The Journal of Systems & Software 193 (2022) 111429

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Fault localization using function call frequencies☆

Béla Vancsics *, Ferenc Horváth, Attila Szatmári, Árpád Beszédes

Department of Software Engineering, University of Szeged, 6720 Szeged, Dugonics tér 13., Hungary

ARTICLE INFO

Article history:
Received 26 July 2021
Received in revised form 16 May 2022
Accepted 1 July 2022
Available online 5 July 2022

Keywords:
Spectrum-Based Fault Localization
Function call frequency
Call stacks
Testing
Debugging

ABSTRACT

In traditional Spectrum-Based Fault Localization (SBFL), *hit-based spectrum* is used to estimate a program element's suspiciousness to contain a fault, *i.e.*, only the binary information is used if the code element was executed by the test case or not. Count-based spectra can potentially improve the localization effectiveness due to the number of executions also being available. In this work, we use function-level granularity and define count-based spectra which use *function call frequencies*. We investigate the naïve approach, which simply counts the function call instances. We also define a novel method which is based on counting the different function call contexts, *i.e.*, the frequency of the investigated function occurring in *unique* call stack instances during test execution. The basic intuition is that if a function is called in many different contexts during a failing test case, it will be more probable to be accountable for the fault. We empirically evaluated the fault localization capability of different variations of the approach and compared them to 9 traditional SBFL techniques using the Defects4J benchmark. We show that: (i) naïve counts result in worse rank positions than the hit-based approach, but (ii) unique counts produce better rank positions with some of the algorithm variants.

1. Introduction

Fault Localization (FL) is often a very difficult and time consuming step during debugging. Therefore, its importance in software development and evolution is unquestionable. This paper deals with a class of automated FL methods, which are based on the notion of the "program spectrum" (Reps et al., 1997; Collofello and Cousins, 1987) (*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) (Wong et al., 2016; Parmar and Patel, 2016; de Souza et al., 2016; Pearson et al., 2017; Agarwal and Agrawal, 2014).

Several types of spectra have been defined over the past decades (Harrold et al., 2000; Wong et al., 2016) (for example: program invariants spectrum (Ernst et al., 2001; Sahoo et al., 2013), method calls sequence spectrum (Dallmeier et al., 2005; Liu et al., 2005), time spectrum (Yilmaz et al., 2008)), 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. This information is usually captured by a set of basic *spectrum metrics* (de Souza et al., 2016; Heiden et al., 2019), which count the number of passing and failing test cases that do or do not execute the code element in question, respectively. Dedicated *risk formulae* (Abreu et al., 2009b, 2007, 2009a; Naish et al., 2011; Yoo, 2012; Wong et al., 2013; Lucia et al., 2014) are then used to calculate the *suspiciousness* levels by combining the spectrum metrics, 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 (Keller et al., 2017; Xie et al., 2013; Xia et al., 2016; Parnin and Orso, 2011), but other issues have also been identified (Le et al., 2013; Pearson et al., 2017; Steimann et al., 2013; Kochhar et al., 2016), 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 (Zhang et al., 2017; Li et al., 2019a; Eric Wong and Qi, 2011; Zhang et al., 2005; Liblit et al., 2005).

A straightforward extension of the hit-based spectrum are the *count-based* ones which record the number of executions of a particular element during runtime, and this way the binary spectrum is replaced by a spectrum with integer values. However,

☆ Editor: Earl Barr.

* Corresponding author.
E-mail address: vancsics@inf.u-szeged.hu (B. Vancsics).

https://doi.org/10.1016/j.jss.2022.111429

count-based spectra are comparably rarely investigated in literature. Some early results have been published by Harrold et al. (1998, 2000), but more recently Abreu et al. (2010) 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 first verify the mentioned weakness of the simple count-based approach (we call it the *naïve counts* approach), and empirically evaluate its fault localization capability compared to the hit-based approach, which we use as a baseline throughout this paper. Then, we propose a method to improve hit-based spectra using a more advanced count-based approach, which we call the *unique counts* approach. Here, 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 function-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. 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 the approach on several traditional hit-based SBFL formulae by adapting the spectrum metric calculations and the formulae to handle integer spectra. This adaptation was not trivial, and we experimented with several different approaches to this end. For the empirical assessment we relied on the popular bug benchmark, often used in SBFL research, Defects4J (Just et al., 2014). Results indicate that the naïve counts method is indeed not suitable to replace the hit-based approach (it consistently worsened the results in a statistically significant amount). However, some of the unique counts algorithms were able to statistically significantly outperform hit-based formulae.

The main contributions of the paper can be summarized as follows:

1. Definition of different count-based spectra and the adaptation of the associated spectrum metrics.
2. Adaptation of existing SBFL formulae to handle count-based spectra using four different strategies.
3. Empirical evaluation of the fault localization effectiveness of the naïve counts approach compared to the hit-based method.
4. Empirical evaluation of the fault localization effectiveness of the unique counts approach compared to the hit-based method.

This paper is an extension of an earlier version of our work (Vancsics et al., 2021a), and it provides a more thorough investigation of count-based spectra and gives a more general and complete overview of the topic. The previous study did not include the results of the naïve counts approach, and the paper presented only one kind of adaptation of the formulae. The latter was supplemented with 3 new adaptations and their results (using the naïve and unique counts concepts). The new adaptations allowed us to examine *four other formulas* to get a more comprehensive picture of the effectiveness of our approach. In addition, we involved new projects in the evaluation by making the evaluation based on the latest version of Defects4J, which includes several small, medium, and large projects that can provide a more comprehensive picture of the efficiency of the algorithms.

2. Overview

In this paper, we introduce new kinds of Spectrum-Based Fault Localization algorithms building on existing concepts known as the hit-based approach. The proposed new methods are then empirically evaluated by comparing their fault localization capability to the traditional hit-based approaches used as a baseline. We first introduce the basic notations, data structures and algorithms using a running example (Sections 3–6), then the empirical evaluation is provided. Fig. 1 shows the workflow of this paper in more detail. The three main phases are the following:

(i) **Input data generation:** preparation of input data, *i.e.*, the different spectra required for algorithms using the production and the test code. We distinguished three types of spectra. The "classical" hit-based data are presented in more detail in Section 3, while the naïve count spectra and the unique count spectra containing the added information are described in Sections 4 and 5, respectively.

(ii) **Fault localization algorithm:** at this stage of the process, we demonstrate the FL algorithms that rely on the different spectra, and quantify their performance. This procedure can take place on two threads, depending on the type of input data: for hit spectra, we determine the efficiency of the method using traditional SBFL risk formulae (Section 3), but for naïve and unique count data, we need certain adaptations. These mappings redefine the spectrum metrics needed to execute the algorithm (Section 6.1) and redefine FL formulae using them (Section 6.2). In our research, we present nine traditional SBFL risk formulae and four different adaptations of them that can be interpreted on both naïve and unique count data, thus providing a set of algorithm combinations to experiment with.

(iii) **Evaluation:** Section 7 contains the description of our empirical evaluation with results in Section 8. We compared our FL algorithms to the hit-based approach and quantified the differences between the efficiencies. We evaluate the effectiveness of the naïve and unique count-based methods, using a number of evaluation metrics often used in FL research, complemented with significance testing.

In Section 9 we elaborate on the details of our approach by discussing two special examples from the benchmark, and we address threats to validity in this section as well. We discuss the relevant related literature and previous works in Section 10. Finally, conclusions are provided in Section 11.

Note, that this paper deals with function-level granularity for the analysis, which means that the basic element for localizing a fault is a function or method of a class. Hence, all the discussion that follows will assume a basic code element to be a function or a method. This 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 (Le et al., 2016; Zou et al., 2018).

3. Hit-based spectra and risk formulae

Hit-based SBFL uses a binary coverage matrix (noted by Cov^H in this paper) and a test results vector (noted by R) as the basic data structures to calculate the suspiciousness scores for program elements (Abreu et al., 2007, 2009a). The formal descriptions of these components are shown in Fig. 2. In the coverage matrix, the columns represent the tests and the rows are the program elements, methods or functions in our case. The value of an

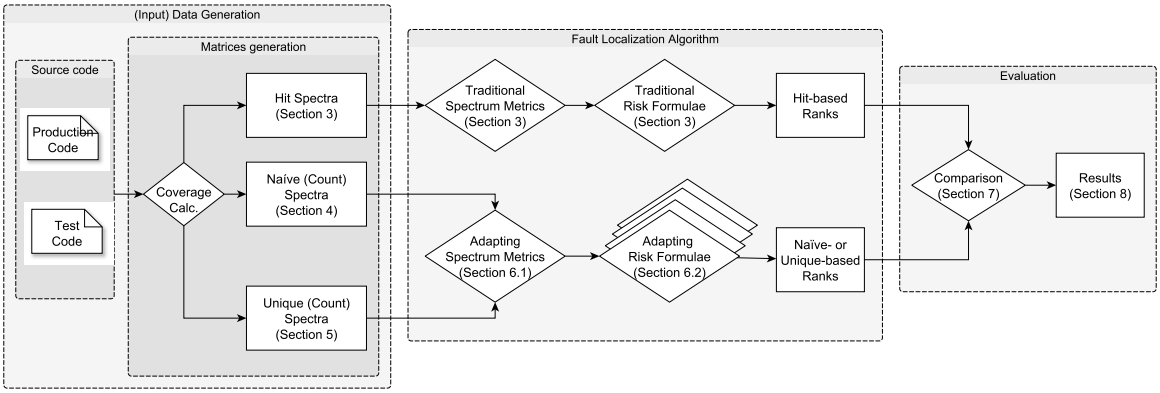


Fig. 1. Overview of the proposed approach, the evaluation and paper structure.

$$Cov^H = \begin{bmatrix} Cov^H_{t_1,m_1} & \dots & Cov^H_{t_i,m_1} \\ \vdots & \ddots & \vdots \\ Cov^H_{t_1,m_j} & \dots & Cov^H_{t_i,m_j} \end{bmatrix} \quad , \quad Cov^H_{t,m} = \begin{cases} 1 & \text{if } m \text{ is covered by } t \\ 0 & \text{otherwise} \end{cases}$$

$$R = \begin{bmatrix} R_{t_1} \dots R_{t_i} \end{bmatrix} \quad , \quad R_t = \begin{cases} 1 & \text{if test } t \text{ fails} \\ 0 & \text{if test } t \text{ passes} \end{cases}$$

$$(m \in M \text{ (methods)} \ , \ t \in T \text{ (tests)} \ , \ i, j \in \mathbb{N})$$

Fig. 2. Formal description of the binary coverage matrix (Cov^H) and results vector (R).

element in the Cov^H matrix is 1 or 0, depending on whether the method is covered by the test or not. An element in the results vector is 0 if the given test passed, otherwise it is 1.

Fig. 3 contains a short code snippet which is an adapted version of the example from Beszédes et al. (2020). In this example, {a, b, f, g} is the set of program elements (methods), and {t1, t2, t3, t4} are the test cases. As can be seen, the four program elements are dependent on each other: method a calls methods f and g directly, while method b calls a and g also directly. We set the bug to be in method g, which is called by a and b. Tests t1 and t2 fail due to the bug, the other two tests are passing. The resulting coverage matrix and results vector are shown in Table 1 (leftmost matrix). This method is constructed to emphasize the importance of caller–callee relationships, different call contexts, and the frequency of method calls. More realistic examples which are actual bugs from the benchmark are shown in Section 9.

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

(a) $m_{ep} = \{t | t \in T \wedge Cov^H_{t,m} = 1 \wedge R_t = 0\}$: the set of passed tests covering m
(b) $m_{ef} = \{t | t \in T \wedge Cov^H_{t,m} = 1 \wedge R_t = 1\}$: the set of failed tests covering m
(c) $m_{nf} = \{t | t \in T \wedge Cov^H_{t,m} = 0 \wedge R_t = 1\}$: the set of failed tests not covering m
(d) $m_{np} = \{t | t \in T \wedge Cov^H_{t,m} = 0 \wedge R_t = 0\}$: the set of passed tests not covering m

The hit-based spectrum for our running example in Fig. 3 is presented in Table 1 and the spectrum metrics can be seen in Table 2.

For this paper, we selected nine well-known risk formulae to experiment with, whose details are shown in Table 3. We selected these formulae because their effectiveness is surprisingly varying as presented in the literature, and we wanted to experiment with the possibly most diverse set of algorithms. It can be observed that all the selected formulae rely on $|m_{ef}|$ 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, but the other metrics can also be found in many formulae. Although, other formulae were not included in the current experiment, in Section 10 we present an overview of research papers that in fact explain those in detail.

The corresponding suspiciousness scores for each method in our example and for each risk formula are shown in Table 4. As can be seen, the buggy method (g) is hardly distinguishable from the other methods based on the suspiciousness scores. Two algorithms (*Barinel* and *Tarantula*) cannot distinguish between the methods, each with a score of 0.5. For the other fault localization algorithms, three methods have the same suspiciousness value (a, b and g), *i.e.*, according to the procedures, these methods have the same chance of containing a defect.

4. Naïve count-based spectra

Before we introduce our approach of a more elaborate count-based spectra, we first recall the naïve count-based method as it was proposed in previous literature, but gained little popularity due to its inefficiency (Harrold et al., 1998, 2000).

```java
public class Example {
  private int _x = 0;
  private int _s = 0;
  public int x() {return _x;}

  public void a(int i) {
    _s = 0;
    if (i==0) return;
    if (i<0) {
      for (int y=0;y<=9;y++)
        f(i);
    } else {
      g(i);
    }
  }

  public void b(int i) {
    _s = 1;
    if (i==0) return;
    if (i<0) {
      for (int y=0;y<Math.abs(i);y++)
        a(0);
    } else {
      for (int y=0;y<=1;y++)
        g(i);
    }
  }

  private void f(int i) {
    _x -= i;
  }

  private void g(int i) {
    _x += (i+_s); //should be _x += i;
  }
}
```

(a) Running example – program

```java
public class ExampleTest {
  @Test public void t1() {
    Example tester = new Example();
    tester.a(-1);
    tester.a(1);
    tester.b(1);
    tester.b(-8);
    assertEquals(13, tester.x()); // failed
  }

  @Test public void t2() {
    Example tester = new Example();
    tester.a(1);
    tester.b(1);
    assertEquals(3, tester.x()); // failed
  }

  @Test public void t3() {
    Example tester = new Example();
    tester.a(1);
    tester.b(0);
    assertEquals(1, tester.x());
  }

  @Test public void t4() {
    Example tester = new Example();
    tester.a(-1);
    tester.a(1);
    tester.b(0);
    assertEquals(11, tester.x());
  }
}
```

(b) Running example – test cases

Fig. 3. Running example. Corresponding matrices and spectrum metrics are presented in Tables 1 and 2.

