Systematically Generated Formulas for Spectrum-Based Fault Localization

Qusay Idrees Sarhan[1, 2], Tamás Gergely[1], and Árpád Beszédes[1]
[1] Department of Software Engineering, University of Szeged, Szeged, Hungary
[2] Department of Computer Science, University of Duhok, Duhok, Iraq
{sarhan, gertom, beszedes}@inf.u-szeged.hu

***Abstract*—The basic element in Spectrum-Based Fault Localization (SBFL) are the risk evaluation formulas, which calculate a suspiciousness score for each program element based on test coverage and test case outcome information. This score can be used in debugging to identify the faulty element more efficiently. A large number of manually crafted formulas have been proposed, but a line of research tries to generate formulas (semi-)automatically. Some of these approaches are based on heuristic search (e.g., genetic algorithms), and researchers started only recently examining systematic ways to generate all possible formulas corresponding to a particular class of formula structures. In a recent work, we explored a very simple formula template as a proof of concept but this research failed to find a new formula that outperformed already published ones. In this paper, we take a next step and investigate a class of formula templates that are more elaborate but still feasible to explore fully for generating new formulas. Many of the generated formulas cover some well-known existing ones, but we were also able to find two new ones that are superior to the majority of the previously published formulas (evaluated on the Defects4J dataset) and are not present in literature.**

***Index Terms*—Spectrum-Based Fault Localization, debugging, suspiciousness score formulas, systematic search.**

I. Introduction

Software fault localization helps developers to find the locations of bugs in their programs and it is performed using different techniques. However, Spectrum-Based Fault Localization (SBFL) is the most used technique as it only uses code coverage and test results for locating faults [1]–[3]. In SBFL, formulas are the base to compute the probability of each program element (e.g., statement, method, or class) of being faulty based on statistical numbers calculated from the program spectra (tests and their results). In particular, how often each element is executed or not executed by passing versus failing test cases.

One of the main challenges in SBFL is how to introduce new formulas to enhance the performance by putting faulty elements at the beginning of the ranking list produced by SBFL as much as possible [4]. Thus, they can be examined and found efficiently.

The majority of the published formulas have been manually crafted, and some of the well-known examples include Tarantula [5], Ochiai [6] and DStar [7]. All these are based on some intuition and/or previous results from other domains. Researchers also experimented with combining existing formulas, adding external information, or generating new formulas by meta-heuristic search or artificial intelligence.

Recently, we proposed an approach based on systematic search – in contrast to ad-hoc, intuitive, search-based or machine learning (ML) methods – for introducing new formulas for SBFL [8]. But, we were able to check only a limited number of generated formulas, and were not able to find new better formulas compared to other top existing ones. This is not surprising since the template we used was very simple.

In this paper, we extend our previous work by using more formula templates, and we apply them on the Defects4J dataset [9] to determine their effectiveness. The systematic search for formulas means that we generate the formulas that conform to a predefined formula template, and we enumerate all possible ones. Then, the formula candidates are evaluated on a benchmark suite for fault localization effectiveness.

Our experimental results show that the systematic approach for finding new SBFL formulas is successful. We were able to find two new formulas that are better than all of the generated formulas presented in [8], and most of the formulas from related literature as well. The two new formulas achieved better performance in terms of average ranking (see Section VI-B) compared to others. Also, they gained positive improvements in the Top-N categories (see Section VI-C).

The main contributions of this paper are the following:

1) A large number of SBFL formulas were systematically generated and evaluated using two formula templates.
2) Two new SBFL formulas that are generated using a systematic approach based on formula templates are proposed.
3) The analysis of the impact of the new SBFL formulas on the overall SBFL effectiveness is discussed.

Note, that our approach significantly differs from heuristic approaches including search-based and ML-based methods. We are checking all possibilities systematically, and that way we can ensure that no option is left out in the search space.

The Research Questions of the paper are the following:

- **RQ1:** Can systematic search lead to new formulas that could outperform the existing ones?
- **RQ2:** What level of average rank improvements can we achieve using the systematically generated formulas?
- **RQ3:** What is the overall effect of the systematically generated formulas on SBFL effectiveness in terms of Top-N?

II. Background on SBFL

SBFL is a well-known software fault localization technique that only uses test coverage and test results to compute suspiciousness scores for program elements. In this section, we will present SBFL and how it can be used to find faults in software code.

The execution of test cases on program elements is recorded to extract the spectra (i.e., test coverage and test results) for the program under test. Program spectra information is represented as a matrix. The tests are represented by the columns, while the program elements are represented by the rows (often, a transposed version is used). If a test case covers a code element, the matrix element becomes 1; otherwise, it becomes 0. The test results are also stored in the matrix (i.e., the last row), where 0 denotes a passing test case and 1 denotes a failing test case.

Using program spectra, for each program element e, the following four basic statistical numbers, called the *spectrum metrics*, are then computed:

- **ep**: represents the number of passed test cases covering the program element e.
- **ef**: represents the number of failed test cases covering the program element e.
- **np**: represents the number of passed test cases not covering the program element e.
- **nf**: represents the number of failed test cases not covering the program element e.

Then, these four spectrum metrics can be used by an SBFL formula to suggest a ranked list of suspicious elements as an output for the tester. The element with the highest ranking on the list is the most likely to have a fault. As a result, SBFL can make it easier for developers to locate the faults in the target program's code.

