

New Algorithms and Benchmarks for Supporting Spectrum-Based Fault Localization

Béla Vancsics

Department of Software Engineering
University of Szeged

Szeged, 2023

Supervisor:

Dr. Árpád Beszédes

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
OF THE UNIVERSITY OF SZEGED



University of Szeged
Ph.D. School in Computer Science

“Research is seeing what everybody else has seen and thinking what nobody else has thought.”

— Albert Szent-Györgyi

Preface

I never thought I would get this far... Until I started working as a BSc student at the Department of Software Engineering, I did not think about choosing a research career. I was always interested in new things and attracted by the joy of discovery, but a scientific career was a mystical world that seemed unattainable to me. Then, as I became more and more involved in research since 2012, it seemed that this is what I wanted to do. I liked that we examined a specific problem together from several aspects, tried to find a solution and if we succeeded, the team was happy with the result achieved. Of course, there were ups and downs, periods when the results did not come as we wanted, but there was always something (a new idea, a new challenge) or someone (one of my colleagues or my family) who pushed me through these stages. Complementing and motivating each other - I think I can say that we have achieved great success...

Two people have influenced my research career so far the most: Tibor Gyimóthy and Árpád Beszédes. I would like to thank Tibor for offering me the opportunity to join the department's team. In the last lecture of the semester, he asked me if I would like to work for them. I really liked the idea, there was no question on my part whether I should go for it. This was the point where my thesis really began... I am grateful to Árpád for constantly being my mentor over the past decade. Without his professional and human support, this thesis would certainly not have been possible. I learned a lot (and am still learning) from him, for which I am very grateful.

I thank the team for welcoming me so kindly and for being so patient and helpful when I joined them. I would also like to thank Tamás Gergely, Ferenc Horváth, Gergő Balogh, and Attila Szatmári, with whom I have worked for years and am still working. They are not only excellent colleagues but also excellent researchers who have continuously supported me during the research of the past years. Thanks for.

And, of course, I thank my family and my girlfriend for supporting me and trusting me all the way, for creating the conditions for me to focus only on my studies. I can not thank them enough for that! THANK YOU!

Béla Vancsics, 2023

Contents

Preface	iii
1 Introduction	1
2 Fault Localization Algorithms	3
2.1 Introduction	3
2.2 Related Work	4
2.2.1 Spectrum-based Fault Localization	4
2.2.2 Extending Hit-based Spectra	5
2.2.3 Call Stacks and Function Call Information	5
2.2.4 Count-based Fault Localization Algorithms	6
2.3 Overview	6
2.4 Hit-Based Fault Localization Algorithms	8
2.4.1 Hit-Based Spectra and Risk Formulae	8
2.5 Graph-based Fault Localization Algorithms	11
2.5.1 Neighbors-based Localization Algorithms (<i>NFL</i>)	11
2.5.2 Extended Neighbors-based FL (<i>ENFL</i>)	14
2.6 Unique Count-Based Spectra	16
2.6.1 Adapting the Basic Spectrum Metrics	18
2.6.2 Adapting the Risk Formulae	19
2.7 Weighted <i>NFL</i> (<i>WNFL</i>) and Weighted Extended <i>NFL</i> (<i>WENFL</i>) . . .	21
2.8 Empirical Evaluation	21
2.8.1 Subject Programs	21
2.8.2 Evaluation Metrics	22
2.9 Evaluation Results	25
2.10 Discussion	32
2.10.1 A Positive Example from the Benchmark	32
2.10.2 A Negative Example from the Benchmark	33
2.10.3 Time and Space Complexity	34
2.10.4 Threats to Validity	35
2.11 Conclusion	36
3 BugsJS: a Benchmark of JavaScript Bugs	37
3.1 Introduction	37
3.2 Studies on JavaScript analysis and testing	38
3.2.1 Bug-related Studies for JavaScript	39
3.2.2 Other Analysis and Testing Studies for JavaScript	40
3.2.3 Findings	41
3.3 BUGSJS – the Proposed Benchmark	42

3.3.1	Subject Systems Selection and Bugs Collection	42
3.3.2	Manual Patch Validation	44
3.3.3	Sanity Checking through Dynamic Validation	46
3.3.4	Benchmark Infrastructure and Implementation	47
3.3.5	BugsJS Dissection	49
3.4	Taxonomy of Bugs in BUGSJS	50
3.4.1	Manual Labeling of Bugs	50
3.4.2	Taxonomy Construction	51
3.4.3	Taxonomy Internal Validation	51
3.4.4	The Final Taxonomy	52
3.4.5	Analysis of Bug-Fixes Patterns	58
3.5	Discussion	61
3.5.1	Directing Research Efforts	61
3.5.2	Limitations	64
3.5.3	Threats to Validity	64
3.6	Potential Uses of Benchmark: Bug Localization	65
3.6.1	Overview	65
3.6.2	Results	67
3.7	Related Works	75
3.7.1	C, C++, and C# Benchmarks	75
3.7.2	Java Benchmarks	76
3.7.3	Multi-language Benchmarks	76
3.7.4	Benchmarks Comparison	76
3.8	Conclusions	77
Appendices		79
A Summary in English		81
B Magyar nyelvű összefoglaló		85
Bibliography		91

List of Tables

1.1	The main publications and how they relate to the thesis points	2
2.1	Hit-based spectrum (Cov^H) and spectrum metrics for the example. . .	10
2.2	Details of the hit-based SBFL formulae used in our experiment.	10
2.3	Hit-based example scores for the example.	10
2.4	Unique count-based spectrum (Cov^U) for the running example.	18
2.5	Hit and Unique count-based spectrum metrics for the running example.	19
2.6	Adapted <i>Russell-Rao</i> formulae using the unique count-based spectrum.	20
2.7	Suspiciousness scores of the methods (for the running example) calculated using the adapted <i>Russell-Rao</i> formulae in Figure 2.6.	20
2.8	Properties of subject programs	22
2.9	Basic statistics of formulae. Losses are in favor of the hit-based formulae	26
2.10	Basic statistics of formulae. Wins are in favor of the graph-based formulae	27
2.11	Absolute average ranks (wasted effort, E)	28
2.12	Improvement in average ranks (wasted effort, E) of unique count-based and graph-based formulae w.r.t. the hit-based results (negative values are in favor of the hit-based formula).	29
2.13	Results of the Wilcoxon signed-rank test of the unique count-based and graph-based ranks (p and d show the p -value and Cliff's Delta effect size, negative d values are in favor of the hit-based formula).	29
2.14	Accuracy (number of bugs in the Top-5 category) and enabling improvements achieved by the hit-based and unique count-based formulae. Percentages are shown w.r.t. the number of all bugs.	31
3.1	Subject distribution among surveyed papers	41
3.2	Subjects included in BUGSJS	43
3.3	Bug-fixing commit inclusion criteria	44
3.4	Manual and dynamic validation statistics per application for all considered commits	45
3.5	Bug-fixing change types (Pan et al. [91])	59
3.6	Bug-fixing patterns (<i>Pan</i> [91])	62
3.7	Average ranks	68
3.8	Top-N ranks	68
3.9	Overall bug-fix type statistics based on Tarantula	70
3.10	Number of labels (per metric)	72
3.11	Percents of labels (per metrics)	72
3.12	Significance in top-N based on Fisher's exact test	72
3.13	Number of labels in Top-N categories (T – <i>Tarantula</i> , O – <i>Ochiai</i> , D – <i>DStar</i>)	73

3.14	Significance in Top-N within the IF and SQ categories based on Fisher exact test	75
3.15	Properties of the benchmarks	77

List of Figures

2.1	Overview of the proposed approach, the evaluation and paper structure	7
2.2	Formal description of the binary (or hit) coverage matrix (Cov^H) and results vector (R)	8
2.3	Running example – program	9
2.4	Running example – test cases	9
2.5	Coverage graph	11
2.6	“Specificity of tests”	12
2.7	“Projection”	12
2.8	“Aggregation”	13
2.9	“Responsibility”	13
2.10	“Paired comparison”	14
2.11	“Averaging”	15
2.12	“Combining”	15
2.13	Dynamic call tree and the corresponding unique deepest call stacks (UDCS) collected during the execution of the <code>t1</code> test from the running example.	17
2.14	Ranks of the average ranks	30
2.15	Methods and tests related to the Math-103 bug	32
2.16	Methods and tests related to the Cli-19 bug	33
3.1	Overview of the bug selection and inclusion process.	42
3.2	Overview of BUGSJS architecture	47
3.3	BugsJS Dissection overview page	49
3.4	BugsJS Dissection page for one bug	50
3.5	Taxonomy of bugs in the benchmark of JavaScript programs of BUGSJS.	52
3.6	AS-CE bug-fixing pattern (bug: ESLint-11)	60
3.7	IF-CC bug-fixing pattern (bug: Pencilblue-6)	60
3.8	MC-DAP bug-fixing pattern (bug: Mongoose-22)	61
3.9	JS-Return bug-fixing pattern (bug: Karma-9)	61
3.10	Experiment overview	66
3.11	Bug labeling process	66
3.12	Fault Localization process	67
3.13	Interval of algorithms (<i>Tarantula</i> , <i>Ochiai</i> , and <i>DStar</i>)	69
3.14	Interval statistics (<i>Tarantula</i> , <i>Ochiai</i> and <i>DStar</i>)	71
3.15	Ochiai Ranks in IF labels	74
3.16	Ochiai Ranks in SQ labels	74

1

Introduction

There is no software without bugs, and detecting and fixing these faults is a resource-intensive process. For years, the number of research and industrial solutions that try to support this process with various automatic or semi-automatic solutions has been continuously increasing. One important part of the support is the development of appropriate fault localization algorithms and the integration of these algorithms into the workflow and/or the environment (*e.g.*: IDE plugin, various service-based solutions).

The algorithms with the largest technical literature are based on the connections between tests and methods that can be extracted by executing the tests (code coverage) and trying to use them to identify the location of the bug, thereby helping developers to fix the fault quickly and efficiently. In my thesis, I present the most important algorithms recognized by the literature and used as a reference, as well as describe the new kind of approach I have created. This new concept uses two “properties” of the code coverage data that have not been (or not extensively) used so far to increase efficiency: the (method) call frequency and the “(graph) adjacency relation”. I present the quantitative results and compare the performance that can be achieved with the state-of-the-art methods, as well as in cases where the two new characteristics/“properties” are used separately (frequency-based FL and unweighted graph-based FL) or together (weighted graph-based FL) in the algorithm.

However, in order to judge the efficiency of an algorithm objectively, we need appropriate benchmarks, which can be used to quantify the performance of different methods. In the second half of my thesis, I review the already accepted and used fault localization benchmarks and their capabilities, as well as the new data set (BugsJS) containing JavaScript programs created by us and the corresponding “features”. We prepared the taxonomy of the data set, analyzed the bug-fixing patches, and also used it to evaluate the fault localization algorithms, thereby presenting that it is suitable to serve as a basis for future studies.

Contributions

The ideas/concepts, figures, tables, and results included in this thesis were published in scientific papers (Table 1.1), except for the methods described in Section 2.7, which have not yet been published. The author is responsible for the following contributions:

I. Fault Localization Algorithms:

The author created graph-based fault localization concepts, implemented the (measurement)framework, performed and evaluated the experiment, visualized the examples and the results, and drew conclusions. The frequency-based fault localization concept is based on his idea; he implemented the framework and took part in the planning and coordination of the experiments, including the analyses and the visualization. He calculated the evaluation results (using numerous metrics) and together with the co-authors summarized the experiences. Combining the two concepts was the author's idea; he performed the necessary measurements, comparisons, and evaluation of the results.

II. BUGSJS: a Benchmark of JavaScript Bugs:

The author actively participated in the compilation of the BugsJS data set and played an important role in the development of the environment. He surveyed the already existing benchmarks and summarized their main characteristics, thereby helping to build the most relevant functions. Several analyses (*e.g.* creation of a taxonomy) were based on the fact that the participants in the research reached a consensus regarding the results, therefore, these are also part of the author's scientific contribution. The fault localization experiments conducted on the new data set were led by the author; he formulated the goals, planned the process, and also took an important role in the implementation and evaluation.

Table 1.1: The main publications and how they relate to the thesis points

№	[120]	[121]	[122]	[124]	[42]	[43]	[125]	[118]	[123]
I.	★	★	★	★					
II.					★	★	★	★	★

2

Fault Localization Algorithms

2.1 Introduction

Fault Localization (FL) is often a very difficult and time-consuming step during debugging. Therefore, its importance in software development and evolution is unquestionable. This part deals with a class of automated FL methods, which are based on the notion of the “program spectrum” [103, 19] (*Spectrum-Based Fault Localization-SBFL*). Spectrum refers to the statistical information about how the executed test cases relate to program elements and what their outcomes are (hence, Statistical Fault Localization is also a commonly used term for these methods) [29, 94, 24, 96, 6].

Several types of spectra have been defined over the past decades [46, 29] (for example program invariants spectrum [32, 106], and method calls sequence spectrum [21, 69], time spectrum [138]), but the most common approach is to use the so-called “hit-based” spectrum. This refers to the simple binary information if a code element (*e.g.*, statement or function in a procedural, or method in an object-oriented context) is covered during the execution of a test case or not. Using this information, the basic intuition is that those code elements that are exercised by comparably more failing test cases than passing ones are more likely to contain a fault, while non-suspicious elements are traversed mostly by passing tests. This information is usually captured by a set of basic *spectrum metrics* [24, 49], which count the number of passing and failing test cases that do or do not execute the code element in question, respectively. Dedicated *risk formulae* [2, 4, 3, 83, 139, 130, 71] are then used to calculate the *suspiciousness* levels by combining the spectrum metrics, which in turn *rank* the code elements to provide a debugging aid to the developer.

The traditional hit-based methods are generally seen as providing modest performance in terms of ranking precision [57, 135, 134, 95], but other issues have also been identified [60, 96, 116, 58], which contributes to the fact that automated fault localization is still ignored by the industry for the most part. Consequently, researchers proposed different approaches that go beyond the hit-based spectrum and utilize other information available that could help improve the overall ranking performance [141, 64, 30, 142, 67].

A straightforward extension of the hit-based spectrum is the consideration of other characteristics in formulas (for example relationship of tests to each other) or the *count-based* ones which record the number of executions of a particular element during runtime, and this way the binary spectrum is replaced by a spectrum with integer values. By comparison, count-based spectra are rarely investigated in the literature. Some early results have been published by Harrold et al. [47, 46], but more recently Abreu et al. [1] concluded that counts do not provide additional value compared to hits. There might be several reasons for this phenomenon, but a popular explanation is that repeating program elements many times during execution (due to loops) may lead to unwanted distortion in the test case statistics.

I would like to outline a “fault localization algorithm evolution”. First, I present a hit-based concept that uses a graph-based approach to increase the efficiency of the algorithms by considering new aspects. After that, I propose a method to improve hit-based spectra using an advanced count-based approach (which does not count all occurrences of a program element during execution but only those that occur in *unique call contexts*), and finally, I would evaluate the performance of the weighted graph-based procedure obtained by crossing the two methods.

2.2 Related Work

2.2.1 Spectrum-based Fault Localization

Automated fault localization techniques have been around for more than three decades. There have been several surveys written [29], [94], [6], [131], and various empirical studies performed [96], [146], [3] to compare the effectiveness of various methods. While there are other fault localization techniques as well [142, 18, 114, 93, 92, 66], SBFL emerged as one of the main approaches to Software Fault Localization.

Despite the immense literature, SBFL is still to find its way to be used in practice [60], [116], [3]. Often the faulty element is placed far from the top of the ranklist [134], and this way the developer will refuse to use any help for localizing bugs. Abreu et al. [4] made a study on how accurate the SBFL approaches are. They found that SBFL’s accuracy is highly dependent on the quality of test design.

The most universally used spectra are based on individual statements or methods [111, 107, 90]. Harold *et al.* [47] proposed several other types of program spectra, *i.e.*, counting branches, execution trace, execution path, etc., however, the hit-based approach remained the most studied one.

There is a plethora of different risk formulae proposed by researchers that use hit-based spectrum metrics (good summaries can be found in [49, 83]). Moreover, some researchers experimented with automatically deriving new formulae [139, 84].

There have been several surveys written [29, 94, 6, 131] on Fault Localization, and various empirical studies performed [96, 146, 3] to compare the effectiveness of various methods. Even though there are other emerging fault localization techniques in the research community [142, 18, 114, 93, 92, 66], SBFL still remains one of the main approaches to Software Fault Localization.

2.2.2 Extending Hit-based Spectra

The basic constituent of SBFL is code coverage information. The most universally used spectra are based on individual statements or methods [111], [107], [90].

Harold et al. [47] proposed several other types of program spectra, however, the hit-based approach remained the most studied one. Abreu et al. [1] performed an empirical study on the count spectra for Fault Localization using Barinel [3]. They compared it to classical SBFL algorithms, however, it did not improve the average ranks on real programs and bugs.