Table 1
Hit-based spectrum (Cov^H) for the running example in Fig. 3. (Other spectra are shown for reference.).

		Hit spectrum (Cov^H)				Naïve spectrum (Cov^N)				Unique spectrum (Cov^U)			
		t1	t2	t3	t4	t1	t2	t3	t4	t1	t2	t3	t4
Methods	a	1	1	1	1	10	1	1	2	3	1	1	2
	b	1	1	1	1	2	1	1	1	2	1	1	1
	f	1	0	0	1	10	0	0	10	1	0	0	1
	g	1	1	1	1	3	3	1	1	2	2	1	1
Results (R)		1	1	0	0	1	1	0	0	1	1	0	0

Table 2
Hit-based spectrum metrics for the running example in Fig. 3.

		$\lvert m_{ef}\rvert$	$\lvert m_{ep}\rvert$	$\lvert m_{nf}\rvert$	$\lvert m_{np}\rvert$
Methods	a	2	2	0	0
	b	2	2	0	0
	f	1	1	1	1
	g	2	2	0	0

This technique incorporates the call frequency by simply counting the number of invocations of the methods while executing a test case. However, there is a fundamental issue with this technique, which has been raised by other authors as well, but not actually investigated empirically in detail previously (Abreu et al., 2010). Namely, 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.

In Fig. 4 we can see a dynamically built call-tree for the example in Fig. 3. A node is a caller when it is a parent node and it is a callee otherwise. For example, `a` is a callee because `t1` and `b` call it (lines 4–5 in Fig. 3(b) and line 21 in Fig. 3(a)), but `a` is also caller for the reason that it "uses" methods `g` in the else branch

Table 3
Details of the hit-based SBFL formulae used in our experiment.

Barinel (Abreu et al., 2009b): $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{ep}\rvert}$	*DStar* (Wong et al., 2013): $\frac{\lvert m_{ef}\rvert^2}{\lvert m_{ep}\rvert + \lvert m_{nf}\rvert}$
GP13 (Yoo, 2012): $\lvert m_{ef}\rvert \cdot \left(1 + \frac{1}{2 \cdot \lvert m_{ep}\rvert + \lvert m_{ef}\rvert}\right)$	*Jaccard* (Abreu et al., 2007): $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$
Naish2 (Naish et al., 2011): $\lvert m_{ef}\rvert - \frac{\lvert m_{ep}\rvert}{\lvert m_{ep}\rvert + \lvert m_{np}\rvert + 1}$	*Ochiai* (Abreu et al., 2007) : $\frac{\lvert m_{ef}\rvert}{\sqrt{(\lvert m_{ef}\rvert + \lvert m_{nf}\rvert) \cdot (\lvert m_{ef}\rvert + \lvert m_{ep}\rvert)}}$
Russell-Rao (Abreu et al., 2009a): $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert + \lvert m_{np}\rvert}$	*Sørensen-Dice* (Lucia et al., 2014) : $\frac{2 \cdot \lvert m_{ef}\rvert}{2 \cdot \lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$
Tarantula (Jones and Harrold, 2005) : $\frac{\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert}}{\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert} + \frac{\lvert m_{ep}\rvert}{\lvert m_{ep}\rvert + \lvert m_{np}\rvert}}$	

Table 4
Hit-based example scores for the running example in Fig. 3.

Methods	*Barinel*	*DStar*	*GP13*	*Jaccard*	*Naish2*	*Ochiai*	*Russell-Rao*	*Sørensen-Dice*	*Tarantula*
a	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50
b	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50
f	0.50	0.50	1.25	0.33	0.67	0.50	0.25	0.50	0.50
g	0.50	2.00	2.25	0.50	1.33	0.71	0.50	0.67	0.50

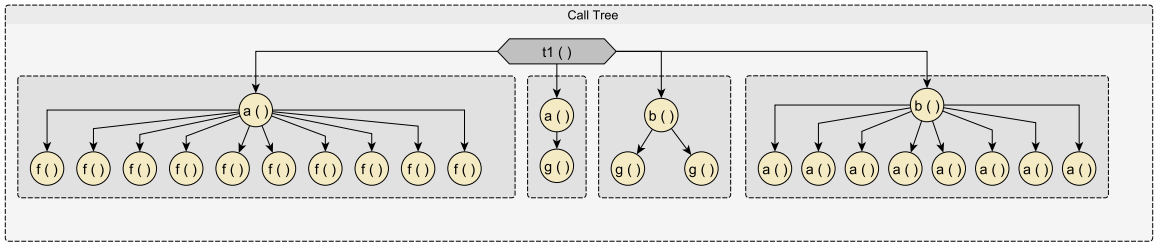


Fig. 4. Dynamic call-tree collected during the execution of the `t1` test from the running example in Fig. 3.

(line 13 in Fig. 3(a)) and `f` in the iteration (line 11 in Fig. 3(a)). The issue mentioned above can be observed in Figs. 3 and 4: method `f` is called ten times by `t1` (failed test) in a for loop, as opposed to `g` (faulty method), which is called six times in total by the failed tests (`t1` and `t2`).

Measuring the spectra, the higher execution count of non-faulty methods, such as `f` and `a`, compared to the actual faulty method `g` would result in failing to locate `g` successfully. In both techniques, repeated execution in the *for* loop results in high call-value: `a`(−1), caller method, executes `f` ten times (Fig. 3(a) lines 9–11) and the parameter of `b` can also result in cyclic repetition. In this example, method `a` is called eight times by `b` because the parameter (−8) affects the stop condition of the loop (Fig. 3(a) lines 19–21), so only counting the execution of a method results in missing the faulty method. The two cases described above illustrate well one of the most important features of the naïve technique (which is also one of the biggest drawbacks): the high degree of sensitivity to repeating code elements.

The naïve coverage matrix for the example is shown in Table 5 (middle part). We will refer to this spectrum as the naïve count-based spectrum and denote the matrix as Cov^N (the results vector remains R with the same properties). As can be seen, the naïve matrix is similar to the corresponding binary coverage matrix (Fig. 2): where there is a 0 in the hit-matrix, there will be a 0 in the naive-matrix as well, but Cov^N can contain not only 1s but other positive integer values as well, *i.e.*, the number of times a particular test executes an actual code element (a method or function in our case).

5. Unique count-based spectra

Numerous researchers investigated the efficiency of traditional spectrum-based fault localization. However, using the hit coverage does not always lead to finding the faulty element. Thus, tackling this problem researchers (Shu et al., 2016; Beszédes et al., 2020; Zhu et al., 2017; Abreu et al., 2010) used extra information to boost the algorithm's performance. With the following two paragraphs, we want to confirm that the use of (call) context is good intuition and that it may be suitable to improve the efficiency of fault localization algorithms.

Such information is investigating the relationship of the code and the tests then giving weights to them. Li and Liu (2014) approached the problem by weighting test cases based on how many blocks of code they had executed. They concluded that the less blocks have been executed by a given failing test case, the more SBFL's efficiency enhances. Ren and Ryder (2007) made a heuristic ranking on failing test cases to strengthen SBFL. They based their heuristic on the premise from slicing techniques that methods with large numbers of callers and callees in the call graph are more likely to be failure-prone than other changed methods.

Several studies have been presented that use (call context) information extracted from static and/or dynamic call graphs (Zhang et al., 2017; He et al., 2020; Zhao et al., 2021) or program-slice (Mao et al., 2014; Zhang et al., 2005; Wong et al., 2005; Wong and Qi, 2006) in fault localization algorithms. By weighting the nodes of call graph using some algorithm (*e.g.* Pagerank) and "expanding" the resulting information to the classic SBFL methods, thereby increasing their efficiency. Pagerank uses the *caller* and *callee* relation of code elements and new FL concepts transform this "structure" into the formulae. These methods are similar to the method we present in that these do not create a new formula but "redefine" existing ones using the newly added information and adapting spectrum metrics.

Table 5
Naïve count-based spectrum (Cov^N) for the running example in Fig. 3. (Other spectra are shown for reference.).

		Hit spectrum (Cov^H)				Naïve spectrum (Cov^N)				Unique spectrum (Cov^U)			
		t1	t2	t3	t4	t1	t2	t3	t4	t1	t2	t3	t4
Methods	a	1	1	1	1	10	1	1	2	3	1	1	2
	b	1	1	1	1	2	1	1	1	2	1	1	1
	f	1	0	0	1	10	0	0	10	1	0	0	1
	g	1	1	1	1	3	3	1	1	2	2	1	1
Results (R)		1	1	0	0	1	1	0	0	1	1	0	0

Our idea to add contextual information to the simple hit-based FL formulae and to enhance the naïve count-based approach is to incorporate how often a specific method has been called (directly or indirectly) and in which context from the test cases.

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 and the number of invocations during the course of executing the test cases. Based on the findings of graph- and slice-based papers mentioned above (*e.g.* Zhang et al., 2017; He et al., 2020; Mao et al., 2014; Zhang et al., 2005), we conclude 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.

Contextual information can vary, from using slicing techniques to using other ranking methods, *e.g.*, Pagerank. One of which is using "call-chains" to improve the efficiency of Fault Localization algorithms. Call-chains are a set of function calls the test cases invoke, which is well-known to programmers who are debugging the code. This gives the opportunity to use more precise coverage data and can show whether a function fails when called from one place but does not when invoked from another. Call-chains are harvested from stack traces and there is strong evidence that using stack-traces can help programmers fixing bugs (Schröter et al., 2010; Zou et al., 2018).

Instead of the naïve concept, in this approach we rely on method call stack traces. 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.

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).

To extract test results and stack-trace information from projects on per-test level, we developed a tool[1] for measurements. This includes an online (on-the-fly) bytecode instrumentation tool, which we used to collect UDCS-s during test execution. During this process, probes are inserted into all methods that are relevant for the analysis. These probes are guarding every methods' entry and exit points i.e., they trigger every time the execution reaches a method call and when the execution leaves a method e.g., by reaching a *return* or a *throw* statement. During the execution of the test cases, the information provided by the probes is incorporated into a tree-like data structure which ensures that unique stack-traces are collected, and which also minimizes memory consumption.

Unique count-based spectrum: 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 risk formulae instead of the basic spectrum metrics 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.

Fig. 5 shows the UDCS-s generated by test case `t1` in our example: four method-calls are made directly from the test, method `a` is called three times and `b` is called twice. The frequencies in the resulting unique deepest call stacks will provide the basis for our new approach. Table 6 shows the summarized call frequencies in the UDCS-s for each method in the example.

We will refer to this spectrum as the unique count-based spectrum and denote the matrix as Cov^U (the results vector remains R with the same properties as earlier). Like in the case of the naive-matrix, if there is 0 in a hit-matrix position, there will be a 0 in the unique matrix as well, but Cov^U can contain not only 1s but other positive integer values as well.

Using the unique count-based spectrum we can mitigate the issue of repeating method calls (caused by loops), and we also incorporate some degree of contextual information into the coverage data. However, this implies the adaptation of the corresponding spectrum metrics and risk formulae as well, which we present in Section 6.

6. Count-based risk formulae

The formulae presented earlier in Table 3 are based on the traditional version of the spectrum metrics, which are interpreted on binary coverage matrices. In this section, we show how these metrics can be extended to non-binary matrices and how these can be used in risk formulae to increase the efficiency of fault localization algorithms.

6.1. Adapting the basic spectrum metrics

We introduce the four spectrum metrics for the non-binary matrix as follows. The definitions can be interpreted for both naïve and unique matrices, as the method of calculation is independent from the type of the matrix.

Calculating the two values associated with the tests that executed the code element ($|m_{ef}|$ and $|m_{ep}|$) is simple: we summarize the (matrix) elements belonging to the m method for which the test was failed or passed. We will use the notations $C(m_{ef})$ and $C(m_{ep})$ for these quantities respectively and define them more precisely as follows:

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

where $c_{t,m}$ is an element of the Cov^N or Cov^U matrix.

[1] https://github.com/sed-szeged/java-instrumenter.

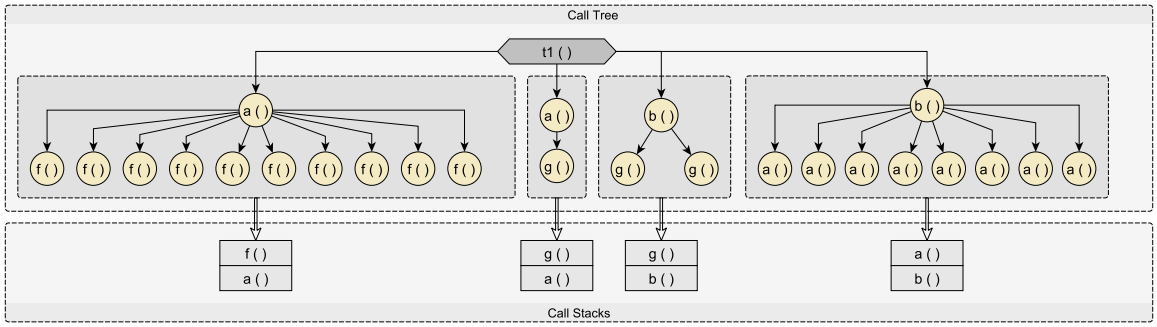


Fig. 5. Dynamic call-tree and the corresponding unique deepest call stacks (UDCS) collected during the execution of the `t1` test from the running example in Fig. 3.

