Influence of Pure and Unit-Like Tests on SBFL Effectiveness: An Empirical Study

Attila Szatmári
Software Engineering Department
University of Szeged
Szeged, Hungary
szatma@inf.u-szeged.hu

Tamás Gergely
Software Engineering Department
University of Szeged
Szeged, Hungary
gertom@inf.u-szeged.hu

Árpád Beszédes
Software Engineering Department
University of Szeged
Szeged, Hungary
beszedes@inf.u-szeged.hu

***Abstract*—One of the most challenging and time-consuming aspects of debugging is identifying the exact location of the bug. We propose the concept of a Pure Unit Test (PUT), which, when fails, can unambiguously determine the location of the faulty method. Based on developer experience, we established three heuristics to evaluate the degree to which a test can be considered a unit test, if it cannot be considered as a PUT (we call these Unit-Like Tests, or ULTs). We examined how and when PUTs and ULTs affect Spectrum-Based Fault Localization efficiency. The results demonstrate that, for more complex systems, a higher proportion of unit tests in the relevant test cases can enhance the effectiveness of fault localization. When the number of PUTs is high enough, fault localization becomes trivial, in that case running SBFL is not necessary. Moreover, our findings indicate that different kinds of ULTs can have a large impact on the efficiency of fault localization, particularly for simpler bugs where they can quickly and effectively pinpoint the problem areas.**

***Index Terms*—Fault localization, Spectrum-Based Fault Localization, unit testing, test levels.**

I. Introduction

A technique called Spectrum-based Fault Localization [1]–[5] (SBFL) aims to automatically identify the exact origin of the bug. By analyzing statistical data on different program elements, this technique generates an ordered list of suspicious elements, which can guide developers during debugging. This paper deals with function-level granularity for analysis, meaning functions or class methods serve as the basic fault localization elements. This approach offers granularity higher than the commonly used statement-level; however, focusing on method-level granularity aligns with the scope of unit testing. The paper analyzes the influence of unit tests, which typically cover individual methods, and their variations (PUTs and ULTs) on SBFL efficiency. Furthermore, some argue that method-level granularity is more beneficial to users [5], [6].

There are cases where fault localization at method level granularity may not always be the best technique for fault detection. A particular scenario arises when the test suite of a software program contains only unit tests, which simplifies fault detection. The faulty method would be identified by a single failed test that calls it directly. However, the cost of creating and maintaining a unit-only test suite could easily outweigh the benefits. This would require extensive mocking, since very few methods operate completely in isolation. In general, industrial applications cover multiple levels of testing, even bug datasets consisting of real projects and bugs have different levels of testing in their test suites [7]–[10]. In addition, developers often do not fully follow the ISTQB rules when writing unit tests [11], leaving the test suite with only a small fraction of tests that strictly follow the definition of unit tests. This creates the challenge of recognizing tests that meet the unit test criteria and separating them from tests with similar features that do not meet the strict criteria. Many of these ”unit-like” tests are helpful in fault localization because they provide the benefits of unit testing, such as isolated coverage of specific functionality, without fully adhering to the formal definitions. To address this challenge, we introduce the concept of Pure Unit Tests (PUTs) as a precise criterion for identifying unit tests in the context of fault localization. In addition, we propose three other heuristics: Single Method Chain Test (SMC), Limited Method Chain Test (LMC) and Short Multichain Test (SMT) to evaluate how much of the test suite can be considered Unit-Like Tests (ULT).

The goal of this study is to determine what proportion of tests must be PUT or ULT for SBFL to achieve optimal effectiveness, that is, to place the faulty method near the top of the suspect list. By analyzing the role of both strict and relaxed unit test definitions, we aim to provide actionable insights into test suite design and offer practical guidance to developers on balancing test coverage and granularity to make debugging easier for developers. Our research is guided by the following key questions.

1) **RQ1: What percentage of tests should consist of pure unit tests to optimize the effectiveness of SBFL?** This question aims to determine the minimum proportion of PUTs necessary in relevant test cases which cover the same methods as the analyzed test, to maximize the accuracy of SBFL.
2) **RQ2: What is the relationship between the density of ULTs and the fault localization precision?** This question examines the optimal percentage of Unit-like tests required for SBFL to achieve optimal performance.

Our results indicate that more PUTs typically improve defect localization. However, when the test set consists entirely of PUTs or contains a failed test case that is a PUT, the faulty method can be clearly identified even without using complex techniques such as SBFL. As a recommendation for

future researchers, bugs revealed solely by failed PUTs could be excluded from bug benchmarks used for SBFL, as these cases do not benefit from fault localization techniques because they do not pose a meaningful challenge for SBFL to solve. As a result, including them in evaluations could obscure the true potential of novel methods, hindering their progress and validation.

II. Background

A. Test Levels

According to the ISTQB CTFL syllabus [12], which is a generally accepted source for testers, there are 5 levels of testing, i.e. **Component testing, Component integration testing, System testing, System integration testing, and Acceptance testing**. In this paper, we focus on component/unit tests. In addition to ISTQB, IEEE [13] defines a unit as:

- a separately testable element specified in the design of a computer software component,
- a logically separable part of a computer program, or
- a software component that is not subdivided into other components.

Therefore, a unit, by its very definition, cannot be divided into further components. Each unit test must individually cover only one such distinct unit. Usually in object-oriented programming languages, a unit is considered as a method/-function.