Classical SBFL algorithms are limited to locating faults in loops. Shu et al. [112] improved the Tarantula metric by extending it with counting the statement frequency. Likewise, our method tackles the same issue but in a different manner. Abreu *et al.* [1] raised the issue of repeating function calls distorting the count spectra, but they did not investigate this issue in detail. Harrold *et al.* [47, 46] investigated several types of program spectra on small programs and they concluded that there is a correlation between the differences of the spectrum and the occurrence of faults, however, they did not consider fault localization efficiency in general.

Lee et al. [63] proposed a new approach that improves SBFL by using frequency counts of test coverage. However, only counting the statement frequency can lead to distortion in the statistics. We use UDSCS-s to mitigate this issue.

Laghari et al. [59] proposed a heuristic for SBFL using call sequences to rank the classes. Their method can pinpoint 56% of the faulty classes of NanoXML in all test runs. Their approach filters out the repetitive method calls, which is similar to our approach.

2.2.3 Call Stacks and Function Call Information

Not many studies investigated call stacks and function call information in the context of fault localization. Beszédes et al. [15] used the Ochiai formula and expanded it with function call chains' context information. Function call chains and UDSCS-s are both based on dynamic call information, but while it is expensive to compute call chains for large programs, UDSCS-s are cheaper, hence our approach has better performance. Also, our method does not need expensive data structures such as call chain matrices.

Jiang et al. [53] used the stack trace to locate null pointer exceptions more efficiently. Gong et al. [39] generate stack traces to help localize crash faults. They were able to find 64% crashing faults in Firefox 3.6.

Zhu *et al.* [145] investigated whether function call sequences improve the efficiency of fault localization. There is empirical evidence that stack traces help developers fix bugs [108]. Furthermore, Zou *et al.* [146] showed that stack traces can be used to locate crash faults. Jiang *et al.* [53] used the stack trace to locate null pointer exceptions more efficiently. Gong *et al.* [39] generated stack traces to help localize crash faults and combined the generated passing and failing traces, thus creating spectra, which are used by the SBFL algorithm, *e.g.*, Ochiai (they were able to find 64% of crashing faults in Firefox 3.6.)

In an earlier version of our work [122], we extended the idea of call chains context, by investigating the Unique Deepest Call Stacks (UDSCS). We used five formulae that are using the same numerator: ef , *i.e.*, the number of failing test cases executing a function. We concluded that the call-frequency-based *Jaccard* formula is the most successful at localizing bugs among the other four formulae. This paper provides a more thorough

investigation of count-based spectra and gives a more general and complete overview of the topic.

2.2.4 Count-based Fault Localization Algorithms

Abreu *et al.* [1] performed an empirical study on the count spectra for Fault Localization using Barinel [3]. They compared it to classical SBFL algorithms, however, it did not improve the performance on real programs and bugs. The result was that their method can assist developers in finding the root cause of the discovered bugs. The reasons for that are:

- (i) Barinel is based on Bayesian probability which expects the failure probability to be an or-model
- (ii) the error is due to the choice of the test suite.

Lee *et al.* [63] proposed a new approach that improves SBFL by using frequency counts of test coverage. The authors evaluated their approach on several metrics (*e.g.*, Kulczynski2, Ochiai, Jaccard), however, they concluded that only counting the statement frequency can lead to distortion in the statistics. Laghari *et al.* [59] proposed a heuristic for SBFL using call sequences to rank the classes. Their method can pinpoint 56% of the faulty classes of NanoXML in all test runs.

2.3 Overview

In this section, new kinds of Spectrum-Based Fault Localization algorithms will be introduced building on existing concepts known as the hit-based approach. The proposed new methods are then empirically evaluated by comparing their fault localization capabilities to the traditional hit-based approaches used as a baseline. I first introduce the basic notations, data structures, and algorithms using an example, then the empirical evaluation is provided. Figure 3.10 shows the workflow of this thesis point in more detail. The three main phases are the following:

- (i) **Input data generation:** preparation of input data, *i.e.*, the different spectra required for algorithms using the production and the test code. I distinguished two types of spectra, the “classical” hit-based data are presented in more detail, while the count spectra containing the added information are described, one by one.
- (ii) **Fault localization algorithm:** at this stage of the process, I demonstrate the FL algorithms that rely on the different spectra, and quantify their performance. This procedure can take place on four threads, depending on the type of input data and the concepts:
 - (a) Hit-based FL: I determine the efficiency of the method using traditional SBFL risk formulae
 - (b) Unweighted graph-based FL: two new, unweighted graph-based algorithms will be presented that contain added information about the relation between the code elements

- (c) Frequency-based FL: this concept extends the traditional SBFL coverage matrix and spectra, these mappings redefine the spectrum metrics needed to execute the algorithm and redefine FL formulae using them
- (d) Weighted graph-based FL: these methods combine the features/aspects of unweighted graph-based and frequency-based algorithms, thus exploiting their combined potential

Eight traditional SBFL risk formulae and four different adaptations of them will be present, in addition, two unweighted and two weighted graph-based algorithms will be evaluated in later sections.

- (iii) **Evaluation:** It contains the description of our empirical evaluation with results. I compared the (new) FL algorithms to the hit-based approach and quantified the differences between the efficiencies, then I evaluated their effectiveness, using a number of evaluation metrics often used in FL research, complemented by significance testing.

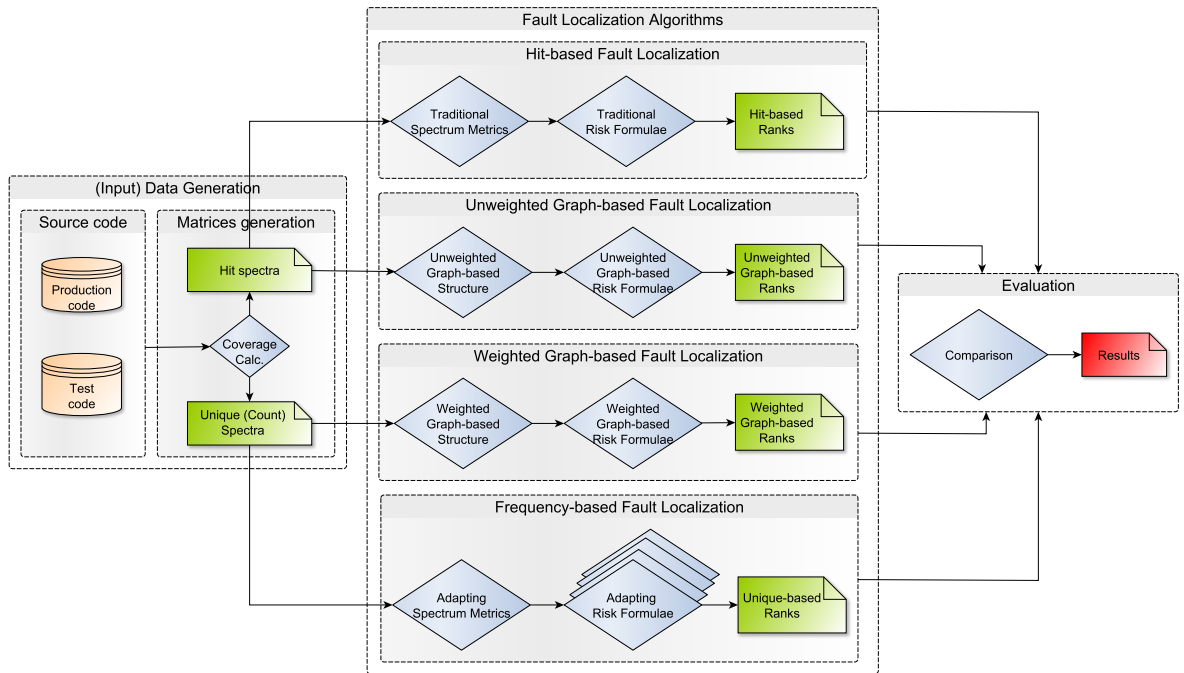


Figure 2.1: Overview of the proposed approach, the evaluation and paper structure

Note, that this research deals with function-level granularity for the analysis, which means that the basic element for localizing a fault is a function or a method of a class. Hence, all the discussion that follows will assume a basic code element to be a function or a method. This is a higher granularity than the currently prevalent statement-level approach, but it has at least two advantages. First, more global contextual information has been defined about the investigated program element, and it is scalable to large programs and executions. Furthermore, there are some views that method-level is a better granularity for the users as well [14, 146]. For the empirical assessment, we relied on the popular bug benchmark, often used in SBFL research, Defects4J [55].

2.4 Hit-Based Fault Localization Algorithms

There are several kinds of FL approaches based on the information they use. In this section, I present the Spectrum-based Fault Localization (SBFL) technique, which captures program execution data. SBFL is a dynamic program analysis technique that is performed through program execution. In SBFL, code coverage information (also called spectra) obtained from executing a set of test cases and test results are used to calculate the probability of each program entity (e.g., statements, blocks, or methods) being faulty.

2.4.1 Hit-Based Spectra and Risk Formulae

Hit-based SBFL uses a binary coverage matrix (noted by Cov^H) and a test results vector (noted by R) as the basic data structures to calculate the suspiciousness scores for program elements [4, 3]. The formal descriptions of these components are shown in Figure 2.2. In the coverage matrix, the columns represent the tests and the rows are the program elements, methods, or functions in our case. The value of an element in the Cov^H matrix is 1 or 0, depending on whether the method is covered by the test or not. An element in the results vector is 0 if the given test passed, otherwise, it is 1.

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

$$R = [R_{t_1} \dots R_{t_i}] \quad , \quad R_t = \begin{cases} 1 & \text{if } t \text{ failed} \\ 0 & \text{if } t \text{ passed} \end{cases}$$

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

Figure 2.2: Formal description of the binary (or hit) coverage matrix (Cov^H) and results vector (R)

Figure 2.3 and Figure 2.4 contain short code snippets which are adapted versions of the example from [15]. In this example, $\{a, b, f, g\}$ is the set of program elements (methods), and $\{t1, t2, t3, t4\}$ are the test cases. As can be seen, the four program elements are dependent on each other: method **a** calls methods **f** and **g** directly, while method **b** also calls **a** and **g** directly. We set the bug to be in method **g**, which is called by **a** and **b**. Tests **t1** and **t2** fail due to the bug, the other two tests are passing. The resulting coverage matrix and results vector are shown in Table 2.1 (leftmost matrix). This method is constructed to emphasize the importance of caller-callee relationships, different call contexts, and the frequency of method calls. More realistic examples that are actual bugs from the benchmark are shown in Section 2.10.

```

1 public class Example {
2     private int _x = 0;
3     private int _s = 0;
4     public int x() {return _x;}
5
6     public void a(int i) {
7         _s = 0;
8         if(i==0) return;
9         if(i<0) {
10             for(int y=0;y<=9;y++)
11                 f(i);
12         } else {
13             g(i);
14         }
15     }
16
17     public void b(int i) {
18         _s = 1;
19         if (i==0) return;
20         if (i<0) {
21             for(int y=0;y<Math.abs(i);y++)
22                 a(0);
23         } else {
24             for(int y=0;y<=1;y++)
25                 g(i);
26         }
27     }
28
29     private void f(int i) {
30         _x -= i;
31     }
32
33     private void g(int i) {
34         //should be _x+=i;
35         _x += (i+_s);
36     }
37 }
    
```

Figure 2.3: Running example – program

```

1 public class ExampleTest {
2     @Test public void t1() {
3         Example tester = new Example();
4         tester.a(-1);
5         tester.a(1);
6         tester.b(1);
7         tester.b(-8);
8         // failed
9         assertEquals(13, tester.x());
10    }
11
12    @Test public void t2() {
13        Example tester = new Example();
14        tester.a(1);
15        tester.b(1);
16        // failed
17        assertEquals(3, tester.x());
18    }
19
20    @Test public void t3() {
21        Example tester = new Example();
22        tester.a(1);
23        tester.b(0);
24        assertEquals(1, tester.x());
25    }
26
27    @Test public void t4() {
28        Example tester = new Example();
29        tester.a(-1);
30        tester.a(1);
31        tester.b(0);
32        assertEquals(11, tester.x());
33    }
34 }
    
```

Figure 2.4: Running example – test cases

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

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

The hit-based spectrum and the spectrum metrics can be seen in Table 2.1 for our running example in Figure 2.3.

I selected eight well-known risk formulae to experiment with, whose details are shown in Table 2.2. These formulae were chosen because their levels of effectiveness vary surprisingly as presented in the literature, and I wanted to experiment with the possibly most diverse set of algorithms. It can be observed that all the selected formulae rely on $|m_{ef}|$ in one way or the other since it is very straightforward that the suspiciousness of a program element is mostly affected by how many failing test cases are going through it, but the other metrics can also be found in many formulae.

Table 2.1: Hit-based spectrum (Cov^H) and spectrum metrics for the example.

		Hit spectrum (Cov^H)				Spectrum metrics			
		t1	t2	t3	t4	$ m_{ef} $	$ m_{ep} $	$ m_{nf} $	$ m_{np} $
Methods	a	1	1	1	1	2	2	0	0
	b	1	1	1	1	2	2	0	0
	f	1	0	0	1	1	1	1	1
	g	1	1	1	1	2	2	0	0
Results (R)		1	1	0	0				

Table 2.2: Details of the hit-based SBFL formulae used in our experiment.

$$\begin{aligned}
 \text{Barinel [2]: } & \frac{|m_{ef}|}{|m_{ef}| + |m_{ep}|} & \text{DStar [130]: } & \frac{|m_{ef}|^2}{|m_{ep}| + |m_{nf}|} \\
 \text{GP13 [139]: } & |m_{ef}| \cdot \left(1 + \frac{1}{2 \cdot |m_{ep}| + |m_{ef}|}\right) & \text{Jaccard [4]: } & \frac{|m_{ef}|}{|m_{ef}| + |m_{nf}| + |m_{ep}|} \\
 \text{Ochiai [4]: } & \frac{|m_{ef}|}{\sqrt{(|m_{ef}| + |m_{nf}|) \cdot (|m_{ef}| + |m_{ep}|)}} & \text{Russell-Rao [3]: } & \frac{|m_{ef}|}{|m_{ef}| + |m_{nf}| + |m_{ep}| + |m_{np}|} \\
 \text{Sørensen-Dice [71]: } & \frac{2 \cdot |m_{ef}|}{2 \cdot |m_{ef}| + |m_{nf}| + |m_{ep}|} & \text{Tarantula [54]: } & \frac{\frac{|m_{ef}|}{|m_{ef}| + |m_{nf}|}}{\frac{|m_{ef}|}{|m_{ef}| + |m_{nf}|} + \frac{|m_{ep}|}{|m_{ep}| + |m_{np}|}}
 \end{aligned}$$

The corresponding suspiciousness scores for each method in our example and for each risk formula are shown in Table 2.3. As can be seen, the buggy method (g) is hardly distinguishable from the other methods based on the suspiciousness scores. Two algorithms (*Barinel* and *Tarantula*) cannot distinguish between the methods, each with a score of 0.5. For the other fault localization algorithms, three methods have the same suspiciousness value (a, b, and g), *i.e.*, according to the procedures, these methods have the same chance of containing a defect.

Table 2.3: Hit-based example scores for the example.

Methods	<i>Barinel</i>	<i>DStar</i>	<i>GP13</i>	<i>Jaccard</i>	<i>Ochiai</i>	<i>Russell-Rao</i>	<i>Sørensen-Dice</i>	<i>Tarantula</i>
a	0.50	2.00	2.25	0.50	0.71	0.50	0.67	0.50
b	0.50	2.00	2.25	0.50	0.71	0.50	0.67	0.50
f	0.50	0.50	1.25	0.33	0.50	0.25	0.50	0.50
g	0.50	2.00	2.25	0.50	0.71	0.50	0.67	0.50

2.5 Graph-based Fault Localization Algorithms

In the following section, I would like to present a different approach to hit-based algorithms. The input parameters are the same as in the previous concept (coverage data and test results), but instead of a matrix, it uses a graph representation and uses the properties extracted from it to determine the location of the bug (and the new concept cannot be written with only the four spectrum metrics).

The coverage graph of the example program can be seen in Figure 2.5 where the tests and the methods are the nodes, and the $t \mapsto m$ edge exists if an $m \in M$ is covered by the $t \in T$, *i.e.* $Cov_{t,m}^H = 1$, otherwise it does not. In Figure 2.5, the red circles indicate the failed tests and the greens show the passes.

