Call Frequency-Based Fault Localization

Béla Vancsics
Software Engineering Department
University of Szeged
Szeged, Hungary
vancsics@inf.u-szeged.hu

Ferenc Horváth
Software Engineering Department
University of Szeged
Szeged, Hungary
hferenc@inf.u-szeged.hu

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

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

***Abstract*—Spectrum-Based Fault Localization (SBFL), in its basic form, uses only local information about a program element's (such as a method's) coverage to predict its faultiness, and rarely is any additional (contextual) information leveraged about the element itself, nor the test cases. As such an additional context, in the presented approach, we rely on the *frequency of the investigated method occurring in call stack instances* during the course of executing the failing test cases. The basic intuition is that if a method is called in many different contexts during a failing test case, it will be more probable to be accountable for the fault compared to other methods. We empirically evaluated the fault localization capability of the approach compared to five traditional SBFL techniques using the bug benchmark Defects4J. We found that the new algorithms (i) find the location of bugs at higher rank positions more often, (ii) can achieve 38%–52% rank position improvement compared to the baseline algorithms with statistical significance, and (iii) place more items at the top-10 positions of the suspiciousness ranking.**

***Index Terms*—Spectrum-Based Fault Localization, Method Call Frequency, Call Stacks, Testing, Debugging**

I. Introduction

Fault Localization (FL) is inevitable, and often a very difficult and time consuming step during debugging, hence, its importance in maintenance and evolution is unquestionable. This paper deals with a class of automated FL methods, which are based on the notion of the "program spectrum" [1], [2] (*Spectrum-Based Fault Localization–SBFL*). Spectrum refers to the statistical information about how the executed test cases relate to program elements and what are their outcomes (hence, also Statistical Fault Localization is a commonly used term for these methods) [3]–[7].

Several types of spectra have been defined over the past decades, but the most common approach is to use the so-called "hit-based" spectrum. This refers to the simple binary information if a code element (e. g., statement or function in a procedural, or method in an object-oriented context) is covered during the execution of a test case or not. Using this information, the basic intuition is that those code elements are more suspicious to contain a fault that are exercised by comparably more failing test cases than passing ones, while non-suspicious elements are traversed mostly by passing tests. Dedicated *formulas* are used to calculate the *suspiciousness* levels, which in turn *rank* the code elements to provide a debugging aid to the developer.

The traditional hit-based methods are generally seen as providing modest performance in terms of ranking precision [8]–[11], but other issues have also been identified [6], [12]–[14], which contributes to the fact that automated fault localization is still ignored by industry for the most part. Consequently, researchers proposed different approaches that go beyond the hit-based spectrum and utilize other information available that could help improve the overall ranking performance [15]–[19].

However, it is quite intriguing that the most straightforward extension of the hit-based spectrum, the *count-based* one is rarely investigated. Some early results have been published by Harrold et al [20], [21], but more recently Abreu et al. [22] concluded that counts do not provide additional value compared to hits. There might be several reasons to this phenomenon, but a popular explanation is that many times repeating program elements during execution (due to loops) may lead to unwanted distortion in the test case statistics.

In this paper, we propose a method to improve hit-based spectra using a more advanced count-based approach. In particular, we do not count all occurrences of a program element during execution but only those that occur in *unique call contexts*. Our algorithm is at method-level granularity, meaning that the basic program element considered for fault localization is a function or a method. As a call context, we rely on *call stack instances*. In particular, we build on observing the unique deepest call stack instances upon executing a test case, and count the occurrences of methods in these (hence we refer to the approach as *Call Frequency-Based Fault Localization*). This way, repeating patterns of method invocations due to, e.g., loops are excluded and only the relevant call context patterns are considered.

We applied this approach on five traditional hit-based SBFL formulas by replacing their crucial element, the numerator, and empirically investigated if it brings improvement to the base algorithms' performance. We found that, for all but one formulas, this modification gained statistically significant improvement in the average rank position of the faulty method in the range 38%–52%. Other improvement measures also showed that call frequency-based formulas can notably enhance hit-based SBFL, such as in the number of best ranking positions and high-rank position improvements. A surprising result is that one of the base formulas that is traditionally seen as one of the best ones, $Ochiai$ [23] did not improve with this new information, but a usually less successful one, $Jaccard$ [24], produced such great improvements that it outperformed all the other candidates on all subjects and in all measurements.

The remainder of the paper is organized as follows. In

Section II, we introduce our novel approach building on traditional methods. Section III contains the description of our empirical evalution with results in Section IV. This is followed by a discussion in Section V, overview of related work in Section VI, and conclusions in Section VII.

II. Call Frequency-Based Fault Localization Algorithms

This paper deals with method-level granularity for the analysis, which means that the basic element for localizing a fault is a function or method of a class. It is a higher granularity than the currently prevalent statement level approach, but for our purposes it has at least two advantages. First, we may define a more global contextual information about the investigated program element, and it is scalable to large programs and executions. Furthermore, there are some views that method-level is a better granularity for the users as well [23], [25].

The traditional hit-based SBFL methods rely purely on local information about a program element's (in our case, a method's) coverage by the test cases, and no additional (contextual) information is leveraged about the element itself, nor the test cases. In this paper, we use as an additional context the *frequency of the investigated method occurring in call stack instances* during the course of executing the failing test cases. The basic intuition is that if a method is called in many different contexts during a failing test case, it will be more probable to be accountable for the fault compared to other methods. However, we do not use the simple count of method calls because it has been shown that a simple count-based modification does not bring additional value to the fault localization process [22].

In this section, we first overview the basic hit-based SBFL approach together with a selection of well-known formulas, which we chose so that they are the most appropriate for our extension. Then, we introduce our call-frequency based method which replaces certain parts of the traditional formulas by adding this new contextual information.

We will use a running example throughout this section which is shown in Figures 1 and 2 (method (g) contains the bug). It is an adapted version of the example from [26] with added features to illustrate the benefits of the call frequencies.

A. Hit-Based Methods

Hit-based SBFL uses a binary coverage matrix ($\mathbf{H}$) and a test results vector ($\mathbf{R}$) as the basic data structures to calculate the suspiciousness scores for program elements [24], [27]. In the coverage matrix, the rows represent the tests and the columns are the program elements, methods in our case. The value of an element in the matrix is 1 or 0, depending on whether the method is covered by the test or not, respectively (denoted by $h_{t,m} \in \{0,1\}$, where $m \in$ methods and $t \in$ tests). An element in the results vector is 0 if the given test passed, otherwise it is 1 ($r_{t,m} \in \{0,1\}$).