Several articles have been presented on what makes a good unit test [14]–[17]. Although these indicators and guidelines are helpful, the ambiguous and strict definition of unit tests often leads developers to relax the unit test rules to ensure that the system is tested. They prioritize having the system tested over strictly following the principles of unit tests.

B. Fault Localization

SBFL is a well-known technique for software fault localization that relies solely on test coverage and test results to calculate suspiciousness scores for program elements. The execution of test cases on program elements is documented to derive the spectra (i.e., test coverage and test results) for the program being tested. Using program spectra, for each program element **e** the following basic statistical numbers, called the spectrum metrics, are then computed:

- *ep*: number of passed tests covering **e**.
- *ef*: number of failed tests covering **e**.
- *np*: number of passed tests not covering **e**.
- *nf*: number of failed tests not covering **e**.

$$\text{score}(\mathbf{e}) = \frac{|ef|}{|ef|+|ep|} \tag{1}$$

After that, these spectrum metrics can be utilized by an SBFL formula which assigns a score to each program element **e**. In this study, we will use the Barinel [18] formula for evaluation (Equation 1). Then, the elements are sorted in descending order by their score. The rank (position in the sorted list) of the faulty element provides an effective comparison of the efficiency of various SBFL methods, as it clearly indicates the effort required by developers to examine the items in the list. Various studies [19], [20] have evaluated the reliability of fault localization and concluded that software engineers often examine only the top 3 or 5 positions on the list.

III. Motivation

Each unit test is designed to isolate a specific piece of functionality, ensuring that when a test fails, the issue can be pinpointed directly to the code under test. However, in real-life settings, test suites contain various levels of tests [21], [22]. Real-world software systems are inherently complex due to the numerous relationships and interactions between different components.

Although spectrum-based fault localization (SBFL) has proven useful for debugging, its statistical nature makes it inherently imprecise. As a result, developers often seek to optimize the composition of the test suite to ensure a balance that enhances the effectiveness of fault location. In particular, finding the right balance of unit tests is critical. These tests, by their very nature, simplify fault isolation, potentially reducing the need for complex SBFL techniques; if a failed test is a (true) unit test, it will isolate the faulty method and the developer can easily identify it.

In software testing, the concept of code coverage is often used to evaluate the effectiveness of tests. Using code coverage and call stack traces, we can identify the call chain of a test. A call chain is a sequence of methods in which each method calls the next in the list. Typically, code coverage and call chains help determine whether a test is a unit test or not. [23]

```
1  public class PythagoreanTheorem {
2      private double a, b;
3      public PythagoreanTheorem() {
4          this.a = 1.0;
5          this.b = 1.0;
6      }
7      public PythagoreanTheorem(double a, double b) {
8          this.a = a;
9          this.b = b;
10     }
11     public double calculateHypotenuse() {
12         return Math.sqrt(square(a) + square(b));
13     }
14     public double calculateRightTriangle() {
15         return (square(a) + square(b));
16     }
17     public double square(double x) {
18         return x + x; //bug, should be x * x
19     }
20 }
```

Listing 1. Example buggy code (Pythagorean Theorem)

```
1      PythagoreanTheorem theorem = new PythagoreanTheorem();
2      final double EPSILON = 0.05;
3      @Test
4      public void testCalculateHypotenuse() {
5          assertEquals(1.41, theorem.calculateHypotenuse(),
               EPSILON);
6      }
7      @Test
8      public void testIsRightTriangle() {
9          assertEquals(theorem.square(13), new
               PythagoreanTheorem(5, 12).calculateRightTriangle
               (), EPSILON);
10     }
11     @Test
12     public void testSquare() {
13         assertEquals(9.0, theorem.square(3), EPSILON);
14     }
15     @Test
16     public void testSquareWithParameterizedConstructor() {
17         assertEquals(9.0, new PythagoreanTheorem(3,4).square
               (3), EPSILON);
18     }
```

Listing 2. Example tests (with various test levels)

To illustrate the problem and clarify the concepts of PUT and ULTs, we present Listing 1, which includes a working example of the Pythagorean theorem, and Listing 2, which shows the associated tests. In this scenario, the bug exists in the *square* method, which mistakenly computes the square of a number as $x+x$ instead of the correct $x \times x$. This error has a domino effect, causing all tests that rely on the *square* method to fail. The three test cases provided exemplify different ULT categories, emphasizing the importance of PUTs to quickly and effectively pinpoint faults without SBFL.

The *testSquare* test case exemplifies a PUT by exclusively covering and validating the *square* method. Its failure directly identifies the faulty method without requiring additional analysis, demonstrating how PUTs simplify debugging. Conversely, tests like *testCalculateHypotenuse* and *testIsRightTriangle* represent higher-level tests, where the faulty method's location is less immediately apparent and would typically require SBFL to prioritize suspicious methods. Typically, minor helper methods like *square* are made private and static to signal to developers that they will not change the object's state. In such a case, the *testSquare* test would not exist, and other tests would reveal the fault. Even without unit tests in the suite, the defective method is still easily traceable by ULTs. Thus, understanding the role and proportion of PUTs and ULTs in test suites is essential to optimize fault localization processes.