To illustrate the work of SBFL, consider a Java program that performs some specific mathematical operations, and comprises four methods M_i ($1 \leq i \leq 4$) and six test cases T_j ($1 \leq j \leq 6$), as shown in Figure 1. The program has a fault in the method M1 (the first statement should be z = x + y). The test cases are executed on the program and then the execution information (the spectrum) of the four methods in passed and failed test cases are recorded as presented in Table I. This table also includes the four spectrum metrics. The program spectra is then used by an SBFL formula such as Tarantula (see Table III) to compute the suspiciousness of each method of being faulty. Table II presents the scores and ranks of the methods of our code example. It can be noted that the method M1 is ranked 1 while the others share the rank 3; thus, the method M1 should be examined before the others.

Table III includes some of the most widely used SBFL formulas, which are also the baselines used in this work for evaluating the candidate new formulas. We note that literature includes about 80 formulas altogether, which can be found in various survey works [1]–[3].

```
class Example
{
 M1: double add(double x, double y)
 {
   double z = x * y;
   return z;
 }

 M2: double sub(double x, double y)
 {
   double z = x - y;
   return z;
 }

 M3: double mult(double x, double y)
 {
   double z = x * y;
   return z;
 }

 M4: double div(double x, double y)
 {
   double z = x / y;
   return z;
 }
}
```

```
class ExampleTest {
@Test void T1() {
Example tester = new Example();
assertEquals(6, tester.add(3 , 3));}

@Test void T2() {
Example tester = new Example();
assertEquals(4, tester.add(2 , 2));}

@Test void T3() {
Example tester = new Example();
assertEquals(1, tester.sub(3 , 2));}

@Test void T4() {
Example tester = new Example();
assertEquals(25, tester.mult(5 , 5));}

@Test void T5() {
Example tester = new Example();
assertEquals(2, tester.div(4 , 2));}

@Test void T6() {
Example tester = new Example();
assertEquals(4, tester.div(8 , 2));}}
```

Fig. 1. SBFL example – Java code and test cases

TABLE I
SBFL EXAMPLE – SPECTRA AND BASIC STATISTICS

	T1	T2	T3	T4	T5	T6	ef	ep	nf	np
M1	1	1	0	0	0	0	1	1	0	4
M2	0	0	1	0	0	0	0	1	1	4
M3	0	0	0	1	0	0	0	1	1	4
M4	0	0	0	0	1	1	0	2	1	3
Results	1	0	0	0	0	0				

TABLE II
SBFL EXAMPLE – SCORES AND RANKS

	Tarantula score	Tarantula rank
M1	0.83	1
M2	0.0	3
M3	0.0	3
M4	0.0	3

TABLE III
SBFL FORMULAS

Barinel [10] =	$\frac{ef}{ef+ep}$
Cohen [11] =	$\frac{2\cdot(ef\cdot np)-2\cdot(nf\cdot ep)}{(ef+ep)\cdot(ep+np)+(nf+np)\cdot(ef+nf)}$
Dice [11] =	$\frac{2\cdot ef}{ef+nf+ep}$
DStar [7] =	$\frac{ef^2}{ep+nf}$
Jaccard [12] =	$\frac{ef}{ef+nf+ep}$
Kulczynski1 [11] =	$\frac{ef}{nf+ep}$
Ochiai [6] =	$\frac{ef}{\sqrt{(ef+nf)\cdot(ef+ep)}}$
SorensenDice [13] =	$\frac{2\cdot ef}{2\cdot ef+nf+ep}$
Tarantula [5] =	$\frac{\frac{ef}{ef+nf}}{\frac{ef}{ef+nf}+\frac{ep}{ep+np}}$

III. SBFL Formula Improvement

This section summarizes the most important efforts to improve SBFL by focusing on its formulas.

A. Introducing new SBFL formulas

The first approach is to design new SBFL formulas based on intuition, past experience or by reusing results from other disciplines. For example, Ochiai [14] and Binary [15] came from the fields of biological research. Authors in [7] and in [10] proposed new SBFL formulas called "DStar" and "Barinel", respectively, based on intuition. Each proposed formula has been compared with several widely used formulas and it showed good performance compared to others. The authors in [16] proposed a new formula called Metrics Combination (MECO) which effectively finds errors without the need for prior knowledge of program structure or semantics. Their idea is that several metrics (e.g., Failed Execution Flag, Assumption Proportion) can be extracted from the target program spectra and combined to propose a new formula.

Many studies such as [10] and [17] claimed (on a theoretical level) that an optimal SBFL formula exists. However, this is not necessarily true in practice. The fact that faulty programs in practice might not adhere to the same theoretical assumptions is one potential explanation for the discrepancy between the theoretical and empirical results of SBFL formulas, as discussed in [18], [19]. As shown in [20], there is no optimal formula for all types of faults.

B. Modifying existing SBFL formulas

The second approach is formula modification. The authors in [21] improved the performance of the Tarantula formula by modifying some part of it to amplify its scores. However, the improved Tarantula does not always make improvements in the ranking. Also, the authors did not evaluate the improved Tarantula using well-known evaluation metrics. The authors in [22] modified three well-known SBFL formulas based on the idea that some failed test cases may provide more testing information than other failed test cases. Therefore, for the three used formulas, different weights for failed test cases were assigned and then applied with multi-coverage spectra.

C. Combining existing SBFL formulas

Another way is to combine existing formulas. The authors in [23] proposed a new SBFL formula by combining 40 different formulas using different voting systems. The proposed method extracts information from the program using mutation testing and then combines multiple formulas based on the gathered information using different voting systems to generate a new formula. The results of experiments have shown that the formula generated by their method is better than several existing ones. Multiple formulas also can be combined into a single new one. The resulting formula is a hybrid formula; which combines the advantages of the formulas that have been used in the combination as in [24].