In our example, $\{a,b,f,g\}$ is the set of program elements (methods), and $\{t1,t2,t3,t4\}$ are the test cases, with the resulting coverage matrix and results vector shown in Table I.

```
1 public class Example{
2   private int _x = 0;
3   private int _s = 0;
4   public int x() {return _x;}
5
6   public void a(int i){
7     _s = 0;
8     if (i==0) return;
9     if (i<0)
10      for (int y=0;y<=4;y++)
11        f(i);
12    else
13      g(i);
14  }
15
16  public void b(int i){
17    _s = 1;
18    if (i==0) return;
19    if (i<0)
20      a(Math.abs(i));
21    else
22      for (int y=0;y<=1;y++)
23        g(i);
24  }
25
26  private void f(int i){
27    _x -= i;
28  }
29
30  private void g(int i){
31    _x += (i+_s); //should be _x += i;
32  }
33 }
```

Fig. 1: Running example – program

```
1 public class ExampleTest {
2  @Test public void t1() {
3   Example tester = new Example();
4   tester.a(-1);
5   tester.a(1);
6   tester.b(1);
7   assertEquals(9, tester.x()); // failed -> 8
8   }
9
10  @Test public void t2() {
11  Example tester = new Example();
12  tester.a(1);
13  tester.b(1);
14  assertEquals(4, tester.x()); // failed -> 3
15  }
16
17  @Test public void t3() {
18  Example tester = new Example();
19  tester.a(1);
20  tester.b(0);
21  assertEquals(1, tester.x());
22  }
23
24  @Test public void t4() {
25  Example tester = new Example();
26  tester.a(-1);
27  tester.a(1);
28  tester.b(-1);
29  assertEquals(7, tester.x());
30  }
31 }
```

Fig. 2: Running example – test cases

All basic hit-based SBFL formulas rely on four fundamental statistics that are calculated from the spectrum. For each element m, the following sets are obtained, whose cardinalities are then used in the formulas:

a) m_{ep}: set of passed tests covered by m
b) m_{ef}: set of failed tests covered by m
c) m_{nf}: set of failed tests not covered by m
d) m_{np}: set of passed tests not covered by m

The four basic statistics for the methods in our example are also presented in Table I.

There is a plethora of different formulas proposed by researchers using these basic statistics (good summaries can be

TABLE I: Coverage hit spectrum (with four basic statistics) and frequency spectrum

		hit matrix				frequency matrix				results
		a	b	f	g	a	b	f	g	
coverage	t1	1	1	1	1	2	1	1	2	1
	t2	1	1	0	1	1	1	0	2	1
	t3	1	1	0	1	1	1	0	1	0
	t4	1	1	1	1	3	1	1	2	0
basic statistics	ef	2	2	1	2					
	ep	2	2	1	2					
	nf	0	0	1	0					
	np	0	0	1	0					

found in [28], [29]). Moreover, some researchers experimented with automatically deriving new formulas [30], [31]. For this paper, we selected five well-known formulas to experiment with, which are shown in Table II. As one can notice, all of the chosen formulas share a common property that their numerator is based on the same value, $|m_{ef}|$. In fact, all formulas rely on this basic statistic in one way or the other since it is very straightforward that the suspiciousness of a program element is mostly affected by how many failing test cases are going through it. The denominators use various approaches to incorporate the non-suspicious cases, hence reduce the scores, typically by using m_{ep} in some arithmetic combination with the other metrics. The fact that all these formulas share the same numerator will enable us to propose a common way of extending them using our call frequency counts, and their straightforward comparison in the experimental study.

TABLE II: Hit-based SBFL formulas

Barinel (B) [32]:	$\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{ep}\rvert}$,
Jaccard (J) [24]:	$\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$,
Ochiai (O) [24]:	$\frac{\lvert m_{ef}\rvert}{\sqrt{(\lvert m_{ef}\rvert + \lvert m_{nf}\rvert) \cdot (\lvert m_{ef}\rvert + \lvert m_{ep}\rvert)}}$,
Russell-Rao (R) [29]:	$\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert + \lvert m_{np}\rvert}$,
Sørensen-Dice (S) [33]:	$\frac{2 \cdot \lvert m_{ef}\rvert}{2 \cdot \lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$,

The corresponding suspiciousness scores for each method in our example and for each formula are shown in the upper part of Table IV. As can be seen, the buggy method (g) is hardly distinguishable from the other methods based on the suspiciousness scores. Method f would be at the last place in the suspiciousness list based on almost all metrics, except $Barinel$. Hence, a programmer could rule this method out during debugging. As for the other three remaining methods, there would be a tie in the ranking.

B. Proposed Technique

Our idea to add contextual information to the simple hit-based FL formulas is to incorporate how often a specific method has been called (directly or indirectly) and in which context from the test cases. A naïve technique to incorporate such a call frequency would be to count the number of invocations of the methods while executing a test case. This would correspond to the simple *count-based* SBFL. However, it is easy to see that with this approach, in a situation where a method is called directly or indirectly from a loop, it will be counted potentially many times. If this call belongs to a failing test case, then it will unnecessarily raise the suspiciousness score of the affected non-faulty methods, which could cause that the actually faulty elements (that are executed less times) remain hidden.

This can be seen in the example in Fig. 1 and Fig. 2 where method f was called five times by $t1$ (failed test) in a for loop, as opposed to g (faulty method), which was called four times in total by the two ($t1$ and $t2$) failed tests. This is illustrated with $t1$ and its *Call Tree* in the upper part of Figure 3. Higher execution count of a non-faulty method f compared to the faulty g would result in a simple count-based SBFL approach to miss successfully locating g.

Instead, in our approach we rely on *method call stack traces*, and their different snapshots during the execution of a test case. Call stack traces are artifacts occurring during program execution which are well-known to programmers who perform debugging, and can show, for instance, that a method may fail if called from one place and perform successfully when called from another. There is empirical evidence that stack traces help developers fix bugs [34]. Furthermore, Zou *et al.* [23] showed that stack traces can be used to locate *crash-faults*.

In particular, we extract *unique deepest call stacks – UDCS-s*, which means that we build a particular instance of a call stack snapshot until additional methods are transitively called, and stop when a method returns. This way, repeated invocations of methods from the same calling context (due to loops) will not induce new call stack instances. UDCS is a very similar concept to *call chains* introduced by Beszédes *et al.* [26]. Similar concepts have been explored by other researchers as well [22], [35], [36], however not in this detail and not with these applications in focus.

We define UDCS more precisely as follows. Let M be the set of methods in a program P, and T a set of test cases used to test P. Then, a *unique deepest call stack* c is a sequence of methods $m_1 \rightarrow m_2 \rightarrow \cdots \rightarrow m_n$ ($m_i \in M$), which occur during the execution of some test case $t \in T$, and for which:

- m_1 is the entry point called by t,
- each m_i directly calls m_{i+1} ($0 < i < n$), and
- m_n returns without calling further methods in that sequence (we call such a method *stack-terminating*).

UDCS-s provide an opportunity to use coverage not only as "hit/no hit" data but to compute the frequency of methods occurring in such stacks and use this number in the SBFL formulas instead of the basic statistics introduced earlier. More

precisely, for a method m under investigation, we calculate the sum of its frequencies occurring in different UDCS-s generated by the relevant test cases. In particular, if we summarize these frequencies for each failing test case, we can get a substitute for m_{ef}, which is, in a sense, its weighted version that incorporates the desired context of method calls.

Figure 3 shows the UDCS-s generated by test case $t1$ in our example: three method-calls are made directly from the test, method a is called twice and b is called once. The frequencies in the resulting unique deepest call stacks will provide the basis for the new approach: Table I shows the summarized call frequencies in the UDCS-s for each method in the example.

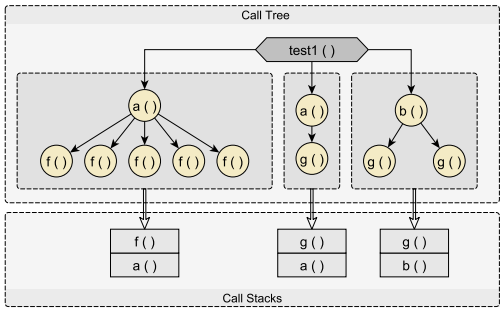

Fig. 3: Call tree and unique deepest call stacks (UDCS)

We will refer to this spectrum as the *call frequency spectrum* and denote the matrix as $\mathbf{C}$ (the results vector remains $\mathbf{R}$). As can be seen, the result is similar to the corresponding binary coverage matrix: if there is 0 in a hit-matrix position, it will be 0 in the count-matrix as well, but the latter can contain not only 1 but integer values as well.