IV. Method for Identifying Tests

Call chains in software engineering refer to a sequence of function or method calls where one function calls another, which in turn calls another, forming a chain of calls. These chains can be simple, involving just a few functions, or complex, including multiple modules and layers within a software system. In our approach, we collect all the distinct call chains that occur during the execution of T (a set of test cases), called the call chain set C. Furthermore, we maintain a chain set $C(t)$ for each individual test case $t \in T$ (indicating that t produces c if $c \in C(t)$). We consider c a single chain of a test if $c \in C(t)$. The collection of functions found within a chain c is represented by $F(c)$, where $F(C(t)) = \cup_{c \in C(t)} F(c)$.

To identify the **relevant test cases** for a given test case, we define $R(t) = \{r \in T | F(C(t)) \cap F(C(r)) \neq \emptyset\}$, so a list of test cases that cover the same methods as the given test. For example, one of the relevant test cases for *testSquare* in Listing 2 would be the *testIsRightTriangle*, since they both cover the *square* method.

Figure 1 represents the heuristics, specifically the PUT and ULT, which show the depth and structure of the call chain. Each arrow represents an invocation of a method, indicating its source and destination. Although the "Test" is shown in the figure, they do not belong to the call chains. With a falling PUT, SBFL and its measurements are unnecessary. ULTs are disjoint from PUT as the study focuses on the ratio of relevant tests. In ULTs only, LMC and SMC are not disjoint, as LMC is a subset of SMC. A PUT represents a stricter variation of unit testing, targeting only a single method, therefore, we can decide whether a test is PUT by using $PUT(t) \Leftrightarrow |C(t)| = 1 \wedge \forall c \in C(t) : len(c) = 1$. The key feature of a PUT is that its call chain length is exactly one, which means that the test triggers only the method being tested without involving any other methods, such as the *testSquare* in Listing 2.

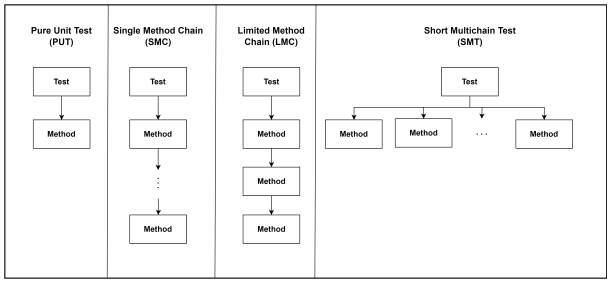

Fig. 1. Pure Unit Test (PUT) and Unit-Like Test (ULT) heuristics

Based on developer experience, we have established three heuristics to evaluate the degree to which a test can be considered a unit test. Due to the ambiguity of unit test definitions, only the PUT strictly meets the unit test requirements. Hence, the necessity for heuristics.

The Single Method Chain (SMC) focuses on minimizing the infection route of a bug to just one pathway. This implies that the developer only needs to examine a single sequence of method calls, which must surpass the length of the PUT, otherwise we would consider it classified as a PUT. Therefore, to decide whether a test is SMC, we use $SMC(t) \Leftrightarrow |C(t)| = 1 \wedge \forall c \in C(t) : 1 < len(c)$. Due to the limited scope that needs investigation, this is seen as a more relaxed interpretation of the unit test rules.

The Limited Method Chain (LMC) is a stricter rule than the SMC. Within this approach, we limit the sequence of method calls to three. Therefore, to decide whether a test is LMC, we use $LMC(t) \Leftrightarrow |C(t)| = 1 \wedge \forall c \in C(t) : 1 < len(c) \leq 3$. This rule is based on the idea that if the method is reasonably close to the test, it is considered a good estimate of a unit test. When software is refactored using the *Extract Method* design pattern, a single unit is often split into several smaller units. The *testCalculateHypotenuse* from Listing 2 falls into the SMC and LMC categories, because it calls the *calculateHypotenuse* method, which then invokes the *square* method (Listing 1).

Lastly, the Short Multichain Test (SMT) is based on the notion that while a test may involve multiple method calls, if these calls are at a single depth, they are likely related, making the area to be investigated manageable. For instance, when a test calls a constructor of an object and then interacts with that object (e.g., calling one of its methods), it tests the object but would not be considered a pure unit test by strict definition. Therefore, to decide whether a test is SMT, we use $SMT(t) \Leftrightarrow |C(t)| > 1 \wedge \forall c \in C(t) : len(c) = 1$. The *testSquareWithParameterizedConstructor* test from Listing 2 falls into this category, while the test invokes the *Parameterized Constructor* and *square* methods. Consequently, to decide whether a test is Unit-Like, we use the $ULT(t) = SMC(t) \vee LMC(t) \vee SMT(t)$ formula.

V. Evaluation

A. Benchmark

For our evaluation, we selected Defects4J (v2.0.0) [24], a widely recognized benchmark of Java programs and curated bugs used in fault localization research. This benchmark includes 17 open-source Java projects that contain 835 manually validated non-trivial real bugs. However, 45 bugs had to be omitted from the study due to instrumentation issues due to conflicts with specific versions of the programming language and libraries or inconsistent test results, such as flakiness. The whole project, Closure (160 bugs), was omitted because it had considerably higher testing levels than the other projects [7], i.e. the number of unit tests makes it an insufficient subject program for this study. Additionally, for 13 out of 835 bugs, the modification was limited to method addition (i.e., the committers did not alter an existing method during the rectification). We omitted these bugs from the study, because SBFL cannot localize the non-existent method in the buggy version.