Table 6
Unique count-based spectrum (Cov^U) for the running example in Fig. 3. (Other spectra are shown for reference.).

		Hit spectrum (Cov^H)				Naïve spectrum (Cov^N)				Unique spectrum (Cov^U)			
		t1	t2	t3	t4	t1	t2	t3	t4	t1	t2	t3	t4
Methods	a	1	1	1	1	10	1	1	2	3	1	1	2
	b	1	1	1	1	2	1	1	1	2	1	1	1
	f	1	0	0	1	10	0	0	10	1	0	0	1
	g	1	1	1	1	3	3	1	1	2	2	1	1
Results (R)		1	1	0	0	1	1	0	0	1	1	0	0

Our intuition was to present the frequency information in the spectrum metrics, that is, our concept "rewards" the methods that appear multiple times on UDCS during the execution of a passed or failed test. This weights the role of code elements in the execution process.

In the other two cases ($|m_{nf}|$ and $|m_{np}|$) the adaptation of the metrics is a bit more difficult because not executing an element cannot be simply associated with something like "how many times not executed". The approach we use calculates the average coverage of uncovered tests by the other methods. More precisely,

$$C(m_{nf}) = \sum_{t \in m_{nf}} \frac{\sum_{m' \in M'} c_{t,m'}}{|M| - 1} \quad \text{and} \quad C(m_{np}) = \sum_{t \in m_{np}} \frac{\sum_{m' \in M'} c_{t,m'}}{|M| - 1}$$

where M is the set of methods, M' is $M \setminus m$ and $c_{t,m}$ is an element of the Cov^N or Cov^U matrix. These two values indicate the average amount of coverage a method "loses" for passed and failed tests. This is a fairly intuitive way to calculate the values of non covering metrics, but obviously there are plenty of other options to define these, however due to space limitations we are not investigating the effect of differently defined $C(m_{nf})$ and $C(m_{np})$ values in this paper.

Let us consider the naïve count-based spectrum (Table 5) and method `f` of our running example to illustrate the adapted spectrum metrics:

- $\mathtt{f}_{ef} = \{\mathtt{t1}\}$ and $C(\mathtt{f}_{ef}) = c_{\mathtt{t1,f}} = 10$: `t1` test calls `f` ten times
- $\mathtt{f}_{ep} = \{\mathtt{t4}\}$ and $C(\mathtt{f}_{ep}) = c_{\mathtt{t4,f}} = 10$: `f` is used ten times by `t4`
- $\mathtt{f}_{nf} = \{\mathtt{t2}\}$ and $C(\mathtt{f}_{nf}) = \frac{c_{\mathtt{t2,a}} + c_{\mathtt{t2,b}} + c_{\mathtt{t2,g}}}{|\{\mathtt{a,b,f,g}\}| - 1} = \frac{1+1+3}{3} = 1.67$: "average coverage" of the failed `t2` not covering `f`
- $\mathtt{f}_{np} = \{\mathtt{t3}\}$ and $C(\mathtt{f}_{np}) = \frac{c_{\mathtt{t3,a}} + c_{\mathtt{t3,b}} + c_{\mathtt{t3,g}}}{|\{\mathtt{a,b,f,g}\}| - 1} = \frac{1+1+1}{3} = 1$: "average coverage" of the passed `t3` not covering `f`

Table 7 shows the four spectrum metrics for the binary (hit-based) and the two non-binary (naïve and unique count-based) spectra side by side to enable their comparison.

Subsequently, the values for the naïve matrices are marked Cov^N and the values for the unique matrices are noted by Cov^U.

6.2. Adapting the risk formulae

The statistics defined above can be adapted in the formulae in several ways, *i.e.*, these can be combined in different ways in the original formula. We examined in our recent paper (Vancsics et al., 2021a) how the efficiency of FL algorithms changes when unique count-based $C(m_{ef})$ is used instead of hit-based $|m_{ef}|$ in the numerator of formulae, however, by redefining/extending the other three metrics, the possible scenarios are further expanded.

In this paper, we examine whether the efficiency of algorithms can be increased using the count-based spectra and any of the following strategies:

- $\Delta^*_{ef^{num}}$ We replace only the $|m_{ef}|$ in the numerator provided that it contains only $|m_{ef}|$ (this approach cannot interpreted for the *GP13*, *Naish2* and *Tarantula* formulae).
- Δ^*_{ef} Each occurrence of $|m_{ef}|$ is overwritten with the new ($C(m_{ef})$) value.
- Δ^*_{e} The values of all occurrences of $|m_{ef}|$ and $|m_{ep}|$ are changed in the formula with the corresponding adapted values.
- Δ^*_{all} The count-based matrix is used for the calculation of all four metric values in all their occurrences.

The results of the previous paper (Vancsics et al., 2021a) were encouraging, but they can only be applied to formulas that have only $|m_{ef}|$ in their numerator. A logical and intuitive "continuation" of this line of reasoning is the case where we also replace the $|m_{ef}|$ value in the denominator with its adapted value (because the value of $|m_{ef}|$ greatly affects the efficiency of the algorithms). In addition to the $|m_{ef}|$ value, another spectrum metric that is also very influential and present in most formulas is $|m_{ep}|$, so we also examined the scenarios when we changed the $|m_{ep}|$ values in the formulas to the adapted version (in addition to $|m_{ef}|$), thus further enhancing the frequency usage. This approach already takes into account how often a given method appears in the UDCS of passed tests (for example, it will be able to differentiate between two elements with the same $C(m_{ef})$ value — thus reducing the chance of a tie (Debroy et al., 2010)). We also examined the case where we used adapted version of all the metrics in the formula, so that the (passed and failed) tests will also affect the suspiciousness value.

Table 7
Naïve and unique count-based spectrum metrics for the running example in Fig. 3. (Hit-based metrics are shown for reference.).

		Hit-based				Naïve count-based				Unique count-based			
		$\|m_{ef}\|$	$\|m_{ep}\|$	$\|m_{nf}\|$	$\|m_{np}\|$	$C(m_{ef})^N$	$C(m_{ep})^N$	$C(m_{nf})^N$	$C(m_{np})^N$	$C(m_{ef})^U$	$C(m_{ep})^U$	$C(m_{nf})^U$	$C(m_{np})^U$
Methods	a	2	2	0	0	11	3	0	0	4	3	0	0
	b	2	2	0	0	3	2	0	0	3	2	0	0
	f	1	1	1	1	10	10	1.67	1	1	1	1.33	1
	g	2	2	0	0	6	2	0	0	4	2	0	0

Table 8
Adapted *Russell-Rao* formulae using the naïve count-based spectrum.

$$R^N_{ef^{num}}: \frac{C(m_{ef})^N}{|m_{ef}| + |m_{nf}| + |m_{ep}| + |m_{np}|} \qquad R^N_{ef}: \frac{C(m_{ef})^N}{C(m_{ef})^N + |m_{nf}| + |m_{ep}| + |m_{np}|}$$

$$R^N_{e}: \frac{C(m_{ef})^N}{C(m_{ef})^N + |m_{nf}| + C(m_{ep})^N + |m_{np}|} \qquad R^N_{all}: \frac{C(m_{ef})^N}{C(m_{ef})^N + C(m_{nf})^N + C(m_{ep})^N + C(m_{np})^N}$$

Table 9
Suspiciousness scores of the methods for the running example in Fig. 3 calculated using the adapted *Russell-Rao* formulae in Table 8.

		Naïve count-based				Unique count-based			
		$R^N_{ef^{num}}$	R^N_{ef}	R^N_{e}	R^N_{all}	$R^U_{ef^{num}}$	R^U_{ef}	R^U_{e}	R^U_{all}
Methods	a	2.75	0.85	0.79	0.79	1.00	0.67	0.57	0.57
	b	0.75	0.60	0.60	0.60	0.75	0.60	0.60	0.60
	f	2.50	0.77	0.45	0.44	0.25	0.25	0.25	0.23
	g	1.50	0.75	0.75	0.75	1.00	0.67	0.67	0.67

The notations above are templates which can be "instantiated" with different parameters: Δ symbolizes the formula (B: *Barinel*, D: *DStar*, G: *GP13*, J: *Jaccard*, N: *Naish2*, O: *Ochiai*, R: *Russell-Rao*, S: *Sørensen-Dice* or T: *Tarantula*), $*$ shows what kind of spectrum is used (N: naïve or U: unique), and the replacement strategy is indicated in subscript.

An example of this is shown in Table 8, which presents different versions of *Russell-Rao* using the naïve count-based approach. It is important to note that these strategies are independent from the parameters, *i.e.*, the matrices are interchangeable, therefore the unique count-based values could be calculated in a similar way to the naïve ones, except that the initial matrix would be Cov^U in that case.

Table 9 shows the suspiciousness values obtained by *Russell-Rao* with the techniques described above for naïve and unique count-based spectra. One of the most striking differences from the values in the hit-based approach is that suspiciousness score values are typically higher. The highest value according to hit-based approach was 0.5 (see Table 4, *Russell-Rao* column for reference), while for the naïve and unique count-based concepts, most of the methods scored 0.6 or higher. In addition, the number of ties is much less than in the case of the traditional algorithms.

As can be seen, all naive count-based versions of the *Russell-Rao* formula associate the highest suspiciousness scores to method a, which can be attributed to the corresponding $C(m_{ef})$ value being high while $C(m_{ep})$ and other values being relatively low. For a similar reason, method f and the actually buggy method g has the second and third highest suspiciousness values in the case of $R^N_{ef^{num}}$ and R^N_{ef}, where only the $|m_{ef}|$ values are replaced. The emphasis that the covering failed tests put on these methods is suppressed by R^N_e and R^N_{all}, where other metric values are also replaced. Interestingly, in these cases method f is the least suspicious, hence g inherits the second position in the ranked lists.

Contrarily, in the case of unique count-based *Russell-Rao* formulae the buggy method is successfully located as expected. Although, using $R^U_{ef^{num}}$ and R^U_{ef} method g is in tie with method a based on the suspiciousness scores. Having higher suspiciousness score based on the new spectrum metrics and the additional contextual information that the UDCS-s contain, g, the actual buggy method, can be easily distinguished from the other methods in the case of R^U_e and R^U_{all}.

7. Empirical evaluation

The main goal of empirically evaluating the proposed approach was to compare the new algorithms' fault localization effectiveness to their traditional hit-based counterparts. Also, we were interested in finding out which of the traditional SBFL formulae 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 metrics, and we also formulate the research questions more precisely.

7.1. Subject programs

We implemented our approach for analyzing Java programs, and for the evaluation, we selected Defects4J (v2.0.0),[2] a widely used collection of Java programs and curated bugs in FL research. This benchmark contains seventeen open source Java projects with manually validated, non-trivial real bugs.

The original dataset contains 835 bugs, however, there were cases that we had to exclude from the study due to instrumentation errors or unreliable test results. For many bugs, the modification only included a method addition (i.e., an existing method was not modified by the committers during the fix). A total of 786 defects were included in the final dataset. Table 10 shows each project and their main properties. The last two columns include the statistics about the UDCS-s generated by the test cases.

7.2. Evaluation metrics

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 formulae, their relative position in the ranking list is used instead. The

[2] https://github.com/rjust/defects4j/tree/v2.0.0.

Table 10
Properties of subject programs.

Subject	Number of bugs	Size (KLOC)	Number of tests	Number of methods	Number of UDCS-s	Avg. length of UDCS-s
Chart	25	96	2.2k	5.2k	122k	8.3
Cli	39	4	0.1k	0.3k	91k	3.7
Closure	168	91	7.9k	8.4k	889k	26.0
Codec	16	10	0.4k	0.5k	6k	9.6
Collections	1	46	15.3k	4.3k	387k	4.2
Compress	47	11	0.4k	1.5k	28k	5.0
Csv	16	1	0.2	0.1k	4k	3.8
Gson	15	12	0.9k	1.0k	126k	1043.1
JacksonCore	25	31	0.4k	1.8k	27k	4.5
JacksonDatabind	101	4	1.6k	6.9k	3467k	17.8
JacksonXml	5	6	0.1k	0.5k	7k	3.8
Jsoup	89	14	0.5k	1.4k	127k	13.5
JxPath	21	21	0.3k	1.7k	215k	57.8
Lang	61	22	2.3k	2.4k	6k	4.4
Math	104	84	4.4k	6.4k	228k	14.8
Mockito	27	11	1.3k	1.4k	11k	7.8
Time	26	28	4.0k	3.6k	150k	10.1

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. Note that, if there are multiple bugs for a program version, we will use the highest rank of buggy methods.

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 algorithms are in localizing faults.

Tie-breaking

In the case of comparing ranking lists a particular issue is to handle *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 (Xu et al., 2011): (i) the average of the ranks of the faulty methods, (ii) the minimum of the ranks, or (iii) 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.

Wasted effort

Eq. (1) shows the *absolute average rank* calculation (Abreu et al., 2007; Xuan and Monperrus, 2014), where i and f are methods, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E(f) = \frac{\left|\{i|s_i > s_f\}\right| + \left|\{i|s_i \geq s_f\}\right| + 1}{2} \tag{1}$$

This metric approximates the number of methods the developers have to investigate before finding the faulty one. In other words it indicates the effort wasted by developers on non-faulty methods before finding the root cause of a bug. Smaller values are better for this metric. Note that, in case that the actual version of the subject program contains more than one faulty method, we use the E value associated with the method with the highest suspiciousness score ($min(E(f))$, where $f \in$ {faulty methods}).

A "normalized approach" of the absolute average rank is the *EXAM score* (Zakari et al., 2019) which value is the quotient of the number of methods and the bug's rank, expressed as a percentage. In other words, this metric describes the percentage of methods that need to be reviewed to find the location of the bug. The EXAM score and the absolute average rank follow from each other and result in the same order of efficiency therefore, we discussed only the latter in our paper.