D. Adding new information to existing SBFL formulas

Involving new information to existing SBFL formulas can also lead to improvements. For example, the authors in [25] utilized the method calls frequency during the execution of failed tests to add new contextual information to existing formulas. Thus, the ef of each formula was changed to the frequency ef. The experiments improved SBFL effectiveness. However, this approach can only be applied to the formulas that have the ef numerator. Also, it is considered heavy as it requires tracing the execution of each method call, as caller or callee, in the failed test cases.

E. Generating new SBFL formulas by meta-heuristic search

The authors in [26] used genetic programming (GP) to evolve new formulas from a hybrid dataset (i.e., from different benchmark datasets). They were able to produce several new formulas that outperformed many existing ones. However, this approach poses several issues: (a) it is not systematic, thus it does not guarantee that even a simple formula is examined. (b) the results of applying GP may vary greatly from one run to another as it depends on the initial selection of population. (c) a couple of parameters (e.g., population size, number of generations, mutation rate, etc.) must be set by human, thus the probability of finding the optimal solution is not too much. (d) generating formulas based on this approach has a disadvantage that existing datasets do not cover all possible types of bugs. Thus, a generated formula may fail to locate a bug that is not included in the used datasets.

A final critique with this approach is that the generated formulas are often difficult to comprehend and non-intuitive, including complex computations and magic constants such as the following formulas:

$$\text{HDGPCR02} = (\sqrt[5]{\sqrt[4]{np}} + 3ef + np) - (\frac{ef}{ep} - 2ef)(ef + np + \frac{np}{ep})$$

$$\text{HDGPCR20} = \sqrt[5]{\sqrt{ep}(\frac{np}{ep})} - \sqrt[3]{nf} - \frac{np}{ef} + (nf - ef - \sqrt{ep}(np))$$

$$\text{HDGPCR23} = \frac{\sqrt{ep}+\frac{np}{ef}}{ep+ef+nf} - (nf)(\frac{ef}{ep}) + (\frac{\sqrt[3]{nf}}{np})(nf)$$

F. Machine Learning

Machine learning has also been used for fault localization to learn scores from spectra [27], [28], or to use likely invariant diffs and suspiciousness scores as features to learn the ranks [29]. But, such approaches do not produce a resulting formula that can be reused and are typically specific to a particular subject system. Also, the search space cannot be meaningfully controlled, and the decisions made by ML and parts of the search space explored will remain blackbox.

G. **This work:** *Systematic formula generation*

To overcome the problems, mentioned in sections III-E and III-F, assigned with meta-heuristic search or machine learning to find new formulas, in this paper, we follow a completely different direction to automatically generate formulas, and explore the formulas in a *systematic* manner. Of course, there are infinite number of formulas that can be generated based on the spectrum metrics, which makes it impossible to

exhaustively examine all of them. The idea we proposed in our previous study [8] was to limit the search space using formula templates, and explore a particular class of formulas *exhaustively*.

We follow this line of research, and differently from heuristic search attempts, we perform the search systematically using different formula templates that individually cover a set of formulas sharing similar structure. In the mentioned preliminary study, we found formulas that outperformed some existing ones but failed to achieve significant improvement over the most successful existing techniques.

In this paper, we introduce our experimental results with extended formula templates compared to [8]. Our goal is to systematically examine a broader set of formulas and to find out how different the generated formulas will be in terms of their ranking ability. We also outline the possible extensions for future work, primarily in terms of more advanced formula templates. The goal is to be able to cover some already published formulas (as a sanity check that the method is meaningful), and potentially discover new, better ones as we managed to do in this study.

Comparing to heuristic search and machine learning approaches, our approach is complementary and it cannot replace them because they can cover much larger search space but not completely; while our approach can cover smaller search space with each formula template but completely.

IV. Formula templates

The basic idea of systematic formula generation is that we combine the four spectrum metrics ef, ep, nf and np using mathematical operations in all possible combinations. While at first this seems straightforward, systematic enumeration of all possible SBFL formulas is not simple. Since, in general, the number of possible formulas is infinite we must limit the types of formulas to a meaningful subclass. For example, all four values can have a fixed exponent at most. But even using the basic mathematical operations only, like addition, multiplication, fraction, and only linear combinations of the elements we will face combinatorial explosion.

Further issues are that many generated formulas will contain elements that can be mathematically simplified or rewritten, and that they will still sometimes produce syntactically different, but semantically *equivalent* formulas to each other.

Furthermore, even if two formulas are not mathematically equivalent, they can be *rank-equivalent*. This means that they will produce the same ranking lists; despite the score values being different, in this case there is a monotonic transformation between the values [30].

At the same time, many of the previously published, manually crafted formulas can fit to a relatively simple structure (as opposed to GP-generated ones). Hence, our goal in this research is an *exhaustive exploration of all possible formulas that conform to a specific formula template*. Our goal is to define templates that have the following properties:

- They cover as many existing formulas as possible (this means these are probably useful structures).
- Are combinatorically feasible.
- Can be competitive with manually crafted formulas.

We build on our previous work, where we proposed a systematic search to evaluate SBFL formulas and possibly find new ones [8]. We reported the evaluation of 24 formulas generated by a simple template (see Table IV):

$$\frac{\sum_{t\in\{ef,ep\}} n_t t}{\sum_{t\in\{ef,ep\}} d_t t} = \frac{n_{ef}\,ef + n_{ep}\,ep}{d_{ef}\,ef + d_{ep}\,ep}, \tag{1}$$

where $n_{ef}, n_{ep}, d_{ef}, d_{ep} \in \{-1, 0, 1\}$. The template in Equation 1 covers previously reported formulas Barinel (= Braun = Coef = SBI = F10) and Wong I (= F2) and Wong II (= F16) [11].