Now, we can introduce our **new SBFL formulas** based on the above as follows. Our approach is basically very simple: we substitute the value $|m_{ef}|$ in the numerator of each technique presented in Section II-A with the counts introduced above. More precisely, we will use the notation $C(m_{ef})$ for this quantity and define it as follows (for clarity, we omit the code element m from the formulas for which the score is computed):

$$C(m_{ef}) = \sum_{t \in m_{ef}} c_{m,t}\,,$$

where $c_{m,t}$ is an element in $\mathbf{C}$. The modified formulas are shown in Table III.

TABLE III: Call frequency-based SBFL formulas

$$B^C: \frac{C(m_{ef})}{|m_{ef}| + |m_{ep}|} \qquad J^C: \frac{C(m_{ef})}{|m_{ef}| + |m_{nf}| + |m_{ep}|}$$

$$O^C: \frac{C(m_{ef})}{\sqrt{(|m_{ef}| + |m_{nf}|) \cdot (|m_{ef}| + |m_{ep}|)}}$$

$$R^C: \frac{C(m_{ef})}{|m_{ef}| + |m_{nf}| + |m_{ep}| + |m_{np}|} \qquad S^C: \frac{2 \cdot C(m_{ef})}{2 \cdot |m_{ef}| + |m_{nf}| + |m_{ep}|}$$

$C(m_{ef})$ values and the suspiciousness scores computed for our example program with the modified formulas are shown in Table IV. As can be seen, the buggy method (g) has higher suspiciousness scores using the call frequency-based SBFL technique, which can be attributed to the corresponding $C(m_{ef})$ values that weigh the candidate code elements as expected. In other words, the extra information execution stack traces carry helped to better rank the buggy element. Both failing and passing test cases contained methods that eventually called the buggy method. But call stack traces also captured the different call contexts, i.e., the call sequences leading to either failing or passing cases, and since the buggy method occurred multiple times in call stacks of failing tests, this resulted in more successful localization.

TABLE IV: Hit-based and call frequency-based SBFL scores

		a	**b**	**f**	**g**
hit-based scores	*Barinel*	0.500	0.500	0.500	0.500
	Jaccard	0.500	0.500	0.333	0.500
	Ochiai	0.707	0.707	0.500	0.707
	Russell-Rao	0.500	0.500	0.250	0.500
	Sørensen-Dice	0.667	0.667	0.500	0.667
frequency-based scores	$C(m_{ef})$	3	2	1	4
	$Barinel^C$	0.750	0.500	0.500	1.000
	$Jaccard^C$	0.750	0.500	0.333	1.000
	$Ochiai^C$	1.061	0.707	0.500	1.414
	$Russell\text{-}Rao^C$	0.750	0.500	0.250	1.000
	$Sørensen\text{-}Dice^C$	1.000	0.667	0.500	1.333

III. Empirical Evaluation

The main goal of empirically evaluating the proposed approach was to compare the new algorithms' fault localization effectiveness compared to their traditional hit-based counterparts. Also, we were interested in finding out which of the traditional SBFL formulas are best fitted to be extended with call frequency information. In this section, we overview the main parameters of the experiments: the benchmark used, the evaluation measures, and we also formulate the research questions more precisely.

A. Subject Programs

We implemented our approach for analyzing Java programs, and for the evaluation we selected Defects4J (v1.5.0),[1] a widely used collection of Java programs and curated bugs in FL research. This benchmark contains six open source Java projects with manually validated, non-trivial real bugs. To extract test results and stack-trace information from these projects on per-test level, we extended Defects4J's measurement framework with a Java Agent-based tool. This includes an online (on-the-fly) bytecode instrumentation tool, which we used to collect UDCS-s during test execution.

The original dataset contains 438 bugs, however, there were cases which we had to exclude from the study due to instrumentation errors or unreliable test results. A total of 411 defects were included in the final dataset. Table V shows each project and their main properties. The last column includes the statistics about the UDCS-s generated by the test cases.

[1] https://github.com/rjust/defects4j/tree/v1.5.0

TABLE V: Subject programs

Program	Number of bugs	Size (KLOC)	Number of tests	Number of methods	Number of unique deepest call stacks
Chart	25	96	2.2k	5.2k	122k
Closure	168	91	7.9k	8.4k	889k
Lang	61	22	2.3k	2.4k	6k
Math	104	84	4.4k	6.4k	228k
Mockito	27	11	1.3k	1.4k	11k
Time	26	28	4.0k	3.6k	150k

B. Evaluation of Effectiveness

Comparing the effectiveness of SBFL algorithms means comparing the suspiciousness ranking lists, and how successfully they approximate the actually faulty code element. An algorithm is successful in locating the fault if the faulty element is at or near the first position in the rank list. But, in order to be able to compare the results of the algorithms their outputs need to be compatible. Because the suspiciousness score values are not necessarily produced in the same interval by the different formulas, their relative position in the ranking list is used instead.

The position (rank) of the faulty method gives a good approximation of effectiveness because it indicates how many methods developers or testers need to examine before finding the bug. However, there are different ways to compare the ranking lists, and various research reports prefer one or the other. In this paper, we followed several different approaches used earlier and also employed new ones. Thus, we believe this thorough evaluation can highlight all the different aspects of how successful each of the alorithms are in localizing faults.

A particular issue to handle are *ties*, that is, cases when two or more elements share the same score. Essentially, there are three ways to determine the ranks of such elements [37]: (i) the average of the ranks of the faulty methods, (ii) the minimum of the ranks, or (iii) the their maximum. In each case, the same value is assigned to all elements with the tied values.

We used the average rank approach in our research. If there are multiple bugs for a program version, we will use the highest rank of buggy methods. Formula 1 shows the *absolute average rank* calculation [24], where i and f are methods, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E = \frac{|\{i|s_i > s_f\}| + |\{i|s_i \geq s_f\}| + 1}{2} \tag{1}$$

The *relative average rank* (Formula 2) specifies how much of the entire method set needs to be examined before finding the bug.

$$E' = \frac{E-1}{N} \cdot 100\ [\%]\text{, } N \text{ is the number of methods} \tag{2}$$

Both absolute and relative measures are often used in FL literature, and they have their benefits and drawbacks, hence we will use both of them. Table VI shows the average ranks of the example program for the hit-based and for the call frequency-based algorithms.

Using the average rank position, we can compare the algorithms by following our first two Research Questions:

TABLE VI: Ranks of the example program

alg.	E (E') a	b	f	g (the bug)
B	2.5 (37.5%)	2.5 (37.5%)	2.5 (37.5%)	2.5 (37.5%)
B^C	2.0 (20.0%)	3.5 (62.5%)	3.5 (62.5%)	1.0 (0%)
J	2.0 (20.0%)	2.0 (20.0%)	4.0 (75.0%)	2.0 (20.0%)
J^C	2.0 (20.0%)	3.0 (50.0%)	4.0 (75.0%)	1.0 (0%)
O	2.0 (20.0%)	2.0 (20.0%)	4.0 (75.0%)	2.0 (20.0%)
O^C	2.0 (20.0%)	3.0 (50.0%)	4.0 (75.0%)	1.0 (0%)
R	2.0 (20.0%)	2.0 (20.0%)	4.0 (75.0%)	2.0 (20.0%)
R^C	2.0 (20.0%)	3.0 (50.0%)	4.0 (75.0%)	1.0 (0%)
S	2.0 (20.0%)	2.0 (20.0%)	4.0 (75.0%)	2.0 (20.0%)
S^C	2.0 (20.0%)	3.0 (50.0%)	4.0 (75.0%)	1.0 (0%)

RQ1 In how many cases does the new, call frequency-based SBFL approach give lower/equal/higher ranks than its hit-based counterpart?
RQ2 How do the absolute and relative average rank positions compare in the two sets of algorithms?