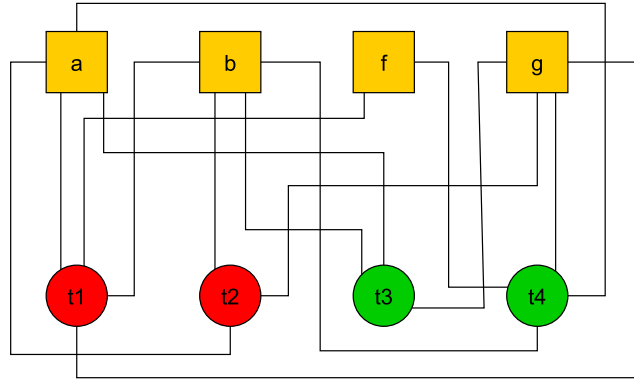


Figure 2.5: Coverage graph

2.5.1 Neighbors-based Localization Algorithms (*NFL*)

The algorithm consists of 4 steps and it examines the problem in terms of the two aspects of the relationship between tests and methods: the “method-specificity” of the failed tests (rewards methods that are strongly related to failed tests) and the “bug-specificity” of failed tests (failure tests — and thus indirectly the associated methods — that cover many methods are penalized).

NFL works as though the failed tests would “infect” the methods that cover them. If a method has many connections with failed tests and the proportion of these tests is high compared to the passed tests it is suspicious.

Every method has a start value (score=1.0). Then we continue the algorithm with the following steps:

1. “Specificity of tests” (Figure 2.6): The number of methods and number of covered methods are divided. If this value is high, it indicates that the failed t test only covers a small portion of the methods, *i.e.* the tests are “method specific”. I propagandize these values on the $t \mapsto m$, it will be the weight of the $t \mapsto m$ edge (where $t \in T_F$ and $m \in M$).

$$t \mapsto m = \frac{|M|}{|CM(t)|}$$

$$T_F = \{t | t \in T \wedge R_t = 0\}$$

$$CM(t) = \{m | m \in M \wedge Cov_{t,m}^H = 1\}$$

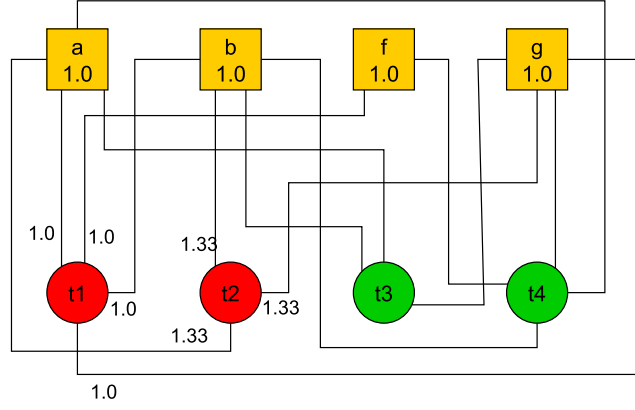


Figure 2.6: “Specificity of tests”

2. “Projection” (Figure 2.7): These values are projected to the method edges. It determines the average value of the $t \mapsto m$ edge weight. In other words, a method is covered by the more tests, the more its strength is “dispersed”.

$$t \mapsto m = \frac{t \mapsto m}{|CPT(m) \cup CFT(m)|}$$

$$T_P = \{t | t \in T \wedge R_t = 1\}$$

$$CFT(m) = \{t | t \in T_F \wedge Cov_{t,m}^H = 1\}$$

$$CPT(m) = \{t | t \in T_P \wedge Cov_{t,m}^H = 1\}$$

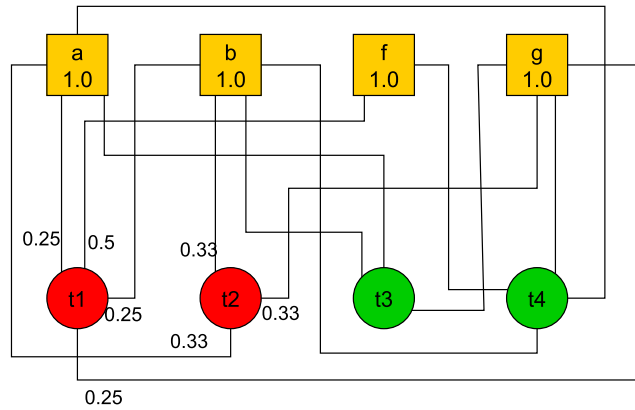


Figure 2.7: “Projection”

3. “Aggregation” (Figure 2.8): The $t \mapsto m$ edge weight will be aggregated into the scores of methods, thus expressing the specificity of the method.

$$NFL_m = 1 + \sum_{t \in CFT(m)} t \mapsto m$$

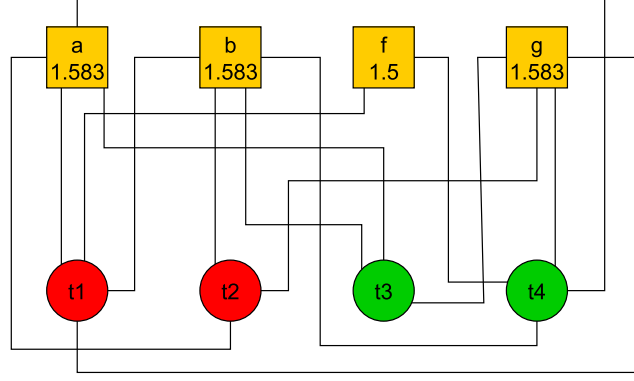


Figure 2.8: “Aggregation”

4. “Responsibility” (Figure 2.9): We calculate the ratio of the number of failed edges (from m) and the number of total failed edges. With this, we would like to capture the extent to which the method takes its share of the coverage of faulty tests.

$$NFL_m = NFL_m \cdot \frac{|CT(m) \cap T_F|}{|FE|}$$

$$CT(m) = \{t | t \in T \wedge Cov_{t,m}^H = 1\}$$

$$FE = \{t \mapsto m | t \in T_F \wedge m \in M \wedge Cov_{m,t}^H = 1\}$$

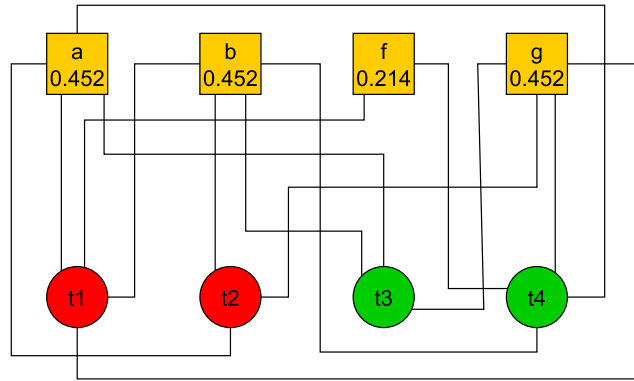


Figure 2.9: “Responsibility”

The four steps described above can be summarized in the following formula:

$$NFL_m = \frac{\left(1 + \sum_{t \in CFT(m)} \frac{\frac{|M|}{|CM(t)|}}{|CPT(m) \cup CFT(m)|}\right) |CT(m) \cap T_F|}{|FE|}$$

$$NFL_m = \left(1 + \sum_{t \in m_{ef}} \frac{|M|}{|CM(t)| \cdot (|m_{ef}| + |m_{ep}|)}\right) \cdot \frac{|m_{ef}|}{\sum_{m' \in M} |m'_{ef}|}$$

2.5.2 Extended Neighbors-based FL (ENFL)

The two aspects mentioned above can be supplemented by a relatively used test property: the “failed-test-specificity of methods”. In this approach, we reward methods that are more strongly associated with failed tests than passed ones, *i.e.* we examine the similarities/differences between the coverage of passed and failed tests. Incorporating this into the previous line of thought, the *NFL* algorithm was extended with three new steps.

1. “Paired comparison” (Figure 2.10): all passed-failed test pairs were examined and increased the *score_m* of those methods (*score_m* initially 0.0) that were only covered by the failed test. The (edge)weight of the dashed line between the tests shows how much to increase the score of the method that only covers the failed test of the test pair. The values in the octagon show these summarized weights.

$$score_m = \sum_{\substack{f_m \in CFT(m) \\ p_m \in NCPT(m)}} \frac{|CM(f_m) \setminus (CM(p_p) \cap CM(f_m))|}{|CM(f_m)|}$$

$$NCPT(m) = \{t | t \in T_P \wedge Cov_{m,t}^H = 0\}$$

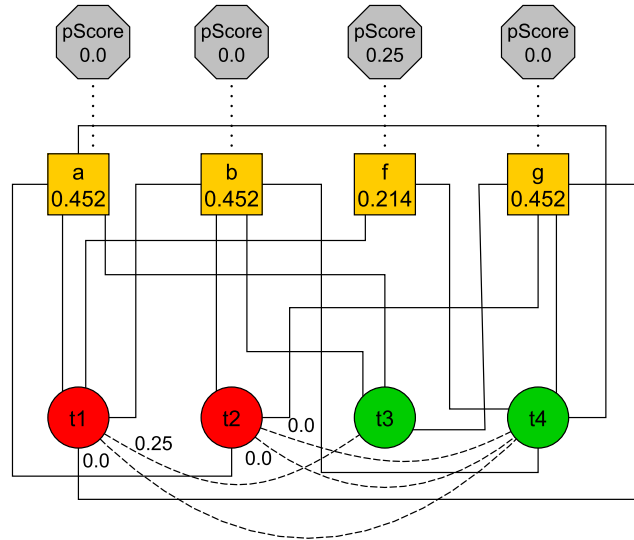


Figure 2.10: “Paired comparison”

2. “Averaging” (Figure 2.11): The value associated with the method is divided by the number of covered failed tests. (Note: in the example, only the *f* method has a non-0 score, but it covers only 1 failed test, so the final score will be the same as the value calculated in the previous step.)

$$score_m = \frac{score_m}{|CFT(m)|}$$

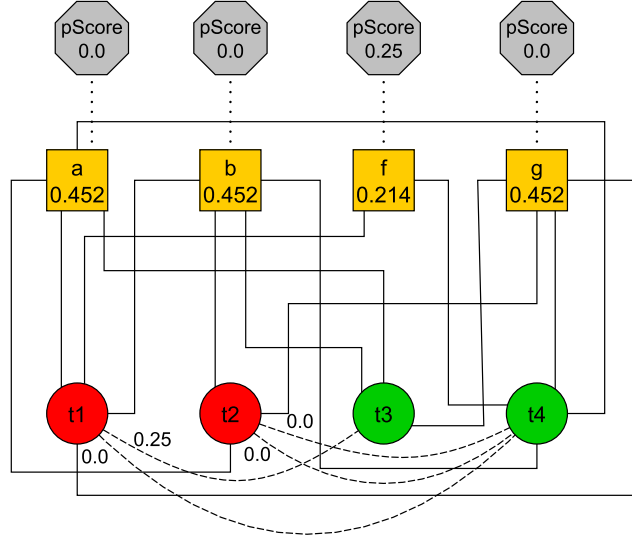


Figure 2.11: “Averaging”

3. “Combining” (Figure 2.12): Finally, we multiply the values given by *NFL* and the values obtained by comparing the test pairs, thus supplementing the previous (*NFL*) approach.

$$ENFL_m = NFL \cdot score_m$$

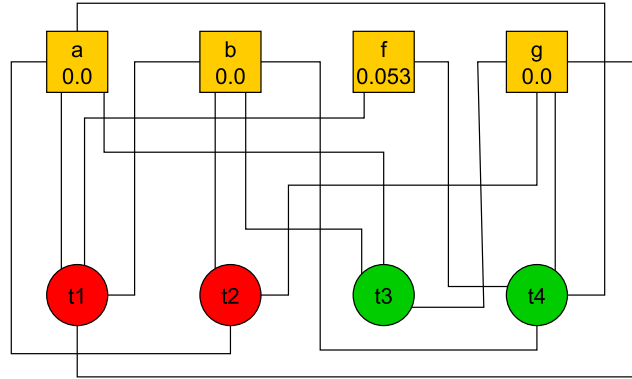


Figure 2.12: “Combining”

The seven steps described above can be summarized in the following formula:

$$ENFL_m = \left(1 + \sum_{t \in m_{ef}} \frac{|M|}{|CM(t)| \cdot (|m_{ef}| + |m_{ep}|)} \right) \cdot \frac{|m_{ef}|}{\sum_{m' \in M} |m'_{ef}|} \cdot \sum_{\substack{t_f \in m_{ef} \\ t_p \in m_{np}}} \frac{|CM(t_f) \setminus CM(t_p)|}{|CM(t_f)| \cdot |m_{ef}|}$$

$$ENFL_m = \left(1 + \sum_{t \in m_{ef}} \frac{\frac{|M|}{\sum_{m' \in M} |m'_{ef}|}}{|CM(t)| \cdot (|m_{ef}| + |m_{ep}|)} \right) \cdot \sum_{\substack{t_f \in m_{ef} \\ t_p \in m_{np}}} \frac{|CM(t_f) \setminus CM(t_p)|}{|CM(t_f)|}$$

The values thus obtained will be the basis of rank-order.

2.6 Unique Count-Based Spectra

Numerous researchers investigated the efficiency of traditional spectrum-based fault localization. However, using the hit coverage does not always lead to finding the faulty element. Thus, for tackling this problem, researchers [112, 15, 145, 1] used extra information to boost the algorithm’s performance. With the following two paragraphs, we want to confirm that the use of (call) context is good intuition and that it may be suitable to improve the efficiency of fault localization algorithms.

Such information is investigating the relationship between the code and the tests, then giving weights to them. Li et al. [65] approached the problem by weighting test cases based on how many blocks of code they had executed. They concluded that the fewer blocks executed by a given failing test case, the more SBFL’s efficiency enhances. Ren et al. [102] made a heuristic ranking on failing test cases to strengthen SBFL. They based their heuristic on the premise from slicing techniques that methods with large numbers of callers and callees in the call graph are more likely to be failure-prone than other changed methods.

Several studies have been presented that use (call context) information extracted from static and/or dynamic call graphs [141, 48, 143] or program slices [75, 142, 133, 132] in fault localization algorithms. By weighting the nodes of call graphs using some algorithm (*e.g.* Pagerank) and “expanding” the resulting information with the classic SBFL methods, their efficiency is increased. Pagerank uses the *caller* and *callee* relation of code elements, and new FL concepts transform this “structure” into the formulae. These methods are similar to the method we present in that they do not create a new formula but “redefine” existing ones using the newly added information and adapting spectrum metrics.

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

The traditional hit-based SBFL methods rely purely on local information about a program element’s (in our case, a method’s) coverage by the test cases, and no additional (contextual) information is leveraged about the element itself, nor the test cases. We use as an additional context, the frequency of the investigated method occurring in call stack instances and the number of invocations during the course of executing the test cases. Based on the findings of the graph- and slice-based papers mentioned above (*e.g.* [141, 48, 75, 142]), we conclude that if a method is called in many different contexts during a failing test case, it will more likely be accountable for the fault compared to other methods.

Contextual information can vary, from using slicing techniques to using other ranking methods, *e.g.*, Pagerank, one of which is using “call chains” to improve the efficiency of Fault Localization algorithms. Call chains are a set of function calls the test cases invoke, which is well-known to programmers who are debugging the code. This gives the opportunity to use more precise coverage data and can show whether a function fails when called from one place but does not when invoked from another. Call chains are harvested from stack traces, and there is strong evidence that using stack traces can help programmers fix bugs [108, 146].

Instead of the hit-based concept, in this approach, we rely on method call stack traces. In particular, we extract *unique deepest call stacks* – *UDCS*-s, which means that we build a particular instance of a call stack snapshot until additional methods

are transitively called, and stop when a method returns. This way, repeated invocations of methods from the same calling context (due to loops) will not induce new call stack instances.

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

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

To extract test results and stack-trace information from projects on a per-test level, we developed a tool¹ for measurements. This includes an online (on-the-fly) bytecode instrumentation tool, which we used to collect UDCS-s during test execution. During this process, probes are inserted into all methods that are relevant to the analysis. These probes are guarding every method’s entry and exit points i.e., they trigger every time the execution reaches a method call and when the execution leaves a method e.g., by reaching a *return* or a *throw* statement. During the execution of the test cases, the information provided by the probes is incorporated into a tree-like data structure, which ensures that unique stack traces are collected, and also minimizes memory consumption.

Unique count-based spectrum: UDCSs provide an opportunity to use coverage not only as “hit/no hit” data but to compute the frequency of methods occurring in such stacks and use this number in the SBFL risk formulae instead of the basic spectrum metrics introduced earlier. More precisely, for a method m under investigation, I calculate the sum of its frequencies occurring in different UDCSs generated by the relevant test cases.

Figure 2.13 shows the UDCSs generated by test case **t1** in our example: four method calls are made directly from the test, method **a** is called three times and **b** is called twice. The frequencies in the resulting unique deepest call stacks will provide the basis for our new approach. Table 2.4 shows the summarized call frequencies in the UDCSs for each method in the example.