Win–draw–loss

A simple comparison is to look at how many times the algorithms improve effectiveness, that is, they result in a higher absolute average rank (E) (closer to 1) than their traditional hit-based counterparts, and how many times are the E lower or equal. For this, we borrowed a basic approach from game statistics. We calculate whether an algorithm wins, draws or loses over another one based on the difference between the highest rank of faulty methods (Eq. (2)).

$$\begin{aligned} \text{win} &= \left|\{X|E_H^X > E_C^X\}\right| \\ \text{draw} &= \left|\{X|E_H^X = E_C^X\}\right| \\ \text{loss} &= \left|\{X|E_H^X < E_C^X\}\right| \\ X &\in \text{ bugs of Defects4J} \\ E_H^X &: \text{E of X using hit-based FL} \\ E_C^X &: \text{E of X using count-based FL} \\ C &\in \{\text{naïve count-based FL, unique count-based FL}\} \end{aligned} \tag{2}$$

Specifically, "loss" represents the number (percentage) of bugs for which the corresponding algorithm resulted in worse ranks than its hit-based counterpart. The "draw" value shows the number of cases where both algorithms provided the same rank, while the "win" value reveals how many times the new approach performed better. Obviously, algorithms with more wins are better.

Relative improvements

When examining ranks, the relative position is not the only interesting thing to consider. The difference between the ranks is also important. However, the evaluation approach presented in the previous subsection does not take the latter aspect into account. To determine this difference, we calculate the *relative improvement* (Beszédes et al., 2020; Le et al., 2015), which gives the mean difference between E values and their normalized values as well (Eq. (3)).

$$\begin{aligned} \text{relative improvement} &= E_H^X - E_C^X \\ \text{normalized relative improvement} &= \frac{E_H^X - E_C^X}{E_H^X} \cdot 100\% \end{aligned} \tag{3}$$

Accuracy

Several studies report that developers investigate only the first 5 or 10 elements in the recommendation (rank-)list by fault localization algorithms before giving up using the ranking (Xia et al., 2016; Kochhar et al., 2016). Therefore, we distinguished

between bugs where the minimum of faulty methods rank is less than or equal to five (*Top-5* - Eq. (4)).

$$\begin{aligned} \text{Top-5}_* &= \left|\{X | E_*^X \leq 5\}\right| \\ * \in \{&\text{hit-based FL, naïve count-based FL,} \\ &\text{unique count-based FL}\} \end{aligned} \tag{4}$$

The family of similar metrics is commonly referred to as *Top-N* or *acc@N* (Parnin and Orso, 2011). This metric represents the number of successfully localized bugs within the top-n elements of the ranking lists. Higher values are better for this metric.

Enabling improvements

A particularly interesting case of the difference between different algorithms considering the Top-N elements is when one approach produces a very low rank (*e.g.*, > 10 or > 5) while the other algorithm moves this to a higher Top-N category. As this brings a "new hope" that a bug could possibly be found by the user with the latter algorithm while it was very improbable with the former, we call these cases *enabling improvements* (Beszédes et al., 2020). Besides enabling improvements we also report *deteriorations* (Eq. (5)), *i.e.*, those cases where the algorithm produces a rank that is in a worse Top-N category than it was before. More enabling improvements (Eq. (5)) could indicate a better algorithm, but the overall accuracy of the algorithm and the baseline should also be considered during the analysis.

$$\begin{aligned} \text{deteriorations} &= \left|\left\{X | E_H^X \leq 10 \cap E_C^X > 10\right\}\right| \\ \text{enabling improvements} &= \left|\left\{X | E_H^X > 10 \cap E_C^X \leq 10\right\}\right| \end{aligned} \tag{5}$$

Significance testing

The results for the ranking comparison will be checked by testing statistical significance. Usually, a Wilcoxon sign-rank test (Conover, 1998) is used for this. However, in the context of SBFL, the test could encounter ties, *i.e.*, when both approaches report the same rank for a function. To overcome this limitation, we used an implementation of the Wilcoxon test which copes with the ties by discarding the zero-differences. In addition, we complement the Wilcoxon test with the Cliff's Delta (d) effect size measure (Grissom and Kim, 2005). Since there is no consensus in the literature about how the magnitude of the effect size should be determined we report only the d values. Note that typical thresholds are $d < 0.147$ "negligible", $d < 0.33$ "small", $d < 0.474$ "medium", otherwise "large", but $d < 0.1$, $d < 0.3$, $d < 0.5$ is used as well.

7.3. Research questions

Using the metrics defined above, we can investigate which of the new adapted formulae perform best compared to the others, which can tell us what formulas are best fitted to be extended with call frequency information. To compare the techniques we formulate the following Research Questions:

RQ1 Can naïve count-based SBFL approaches improve the performance of their traditional hit-based counterparts?

RQ2 Can unique count-based SBFL approaches perform better than their traditional hit-based counterparts?

Based on our earlier research work and observations (Vancsics et al., 2021a) we expect that we get negative results in response to **RQ1**, but the outcome of the analysis considering **RQ2** will reveal the advantages of the unique count-based approach. We investigate and analyze evidence in favor or against **RQ1** and **RQ2** in Sections 8.1 and 8.2 respectively.

8. Evaluation results

In this section, we present the results of our evaluation according to the measures described in Section 7.2. This will enable a quantitative comparisons and objective judgments of the proposed formulae in terms of fault localization effectiveness. Note that some data is absent from some tables or figures because the actual variant of the particular formula cannot be interpreted (see Section 6.2).

Table 11 shows the baseline absolute average rank values for each traditional SBFL formula and subject program. In this setting, the *Ochiai* and *DStar* formulae produce the best results (33.98–33.99) but these are closely followed by *Barinel*, *Jaccard*, *Sørensen-Dice* and *Tarantula* (with average ranks around 36). The next are *GP13* and *Naish2* (43.3 in both cases) with a more significant gap to the other formulae. Finally, the worst performing formula by far is *Russell-Rao* (135.96).

As it can be seen Closure has a much higher average than the rest of the programs. This is due to the fact that Closure has a very different purpose and consequently a special program and test suite structure as well. While the rest of the subjects are smaller Java libraries, Closure is a large compiler tool for JavaScript. Therefore, most of its tests are complex ones, and also these test cases have to go through a common starting phase (the initialization of the compiler, preprocessing, parsing, etc.) before they reach the sophisticated parts of the compiler itself, which usually enclose the root cause of bugs. Hence, they generate very large UDCS-s (see Table 10) which results in above average ranks for the faulty methods.

8.1. Naïve count-based risk formulae

Our first set of experiments investigated the relationship between the naïve count-based method and the hit-based baseline. The associated results using the win–draw–loss measure are shown in Fig. 6. The results regarding a formula and all its variants are shown on sub-figures. Red, blue and green bars with the corresponding "/", "X" and "\" patterns represent the basic statistics of losses, draws and wins respectively (wins are in favor of the naïve count-based approach). We can observe that every new formula is able to improve in a limited number of cases, but overall the hit-based approach outperforms the naïve by a huge margin. The only exception is *Russell-Rao* in Fig. 6(g) where the R_e^N and the R_{all}^N variants achieve 550/201 and 406/346 wins/losses.

Another observation is that despite the overall bad results, the $\Delta_{ef^{num}}^N$, Δ_{ef}^N and Δ_e^N, Δ_{all}^N variants seem to induce the same changes in the results. In this naïve count-based setting Δ_e^N and Δ_{all}^N variants of any formula have slightly better results compared to the $\Delta_{ef^{num}}^N$ and Δ_{ef}^N variants.

The next set of experiments dealt with the average ranks the new formulae produced. Table 12 shows the arithmetic means of the overall absolute average rank (wasted effort, E) values for each naïve count-based formula variant. For reference, we included the results of the hit-based SBFL formulae (row "hit", see Table 12), and highlighted the best results in bold in both parts.

This set of data can further be analyzed with the help of Table 13. Here, we present the relative change between the hit-based and naïve count-based formula variants. The difference is shown as a simple difference between the two formula variants for E (column "Diff."), and also as a relative change (column "Impr.") compared to the traditional hit-based formula. If these values are positive, it means that we could achieve improvement with our new formulae. Best results are highlighted in bold.

It can be seen from this data that all naïve count-based formulae performed worse than their hit-based counterparts, except for two cases, when the Δ_e^N and Δ_{all}^N variants of *Russell-Rao* achieved

Table 11
Absolute average ranks (wasted effort, *E*) of hit-based formulae. Row "All" represents the mean calculated on all bugs of the dataset. (Notations in the header: *B - Barinel*, *D - DStar*, *G - GP13*, *J - Jaccard*, *N - Naish2*, *O - Ochiai*, *R - Russell-Rao*, *S - Sørensen-Dice* and *T - Tarantula*).

Subject	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
Chart	15.94	9.18	34.34	9.46	34.34	8.82	50.36	9.46	15.94
Cli	16.68	15.29	13.94	16.58	13.94	15.40	23.10	16.58	16.68
Closure	79.44	71.63	95.49	81.10	95.49	71.64	346.62	81.10	79.43
Codec	6.78	6.50	5.25	6.53	5.25	6.53	7.22	6.53	6.78
Collections	1.00	1.00	1.00	1.00	1.00	1.00	8.00	1.00	1.00
Compress	17.49	15.94	15.36	17.14	15.36	16.02	22.74	17.14	17.49
Csv	6.50	6.50	6.50	6.50	6.50	6.50	14.81	6.50	6.50
Gson	19.27	19.23	19.50	19.17	19.50	19.23	26.53	19.17	19.27
JacksonCore	6.84	6.64	9.44	6.36	9.44	6.92	28.18	6.36	6.78
JacksonDatabind	59.53	59.11	64.45	59.57	64.45	59.12	251.63	59.57	59.53
JacksonXml	18.60	18.60	17.80	18.60	17.80	18.60	29.90	18.60	18.60
Jsoup	31.25	30.48	33.31	31.19	33.31	30.44	84.30	31.19	31.25
JxPath	44.24	54.36	116.86	45.00	116.86	54.07	221.95	45.00	44.24
Lang	5.18	4.46	4.39	4.55	4.39	4.46	5.46	4.55	5.18
Math	10.20	10.25	10.83	10.08	10.83	10.32	21.45	10.08	10.20
Mockito	26.11	25.93	42.44	26.07	42.44	25.89	81.81	26.07	26.11
Time	19.79	18.65	22.81	19.67	22.81	18.38	55.50	19.67	19.79
All	36.01	33.99	43.30	36.06	43.30	33.98	135.96	36.06	36.01

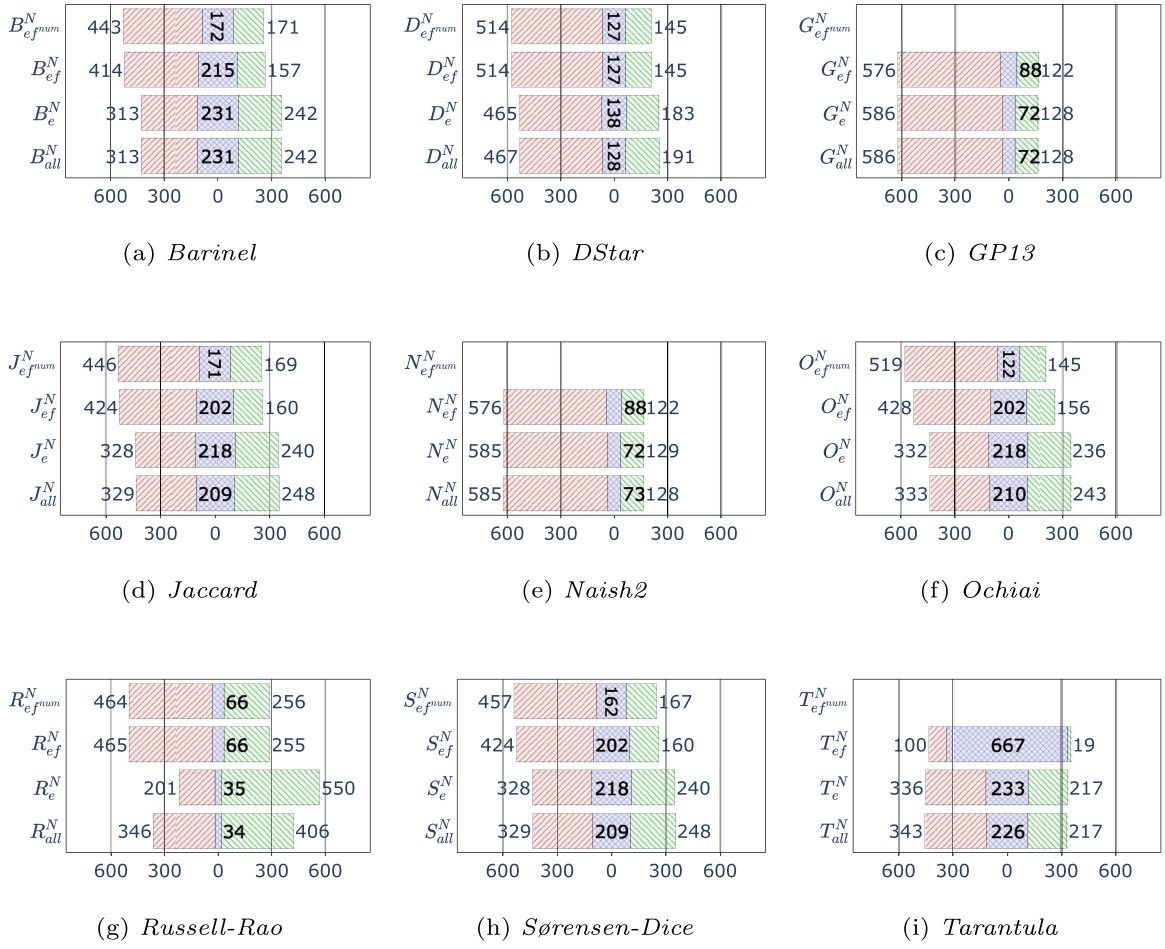


Fig. 6. Basic statistics of naïve count-based formulae (losses: red "/" pattern, draws: blue "X" pattern, wins: green "\" pattern). Wins are in favor of the naïve formulae. Results for uninterpreted formulae are absent from this figure.