Several studies report that developers investigate only first 5 or first 10 elements in the recommendation (rank-)list by fault localization algorithms before giving up using the ranking [10], [14]. Therefore, we distinguished between bugs where the minimum of faulty methods rank is equal to 1 (*Top-1*), it is less or equal to three (*Top-3*), less or equal to five (*Top-5*), less or equal to ten (*Top-10*), and when it is over ten (*Other*), commonly referred to as *Top-N* [11].

With this in mind, we compare the techniques using the following Research Questions:

RQ3 What proportion of faulty methods are among the most suspicious elements? That is, what is the number of buggy methods that are among the 1, 3, 5 and 10 most suspicious methods?
RQ4 What improvement can be achieved when the hit-based methods assign the bugs with very bad ranks, beyond 10 (the *Other* category), but call frequency-based SBFL classifies them in one of the higher categories? We call these cases *enabling improvements* [26].

Finally, besides comparing the pairs of SBFL formulas to each other, using the same measurement data, we can investigate which of the new frequency-based SBFL formulas perform best compared to the others, which can tell us what formulas are best fitted to be extended with call frequency information:

RQ5 Is there a call frequency-based SBFL formula that can be seen as a clear winner among all the others?

IV. Results

In this section, we present the results of our evaluation according to the measures described in Section III-B. This will enable a quantitative comparisons and objective judgments of the proposed algorithms in terms of fault localization effectiveness.

A simple comparison is to look at how many times the call freqency-based algorithms improve effectiveness, that is, they result in a higher rank (closer to 1) than their traditional hit-based counterparts, and how many times are the ranks lower or equal (**RQ1**). Such a comparison of each algorithm pair is shown in Table VII (counts and percentages with respect to all bugs).

The 'lose' row represents the number (percentage) of bugs for which the corresponding call frequency-based algorithm resulted in worse ranks. The 'draw' rows show the cases where the two ranks were equal, while the 'win' rows reveal how many times the new approach performed better. 'not-win' and 'not-lose' are values derived from the above three numbers by incorporating the 'draw' cases, correspondingly.

TABLE VII: Number (and percent) of times call frequency-based methods outperformed hit-based ones

	B	J	O	R	S
..lose	135 (32.8%)	137 (33.3%)	181 (44.0%)	77 (18.7%)	137 (33.3%)
..draw	89 (21.7%)	90 (21.9%)	74 (18.0%)	39 (9.5%)	89 (21.7%)
..win	187 (45.5%)	184 (44.8%)	156 (38.0%)	295 (71.8%)	185 (45.0%)
..not-win	224 (54.5%)	227 (55.2%)	255 (62.0%)	116 (28.2%)	226 (55.0%)
..not-lose	276 (67.2%)	274 (66.7%)	230 (56.0%)	334 (82.3%)	274 (66.7%)

It can be seen from this data that in all cases the improvement was notable except for one algorithm, when test-hit performed better than call frequency (for $Ochiai$: 44% vs. 38%). However, when taking into account the equal cases, $Ochiai$ is not worse either with 44.0% lost cases compared to 56.0% better or equal. This performance improvement is the highest for $Russell\text{-}Rao$: 18.7% vs. 71.8%.

RQ1: Overall, from the five new algorithms, all produced at least the same ranking or higher in more cases than the opposite. In four cases, the new algorithms are at least as good as their traditional counterparts for more than two-thirds of the bugs.

The next set of experiments dealt with the average ranks the new algorithms produced (**RQ2**). The upper part of Table VIII shows, for each project, new and old algorithm, the arithmetic means of the absolute average rank (E) values (the best results are highlighted in bold) as defined in the previous section. It can be seen that in only one case (for subject Chart) was one of the hit-based algorithms ($Ochiai$) better performing than any of the call frequency-based ones. By comparing the corresponding hit-based vs. call frequency-based formulas, it is obvious that in the majority of the cases, the new approach outperforms the traditional one. Except $Ochiai$, all algorithms consistently produce better average ranks. In the case of this algorithm, subject Closure makes the overall results worse, so by ignoring this project (which seems to be an outlier anyway with its very low overall ranks) it will be similar to the others. The overall statistics can be seen in the last two rows: for all subjects and excluding Closure.

Table VIII also shows that, overall, $Jaccard^C$ performed best: on 3 projects and the full bug data set this method placed the faulty methods at the highest ranks. This was followed by $Barinel^C$, with the best results in two projects, and $Ochiai^C$ and $Sørensen\text{-}Dice^C$, with one each. The $Russell\text{-}Rao$-based algorithm produced mixed results: it managed to improve relative to base algorithm, but compared to the rest it was relatively poorly performing.

To determine if the differences we found are statistically significant, we used Wilcoxon signed-rank [38], which examines whether two related paired ranks come from the same distribution or not. This provided an answer to the question of whether there is a significant difference between the results of the algorithms (at threshold 0.05): if $p\text{-}value$ is less than 0.05, then we can say that there is a significant difference between the results of two algorithms.

Next to the average values, we show the corresponding $p\text{-}value$ in Table VIII. The call frequency-based approach produced statistically significant improvement in altogether 16 cases, while there were two cases ($Ochiai$ on Closure and Chart), where the difference was significant but in the opposite direction. By more carefully observing the cases where $p\text{-}value$ was higher than 0.05, we can conclude that these results could probably be attributed to the fact that the corresponding subjects included too few bugs for the sample size to be large enough. An important result is, however, that taking into account the full dataset, all new approaches except $Ochiai$ achieved statistically significant improvement.

In Table VIII, we present some additional statistics about this data: a) *max*: the lowest rank, b) *median*: this value is the rank in the middle of the ordered results (compared to the average, this has the positive feature that it is much less sensitive to outlier values).

The highest ranks (values of *min*) are the same for all algorithms, 1, but the other metrics are different. For each algorithm, the maximum values among the results obtained with the call frequency-based method is consistently better than the highest rank obtained with the hit-based algorithm (row 'Max.' in Table VIII), i.e. the call frequency-based algorithm produces less extreme ranks. The results are different in terms of the median values. This relationship is aligned with the relation of the averages, i.e., in all cases except $Ochiai$, the call frequency-based method achieves a lower median (exluding Closure does not change the result here).

This set of data can further be analyzed with the help of Table IX. Here, we also presented the relative average ranks (E'), as well as the relative change between the hit-based and call frequency-based algorithm variants. The difference is shown as a simple difference between the two algorithm variants both for E and E', and also as a relative change compared to the traditional hit-based algorithm. If this value is negative, it means that we could achieve improvement with our new algorithms, and these results are highlighted in bold. Data is presented per subject program and also for the whole benchmark as well in the last column.

