A Case Against Coverage-Based Program Spectra

Péter Attila Soha[1], Tamás Gergely[1], Ferenc Horváth[1], Béla Vancsics[1], Árpád Beszédes[1]
[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
{psoha, gertom, hferenc, vancsics, beszedes}@inf.u-szeged.hu

***Abstract*—Spectrum-Based Fault Localization (SBFL) is a semi-automated debugging technique that gained popularity in the last decades due to its intuitive approach and relatively simple implementability. Despite this, the performance of practical SBFL techniques in terms of fault localization capability does not reach the threshold that would enable their acceptance by professional programmers. Almost all modern SBFL approaches are based on the *code coverage-based spectrum*, and on the assumption that a code element covered by failing tests should be treated as suspicious. However, it is easy to see that this is an over-approximation because many code elements may be executed that do not contribute to the test output, hence serving as noise in the process. A possible solution is to use *backward dynamic program slices* as program spectra computed from the output statement as the criterion, instead of the coverage. There are very few theoretical and practical results about this approach, so in this work we revisit the method and show how much more inferior coverage-based spectra are compared to slice-based spectra, both on theoretical and practical levels. We argue that code coverage-based SBFL is currently in a research pit due to this inherent approximation, and research on slice-based spectra should once more attain a much higher focus.**

***Index Terms*—Spectrum-Based Fault Localization, Automated Debugging, Assertions, Backward Dynamic Program Slice.**

I. Introduction

In Spectrum-Based Fault Localization (SBFL) [1], [2], [3], [4], [5], test executions are recorded to produce the *program spectrum*. It includes dynamic information about test cases with respect to the code executed. It might be simply the code coverage (hit-based spectrum), or more elaborated (count-based spectrum). Code elements can be treated at different levels of granularity. The spectrum also includes test outcomes, often simply as pass or fail. The spectrum is then used to infer statistical information about the probability of code elements being faulty.

The precursor works to SBFL are a couple of decades old [6], [7], and the technique attained popularity in the software maintenance, testing, and automated program repair communities due to its relative ease of implementation and low computation cost in the basic case. Although there have been different types of program spectra proposed over the years [1], [8], [9], [10], [11], [12], [13], the prevalent approach is still the simplest one, the *hit-based* spectrum. It simply records the binary coverage information of code elements upon test case execution.

Coverage-based SBFL is straightforward to implement using existing profiling tools, and the corresponding algorithms to calculate faultiness are simple, hence this provided a fruitful ground for a large set of SBFL techniques proposed in the literature. A crucial element with these algorithms is the *suspiciousness formulas* that rely on four basic statistical measures on the matrix (the *spectrum metrics*), which count the number of passing and failing test cases that do or do not execute the code element in question, respectively [3], [14]. The formulas themselves use these numbers to rank the code elements according to their suspiciousness [15], [16], [17], [18], [19], [20], [21].

Researchers experimented with various techniques to come up with new formulas, such as combining existing ones [22], applying genetic programming [23], [24], or using systematic search [25] to automatically infer new ones. Xie et al. [26] examined the equivalence and hierarchy between a number of formulas, while Yoo et al. [24] showed that there does not exist a perfect scoring formula that outperforms known techniques found by humans or even by automatic search-based methods.

Yet, it seems that we are still struggling to find an SBFL technique using the coverage spectrum that produces good enough results in practical situations. The average ranking positions relative to the program size in various popular benchmarks are 4%–20.6% for Defects4J [4], around 1.2%–21.1% for SIR [27], and 0.5%–13.5% for BugsJS [28]. Although these are seemingly good results, their absolute values can be unacceptably bad (around 88–1978 for Defects4J, 1–35 for SIR, and 2–81 for BugsJS, on average). Recent user studies report that developers tend to investigate only the top 5 or at most the top 10 elements in the recommendation list provided by localization methods before giving up [29], [30]. Hence, any improved rank position which is beyond these thresholds will probably be less useful, no matter how much relative improvement they can achieve. Not to mention the application of SBFL for *automated program repair*, where the top positions are implicitly expected [31], [32].

So, it seems that we reached the limit when it comes to coverage-based SBFL approaches, and any additional variation of the technique could only bring modest improvement over the state-of-the-art. One possibility is to include information external to the spectrum into the process and thus aid the localization process [33], [34], [35], but these are beyond the scope of this paper. We instead concentrate on the pure spectrum-based approaches, and in particular, we would like to find out **how suitable the prevalent coverage-based spectra are for the task in the first place.**

An important insight about coverage-based spectra is that these techniques are based on the assumption that a code element covered by failing tests should be treated as suspicious. However, it is easy to see that this is an over-approximation of

the original intent to look for the code *responsible for the fault*. The faulty code element must be executed in a failing run, but it also needs to cause a failure-inducing chain toward the output [36]. If a statement is executed but it is not participating in the corresponding computation responsible for the failure, it causes noise in the SBFL process.

In other words, instead of the simple coverage information, only its subset should be used, which takes part in the computation. This is, precisely, the concept of the *program slice* [37]. In particular, we are interested in the *backward dynamic program slice* [38] computed from the output statement as the criterion, and use this information in the program spectra. If computed properly, a dynamic program slice will be a subset of the coverage information and in the spectrum, it will provide exactly the information that is needed by SBFL formulas. A further advantage of program slicing is that most practical approaches are able to provide structural information about the computation path as well, not just the program subset.

The question then is: how big is the influence of the over-approximation caused by the coverage-based spectrum compared to the slice-based? This depends on the relative size of the slices with respect to the coverage information, and the way superfluous elements affect the SBFL algorithm. In other words, what is the effect of the executed code elements that are not in the slice on the final ranking lists?

The idea of combining program slices and SBFL is not new, and there are various approaches to do the same (overviewed later in this paper). Surprisingly, relatively few studies among these utilize backward dynamic slices in place of the coverage in the program spectrum [39], [40], [41]. Also, these studies do not elaborate on the relationship of the coverage-based and slice-based spectra, typically only high-level measurement results are provided. The main reason for this modest visibility could be very pragmatic: computing precise slices requires difficult algorithms, and the computation costs can be very high compared to simply using the coverage.

With this paper, we aim at filling this gap and providing more insight into "how bad is the coverage spectrum?" when compared to the slice-based spectrum, and what are the typical situations where the deficiencies manifest. Given the fact that dynamic slices can be quite small (about 33% [42] to 50% [43] of the executed instructions on average), we expect a large impact on the overall algorithm effectiveness.

It is not the aim of this paper to discuss concrete slicing techniques and their effect on SBFL, just the conceptual relationship backed up by an empirical case study to illustrate the differences. The contributions of this paper are the following:

1) We provide a theoretical analysis of why coverage-based spectra necessarily produce suboptimal results compared to dynamic slice-based spectra.
2) We implemented a dynamic slice-based SBFL method using a precise yet feasible approach.
3) In a case study using a well-known subject program and real faults, we found that slice-based spectra outperformed traditional coverage-based spectra by a large margin in terms of the faulty element's rank position.
4) We thoroughly analyzed every fault in the case study to understand the most typical causes of suboptimal performance of the coverage-based approach.