TABLE IV
The formulas examined in [8].

F1 =	ep	F13 =	$-ep$
F2 (Wong I) =	ef	F14 =	$-ef$
F3 =	$ep + ef$	F15 =	$-ep - ef$
F4 =	$ep - ef$	F16 (Wong II) =	$-ep + ef$
F5 =	$\frac{1}{-ep}$	F17 =	$\frac{1}{ep}$
F6 =	$\frac{ef}{ep}$	F18 =	$\frac{-ef}{ep}$
F7 =	$\frac{1}{-ef}$	F19 =	$\frac{1}{ef}$
F8 =	$\frac{-ep}{ef}$	F20 =	$\frac{ep}{ef}$
F9 =	$\frac{1}{-ep-ef}$	F21 =	$\frac{1}{ep+ef}$
F10 (Barinel) =	$\frac{ef}{ep+ef}$	F22 =	$\frac{-ef}{ep+ef}$
F11 =	$\frac{1}{-ep+ef}$	F23 =	$\frac{1}{ep-ef}$
F12 =	$\frac{ef}{ep-ef}$	F24 =	$\frac{-ef}{ep-ef}$

In this paper, we define more elaborate formula templates. The template in Equation 2 extends template in Equation 1 with a single spectrum metric (ef, ep, nf, np) by applying a simple arithmetic operation ($+$, $-$, $\cdot$, $/$) between them.

$$\begin{aligned}\frac{\sum_{t\in\{ef,ep\}} n_t t}{\sum_{t\in\{ef,ep\}} d_t t} \otimes \{ef, ep, nf, np\} = \\ \frac{n_{ef}\,ef + n_{ep}\,ep}{d_{ef}\,ef + d_{ep}\,ep} \otimes \{ef, ep, nf, np\}\,,\end{aligned} \tag{2}$$

where $n_{\{ef,ep\}}, d_{\{ef,ep\}} \in \{-1, 0, 1\}$ and $\otimes \in \{+, -, \cdot, /\}$.

Equation 1 resulted in 81 literal templates, 24 of which remained after sorting out constant and equivalent ones. Equation 2 results in $4 \cdot 4 \cdot 81 = 1,296$ formulas literally. However, based on the 24 different non-trivial formula instances of paper [8], there remain only $4 \cdot 4 \cdot 24 = 384$ candidates. Plus $4 \cdot 2 = 8$, the four basic mathematical operations for the two additional spectrum metrics nf and np, which are added because the 24 formula instances from [8] do not include constant-equivalent ones. For example, $\frac{ef}{ef}$ was not examined in [8], but $\frac{ef}{ef} \cdot nf = nf$ is a valid formula instance. At least 80 formula instances generated from this new template have

already been covered in [8], and there are at least two pairs of new generated formulas that are equivalent with each other.

The next formula template we examined is shown in Equation 3:

$$\frac{\sum_{t\in\{ef,ep\}} n_{1,t}t}{\sum_{t\in\{ef,ep\}} d_{1,t}t} \otimes \frac{\sum_{t\in\{ef,ep\}} n_{2,t}t}{\sum_{t\in\{ef,ep\}} d_{2,t}t} = \frac{n_{1,ef}\,ef + n_{1,ep}\,ep}{d_{1,ef}\,ef + d_{1,ep}\,ep} \otimes \frac{n_{2,ef}\,ef + n_{2,ep}\,ep}{d_{2,ef}\,ef + d_{2,ep}\,ep}, \tag{3}$$

where $n_{\{1,2\},\{ef,ep\}}, d_{\{1,2\},\{ef,ep\}} \in \{-1, 0, 1\}$ and $\otimes \in \{+, -, \cdot, /\}$.

In this template, we have put together two instances of the formulas generated by the template in Equation 1 using a basic arithmetic operation. This would result in $81 \cdot 81 \cdot 4 = 26,244$ formula instances literally. However, counting on the non-equivalent formulas defined in [8], we can restrict our measurements to $24 \cdot 24 \cdot 4 = 2,304$ literally generated formulas. Furthermore, addition and multiplication are commutative operations, technically halving the resulting formula instances, while addition-subtraction and multiplication-division are inverses that can also yield in the same formula when applied on different operands (e.g. $\mathrm{F1}+\mathrm{F2} = \mathrm{F2}-\mathrm{F13}$). Subtraction and division can result in identical (constant) formulas, and some combinations can also yield in formulas already reported in our previous work [8]. Thus, by rough calculation, at most 400 different formulas exist, but these formulas may contain equivalent ones too.

The two presented formula templates also overlap, generating literally equivalent formulas. However, we did not make a thorough equivalence or ranking equivalence analysis on the resulting formulas for either of the formula templates (the equivalence proofs of different SBFL formulas has been investigated in [30] via a theoretical comparison approach). Instead, we utilized previous equivalence calculations and ran the measurements for all the resulting formulas (including the equivalent ones) and sanity-checked the results based on the discovered equivalences (i.e., checked manually by random sampling if two formulas that should be equivalent really produce the same results).

The above templates cover Hamming (= Lee = NFD) and equivalents of Euclid and Ochiai3 formulas. These formulas [31] are presented in Table V.

TABLE V
SBFL FORMULAS COVERED BY NEW TEMPLATES
(*: ONLY RANK-EQUIVALENTS ARE COVERED)

	Formulas
Hamming	$ef + np$
Euclid*	$\sqrt{ef + np}$
Ochiai3*	$\frac{ef^2}{(ef+ep+nf+np)\cdot(ef+ep)}$

Figure 2 shows the systematic steps followed in this section in order to search for new SBFL formulas.