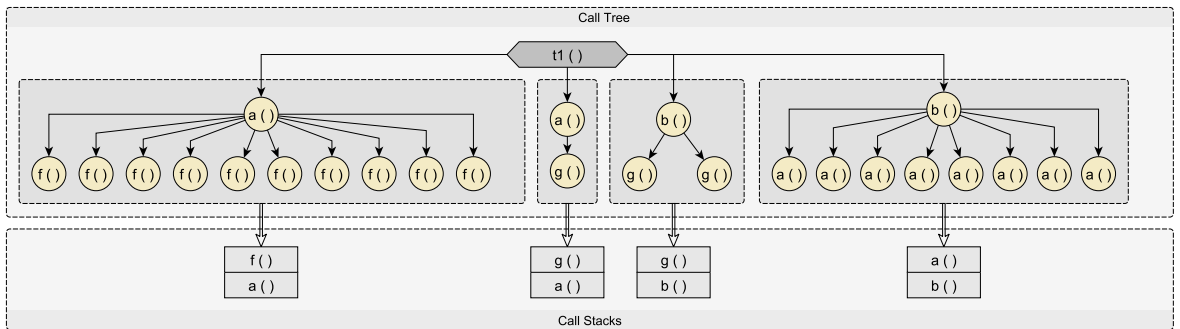


Figure 2.13: Dynamic call tree and the corresponding unique deepest call stacks (UDCS) collected during the execution of the **t1** test from the running example.

¹<https://github.com/sed-szeged/java-instrumenter>

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

Table 2.4: Unique count-based spectrum (Cov^U) for the running example.

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

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

2.6.1 Adapting the Basic Spectrum Metrics

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

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

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

My idea was to present the frequency information in the spectrum metrics, that is, our concept “rewards” the methods that appear multiple times on UDCS during the execution of a passed or failed test. This weighs the role of code elements in the execution process.

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

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

where M is the set of methods, M' is $M \setminus m$, and $c_{t,m}$ is an element of the Cov^U matrix. These two values indicate the average amount of coverage a method “loses” for passed and failed tests. This is a fairly intuitive way to calculate the values of non-covering metrics, but obviously, there are plenty of other options to define these, however, due

to space limitations, we are not investigating the effect of differently defined $C(m_{nf})$ and $C(m_{np})$ values in this paper.

Let us consider the count-based spectrum (Table 2.4) and method **f** of my example to illustrate the adapted spectrum metrics:

- $\mathbf{f}_{ef} = \{\mathbf{t1}\}$ and $C(\mathbf{f}_{ef}) = c_{\mathbf{t1},\mathbf{f}} = 1$: **t1** test calls **f** once
- $\mathbf{f}_{ep} = \{\mathbf{t4}\}$ and $C(\mathbf{f}_{ep}) = c_{\mathbf{t4},\mathbf{f}} = 1$: **f** is used once by **t4**
- $\mathbf{f}_{nf} = \{\mathbf{t2}\}$ and $C(\mathbf{f}_{nf}) = \frac{c_{\mathbf{t2},\mathbf{a}} + c_{\mathbf{t2},\mathbf{b}} + c_{\mathbf{t2},\mathbf{g}}}{|\{\mathbf{a},\mathbf{b},\mathbf{f},\mathbf{g}\}|-1} = \frac{1+1+2}{3} = 1.33$: “average coverage” of the failed **t2** not covering **f**
- $\mathbf{f}_{np} = \{\mathbf{t3}\}$ and $C(\mathbf{f}_{np}) = \frac{c_{\mathbf{t3},\mathbf{a}} + c_{\mathbf{t3},\mathbf{b}} + c_{\mathbf{t3},\mathbf{g}}}{|\{\mathbf{a},\mathbf{b},\mathbf{f},\mathbf{g}\}|-1} = \frac{1+1+1}{3} = 1$: “average coverage” of the passed **t3** not covering **f**

Table 2.5 shows the four spectrum metrics for the binary (hit-based) and the non-binary (unique count-based) spectra side by side to enable their comparison.

Table 2.5: Hit and Unique count-based spectrum metrics for the running example.

		Hit-based				Unique count-based			
		$ m_{ef} $	$ m_{ep} $	$ m_{nf} $	$ m_{np} $	$C(m_{ef})^U$	$C(m_{ep})^U$	$C(m_{nf})^U$	$C(m_{np})^U$
Methods	a	2	2	0	0	4	3	0	0
	b	2	2	0	0	3	2	0	0
	f	1	1	1	1	1	1	1.33	1
	g	2	2	0	0	4	2	0	0

2.6.2 Adapting the Risk Formulae

The statistics defined above can be adapted in the formulae in several ways, *i.e.*, they can be combined in different ways in the original formula. I examine whether the efficiency of algorithms can be increased using the count-based spectra and any of the following strategies:

Δ_{ef}^{*num} The $|m_{ef}|$ was changed in the numerator provided that it contains only $|m_{ef}|$ (this approach cannot be interpreted for the *GP13* and *Tarantula* formulae).

Δ_{ef}^{*} Each occurrence of $|m_{ef}|$ is overwritten with the new $(C(m_{ef}))$ value.

Δ_e^{*} The values of all occurrences of $|m_{ef}|$ and $|m_{ep}|$ are changed in the formula with the corresponding adapted values.

Δ_{all}^{*} The count-based matrix is used for the calculation of all four metric values in all their occurrences.

The results of the paper [122] were encouraging, but they can only be applied to formulas that have only $|m_{ef}|$ in their numerator. A logical and intuitive “continuation” of this line of reasoning is the case where we also replace the $|m_{ef}|$ value in the denominator with its adapted value (because the value of $|m_{ef}|$ greatly affects the efficiency of the algorithms). In addition to the $|m_{ef}|$ value, another spectrum metric that is also very influential and present in most formulas is $|m_{ep}|$, so we also examined the

scenarios when we changed the $|m_{ep}|$ values in the formulas to the adapted versions (in addition to $|m_{ef}|$), thus further enhancing the frequency usage. This approach already takes into account how often a given method appears in the UDCS of passed tests (for example, it will be able to differentiate between two elements with the same $C(m_{ef})$ value - thus reducing the chance of a tie [25]). I also examined the case where I used adapted versions of the metrics in the formula so that the (passed and failed) tests will also affect the suspiciousness value.

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

An example of this is shown in Table 2.6, which presents *Russell-Rao* using the count-based approach.

Table 2.6: Adapted *Russell-Rao* formulae using the unique count-based spectrum.

$$R_{ef}^{U,num}: \frac{C(m_{ef})^U}{|m_{ef}| + |m_{nf}| + |m_{ep}| + |m_{np}|} \quad R_{ef}^U: \frac{C(m_{ef})^U}{C(m_{ef})^U + |m_{nf}| + |m_{ep}| + |m_{np}|}$$

$$R_e^U: \frac{C(m_{ef})^U}{C(m_{ef})^U + |m_{nf}| + C(m_{ep})^U + |m_{np}|} \quad R_{all}^U: \frac{C(m_{ef})^U}{C(m_{ef})^U + C(m_{nf})^U + C(m_{ep})^U + C(m_{np})^U}$$

Table 2.7: Suspiciousness scores of the methods (for the running example) calculated using the adapted *Russell-Rao* formulae in Figure 2.6.

		Unique count-based			
		$R_{ef}^{U,num}$	R_{ef}^U	R_e^U	R_{all}^U
Methods	a	1.00	0.67	0.57	0.57
	b	0.75	0.60	0.60	0.60
	f	0.25	0.25	0.25	0.23
	g	1.00	0.67	0.67	0.67

Table 2.7 shows the suspiciousness values obtained by *Russell-Rao* with the techniques described above for unique count-based spectra. One of the most striking differences from the values in the hit-based approach is that the suspiciousness score values are typically higher. The highest value according to the hit-based approach was 0.5 (see Table 2.3, *Russell-Rao* column for reference), while for the unique count-based concepts, most of the methods scored 0.6 or higher. In addition, the number of ties is much smaller than in the case of the traditional algorithms.

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

2.7 Weighted *NFL* (*WNFL*) and Weighted Extended *NFL* (*WENFL*)

The concept presented in the previous section can be used to further develop graph-based methods, the logic of which is similar to the one according to which hit-based methods have been extended: I extend the spectrum metrics to an approach based on non-binary values and use these modified values (as the weight of the graph's edges) to calculate the formula.

The formula presented earlier (*NFL* in Section 2.5) can be extended to weighted graphs if the “building blocks” in the formula are calculated according to the new (Section 2.6) method (this means using the frequency and the adapted spectrum metrics to calculate the score). Based on this, the modified *NFL* formula changes as follows:

$$WNFLm = \left(1 + \sum_{t \in m_{ef}} \frac{|M|}{|CM(t)| \cdot (C(m_{ef})^U + C(m_{ep})^U)} \right) \cdot \frac{C(m_{ef})^U}{\sum_{m' \in M} C(m'_{ef})^U}$$

If we want to adapt the *ENFL* to the weighted graph, we need to “reinterpret” the *C* function (see Section 2.6.1). This *C* function was able to define the modified spectrum metrics for the methods, but now it is necessary to extend it to the tests as well. $C(m_{ef})$ and $C(m_{ep})$ (where $m_{ef} \in T$ and $m_{ep} \in T$) were practically the sums of the edge weights of the tests related to the method. This “direction” is easily reversed: function $C'(m)$ returns the weight of the edges between a given method and its associated tests.

$$C'(m) = \sum_{t \in CT(m)} c_{t,m}, \quad m \in M$$

With the help of the two weight-based functions ($C()$ and $C'()$), we can now transform the *ENFL* to try to determine the location of the bugs as accurately as possible.

$$WENFLm = \left(1 + \sum_{t \in m_{ef}} \frac{\frac{|M|}{\sum_{m' \in M} C(m'_{ef})^U}}{|CM(t)| \cdot (C(m_{ef})^U + C(m_{ep})^U)} \right) \cdot \sum_{\substack{t_f \in m_{ef} \\ t_p \in m_{np}}} \frac{\sum_{m' \in (CM(t_f) \setminus CM(t_p))} C'(m')}{\sum_{m'' \in CM(t_f)} C'(m'')}$$

2.8 Empirical Evaluation

The main goal of empirically evaluating the proposed approach was to compare the new algorithms' fault localization effectiveness. Also, I was interested in finding out which of the traditional SBFL formulae are best fitted to be extended with call frequency information. In this section, I overview the main parameters of the experiments: the benchmark used and the evaluation metrics, and I evaluate and summarize the results.

2.8.1 Subject Programs

I implemented my approach for analyzing Java programs, and for the evaluation, Defects4J was selected (v2.0.0),² a widely used collection of Java programs and curated

²<https://github.com/rjust/defects4j/tree/v2.0.0>

bugs in FL research. This benchmark contains seventeen open-source Java projects with manually validated, non-trivial real bugs.

The original dataset contains 835 bugs, however, there were cases that I had to exclude from the study due to instrumentation errors or unreliable test results. For many bugs, the modification only included a method addition (i.e., an existing method was not modified by the committers during the fix). A total of 786 defects were included in the final dataset. Table 2.8 shows each project and its main properties. The last two columns include the statistics about the UDCS-s generated by the test cases.

Table 2.8: Properties of subject programs

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

2.8.2 Evaluation Metrics

Comparing the effectiveness of SBFL algorithms means comparing the suspiciousness ranking lists, and how successfully they approximate the actual faulty code element. An algorithm is successful in locating the fault if the faulty element is at or near the first position in the rank list. But, in order to be able to compare the results of the algorithms their outputs need to be compatible. Because the suspiciousness score values are not necessarily produced in the same interval by the different formulae, their relative position in the ranking list is used instead. The position (rank) of the faulty method gives a good approximation of effectiveness because it indicates how many methods developers or testers need to examine before finding the bug. Note that, if there are multiple bugs for a program version, I will use the highest rank of buggy methods.

There are different ways to compare the ranking lists, and various research reports prefer one or the other. I followed several different approaches used earlier and also employed new ones. Thus, I believe this thorough evaluation can highlight all the different aspects of how successful each of the algorithms is in localizing faults.

In the case of comparing ranking lists, a particular issue is handling *ties*, that is, cases when two or more elements share the same score. Essentially, there are three ways to determine the ranks of such elements [136]:

- (i) the average of the ranks of the faulty methods,
- (ii) the minimum of the ranks, or
- (iii) their maximum.

In each case, the same value is assigned to all elements with the tied values. I used the average rank approach in the evaluation.

Wasted Effort

Equation 2.1 shows the *absolute average rank* calculation [4, 137], where i and f are methods, the latter being the faulty one, while s_i and s_f are the respective suspiciousness score values.

$$E(f) = \frac{|\{i | s_i > s_f\}| + |\{i | s_i \geq s_f\}| + 1}{2} \quad (2.1)$$

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

A “normalized approach” to the absolute average rank is the *EXAM score* [140] whose value is the quotient of the number of methods and the bug’s rank, expressed as a percentage. In other words, this metric describes the percentage of methods that need to be reviewed to find the location of the bug. The EXAM score and the absolute average rank follow from each other and result in the same order of efficiency, therefore, I discussed only the latter in my thesis.

Win-loss

A simple comparison is to look at how many times the algorithms improve effectiveness, that is, they result in a higher absolute average rank (E) (closer to 1) than other new algorithms, and how many times are the E values lower. For this, I borrowed a basic approach from game statistics. I calculate whether an algorithm wins or loses over another one based on the differences between the highest rank of faulty methods (Eq. 2.2).

Specifically, “loss” represents the number (percentage) of bugs for which the corresponding algorithm resulted in worse ranks than the others, while the “win” value reveals how many times the new approach performed better. Algorithms with more wins are better.

$$\begin{aligned}
 \text{win} &= |\{X | E_H^X > E_\varphi^X\}| \\
 \text{loss} &= |\{X | E_H^X < E_\varphi^X\}| \\
 X &\in \text{bugs of Defects4J} \\
 E_H^X &: E \text{ of } X \text{ using hit-based FL algorithm} \\
 E_\varphi^X &: E \text{ of } X \text{ using } \varphi \text{ algorithms} \\
 \varphi &\in \{\text{graph-based FL, unique count-based FL}\}
 \end{aligned} \tag{2.2}$$

Relative Improvements

When examining ranks, the relative position is not the only interesting thing to consider. The difference between the ranks is also important. However, the evaluation approach presented in the previous subsection does not take the latter aspect into account. To determine this difference, I calculate the *relative improvement (RI)* [15, 61], which gives the mean difference between E values and their normalized (NRI) values as well (Eq. 2.3).

$$\begin{aligned}
 RI &= E_H^X - E_\varphi^X \\
 NRI &= \frac{E_H^X - E_\varphi^X}{E_H^X} \cdot 100\% \\
 \varphi &\in \{\text{graph-based FL, unique count-based FL}\}
 \end{aligned} \tag{2.3}$$

Accuracy

Several studies report that developers only investigate the first 5 or 10 elements in the recommendation (rank-)list by fault localization algorithms before giving up using the ranking [134, 58]. Therefore, I distinguished between bugs where the minimum of faulty methods rank is less than or equal to five (*Top-5* - Eq. 2.4).

$$\begin{aligned}
 \text{Top-5}_\varphi &= |\{X | E_\varphi^X \leq 5\}| \\
 \varphi &\in \{\text{hit-based FL, graph-based FL, unique count-based FL}\}
 \end{aligned} \tag{2.4}$$

The family of similar metrics is commonly referred to as *Top-N* or *acc@N* [95]. This metric represents the number of successfully localized bugs within the top-n elements of the ranking lists. Higher values are better for this metric.

Enabling Improvements

A particularly interesting case of the difference between different algorithms considering the Top-N elements is when one approach produces a very low rank (*e.g.*, > 10 or > 5) while the other algorithm moves this to a higher Top-N category. As this brings a “new hope” that a bug could possibly be found by the user with the latter algorithm while it was very improbable with the former, I call these cases *enabling improvements* [15].

Besides enabling improvements I also report *deteriorations* (Eq. 2.5), *i.e.*, those cases where the algorithm produces a rank that is in a worse Top-N category than it was before. More enabling improvements (Eq. 2.5) could indicate a better algorithm, but the overall accuracy of the algorithm and the baseline should also be considered during the analysis.

$$\begin{aligned} \text{deteriorations} &= \left| \left\{ X | E_H^X \leq 5 \cap E_C^X > 5 \right\} \right| \\ \text{enabling improvements} &= \left| \left\{ X | E_H^X > 5 \cap E_C^X \leq 5 \right\} \right| \end{aligned} \quad (2.5)$$

Using the metrics defined above, I can investigate which of the newly adapted formulae perform best compared to the others, which can tell us what formulas are best fitted to be extended with call frequency information.

Significance Testing

The results for the ranking comparison will be checked by testing statistical significance. Usually, a Wilcoxon sign-rank test [20] is used for this. However, in the context of SBFL, the test could encounter ties, *i.e.*, when both approaches report the same rank for a function. To overcome this limitation, I used an implementation of the Wilcoxon test which copes with the ties by discarding the zero-differences. In addition, I complement the Wilcoxon test with the Cliff’s Delta (d) effect size measure [41]. Since there is no consensus in the literature about how the magnitude of the effect size should be determined, I only report the d values. Note that typical thresholds are $d < 0.147$ “negligible”, $d < 0.33$ “small”, $d < 0.474$ “medium”, otherwise “large”, but $d < 0.1$, $d < 0.3$, $d < 0.5$ is used as well.

