.1145/2824251.

Malburg, J., Finder, A., Fey, G., 2014. A simulation-based approach for automated feature localization. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 33 (12), 1886–1899. http://dx.doi.org/10.1109/TCAD.2014.2360462.

Martinez, J., Assunção, W.K.G., Ziadi, T., 2017. ESPLA: a catalog of extractive SPL adoption case studies. In: 21st International Systems and Software Product Line Conference. SPLC '17, Volume B, Sevilla, Spain, September 25-29, 2017, ACM, pp. 38–41. http://dx.doi.org/10.1145/3109729.3109748.

Martinez, J., Ordoñez, N., Tërnava, X., Ziadi, T., Aponte, J., Figueiredo, E., Valente, M.T., 2018a. Feature location benchmark with argouml SPL. In: 22nd International Systems and Software Product Line Conference - Volume 1. SPLC '18, ACM, New York, USA, pp. 257–263. http://dx.doi.org/10.1145/3233027.3236402.

Martinez, J., Wolfart, D., Assunção, W.K.G., Figueiredo, E., 2020. Insights on software product line extraction processes: ArgoUML to argoUML-SPL revisited. In: 24th ACM International Systems and Software Product Line Conference. SPLC '20, Montreal, Quebec, Canada, October 19-23, 2020, Volume A, ACM, New York, USA, pp. 6:1–6:6. http://dx.doi.org/10.1145/3382025.3414971.

Martinez, J., Ziadi, T., Bissyandé, T.F., Klein, J., Traon, Y.L., 2015. Bottom-up adoption of software product lines: a generic and extensible approach. In: Schmidt, D.C. (Ed.), 19th International Conference on Software Product Line. SPLC '15, Nashville, TN, USA, July 20-24, 2015, ACM, pp. 101–110. http://dx.doi.org/10.1145/2791060.2791086.

Martinez, J., Ziadi, T., Papadakis, M., Bissyandé, T.F., Klein, J., Traon, Y.L., 2018b. Feature location benchmark for extractive software product line adoption research using realistic and synthetic eclipse variants. Inf. Softw. Technol. 104, 46–59. http://dx.doi.org/10.1016/j.infsof.2018.07.005.

Meixner, K., Rabiser, R., Biffl, S., 2020. Feature identification for engineering model variants in cyber-physical production systems engineering. In: 14th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS '20, Magdeburg Germany, February 5-7, 2020, ACM, pp. 18:1–18:5. http://dx.doi.org/10.1145/3377024.3377043.

Michelon, G.K., Linsbauer, L., Assunção, W.K.G., Egyed, A., 2019. Comparison-based feature location in argouml variants. In: 23rd International Systems and Software Product Line Conference. SPLC '19, Volume a, Paris, France, September 9-13, 2019, ACM, New York, USA, pp. 17:1–17:5. http://dx.doi.org/10.1145/3336294.3342360.

Michelon, G.K., Linsbauer, L., Assunção, W.K., Fischer, S., Egyed, A., 2021a. 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, Association for Computing Machinery, New York, NY, USA, pp. 1–9. http://dx.doi.org/10.1145/3442391.3442403.

Michelon, G.K., Obermann, D., Assunção, W.K.G., Linsbauer, L., Grünbacher, P., Fischer, S., Lopez-Herrejon, R.E., Egyed, A., 2022. Evolving software system families in space and time with feature revisions. Empir. Softw. Eng. 27 (5), 112. http://dx.doi.org/10.1007/s10664-021-10108-z.

Michelon, G.K., Obermann, D., Linsbauer, L., Assunção, W.K.G., Grünbacher, P., Egyed, A., 2020. Locating feature revisions in software systems evolving in space and time. In: 24th ACM International Systems and Software Product Line Conference. SPLC '20, Montreal, Quebec, Canada, October 19-23, 2020, Volume A, ACM, pp. 14:1–14:11. http://dx.doi.org/10.1145/3382025.3414954.

Michelon, G.K., Sotto-Mayor, B., Martinez, J., Arrieta, A., Abreu, R., Assunção, W.K.G., 2021b. Spectrum-based feature localization: a case study using argouml. In: 25th ACM International Systems and Software Product Line Conference. SPLC '21, Leicester, United Kingdom, September 6-11, 2021, Volume A, ACM, pp. 126–130. http://dx.doi.org/10.1145/3461001.3473065.