TABLE I
Properties of subject programs

Subject	Number of bugs	Size (KLOC)	Number of tests	Number of methods	Number of PUT	Number of SMC	Number of LMC	Number of SMT
Chart	25	96	2.2k	5.2k	36.8	57.6	53.6	241.3
Cli	39	4	0.1k	0.3k	1.4	2.4	2.4	8.5
Codec	16	10	0.4k	0.5k	45.6	72.2	69.8	140.9
Compress	47	11	0.4k	1.5k	20.5	34.7	34.6	81.3
Csv	16	1	0.2	0.1k	4.9	18.3	15.9	37.9
Gson	15	12	0.9k	1.0k	3.2	9.2	9.2	20.5
JacksonCore	25	31	0.4k	1.8k	5.28	6.92	6.88	17.52
JacksonDatabind	101	4	1.6k	6.9k	7.2	10.2	10.2	25.2
JacksonXml	5	6	0.1k	0.5k	0	0	0	0.2
Jsoup	89	14	0.5k	1.4k	5.8	10.8	10.8	27.8
JxPath	21	21	0.3k	1.7k	0	0	0	0.2
Lang	61	22	2.3k	2.4k	366.1	644.6	608.7	956
Math	104	84	4.4k	6.4k	47.7	98	91.4	246.7
Mockito	27	11	1.3k	1.4k	4.5	6.5	6.5	22.8
Time	26	28	4.0k	3.6k	26.2	39.9	39.9	208.6
All	**617**	**355**	**19.1k**	**34.7k**	**575.3**	**1011.4**	**959.8**	**2035.7**

In total, 617 defects were incorporated into the final dataset. Table I presents each project alongside its main attributes. For each bug, the average number of tests, methods, PUTs, SMCs, LMCs, and SMTs are provided for the project.

B. Generating Suspiciousness Rank Lists

To determine the suspiciousness values of code elements, we relied on the Barinel metric, as discussed in Section II-B. Our implementation adopts the average position tie-breaking strategy, which is the most commonly chosen approach compared to best-case and worst-case scenarios. To address the division by zero issue that can impact many formulas, we applied the $c+\epsilon$ method [25], where c represents any coefficient in the formula and ϵ is the smallest representable floating-point number.

Several studies indicate that developers typically examine only the initial 5 or 10 entries in the recommendation lists (rank) when employing fault localization algorithms, before stopping their ranking evaluation [19], [20]. The group of metrics that differentiates bugs with the minimum rank of faulty elements being less than or equal to N is often referred to as $topN$ or $acc@N$ [26]. This metric signifies the number of bugs accurately pinpointed within the top N elements of the ranking lists. As the value of N decreases, the performance of SBFL improves. Common selections for N include 1, 3, 5, and 10. Code elements placed beyond the top 10 are categorized as Other.

VI. Results

A. Optimal Proportion of Pure Unit Tests for Fault Localization Effectiveness (RQ1)

The main goal of this research question is to find out what is the percentage of pure unit test ratio in the relevant test cases where Spectrum-based Fault Localization is needed. To address this research question, we examine only the ranks of those bugs for which the PUT percentage in relevant test cases is above 0, indicating that at least one PUT test case covers the buggy method. The reason for this is that test cases that are not unit-level have varying success in detecting bugs. Certain bugs e.g. bugs in program workflow can be easily revealed by integration and system tests, while others cannot.

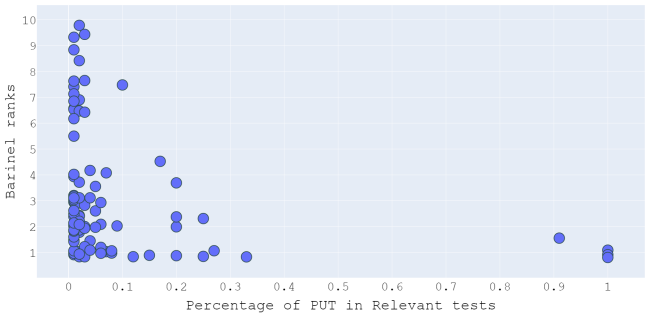

Fig. 2. Percentage of PUT in relevant test cases top 10

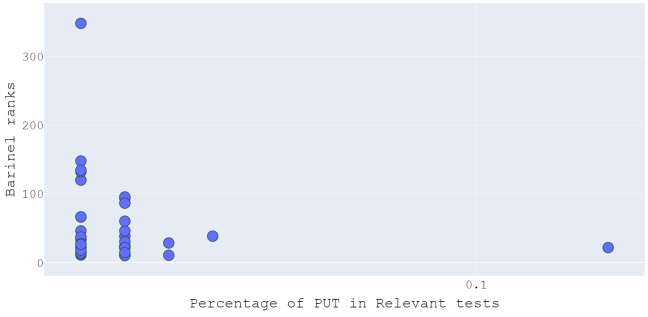

Fig. 3. Percentage of PUT in relevant test cases Others

We analyze the PUT percentage distributions in disjoint top N rank categories in order to see how much of the relevant test cases need to be pure unit tests so that SBFL provides accurate localization. Figure 2 shows the percentage of PUTs in relevant test cases where buggy methods rank in the top 10 category on the list of suspicious elements. Several data points show high PUT percentages in the top 1 category, suggesting that a high percentage of relevant unit test cases contribute to effective fault localization.