It can be seen that call frequency-based algorithms that achieve an improved E can produce relative improvement between 39% and 52%. It is interesting to note that on the full bug data set, the new $Ochiai$ variant performs worse

TABLE VIII: Average ranks, maximum and median of the ranks and $p\text{-}value$ of Wilcoxon signed-rank test

project	B	B^C	$p\text{-}value$	J	J^C	$p\text{-}value$	O	O^C	$p\text{-}value$	R	R^C	$p\text{-}value$	S	S^C	$p\text{-}value$
Chart	15.94	13.18	0.984	9.46	11.80	0.211	**8.82**	14.32	*0.027*	50.36	21.18	*0.000*	9.46	12.72	0.083
Closure	76.35	**46.75**	0.083	77.68	46.84	0.050	69.63	75.98	*0.011*	338.66	163.82	*0.000*	77.68	47.17	0.059
Lang	5.18	3.83	*0.011*	4.55	3.75	0.095	4.46	**3.56**	0.310	5.46	3.92	*0.020*	4.55	3.62	0.089
Math	10.20	6.04	*0.000*	10.08	**6.00**	*0.000*	10.32	6.18	*0.000*	21.45	9.23	*0.000*	10.08	6.03	*0.000*
Mockito	26.09	**17.15**	0.742	26.06	**17.15**	0.753	25.87	23.44	0.213	81.81	49.89	0.064	26.06	17.26	0.732
Time	19.77	8.31	0.077	19.67	**8.19**	0.075	18.38	8.88	0.357	55.46	16.98	*0.000*	19.67	**8.19**	0.068
all	38.50	23.66	*0.000*	38.51	**23.58**	*0.001*	35.13	36.12	0.096	156.61	75.52	*0.000*	38.51	23.77	*0.001*
all w/o Closure	12.32	7.70	*0.000*	11.43	**7.51**	*0.000*	11.28	8.57	0.112	30.76	14.47	*0.000*	11.43	7.60	*0.001*
Max.	927.5	523.0		927.5	520.0		927.5	653.5		1200.5	957.5		927.5	510.0	
Median	5.5	4.0		5.5	4.0		5.0	6.0		46.5	12.5		5.5	4.0	
Median w/o Closure	3.0	2.5		3.0	2.0		2.5	3.0		14.0	4.0		3.0	2.0	

TABLE IX: Absolute and relative average rank improvements

	Closure		Chart		Lang		Math		Mockito		Time		all	
B-E (E')	76.4	(0.80%)	15.9	(0.15%)	5.2	(0.24%)	10.2	(0.20%)	26.1	(2.01%)	19.8	(0.47%)	38.5	(0.70%)
B^C-E (E')	46.7	(0.49%)	13.2	(0.13%)	3.8	(0.17%)	6.0	(0.12%)	17.1	(1.32%)	8.3	(0.20%)	23.7	(0.43%)
Diff. E (E')	-29.7	(-0.31%)	-2.7	(-0.02%)	-1.4	(-0.07%)	-4.2	(-0.08%)	-9.0	(-0.69%)	-11.5	(-0.27%)	-14.8	(-0.27%)
Relative change	**-39%**		**-17%**		**-27%**		**-41%**		**-34%**		**-58%**		**-38%**	
J-E (E')	77.7	(0.81%)	9.5	(0.09%)	4.5	(0.20%)	10.1	(0.19%)	26.1	(2.01%)	19.7	(0.47%)	38.5	(0.70%)
J^C-E (E')	46.8	(0.49%)	11.8	(0.11%)	3.7	(0.17%)	6.0	(0.12%)	17.1	(1.32%)	8.2	(0.20%)	23.6	(0.43%)
Diff. E (E')	-30.9	(-0.32%)	2.3	(0.02%)	-0.8	(-0.03%)	-4.1	(-0.07%)	-9.0	(-0.69%)	-11.5	(-0.27%)	-14.9	(-0.27%)
Relative change	**-40%**		24%		**-18%**		**-41%**		**-34%**		**-58%**		**-39%**	
O-E (E')	69.6	(0.72%)	8.8	(0.09%)	4.5	(0.20%)	10.3	(0.20%)	25.9	(1.99%)	18.4	(0.44%)	35.1	(0.64%)
O^C-E (E')	76.0	(0.79%)	14.3	(0.14%)	3.6	(0.16%)	6.2	(0.12%)	23.4	(1.80%)	8.9	(0.21%)	36.1	(0.66%)
Diff. E (E')	6.4	(0.07%)	5.5	(0.05%)	-0.9	(-0.04%)	-4.1	(-0.08%)	-2.5	(-0.19%)	-9.5	(-0.23%)	1.0	(0.02%)
Relative change	9%		62%		**-20%**		**-40%**		**-10%**		**-52%**		3%	
R-E (E')	338.7	(3.53%)	50.4	(0.49%)	5.5	(0.25%)	21.4	(0.41%)	81.8	(6.29%)	55.5	(1.32%)	156.6	(2.85%)
R^C-E (E')	163.8	(1.71%)	21.2	(0.21%)	3.9	(0.18%)	9.2	(0.18%)	49.9	(3.84%)	17.0	(0.4%)	75.5	(1.37%)
Diff. E (E')	-174.9	(-1.82%)	-29.2	(-0.28%)	-1.6	(-0.07%)	-12.2	(-0.23%)	-31.9	(-2.45%)	-38.5	(-0.92%)	-81.1	(-1.48%)
Relative change	**-52%**		**-58%**		**-29%**		**-57%**		**-39%**		**-69%**		**-52%**	
S-E (E')	77.7	(0.81%)	9.5	(0.09%)	4.5	(0.20%)	10.1	(0.19%)	26.1	(2.01%)	19.7	(0.47%)	38.5	(0.70%)
S^C-E (E')	47.2	(0.49%)	12.7	(0.12%)	3.6	(0.16%)	6.0	(0.12%)	17.3	(1.33%)	8.2	(0.20%	23.8	(0.43%)
Diff. E (E')	-30.5	(-0.32%)	3.2	(0.03%)	-0.9	(-0.04%)	-4.1	(-0.07%)	-8.8	(-0.68%)	-11.5	(-0.27%)	-14.7	(-0.27%)
Relative change	**-39%**		34%		**-20%**		**-41%**		**-34%**		**-58%**		**-38%**	

than the hit-based formulas, but looking at the per project data individually, we can observe that the traditional methods achieve better ranks in only two of the six projects (Closure and Chart). However, as these two projects contain almost half of the bugs in the benchmark (47%) these can significantly affect the aggregate result.

As an aggregate statistics, we can conclude that out of a possible 35 project-algorithm pairs (5 algorithms times 6 projects and one aggregated set), there was a total of 5 cases where the call frequency-based algorithm did not achieve some improvement in the average ranks, which is only 14.3% of the possible scenarios.

By comparing these results to those that we obtained for RQ1 reveals an interesting insight. Despite some of the new algorithms managed to outperform the original ones many times and consistently to a large extent ($Russell\text{-}Rao$, for instance), the improved version could still not outperform some other algorithms. This was because the base version of such algorithms performed very badly in the first place.

RQ2: For all but one subject, some of the proposed call frequency-based algorithms achieved the best relative average rank. Of the new algorithms, $Jaccard^C$ performed the best overall, producing the lowest rank average on half of the subjects and on the full data set as well. In 81% of the algorithm-project pairs, the approach using call frequencies gave better results, of which $Barinel^C$ stands out because it achieved a lower rank than $Barinel$ in all cases.

Looking at the average ranks has its drawbacks. First, outliers could distort the overall information on the performance of our algorithm, as we have seen with the subject Closure. Second, it can tell us nothing about the distribution of the different rank values, and how do they change from the traditional to the new algorithm. We believe that not all (absolute) rank positions are equally important, as we discussed in Section III-B. Hence, in the next set of experiments we will concentrate on the *Top-N* findings (**RQ3**). In Table X, we can see the number of bugs belonging to the corresponding Top-N categories (with their percentages), accumulated for the whole benchmark, for each hit-based and call frequency-based algorithm. The best values for each category are highlighted in bold.