Mortara, J., Tërnava, X., Collet, P., 2020. Mapping features to automatically identified object-oriented variability implementations: the case of argouml-SPL. In: 14th International Working Conference on Variability Modelling of Software-Intensive Systems. VaMoS '20, Magdeburg Germany, February 5-7, 2020, ACM, New York, USA, pp. 20:1–20:9. http://dx.doi.org/10.1145/3377024.3377037.

Müller, R., Eisenecker, U.W., 2019. A graph-based feature location approach using set theory. In: 23rd International Systems and Software Product Line Conference. SPLC '19, Volume a, Paris, France, September 9-13, 2019, ACM, New York, USA, pp. 16:1–16:5. http://dx.doi.org/10.1007/978-3-642-20844-7_21.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32. http://dx.doi.org/10.1145/2000791.2000795.

Perez, A., Abreu, R., 2014. A diagnosis-based approach to software comprehension. In: 22nd International Conference on Program Comprehension. ACM, New York, USA, pp. 37–47. http://dx.doi.org/10.1145/2597008.2597151.

Perez, A., Abreu, R., 2016. Framing program comprehension as fault localization. J. Softw. Evol. Process 28 (10), 840–862. http://dx.doi.org/10.1002/smr.1799.

Pleumann, J., Yadan, O., Wetterberg, E., 2010. Antenna. https://antenna.sourceforge.net.

Razzaq, A., Le Gear, A., Exton, C., Buckley, J., 2020. An empirical assessment of baseline feature location techniques. Empir. Softw. Eng. 25 (1), 266–321. http://dx.doi.org/10.1007/s10664-019-09734-5.

Razzaq, A., Wasala, A., Exton, C., Buckley, J., 2018. The state of empirical evaluation in static feature location. ACM Trans. Softw. Eng. Methodol. 28 (1), http://dx.doi.org/10.1145/3280988.

Revelle, M., Dit, B., Poshyvanyk, D., 2010. Using data fusion and web mining to support feature location in software. In: 18th IEEE International Conference on Program Comprehension. ICPC '10, Braga, Minho, Portugal, June 30-July 2, 2010, IEEE Computer Society, pp. 14–23. http://dx.doi.org/10.1109/ICPC.2010.10.

Robillard, M.P., 2008. Topology analysis of software dependencies. ACM Trans. Softw. Eng. Methodol. 17 (4), http://dx.doi.org/10.1145/13487689.13487691.

Robillard, M.P., Murphy, G.C., 2002. Concern graphs: Finding and describing concerns using structural program dependencies. In: 24th International Conference on Software Engineering. ICSE '02, ACM, New York, NY, USA, pp. 406–416. http://dx.doi.org/10.1145/581339.581390.

Rohatgi, A., Hamou-Lhadj, A., Rilling, J., 2008. An approach for mapping features to code based on static and dynamic analysis. In: 16th IEEE International Conference on Program Comprehension. ICPC '08, IEEE, pp. 236–241. http://dx.doi.org/10.1109/ICPC.2008.35.

Rubin, J., Chechik, M., 2012. Locating distinguishing features using diff sets. In: IEEE/ACM International Conference on Automated Software Engineering. ASE'12, Essen, Germany, September 3-7, 2012, pp. 242–245. http://dx.doi.org/10.1145/2351676.2351712.

Rubin, J., Chechik, M., 2013a. N-way model merging. In: Meyer, B., Baresi, L., Mezini, M. (Eds.), Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE '13, Saint Petersburg, Russian Federation, August 18-26, 2013, ACM, pp. 301–311. http://dx.doi.org/10.1145/2491411.2491446.

Rubin, J., Chechik, M., 2013b. A survey of feature location techniques. In: Reinhartz-Berger, I., Sturm, A., Clark, T., Cohen, S., Bettin, J. (Eds.), Domain Engineering, Product Lines, Languages, and Conceptual Models. Springer, New York, USA, pp. 29–58. http://dx.doi.org/10.1007/978-3-642-36654-3_2.