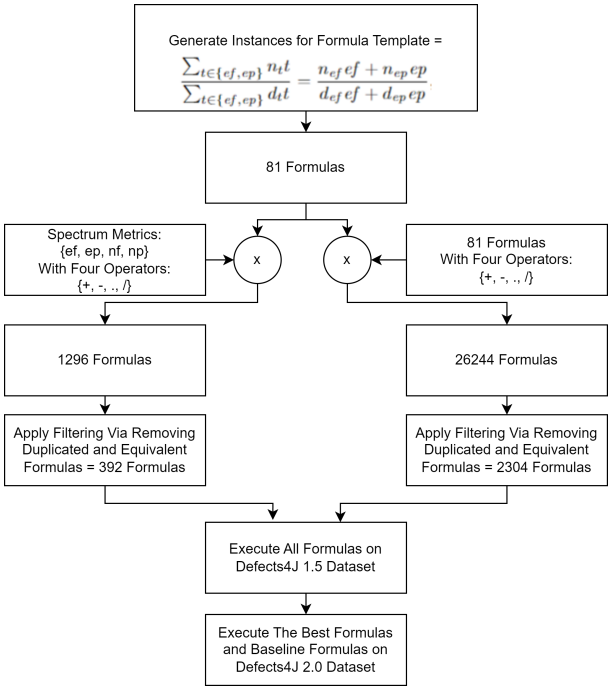

Fig. 2. Systematic search for new SBFL formulas

V. MEASUREMENTS

A. Evaluation

In this study, we performed two types of measurements. In the first, lightweight measurement, we generated all the formulas from our templates, and checked their fault localization performance on the six programs (i.e., Chart, Closure, Lang, Math, Mockito and Time) of Defects4J 1.5. We used this measurement to check the potential of all the generated formulas. Note, that we did not filter out equivalent formulas, all generated ones were measured. We used the results as a sanity check of our implementation: after the measurement, we checked if some theoretically equivalent formulas produce the same numbers. The results are presented in Section VI-A.

Then, based on the results of this first measurement, we re-measured the most promising well-performing formulas (together with the chosen existing baseline formulas) on Defects4J 2.0 as it has more programs and bugs. Sections VI-B to VI-C present the results of this second measurement.

To evaluate the effectiveness of SBFL formulas, we use *average rank* (see Section VI-B) and *Top-N categories* (see Section VI-C) metrics that other researchers in the literature have previously used [32], [33].

As mentioned above, the SBFL method ranks program elements by their probability of being faulty, which is computed using the formulas. Thus, for a given program version and its corresponding tests, we can compute the ranked list of program elements and check where the faulty program element is placed on this ranked list. The closer the element is to the beginning of the list, the better.

The average rank metric tells us that on a dataset (set of faulty program versions and their tests) what is the average rank of the faulty program elements computed using the formula being evaluated. The smaller value means that (on average) the given formula ranks the faulty elements in a way that it can be found earlier using the ranked list. The Top-N categories metric is based on the observation that developers tend to use only the beginning of the list, and if the faulty element is not in the first N (usually 5 to 10) places, they go for alternative methods to find the bug. Thus, a formula is better if it ranks more faulty elements in the first N places.

B. Subject programs

In this study, we used the faulty programs of version v2.0 of Defects4J [34]; where 17 open-source Java programs have 835 real single and multiple faults[1]. We excluded some faults in this study due to instrumentation errors or unreliable test results. Thus, a total of 782 faults were included in our final dataset. Table VI presents our subject programs.

TABLE VI
SUBJECT PROGRAMS (ITALICS INDICATE PROGRAMS USED IN THE LIGHTWEIGHT MEASUREMENT)

Project	Number of bugs	Size (KLOC)	Number of tests	Number of methods
Chart	25	96	2.2k	5.2k
Cli	39	4	0.1k	0.3k
Closure	171	91	7.9k	8.4k
Codec	17	10	0.4k	0.5k
Collections	1	46	15.3k	4.3k
Compress	36	11	0.4k	1.5k
Csv	16	1	0.2k	0.1k
Gson	15	12	0.9k	1.0k
JacksonCore	25	31	0.4k	1.8k
JacksonDatabind	101	4	1.6k	6.9k
JacksonXml	5	6	0.1k	0.5k
Jsoup	90	14	0.5k	1.4k
JxPath	21	21	0.3k	1.7k
Lang	60	22	2.3k	2.4k
Math	104	84	4.4k	6.4k
Mockito	30	11	1.3k	1.4k
Time	26	28	4.0k	3.6k

It is worth mentioning that the systematic search process used in this study has two types of measurements: lightweight and complete. For the first, lightweight measurement, we used bugs from Defects4J 1.5. For the second, complete measurement, we used bugs from all the programs of the dataset Defects4J 2.0, as presented in Table VI.

C. Granularity of data collection

In this study, we used granularity at the method-level. This level of granularity scales nicely to large-scale programs and gives users an abstraction at a more reasonable level [35], [36]. However, our approach could be used at the statement-level too, and this is a future work.

VI. RESULTS

Our experimental results are based on evaluation metrics that have been used widely in literature [32], [35], [37].

[1]https://github.com/rjust/defects4j

A. Generated formulas

Our template in Equation 2 generates 1,296 formula instances, while Equation 3 yields 26,244 formula instances. Based on the equivalence calculations of the previous work [8], we generated 392 and 2,304 candidate formulas, respectively. All formulas left out are equivalent with one or more of the candidate ones, and there were several equivalent formulas within these candidate sets too. However, we did not further filter the set, but measured all of the formulas in it, performing a lightweight analysis as described in Section V.