II. Dynamic Slice as Program Spectra

In this paper, we will use the following notations and representations of SBFL and slicing concepts. Note that different articles may use different representations and notations of the same concepts. This section is also intended to introduce the concept of dynamic slice as program spectra and related implementations.

A. Background on Spectrum-based Fault Localization

Let P denote the program under investigation, T the set of test cases that test P, and E the set of code elements in P according to the chosen granularity level.

In the SBFL approach, the dynamic information from running test cases is contained in the *program spectrum*, which consists of two parts, the *spectrum matrix* M of size $|T| \times |E|$ and the *results vector* R of size $|T|$. Columns of the spectrum matrix represent elements of E while the rows contain elements of T. The basic form of the spectrum matrix is the *hit-based* matrix in which each cell can be either 1 or 0 denoting if there is a dynamic relationship between an element $e \in E$ and $t \in T$ upon its execution or not. Thus, in the *coverage-based* spectrum matrix, $m_{i,j} = 1$ if the i-th test covers the j-th element and 0 otherwise. In the rest of the paper, we will use M to denote the hit-based coverage spectrum matrix, and $C(t) \subseteq E$ for the set corresponding to the t-th row, i.e., the set of covered elements by test case t.

Elements of the results vector R are defined as $r_i = 0$ if the i-th test was completed without failure and 1 otherwise. In addition, to evaluate the fault localization effectiveness, information about the known faults will be used from benchmark programs. It will be represented by the *faults vector* F of size $|E|$ in which $f_j = 1$ if the j-th code element contains a fault. For simplicity, we will also use the same notations M, R, and F to represent not only the matrix/vectors but the corresponding sets and functions as well, depending on the context.

The next step in the fault localization process is calculating the four *spectrum metrics* on the matrix, which count the number of passing and failing test cases that do or do not include the code element e in question, in various combinations. The following four sets provide the basis for these numbers:

$$\begin{aligned} ef(e) &= \{t \in T \mid M(t,e) = 1 \wedge R(t) = 1\} \\ nf(e) &= \{t \in T \mid M(t,e) = 0 \wedge R(t) = 1\} \\ ep(e) &= \{t \in T \mid M(t,e) = 1 \wedge R(t) = 0\} \\ np(e) &= \{t \in T \mid M(t,e) = 0 \wedge R(t) = 0\} \end{aligned}$$

For the sake of simplicity, we will use the notations ef, nf, ep, and np to denote the sizes of these sets, respectively.

Suspiciousness formulas use some or all of these values to calculate the score for each code element in the program,

giving their final ranking as the output to the user (by convention, a bigger score means more suspicion). As mentioned, the literature has provided a large number of formulas. In this paper, we will rely on several well-known ones, that are defined in Table I.

TABLE I
DETAILS OF THE SBFL FORMULAS USED IN OUR EXPERIMENT
Barinel- Bar, *DStar- Dst*, *Jaccard- Jac*, *Ochiai- Och*, *Sørensen-Dice- Sor* AND *Tarantula- Tar*

$$Bar\ [44]: \frac{ef}{ef+ep} \qquad Dst\ [20]: \frac{ef^2}{ep+nf}$$

$$Jac\ [16]: \frac{ef}{ef+nf+ep} \qquad Och\ [16]: \frac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$$

$$Sor\ [21]: \frac{2\cdot ef}{2\cdot ef+nf+ep} \qquad Tar\ [45]: \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$$

B. Background on Dynamic Program Slicing

Program slicing [46] is a classical code analysis technique, which aims to determine a subset of a program P, called the *program slice*, by omitting the irrelevant code elements, such as statements, with respect to a specific calculation and from a specific perspective. Every slice is related to a tuple named the *slicing criterion*, which can be written as $Cr = \langle P, v, l\rangle$ where v is a variable at program position l for which the slice needs to be computed. Generally speaking, a slice of program P with respect to a slicing criterion Cr includes only computations that are related to variable v at program point l. Slicing approaches can be categorized by the direction of computation, i.e., *forward* or *backward*, depending on whether the result contains those lines which depend on the value of the criterion variable, or affect it, respectively. The other classification of slicing approaches is according to whether the slice includes information computed for all possible executions of the program (*static slicing*), or only for a specific program input, i.e., a test case (*dynamic slicing*).

This paper deals with backward dynamic program slicing with the slicing criterion being the "output statement" of the test case. In practice, the output statement may correspond to an assertion point in the test case with the asserted variable. In the following chapters, we will use $DS(t) \subseteq E$ to denote the backward dynamic program slice corresponding to test case t.

C. Dynamic Slices as Program Spectra

In fault localization literature, several works explore the possibilities of combining the spectrum-based approach with program slicing. Various strategies are possible to this end, and the most important ones are overviewed in Section VI.

In this work, we concentrate on the class of methods in which the traditional SBFL method based on code coverage is modified by replacing the spectrum matrix with slice information. Several variations are possible at this point – which we will discuss later in this section –, but for illustration purposes, we define the *slice-based spectrum* as the spectrum matrix M' whose **rows include the backward dynamic slices computed from the corresponding test cases instead of their coverages** (each $C(t)$ is replaced by $DS(t)$).

```
1  public class Circle {
2     private double area;
3     private double perimeter;
4     public Circle(double radius) {
5        area = radius * radius * Math.PI;
6        perimeter = radius * Math.PI; // faulty statement
7     }
8     double getArea() {
9        return area;
10    }
11    double getPerimeter() {
12       return perimeter;
13    }
14 }
```

Listing 1. Faulty code example

```
1  public class CircleTest extends TestCase {
2     static Circle circle = new Circle(0);
3     public void t1() {
4        assertEquals(Math.PI, new Circle(1).getArea(),
              1e-10);
5     }
6     public void t2() {
7        assertEquals(2.0*Math.PI, new
              Circle(1).getPerimeter(), 1e-10);
8     }
9     public void t3() {
10       assertEquals(0, circle.getPerimeter(), 1e-10);
11    }
12 }
```

Listing 2. Tests for the faulty code

The benefit of a slice-based SBFL over a coverage-based one can be easily illustrated in a simple example. Listing 1 includes a Java snippet of a circle implementation with methods for calculating the area and perimeter, along with three associated unit tests in Listing 2. The execution of the tests results in `t2()` failing due to the bug in line 6 for calculating the perimeter, and the other two passing. The coverage-based spectrum, along with the spectrum metrics, and the suspiciousness scores computed by the Barinel formula are shown in Table II.

TABLE II
COVERAGE-BASED SPECTRUM AND FAULT LOCALIZATION RESULT

	4	5	6	8	9	11	12	R
$C(t1)$	1	1	1	1	1	0	0	0
$C(t2)$	1	1	1	0	0	1	1	1
$C(t3)$	0	0	0	0	0	1	1	0
ef	1	1	1	0	0	1	1	
ep	1	1	1	1	1	1	1	
nf	0	0	0	1	1	0	0	
np	1	1	1	1	1	1	1	
Bar	0.5	0.5	0.5	0.0	0.0	0.5	0.5	

The SBFL formula cannot distinguish between code lines 5 and 6 as the simple coverage is over-approximating the

actual calculation chains: both constructor lines are included in all passing and failing tests. The slice-based spectrum differs from the coverage one exactly at these two critical lines. The backward dynamic slices computed from the test cases correctly include only the appropriate lines setting the `area` or `perimeter` fields, respectively (see Table III). The result is that the faulty line 6 is now correctly localized at the first ranking position with the score 1 ($ef = 1$ and $ep = 0$), while the non-faulty line 5 gets a score 0 with $ef = 0$ and $ep = 1$).