46% and 7% improvement. Although this is a significant improvement, due to the poor performance of the hit-based *Russell-Rao* formula the naïve version of *Russell-Rao* is still 2–4 times worse (73.15 and 126.12) than the best performing hit-based formulae (33.98–43.30).

Table 14 addresses the comparison between the hit-based approaches and the naïve approaches from a statistical point of view. Columns noted with *p* represent the result of the Wilcoxon signed-rank tests (statistically significant values using the 0.05 threshold are highlighted in bold). The Cliff's Delta effect sizes are

Table 12
Absolute average ranks (wasted effort, E) of naïve count-based formulae. Rows "hit" represent the corresponding means from Table 11. Results for uninterpreted formulae are absent from this table. (Notations in the header: *B - Barinel*, *D - DStar*, *G - GP13*, *J - Jaccard*, *N - Naish2*, *O - Ochiai*, *R - Russell-Rao*, *S - Sørensen-Dice* and *T - Tarantula*).

Var.	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
hit	**36.01**	**33.99**	**43.30**	**36.06**	**43.30**	**33.98**	135.96	**36.06**	**36.01**
$\Delta^N_{ef\,num}$	63.46	97.94		63.55		94.76	167.78	64.11	
Δ^N_{ef}	62.60	97.94	153.43	62.89	153.43	62.59	167.41	62.89	37.09
Δ^N_{e}	42.73	66.14	153.98	43.03	153.60	41.85	**73.15**	43.03	44.34
Δ^N_{all}	42.73	66.38	153.98	43.10	153.45	41.74	126.12	43.10	44.80

Table 13
Improvement in average ranks (wasted effort, E) of naïve count-based formulae w.r.t. the hit-based results (positive values are in favor of the naïve formula). Results for uninterpreted formulae are absent from this table.

Var.	*Barinel*		*DStar*		*GP13*		*Jaccard*		*Naish2*		*Ochiai*		*Russell-Rao*		*Sørensen-Dice*		*Tarantula*	
	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.
$\Delta^N_{ef\,num}$	−27.5	−76%	−63.9	−187%			−27.5	−76%			−60.8	−178%	−31.8	−23%	−28.0	−77%		
Δ^N_{ef}	−26.6	−73%	−63.9	−187%	**−110.1**	**−254%**	−26.8	−74%	**−110.1**	**−254%**	−28.6	−84%	−31.4	−23%	−26.8	−74%	**−1.1**	**−3%**
Δ^N_{e}	**−6.7**	**−18%**	**−32.1**	**−94%**	−110.7	−255%	**−7.0**	**−19%**	−110.3	−254%	−7.9	−23%	**62.8**	**46%**	**−7.0**	**−19%**	−8.3	−23%
Δ^N_{all}	**−6.7**	**−18%**	**−32.4**	**−95%**	−110.7	−255%	**−7.0**	**−19%**	**−110.1**	**−254%**	**−7.8**	**−22%**	9.8	7%	**−7.0**	**−19%**	−8.8	−24%

Table 14
Results of the Wilcoxon signed-rank test of the naïve count-based ranks (p and d show the p-value and Cliff's Delta effect size, positive d values are in favor of the naïve formula). Results for uninterpreted formulae are absent from this table.

Var.	*Barinel*		*DStar*		*GP13*		*Jaccard*		*Naish2*		*Ochiai*		*Russell-Rao*		*Sørensen-Dice*		*Tarantula*	
	p	*d*	*p*	*d*	*p*	*d*	*p*	*d*	*p*	*d*	*p*	*d*	*p*	*d*	*p*	*d*	*p*	*d*
$\Delta^N_{ef\,num}$	<**0.001**	−0.117	<**0.001**	−0.281			<**0.001**	−0.121			<**0.001**	−0.29	<**0.001**	−0.011	<**0.001**	−0.133		
Δ^N_{ef}	<**0.001**	−0.11	<**0.001**	−0.281	<**0.001**	−0.416	<**0.001**	−0.111	<**0.001**	−0.416	<**0.001**	−0.121	<**0.001**	−0.01	<**0.001**	−0.111	<**0.001**	−0.006
Δ^N_{e}	<**0.001**	−0.057	<**0.001**	−0.164	<**0.001**	−0.425	<**0.001**	−0.059	<**0.001**	−0.424	<**0.001**	−0.062	<**0.001**	0.272	<**0.001**	−0.059	<**0.001**	−0.06
Δ^N_{all}	<**0.001**	−0.057	<**0.001**	−0.162	<**0.001**	−0.425	<**0.001**	−0.057	<**0.001**	−0.424	<**0.001**	−0.064	0.008	0.118	<**0.001**	−0.057	<**0.001**	−0.069

in columns d (positive values are in favor of the naïve formulae). Results show that, while the statistical tests show significant differences, the effects sizes are negligible to small, and in favor of the hit-based approach. The exceptions are R^N_{all} where there is no significant difference, and *GP13* and *Naish2* where the effect sizes fall into the medium category which is aligned with the data in Tables 12 and 13.

Looking at the average ranks has its drawbacks. First, outliers could distort the overall information on the performance of our formula. 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 formula. We believe that not all (absolute) rank positions are equally important, as we discussed in Section 7. Hence, in the next set of experiments we will concentrate on the *Top-5* findings.

In Table 15, we can see the number of bugs belonging to the Top-5 category (with the respective percentages), accumulated for the whole benchmark, for each naïve and, for reference, the hit-based formula. The best values for each category are highlighted in bold.

We can instantly observe that the hit-based formulae outperformed all the rest in each of the categories by a huge margin. The only exceptions are *Tarantula* whose T^N_{ef} variant performed only a bit worse than its hit-based counterpart, and *Russell-Rao* whose naïve variants were able to improve the overall worst result among the hit-based formulae. Similarly to earlier findings, the least worse performance is achieved by the variants of *Ochiai*, *Jaccard*, *Barinel*, *Sørensen-Dice* and *Tarantula*. Considering the overall performance, the best results were achieved by the Δ^N_e and Δ^N_{all} variants.

Table 15 also summarizes the enabling improvements in the columns noted with "E. Im. " for each formula. In addition, the columns noted with "Det." show the number of bugs where the new formula moved the bug in the opposite direction *i.e.*, the rank calculated by the hit-based formula was ≤ 5 while the naïve formula produced a rank that was >5.

Although each naïve formula achieves enabling improvements, the number of these cases is very low, except for *Russell-Rao* (which is aligned to what we have observed earlier).

Answer to RQ1: The naïve count-based approach cannot improve the effectiveness of the baseline, the hit-based method. What is more, there are statistically significant differences between the hit-based and naïve formula families in favor of the former. The only exception is R^N_e which achieves a significant improvement (62.8 positions on average), however this result is still behind the hit-based formulae by a huge margin. Although the naïve approaches are able to achieve enabling improvements, the number of these cases is very low (19–114 cases), and also the accuracy, regarding the Top-5 category (142–350 cases) is comparably worse than the hit-based formulae (367 cases).

Table 15
Accuracy (number of bugs in the Top-5 category) and enabling improvements achieved by the hit-based and naïve count-based formulae. Percentages are shown w.r.t. the number of all bugs. Results for uninterpreted formulae are absent from this table.

Formula	hit		$\Delta^N_{ef^{num}}$				Δ^N_{ef}				Δ^N_{e}				Δ^N_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
Barinel	357	(45.4%)	**288**	**(36.6%)**	101	32	297	(37.8%)	86	26	320	(40.7%)	65	28	320	(40.7%)	65	28
DStar	366	(46.6%)	232	(29.5%)	164	30	232	(29.5%)	164	30	276	(35.1%)	119	29	279	(35.5%)	118	31
GP13	361	(45.9%)					166	(21.1%)	214	19	163	(20.7%)	219	21	163	(20.7%)	219	21
Jaccard	358	(45.5%)	286	(36.4%)	105	33	297	(37.8%)	91	30	319	(40.6%)	69	30	**321**	**(40.8%)**	68	31
Naish2	361	(45.9%)					166	(21.1%)	214	19	163	(20.7%)	219	21	163	(20.7%)	219	21
Ochiai	**367**	**(46.7%)**	227	(28.9%)	170	30	295	(37.5%)	101	29	**322**	**(41.0%)**	74	29	**321**	**(40.8%)**	74	28
Russell-Rao	111	(14.1%)	142	(18.1%)	**10**	**41**	143	(18.2%)	9	**41**	220	(28.0%)	**5**	**114**	172	(21.9%)	**9**	**70**
Sørensen-Dice	358	(45.5%)	279	(35.5%)	113	34	297	(37.8%)	91	30	319	(40.6%)	69	30	**321**	**(40.8%)**	68	31
Tarantula	357	(45.4%)					**350**	**(44.5%)**	**8**	1	320	(40.7%)	56	19	311	(39.6%)	65	19

8.2. Unique count-based risk formulae

In this section we present the results of our experiments on the unique count-based formulae in a similar fashion to Section 8.1.

First, we investigate whether the formulae were able to improve effectiveness. A comparison of each formula with respect to the hit-based formula in terms of the win–draw–loss measure is shown in Fig. 7. The results regarding a formula and all its variants are shown on each sub-figure. Red, blue and green bars with the corresponding "/", "X" and "\" patterns represent the basic statistics of losses, draws and wins respectively (wins are in favor of the unique count-based approach).

These results draw a completely different picture than in the case of the naïve approaches: the improvement that the unique count-based formulae achieved seems to be more prevalent. We can observe 4 different patterns emerging from the results. In Fig. 7(g) we can see that all variants of *Russell-Rao* have a huge advantage over the hit-based formula. The opposite is shown in Fig. 7(i) where the hit-based *Tarantula* outperforms the unique count approaches (there are some cases of improvement, though). Next, *GP13* and *Naish2* have very similar results. Despite that they both achieve 270–277 wins, the hit-based approach still has an advantage over them. Finally, all variants of *Barinel*, *Jaccard* and *Sørensen-Dice* have an advantage in all settings. *Ochiai* and *DStar* are also similar to *Barinel*, *Jaccard* and *Sørensen-Dice*, however $D^U_{ef^{num}}$, D^U_e and $O^U_{ef^{num}}$ are behind the hit-based formulae. Overall, it seems like that the Δ^U_{ef} and Δ^U_e variants are the most beneficial configuration in this setting.

Next, we take a look at the average ranks the new formulae produced. Table 16 shows the arithmetic means of the overall absolute average rank (wasted effort, E) values for each unique count-based formula variant. Best results are highlighted in bold, and the hit-based results are included in row "hit" (see Table 16). In addition, Table 17 shows the relative change between the hit-based and unique count-based formulae. Column "Diff." is the absolute difference, while column "Impr." is relative to the corresponding hit-based formula. Furthermore, Table 18 helps in addressing the comparison from a statistical point of view. Columns noted with p represent the result of the Wilcoxon signed-rank tests (statistically significant values are highlighted in bold). The Cliff's Delta effect sizes are in columns d (positive values are in favor of the unique count-based formulae).

In this data we can recognize similar patterns to what we have seen previously. Generally, in case of 6 formulae there are statistically significant improvements. The unique count variants of *Russell-Rao* achieve huge improvements: 65.2–100.8 ranks (47%–74%) on average on the whole dataset. Despite that these are statistically significant differences with medium to large effect sizes, due to the poor performance of the hit-based *Russell-Rao* formula the best result (35.12) of R^U_e is only around the average

Table 16
Absolute average ranks (wasted effort, E) of unique count-based formulae. Row "hit" represents the corresponding means from Table 11. Results for uninterpreted formulae are absent from this table. (Notations in the header: *B - Barinel*, *D - DStar*, *G - GP13*, *J - Jaccard*, *N - Naish2*, *O - Ochiai*, *R - Russell-Rao*, *S - Sørensen-Dice* and *T - Tarantula*).

Var.	*B*	*D*	*G*	*J*	*N*	*O*	*R*	*S*	*T*
hit	36.01	33.99	**43.30**	36.06	**43.30**	33.98	135.96	36.06	**36.01**
$\Delta^U_{ef^{num}}$	24.68	35.60		24.63		36.05	70.80	24.89	
Δ^U_{ef}	**24.55**	35.60	66.73	**24.40**	66.73	**24.30**	70.60	**24.40**	36.87
Δ^U_{e}	38.63	**29.08**	67.19	38.64	67.04	36.44	**35.12**	38.64	85.35
Δ^U_{all}	38.63	29.13	67.19	38.34	67.00	36.99	54.95	38.34	74.56

of the better hit-based approaches (33.98–33.99). Considering *Tarantula* we also have statistically significant results, however the unique count-based approaches cannot improve the hit-based results, the least worse variant is T^U_{ef} which is behind the original *Tarantula* by 0.9 (2%). *GP13* and *Naish2* have almost the same results once again: their unique count-based versions have a statistically significant disadvantage behind the hit-based formulae. The unique count variants of *Barinel*, *Jaccard* and *Sørensen-Dice* achieve about 31%–32% improvement with the Δ^U_{ef} configuration. Different variants of *DStar* and *Ochiai* have 14%–28% advantage over the hit-based versions. Overall, the best result are 24.30 - $Ochiai^U_{ef}$, 24.40 - $Jaccard^U_{ef}$ and $Sørensen-Dice^U_{ef}$, 24.55 - $Barinel^U_{ef}$, and 29.08 - $DStar^U_e$.

In Table 19, we can see the number of bugs belonging to the Top-5 category (with the respective percentages), accumulated for the whole benchmark, for each unique count-based and the hit-based formula. Table 19 also presents the enabling improvements and the number of bugs which were moved in the opposite direction.