TABLE X: Top-N (count and percent)

	Top-1 (%)	Top-3(%)	Top-5(%)	Top-10(%)	Other(%)
B	65 (15.82%)	165 (40.15%)	201 (48.91%)	248 (60.34%)	163 (39.66%)
B^C	74 (18.00%)	179 (43.55%)	222 (54.01%)	**269 (65.45%)**	**142 (34.55%)**
J	66 (16.06%)	168 (40.88%)	203 (49.39%)	250 (60.83%)	161 (39.17%)
J^C	**77 (18.73%)**	**185 (45.01%)**	**223 (54.26%)**	**269 (65.45%)**	**142 (34.55%)**
O	68 (16.55%)	172 (41.85%)	209 (50.85%)	254 (61.80%)	157 (38.20%)
O^C	70 (17.03%)	162 (39.42%)	191 (46.47%)	245 (59.61%)	166 (40.39%)
R	10 (2.43%)	57 (13.87%)	72 (17.52%)	110 (26.76%)	301 (73.24%)
R^C	34 (8.27%)	119 (28.95%)	154 (37.47%)	196 (47.69%)	215 (52.31%)
S	66 (16.06%)	168 (40.88%)	203 (49.39%)	250 (60.83%)	161 (39.17%)
S^C	**77 (18.73%)**	183 (44.53%)	221 (53.77%)	266 (64.72%)	145 (35.28%)

We can instantly observe that $Jaccard^C$ outperformed all the rest in each of the categories, with two cases where

it produced the same results as with some of the other call frequency-based approach ($Sørensen\text{-}Dice^C$ for Top-1 and $Barinel^C$ for Top-10). Similarly to earlier findings, only $Ochiai^C$ did not manage to outperform its hit-based counterpart, except for Top-1.

RQ3: In each Top-N category, one of the call frequency-based approaches provided the best results overall: they put the most buggy elements in these high rank categories ($Jaccard^C$ and $Barinel^C$ performed specifically well in this regard). Furthermore, four of them gave better results in every category than their hit-based variants.

A particularly interesting case of the difference between the hit-based and the call frequency-based results considering the Top-N elements is when the traditional approach produces a very high rank (Other category with rank > 10) while the new algorithm moves this to a lower Top-N category. As this brings a "new hope" that a bug could possibly be found by the user with the new algorithm while it was very improbable with the old, we call this these cases *enabling improvement*s (**RQ4**). Table XI summarizes such cases.

The second column shows the number of bugs where the rank calculated by the hit-based algorithm was greater than 10. In the third column, we can see how many of these bugs the call frequency-based algorithms classified in the Top-10 or better groups. The percentages in the 2nd and 3rd columns were calculated with respect to the number of bugs in the whole dataset. The last column shows the average difference between the ranks in the enabling cases together with the relative improvement compared to the original rank that was calculated by the hit-based approach.

TABLE XI: Enabling improvements

	test-hit rank >10 (%)	Enabling improv. (%)	Relative improv. (%)
B vs. B^C	163 (39.7%)	51 (12.4%)	-34.9 (-87.5%)
J vs. J^C	161 (39.2%)	50 (12.2%)	-34.9 (-87.1%)
O vs. O^C	157 (38.2%)	50 (12.2%)	-35.2 (-87.5%)
R vs. R^C	301 (73.2%)	91 (22.1%)	-58.7 (-92.8%)
S vs. S^C	161 (39.2%)	49 (11.9%)	-35.1 (-87.4%)

We can observe that each call frequency-based algorithm achieves enabling improvements for at least 11% of the bugs. In these cases the test-hit algorithm ranked the faulty method in the *other* category, but the frequency-based algorithm managed to bring it forward into the *Top-10* (or better) groups. Considering the improvement of the actual ranks in these cases, we can see that the frequency-based algorithms were able to lower the ranks by about 34-59 positions, which corresponds to a 87-93% relative improvement. Note that, $Russell\text{-}Rao^C$ seems to be the best in this aspect, but since $Russell\text{-}Rao$ has the worst performance among other hit-based algorithms there are twice as much bugs outside the Top-10 as in the case of other algorithms. Overall, each algorithm was able to achieve enabling improvements in about third of the possible cases.

RQ4: Call frequency-based algorithms move more than 11% of bugs in the total dataset from the "hopeless" category of ranks > 10 to some of the higher rank positions that are more probable to be actually useful for the users. The improvement in rank value in such cases exceeded 87% for all newly proposed algorithms.

TABLE XII: Global comparison of all algorithms

	B	B^C	J	J^C	O	O^C	R	R^C	S	S^C
B	○	↑	←	↑	↑	↑	←	←	←	↑
B^C	←	○	←	↑	←	←	←	←	←	←
J	↑	↑	○	↑	↑	↑	←	←	↑	↑
J^C	←	←	←	○	←	←	←	←	←	←
O	←	↑	←	↑	○	←	←	←	←	↑
O^C	←	↑	←	↑	↑	○	←	←	←	↑
R	↑	↑	↑	↑	↑	↑	○	↑	↑	↑
R^C	↑	↑	↑	↑	↑	↑	←	○	↑	↑
S	↑	↑	↑	↑	↑	↑	←	←	○	↑
S^C	←	↑	←	↑	←	←	←	←	←	○
better	4	8	3	9	6	5	0	1	3	7
worse	5	1	6	0	3	4	9	8	6	2

As a summarization of previous data, we compared all algorithms – both the hit-based and call frequency-based ones – with each other and examined how many times each method results in a better (absolute) average rank than the others. This comparison is included in Table XII. The arrows in this table are used to denote which of the two corresponding algorithms performed better, i.e., which achieved a better average rank across the whole dataset. If the arrow points to the left then the algorithm shown in the row is more efficient, and if it points up the method in the column performed better. The last two rows of the table summarize how many times a given method proved to be better and worse than the others.

It can clearly be seen that $Jaccard^C$ is a clear winner among all methods as it provided lower average ranks than any other. It is followed by $Barinel^C$ and $Sørensen\text{-}Dice^C$. The last two call frequency-based formulas did not perform particularly well: $Ochiai$ was better than five other algorithms but was superseded by the rest four, while $Russell\text{-}Rao^C$ was beaten by almost all the other algorithms. The hit-based methods follow after this set, which underlines that the new proposed methods can generally provide notable improvements. Also, each call frequency-based method outperforms its hit-based counterpart, except $Ochiai^C$.

When we compare the relative order of the algorithms according to this set of data within their families (i.e., hit-based ones and call frequency-based ones to each other), we can reveal interesting insights. The traditional algorithms give the following order starting with the most successful one: $Ochiai$, $Barinel$, $Jaccard$/$Sørensen\text{-}Dice$, $Russell\text{-}Rao$. Previous literature reported similar relationships between these techniques, e.g. [32]. On the other hand, the new approaches generate this list: $Jaccard^C$, $Barinel^C$, $Sørensen\text{-}Dice^C$, $Ochiai^C$, $Russell\text{-}Rao^C$, which is different than what would be expected. An interesting finding is, for instance, that $Jaccard^C$made to the first position, and $Ochiai^C$turned out

to be much worse than expected, while this algorithm being constantly reported as one of the most successful ones among the hit-based SBFL families.