Salman, H.E., Seriai, A., Dony, C., 2013. Feature-to-code traceability in legacy software variants. In: Demirörs, O., Türetken, O. (Eds.), 39th Euromicro Conference on Software Engineering and Advanced Applications. SEAA 2013, Santander, Spain, September 4-6, 2013, IEEE Computer Society, pp. 57–61. http://dx.doi.org/10.1109/SEAA.2013.65.

Schultheiß, A., Bittner, P.M., El-Sharkawy, S., Thüm, T., Kehrer, T., 2022. Simulating the evolution of clone-and-own projects with VEVOS. In: International Conference on Evaluation and Assessment in Software Engineering 2022. In: EASE '22, Association for Computing Machinery, New York, NY, USA, pp. 231–236. http://dx.doi.org/10.1145/3530019.3534084.

Schultheiß, A., Bittner, P.M., Grunske, L., Thüm, T., Kehrer, T., 2021. Scalable N-way model matching using multi-dimensional search trees. In: 24th International Conference on Model Driven Engineering Languages and Systems. MODELS '21, Fukuoka, Japan, October 10-15, 2021, IEEE, pp. 1–12. http://dx.doi.org/10.1109/MODELS50736.2021.00010.

Shatnawi, A., Seriai, A., Sahraoui, H.A., 2016. Recovering architectural variability of a family of product variants. CoRR abs/1606.00137, arXiv:1606.00137.

Sonatype, 2011. Munge. https://github.com/sonatype/munge-maven-plugin.

Spanoudakis, G., Zisman, A., 2005. Software traceability: A roadmap. Handb. Softw. Eng. Knowl. Eng. 3, http://dx.doi.org/10.1142/9789812775245_0014.

Strüber, D., Mukelabai, M., Krüger, J., Fischer, S., Linsbauer, L., Martinez, J., Berger, T., 2019. Facing the truth: benchmarking the techniques for the evolution of variant-rich systems. In: 23rd Int. Systems and Software Product Line Conference. SPLC '19, Volume A, Paris, France, September 9-13, 2019, ACM, New York, USA, pp. 26:1–26:12. http://dx.doi.org/10.1145/3336294.3336302.

Thomas, S.W., Nagappan, M., Blostein, D., Hassan, A.E., 2013. The impact of classifier configuration and classifier combination on bug localization. IEEE Trans. Softw. Eng. 39 (10), 1427–1443. http://dx.doi.org/10.1109/TSE.2013.27.

Wilde, N., Buckellew, M., Page, H., Rajlich, V., 2001. A case study of feature location in unstructured legacy fortran code. In: Sousa, P., Ebert, J. (Eds.), 5th Conference on Software Maintenance and Reengineering. CSMR '01, Lisbon, Portugal, March 14-16, 2001, IEEE Computer Society, pp. 68–76. http://dx.doi.org/10.1109/.2001.914970.

Wilde, N., Scully, M.C., 1995. Software reconnaissance: Mapping program features to code. J. Softw. Maintenance: Res. Prac. 7 (1), 49–62. http://dx.doi.org/10.1002/smr.4360070105.

Wolfart, D., Assunção, W.K.G., Martinez, J., 2021. Variability debt: Characterization, causes and consequences. In: XX Brazilian Symposium on Software Quality. Association for Computing Machinery, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/3493244.3493250.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740. http://dx.doi.org/10.1109/TSE.2016.2521368.

Xing, Z., Xue, Y., Jarzabek, S., 2013. A large scale linux-kernel based benchmark for feature location research. In: 5th International Conference on Software Engineering. ICSE '13, IEEE, pp. 1311–1314. http://dx.doi.org/10.1109/ICSE.2013.6606705.

Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M., 2012. Feature identification from the source code of product variants. In: 16th European Conference on Software Maintenance and Reengineering. CSMR '12, Szeged, Hungary, March 27-30, 2012, IEEE Computer Society, pp. 417–422. http://dx.doi.org/10.1109/CSMR.2012.52.

Ziadi, T., Henard, C., Papadakis, M., Ziane, M., Traon, Y.L., 2014. Towards a language-independent approach for reverse-engineering of software product lines. In: Symposium on Applied Computing. SAC '14, pp. 1064–1071. http://dx.doi.org/10.1145/2554850.2554874.