Every unique count-based formula achieves enabling improvements, but there are about the same number of deteriorations as well. On average, the number of enabling improvements is around 70–110, the two outliers in this aspect are *Russell-Rao* and *Tarantula*. In additional, *Jaccard*, *Ochiai* and *Sørensen-Dice* algorithms have fewer improvements for the Δ^U_e and Δ^U_{all} cases (42–48). Variants of *Russell-Rao* have the most improvements around 150 and 260–270 among all formulae, while variants of *Tarantula* have the least, only 6–14. There are significant improvements regarding the Top-5 accuracy as well. Traditional formulae rank at most 367 (46.7%) bugs into the Top-5, while the best unique count-based formulae have 380–391 (48.3–49.7%) bugs in the Top-5 which is about 4-7% improvement. The best accuracy values can be accounted to the Δ^U_e variant of *DStar* (391), $DStar^U_{all}$ (388) closely followed by $Jaccard^U_{ef}$, $Ochiai^U_{ef}$ and $Sørensen-Dice^U_{ef}$ (380) and $Barinel^U_{ef^{num}}$ (373). Interestingly, the improved *Russell-Rao* outperforms the baseline hit-based formulae by a huge margin.

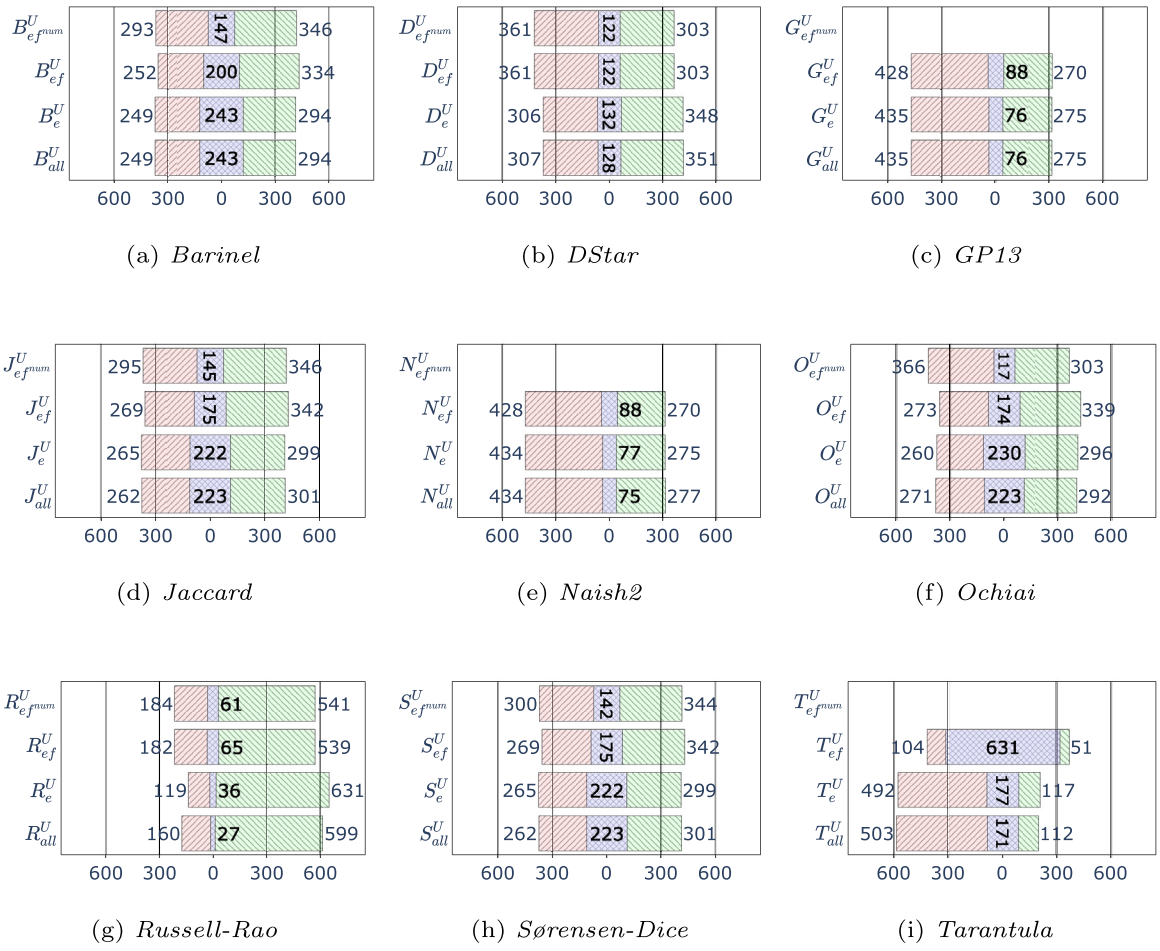


Fig. 7. Basic statistics of naïve count-based formulae (losses: red "/" pattern, draws: blue "X" pattern, wins: green "\" pattern). Wins are in favor of the unique count-based formulae. Results for uninterpreted formulae are absent from this figure.

Table 17
Improvement in average ranks (wasted effort, E) of unique count-based formulae w.r.t. the hit-based results (positive values are in favor of the unique count-based formula). Results for uninterpreted formulae are absent from this table.

Var.	*Barinel*		*DStar*		*GP13*		*Jaccard*		*Naish2*		*Ochiai*		*Russell-Rao*		*Sørensen-Dice*		*Tarantula*	
	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.
$\Delta^U_{ef^{num}}$	11.3	31%	−1.6	−4%			11.4	31%			−2.1	−6%	65.2	47%	11.2	31%		
Δ^U_{ef}	**11.5**	**31%**	−1.6	−4%	**−23.4**	**−54%**	**11.7**	**32%**	**−23.4**	**−54%**	**9.7**	**28%**	65.4	48%	**11.7**	**32%**	**−0.9**	**−2%**
Δ^U_e	−2.6	−7%	**4.9**	**14%**	−23.9	−55%	−2.6	−7%	−23.7	−54%	−2.5	−7%	**100.8**	**74%**	−2.6	−7%	−49.3	−136%
Δ^U_{all}	−2.6	−7%	**4.9**	**14%**	−23.9	−55%	−2.3	−6%	−23.7	−54%	−3.0	−8%	81.0	59%	−2.3	−6%	−38.6	−107%

Table 18
Results of the Wilcoxon signed-rank test of the unique count-based ranks (p and d show the p-value and Cliff's Delta effect size, positive d values are in favor of the unique count-based formula). Results for uninterpreted formulae are absent from this table.

Var.	*Barinel*		*DStar*		*GP13*		*Jaccard*		*Naish2*		*Ochiai*		*Russell-Rao*		*Sørensen-Dice*		*Tarantula*	
	p	d	p	d	p	d	p	d	p	d	p	d	p	d	p	d	p	d
$\Delta^U_{ef^{num}}$	0.002	0.045	0.011	−0.056			0.003	0.042			0.006	−0.064	**<0.001**	0.29	0.009	0.034		
Δ^U_{ef}	**<0.001**	0.047	0.011	−0.056	**<0.001**	−0.193	**<0.001**	0.046	**<0.001**	−0.193	0.004	0.037	**<0.001**	0.291	**<0.001**	0.046	**<0.001**	−0.004
Δ^U_e	0.09	0.01	0.049	0.043	**<0.001**	−0.197	0.152	0.013	**<0.001**	−0.196	0.143	0.013	**<0.001**	0.514	0.152	0.013	**<0.001**	−0.19
Δ^U_{all}	0.09	0.01	0.054	0.04	**<0.001**	−0.197	0.108	0.01	**<0.001**	−0.195	0.389	0.006	**<0.001**	0.446	0.108	0.01	**<0.001**	−0.189

As the results show, on Defects4J those formulae perform better that utilize the call frequency values generated by the failing test cases, which emphasizes the importance of the relationship between the failing tests and a particular method. The reason for this could be that our subjects programs have relatively large test suites with test cases that yield heavily overlapping coverage, *i.e.*, usually there are lots of tests which exercise nearly the same parts of the code in very similar ways. This contributes to the

Table 19
Accuracy (number of bugs in the Top-5 category) and enabling improvements achieved by the hit-based and unique count-based formulae. Percentages are shown w.r.t. the number of all bugs. Results for uninterpreted formulae are absent from this table.

Formula	hit		$\Delta^U_{ef^{num}}$				Δ^U_{ef}				Δ^U_{e}				Δ^U_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
Barinel	357	(45.4%)	**373**	**(47.5%)**	78	94	376	(47.8%)	65	84	354	(45.0%)	45	42	354	(45.0%)	45	42
DStar	366	(46.6%)	327	(41.6%)	128	89	327	(41.6%)	128	89	**391**	**(49.7%)**	86	111	**388**	**(49.4%)**	88	110
GP13	361	(45.9%)					266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
Jaccard	358	(45.5%)	372	(47.3%)	81	95	**380**	**(48.3%)**	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
Naish2	361	(45.9%)					266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
Ochiai	**367**	**(46.7%)**	323	(41.1%)	135	91	**380**	**(48.3%)**	74	87	363	(46.2%)	51	47	361	(45.9%)	54	48
Russell-Rao	111	(14.1%)	249	(31.7%)	**12**	**150**	248	(31.6%)	**12**	**149**	380	(48.3%)	**6**	**275**	356	(45.3%)	**10**	**255**
Sørensen-Dice	358	(45.5%)	366	(46.6%)	87	95	**380**	**(48.3%)**	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
Tarantula	357	(45.4%)					351	(44.7%)	**12**	6	258	(32.8%)	111	12	254	(32.3%)	117	14

$|m_{ep}|$ values being higher than usual, which is mitigated by the Δ^U_{ef} and $\Delta^U_{ef^{num}}$ strategies, hence their advantage over the other formulae.

Answer to RQ2: In 6 out of 9 cases, the unique count-based approaches can improve the effectiveness of their hit-based counterparts by 5-101 positions on average, with a relative improvement of 14-74% and statistically significant differences. Also, all new formulae are able to achieve enabling improvements, and the accuracy regarding the number of bugs in the Top-5 category is increased by about 4-7% as well. Considering the magnitude of differences and the consistency of the improvements Δ^U_{ef} offers the best performance in the unique count-based setting, but the other concepts are just marginally behind.

9. Discussion

We hypothesized that there is a correlation between the number of calls from different contexts and the efficiency of fault localization algorithms. To better support this, we present one positive (Section 9.1) and one negative (Section 9.2) example from Defects4J, and we analyze the cases in which the new approach could or could not be an effective solution. The time and space complexity of the algorithms is crucial for usability, so we overview the costs and the resource requirements in Section 9.3. Finally, we discuss threats to validity in Section 9.4.

9.1. A positive example from the benchmark

To understand how the proposed approach achieves 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. Fig. 8 visualizes the test-to-code relations of this bug schematically.

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. $C(m_{ef})$ values are 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 $\frac{1}{8}$. (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 formulae 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.

9.2. A negative example from the benchmark

We also looked for examples where the hit-based algorithm outperforms the count-based approaches. Bug 19 from the *Cli* project is one of these cases. Its "cleaned" structure is shown in Fig. 9 in a similar fashion to Fig. 8. There is one failing test (*testUnrecognizedOption2*) and a myriad of passing tests in this scenario. The faulty method is *processOptionToken*. However, the most suspicious element according to the hit-based algorithm is *burstToken*, and the first item on the unique count-based suspiciousness list is *parse*.

We examined the coverage-based relationships of these code elements. We found that all three methods are covered by the failing test (*testUnrecognizedOption2*) and by several passing tests as well. The buggy method and the two correct methods are associated with 61, and 18–98 passing test cases respectively. Consequently, the hit-based approach of *Barinel* ranks the faulty method to the 3rd position because it has the same $|m_{ef}|$ value as the other two methods, but there are two methods that have lower $|m_{ep}|$ values than the faulty one. The analysis showed that there are a number of other methods that are related to the failing test. They are covered by many (100+) passing tests, but – due to their higher $|m_{ep}|$ values – these are behind the faulty method on the suspiciousness list.

If we consider the adapted spectrum metrics, then the weight between the faulty method and the failed test remains 1, but

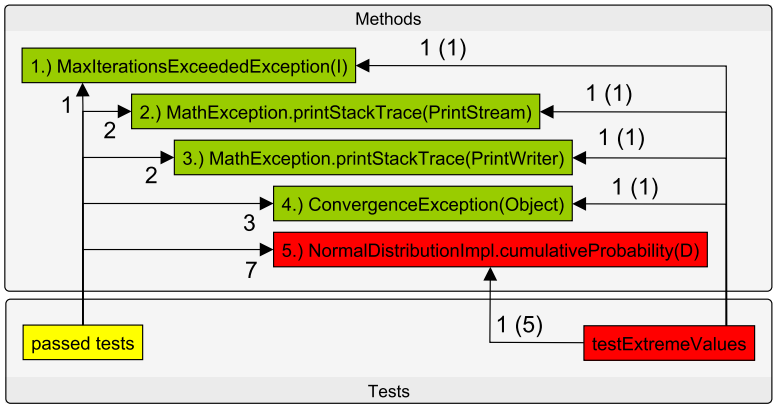


Fig. 8. Methods and tests related to the Math-103 bug. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

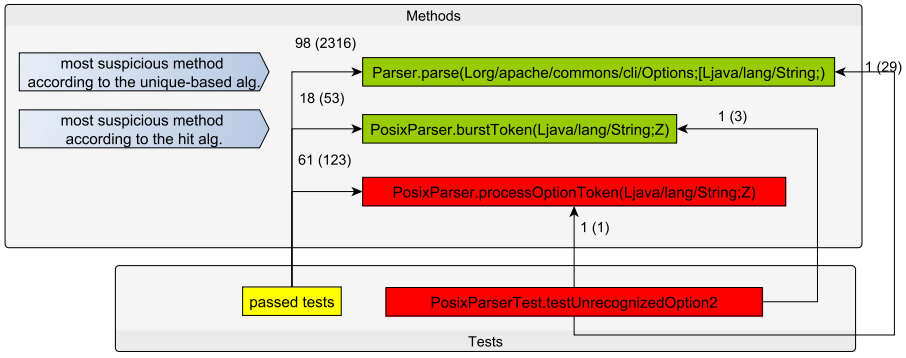


Fig. 9. Methods and tests related to the Cli-19 bug.

for the other two methods it becomes 3 and 29. In addition to these two methods there are many more that have higher $C(m_{ef})$ values than the faulty method, and although some have higher $|m_{ep}|$ values as well their failing/passing ratio is still higher than the faulty's, hence (according to the adapted formulae) they are more suspicious than the faulty. For example, the *parse* method is covered by many passed tests (1 failing and 98 passing), but the value of $C(m_{ef})$ is also high (29); or in the case of *burstToken*, there are $1 + 18$ (failing + passing) tests and $C(m_{ef})$ is 3, but the calculated score is still higher than the faulty method's score.