As we started to generate formula instances from the ones reported in [8], we could check if the resulting combined formula instance improved fault localization performance. Based on the average ranks, 25.97% of the newly generated formulas improved the ones used in the combination. However, we found 8 formulas that made absolute improvement, i.e. better average ranks than any of the formulas in [8]. By examining these 8 formulas, they were equivalent versions of two formulas. The two new formulas are the following:

$$\text{SGF-1} = \frac{ef^2}{ef + ep} \tag{4}$$

$$\text{SGF-2} = ef \cdot np \tag{5}$$

After finding these formulas, we performed a further analysis on our full set of subject programs using these two new formulas. We compared them to our baselines, that is, to some of the state-of-the-art SBFL formulas listed in Table III. This second analysis has shown that these two formulas can outperform many already published state-of-the-art formulas too (details are presented in the next sections).

One of the advantages of our new formulas is that they produce less ties (i.e., less number of program elements that share the same suspicion score [38]) compared to the existing ones used in this study. Take this example: we have two program elements with the following spectrum metrics: Element A ($ef = 1$, $ep = 0$, $nf = 3$, $np = 6$) and Element B ($ef = 2$, $ep = 0$, $nf = 2$, $np = 6$). Applying Tarantula will result the same suspicion score (i.e., 1.0) for both elements; this is considered an issue for the developer as which element s/he will examine first. However, SGF-1 will result in different score for each element; score 1.0 for Element A and score 2.0 for Element B, while SGF-2 gives score 6.0 to Element A and score 12.0 to Element B. As a result, this reduces the ties between program elements and the new formulas give greater suspicion scores to program elements that are executed by more failed test cases compared to others.

Also, notice the similarity of SGF-1 with Barinel. The difference is the square of ef in the numerator, which essentially means that our new formula puts a bigger emphasis on the failed tests executing the code element, which is the most important constituent of SBFL.

The success of SGF-2 also seems logical, as it combines the previously mentioned important element ef with the counter of passing tests that are not executing the element in question. This measure, np is in essence the opposite of the former,

and intuition dictates that the bigger this number the more suspicious the current element should be.

Answer to RQ1: Analysis shows that systematic search can lead to new SBFL formulas not present in literature, whose performance is comparable to or can even outperform existing ones.

B. Achieved improvements in the average ranks

Here, we use the average rank approach, which assigns to all of the tied elements (those having the same suspicion score) their average rank. Thus, if there are E pieces of tied elements having consecutive ranks (in random order) starting from rank S in the ranked list, we assign the same rank, using Equation 6, to all of these elements.

$$\text{Average Rank} = S + \left(\frac{\text{E - 1}}{2}\right) \tag{6}$$

For handling the exceptional case when the denominator of the formula is 0 for a program element (the div/0 problem), we assign the 0 suspicion score to the given element.

Table VII presents the average ranks of SGF-1 and SGF-2 in equations 4 and 5 compared to our baseline formulas, for each subject system separately.

We can observe that some of the new formulas SGF-1 and SGF-2 can outperform almost all the selected formulas in terms of reducing the average rank, or produce the same results. Interestingly, the two formulas are better performing in mutually exclusive cases. For 3 projects almost all of the formulas produced the same ranks, in 3 cases SGF-1 (together with Ochiai) produced the best average ranks while in 3 cases SGF-2 was the best ranking formula in average. Regarding the median values, in 12 projects SGF-1 could produce the best value of the other formulas, while SGF-2 did the same for 7 projects and produced even better medians in 5 additional cases. In two cases SGF-2 produced better maximum ranks than any other examined formula. Overall, DStar is still the best average performing formula, but as we can observe, the margin is very small, and as we can observe later on the detailed data it is not necessarily a clear winner.

Another interesting phenomenon is that SGF-1 produced the same average ranks as Ochiai for all but one case; for the Closure program it was 0.02 ranks better. A possible explanation for this is the following. By squaring the Ochiai formula, – this transformation being monotonic – it will result in the same ranking in many cases because we get SGF-1 divided by $ef + nf$. But, in most of the cases, we have only a single failing test case, which means this factor will be 1, not modifying the score and the rank of the two formulas.

Answer to RQ2: We noticed that the two new formulas can improve the performance of SBFL by reducing the ranks compared to most of baseline formulas, while the results are very similar to some of the existing ones. The two formulas produce better results in mutually exclusive cases. The average improvement of rank positions in the used benchmark was about 2 positions overall.

C. Achieved improvements in the Top-N categories

Several studies including [37], [39] report that developers think that inspecting the first five program elements in the ranking list produced by an SBFL technique is acceptable and that the first ten elements are the upper bound for inspection before ignoring the ranking list. Therefore, the performance of SBFL can also be evaluated by focusing on these rank positions, collectively called Top-N, as follows: (a) Top-N: When the rank of a faulty program element is less or equal to N. (b) Other: When the rank of a faulty program element is more than the highest N value used in the categorizations.

In our experiments, we will use 1, 3, 5 and 10 in place of N, and call the cases when a rank is moved to an upper rank interval *enabling improvement*.

Table VIII presents the number of bugs in the Top-N categories and their percentages for the used dataset. It can be noted that the new generated formulas achieve improvements in all categories by moving many bugs to higher ranked categories compared to almost all the other formulas. For example, both new formulas move more bugs to the Top-1 category compared to almost all the other formulas and move more bugs also from the "Other" category to the higher ranks categories compared to almost all the others. In particular, SGF-2 managed to put the most bugs to the first place, compared to any of the baselines and to SGF-1 too.