TABLE III
SLICE-BASED SPECTRUM AND FAULT LOCALIZATION RESULT

	4	5	6	8	9	11	12	R
$DS(t1)$	1	1	0	1	1	0	0	0
$DS(t2)$	1	0	1	0	0	1	1	1
$DS(t3)$	0	0	0	0	0	1	1	0
ef	1	0	1	0	0	1	1	
ep	1	1	0	1	1	1	1	
nf	0	1	0	1	1	0	0	
np	1	1	2	1	1	1	1	
Bar	0.5	0.0	1.0	0.0	0.0	0.5	0.5	

As mentioned, there are only a handful of researchers who utilized this concept to develop a combined SBFL and slicing approach. Mao *et al.* [40] presented an approach that used slices to construct more precise program spectra, and they experimented with various slicing algorithms including Approximate Dynamic Backward Slicing and Relevant Slicing. The algorithm called *Tandem-FL* was proposed by Reis *et al.* [39], which is able to locate and reduce the suspiciousness of those components that are mostly involved in failing tests but seldom covered by passing ones. Their idea uses SBFL to calculate the suspiciousness of each element, then select the top k from the list. Thus there is no need to slice the whole program, which is great because slicing is expensive. Alves *et al.* [41] use the basic slice-based approach but introduce variations to accommodate for change-based analysis.

None of these reports deal with analyzing the differences between the coverage and the slice information nor seek to understand how much the former one is over-approximated. Furthermore, the other related research we are aware of (see Section VI) combines the two techniques differently.

III. COVERAGE VS. SLICE SPECTRA

The purpose of this section is to analyze in more detail the relationship between the coverage-based and slice-based spectra and the influence of this difference on the fault localization effectiveness. We do this first on a theoretical level in order to assess the expected effects in actual implementations. Then, we summarize our findings and motivate the remaining parts of the study with a set of research questions.

A. Theoretical Analysis of Coverage and Slice

We will use notations M', ef', nf', ep', and np' to denote the slice-based spectrum matrix, and the associated spectrum metrics. For this theoretical analysis, we assume the following:

1) Program P includes exactly one fault which can be identified at a single code location, i.e., $|F| = 1$. The faulty code element will be denoted by f in the following (n will be used for all other elements).
2) All coverages include at least one code element, i.e., $\forall t \in T : |C(t)| > 0$
3) The faulty code element is executed by all failing test cases, i.e., $\forall t \in T : R(t) = 1 \Rightarrow M(t, f) = 1$
4) Each test case t can be associated with exactly one slicing criterion, which means that rows of matrices M and M' are compatible.
5) The backward dynamic slice is computed correctly, meaning $\forall t \in T : DS(t) \subseteq C(t) \subseteq E$, furthermore
6) f contributes to the slicing criterion in all failing test cases, implying that
7) f is included in all slices for failing test cases, i.e., $\forall t \in T : R(t) = 1 \Rightarrow M'(t, f) = 1$.
8) All slices include at least one code element, i.e., $\forall t \in T : |DS(t)| > 0$.

Following the dynamic slice's property of being the subset of the code coverage, we can look at how much more precise it is. We can express this in terms of the slice size with respect to the coverage size for each test case t in a program P. The average slice size will be used as a proxy to the probability $p \in (0, 1]$ that a covered code element e will be also in the slice:

$$p = \frac{\sum_{t \in T} \frac{|DS(t)|}{|C(t)|}}{|T|}$$

Based on the assumptions above, we can make the following observations. For the faulty element f, $ef'(f) = ef(f)$ and $nf'(f) = nf(f)$ because each failing test's slice must include f and both M and M' have the same T set of tests. Regarding the passing tests, there are no such requirements, so $ep'(f) \subseteq ep(f)$ and $np'(f) \supseteq np(f)$. In the case of any other non-faulty element n, the subset relationship will be the same for all four sets. We can then calculate the expected values of the four spectrum metrics for the slice-based spectrum as follows.

For f:	For any n:
$ef' = ef$	$ef' = p \cdot ef$
$nf' = nf$	$nf' = (1 - p) \cdot ef + nf$
$ep' = p \cdot ep$	$ep' = p \cdot ep$
$np' = (1 - p) \cdot ep + np$	$np' = (1 - p) \cdot ep + np$

We can now investigate how values of the suspiciousness formulas relate to each other for the same program P but computed on M and M'. For the faulty element, the value of ef does not change while the others get smaller, and since many of the formulas include ef in the numerator and the others in some form in the denominator, the score value will usually be bigger in the slice-based spectrum. For the non-faulty element, the score will typically either be the same or

smaller.[1] Let us look at the formulas we work with in this paper in more detail (the slice-based spectrum metrics are simply substituted to form the slice-based formulas):

$$\begin{aligned} \mathit{Bar}'(f) &= \frac{ef}{ef + p \cdot ep} &\geq \quad \mathit{Bar}(f) \\ \mathit{Bar}'(n) &= \frac{p \cdot ef}{p \cdot ef + p \cdot ep} &= \quad \mathit{Bar}(n) \end{aligned}$$

$$\begin{aligned} \mathit{Tar}'(f) &= \frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf} + \frac{p \cdot ep}{ep+np}} &\geq \quad \mathit{Tar}(f) \\ \mathit{Tar}'(n) &= \frac{\frac{p \cdot ef}{ef+nf}}{\frac{p \cdot ef}{ef+nf} + \frac{p \cdot ep}{ep+np}} &= \quad \mathit{Tar}(n) \end{aligned}$$

$$\begin{aligned} \mathit{Och}'(f) &= \frac{ef}{\sqrt{(ef + nf) \cdot (ef + p \cdot ep)}} &\geq \quad \mathit{Och}(f) \\ \mathit{Och}'(n) &= \frac{p \cdot ef}{\sqrt{(ef + nf) \cdot (p \cdot ef + p \cdot ep)}} \\ &= \sqrt{p} \cdot \mathit{Och}(n) &\leq \quad \mathit{Och}(n) \end{aligned}$$

$$\begin{aligned} \mathit{Jac}'(f) &= \frac{ef}{ef + nf + p \cdot ep} &\geq \quad \mathit{Jac}(f) \\ \mathit{Jac}'(n) &= \frac{p \cdot ef}{p \cdot ef + (1-p) \cdot ef + nf + p \cdot ep} \\ &= \frac{ef}{\frac{ef}{p} + \frac{nf}{p} + ep} &\leq \quad \mathit{Jac}(n) \end{aligned}$$

$$\begin{aligned} \mathit{Sor}'(f) &= \frac{2 \cdot ef}{2 \cdot ef + nf + p \cdot ep} &\geq \quad \mathit{Sor}(f) \\ \mathit{Sor}'(n) &= \frac{2p \cdot ef}{2p \cdot ef + (1-p) \cdot ef + nf + p \cdot ep} \\ &= \frac{2 \cdot ef}{\frac{p+1}{p} ef + \frac{nf}{p} + ep} &\leq \quad \mathit{Sor}(n) \end{aligned}$$

$$\begin{aligned} \mathit{Dst}'(f) &= \frac{ef^2}{p \cdot ep + nf} &\geq \quad \mathit{Dst}(f) \\ \mathit{Dst}'(n) &= \frac{p^2 \cdot ef^2}{p \cdot ep + (1-p) \cdot ef + nf} \\ &= \frac{ef^2}{\frac{ep}{p} + \frac{nf}{p^2} + \frac{1-p}{p^2} ef} &\leq \quad \mathit{Dst}(n) \end{aligned}$$

We could show that, in all investigated cases, the suspiciousness score of the faulty element is the same or bigger in the slice-based spectrum than for the coverage-based one, while all non-faulty elements' scores are either smaller or the same for the slice-based spectrum. We checked several other published formulas if they exhibit this property and found that they do, but we cannot rule out the possibility that there are some counter-examples. However, we expect that any formula that has a meaningful combination of the spectrum metrics will behave similarly.