Overall, hit-based algorithms "skip" the differences in the frequency of calls. Since they only utilize binary information of coverage they cannot realize the number of different contexts from which the actual code element was executed, thus, they "mislead" themselves. The missing information helps the fault localization methods in cases where the faulty code is only covered by a few contexts. However, if this condition is not met, the frequency information supports debugging methods with relevant data. That is, in cases where the value of $C(m_{ep})$ is low, count-based functions perform worse than hit-based approaches in many cases.

It seems like that unusually high $C(m_{ef})$ values hinder the fault localization capabilities of count-based approaches. In addition, similar $C(m_{ef})$ and $|m_{ef}|$ values and also their ratio to the $C(m_{ef})$ values have an impact on the performance. In cases where the number of unique contexts is high and the use of frequency does not improve the performance of the algorithms (see negative example), further analyzes are needed to investigate the causes of the performance drops.

9.3. Time and space complexity

Space and time complexity are important aspects of FL algorithms. If the algorithms have to work on a very large data set and therefore their execution time is high, them the usability of them in practice deteriorates significantly. (It is hard to find a place for an algorithm in the industry which searches for faulty methods for hours or days, as developers' needs dictate a more seamless and faster integration into their debugging process.) We examined the resource requirements of the hit-based, naïve and unique count-based methods in terms of storage.

The hit procedures (as we saw in Section 3) store binary data (whether a test covers a method or not) that is, we can store the necessary information in an $N \times M$-sized binary matrix (where N is the number of tests and M is the number of methods). The naïve approach also works on a matrix of similar size ($N \times M$), however, the values in the matrix are not binary (0 or 1), but natural numbers. Binary values are stored in the boolean variable (typically 1 byte) however, natural numbers are usually stored in 4 bytes (for example: using integer type). That is, we can conclude that the naïve approach has four times greater space complexity than the hit-based algorithm.

To calculate the coverage matrix (Cov^U) of the unique count-based method, we need another auxiliary matrix whose items are tests and UDCS-s. Its size is $N \times U$, where N = number of tests, U = number of UDCS, and its value is either 0 or 1, depending on whether the (unique deepest) call stack was generated during the test or not. By traversing this matrix, we aggregate the frequency of the methods in the UDCS-s generated by the test and collect them in the Cov^U matrix. As can be seen in Table 10, the number

of UDCS-s (column 6) is on average 10–100 times larger than the number of methods (column 5), thus increasing the space and time complexity of the fault localization algorithms (due to storage and "traversal") therefore will be somewhat more expensive to execute the algorithm than naïve or hit-based methods. The unique count-based matrix (Cov^U) has similar parameters to the naïve matrix (Cov^N): 4 bytes are required to represent an element, *i.e.* this method also has four times the complexity compared to Cov^H.

The total space-cost of the UDCS-based concept is $N \cdot U \cdot$(1 byte)$+N \cdot M \cdot$(4 bytes), as opposed to the cost of the hit-based and the naive-based algorithms' $N \cdot M \cdot$(1 byte) and $N \cdot M \cdot$(4 bytes). However, modern hardware has such large storage capacity that it can handle the storage needs of count-based algorithms without any problems.

It is enough to examine each vector once (for a given code element) to calculate the spectrum metrics (Section 3) which form the basis of the fault localization formulae (Table 3). For count-based algorithms, if the value of a matrix element is 0 (*i.e.*, not covered), we must also examine the coverage vector for the given test to determine m_{nf} and m_{np}. These "extra" operations increase the time complexity of these methods. If the matrix contains many 0 values, we have to examine the test-vectors very often, but with the current (hardware-)performance, the resulting increase in execution time is not significant compared to the hit-based concept.

Overall, although the space and time complexity of count-based algorithms are slightly higher, it does not have a significant negative impact on usability, so it may represent an easy trade-off.

9.4. Threats to validity

One of the most critical points of the presented method is the adaptation of the spectrum metrics. If the "transformation" is not appropriate, the performance of the formulae is significantly reduced. The calculation of $C(m_{ef})$ and $C(m_{ep})$ can be given relatively intuitively (the values in the naïve or unique matrix are aggregated instead of the number of 1 values in the hit matrix) however, defining $C(m_{nf})$ and $C(m_{nf})$ metrics require much more care. In these cases, our approach used the average "full-coverage" of the examined methods and took these as the basis for aggregation. However, a number of other adaptive functions can be created that can increase efficiency with the modified formula but these require further investigation and form the basis of future work.

In our study we experimented with only four strategies of adapting the risk formulae. However, there are other possible scenarios, for example the combined use of the m_{ef}, $C(m_{ef})$ and m_{np}, $C(m_{np})$ values, etc. The presented concepts contain no technical or a theoretical limitations that would prevent other possible scenarios from being considered. However, the analysis of them requires further work and may provide grounds for future research.

Another crucial question is the suitability of the proposed algorithms for locating crash-faults. The naïve count-based algorithm is a special case of the hit-based concept and (similar to the hit-based methods (Tang et al., 2017)) it can be used to analyze crash-type faults. Several papers (Schröter et al., 2010; Wong et al., 2014; Wu et al., 2014; Gong et al., 2014) have presented that call stacks are suitable for FL algorithms to find the location of the bugs in the source code. Our unique count-based approach combines "classic" (hit) coverage-based and call stack-based methods by completing the matrix with information extracted from the (unique deepest) call stack.

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 49 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 or for other practical reasons (*e.g.*, there were no modified methods) these were not examined and analyzed. This selection was in no ways influenced by the results 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.

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. Currently, it is not known if the concept could be successfully adapted to other granularities such as statements. It remains future work to investigate this aspect.

10. Related work

10.1. Spectrum-based fault localization

Automated fault localization techniques have been around for more than three decades. There have been several shorter (Parmar and Patel, 2016; Agarwal and Agrawal, 2014) and more elaborated surveys (Wong et al., 2016) written that summarize the current state, the challenges and the future work of Software Fault Localization. In addition, researchers have performed various empirical studies to compare the effectiveness of different methods on artificial and real faults (Abreu et al., 2009a; Pearson et al., 2017), and also found clever ways to combine various methods to improve the overall efficiency and effectiveness (Zou et al., 2019). While there are a lot of other fault localization techniques, *e.g.*, program slicing-based (Zhang et al., 2005; Cao et al., 2014), mutation-based (Papadakis and Le Traon, 2015, 2014; Li et al., 2019b), and mixed (Simomura, 1993) methods as well, 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 (Le et al., 2013; Steimann et al., 2013). Often the faulty element is placed far from the top of the rank-list (Xia et al., 2016), and this way the developer will refuse to use any help for localizing bugs. Abreu et al. (2007) 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.

The most universally used spectra are based on individual statements or methods (Shu et al., 2013; Santelices et al., 2009; Oo and Oo, 2020). Harrold et al. (1998) proposed several other types of program spectra, *i.e.*, counting branches, execution trace, execution path, etc., however, the hit-based approach remained the most studied one.

There is a plethora of different risk formulae proposed by researchers that use hit-based spectrum metrics (good summaries can be found in Heiden et al. (2019), Naish et al. (2011)). Moreover, some researchers experimented with automatically deriving new formulae (Yoo, 2012; Neelofar et al., 2018).

10.2. 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 (Shu et al., 2013; Santelices et al., 2009; Oo and Oo, 2020).

Harrold et al. (1998) proposed several other types of program spectra, however, the hit-based approach remained the most studied one. Abreu et al. (2010) performed an empirical study on the count spectra for Fault Localization using Barinel (Abreu et al., 2009a). They compared it to classical SBFL algorithms, however it did not improve the average ranks on real programs and bugs.

Classical SBFL algorithms are limited for locating faults in loops. Shu et al. (2016) improved the Tarantula metric by extending it with counting the statement frequency. Likewise, our method tackles the same issue but in a different manner. Abreu et al. (2010) raised the issue of repeating function calls distorting the count spectra, but they did not investigated this issue in detail. Harrold et al. (1998, 2000) investigated several types of program spectra on small programs and they concluded that there is a correlation between the differences of the spectrum and the occurrence of faults, however they did not consider fault localization efficiency in general.

Lee et al. (2010) 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. (2015) 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 approach filters out the repetitive method calls, which is similar to our approach.

10.3. 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. (2020) used the Ochiai formula and expanded it with function call-chains context information.

Jiang et al. (2012) used the stack trace to locate null pointer exceptions more efficiently. Gong et al. (2014) generate stack traces to help localize crash faults. They were able to find 64% crashing faults in Firefox 3.6.

Zhu et al. (2017) investigated whether function call sequences improve the efficiency of fault localization. There is empirical evidence that stack traces help developers fix bugs (Schröter et al., 2010). Furthermore, Zou et al. (2018) showed that stack traces can be used to locate crash-faults. Jiang et al. (2012) used the stack trace to locate null pointer exceptions more efficiently. Gong et al. (2014) generate stack traces to help localize crash faults and combine the generated passing and failing trace, thus creating spectra, which is used by the SBFL algorithm, *e.g.*, Ochiai (they were able to find 64% crashing faults in Firefox 3.6.)

In an earlier version of our work (Vancsics et al., 2021a), we extended the idea of call-chains context, by investigating the Unique Deepest Call Stacks (UDCS). We used five formulae that are using the same numerator: ef, *i.e.*, the number of failing test cases executing a function. We concluded that the call-frequency based *Jaccard* formula is the most successful at localizing bugs among the other four formulae. This paper provides a more thorough investigation of count-based spectra and gives a more general and complete overview of the topic.

10.4. Count-based fault localization algorithms

Abreu et al. (2010) performed an empirical study on the count spectra for Fault Localization using Barinel (Abreu et al., 2009a). They compared it to classical SBFL algorithms, however it did not improve the performance on real programs and bugs. The result was that their method can assist developers finding the root cause of the discovered bugs. The reasons for that are: (i) Barinel is based on Bayesian probability which expects the failure probability to be an or-model (ii) the error is due to the choice of the test suite. Lee et al. (2010) proposed a new approach that improves SBFL by using frequency counts of test coverage. The authors evaluated their approach on several metrics (*e.g.*, Kulczynski2, Ochiai, Jaccard) however, they concluded that counting only the statement frequency can lead to distortion in the statistics. Laghari et al. (2015) 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.

11. Conclusion

11.1. Summary

We proposed several new Spectrum-Based Fault Localization formulae that are based on two kinds of method call frequency information. The basic naïve concept is using simple call counts. The more advanced unique counts concept relies 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. The unique count-based concept eliminates the problem of very large number of repetitions due to loops. We adapted nine traditional hit-based SBFL formulae to both kinds of count-based information, and empirically verified how much we can improve fault localization effectiveness on the Defects4J benchmark.

We showed that the naïve approaches cannot improve the effectiveness of their hit-based counterparts. What is more, there are statistically significant differences between the hit-based and naïve formula families in favor of the former. The only exception is R_e^N (an adapted *Russell-Rao*) which achieves a significant improvement (62.8 positions on average), however this result is still behind the hit-based formulae by a huge margin. Although the naïve approaches are able to achieve enabling improvements, the number of these cases is very low (19-114 cases), and also the accuracy, regarding the Top-5 category (142-350 cases) is comparably worse than the hit-based formulae (367 cases).

On the contrary, the unique count approaches can improve the effectiveness of their hit-based counterparts in 6 out of 9 cases by 5-101 positions on average, with a relative improvement of 14-74% and statistically significant differences. Also, all new formulae are able to achieve enabling improvements, and the accuracy regarding the number of bugs in the Top-5 category is increased by about 7% as well which is important in terms of practical usability of SBFL in general. We conclude that the best candidates to benefit from call frequency information are the unique count-based adaptations of *Jaccard*, *Ochiai*, *Sørensen-Dice*, *DStar* and *Barinel*. In addition, considering the magnitude of differences and the consistency of the improvements Δ_{ef}^{U} offers the best performance in the unique-based setting, but other formulae are just marginally behind.

11.2. Future work

We have several plans for future work. We would like to work on adapting the approach to statement-level granularity, as well as experimenting with other SBFL formulae and on other

benchmarks. We believe that there is still potential in the count-based spectra beyond the approaches investigated by this paper. The naïve counts could be improved by other variations using other kinds of contexts, not just call information. Additionally, other adaptive functions can be developed and investigated.

To enable the reproduction of our experiments and additional analyses, we made the measurement framework and the final results publicly available on GitHub (Vancsics et al., 2021c) and Figshare (Vancsics et al., 2021b).

CRediT authorship contribution statement

Béla Vancsics: Conceptualization, Methodology, Software, Investigation, Formal analysis, Validation, Writing – original draft, Writing – review & editing. **Ferenc Horváth:** Methodology, Resources, Software, Investigation, Formal analysis, Writing – original draft, Writing – review & editing. **Attila Szatmári:** Methodology, Writing – original draft. **Árpád Beszédes:** Methodology, Writing – review & editing, Supervision.

Declaration of competing interest

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

Acknowledgments

This research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (MILAB) and grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary. This publication was also supported by the University of Szeged Open Access Fund under the grant number 5416.

References

Abreu, R., Gonzalez-Sanchez, A., van Gemund, A.J., 2010. Exploiting count spectra for Bayesian fault localization. In: Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, Association for Computing Machinery, New York, NY, USA, pp. 1–10.

Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J., 2009a. A practical evaluation of spectrum-based fault localization. J. Syst. Softw. 82 (11), 1780–1792.

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

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2009b. Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, IEEE Press, pp. 88–99.

Agarwal, P., Agrawal, A., 2014. Fault-localization techniques for software systems. ACM SIGSOFT Softw. Eng. Not. 39, 1–8. http://dx.doi.org/10.1145/2659118.2659125.

Beszédes, A., Horváth, F., Di Penta, M., Gyimóthy, T., 2020. Leveraging contextual information from function call chains to improve fault localization. In: Proceedings of 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, IEEE Press, pp. 468–479.

Cao, H., Jiang, S., Ju, X., Yanmei, Z., Yuan, G., 2014. Applying association analysis to dynamic slicing based fault localization. IEICE Trans. Inf. Syst. E97.D, 2057–2066. http://dx.doi.org/10.1587/transinf.E97.D.2057.

Collofello, J.S., Cousins, L., 1987. Towards automatic software fault location through decision-to-decision path analysis. In: Proceedings of the 1987 International Workshop on Managing Requirements Knowledge (AFIPS). IEEE, p. 539.

Conover, W.J., 1998. Practical Nonparametric Statistics, Vol. 350. John Wiley & Sons.

Dallmeier, V., Lindig, C., Zeller, A., 2005. Lightweight defect localization for java. In: European Conference on Object-Oriented Programming. Springer, pp. 528–550.

de Souza, H.A., Chaim, M.L., Kon, F., 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. arXiv abs/1607.04347.

Debroy, V., Wong, W.E., Xu, X., Choi, B., 2010. A grouping-based strategy to improve the effectiveness of fault localization techniques. In: 2010 10th International Conference on Quality Software. IEEE, pp. 13–22.

Eric Wong, W., Qi, Y., 2011. Bp neural network-based effective fault localization. Int. J. Softw. Eng. Knowl. Eng. 19, http://dx.doi.org/10.1142/S021819400900426X.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. 27 (2), 99–123.

Gong, L., Zhang, H., Seo, H., Kim, S., 2014. Locating crashing faults based on crash stack traces. CoRR abs/1404.4100. URL http://arxiv.org/abs/1404.4100.

Grissom, R.J., Kim, J.J., 2005. Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates Publishers.

Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L., 2000. An empirical investigation of the relationship between spectra differences and regression faults. Softw. Test. Verif. Reliab. 10 (3), 171–194.

Harrold, M.J., Rothermel, G., Wu, R., Yi, L., 1998. An empirical investigation of program spectra. In: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, Association for Computing Machinery, New York, NY, USA, pp. 83–90. http://dx.doi.org/10.1145/277631.277647.

He, H., Ren, J., Zhao, G., He, H., 2020. Enhancing spectrum-based fault localization using fault influence propagation. IEEE Access 8, 18497–18513.

Heiden, S., Grunske, L., Kehrer, T., Keller, F., Van Hoorn, A., Filieri, A., Lo, D., 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. Softw. - Pract. Exp. 49 (8), 1197–1224.

Jiang, S., Li, W., Li, H., Yanmei, Z., Zhang, H., Liu, Y., 2012. Fault localization for null pointer exception based on stack trace and program slicing. In: Proceedings of the 2012 12th International Conference on Quality Software. IEEE, IEEE Computer Society, USA, pp. 9–12. http://dx.doi.org/10.1109/QSIC.2012.36.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. IEEE/ACM, IEEE/ACM, pp. 273–282.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, Association for Computing Machinery, New York, NY, USA, pp. 437–440. http://dx.doi.org/10.1145/2610384.2628055.

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

Kochhar, P.S., Xia, X., Lo, D., Li, S., 2016. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, ACM, pp. 165–176.

Laghari, G., Murgia, A., Demeyer, S., 2015. Localising faults in test execution traces. In: Proceedings of the 14th International Workshop on Principles of Software Evolution. ACM, ACM, pp. 1–8. http://dx.doi.org/10.1145/2804360.2804361.

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

Le, T.-D.B., Lo, D., Thung, F., 2015. Should I follow this fault localization tool's output? Empir. Softw. Eng. 20 (5), 1237–1274.

Le, T.B., Thung, F., Lo, D., 2013. Theory and practice, do they match? A case with spectrum-based fault localization. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. IEEE, IEEE Press, pp. 380–383.

Lee, H.J., Naish, L., Ramamohanarao, K., 2010. Effective software bug localization using spectral frequency weighting function. In: Proceedings of the 34th Annual Computer Software and Applications Conference. IEEE, IEEE Press, pp. 218–227. http://dx.doi.org/10.1109/COMPSAC.2010.26.

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

Li, Y., Liu, C., 2014. Effective fault localization using weighted test cases.. J. Softw. 9 (8), 2112–2119.

Li, Z., Wu, Y., Wang, H., Liu, Y., 2019b. Test oracle prediction for mutation based fault localization. In: Li, Z., Jiang, H., Li, G., Zhou, M., Li, M. (Eds.), Software Engineering and Methodology for Emerging Domains. Springer Singapore, Singapore, pp. 15–34.

Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I., 2005. Scalable statistical bug isolation. SIGPLAN Not. 40 (6), 15–26. http://dx.doi.org/10.1145/1064978.1065014.

Liu, C., Yan, X., Yu, H., Han, J., Yu, P.S., 2005. Mining behavior graphs for "backtrace" of noncrashing bugs. In: Proceedings of the 2005 SIAM International Conference on Data Mining. SIAM, pp. 286–297.

Lucia, L., Lo, D., Jiang, L., Thung, F., Budi, A., 2014. Extended comprehensive study of association measures for fault localization. J. Softw. Evol. Process 26 (2), 172–219.

Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C., 2014. Slice-based statistical fault localization. J. Syst. Softw. 89, 51–62.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32.

Neelofar, N., Naish, L., Ramamohanarao, K., 2018. Spectral-based fault localization using hyperbolic function. Softw. - Pract. Exp. 48 (3), 641–664.

Oo, C., Oo, H.M., 2020. Spectrum-based bug localization of real-world java bugs. In: Lee, R. (Ed.), Software Engineering Research, Management and Applications. Springer International Publishing, Cham, pp. 75–89. http://dx.doi.org/10.1007/978-3-030-24344-9_5.

Papadakis, M., Le Traon, Y., 2014. Effective fault localization via mutation analysis: A selective mutation approach. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. ACM, Association for Computing Machinery, New York, NY, USA, pp. 1293–1300. http://dx.doi.org/10.1145/2554850.2554978.

Papadakis, M., Le Traon, Y., 2015. Metallaxis-FL: Mutation-based fault localization. Softw. Test. Verif. Reliab. 25 (5–7), 605–628. http://dx.doi.org/10.1002/stvr.1509.

Parmar, P., Patel, M., 2016. Software fault localization: A survey. Int. J. Comput. Appl. 154, 6–13. http://dx.doi.org/10.5120/ijca2016912206.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers?. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, ACM, pp. 199–209.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering. IEEE, IEEE Press, pp. 609–620. http://dx.doi.org/10.1109/ICSE.2017.62.

Ren, X., Ryder, B.G., 2007. Heuristic ranking of java program edits for fault localization. In: ISSTA '07.

Reps, T., Ball, T., Das, M., Larus, J., 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. ACM SIGSOFT Softw. Eng. Not. 22 (6), 432–449.

Sahoo, S.K., Criswell, J., Geigle, C., Adve, V., 2013. Using likely invariants for automated software fault localization. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 139–152.

Santelices, R., Jones, J.A., Yanbing Yu, Harrold, M.J., 2009. Lightweight fault-localization using multiple coverage types. In: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering. IEEE, IEEE Press, pp. 56–66. http://dx.doi.org/10.1109/ICSE.2009.5070508.

Schröter, A., Bettenburg, N., Premraj, R., 2010. Do stack traces help developers fix bugs?. In: Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, pp. 118–121. http://dx.doi.org/10.1109/MSR.2010.5463280.

Shu, G., Sun, B., Podgurski, A., Cao, F., 2013. MFL: Method-level fault localization with causal inference. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, IEEE Press, pp. 124–133. http://dx.doi.org/10.1109/ICST.2013.31.

Shu, T., Ye, T., Ding, Z., Xia, J., 2016. Fault localization based on statement frequency. Inform. Sci. 360, 43–56. http://dx.doi.org/10.1016/j.ins.2016.04.023, URL http://www.sciencedirect.com/science/article/pii/S0020025516302663.

Simomura, T., 1993. Critical slice-based fault localization for any type of error. IEICE Trans. Inf. Syst. 76, 656–667.

Steimann, F., Frenkel, M., Abreu, R., 2013. 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. ACM, Association for Computing Machinery, New York, NY, USA, pp. 314–324. http://dx.doi.org/10.1145/2483760.2483767.

Tang, C.M., Chan, W., Yu, Y.T., Zhang, Z., 2017. Accuracy graphs of spectrum-based fault localization formulas. IEEE Transactions on Reliability 66 (2), 403–424.

Vancsics, B., Horváth, F., Szatmári, A., Beszédes, A., 2021a. Call frequency-based fault localization. In: Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'21). IEEE, IEEE Press, pp. 365–376.

Vancsics, B., Horváth, F., Szatmári, A., Beszédes, A., 2021b. Supplemental material for paper 'Fault Localization Using Function Call Frequencies' - Figshare. online. URL https://doi.org/10.6084/m9.figshare.13537298.v3.

Vancsics, B., Horváth, F., Szatmári, A., Beszédes, A., 2021c. Supplemental material for paper 'Fault Localization Using Function Call Frequencies' - GitHub. online. URL https://github.com/sed-szeged/SpectrumBasedFaultLocalization/tree/CallFrequencyBasedFL.

Wong, W.E., Debroy, V., Gao, R., Li, Y., 2013. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.

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

Wong, W.E., Qi, Y., 2006. Effective program debugging based on execution slices and inter-block data dependency. J. Syst. Softw. 79 (7), 891–903.

Wong, W.E., Sugeta, T., Qi, Y., Maldonado, J.C., 2005. Smart debugging software architectural design in SDL. J. Syst. Softw. 76 (1), 15–28.

Wong, C.-P., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 181–190.

Wu, R., Zhang, H., Cheung, S.-C., Kim, S., 2014. Crashlocator: locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 204–214.

Xia, X., Bao, L., Lo, D., Li, S., 2016. "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: Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME). IEEE, IEEE Press, pp. 267–278.

Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22 (4), 31:1–31:40.

Xu, X., Debroy, V., Eric Wong, W., Guo, D., 2011. Ties within fault localization rankings: Exposing and addressing the problem. Int. J. Softw. Eng. Knowl. Eng. 21 (06), 803–827.

Xuan, J., Monperrus, M., 2014. Learning to combine multiple ranking metrics for fault localization. In: 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, pp. 191–200.

Yilmaz, C., Paradkar, A., Williams, C., 2008. Time will tell. In: 2008 ACM/IEEE 30th International Conference on Software Engineering. IEEE, pp. 81–90.

Yoo, S., 2012. Evolving human competitive spectra-based fault localisation techniques. In: Proceedings of the 2012 International Symposium on Search Based Software Engineering. Springer, pp. 244–258.

Zakari, A., Lee, S.P., Hashem, I.A.T., 2019. A single fault localization technique based on failed test input. Array 3, 100008.

Zhang, X., He, H., Gupta, N., Gupta, R., 2005. Experimental evaluation of using dynamic slices for fault location. In: Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging, AADEBUG 2005. ACM, ACM, New York, NY, USA, pp. 33–42. http://dx.doi.org/10.1145/1085130.1085135.

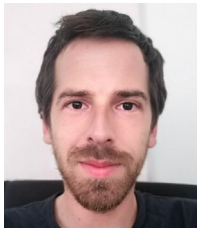
Zhang, M., Li, X., Zhang, L., Khurshid, S., 2017. Boosting spectrum-based fault localization using PageRank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, ACM, pp. 261–272.

Zhao, G., He, H., Huang, Y., 2021. Fault centrality: boosting spectrum-based fault localization via local influence calculation. Appl. Intell. 1–23.

Zhu, H., Peng, T., Xiong, L., Peng, D., 2017. Fault localization using function call sequences. Procedia Comput. Sci. 107, 871–877. http://dx.doi.org/10.1016/j.procs.2017.03.186.

Zou, D., Liang, J., Xiong, Y., Ernst, M., Zhang, L., 2018. An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng. PP, http://dx.doi.org/10.1109/TSE.2019.2892102.

Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L., 2019. An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng..



Béla Vancsics got M.Sc. degrees in Business Information Technology at University of Szeged in 2014.

After, he started the Ph.D. studies (Doctoral School of Computer Science) and currently he works in pre-doctoral researcher position at the same institution.

His main interest is the analysis and improvement of software testing processes and he focuses on the topic of software measurement especially on code coverage measurement and its applications in software testing and fault localization algorithms.

He has published several (conference- and journal) publications on these topics with many co-authors.

Ferenc Horváth is a pre-doctoral research fellow and an assistant lecturer at the Department of Software Engineering, University of Szeged.

His main research topics are the analysis and the improvement of software testing processes.

He is also interested in fault detection and fault localization, usability testing and software metrics in general.

He is a co-author of several publications of these topics.

Besides the academic activities, Horváth works as a software developer in R&D projects of the university.

Attila Szatmári got his M.Sc. degree in Computer Science at the University of Szeged in 2020.

He continued his studies as a Ph.D. student in Computer Science at the University of Szeged.

His research topic is finding contextual information to help fault localization algorithms successfully localize bugs.

He is a co-author of publications in this topic.

Dr. Árpád Beszédes obtained his Ph.D. degree in computer science from the University of Szeged in 2005, and currently is an associate professor at the same institution.

His active research area is static and dynamic program analysis with special emphasis on software testing and debugging applications.

Dr. Beszédes has over 100 publications, and is regularly invited to serve in the Program Committees of various software engineering conferences, and as a reviewer for software engineering and computer science journals.