TABLE VIII
TOP-N CATEGORIES

	Top-1		Top-3		Top-5		Top-10		Other	
	#	%	#	%	#	%	#	%	#	%
Barinel	98	12.53%	265	33.89%	344	43.99%	437	55.88%	345	44.12%
Cohen	104	13.30%	271	34.65%	344	43.99%	438	56.01%	344	43.99%
Dice	104	13.30%	271	34.65%	344	43.99%	438	56.01%	344	43.99%
DStar	103	13.17%	274	35.04%	352	45.01%	450	57.54%	332	42.46%
Jaccard	104	13.30%	271	34.65%	344	43.99%	438	56.01%	344	43.99%
Kulczynski1	103	13.17%	251	32.10%	319	40.79%	407	52.05%	375	47.95%
Ochiai	106	13.55%	276	35.29%	353	45.14%	450	57.54%	332	42.46%
SorensenDice	104	13.30%	271	34.65%	344	43.99%	438	56.01%	344	43.99%
Tarantula	98	12.53%	265	33.89%	344	43.99%	437	55.88%	345	44.12%
SGF-1	106	13.55%	276	35.29%	353	45.14%	450	57.54%	332	42.46%
SGF-2	119	15.22%	275	35.17%	348	44.50%	449	57.42%	333	42.58%

Answer to RQ3: The two new formulas showed improvements in the Top-N categories. Using SGF-1, we were able to increase the number of cases where the faulty method became the top-ranked element by 2–8%, and by using SGF-2 this rate was 13–21%. SGF-2 produced the most Top-1 elements overall. In some cases we were able to achieve 13% enabling improvement by moving 12–43 bugs from the "Other" category into one of higher-ranked categories by using the formula SGF-1, while by using SGF-2 this rate was 12% (enabling improvements for 11–42 bugs).

VII. THREATS TO VALIDITY

In this study, we considered the following actions to avoid or minimize different threats of validity:

- Selection of evaluation metrics: we used a number of well-known evaluation metrics to ensure multiple-dimension comparisons and to assure the validity of our results and the conclusions that follow. Additionally, each evaluation metric used was thoroughly described.

TABLE VII
AVERAGE RANKS (THE BEST VALUES FOR A PARTICULAR ROW ARE SHOWN IN BOLD)

Project	Barinel	Cohen	Dice	DStar	Jaccard	Kulczynski1	Ochiai	SorensenDice	Tarantula	SGF-1	SGF-2
Chart	15.94	9.42	9.46	9.18	9.46	609.54	**8.82**	9.46	15.94	**8.82**	34.34
Cli	16.68	16.63	16.58	15.29	16.58	19.64	15.4	16.58	16.68	15.4	**14.01**
Closure	97.27	98.67	98.58	87.61	98.58	98.55	88.48	98.58	97.27	88.46	**84.23**
Codec	28.29	**26.44**	28.06	28.03	28.06	28	28.06	28.06	28.29	28.06	26.85
Collections	**1**	**1**	**1**	**1**	**1**	2155	**1**	**1**	**1**	**1**	**1**
Compress	17.62	17.62	17.54	16.01	17.54	43.43	16.12	17.54	17.62	16.12	**15.54**
Csv	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**	**6.5**
Gson	19.27	19.17	19.17	19.23	19.17	**18.9**	19.23	19.17	19.27	19.23	19.23
JacksonCore	6.84	**6.36**	**6.36**	6.64	**6.36**	136.64	6.92	**6.36**	6.78	6.92	7.36
JacksonDatabind	59.53	59.56	59.57	**59.11**	59.57	234.04	59.12	59.57	59.53	59.12	61.39
JacksonXml	**18.6**	**18.6**	**18.6**	**18.6**	**18.6**	72.3	**18.6**	**18.6**	**18.6**	**18.6**	**18.6**
Jsoup	34.77	35.88	34.71	34.01	34.71	45.15	**33.97**	34.71	34.77	**33.97**	34.29
JxPath	**44.24**	44.81	45	54.36	45	97.02	54.07	45	**44.24**	54.07	74.38
Lang	5.37	4.74	4.72	**4.62**	4.72	147.77	4.67	4.72	5.37	4.67	4.72
Math	9.94	**9.81**	9.82	9.94	9.82	190.04	10	9.82	9.94	10	10.56
Mockito	32.15	31.07	30.77	**28.57**	30.77	30.77	28.73	30.77	32.15	28.73	32.17
Time	19.71	19.67	19.63	18.69	19.63	106.5	**18.4**	19.63	19.71	**18.4**	21.79
Total Average Rank	41.38	41.46	41.33	**38.80**	41.33	132.07	38.99	41.33	41.38	38.99	39.97

- Correctness of implementation: we reviewed the code to make sure that our experiment's implementation is accurate and correct. Furthermore, we have tested our approach multiple times to make sure it works as intended.
- Selection of subject programs: we evaluated the effectiveness of the new generated formulas on fault localization using 17 Java subject programs. Thus, we cannot generalize our findings to other programs. However, we believe that the selected subject programs are representative as they have real faults, varying in size and complexity, and the benchmark containing them, Defects4J, is used commonly in other studies on software fault localization.
- Exclusion of faults: in our experiment, 53 faults (about 6% of the total faults) of the Defects4J dataset were excluded due to technical limitations. The issue here is whether other researchers using the same dataset will be able to replicate our findings. This exclusion was in no way influenced by the results of the used metrics and the excluded faults are distributed in the dataset approximately evenly, so we believe that this risk can be considered minimal.
- Selection of SBFL formulas: to evaluate the effectiveness of the new generated formulas on fault localization, we selected a set of nine formulas in our experiment, which is just a fraction of the proposed formulas in literature. However, we cannot guarantee that the same results can be obtained by using other formulas. To mitigate the effect of this issue, we selected these formulas that are commonly used in other studies on fault localization.
- Granularity level: we verified the concept on the granularity of functions, however in certain applications, such as automated program repair, statement granularity is required. It is unclear at present if the findings in this paper are generalizable to statements as well.