RQ5: Of the new algorithms examined, three call frequency-based approaches ($Jaccard^C$, $Barinel^C$ and $Sørensen\text{-}Dice^C$) stand out from the others in terms of their average ranks, $Jaccard^C$ being the clear winner by providing better results than all the others including the hit-based approaches. Hence, these formulas seem to be best fitted to benefit from call frequency information. Contrary to what we expected, $Ochiai^C$ did not perform well, while its hit-based version provides typically very good results within its family of methods.

V. Discussion

A. An Example from the Benchmark

To understand how does the proposed approach achieve such improvements, we manually analyzed several bugs from the Defects4J dataset. One interesting case we looked at was bug 103 from the Commons Math project [39], [40]. Figure 4 visualizes schematically the test-to-code relations of this bug.

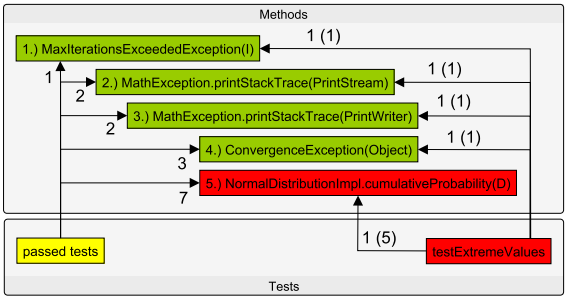

Fig. 4: Methods and tests related to the Math-103 bug

There are 16 test cases in this scenario, from which 15 are passing and one is failing. The *Methods* box contains those methods that are covered by the failing test. They are arranged vertically (top-to-bottom) based on their ranks and the faulty method is marked in red. The *Tests* box encloses the failing test (in red) and a placeholder for all passing tests, which cover the same methods as the failing test. For the sake of clarity, passing tests were aggregated into one node and only the 5 most suspicious methods are shown. The weights on the edges indicate $|m_{ef}|$ and $|m_{ep}|$ values, and $C(m_{ef})$ values are also shown in parentheses. For example, the method called *cumulativeProbability* appears five times in the UDCS-s that were collected during the execution of the failing test and it was covered by 7 passing tests. Similarly, both *printStackTrace* methods were found once in the UDCS-s generated by the failing test, and they were covered by 2 passing tests.

In the case of the hit-based approach, $Jaccard$ scores of these methods (from 1st to 5th by rank) are $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{4}$ and $\frac{1}{8}$. $Jaccard^C$ scores are the same as $Jaccard$ values, except for the buggy method where the $Jaccard^C$ score is $\frac{5}{8}$. (Other formulas produce different scores in this case but the ranks are the same.) As can be seen, the traditional approach emphasizes the exception handling parts of the code, which are covered by the failing test, but otherwise unrelated to the actual bug. However, the proposed formulas incorporate the frequency values into their numerator, which emphasizes the importance of the relationship between the failing tests and a particular method. In case of this bug, *testExtremeValues* calls *cumulativeProbability* directly repeatedly in a loop. Then, *cumulativeProbability* calls several other utility methods to calculate the probability. The loop is executed until the calculated probability reaches an extremely low or high value or an exception is thrown. As a result, *cumulativeProbability* appears 5 times in 5 different UDCS-s, while other related methods appear fewer times, hence the frequency-based approach can distinguish *cumulativeProbability* based on the additional contextual information that is provided by the UDCS-s. Note that a simple count-based algorithm would yield the same scores for those methods that are called in the aforementioned loop.

B. Threats to Validity

A possible threat to validity of our empirical study is that we had to exclude some parts of the Defects4J dataset, so this could make it difficult to compare the results to other studies employing the same benchmark. However, this affected only 27 bugs, which amounts to about 6% of the total bugs in the original set. The reason was that we could not compute UDCS-s for these cases due to technical limitations of the analysis. This selection was in no ways influenced by the results of the algorithms and the skipped bugs are distributed in the benchmark approximately evenly, so we believe that this factor can be considered minimal.

It is also a threat that we used only one benchmark that includes programs written in one language, Java. However, the bugs themselves are real and validated bugs and not manually seeded or generated as is the case with many other benchmarks used in FL research. Nevertheless, it would be useful to examine the performance of the approach on other data sets consisting of programs in other languages and other types and quantity of defects. For example, it is not known how would the approach perform in the presence of multiple bugs.

To compare the effectiveness of the call frequency-based approach, we selected a set of five SBFL formulas that share a common property: their numerators are compatible. Our results indicate that the proposed enhancement to the formulas can result in improvements in almost all cases in this class of algorithms. It is not known, however, if this concept could be successfully applied to other types of SBFL formulas, which is among our plans for future work.

Coarse granularity of analysis is a common criticism of similar experiments. In the present phase of the research, we relied on method-level granularity, and there are studies that find that using the method-level is good enough to help users identify the error [23], [25]. Currently, it is not known if the concept could be successfully adapted to other granularities such as statement. It remains future work to investigate this aspect.

VI. Related Work

A. Spectrum-based Fault Localization

Automated fault localization techniques have been around for more than three decades. There have been several surveys written [3], [4], [7], [41], and various empirical studies performed [6], [23], [27] to compare the effectiveness of various methods. While there are other fault localization techniques as well [18], [42]–[46], SBFL emerged into one of the main approaches of Software Fault Localization.

Despite the immense literature, SBFL is still to find its way to be used in practice [12], [13], [27]. Often the faulty element is placed far from the top of the rank-list [10], and this way the developer will refuse to use any help for localizing bugs. Abreu et al. [24] made a study on how accurate the SBFL approaches are. They found that the SBFL's accuracy is highly independent of the quality of test design.

B. Extending Hit-based Spectra

The basic constituent of SBFL is code coverage information. The most universally used spectra are based on individual statements or methods [47], [48], [49].

Harold et al. [20] proposed several other types of program spectra, however, the hit-based approach remained the most studied one. Abreu et al. [22] performed an empirical study on the count spectra for Fault Localization using Barinel [27]. They compared it to classical SBFL algorithms, however it did not improve the average ranks on real programs and bugs.

Classical SBFL algoritms are limited for locating faults in loops. Shu et al. [50] improved the Tarantula metric by extending it with counting the statement frequency. Likewise, our method tackles the same issue but in a different manner.

Lee et al. [51] proposed a new approach that improves SBFL by using frequency counts of test coverage. However, only counting the statement frequency can lead to distortion in the statistics. We use UDCS-s to mitigate this issue.

Laghari et al. [52] proposed a heuristic for SBFL using call sequences to rank the classes. Their method can pinpoint 56% of the faulty classes of NanoXML in all test runs. Their apporach filters out the repetitive method calls, which is similar to our approach.

C. Call Stacks and Function Call Information

Not many studies investigated call stacks and function call information in the context of fault localization. Beszédes et al. [26] used the Ochiai formula and expanded it with function call-chains context information. Function call chains and UDCS-s are both based on dynamic call information, but while it is expensive to compute call chains for large programs, UDCS-s are cheaper, hence our approach has better performance. Also, our method does not need expensive data structures such as call-chain matrices.

Jiang et al. [53] used the stack trace to locate null pointer exceptions more efficiently. Gong et al. [54] generate stack traces to help localize crash faults. They were able to find 64% crashing faults in Firefox 3.6.

VII. Conclusion

We proposed a new Spectrum-Based Fault Localization concept that is based on method call frequency information. We rely on the notion of Unique Deepest Call Stacks, data structures that capture call stack state information occurring on test case execution, and count the number of method occurrences within these structures. This is an advanced concept over simple call counts, which eliminates the problem of very large number of repetitions due to loops. With this information, we modified five traditional hit-based SBFL formulas, and empirically verified how much we can improve fault localization effectiveness on the Defects4J benchmark.