We can also observe from the above that the smaller p is the bigger the difference will be between the two methods. The effect is that, with these assumptions, **the coverage-based SBFL necessarily produces a worse ranking than the slice-based SBFL, due to the over-approximation of the real dynamic dependencies using the coverage, and the bigger this over-approximation the worse the result will be**.

B. The Need for More Research

The assumptions from above will not necessarily hold for realistic situations, but we believe that they are good approximations of reality. Furthermore, the benchmarks typically used in related research often aim at achieving these ideal situations. Slicing tools are not perfect either, and they may violate one or more of the assumptions. Nevertheless, we believe that further research is necessary to understand what the performance of realistic implementations and real programs and faults is. Also, the imprecision of coverage-based SBFL with respect to slice-based SBFL should be measured in practice by looking at the slice sizes (p) since this turns out to be an essential parameter.

In the remaining parts of the paper, we present the case study that we performed with the goal to verify the above concepts in practice and serve as a motivation for further research. In particular, we seek to answer the following **Research Questions**:

RQ1 *How do the coverage-based and slice-based spectra compare to each other in practice?*
With this question, we seek to find out what the typical slice sizes are compared to coverage size, and whether the parameter p is really small enough to warrant the suboptimal performance of traditional SBFL.

RQ2 *What is the overall performance of slice-based SBFL to the coverage-based one in terms of ranking effectiveness?*
A typical assessment of SBFL is the average rank position of the faulty element, which should be as close to the 1st place as possible.

RQ3 *What are the typical explanations of the differences between the two types of SBFL results?*
This is a qualitative evaluation of the findings. Since we are working with a small number of concrete examples in this case study, we are able to manually assess each case and find out the actual reasons for (both positive and negative) differences between the slice-based and coverage-based SBFL.

IV. Case Study

The goal of our case study was to verify the relationship between the coverage-based and slice-based spectra in depth. Instead of performing an extensive empirical evaluation involving multiple programs and bugs and reporting overall

[1] It is worth noting that this shows the average case and the expected values based on the average slice size, but in reality, the final scores for the individual elements can change in either direction because the individual n elements can have various slice ratios for passing and failing cases.

high-level results, we selected one subject program from a benchmark suite and evaluated each fault separately in detail. Our goal with the case study was to implement the basic method outlined in Section II as closely as possible, i.e., we did not want to use approximate slicing algorithms or other optimizations on the matrix. There were several difficulties, however, including the imperfection of the slicer tool we selected, and the way test cases, slicing criteria, and code elements could be matched, as discussed in the rest of the paper.

A. Creating Slice-Based Spectra

Coverage-based program spectra might include statements that do not affect the tested value. These additional but irrelevant statements can lower the effectiveness of the method. The solution already proposed by previous works is to compute the spectrum from slices [40]. We use this kind of "test-slice" spectrum in our evaluation.

In an ideal case, a test should check only one value, and there are test environments where this is ensured. For example, sometimes the output of the program under test is written to the standard output or a file (using a single statement) for later comparison with a reference output. However, in practice, especially in unit test frameworks, a single test usually checks multiple values. In unit tests, this is implemented as multiple assertions in a single test case, and from the execution logs of a test case, it can be determined which assertion has failed.

Decomposing the tests and creating the spectrum for assertions instead of tests might produce more detailed information on the position of the fault. In other words, we create one row to the slice-based spectrum matrix *for each assert* rather than for each test case. To be able to compare the slice-based results to the traditional coverage-based ones, we then merge assert-slices for each test case by calculating the union of assertion slices per test case.

B. Slicing Tool

During the preparation of our experiments, we tested several tools that are capable of creating dynamic backward slices, Slicer4J [47], [48], JavaSlicer [49], and Java SDG Slicer [50] amongst others. However, most of the publicly available tools are not well-maintained, and they have deprecated or unavailable dependencies. Finally, we decided to use the open-source dynamic slicing tool Slicer4J to collect the slices. It uses low-overhead instrumentation to collect a runtime execution trace; it then constructs a thread-aware, inter-procedural dynamic control-flow graph, and a set of pre-constructed data-flow summaries to compute the slice.

C. Determining the Slicing Criteria

Determining the slicing criterion is relatively straightforward in the case of unit tests. We have asserts in the test cases that check some actual computed values against some expected values. We should simply slice for the actual values used in the assert statements. As the slicer we used is able to slice for a source code line (i.e., it is enough to give a line number as a slicing criterion and it will compute slices for all appropriate variables of that line), we first simply selected those lines of the test cases that contained asserts and passed these lines to the slicer. At the same time, we had to employ some workarounds to specific exception-handling constructs found in the subject program.

D. Spectrum Matrices and Fault Localization

Since we used the traditional coverage-based SBFL approaches as the baseline of our evaluation, we had to calculate the corresponding results as well. To extract the coverage-based program spectra and calculate the results based on them, we used the approach published by Pearson *et al.* [4].

The slicer and coverage tools identified the tests and instructions in a mostly similar but slightly different way. The basis of the instruction ID we used is the fully qualified name of the Java class and the line number. For the test cases, we used the fully qualified Java method name (without return value and parameter specification). The asserts were also identified by test case name and an additional absolute line number of the assert in the file. While the two dimensions of our proposed spectra are asserts and instructions, in the case study, we aggregated our slices by test cases. Thus, we computed the slices of all asserts of a given test case and assigned their union to the test case as its slice set. We used our own scripts to calculate the slice-based spectrum matrix from the raw data produced by the slicer, the spectrum metrics, and the ranks.

E. Subject Program

In this paper, we focus on the qualitative evaluation of the differences among traditional coverage-based and slice-based program spectra, hence we used a subset of the bugs of the program Time from the Defects4J (v2.0.0) [51] benchmark. We chose Time as our subject because its domain is fairly easy to understand and this gives us the opportunity to demonstrate the effects of the different approaches more easily. In addition, the complexity of the program, the tests, and the faults is medium and could be regarded as typical for this benchmark. There are 26 bugs (program versions) in this benchmark item with 12.9k-14.1k executable statements and 3.7k-4.0k tests depending on the version. There are 1-8 faulty statements in each version.