2.9 Evaluation Results

In this section, I present the results of my evaluation according to the measures described in Section 2.8. This will enable quantitative comparisons and objective judgments of the proposed formulae in terms of fault localization effectiveness. Note that some data is absent from some tables or figures because the actual variant of the particular formula cannot be interpreted.

First, I investigate whether the formulae were able to improve effectiveness. A comparison of each formula with respect to the hit-based formula in terms of the win-loss measure is shown in Table 2.9 and Table 2.10, where the *lose* column shows in how many cases the “classic” SBFL resulted in a lower rank than the graph-based or count-based algorithm.

If we compare only the hit-based methods with the graph-based algorithms (columns for *NFL* and *ENFL* in Table 2.9), we can conclude that except for one case (*GP13* vs. *NFL*), the graph-based method resulted in a rank that is several times lower than the hit-based approach (win - lose > 0). Furthermore, it can be seen that the *ENFL* outperforms the SBFL method in almost twice as many cases as the *NFL*, and it is spectacular how many times the new concept results in a lower rank than the *Russell-Rao* (662 and 663, which is close to 85%). The *ENFL* has a “winning percentage” between 28% and 36% compared to the other eight formulas.

We can observe 4 different patterns emerging from the results based on the count-based results (columns Δ_{efnum}^U , Δ_{ef}^U , Δ_e^U , Δ_{all}^U in Table 2.9). We can see that all variants

of *Russell-Rao* have a huge advantage over the hit-based formula. The opposite is shown in row T(arantula) where the hit-based Tarantula outperforms the unique count approaches (win - lose < 0) and G(P13) has very similar results. Despite the fact that it achieves 270-275 wins, the hit-based approach still has an advantage over it. Finally, all variants of B(arinel), J(accard), and S(ørensen-Dice) have an advantage in all settings. *Ochiai* and *DStar* are also similar to *Barinel*, *Jaccard*, and *Sørensen-Dice*, however, D_{efnum}^U , D_e^U and O_{efnum}^U are behind the hit-based formulae. Overall, it seems that the Δ_{ef}^U and Δ_e^U variants are the most beneficial configuration in this setting.

In the case of weighted graph-based procedures, several interesting things can be seen in columns *WNFL* and *WENFL*. These procedures rarely give the same results as SBFL algorithms (win + lose is greater than 650, this is more than 80% of the bugs), *Russell-Rao* is still “out of line” (I improved this result 641 and 693 times - 81% and 88%), overall, *WNFL* performs several times worse than hit-based procedures (except for *Russell-Rao*), but considering all algorithms, *WENFL* outperforms SBFL algorithms the most as can be seen (the highest win values are bold in Table 2.9).

Table 2.9: Basic statistics of formulae. Losses are in favor of the hit-based formulae

	<i>NFL</i>		<i>ENFL</i>		Δ_{efnum}^U		Δ_{ef}^U		Δ_e^U		Δ_{all}^U		<i>WNFL</i>		<i>WENFL</i>	
	lose	win	lose	win	lose	win	lose	win	lose	win	lose	win	lose	win	lose	win
<i>B</i>	56	150	88	286	293	346	252	334	249	294	249	294	355	316	257	384
<i>D</i>	57	120	94	252	361	303	361	303	306	348	307	351	372	294	267	365
<i>G</i>	90	54	121	217	-	-	428	270	435	275	435	275	358	309	262	372
<i>J</i>	49	131	82	266	295	346	269	342	265	299	262	301	363	306	262	375
<i>O</i>	45	111	82	243	366	303	273	339	260	296	271	292	370	299	266	365
<i>R</i>	90	662	91	663	184	541	182	539	119	631	160	599	116	641	64	693
<i>S</i>	49	131	82	266	300	344	269	342	265	299	262	301	363	306	262	375
<i>T</i>	56	149	88	285	-	-	104	51	492	117	503	112	355	316	257	384

Table 2.10 contains a comparison of the results of count-based and weighted graph-based formulas. We can see that *WNFL* performs better in some cases than count-based procedures (G_{ef}^U , G_e^U , G_{all}^U , R_{efnum}^U , R_{ef}^U , T_{ef}^U , and T_e^U), but in most cases, the unique count-based algorithm gives better results (i.e. lose > win).

However, if we take the results of *WENFL* as a basis, we find that it gives better results in all cases than any count-based configuration, i.e. win > lose in all cases. The “most intense competition” was for O_e^U , where the difference between wins and losses was 86, which is 11% of the total data set.

The above results show that, in general, non-hit-based methods perform better than hit-based algorithms, and *WENFL* gives the best results (compared to the other algorithms). However, this is only one approach to evaluating the results, so further examinations are needed to reach reliable conclusions.

Table 2.10: Basic statistics of formulae. Wins are in favor of the graph-based formulae

		<i>Barinel</i>				<i>DStar</i>				<i>GP13</i>			<i>Jaccard</i>			
		Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U
<i>WNFL</i>	lose	385	390	358	358	306	306	387	387	114	135	135	386	394	357	359
	win	173	199	324	324	104	104	187	195	425	462	462	158	178	315	315
<i>WENFL</i>	lose	94	123	265	265	126	126	208	209	133	139	139	91	120	266	267
	win	302	326	378	378	410	410	309	317	512	519	519	290	314	366	369

		<i>Ochiai</i>				<i>Russell-Rao</i>				<i>Sørensen-Dice</i>				<i>Tarantula</i>		
		Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U
<i>WNFL</i>	lose	273	392	360	360	140	141	367	305	379	394	357	359	348	282	282
	win	90	173	308	313	537	534	245	294	147	178	315	315	326	423	424
<i>WENFL</i>	lose	125	118	270	272	156	156	250	206	105	120	266	267	249	185	182
	win	426	307	356	360	557	556	374	395	322	314	366	369	391	496	502

The approach mentioned above ignores an important aspect, namely the quantified value of ranks. This shortcoming is filled by examining the (absolute) average ranks. Table 2.11 shows the arithmetic means of the overall absolute average rank (wasted effort, E) values for each algorithm. Best results are highlighted in bold, and the hit-based results are included in the column “hit”. The results were visualized: Figure 2.14 shows how the rankings of performances (based on E) for each formula have evolved. In addition, Table 2.12 shows the relative change between the hit-based and other formulae. Column “Diff.” is the absolute difference, while column “Impr.” is relative to the corresponding hit-based formula. Furthermore, Table 2.13 helps in addressing the comparison from a statistical point of view. Columns noted with p represent the result of the Wilcoxon signed-rank tests (statistically significant values are highlighted in bold). The Cliff’s Delta effect sizes are in column d (negative values are in favor of the hit-based formulae).

In this data, we can recognize similar patterns to what we have seen previously. Generally, graph-based methods performed significantly better than hit-based algorithms. When we examine the unique count concepts, the variants of *Russell-Rao* achieve huge improvements: 65.2-100.8 ranks (47-74%) on average on the whole dataset. Despite the fact that these are statistically significant differences with medium to large effect sizes, due to the poor performance of the hit-based *Russell-Rao* formula the best result (35.12) of R_e^U is only around the average of the better hit-based approaches (33.98-33.99). Considering *Tarantula*, we also have statistically significant results, however, the unique count-based approaches cannot improve the hit-based results, the least problematic variant is T_{ef}^U which is behind the original *Tarantula* by 0.9 (2%). The unique count variants of *Barinel*, *Jaccard*, and *Sørensen-Dice* achieve about a 31-32% improvement with the Δ_{ef}^U configuration. Different variants of *DStar* and *Ochiai* have a 14-28% advantage over the hit-based versions. Overall, the best result (comparing the SBFL and count-based concepts) are 24.30 - $Ochiai_{ef}^U$, 24.40 - $Jaccard_{ef}^U$ and

$Sørensen-Dice^U_{ef}$, 24.55 - $Barinel^U_{ef}$, and 29.08 - $DStar^U_e$.

Using the graph-based methods, the wasted effort of $ENFL$ (34.05) is close to the best SBFL result (with -0.1 diff and 0% improvement), and the NFL produced medium performance. The results of the weighted graphs are mixed: $WNFL$ performed poorly, producing worse results than most other methods, but $WENFL$ achieved remarkably low wasted effort (20.59) and the improvements are between 39% and 84%. In view of the above, it is not surprising that there was a significant difference between it and the hit-based outcomes.

Table 2.11: Absolute average ranks (wasted effort, E)

	hit	NFL	$ENFL$	Δ_{efnum}^U	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	$WNFL$	$WENFL$
B	36.01			24.68	24.55	38.63	38.63		
D	33.99			35.60	35.60	29.08	29.13		
G	43.30			-	66.73	67.19	67.19		
J	36.06	36.42	34.05	24.63	24.40	38.64	38.34	47.19	20.59
O	33.98			36.05	24.30	36.44	36.99		
R	135.96			70.80	70.60	35.12	54.95		
S	36.06			24.89	24.40	38.64	38.34		
T	36.01			-	36.87	85.35	74.56		

The heatmap (Figure 2.14) shows where the algorithms are ranked in terms of performance, that is, what their rank of waster effort is. The colors range from green (lowest rank) - to yellow (medium rank) - to red (highest rank) scale, and the values in the cells show the ranks of the absolute average rank. For example, G_{efnum}^U gives the sixth-best result out of all the scenarios examined. From Figure 2.14 we can conclude that $WNFL$ gives the lowest average rank on the data set, followed by O_{ef}^U , G_{ef}^U , and S_{ef}^U . Interestingly, both NFL (22nd) and $ENFL$ (13th) perform better in this respect than $WNFL$ (33rd), and all three fall short of the performance of $WENFL$. It can be concluded that both “techniques” used by $WENFL$ (weights and test pairs analysis) provide important information for fault localization. The Δ_e^U and Δ_{all}^U concepts and the formulas based on $Sørensen-Dice$ perform the worst, although in some cases these performed relatively well, *e.g.* D_e^U .

In Table 2.14, we can see the number of bugs belonging to the Top-5 category (with the respective percentages), accumulated for the whole benchmark, for each unique count-based and hit-based formula. In the table, the values that achieved the best result according to the given concept are marked in bold, eg: for Δ_e^U , 391 bugs had a rank of 5 or less (this was achieved with the $DStar$ formula), the smallest deterioration (6) and the largest improvement (275) resulted with $Russell-Rao$ using the Δ_e^U concept. The values that gave the best value for the given formula are underlined. For example, when examining $Barinel$, we compared 9 results (1 hit-based, 4 count-based, 4 graph-based) and found that $WENFL$ gives the highest Top-5 value (389) and percentage (49.5%), the largest improvement (101), and the NFL determines the least deterioration (14).

Every unique count-based formula achieves enabling improvements, but there are about the same number of deteriorations as well. On average, the number of enabling

Table 2.12: Improvement in average ranks (wasted effort, E) of unique count-based and graph-based formulae w.r.t. the hit-based results (negative values are in favor of the hit-based formula).

Var.	Barinel		DStar		GP13		Jaccard		Ochiai		Russell-Rao		Sørensen-Dice		Tarantula	
	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.	Diff.	Impr.
NFL	-0.4	-1%	-2.4	-7%	6.9	15%	-0.4	-1%	-2.4	-7%	99.5	73%	-0.4	-1%	-0.4	-1%
$ENFL$	2.0	5%	-0.1	0%	9.2	21%	2.0	5%	-0.1	0%	101.9	74%	2.0	5%	2.0	5%
Δ_{ef}^{num}	11.3	31%	-1.6	-4%	-	-	11.4	31%	-2.1	-6%	65.2	47%	11.2	31%	-	-
Δ_{ef}^U	11.5	31%	-1.6	-4%	-23.4	-54%	11.7	32%	9.7	28%	65.4	48%	11.7	32%	-0.9	-2%
Δ_{ef}^e	-2.6	-7%	4.9	14%	-23.9	-55%	-2.6	-7%	-2.5	-7%	100.8	74%	-2.6	-7%	-49.3	-136%
Δ_{all}^U	-2.6	-7%	4.9	14%	-23.9	-55%	-2.3	-6%	-3.0	-8%	81.0	59%	-2.3	-6%	-38.6	-107%
$WNFL$	-11.2	-31%	-13.2	-38%	-3.9	-9%	-11.1	-30%	-13.2	-38%	88.8	65%	-11.1	-30%	-11.2	-31%
$WENFL$	15.4	42%	13.4	39%	22.7	52%	15.5	42%	13.4	39%	115.4	84%	15.5	42%	15.4	42%

Table 2.13: Results of the Wilcoxon signed-rank test of the unique count-based and graph-based ranks (p and d show the p -value and Cliff's Delta effect size, negative d values are in favor of the hit-based formula).

Var.	Barinel		DStar		GP13		Jaccard		Ochiai		Russell-Rao		Sørensen-Dice		Tarantula	
	p	d	p	d	p	d	p	d	p	d	p	d	p	d	p	d
NFL	< 0.001	0.029	< 0.001	0.01	0.008	0.02	< 0.001	0.021	< 0.001	0.01	< 0.001	0.51	< 0.001	0.021	< 0.001	0.029
$ENFL$	< 0.001	0.039	< 0.001	0.02	< 0.001	0.029	< 0.001	0.031	< 0.001	0.02	< 0.001	0.522	< 0.001	0.031	< 0.001	0.039
Δ_{ef}^{num}	0.002	0.045	0.011	-0.056	0.003	0.042	-	-	0.006	-0.064	< 0.001	0.29	0.009	0.034	-	-
Δ_{ef}^U	< 0.001	0.047	0.011	-0.056	< 0.001	-0.193	< 0.001	0.046	0.004	0.037	< 0.001	0.291	< 0.001	0.046	< 0.001	-0.004
Δ_{ef}^e	0.09	0.01	0.049	0.043	< 0.001	-0.197	0.152	0.013	0.143	0.013	< 0.001	0.514	< 0.001	0.152	< 0.001	-0.19
Δ_{all}^U	0.09	0.01	0.054	0.04	< 0.001	-0.197	0.108	0.01	0.389	0.006	< 0.001	0.446	< 0.001	0.108	< 0.001	-0.189
$WNFL$	0.009	-0.06	< 0.001	-0.078	0.039	-0.065	0.002	-0.067	< 0.001	-0.078	< 0.001	0.431	0.002	-0.067	0.009	-0.06
$WENFL$	< 0.001	0.09	< 0.001	0.07	< 0.001	0.079	< 0.001	0.082	< 0.001	0.07	< 0.001	0.578	< 0.001	0.082	< 0.001	0.09

	hit	NFL	ENFL	$\Delta_{ef}^{U_{num}}$	Δ_{ef}^U	Δ_e^U	Δ_{all}^U	WNFL	WENFL
B	17,5	22	13	7	5	28,5	28,5	33	1
D	12			15,5	15,5	9	10		
G	32				35	36,5	36,5		
J	20,5			6	3,5	30,5	26,5		
O	11			19	2	23	25		
R	42			39	38	14	34		
S	20,5			8	3,5	30,5	26,5		
T	17,5				24	41	40		

Figure 2.14: Ranks of the average ranks

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

Examining graph-based approaches, we can see that the number of Top-5 bugs is high for *NFL* and *ENFL* (larger than for any SBFL formula) and the values for deterioration and improvement are relatively low, which matches the win-lose outcomes: we can improve the outcome in a few cases (13-26 bugs, 1.6 % - 3.3 %), that is, reduce the original rank of more than 5 to 5 or less. However, “demotion” is not common either, it occurs in 0.3%-2.4% of bugs (in 2-19 cases) *Russell-Rao* is the exception, in which case the improvement is 30% (259-260 bugs).

Focused on weighted graph-based methods, we can see that *WNFL* does not perform very well: accuracy is low (41.5%), and in most cases the difference between improvement and deterioration is negative, meaning we make it deteriorate more. However, *WENFL* produced particularly good results: the highest accuracy (389 bugs - 49.5%) and a high improvement value (which is always the highest value except for *DStar* when examining the “formula dimensions”). The value of negative change cannot be said to be low, but even so, it improves in more cases than deteriorates (22-32 bugs, 2.8% - 4.1% - except for *Russell-Rao* because it produced extreme values in this case as well).

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

Table 2.14: Accuracy (number of bugs in the Top-5 category) and enabling improvements achieved by the hit-based and unique count-based formulae. Percentages are shown w.r.t. the number of all bugs.