However, there are also instances where the PUT percentage is near 0%, indicating that while a high PUT percentage

often suggests precise fault localization, SBFL can effectively identify faulty methods at different test levels. Transitioning from top 1 to top 3, with relatively high PUT percentages in the relevant tests, SBFL always ranks the buggy methods in the top-3 positions.

The number of PUTs decreases drastically as we move towards lower ranks, which means that SBFL will not rank buggy methods with high PUT rate of relevant tests in top-5 and top-10 or worse. Finally, Figure 3 shows the distribution of PUT percentages for tests ranked outside the top 10. Most data points are concentrated below 5%, with very few exceptions; therefore, buggy methods that are not accurately localized have a low percentage of containing pure unit tests in their relevant test case set. Combining these results, for the SBFL to consistently place buggy methods in the top 3, at least 20% of the relevant tests must be PUT. Achieving this requires categorizing each test based on defined heuristics and ensuring that for every method, at least 20% of its relevant test cases are PUT. For example, if a method has 5 relevant test cases, at least 1 of them should be a PUT to maximize the likelihood that SBFL places it in the top 3. If this threshold is not met, developers may need to write additional PUTs to improve fault localization accuracy. However, pure unit tests are not all feasible or inexpensive to create and maintain, as many methods do not operate completely in isolation. Developers should weigh the effort invested in the significant amount of work they put into making a pure unit test against the cost of maintainability later, e.g. debugging costs.

Answer to RQ1: Having over 20% pure unit tests in the relevant test cases enhances SBFL's precision, making sure buggy methods consistently ranked in the top 3. However, when there is a failing test that is a PUT, the buggy method can be easily pinpointed.

B. Assessing the Impact of Relaxed Unit Tests on Fault Localization Efficiency (RQ2)

Although a high proportion of the PUT in the test suite indicates accurate SBFL, it is very strict and tests are rarely written to follow it exactly, especially when unit testing methods require extensive mocking. Therefore, using heuristics to approximate unit tests is a logical choice.

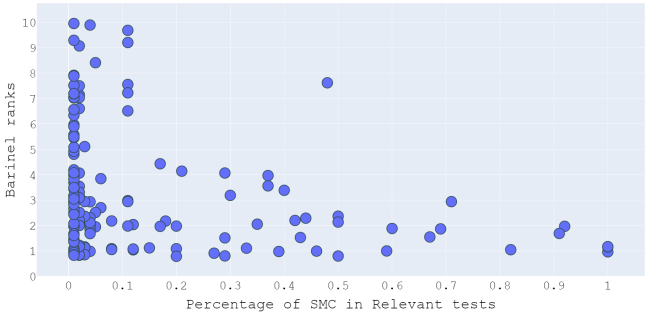

Fig. 4. Percentage of SMC in relevant test cases top 10

Figure 4 shows that the proportion of SMCs within the relevant test cases of faulty methods ranked between the top 10 bugs is overall considerably higher. This means that, compared to Figure 2, more faulty methods with a relatively low percentage are placed in the top 1 and 3 than in the case of PUT.

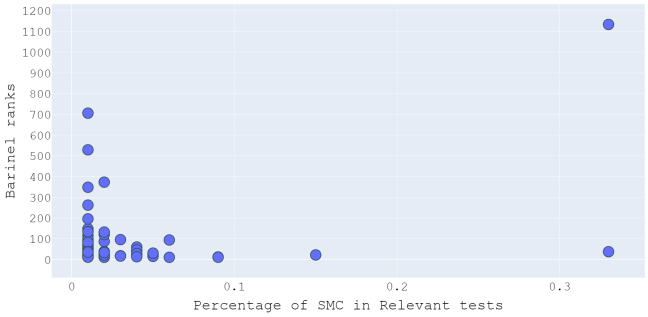

Fig. 5. Percentage of SMC in relevant test cases Others

On the other hand, Figure 5 shows the SMC percentages in the relevant test cases, when SBFL ranked the faulty methods outside the top 10. Overall, the percentages here are relatively low except for a few outliers with approximately 30%. When comparing these results, if faulty methods are covered by at least 50% of the SMC tests, SBFL will always place them in the top 3, as relevant tests of lower-ranked erroneous methods never reach that proportion.

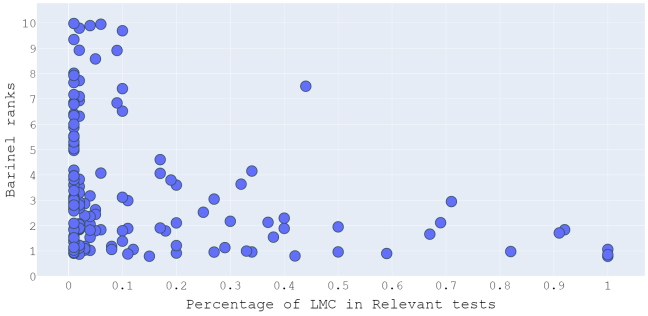

Fig. 6. Percentage of LMC in relevant test cases top 10

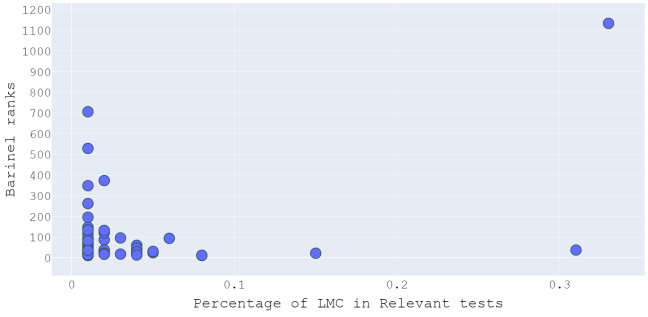

Fig. 7. Percentage of LMC in relevant test cases Others

The LMC is a stricter rule than the SMC, where only short single-method chain tests are categorized. Similarly to the