V. Results and Evaluation

A. Data Preparation

TABLE IV
Properties of the Investigated Bugs

Inc.	Reason	Bugs
✗	*Bad Slice*	{1, 2, 7, 8, 11, 13, 19, 20}
	Omission	{3, 6, 14, 15, 18, 24, 25, 27}
	Exception	{5}
✔	-	{4, 9, 10, 12, 16, 17, 22, 23, 26}

The subject program has 27 buggy versions, but bug 21 is marked as deprecated, resulting in 26 bugs we could

work with. Table IV shows whether the bug was included or excluded in our examination (column "Inc."), as well as, the reason behind the exclusion (column "Reason"), which is described in detail below.

We excluded 8 bugs because their fix contains only added statements (*Omission*). These statements are missing from the buggy versions, i.e., neither the coverage-based nor the slice-based spectra can point to them. Another reason was that the slice did not contain the faulty element, but the test cases covered it (*Bad Slice*). In one case, the test failed before the assertions due to an exception (*Exception*), which made the dynamic slice computation impossible. In 7 cases, we could not find out why the slice did not contain the faulty (and covered) statements. In two cases the faulty statements were spread across multiple lines, and the reported location of the fault (determined from change sets) did not match the location reported by the slicer (the first line of the multiline statements). We corrected these two computations by hand. As a result, we had 9 bugs for which we could compute meaningful slices.

B. Comparing Spectra

As discussed above, in theory, the slice spectra should be a subset of coverage spectra. Unfortunately, with our toolset, this turned out to not hold in about 60% of the slices. We checked the reasons why slices can be inaccurate in this sense, and we found two main reasons. One is that the slicer does not slice into Java library methods, and while losing dependencies through them it also follows some false dependencies. The other cause is that the slicer and the coverage tool can report different source code lines for multiline expressions.

To approximate the theoretical parameter p from Section III on our subject program, we had to deal with this inaccuracy of the slicing tool. We simply ignored statements that are not covered but are part of the slice, hence we computed the size of the intersection of the coverage and slice sets and divided it with the coverage size. We then averaged this value for every test case. Table V shows the resulting values along with some other statistics.

TABLE V
MEASURED AVERAGE SLICE SIZES WITH RESPECT TO THE COVERAGE ON THE SUBJECT PROGRAMS

Bug	Avg.	Med.	Min.	Max.	Std.dev.
4	0.4259	0.4167	0.0	1.0	0.2834
9	0.4321	0.4286	0.0	1.0	0.2847
10	0.4321	0.4286	0.0	1.0	0.2849
12	0.4335	0.4344	0.0	1.0	0.2836
16	0.4369	0.4378	0.0	1.0	0.2828
17	0.4345	0.4357	0.0	1.0	0.2811
22	0.4444	0.4463	0.0	1.0	0.2796
23	0.4442	0.4463	0.0	1.0	0.2798
26	0.4634	0.4706	0.0	1.0	0.2750

Answer to RQ1: The average slice size varies between 42.5% and 46.4% of the coverage. However, we expect that the correct values would be even lower because the slicer seems to be over-approximating as was the case with the not covered elements. Looking at the relationship between the expected fault localization scores for the two types of matrices in Section III, we see this value to be a significant factor responsible for the differences in the final rankings.

C. Comparison of SBFL Effectiveness

There are different ways to compare the effectiveness of SBFL algorithms. One of the most popular methods is to compare the *absolute average rank* [16] of faulty elements. This metric represents the number of statements the developers have to investigate before finding the first faulty element. Ties are dealt with by assigning the average rank for each element sharing the same score. If there are multiple faulty statements for a bug, we use the rank of the first buggy element.

Table VI shows the results in terms of absolute average rank for different formulas on the included bugs for both kinds of spectra. We can see that in most cases the slice-based results are better, sometimes notably. There were a couple of cases where the results were the same, and in two cases (bugs 9 and 16) the coverage-based method performed better. For bugs 4 and 17, the faulty element was placed on the highest rank position by the slice-based approach but since it was tied with several other elements, the value shown is not 1.

The last row of Table VI represents the overall average ranks, from which we can infer the degree of improvement in general. The difference is notable in all cases (between 10 and 32), i.e., the slice-based method ranks the faulty statement higher in the suspiciousness list by 10-32 positions. Overall, *DStar* performed best (9.5), followed by *Ochiai* (10.1) but the other formulas have very similar results as well.

In two cases, the slice-based spectra produced worse results than the coverage-based because the slicer could not slice into Java library methods and thus it followed false dependencies.

Answer to RQ2: Although we cannot draw definitive conclusions from the results due to the small sample size, we can say that the overall performance of slice-based SBFL compared to coverage-based one is positive: on this subject program it improved the ranking position of the faulty elements notably, in many cases achieving the top positions. Only two bugs showed negative results, and we attribute these to inaccuracies in the slicing tool.

D. Qualitative Evaluation

We examined each investigated bug in detail to find out why coverage-based SBFL produced sub-optimal results compared to slice-based SBFL (in the cases when the result was negative, the reasons for it as well). The focus of the comparison was primarily on the spectrum metrics, rather than the score and rank values. (In the following section, the item names are relative to the *org.joda.time* package.)

time-4: Test *TestPartial_Basics.textWith3* fails here because no exception is thrown. The reason for this is that the *Partial.with(DateTimeFieldType, int)* method calls a wrong constructor (in line 464). The slice of the test case contains 4 statements (lines 430, 431, 464, 466), while the coverage has 24 additional ones. In addition, 3 utility statements are covered by only the faulty test case. This resulted in the score of

TABLE VI
RANKS OF FAULTY STATEMENTS

Bug	*Bar*		*Dst*		*Jac*		*Och*		*Sor*		*Tar*	
	cov.	slice	cov.	slice	cov.	slice	cov.	slice	cov.	slice	cov.	slice
4	23.5	1.5	20.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5	23.5	1.5
9	3	11	2	11	3	11	3	11	3	11	3	11
10	21.5	9	11.5	10	19.5	10	15.5	10	19.5	10	21	9
12	2.5	2.5	29.5	2.5	8	2.5	5	2.5	8	2.5	2.5	2.5
16	8	10	8	11	8	11	8	11	8	11	8	10
17	5	5	5	5	5	5	5	5	5	5	5	5
22	23.5	14.5	23.5	13.5	23.5	14.5	23.5	13.5	23.5	14.5	23.5	14.5
23	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5	25.5	20.5
26	270.5	20.5	60.5	10.5	132.5	19.5	71.5	15.5	132.5	19.5	156.5	20.5
avg	42.6	10.5	20.7	9.5	27.6	10.6	20.1	10.1	27.6	10.6	29.8	10.5

the faulty statement ranking at 23.5 on average, together with 14 other statements. As the mentioned utility and additional statements are omitted by the slicer, their ef' values were reduced, allowing the formulas to rank lines 464 and 466 in the first position with the same score.

time-9: Here *DateTimeZone:264* has a slice-based rank 11 with ef'=1 and ep'=5, and shares these values with 8 statements including *DateTimeZone:604*. However, *DateTimeZone:604* was covered by not 5 but ep=11 passed tests. For example, the slice of *TestDateTimeZone.testSerialization2* does not include the above-mentioned statement but covers it.