VIII. CONCLUSIONS

This paper extended the effort to systematically search for SBFL formulas in [8]. We defined new formula templates, which are more elaborate and can cover more existing formulas. The results of our extended formula templates show that the proposed approach led to new formulas that are not reported in the literature and also outperform many well-known existing ones. In particular, formula SGF-2 performed very well in all measurements, and being surprisingly simple, we think that it is very competitive to many previously advised and widely used manually crafted formulas.

This proves that the concept is valid and research on systematic SBFL formula generation is a promising direction. Compared to the GP-generated approaches or ML (Section III), our approach generates readable and explainable formulas.

For future work, we will perform the following studies:

- Extending the template to e.g. polynomial, exponential, and/or logarithmic to generate more elaborate formulas and maybe get even better results.
- Comparing the effectiveness of the formulas generated for all the identified templates with each other.
- Combining the best formulas from different templates into a single formula that has the advantages of others.
- Expanding the four spectrum metrics ef, ep, nf and np of program spectra with other contextual information or possibly involving count-based spectra too.
- Involving other benchmark datasets to measure how much impact do they have on finding good formulas.
- Assessing how the newly generated formulas perform at granularity levels other than method-level.
- Assessing the performance trade-offs between our approach and other heuristic search and ML approaches.

Our measurement data is available at https://doi.org/10.5281/zenodo.7239692.

ACKNOWLEDGEMENT

This research was part of project TKP2021-NVA-09 supported 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] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A Literature Review," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 5, pp. 1–8, sep 2014.

[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, aug 2016.

[3] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges," pp. 1–46, jul 2016.

[4] Q. I. Sarhan and A. Beszedes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022.

[5] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, 2002, pp. 467–477.

[6] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2006, pp. 39–46.

[7] 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, 2014.

[8] 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.

[9] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, 2014, pp. 437–440.

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

[11] Neelofar, "Spectrum-based Fault Localization Using Machine Learning," 2017. [Online]. Available: https://findanexpert.unimelb.edu.au/scholarlywork/1475533-spectrum-based-fault-localization-using-machine-learning

[12] R. Abreu, P. Zoeteweij, and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, 2007, pp. 89–98.

[13] T. J. Sørensen, "A method of establishing groups of equal amplitude in plant sociology based on similarity of species content and its application to analyses of the vegetation on danish commons," *København: I kommission hos E. Munksgaard.*, pp. 1–34, 1948.

[14] A. Ochiai, "Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions–II," *Bulletin of the Japanese Society of Scientific Fisheries*, vol. 22, no. 9, pp. 526–530, 1957.

[15] A. H. Cheetham and J. E. Hazel, "Binary (presence-absence) similarity," *Journal of Paleontology*, vol. 43, no. 5, pp. 1130–1136, 1969.

[16] A. Ajibode, T. Shu, K. Said, and Z. Ding, "A fault localization method based on metrics combination," *Mathematics*, vol. 10, no. 14, 2022. [Online]. Available: https://www.mdpi.com/2227-7390/10/14/2425

[17] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, aug 2011. [Online]. Available: https://doi.org/10.1145/2000791.2000795

[18] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, 2014, pp. 191–200.

[19] 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. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1616

[20] 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, Jun. 2017.

[21] X. Liang, L. Mao, and M. Huang, "Research on improved the tarantula spectrum fault localization algorithm," in *Proceedings of 2nd International Conference on Information Technology and Electronic Commerce*, 2014, pp. 60–63.

[22] Y.-S. You, C.-Y. Huang, K.-L. Peng, and C.-J. Hsu, "Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging," in *2013 IEEE 37th Annual Computer Software and Applications Conference*, 2013, pp. 180–189.

[23] 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.

[24] J. Kim, J. Park, and E. Lee, "A new hybrid algorithm for software fault localization," in *The 9th International Conference on Ubiquitous Information Management and Communication*, 2015, pp. 1–8.

[25] B. Vancsics, F. Horvath, A. Szatmari, and A. Beszedes, "Call frequency-based fault localization," in *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 365–376.

[26] 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.

[27] H. Xie, Y. Lei, M. Yan, Y. Yu, X. Xia, and X. Mao, "A universal data augmentation approach for fault localization," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 48–60.

[28] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *J. Softw. Evol. Process.*, vol. 33, no. 3, 2021.

[29] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 177–188.

[30] 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.

[31] L. H. Jie, "Software Debugging Using Program Spectra," 2011. [Online]. Available: https://minerva-access.unimelb.edu.au/items/41ddbedf-f082-55f3-be94-078c2f35f225

[32] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang, "Combining spectrum-based fault localization and statistical debugging: An empirical study," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 502–514.

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

[34] T. Hirsch and B. Hofer, "A systematic literature review on benchmarks for evaluating debugging approaches," *Journal of Systems and Software*, vol. 192, p. 111423, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222001303

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

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

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

[38] Q. Idrees Sarhan, B. Vancsics, and A. Beszedes, "Method calls frequency-based tie-breaking strategy for software fault localization," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 103–113.

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