SMC plots, the faulty methods in the top 10 are spread over different percentages of LMC (Figure 6). High percentages of LMC can be associated with a higher likelihood of bugs being ranked in the top 1 and top 3, similar to what was observed for SMC. However, the percentages of LMC in the tests for those bugs ranked outside the top 10 are generally low (Figure 7). Comparing the results of the LMC percentages, we can see that at least 50% of the relevant test cases need to be LMCs for SBFL to always place them in the top 3, just as with SMCs.

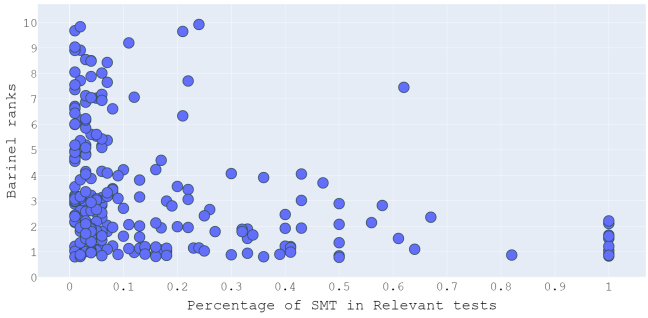

Fig. 8. Percentage of SMT in relevant test cases top 10

The SMT graphs in Figure 8 demonstrate a broader distribution compared to the SMC and LMC graphs, as SMT has less similarity to PUT than the other heuristics, resulting in a larger number of tests falling into this category.

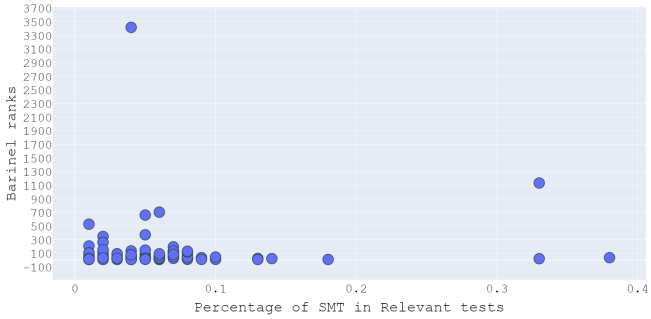

Fig. 9. Percentage of SMT in relevant test cases Others

Figure 9 shows the SMT percentages for bugs that are not ranked in the top 10. The percentages are generally lower, with the highest point around 40%. This shows that when the SMC percentage in the relevant tests is sufficiently high ($\geq$ 60%), SBFL will rank them within the top 3. Unit-like tests are also more feasible than pure unit tests for complex software because they are based on less strict rules, but retain the benefits of unit testing. Although methods need to be covered by a higher percentage of ULTs to ensure that SBFL always gives good results (top 3), it is more sustainable and less expensive than creating pure unit tests, since less mocking is required. With these results in mind, we answer the second research question.

Answer to RQ2: ULTs are vital for improving SBFL efficiency. Notably, when ULTs exceed 50% for SMC and LMC, and 60% for SMT, SBFL consistently ranks faulty methods in the top 3.

VII. Related Works

A. Spectrum-based Fault Localization

SBFL techniques are still striving to be utilized in real-world scenarios [20], [27]–[29]. Although many SBFL suspiciousness measures have been suggested, we used Barinel in this work, which has been shown to perform exceptionally well [29]. Wu et al. integrate the call stack with static call graph data and subsequently compute the suspiciousness scores for the functions [30]. Analogous ideas about function call chains have been investigated by other researchers [31], [32], although not at this level of detail and not centered on SBFL applications. Beszédes et al. [33], [34] used call chains to help improve SBFL's fault localization accuracy. Additionally, Vancsics et al. [35], [36] used the frequency of method calls in call chains to improve SBFL.

B. Unit test categorization

Beck et al. [37] suggest that using isolated tests can simplify debugging simpler and create systems with strong cohesion and loose coupling. Nierstrasz et al. [38] developed a classification system for unit tests. Van Deursen et al. [39] specifically discuss unit tests focusing on a single method, categorizing them using bad smells like indirect testing, which we would classify as independent tests. In another study [40], Van Deursen et al. investigate the connections between testing and refactoring, suggesting that code refactoring should be accompanied by test refactoring. Certain test smell refactorings align with ULTs, such as "Indirect Testing," where test class methods test other objects. This aligns with SMTs and, despite complicating the test-production code relationship, is useful for narrowing the SBFL search space. Trautsch et al. [11] performed a study on Python programs to examine whether developers implement unit tests that follow the ISTQB and IEEE standards in practical projects. They found that while developers believe they are writing unit tests, in reality, they are doing less than they think, with most projects having only a minimal number of unit tests. This observation matches our findings on the number of unit tests in projects. Orellana et al. [41] investigated the difference between unit and integration tests using the TravisTorrent dataset. They discovered that unit tests detect a higher number of defects compared to integration tests. This emphasizes the importance of maintaining a high proportion of unit tests (PUT) in the relevant test cases and ideally throughout the entire test suite.