The computed slice of the test contains only the statements of the test except for the 1011^{th}, 1006^{th}, and 1000^{th} instructions. As `oos` is of a stock Java class type, the slicer does not analyze its method call in line 1004 but seems to treat it as a definition of a `zone` instead of treating it as a use. This can be the reason why line 1000 is (incorrectly) not included in the slice, so the ep' values of the statements accessible through it (e.g. *DateTimeZone:604*) are not increased by the test result of *testSerialization2*, thus, the scores are not decreased, i.e. statements are ranked as more suspicious. Due to cases like this, the results of the (passed) tests will not be counted for certain statements and, therefore, it is possible that the coverage-based result will be better than the slice-based one.

time-10: The second assert fails in both failing test cases. While the bug is covered by both test cases, it is contained only in the slice of the first, not failing assert of test case *TestDays.testFactory_daysBetween_RPartial_MonthDay*. We could not find the reasons for this omission, it is probably due to a slicer issue.

time-12: We found that the slices of the asserts contain only a few statements, so there is a non-negligible difference between the slice-based and coverage-based spectrum metrics. The reason for the difference is that statement *LocalDateTime:612* (in the *isSupported()* method) has ef=4, which makes it among the most suspicious elements according to the coverage-based algorithm, while the ef'=0, which puts it in the bottom of the suspicion ranking of the slice-based approach. The ef, ep, and nf values of the faulty statement *LocalDate:211* are the same in the two spectra, so it precedes several statements in the ranking that are more suspicious than it according to the coverage-based algorithm but ranked lower by their slice-based spectra.

time-16: We examined the tests related to the faulty statement *format.DateTimeFormatter:709* in method *parseInto()*. In the case of the coverage-based measurement, test *TestDateTimeFormatter.testParseInto_monthOnly* covers the buggy statement, however, the slice-set belonging to the assert in line 869 includes only the first line of the *parseInto()* method (line 698). This is interesting because the return value of *f.parseInto(result, "5", 0)* function call (and the other statements affecting it) was omitted due to a probable slicer problem and this misled the slice-based FL algorithms.

time-17: Bug 17 has 9 instructions with the same highest score and average rank of 5. These 9 instructions belong to 3 methods, 2 of which (*DateTime.withEarlierOffsetAtOverlap()* and *DateTime.withLaterOffsetAtOverlap()*) are simple sequential methods, and *DateTimeZone.adjustOffset(long, boolean)* is a bit longer having a decision. All instructions but the alternative return of the last method are both covered and part of the slices. These 9 instructions are always executed together. As a result, their two spectra are identical, resulting in the same scores and ranks.

time-22: 8 passed tests have incorrectly computed slices, e.g., the slice of the assert in line *TestMutablePeriod_Basics:451* contains only 1 statement, and does not include the constructor of class *MutablePeriod* and thus (incorrectly) could not reach the buggy line *base.BasePeriod:222*. Yet, the scores and ranks of the faulty instruction improve, because its $ep' < ep$ due to the bad computations.

time-23: Test *TestDateTimeZone.testForID_String_old* fails when it checks the contents of a map previously filled by the *DateTimeZone.getConvertedID(String)* method. The coverage-based calculations rank *DateTimeZone:314* to first place as it is executed by the failing and a single passing test case. Then a tie with 48 elements follows, including the faulty lines, lines filling the map, and other lines of the *DateTimeZone.getDefault()* method. All of these are covered by the sole failing and multiple passing tests. The slicer is unable to decompose which items in the map are used in the failing test, keeping the whole map filling section in the slice. However, it is able to omit the lines of the *getConvertedID(String)* method from all test cases except for the passing one (and it does the same for two additional instructions of *getDefault()*), while

keeping them in the slice of the failing test case. This reduction of ep' results in a tie of 40 elements, including all faulty lines, in the first place (with an average rank of 20.5).

time-26: There are 8 failing test cases and 8 modified lines for fixing the same bug. The fix replaces the call to *convertLocalToUTC(long, boolean)* with the newly added *convertLocalToUTC(long, boolean, long)*. However, only one buggy line, *chrono.ZonedChronology:467* is exercised by the tests. It is covered by all faulty test cases but contained only in 4 of their slices, while 175 passing tests also cover the line but only 49 slices of passing tests do the same. Thus, while ef is higher than ef', ep is much higher than ep', causing our metrics to give a higher score to the faulty instruction. The slicer also eliminates many instructions from the spectrum of faulty test cases. While coverage shows 1615 instructions with non-zero ef, there are only 872 instructions with non-zero ef'. This also helps improve the rank of the faulty statement.

Answer to RQ3: The cases when slice-based SBFL did not overtake coverage-based SBFL were due to imperfections or defects in the slicer. In one case (Bug 10) the slicer seemed to miscalculate slices for the faulty test, yet the ranks were still able to improve. In all other cases, the slice-based spectrum worked as expected, and either raised the score of the faulty element or lowered the score of non-faulty statements.

E. Discussion

1) Main Concern: We argue with this paper that using code coverage in the SBFL spectrum is such an over-approximation that it could impair the achievable effectiveness of SBFL to a level that makes it not useful in practice. Code coverage is, in essence, a proxy to the dynamic backward program slice which captures the code elements with an actual influence on the defective behavior. In Section III, we showed that, in principle, the coverage-based SBFL will necessarily produce worse code element ranking compared to the slice-based spectrum because a correctly computed backward dynamic slice is a subset of the coverage. Furthermore, it is expected that the rate of imprecision of the coverage will directly and severely influence the performance of the suspiciousness formulas.

2) Theory and Empirical Results: The preconditions set in Section III will not hold in many practical situations, but as our experiment study showed, even under imperfect conditions, the benefits of slicing are clearly visible over coverage. In fact, the case study showed that the slice is less than half of the coverage (RQ1) and that the overall localization effectiveness is typically much better with the slice-based SBFL than with the coverage-based one (RQ2). The study also highlighted several deficiencies in the slicing tool used, but despite of these the results were positive. We cannot rule out the possibility that the coverage-based measurement had flaws too, but since we used a well-established technique and mature tools, we attribute these errors mostly to the slicer tool. Finally, through real examples, the qualitative evaluation (RQ3) showed why the coverage-based spectrum was detrimental. So, the question is, why are we still using code coverage as the basis for SBFL?

3) Difficulty of Program Slicing: One part of the answer is that computing coverage is simple using existing tools, and the SBFL implementations are straightforward. But, despite its several decades-long history, program slicing is still a difficult area, and practically usable tools are not easy to develop. Slicers that produce more precise results often require huge computation resources, while sub-optimal and approximate slicing algorithms may be very imprecise. Another challenge with the slicing approach is that a slicing criterion is needed for each test case, which is often not trivial to determine in practice. Our case study dealt with unit tests, and there the assertions could serve this purpose, but when the tests are higher level or more complex, this can be a challenge.

Furthermore, there are many technical limitations of practical program slicing tools, which increases the risk of their use in fault localization, and this risk is much higher than with the coverage-based tools. If we fail to tell whether an instruction is covered, it will affect only that particular instruction; but when a dependency is missed or falsely added during slicing, it can affect a significant amount of dependent instructions. Difficulties often relate to special features like multi-threading or exception handling, dependencies from 3rd party or mixed language components, or unobservable artifacts e.g., files and databases. Many practical tools either miss dependencies, leading to false results, or employ a conservative approach and imprecise slices, the very essence of the disadvantage of the coverage-based approach discussed in this paper.