We achieved improvement in most cases: four of the five algorithms resulted statistically significant relative improvements of rank position in the range from 38%–52%. The remaining one (modified version of $Ochiai$) showed only a minimal deterioration on the full bug data set (3%), and this was caused by some outlier data. Without it, this algorithm also showed a relative improvement between 10%–52%. Of the total of 10 formulas examined (5 original and 5 modified), the top three best performing were call frequency-based, the best being the enhanced $Jaccard$, whose average ranks were better than all the other candidates'. The same algorithm showed specifically large improvements in the highest rank positions as well (top-1 to top-10), which is important in terms of practical usability of SBFL in general. So, we can conclude that the best candidates to benefit from call frequency information are formulas $Jaccard$, $Barinel$ and $Sørensen\text{-}Dice$.

We have several plans for future work. We would like to verify the performance of the approach on statement-level granularity, as well as with other SBFL formulas and on other benchmarks. In particular, we would like to systematically compare our approach to simple count-based methods and analyze more deeply the reasons behind the improvements. Also, our plan is to experiment with other parts of the formulas, not just the numerators. Finally, we investigated only one property of the call stacks, but we think that there might be other interesting features of these structures that are worth investigating to include them in the SBFL formulas.

To enable the reproduction of our experiments and additional analyses, we made the measurement framework and the final results publicly available on GitHub [55] and Figshare [56].

Acknowledgment

This research was supported by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary. Attila Szatmári was supported by project EFOP-3.6.3-VEKOP-16-2017-0002, co-funded by the European Social Fund. This work was partially supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled Internet of Living Things.

REFERENCES

[1] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 432–449, Nov. 1997.

[2] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *Managing Requirements Knowledge, International Workshop on(AFIPS)*, vol. 00, 12 1899, p. 539.

[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.

[4] P. Parmar and M. Patel, "Software fault localization: A survey," *International Journal of Computer Applications*, vol. 154, pp. 6–13, 11 2016.

[5] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *ArXiv*, vol. abs/1607.04347, 2016.

[6] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, 2017, p. 609–620. [Online]. Available: https://doi.org/10.1109/ICSE.2017.62

[7] P. Agarwal and A. Agrawal, "Fault-localization techniques for software systems," *ACM SIGSOFT Software Engineering Notes*, vol. 39, pp. 1–8, 09 2014.

[8] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 114–125.

[9] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[10] 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)*. IEEE, 2016, pp. 267–278.

[11] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[12] T. 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.

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

[14] 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*, 2016, pp. 165–176.

[15] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[16] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[17] W. Eric Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, 11 2011.

[18] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," ser. AADEBUG'05, New York, NY, USA, 09 2005, pp. 33–42. [Online]. Available: https://doi.org/10.1145/1085130.1085135

[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *SIGPLAN Not.*, vol. 40, no. 6, p. 15–26, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065014

[20] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 83–90. [Online]. Available: https://doi.org/10.1145/277631.277647

[21] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

[22] R. Abreu, A. Gonzalez-Sanchez, and A. J. van Gemund, "Exploiting count spectra for bayesian fault localization," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, ser. PROMISE '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 1–10.

[23] 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*, 2019.

[24] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[25] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[26] Á. Beszédes, F. Horváth, M. Di Penta, and T. Gyimóthy, "Leveraging contextual information from function call chains to improve fault localization," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 468–479.

[27] 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.

[28] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Van Hoorn, A. Filieri, and D. Lo, "An evaluation of pure spectrum-based fault localization techniques for large-scale software systems," *Software: Practice and Experience*, vol. 49, no. 8, pp. 1197–1224, 2019.

[29] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[30] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *International Symposium on Search Based Software Engineering*. Springer, 2012, pp. 244–258.

[31] N. Neelofar, L. Naish, and K. Ramamohanarao, "Spectral-based fault localization using hyperbolic function," *Software: Practice and Experience*, vol. 48, no. 3, pp. 641–664, 2018.

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

[33] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[34] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[35] A. Rountev, S. Kagan, and M. Gibas, "Static and dynamic analysis of call chains in java," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007514

[36] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, p. 85–96, 1997. [Online]. Available: http://doi.acm.org/10.1145/258916.258924

[37] X. Xu, V. Debroy, W. Eric Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 06, pp. 803–827, 2011.

[38] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[39] [Online]. Available: http://program-repair.org/defects4j-dissection/#!/bug/Math/103

[40] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. A. Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of SANER*, 2018.

[41] W. E. Wong and V. Debroy, "A survey of software fault localization," *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, vol. 9, 2009.

[42] H. Cao, S. Jiang, X. Ju, Z. Yanmei, and G. Yuan, "Applying association analysis to dynamic slicing based fault localization," *IEICE Transactions on Information and Systems*, vol. E97.D, pp. 2057–2066, 08 2014.

[43] T. Simomura, "Critical slice-based fault localization for any type of error," *IEICE Transactions on Information and Systems*, vol. 76, pp. 656–667, 1993.

[44] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5–7, p. 605–628, Aug. 2015. [Online]. Available: https://doi.org/10.1002/stvr.1509

[45] ——, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1293–1300. [Online]. Available: https://doi.org/10.1145/2554850.2554978

[46] Z. Li, Y. Wu, H. Wang, and Y. Liu, *Test Oracle Prediction for Mutation Based Fault Localization*, 09 2019, pp. 15–34.

[47] G. Shu, B. Sun, A. Podgurski, and F. Cao, "Mfl: Method-level fault localization with causal inference," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 124–133.

[48] R. Santelices, J. A. Jones, Yanbing Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 56–66.

[49] C. Oo and H. M. Oo, *Spectrum-Based Bug Localization of Real-World Java Bugs*. Cham: Springer International Publishing, 2020, pp. 75–89. [Online]. Available: https://doi.org/10.1007/978-3-030-24344-9_5

[50] T. Shu, T. Ye, Z. Ding, and J. Xia, "Fault localization based on statement frequency," *Information Sciences*, vol. 360, pp. 43 – 56, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025516302663

[51] H. J. Lee, L. Naish, and K. Ramamohanarao, "Effective software bug localization using spectral frequency weighting function," in *2010 IEEE 34th Annual Computer Software and Applications Conference*, 2010, pp. 218–227.

[52] G. Laghari, A. Murgia, and S. Demeyer, "Localising faults in test execution traces," ser. IWPSE 2015, 08 2015, pp. 1–8.

[53] S. Jiang, W. Li, H. Li, Z. Yanmei, H. Zhang, and Y. Liu, "Fault localization for null pointer exception based on stack trace and program slicing," in *Proceedings of the 2012 12th International Conference on Quality Software*, ser. QSIC '12. USA: IEEE Computer Society, 08 2012, pp. 9–12.

[54] L. Gong, H. Zhang, H. Seo, and S. Kim, "Locating crashing faults based on crash stack traces," 04 2014.

[55] "Supplemental material for paper 'Call Frequency-Based Fault Localization' - GitHub," 2021. [Online]. Available: https://github.com/sed-szeged/SpectrumBasedFaultLocalization/tree/CallFrequencyBasedFL

[56] "Supplemental material for paper 'Call Frequency-Based Fault Localization' - figshare," 2021. [Online]. Available: https://doi.org/10.6084/m9.figshare.13537298.v1