Formula	hit		Δ_{ef}^U			Δ_e^U			Δ_{all}^U						
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	
<i>Barinel</i>	357	(45.4%)	373	(47.5%)	78	94	376	(47.8%)	65	84	354	(45.0%)	45	42	
<i>DStar</i>	366	(46.6%)	327	(41.6%)	128	89	327	(41.6%)	128	89	391	(49.7%)	86	111	
<i>GP13</i>	361	(45.9%)	-	-	-	-	266	(33.8%)	172	77	269	(34.2%)	170	78	
<i>Jaccard</i>	358	(45.5%)	372	(47.3%)	81	95	380	(48.3%)	69	91	357	(45.4%)	47	46	
<i>Ochiai</i>	367	(46.7%)	323	(41.1%)	135	91	380	(48.3%)	74	87	363	(46.2%)	51	47	
<i>Russell-Rao</i>	111	(14.1%)	249	(31.7%)	12	150	248	(31.6%)	12	149	380	(48.3%)	6	275	
<i>Sorensen-Dice</i>	358	(45.5%)	366	(46.6%)	87	95	380	(48.3%)	69	91	357	(45.4%)	47	46	
<i>Tarantula</i>	357	(45.4%)	-	-	-	-	351	(44.7%)	12	6	258	(32.8%)	111	12	
														117	14
Formula	hit		<i>NFL</i>			<i>ENFL</i>			<i>WNFL</i>			<i>WENFL</i>			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	
<i>Barinel</i>	357	(45.4%)			<u>14</u>	26			15	26			127	96	
<i>DStar</i>	366	(46.6)%			<u>12</u>	15			13	15			133	93	
<i>GP13</i>	361	(45.9)%			<u>16</u>	24			19	26			132	97	
<i>Jaccard</i>	358	(45.5)%			<u>10</u>	21			11	21			128	96	
<i>Ochiai</i>	367	(46.7)%	369	(46.9%)	<u>11</u>	13	368	(46.8%)	12	13	326	(41.5%)	134	93	
<i>Russell-Rao</i>	111	(14.1)%	2	260	<u>2</u>	259	2	259	2	259	6	221	4	282	
<i>Sorensen-Dice</i>	358	(45.5)%			<u>10</u>	21			11	21			128	96	
<i>Tarantula</i>	357	(45.4)%			<u>14</u>	26			15	26			127	96	

2.10 Discussion

I hypothesized that there is a correlation between the number of calls from different contexts and the efficiency of fault localization algorithms. To better support this idea, I present one positive (Section 2.10.1) and one negative (Section 2.10.2) example from Defects4J, and analyze the cases in which the new approach could or could not be an effective solution. The time and space complexity of the algorithms is crucial for usability, so I overview the costs and the resource requirements in Section 2.10.3 and discuss threats to validity in Section 2.10.4.

2.10.1 A Positive Example from the Benchmark

To understand how the proposed approach achieves such improvements, we manually analyzed (together with my colleagues – thereby increasing reliability/trust) several bugs from the Defects4J dataset. One interesting case we looked at was bug 103 from the *Commons Math* project. Figure 2.15 visualizes the test-to-code relations of this bug schematically.

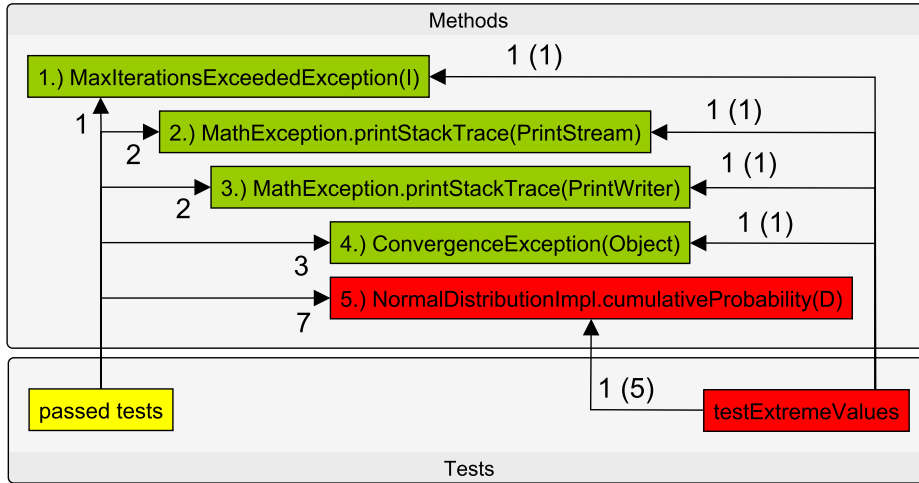


Figure 2.15: Methods and tests related to the Math-103 bug

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

In the case of the hit-based approach, *Jaccard*'s scores of these methods (from 1st to 5th by rank) are $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{3}$, $\frac{1}{4}$, and $\frac{1}{8}$. (Other formulas produce different scores in this case but the ranks are the same.) As can be seen, the traditional approach emphasizes

the exception-handling parts of the code, which are covered by the failing test but are otherwise unrelated to the actual bug. However, the proposed formulae incorporate the frequency values into their numerator, which emphasizes the importance of the relationship between the failing tests and a particular method. In the case of this bug, *testExtremeValues* calls *cumulativeProbability* directly and repeatedly in a loop. Then, *cumulativeProbability* calls several other utility methods to calculate the probability. The loop is executed until the calculated probability reaches an extremely low or high value or an exception is thrown. As a result, *cumulativeProbability* appears 5 times in 5 different UDCSs, while other related methods appear fewer times, hence the frequency-based approach can distinguish *cumulativeProbability* based on the additional contextual information that is provided by the UDCSs. Note that a simple count-based algorithm would yield the same scores for those methods that are called in the aforementioned loop.

2.10.2 A Negative Example from the Benchmark

We also looked for examples where the hit-based algorithm outperforms the count-based approaches. Bug 19 from the *Cli* project is one of these cases. Its “cleaned” structure is shown in Figure 2.16 in a similar fashion to Figure 2.15. There is one failing test (*testUnrecognizedOption2*) and a myriad of passing tests in this scenario. The faulty method is *processOptionToken*. However, the most suspicious element according to the hit-based algorithm is *burstToken*, and the first item on the unique count-based suspiciousness list is *parse*.

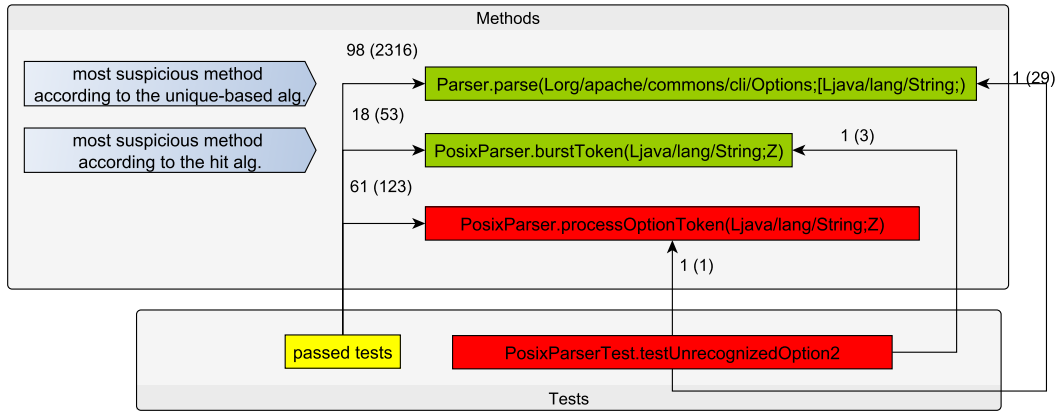


Figure 2.16: Methods and tests related to the Cli-19 bug

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

If we consider the adapted spectrum metrics, then the weight between the faulty method and the failed test remains 1, but for the other two methods, it becomes 3 and

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

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

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

2.10.3 Time and Space Complexity

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

The hit procedures, *NFL*, and *ENFL* store binary data (whether a test covers a method or not) therefore, we can store the necessary information in an $N \times M$ -sized binary matrix (where N is the number of tests and M is the number of methods). Binary values are stored in the boolean variable (typically 1 byte), however, natural numbers are usually stored in 4 bytes (for example: using integer type). Thus, I can conclude that the count-based approach has a four times greater space complexity than the hit-based algorithm.

To calculate the coverage matrix (Cov^U) of the unique count-based method, *WNFL*, and *WENFL*, we need another auxiliary matrix whose items are tests and UDCSs. Its size is $N \times U$, where N = number of tests, U = number of UDCSs, and its value is either 0 or 1, depending on whether the (unique deepest) call stack was generated during the test or not. By traversing this matrix, we aggregate the frequency of the methods in the UDCSs generated by the test and collect them in the Cov^U matrix. As can be seen in Table 2.8, the number of UDCSs (column 6) is on average 10-100 times larger than the number of methods (column 5), thus increasing the space and time complexity of the fault localization algorithms (due to storage and “traversal”),

therefore, will be somewhat more expensive to execute than hit-based methods. The unique count-based matrix (Cov^U) requires 4 bytes to represent an element, *i.e.* this method also has four times the complexity compared to Cov^H .

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

It is enough to examine each vector once (for a given code element) to calculate the spectrum metrics (Section 2.4) that form the basis of the fault localization formulae (Table 2.2). For count-based algorithms, if the value of a matrix element is 0 (*i.e.*, not covered), we must also examine the coverage vector for the given test to determine m_{nf} and m_{np} . These “extra” operations increase the time complexity of these methods. If the matrix contains many 0 values, we have to examine the test vectors very often, but with the current (hardware-)performance, the resulting increase in execution time is not significant compared to the hit-based concept.

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

2.10.4 Threats to Validity

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

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

A possible threat to the validity of the empirical study is that I had to exclude some parts of the Defects4J dataset, so this could make it difficult to compare the results to other studies employing the same benchmark. However, this affected only 49 bugs, which amounts to about 6% of the total bugs in the original set. The reason was that I could not compute UDCSs for these cases due to the technical limitations of the analysis or for other practical reasons (*e.g.*, there were no modified methods). These were not examined and analyzed. This selection was in no way influenced by the results, and the skipped bugs are distributed in the benchmark approximately evenly, so I believe that this factor can be considered minimal.

Another threat could be my use of only one benchmark that includes programs written in one language, Java. However, the bugs themselves are real and validated

bugs and not manually seeded or generated as is the case with many other benchmarks used in FL research. Nevertheless, it would be useful to examine the performance of the approach on other data sets consisting of programs in other languages and on defects of other types and qualities. For example, it is not known how the approach would perform in the presence of multiple bugs.

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

2.11 Conclusion

I proposed two kinds of new methods to increase the efficiency of fault localization algorithms. The unique counts concept relies on the notion of Unique Deepest Call Stacks, data structures that capture call stack state information occurring on test case execution, and counting the number of method occurrences within these structures. Graph-based approaches complement the information used by the classic Spectrum-Based Fault Localization with other features that can be extracted from the coverage graph (*NFL* and *ENFL*) and call frequency information (*WNFL* and *WENFL*). I adapted eight traditional hit-based SBFL formulae to both kinds of new algorithms and empirically verified how much we can improve fault localization effectiveness on the Defects4J benchmark.

I showed that the *NFL* and *ENFL* approaches can improve the effectiveness of hit-based algorithms. Those algorithms often result in better ranks than the traditional SBFL formulas, achieve very similar or better average ranks, and in several cases produce 5 or higher ranks.

The unique count approaches outperformed algorithms using binary data. These approaches can improve the effectiveness of their hit-based counterparts in 6 out of 8 cases by 5-101 positions on average, with a relative improvement of 14-74% and statistically significant differences. Also, all new formulae are able to achieve enabling improvements, and the accuracy regarding the number of bugs in the Top-5 category is increased by about 7% as well, which is important in terms of the practical usability of SBFL in general. I conclude that the best candidates to benefit from call frequency information are the unique count-based adaptations of *Jaccard*, *Ochiai*, *Sørensen-Dice*, *DStar*, and *Barinel*. In addition, considering the magnitude of differences and the consistency of the improvements, Δ_{ef}^U offers the best performance in the unique-based setting, but other formulae are just marginally behind.

However, by “mixing” the *ENFL* and the unique count approaches, we can get better results than before. The average rank of *WENFL* is 20.59, which is significantly better than any of the hit-based formulae (the differences between 13.4 and 115.4, 39%-84%) or their count-based counterparts (3.7-64.8, 15%-76%). Furthermore, in 7 out of 8 cases *WENFL* produced the highest accuracy (389 bugs, 49.5%) and the highest enabling improvement.

3

BUGSJS: a Benchmark of JavaScript Bugs

3.1 Introduction

The renowned JavaScript (JS) is referred to as the most widely used web programming language.¹ Also, it is one of the most often used languages on the GitHub repository.² JS is very popular on the client-side of web applications because it has an efficient performance (V8 JavaScript engine includes just-in-time compilation, inlining, and dynamic code optimization, etc) and at the same time it is very user-friendly. In recent years, thanks to the positive characteristics of JS (eg: performance), the hottest buzzword is “full stack JavaScript”, that is, server-side applications are increasingly written in JS [9]. There are many frameworks (React.js, Angular, and Node.js) that can be used to easily and efficiently develop client-side and server-side Javascript web applications, thus increasing the popularity of JS.

Despite its popularity, JS also has a number of weaknesses that can make code development, maintainability, or security more difficult. Such objectionable or worrisome properties are weak typing, prototypal inheritance, and client-side security. As such, a large part of (software) research has focused on the analysis and testing of JavaScript applications [16, 127, 85, 31, 5, 72].

Research based on empirical evaluation uses software-related artifacts (such as production (source) code and unit test cases) during analyses or experiments, but such a JS program benchmark has not been available to researchers so far. Typically, different programs were selected by the researchers based on the characteristics they considered important, thus the comparison of different experiments and measurement results was not possible (or only at the cost of great efforts). Furthermore, the benchmark and the data set were generally not properly documented and made available, which also made reproduction, comparison, and further analysis impossible.

One of these typical research areas is fault localization, presented in the previous chapter. Real bugs are difficult to identify in the source code, so researchers often

¹<https://sdtimes.com/softwaredev/report-the-top-three-programming-languages-of-2018-are-java-javascript-and-python/>

²<https://octoverse.github.com>

use (manually or automatically) “seeded” bugs or mutation testing techniques [52] to evaluate the fault localization methods. However, these have their disadvantages. In general, it is quite difficult for other researchers to reproduce the bugs, either because of creating the necessary environment (for example the problem of dependencies) or identifying and injecting the bugs. The “quality” (e.g. complexity) of artificially, manually injected bugs does not adequately represent real faults, and these can be “manipulated” by which I mean that the researchers can skip bugs, that would adversely affect the outcome or the judgment of their experiment. Mutation-based error injection solves these problems to some extent [40, 56, 12], but even in the case of a small-sized program, there are many mutation possibilities, the analysis and evaluation of which require enormous resources. For these reasons, benchmarks of manually validated bugs are very important for a lot of research areas, thereby increasing their reliability.

However, thanks to researchers, many reliable and validated benchmarks are available, the most relevant of which are Software-artifact Infrastructure Repository [28], Defects4J [55], ManyBugs [62], and BugSwarm [27]. These benchmark target languages such as Java or C, which have so far been amongst the most popular in studies, but in Section 3.7 these and other notable instances are also presented in detail. Purpose-specific tests and datasets exist to support research in test generation [36], program repair [68], and security [38].

However, to the best of our knowledge, a JavaScript data set similar to these is still missing, even though it would be necessary in order to support rigorous empirical experiments in web testing. For this reason, a repository of well-organized, categorized, and labeled JavaScript programs and bugs is necessary.

BUGSJS features (i) an infrastructure containing detailed reports about the bugs, separating the validated and the original versions of the production and test code changes; (ii) BUGSJS Dissection, which is a web interface for the more efficient review of bug fixes and other relevant information (for example failed test cases); (iii) more precomputed data from the production and test code, and executions to the easier and more efficient comparison related research; (iv) the categories/subcategories of bug taxonomy (and their descriptions) were created in a bottom-up concept, by using different sources and analyzing information about the bugs, and (v) quantitative/qualitative analysis of the bug-fixes related to the bugs in relation to existing classification schemes.

3.2 Studies on JavaScript analysis and testing

To motivate the need for a novel benchmark for JavaScript bugs, I surveyed the works related to software analysis and testing in the JavaScript domain. My review of the literature also allowed me to gain insights about the most active research areas in which our benchmark should aim to be useful.

In the JavaScript domain, the term benchmark commonly refers to collections of programs used to measure and test the *performance* of web browsers with respect to the latest JavaScript features and engines. Instances of such performance benchmarks are JetStream, Kraken, Dromaeo, Octane, and V8. However, I refer to a *benchmark* as a collection of JavaScript programs and artifacts (*e.g.*, test cases or bug reports) used to support empirical studies (*e.g.*, controlled experiments or user studies) related to one or more research areas in software analysis and testing.

I used the databases of scientific academic publishers and popular search engines to look for papers related to different software analysis and testing topics for JavaScript. I adopted various combinations of keywords: `JavaScript`, `testing` (including code coverage measurement, mutation testing, test generation, unit testing, test automation, regression testing), `bugs` and `debugging` (including fault localization, bug and error classification), and `web`. I also performed a lightweight forward and backward snowballing [129] to mitigate the risk of omitting relevant literature. Last, I examined the evaluation section of each paper. I retained only papers in which real-world, open-source JavaScript projects were used, and whose repositories and versions could be clearly identified. This yielded 25 final papers. Nine (9) of these studies are related to bugs, in which 670 subjects were used in total. The remaining 16 papers are related to other testing fields, comprising 494 subjects in total.