4) Additional Benefits of Slicing: On the positive side, it must be noted that program slicing also carries structural information about the program and the computation paths. In a simple SBFL approach, the programmers need to walk down the list of suspicious elements that often do not have any relationship to each other. However, when using program slicing, the dependencies can be followed, which reflects the actual computations, and this information may be a huge aid during debugging [52]. A hybrid approach, such as the one proposed by Soremekun *et al.* [53], could be considered as well. Section VI lists other approaches to take advantage of the additional information in slicing for fault localization and debugging in general. We leave it for a future work to investigate the relation of various slicing approaches to SBFL.

5) Takeaway Message: We are convinced that this area needs more research. From our literature study, we found a surprisingly low number of SBFL implementations that employ slicing. A good alternative would be to experiment with hybrid or approximate slicing algorithms, as some related techniques did [40]. With modern-day computing power, new slicers could replace traditional coverage-based SBFL soon.

VI. Related Work

A. Spectrum-Based Fault Localization

SBFL has a rich literature, including surveys that summarize the actual state of SBFL [1], [2], [5] and evaluate the effectiveness of existing approaches [3], [54], [15]. Many papers have also been published on the application of SBFL [55], [56], [57], [58]. Other than fault localization, SBFL-like approaches

were used in many areas, e.g., flaky test localization [59], and mutation-based fault localization techniques [60], [61]. Others aimed to improve the test suites to increase efficiency by monitoring program executions to generate differential unit tests [62] or to automatically optimize the test suite [63].

B. Program Slicing

Program slicing (PS) is a well-researched code analysis technique. Some of its main application areas are fault localization and debugging [46], [37], [64], [65], [43]. PS is used for many tasks, e.g., aiding debugging processes [66], [67], [68], and optimizing testing [69]. Implementations and applications are available for different languages, operating systems [70], [71], [72], [73], and even for neural networks [74].

C. Combining SBFL and Program Slicing

The most closely related works to the approach presented in this paper are the following. Mao *et al.* [40] showed that with a backward slice-based spectrum, an SBFL algorithm can achieve 14.8%-70.5% cost savings in terms of the number of code elements that need to be examined. The evaluation was carried out on three UNIX utilities (*flex*, *grep*, and *sed*), the *Siemens suite*, and *space*, which are part of the SIR benchmark [75]. Regarding execution cost, compared to traditional SBFL methods, their approach required around 1.07 (in the case of small programs) to 2.56 (in the case of large programs like space, flex, grep, and sed) more time to run.

Reis *et al.* [39] examined a DS-SBFL approach called Tandem-FL. This approach showed promising results on the Defects4J benchmark. The amount of eliminated code was 1.40%-59.30% depending on the subject and the chosen cut-off point. The effectiveness was also good as it was able to find 87.3% of faulty statements on average at n=5 and 91.2% when n=10. This technique also outperformed the traditional SBFL approaches. It captured more faults both at n=5 (on average 86.5% compared to 79.6%) and n=10 (90.4% and 83.5%), and SBFL had better results in only one case out of five. Alves *et al.* [41] use variations of the slice-based approach to accommodate for change-based analysis. They evaluated the impact of omitting unrelated statements on fault localization.

Other approaches to combine SBFL and PS include the following. Wotawa *et al.* [76] combined dynamic slicing with model-based diagnosis. They extracted additional information from the slices for erroneous variables to compute fault probabilities. Hofer *et al.* [77] focused on improving existing fault localization techniques. They combined *slicing-hitting-set-computation* (SHSC) with SBFL in order to create a more fine-grained analysis than SBFL. Parsa *et al.* [78] proposed a technique called Fuzzy-Slice, which computes the full backward dynamic slice of variables used in output statements in several failing and passing executions. Then, different program execution paths are identified and the fault-relevant statements are ranked according to their presence in different clusters.

Soremekun *et al.* [53] used SBFL to enhance the effectiveness of *Dynamic Slicing* (DS) and compared it to the original *automated fault localization* techniques. The idea behind this approach is that programmers should examine only a few top-ranked elements suggested by SBFL and then switch to slicing. They showed that the hybrid approach is able to find 98% of bugs in case of the programmer inspecting at most 20 lines of code, which is far better than statistical fault localization or dynamic slicing. On average, this hybrid FL requires checking 58%-75% of statements compared to the other two techniques.

Shu *et al.* [79] improved the accuracy of SBFL by combining it with *Failed Execution Slices* (FES) i.e., execution slices belonging to failing tests. The main idea is to reduce the number of statements that need to be examined by removing highly suspicious statements from the original ranked list based on an optimal FES. They showed that when the developer checks 10% of the code, the FES-based approach can find more than 65% of the bugs in contrast to the traditional ones' 47%-59%.

Ju *et al.* [80] combined Program Slicing and SBFL by computing *Full Slices* for each failing test case, and *Execution Slices* for the passing ones. The intersection of these two gives the *Hybrid Spectrum Slice* (HSS). They showed that HSS can reduce the average fault-localization cost (percentage of examined code) by 2.98-31.79% compared to SBFL methods.

VII. Conclusions and Future Work

Although not a new idea, using dynamic slices in the SBFL spectrum deserves more focus because the currently prevalent use of simple code coverage can introduce a lot of imprecision. We believe that the literature review, theoretical analysis, and experimental evaluation presented in this paper support this view. Also, there is very little evidence of successfully using program slices in the SBFL spectrum, let alone of practically usable tools of this type. The reason is pragmatic: computing precise slices is very costly compared to the coverage, which is cheap but, as it turns out, not good enough in many cases.

This paper's main message is that code coverage-based SBFL is currently in a research pit due to its inherent approximation, and research on slice-based spectra should once more attain much higher focus, such as research on more efficient approximate or hybrid slicing. One of the open questions to be addressed is to investigate the actual performance of various slicing algorithms and tools, and how they could be improved to better serve this application. On a more conceptual level, one should consider specific aspects of the SBFL process and slicing, such as the issues of multiple faults, ranking ties, the impact of various slice concepts, as well as the question of the slicing criterion in different types of tests.

The interested reader can find more information about the results of this paper, as well as the data and the tools used, on the following website: https://slicefl.github.io

Acknowledgements

The research was supported by the European Union project RRF-2.3.1-21-2022-00004 within the framework of the Artificial Intelligence National Laboratory and by project TKP2021-NVA-09 implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

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

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

[4] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE Press, 2017, pp. 609–620.

[5] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A literature review," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, 2014.

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

[7] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *Proceedings of the 1987 International Workshop on Managing Requirements Knowledge (AFIPS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 1987, p. 539. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/AFIPS.1987.115

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

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE transactions on software engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[10] S. K. Sahoo, J. Criswell, C. Geigle, and V. S. Adve, "Using likely invariants for automated software fault localization," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 2013, pp. 139–152.

[11] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for java," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3586. Springer, 2005, pp. 528–550.

[12] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining behavior graphs for "backtrace" of noncrashing bugs," in *Proceedings of the 2005 SIAM International Conference on Data Mining, SDM 2005, Newport Beach, CA, USA, April 21-23, 2005*. SIAM, 2005, pp. 286–297.