C. Using test suite manipulation to improve Fault Localization

Perez et al. [42] introduced the DDU metric, which focuses on diagnostic informativeness through a general test suite evaluation, our work targets the specific impact of Pure Unit Tests (PUTs) and Unit-Like Tests (ULTs) on SBFL efficiency. Xuan et al. [43] transformed the tests into unit tests to enhance spectrum-based fault localization. Their empirical findings demonstrate that test case purification can substantially

improve the original fault localization methods. The results indicate that test case purification benefits six different fault localization techniques. Our results support this, showing that SBFL works better with a larger number of unit tests. Jiang et al. [44] present an empirical evaluation of the effectiveness of adequate test suites in supporting fault localization, with a particular focus on the integration of test case prioritization and statistical fault localization techniques.

VIII. Threats to Validity

The main threat to the *internal validity* of this study is the heuristics we propose to define Pure Unit Tests (PUT) and Unit-Like tests (SMC, LMC, SMT) that are based on specific assumptions about what constitutes a "unit" in testing. These definitions, while based on conclusions from the existing literature, may not align perfectly with all developer practices, potentially leading to incorrect classification of tests and affecting our results. In addition, evaluating suspiciousness rankings based on single-bug isolation might not be valid for complex systems with multiple bugs. Moreover, setting SBFL granularity to the method level can introduce biases, as conclusions about not needing SBFL when a failing test is a PUT may not apply to finer granularities like statements or branches. For in-line or very short methods, the conclusion remains valid, but otherwise, based on this result, researchers can incorporate a new SBFL algorithm that has narrowed down search spaces.

Regarding *external validity*, even though the study's evaluation was conducted on 617 real defects from Java programs, it is still preferable to replicate the research using a larger and potentially more varied dataset. Replicating this study on various codebases from multiple domains and programming paradigms would help verify the generalizability of our conclusions. The bugs analyzed may not represent the full diversity of real-world bugs, as they originate from open-source projects that follow certain standards. Defects in other software types, such as embedded systems or commercial software, may show different results.

IX. Conclusions and Future Work

In this paper, we investigated how different heuristics for unit tests correlate with the accuracy of SBFL. Therefore, we proposed heuristics to approximate unit tests based on their method call chains. These are the Pure Unit Test (PUT), the Single Method Chain (SMC), the Limited Method Chain (LMC), and the Short Multichain Test (SMT). To examine the proportion of such tests in the relevant tests of defects (tests covering the same methods as the analyzed tests), we conducted a study on Defects4J. The results showed that including more than 20% PUTs in the test cases significantly improves the accuracy of the SBFL, placing the faulty methods in the top 3 of the suspicion rankings. However, if one of the failed tests is a PUT, then identifying the faulty method becomes easy even without using SBFL. We also found that ULTs developed with heuristics such as SMC, LMC, and SMT greatly increase the efficiency of SBFL. When ULTs make up a large fraction of the relevant tests ($\geq 50\%$ for SMC and LMC, $\geq 60\%$ for SMT), SBFL will accurately rank faulty methods in the top 3, providing accurate information to developers.

Our future research will look at cases where there are no PUTs or ULTs in the test suite, especially in high-level tests such as integration or system tests. Pure unit tests are rather rare in the industry, making it difficult to maintain many in the test suite. Our focus is on the role of PUTs and ULTs in SBFL, providing insights for improving test design and defect localization. Developers should aim for a balanced mix of PUTs and ULTs. Furthermore, the heuristics allow these tests to be identified by automated tools. Our findings suggest that test suite analysis tools can use PUT and ULT heuristics to improve SBFL processes, such as evaluating the presence of PUTs during CI/CD to prioritize tests based on fault isolation capability. Future studies could develop dynamic methods for maintaining test suites in evolving software. Further research could refine the heuristics to mimic unit tests using features such as keywords or code patterns.

Data Availability

In order to encourage the reproducibility and replicability of this study, we have made the algorithm implementations and the qualitative evaluation results available in an online appendix https://doi.org/10.5281/zenodo.13880592

References

[1] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," 2017. [Online]. Available: https://arxiv.org/abs/1607.04347

[2] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A Literature Review," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, sep 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2659118.2659125

[3] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, aug 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7390282/

[4] Jeongho Kim and Eunseok Lee, "Empirical evaluation of existing algorithms of spectrum based fault localization," in *The International Conference on Information Networking (ICOIN)*. IEEE, feb 2014, pp. 346–351. [Online]. Available: http://ieeexplore.ieee.org/document/6799702/

[5] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2021.