In presenting the results of the literature, I distinguish (1) studies containing specific bug information and other artifacts (such as source code and test cases), and (2) studies containing only JavaScript programs and other artifacts not necessarily related to bugs.

3.2.1 Bug-related Studies for JavaScript

I analyzed papers using JavaScript systems that included bug data in greater detail because these works can provide us important insights about the kind of analysis researchers used the subjects for, and thus, the requirements that a new benchmark of bugs should adhere to.

I found nine studies in this category. Ocariza et al. [85] present an analysis and classification of bug reports to understand the root causes of client-side JavaScript faults. This study includes 502 bugs from 19 projects with over 2M LOC. The results of the study highlight that the majority (68%) of JavaScript faults are caused by faulty interactions of the JavaScript code with the Document Object Model (DOM). Moreover, most JavaScript faults originate from programmer mistakes committed in the JavaScript code itself, as opposed to other web application components.

Another bug classification presented by Gao et al. [37] focuses on type system-related issues in JavaScript (which is a dynamically typed language). The study includes about 400 bug reports from 398 projects with over 7M LOC. The authors ran a static type checker, such as Facebook’s Flow¹ or Microsoft’s TypeScript², on the faulty versions of the programs. On average, 60 out of 400 bugs were detected (15%), meaning they may have been avoided in the first place if a static type checker was used to warn the developer about the type-related bug.

Hanam et al. [45] present a study of cross-project bug patterns in server-side JavaScript code, using 134 Node.js projects of about 2.5M LOC. They propose a technique called BugAID for discovering such bug patterns. BugAID builds on an unsupervised machine learning technique that learns the most common code changes obtained through AST differencing. Their study revealed 219 bug-fixing change types and 13 pervasive bug patterns that occur across multiple projects. In our evaluation, we conducted a thorough comparison with Hanam et al.’s taxonomy.

Ocariza et al. [89] propose an inconsistency detection technique for MVC-based JavaScript applications which is evaluated on 18 bugs from 12 web applications (7k LOC). A related work [88] uses 15 bugs in 20 applications (nearly 1M LOC). They also present

¹<https://flow.org/>

²<http://www.typescriptlang.org/>

an automated technique to localize JavaScript faults based on a combination of dynamic analysis, tracing, and backward slicing, which is evaluated on 20 bugs from 15 projects (14k LOC) [86]. Also, their technique for suggesting repairs for DOM-based JavaScript faults is evaluated on 22 bugs from 11 applications (1M LOC) [87].

Wang et al. [127] present a study on 57 concurrency bugs in 53 Node.js applications (about 3.5M LOC). The paper proposes several different analyses pertaining to the retrieved bugs, such as bug patterns, root causes, and repair strategies. Davis et al. [23] propose a fuzzing technique for identifying concurrency bugs in server-side event-driven programs and evaluate their technique on 12 real-world programs (around 216k LOC) and 12 manually selected bugs.

3.2.2 Other Analysis and Testing Studies for JavaScript

Empirical studies in software analysis and testing benefit from a large variety of software artifacts other than bugs, such as test cases, documentation, or code revision history. In this section, we briefly describe the remaining papers of our survey.

Milani Fard and Mesbah [35] characterize JavaScript tests in 373 JavaScript projects according to various metrics, *e.g.*, code coverage, test commits ratio, and number of assertions.

Mirshokraie et al. propose several approaches to JavaScript automated testing. This includes *automated regression testing* based on dynamic analysis, which is evaluated on nine web applications [81]. The authors also propose a *mutation testing* approach, which is evaluated on seven subjects [77], and on eight applications in a related work [78]. They also propose a technique to aid *test generation* based on program slicing [80], where unit-level assertions are automatically generated for testing JavaScript functions. Seven open-source JavaScript applications are used to evaluate their technique. The authors also present a related approach for JavaScript unit test case generation, which is evaluated on 13 applications [79].

Adamsen et al. [100] present a hybrid static/dynamic program analysis method to check *code coverage-based properties* of test suites from 27 programs. Dynamic symbolic execution is used by Milani Fard et al. [34] to generate DOM-based *test fixtures and inputs* for unit testing JavaScript functions, and four experimental subjects are used for evaluation. Ermuth and Pradel propose a GUI test generation approach [31] and evaluate it on four programs.

Artzi et al. [13] present a framework for *feedback-directed automated test generation* for JavaScript web applications. In their study, the authors use 10 subjects. Mesbah et al. [76] present Atusa, a *test generation technique for Ajax-based applications* which they evaluate on six web applications. A comprehensive survey of *dynamic analysis and test generation* for JavaScript is presented by Andreasen et al. [11].

Billes et al. [16] present a black-box analysis technique for multi-client web applications to detect *concurrency errors* on three real-world web applications. Hong et al. [50] present a testing framework to detect concurrency errors in client-side web applications written in JavaScript and use five real-world web applications.

Wang et al. [126] propose a modification to the *delta debugging* approach that reduces the event trace, which is evaluated on 10 real-world JavaScript application failures. Dhok et al. [26] present a *concolic testing* approach for JavaScript which is evaluated on 10 subjects.

Table 3.1: Subject distribution among surveyed papers

BUG-RELATED		ALL STUDIES	
# Papers	# Subjects	# Papers	# Subjects
1	607	1	910
2	17	2	91
3	7	3	17
4	2	4	4
5	0	5	1

3.2.3 Findings

In the surveyed papers, I observed that the proposed techniques were evaluated using different sets of programs, with little to no overlap.

Table 3.1 shows the program distribution per paper. In bug-related studies, 633 subject programs were adopted overall, with 607 of these programs (96%) being used in only one study, and no subject being used in more than four papers (Table 3.1, columns 1 and 2). Other studies exhibit the same trend (Table 3.1, columns 3 and 4): overall, 1,164 subjects were used in all the investigated papers, of which 1,023 were unique. From these, 910 (89%) were used in only one paper, and no subject was used in more than five papers.

In conclusion, I observe that the investigated studies involve different sets of programs, since no centralized benchmark is available to support reproducible experiments in analysis and testing related to JavaScript bugs.

To devise a centralized benchmark for JavaScript bugs that enables reproducibility studies in software analysis and testing, I considered the insights and guidelines provided by existing similar datasets (e.g., Defects4J [55]), as well as the knowledge gained from the literature on empirical experiments using JavaScript programs and bugs.

First, the considered subject systems should be real-world, publicly available open-source JavaScript programs. To ensure the representativeness of the benchmark, they should be diverse in terms of the application domain, size, development, testing, and maintenance practices (e.g., the use of continuous integration (CI), or code review process).

Second, the *buggy versions* of the programs must have one or more *test cases* available demonstrating each bug. The bugs must be reproducible under reasonable constraints; this excludes non-deterministic or flaky features.

Third, the versions of the programs in which the bugs were fixed by developers, i.e., the *patches*, must also be available. Typically, when a bug is fixed, new unit tests are also added (often in the same bug-fixing commit) to cover the buggy feature, allowing for better regression testing. This allows extracting the bug-fixing changes, e.g., by *diffing* the buggy and fixed revisions.

Additionally, the benchmark should include the bug report information, including critical times (e.g. when the bug was opened, closed, or reopened), the discussions about each bug, and a link to the commits where the bug was fixed.

To fill this gap, in section 3.3 I overview BUGSJS, a benchmark of real JavaScript bugs, its design, and implementation.

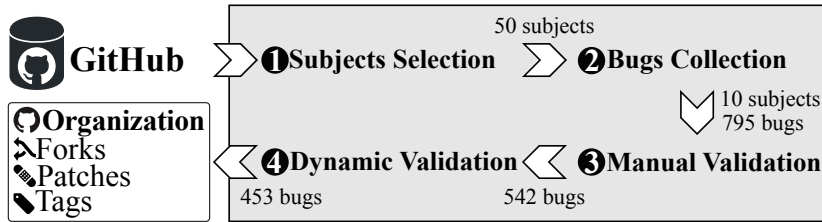
3.3 BugsJS – the Proposed Benchmark

To construct a benchmark of real JavaScript bugs, existing bugs were identified from the programs’ version control histories, and real fixes provided by developers were collected. Developers often manually label the revisions of the programs in which reported bugs are fixed (*bug-fixing commits*, or patches). As such, we referred to the revision preceding the bug-fixing commit as the *buggy commit*. This allowed us to extract detailed bug reports and descriptions, along with the buggy and bug-fixing commits they refer to. Particularly, each bug and fix should adhere to the following properties:

- **Reproducibility.** One or more *test cases* are available in a *buggy commit* to demonstrate the bug. The bug must be reproducible under reasonable constraints. The non-deterministic features and flaky tests are excluded from our data set since replicating them in a controlled environment would be excessively challenging.
- **Isolation.** The bug-fixing commit applies to JavaScript source code files only; changes to other artifacts, such as documentation or configuration files, are not considered. The source code of each commit must be *cleaned* from irrelevant changes (*e.g.*, feature implementations, refactorings, changes to non-JavaScript files). The *isolation* property is particularly important in research areas where the presence of noise in the data has detrimental impacts on the techniques (*e.g.* automated program repair, or fault localization approaches).

Figure 3.1 depicts the main steps of the process we performed to construct our benchmark. First, we adopted a systematic procedure to select the JavaScript subjects to extract the bug information from ❶. Then, we collected bug candidates from the selected projects ❷, and manually validated each bug for inclusion by means of multiple criteria ❸. Next, we performed a dynamic sanity check to make sure that the tests introduced in a bug-fixing commit can detect the bug in the absence of its fix ❹. Finally, the retained bugs were cleaned from irrelevant patches (*e.g.* whitespaces).

Figure 3.1: Overview of the bug selection and inclusion process.



3.3.1 Subject Systems Selection and Bugs Collection

We focused on the most popular JavaScript projects on GitHub and select the programs to use in BUGSJS. These projects often use solutions, tools, and rules that can greatly contribute to the identification of errors (for example GitHub’s *issue tracker* or *issue IDs* in the bug-fix commit message).

In searching for realistic subject systems, first, we overviewed several popular JavaScript projects from GitHub, and excluded projects whose *Stargazers count* was less

Table 3.2: Subjects included in BUGSJS

	STATS (#)				TESTS (#)				COVERAGE (%)			
	kLOC (JS)	Stars	Commits	Forks	All	Passing	Pending	Failing	Statements	Branches	Functions	Lines
BOWER	16	15,290	2,706	1,995	455	103	19	36	81.11	66.91	80.62	81.11
ESLINT	240	12,434	6,615	2,141	18,528	18,474	0	54	99.21	98.19	99.72	99.21
EXPRESS	11	40,407	5,500	7,055	855	855	0	0	98.71	94.32	100	99.95
HESSIAN.JS	6	104	217	23	225	223	2	0	96.42	91.27	98.99	96.42
HEXO	17	23,748	2,545	3,277	875	868	7	0	96.20	90.51	98.54	97.27
KARMA	12	10,210	2,485	1,531	331	331	0	0	54.61	34.03	43.98	54.76
MONGOOSE	65	17,036	9,770	2,457	2,107	2,071	36	0	90.97	85.95	89.65	91.04
NODE-REDIS	11	10,349	1,242	1,245	966	965	0	1	99.06	98.19	97.99	99.06
PENCILBLUE	46	1,596	3,675	276	807	802	0	5	35.21	19.09	22.91	35.22
SHIELDS	20	6,319	2,036	1,432	482	469	13	0	75.98	65.60	83.26	75.97

than 100 from further investigation. Furthermore, we selected web applications developed with the Node.js framework because it is one of the most popular JavaScript technologies on the server-side application [9]. In addition, the project had to be actively maintained (year of last commit ≥ 2017) and mature (number of commits > 200). The existence and use of *bug issue label* on GitHub’s *Issues* page were expected due to the identification of the bugs and their corresponding corrections, as well as the causes of errors. The initial subject list contained 50 Node.js programs, from which we filtered out projects based on the number of candidate bugs found and the adopted testing frameworks.

The official GitHub API was used to query the specific bug label, and for each closed bug, the links existing between commits and issues were utilized to identify the bug-fixing commit. Only those bugs that have only one “commit connection” were kept, similar to the concept used when creating Defects4J [55] and other benchmarks. Bugs that were connected to more than one fixing commit were excluded, because these were fixed in multiple steps, or because the first attempt at fixing them was unsuccessful.

It was also an indispensable condition that there should be at least one unit test that reproduces the error, i.e. without the part of the production code’s fix the test would fail, but after performing the bug-fix it would pass. This is the reason why the bug-fixing patches were examined and checked for whether they contained modifications in the test files. This process was carried out manually by several researchers (including the author of the dissertation). The result of this step is the list of bug candidates for the benchmark. From the initial list of 50 subject systems, we considered the projects having at least 10 bug candidates.

The test framework for potential program candidates was also examined and the researchers found that there is no predominant framework for JavaScript (as compared to, for instance, JUnit, which is used by most Java developers). Three frequently used frameworks were identified (Mocha– 52%, Jasmine– 10%, and QUnit– 8%), but Mocha stands out among them. As a result, projects were included in BUGSJS that use it during testing. The choice of Mocha is also justified based on other empirical studies that examined the popularity of test frameworks [35].

Table 3.2 reports the data and statistics of the 10 JavaScript applications we ultimately retained. It is interesting that all projects have at least 1000 LOC (frameworks excluded), thus being representative of modern web applications (Ocariza et al. [85] report an average of 1,689 LOC for AngularJS web applications on GitHub with at least 50 stars).

The subjects represent a wide range of domains.

- Bower: a package manager tool
- ESLint: a linter tool for ECMAScript/JavaScript
- Express: a web framework for Node.js.
- Hessian.js: a JavaScript binary web service protocol
- Hexo: a blog framework, powered by Node.js
- Karma: a test runner tool for JavaScript
- Mongoose: a MongoDB object modeling tool
- Node-redis: a Redis client for Node.js
- Pencilblue: a CMS and blogging platform, powered by Node.js
- Shields: a web service for badges in SVG and raster format.

3.3.2 Manual Patch Validation

We manually investigated each bug and the corresponding bug-fixing commit to ensure that only bugs meeting certain criteria are included, as described below.

Table 3.3: Bug-fixing commit inclusion criteria

Rule Name	Description
Isolation	The bug-fixing changes must fix only one (1) bug (i.e., must close exactly one (1) issue)
Complexity	The bug-fixing changes should involve a limited number of files (≤ 3), lines of code (≤ 50) and be understandable within a reasonable amount of time (max 5 minutes)
Dependency	If a fix involves introducing a new dependency (e.g., a library), there must also exist production code changes and new test cases added in the same commit
Relevant Changes	The bug-fixing changes must only involve changes in the production code that aim at fixing the bug (whitespace and comments are allowed)
Refactoring	The bug-fixing changes must not involve refactoring of the production code

Methodology. Two researchers manually investigated each bug and its corresponding bug-fixing commit and labeled them according to a well-defined set of *inclusion criteria* (Table 3.3). The bugs that met all criteria were initially marked as “Candidate Bugs” to be considered for inclusion.

Table 3.4: Manual and dynamic validation statistics per application for all considered commits

		BOWER	ESLINT	EXPRESS	HESIAN.JS	HEXO	KARMA	MONGOOSE	NODE-REDIS	PENCILBLUE	SHIELDS	Total
MANUAL	<i>Initial number of bugs</i>	10	559	39	17	24	37	56	25	18	10	795
	✗ Fixes multiple issues	0	18	1	0	1	5	2	5	0	0	32
	✗ Too complex	0	94	0	4	8	4	8	7	9	2	136
	✗ Only dependency	1	9	0	0	1	0	2	0	0	0	13
	✗ No production code	0	20	4	0	1	1	2	0	0	1	29
	✗ No tests changed	1	0	1	0	0	0	0	1	1	0	4
	✗ Refactoring	0	36	0	0	0	1	1	1	0	0	39
DYNAMIC	<i>After manual validation</i>	8	382	33	13	13	26	41	11	8	7	542
	✗ Test does not fail at V_{bug}	1	11	6	4	1	2	8	3	1	3	40
	✗ Dependency missing	3	17	0	0	0	1	1	0	0	0	22
	✗ Error in tests	1	7	0	0	0	0	3	1	0	0	12
	✗ Not Mocha	0	14	0	0	0	1	0	0	0	0	15
✓ <i>Final Number of Bugs</i>		3	333	27	9	12	22	29	7	7	4	453

In detail, for each bug, the authors investigated the code of the commit to ensure relatedness simultaneously to the bug being fixed. During the investigation, however, several bug-fixing commits were too complex to comprehend by the investigators, either because domain knowledge was required, or because the number of files or lines of code being modified was large. We labeled such complex bug-fixing commits as “Too complex”, and discarded them from the current version of BUGSJS. The rationale is to keep the size of the patches within reasonable thresholds, so as to select a high-quality corpus of bugs that can be easily analyzed and processed by both manual inspection and automated techniques. Particularly, we deemed a commit too complex if the production code changes involved more than three (3) files or more than 50 LOC, or if the fix required more than 5 minutes to understand. In all such cases, a discussion was triggered among the authors, and the case was ignored if the authors unanimously decided that the fix was too complex.