[13] C. Yilmaz, A. Paradkar, and C. Williams, "Time will tell," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 81–90.

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

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

[16] ——, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[17] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

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

[19] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Proceedings of the 2012 International Symposium on Search Based Software Engineering*, ser. Lecture Notes in Computer Science, G. Fraser and J. Teixeira de Souza, Eds., vol. 7515, Springer. Berlin, Heidelberg: Springer, 2012, pp. 244–258. [Online]. Available: https://doi.org/10.1007/978-3-642-33119-0_18

[20] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

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

[22] B. Bagheri, M. Rezaalipour, and M. Vahidi-Asl, "An approach to generate effective fault localization methods for programs," in *International Conference on Fundamentals of Software Engineering*, 2019, pp. 244–259.

[23] A. A. Ajibode, T. Shu, and Z. Ding, "Evolving suspiciousness metrics from hybrid data set for boosting a spectrum based fault localization," *IEEE Access*, vol. 8, pp. 198 451–198 467, 2020.

[24] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 1, pp. 4:1–4:30, 2017.

[25] Q. I. Sarhan, T. Gergely, and Á. Beszédes, "New ranking formulas to improve spectrum based fault localization via systematic search," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2022, pp. 306–309.

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

[27] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 2006, pp. 39–46.

[28] B. Vancsics, A. Szatmári, and Á. Beszédes, "Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 2020, pp. 308–319.

[29] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 267–278.

[30] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, 2016, pp. 165–176.

[31] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[32] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 191–201.

[33] B. Vancsics, F. Horváth, A. Szatmári, and Á. Beszédes, "Fault localization using function call frequencies," *The Journal of Systems and Software*, vol. 193, p. 111429, 2022.

[34] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 2012, pp. 67–76.

[35] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.

[36] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*. ACM, 2002, pp. 1–10.

[37] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[38] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[39] S. Reis, R. Abreu, and M. d'Amorim, "Demystifying the combination of dynamic slicing and spectrum-based fault localization." in *IJCAI*, 2019, pp. 4760–4766.

[40] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[41] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 520–523.

[42] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, pp. 143–160, apr 2007.

[43] D. W. Binkley and M. Harman, "A survey of empirical results on program slicing." *Adv. Comput.*, vol. 62, no. 105178, pp. 105–178, 2004.

[44] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. IEEE Computer Society, 2009, p. 88–99.

[45] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, IEEE/ACM. New York, NY, USA: IEEE/ACM, 2005, pp. 273–282. [Online]. Available: https://doi.org/10.1145/1101908.1101949

[46] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, Sep. 1995.

[47] K. Ahmed, M. Lis, and J. Rubin, "Slicer4j: A dynamic slicer for java," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. ACM, 2021, p. 1570–1574.

[48] "Slicer4j's github repository," https://github.com/resess/Slicer4J0, accessed: 2022-10-21.

[49] C. Hammacher, "Design and implementation of an efficient dynamic slicer for Java," Bachelor's Thesis, Nov. 2008.

[50] C. Galindo, S. Pérez, and J. Silva, "Slicing unconditional jumps with unnecessary control dependencies," in *Logic-Based Program Synthesis and Transformation: 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7–9, 2020, Proceedings*. Springer-Verlag, 2020, p. 293–308.

[51] "Defects4j's github repository," https://github.com/rjust/defects4j/releases/tag/v2.0.0, accessed: 2022-10-21.

[52] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11, 2011, p. 199–209.

[53] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

[54] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, "Multiple fault localization of software programs: A systematic literature review," *Information and Software Technology*, vol. 124, p. 106312, 2020.

[55] T. Janssen, R. Abreu, and A. J. Van Gemund, "Zoltar: a spectrum-based fault localization tool," in *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution@ runtime*, 2009, pp. 23–30.

[56] H. L. Ribeiro, R. P. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon, "Jaguar: A spectrum-based fault localization tool for real-world software," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2018, pp. 404–409.

[57] G. Balogh, V. Schnepper Lacerda, F. Horváth, and Á. Beszédes, "iFL for Eclipse – a tool to support interactive fault localization in Eclipse IDE," Presented in the Tool Demo Track of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST'19), 2019.

[58] Q. I. Sarhan, A. Szatmári, R. Tóth, and A. Beszedes, "Charmfl: A fault localization tool for python," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2021, pp. 114–119.

[59] S. Habchi, G. Haben, J. Sohn, A. Franci, M. Papadakis, M. Cordy, and Y. L. Traon, "What made this test flake? pinpointing classes responsible for test flakiness," in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 352–363.

[60] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: A selective mutation approach," in *Proceedings of the 29th annual ACM symposium on applied computing*, 2014, pp. 1293–1300.

[61] ——, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[62] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*, 2006, pp. 82–91.

[63] D. Tiwari, L. Zhang, M. Monperrus, and B. Baudry, "Production monitoring to improve test suites," *IEEE Trans. Reliab.*, vol. 71, no. 3, pp. 1381–1397, 2022.

[64] J. Silva, "A vocabulary of program slicing-based techniques," *ACM computing surveys (CSUR)*, vol. 44, no. 3, pp. 1–41, 2012.

[65] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 319–329.

[66] E. O. Soremekun, M. Böhme, and A. Zeller, "Programmers should still use slices when debugging," Saarland University, Tech. Rep., 2016. [Online]. Available: https://www.st.cs.uni-saarland.de/debugging/faultlocalization/technicalReport.pdf

[67] M. Weiser and J. Lyle, "Experiments on slicing-based debugging aids," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 187–197.

[68] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking," *Software: Practice and Experience*, vol. 23, no. 6, pp. 589–616, 1993.

[69] M. Harman and S. Danicic, "Using program slicing to simplify testing," *Software Testing, Verification and Reliability*, vol. 5, no. 3, pp. 143–162, 1995.

[70] D. W. Binkley, N. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 2014, pp. 109–120.

[71] D. W. Binkley, N. E. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS and the limits of static slicing," in *15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015*. IEEE Computer Society, 2015, pp. 1–10.

[72] T. Azim, A. Alavi, I. Neamtiu, and R. Gupta, "Dynamic slicing for android," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1154–1164.

[73] K. Ahmed, M. Lis, and J. Rubin, "Mandoline: Dynamic slicing of android applications with trace-based alias analysis," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 105–115.

[74] Z. Zhang, Y. Li, Y. Guo, X. Chen, and Y. Liu, "Dynamic slicing for deep neural networks," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 838–850.

[75] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[76] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 161–170.

[77] B. Hofer and F. Wotawa, "Spectrum enhanced dynamic slicing for better fault localization." in *ECAI*, vol. 242, 2012, pp. 420–425.

[78] S. Parsa, F. Zareie, and M. Vahidi-Asl, "Fuzzy clustering the backward dynamic slices of programs to identify the origins of failure," in *International Symposium on Experimental Algorithms*. Springer, 2011, pp. 352–363.

[79] T. Shu, L. Wang, and J. Xia, "Fault localization using a failed execution slice," in *2017 International Conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 2017, pp. 37–44.

[80] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices," *Journal of Systems and Software*, vol. 90, pp. 3–17, 2014.