[6] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," 2016, Conference paper, p. 177 – 188, cited by: 170; All Open Access, Green Open Access. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84984911582&doi=10.1145%2f2931037.2931049&partnerID=40&md5=0adb9a1b8e9217bacb884fdf15a1adea

[7] R. Abou Assi, C. Trad, M. Maalouf, and W. Masri, "Coincidental correctness in the defects4j benchmark," *Software Testing, Verification and Reliability*, vol. 29, no. 3, p. e1696, 2019, e1696 STVR-18-0045.R2. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1696

[8] R. Widyasari, G. A. A. Prana, S. A. Haryono, S. Wang, and D. Lo, "Real world projects, real faults: evaluating spectrum based fault localization techniques on python projects," *Empirical Software Engineering*, vol. 27, no. 6, p. 147, Aug 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10189-4

[9] M. Rezaalipour and C. A. Furia, "An empirical study of fault localization in python programs," *Empirical Software Engineering*, vol. 29, no. 4, p. 92, Jun 2024. [Online]. Available: https://doi.org/10.1007/s10664-024-10475-3

[10] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Árpád Beszédes, R. Ferenc, and A. Mesbah, "BugJS: A benchmark of javascript bugs," in *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019.

[11] F. Trautsch and J. Grabowski, "Are there any unit tests? an empirical study on unit testing in open source python projects," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 207–218.

[12] International Software Testing Qualifications Board, "Cerfified tester foundation level syllabus – v4.0.1," 2024. [Online]. Available: https://www.istqb.org/certifications/certified-tester-foundation-level-ctfl-v4-0/

[13] "ISO/IEC/IEEE International Standard – Systems and software engineering – Vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017.

[14] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[15] E. Daka and G. Fraser, "A survey on unit testing practices and problems," in *2014 IEEE 25th International Symposium on Software Reliability Engineering*, 2014, pp. 201–211.

[16] M. Aniche, C. Treude, and A. Zaidman, "How developers engineer test cases: An observational study," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4925–4946, 2022.

[17] G. R. Bai and K. T. Stolee, "Improving students' testing practices," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2020, pp. 218–221.

[18] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-based multiple fault localization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 88–99.

[19] X. Xia, L. Bao, D. Lo, and S. Li, ""automated debugging considered harmful" considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 267–278.

[20] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176. [Online]. Available: https://doi.org/10.1145/2931037.2931051

[21] M. A. Umar, "Comprehensive study of software testing: Categories, levels, techniques, and types," *International Journal of Advance Research, Ideas and Innovations in Technology*, vol. 5, pp. 32–40, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:208534004

[22] I. Hooda and R. Singh Chhillar, "Software Test Process, Testing Types and Techniques," *International Journal of Computer Applications*, vol. 111, no. 13, pp. 10–14, Feb. 2015.

[23] T. Kanstrén, "Towards a deeper understanding of test coverage," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 1, pp. 59–76, 2008. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.362

[24] G. Gay and R. Just, "Defects4j as a challenge case for the search-based software engineering community," in *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 255–261.

[25] D. Vince, A. Szatmári, Á. Kiss, and Á. Beszédes, "Division by zero: Threats and effects in spectrum-based fault localization formulas," in *Proceedings of the 22nd IEEE International Conference on Software Quality, Reliability, and Security (QRS'22)*, Dec. 2022, pp. 221–230.

[26] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 199–209. [Online]. Available: https://doi.org/10.1145/2001420.2001445

[27] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 380–383.

[28] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 314–324. [Online]. Available: https://doi.org/10.1145/2483760.2483767

[29] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009, sI: TAIC PART 2007 and MUTATION 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121209001319

[30] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 204–214. [Online]. Available: https://doi.org/10.1145/2610384.2610386

[31] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in java," *SIGSOFT Softw. Eng. Notes*, vol. 29, no. 4, p. 1–11, jul 2004. [Online]. Available: https://doi.org/10.1145/1013886.1007514

[32] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: Association for Computing Machinery, 1997, p. 85–96. [Online]. Available: https://doi.org/10.1145/258915.258924

[33] Á. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization," in *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'20)*, Feb. 2020, pp. 468–479.

[34] Q. I. Sarhan, B. Vancsics, and Á. Beszédes, "Method calls frequency-based tie-breaking strategy for software fault localization," in *Proceedings of the 21st IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'21)*, Sep. 2021, pp. 103–113.

[35] B. Vancsics, F. Horváth, A. Szatmári, and Á. Beszédes, "Call frequency-based fault localization," in *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21)*, Mar. 2021, pp. 365–376.

[36] ——, "Fault localization using function call frequencies," *The Journal of Systems and Software*, vol. 193, p. 111429, 2022.

[37] K. Beck and E. Gamma, *Test-infected: programmers love writing tests*. USA: Cambridge University Press, 2000, p. 357–376.

[38] M. Alli, M. Lanza, and O. Nierstrasz, "Towards a taxonomy of unit tests," 08 2004.

[39] A. Deursen, L. M. Moonen, A. Bergh, and G. Kok, "Refactoring test code," NLD, Tech. Rep., 2001.

[40] A. Deursen and L. Moonen, "The video store revisited - thoughts on refactoring and testing," 06 2002.

[41] G. Orellana, G. Laghari, A. Murgia, and S. Demeyer, "On the differences between unit and integration testing in the travistorrent dataset," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 451–454.

[42] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 654–664.

[43] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 52–63. [Online]. Available: https://doi.org/10.1145/2635868.2635906

[44] B. Jiang, W. Chan, and T. Tse, "On practical adequate test suites for integrated test case prioritization and fault localization," in *2011 11th International Conference on Quality Software*, 2011, pp. 21–30.