Another case for exclusion was due to refactoring operations in the analyzed code. First, our intention was to keep the original code’s behavior as written by the developers. As such, we only restored modifications that did not affect the program’s behavior (e.g., whitespaces). Indeed, in many cases, in-depth domain knowledge is very much required to decouple refactoring and bug fixing. JavaScript is a dynamic language, and code can be refactored in many ways. Thus, it is more challenging to observe and account for side-effects only by looking at the code than, for instance, in Java. In addition, refactoring may affect multiple parts of a project; it affects metrics such as code coverage, and it makes restoring the original code changes more challenging.

Results. Overall, we manually validated 795 commits (*i.e.*, bug candidates), of which 542 (68.18 %) fulfilled the criteria. Table 3.4 (Manual) illustrates the result of this step for each application and across all applications.

The most common reason for excluding a bug is that the fix was deemed too complex (136). Other frequent scenarios include cases where a bug-fixing commit addressed

more than one bug (32), or where the fix did not involve production code (29), or contained refactoring operations (39). Also, we found four cases in which the patch did not involve the actual test's source code, but rather comments or configuration files.

3.3.3 Sanity Checking through Dynamic Validation

An important requirement for our benchmark is to include (and distinguish) the unit test code elements which are actually intended to test the buggy feature and make sure that the tests are not passing or failing randomly or based on the order of execution. We follow a systematic approach to ensure meeting this requirement.

Methodology.

As I mentioned in the previous chapter, the commit marked in the closed bug issue (that is, the one that corrects the error) defines the state of the source code that represents its corrected version (V_{fix}). The state before the fix-commit will be the version of the program that still contains the bug (V_{bug}), and there must be at least one test case that produces a failed result in the faulty version and a passed result in the corrected code version. Those test cases that produce a failed result in both versions are not relevant from the point of view of the bug or bug-fixing. This allows us to identify the test used in V_{fix} to demonstrate the bug (isolation).

As the software life cycle progresses, developers often make major changes to the program, its structure, and its environment. Such possible scenarios are, for example, the used dependency is unavailable (because a lot of time has passed since the change and it was either deleted or it is not available for some other reason), or the developers change the test framework (*e.g.*, from QUnit to Mocha). To run the tests successfully, the dependencies were obtained and the environments were configured for each specific revision of the source code.

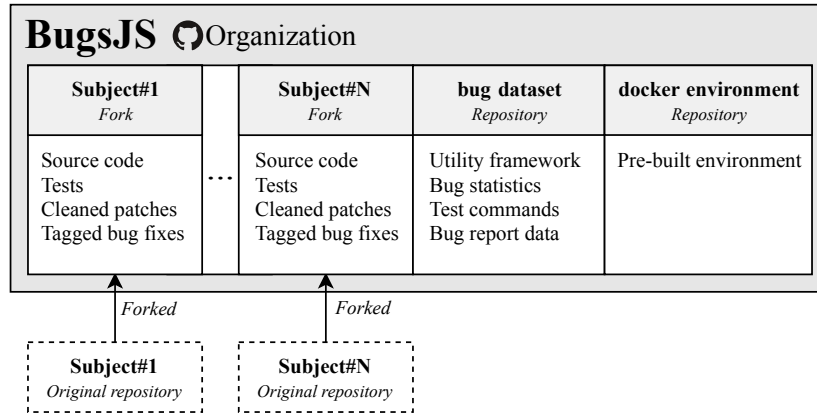
Where the project relies on some kind of automation for the execution of the tests, that is, it uses a script (for example `bash`, `Makefile`), in those cases, these scripts have been "extracted". As a result, the tests can also be run separately, eliminating possible sources of error and side effects that can occur by running the entire test suite. This also allows us to provide additional arguments to the test runner so that we can change its output to a format that is easier to process for the benchmark's users.

Results.

Almost 16% of the 542 bugs (89 cases) were excluded by the dynamic analysis, i.e. the number of errors in the data set changed to 453 after the dynamic "filtering". Table 3.4 (Dynamic) presents the dynamic validation criteria and their results. In 22 cases, we were unable to run the tests because dependencies were removed from the repositories. In 15 cases, the project at revision V_{bug} did not use Mocha for testing *b*. These are from earlier stages of the projects and since we are focusing on Mocha tests, we excluded these bugs. During running the tests we run into 12 cases where there were some errors in the tests, thus we had to skip these bugs as well, whereas in 40 cases no tests failed when executed on V_{bug} . These errors originated primarily from syntax errors, *e.g.* `SyntaxError: Unexpected token foo` on line 34 in one of the ESLint versions¹. We excluded all such bug candidates from the benchmark.

¹<https://github.com/eslint/eslint/blob/f912e217a985388891c9a5069d03336c7b0b47ea/tests/lib/util/source-code.js>

Figure 3.2: Overview of BUGSJS architecture



3.3.4 Benchmark Infrastructure and Implementation

Infrastructure.

Figure 3.2 shows the architecture of BUGSJS, which can be used to perform basic operations on the data set. Four such operations were defined in the infrastructure, thereby supporting researchers in the use of the benchmark. The command-line interface includes the following options:

- **info**: dumps information about a given bug.
- **checkout**: downloads the source code for a given bug.
- **test**: runs all tests for a given bug and measures the aggregated test coverage.
- **per-test**: runs each test individually and measures the per-test coverage.

With the exception of **info**, for the other three commands, we have the option to specify the state of the bug for which the command would like to be executed, thus providing users with greater flexibility.

A common problem with similar benchmarks is that, due to technical issues and incomplete documentation, it is difficult to use them and reproduce the experiments. This problem was overcome by creating the environment in a Docker image (with a detailed step-by-step tutorial), which contains the necessary configurations for each project to execute correctly, and by creating a website where additional information about the framework can be found.

<https://bugsjs.github.io/>

Revision of the source code. All of the projects' history we used is located on GitHub. This way our main data source is exposed to the developers' will. They could rewrite the whole history or even delete the whole repository. In order to preserve it, we used GitHub's forking mechanism. This way, the commit identifiers (commit hash) are preserved, and we can synchronize the repository with the original codebase, but at the same time, the source code remains available in the framework even after deleting the project's storage. In addition, we made use of the fork's property that makes it possible to "insert" new, own commits, so, for example, it was possible to separate the original patch from the manually cleaned commit.

We assigned five different marked revisions to each bug, thus separating the different parts of the patch:

- **Bug-X**: The parent commit of the fixed revision (*i.e.*, it is the buggy revision);
- **Bug-X-original**: This revision containing the original bug-fixing changes (including the production code and the newly added tests);
- **Bug-X-test**: A revision including only the test’s modifications (from the bug-fixing commit), applied to the buggy revision;
- **Bug-X-fix**: This is similar to the **Bug-X-test**, but in this case, the production code changes were applied to the faulty version;
- **Bug-X-full**: A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

Test runner commands and report data.

For easier reproduction, a CSV file was created that contains the test runner commands, so these can be edited manually or automatically as needed. One line in the file represents one bug in the benchmark and contains the following information:

1. The bug ID;
2. The test command required to run the tests;
3. The test command required to measure the test coverage data;
4. The Node.js version of the source code for the bug;
5. The preparatory test runner commands (*e.g.*, to initialize the environment to run the tests, which we call **pre-commands**);
6. The cleaning test runner commands to restore the program’s state.

For the reason of completeness, the framework also stored other information related to the bugs (which was collected with GitHub’s API). We stored the bugID for each bug, the text of the bug description, the issue open and close dates, the hash and the date of the bug-fixing commit, and the commit author identifiers. Furthermore, a number of pre-computed data were uploaded, which were computed using static and dynamic analyses. These available data facilitate the reusability of the benchmark in many research areas such as fault localization or bug prediction.

Precomputed data.

Basic information about each buggy “source code state” was stored in a CSV file. These are the data about the size of the given code (lines of code, number of branches, and number of methods), the coverage of the code (covered lines, branches, and methods), and the test results (number of tests, number of passed tests, number of failed tests, and number of data concerning skipped tests). In addition, coverage per-test data for **Bug-X** and **Bug-X-test** versions of each bug were generated and saved in JSON format, for which **Istanbul**² was used.

Furthermore, static measurements were carried out. Using **SourceMeter**³, a number

²<https://istanbul.js.org/>

³<https://www.sourcemeter.com/>

BugsJS Dissection is showing 453 Bugs

Bug id	# Files	# Lines	# Added	# Removed	# Modified	# Chunks	# Failing tests	# Bug-fixing types
Bower 1	2	27	22	2	3	5	50	3
Bower 2	1	8	2	0	6	3	43	3
Bower 3	1	1	0	0	1	1	41	1
Eslint 1	1	1	0	0	1	1	56	1
Eslint 2	1	13	13	0	0	1	15	1
Eslint 3	1	1	0	0	1	1	15	1
Eslint 4	1	5	5	0	0	1	62	1
Eslint 5	1	9	0	4	5	6	56	2
Eslint 6	1	7	0	1	6	4	62	2
Eslint 7	1	3	0	0	3	3	3	2
Eslint 8	1	1	0	0	1	1	56	1
Eslint 9	1	11	1	0	10	2	24	1
Eslint 10	2	6	3	1	2	4	25	0
Eslint 11	1	1	0	0	1	1	56	1

Figure 3.3: BugsJS Dissection overview page

of (41) static metrics were calculated for the **Bug-X** and **Bug-X-full** versions of each bug, thereby supporting research based on (static) source code metrics using Javascript programs. The results can be downloaded in a zip file.

3.3.5 BugsJS Dissection

Sobreira et al. [115] implemented a web-based interface for the bugs in the Defects4J bug benchmark [55]. It presents data to help researchers and practitioners to better understand this bug dataset.¹ We also utilized this dashboard and ported Dissection to BUGSJS (Figure 3.3), which is available on the BUGSJS website:

<https://bugsjs.github.io/dissection>

BUGSJS Dissection presents the information in the dataset to the user in an accessible and browsable format, which is useful for inspecting the various information related to the bugs, their fixes, their descriptions, and other artifacts such as the precomputed metrics.

More precisely, the information provided in BUGSJS Dissection include:

1. **# Files**: number of changed files.
2. **# Lines**: number of changed lines.
3. **# Added**: number of added lines.
4. **# Removed**: number of removed lines.
5. **# Modified**: number of modified lines.
6. **# Chunks**: number of sections containing sequential line changes.

¹<https://github.com/program-repair/defects4j-dissection>

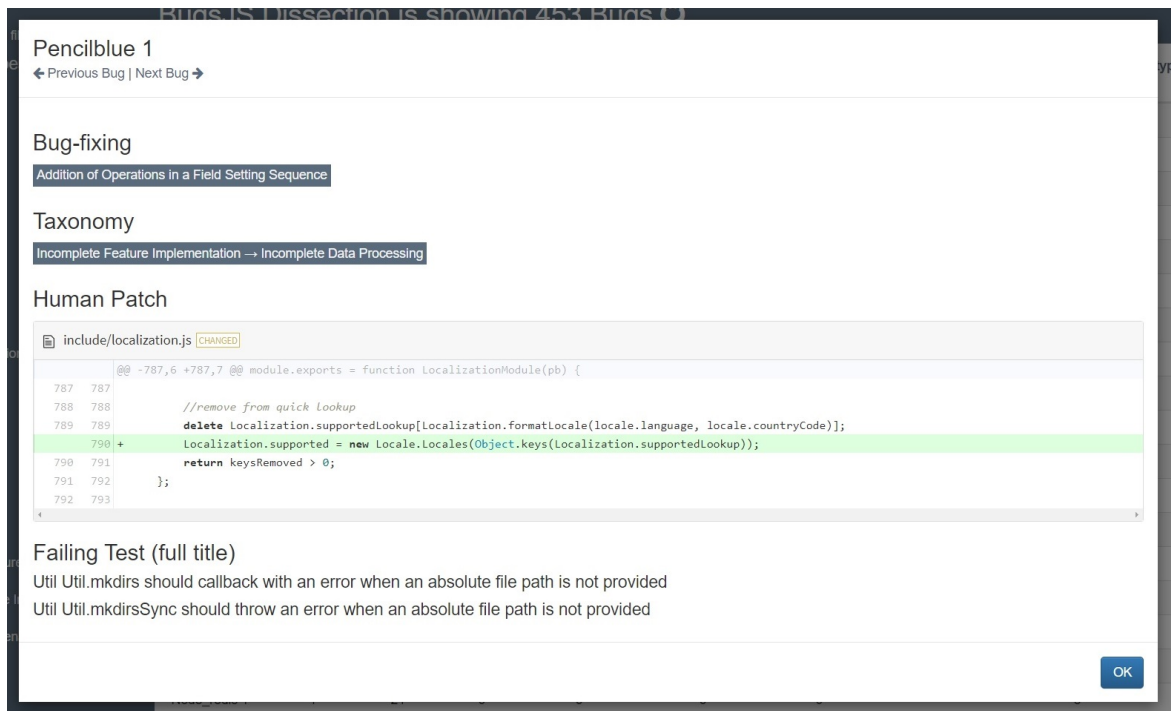


Figure 3.4: BugsJS Dissection page for one bug

7. # Failing tests: number of failed test cases.
8. # Bug-fixing type: number of bug-fixing types based on the taxonomy by Pan et al. [91].

Inspection of the bugs is supported through different filtering mechanisms that are based on the bug taxonomy and bug-fix types. By clicking on a bug, additional details appear (Figure 3.4), including bug-fix types, the patch by the developers, the taxonomy category, and the failed tests.

3.4 Taxonomy of Bugs in BugsJS

In this section, we present a detailed overview of the root causes behind the bugs in our benchmark. We adopted a systematic process to classify the nature of each bug, which we describe next.

3.4.1 Manual Labeling of Bugs

Each bug and associated information (i.e., bug report, and issue description) was manually analyzed by four authors (referred to as “taggers” hereafter) following an open coding procedure [109]. Four taggers specified a descriptive label for each bug assigned to them. The labeling task was performed independently, and the disagreements were discussed and resolved through dedicated meetings. Unclear cases were also discussed and resolved during such meetings.

First, we performed a pilot study, in which all taggers reviewed and labeled a sample of 10 bugs. Bugs for the pilot were selected randomly from all projects in BUGSJS. The consensus on the procedure and the final labels was high, therefore, for the subsequent rounds, the four taggers were split into two pairs, which were shuffled after each round of tagging.

The labels were collected in separate spreadsheets; the agreement on the final labels was found by discussion. During the tagging, the taggers could reuse existing labels previously created, should an existing label apply to the bug under analysis. This choice was meant to limit the introduction of nearly-similar labels for the same bug, and help taggers use consistent naming conventions.

When inspecting the bugs, we looked at several sources of information, namely (1) the bug-fixing commit on GitHub’s web interface containing the commit title, the description as well as at the code changes, and (2) the entire issue and pull request discussions.

In order to achieve internal validation in the labeling task, we performed cross-validation. Specifically, we created an initial version of the taxonomy labeling around 80% of the bugs (353). Then, to validate the initial taxonomy, the remaining 20% (100) were simply assigned to the closest category in the initial taxonomy, or a new category was created, when appropriate. Bugs for the initial taxonomy were selected at random, but they were uniformly selected among all subjects, to avoid over-fitting the taxonomy towards a specific project. Analogously, the validation set was retained so as to make sure all projects were represented. Internal validation of the initial taxonomy is achieved if few or no more categories (i.e., labels) were needed for categorizing the validation bugs. The labeling process involved four rounds: the first round (the pilot study) involved labeling 10 bugs, the second round 43 bugs, and 150 bugs were analyzed in both the third and fourth rounds.

3.4.2 Taxonomy Construction

After enumerating all causes of bugs in BUGSJS, we began the process of creating a taxonomy, following a systematic process. During a physical meeting, for each bug instance, all taggers reviewed the bugs and identified candidate equivalence classes to which descriptive labels were assigned. By following a bottom-up approach, we first clustered tags that correspond to similar notions into categories. Then, we created parent categories, in which these categories and their subcategories follow specialization relationships.

3.4.3 Taxonomy Internal Validation

We performed the validation phase in a physical meeting. Each of the four taggers classified one-fourth of the validation set (25 bugs) independently, assigning each of them to the most appropriate category. After this task, all taggers reviewed and discussed the unclear cases to reach a full consensus. All 100 validation bugs were assigned to existing categories, and no further categories were needed.

3.4.4 The Final Taxonomy

Figure 3.5: Taxonomy of bugs in the benchmark of JavaScript programs of BUGSJS.

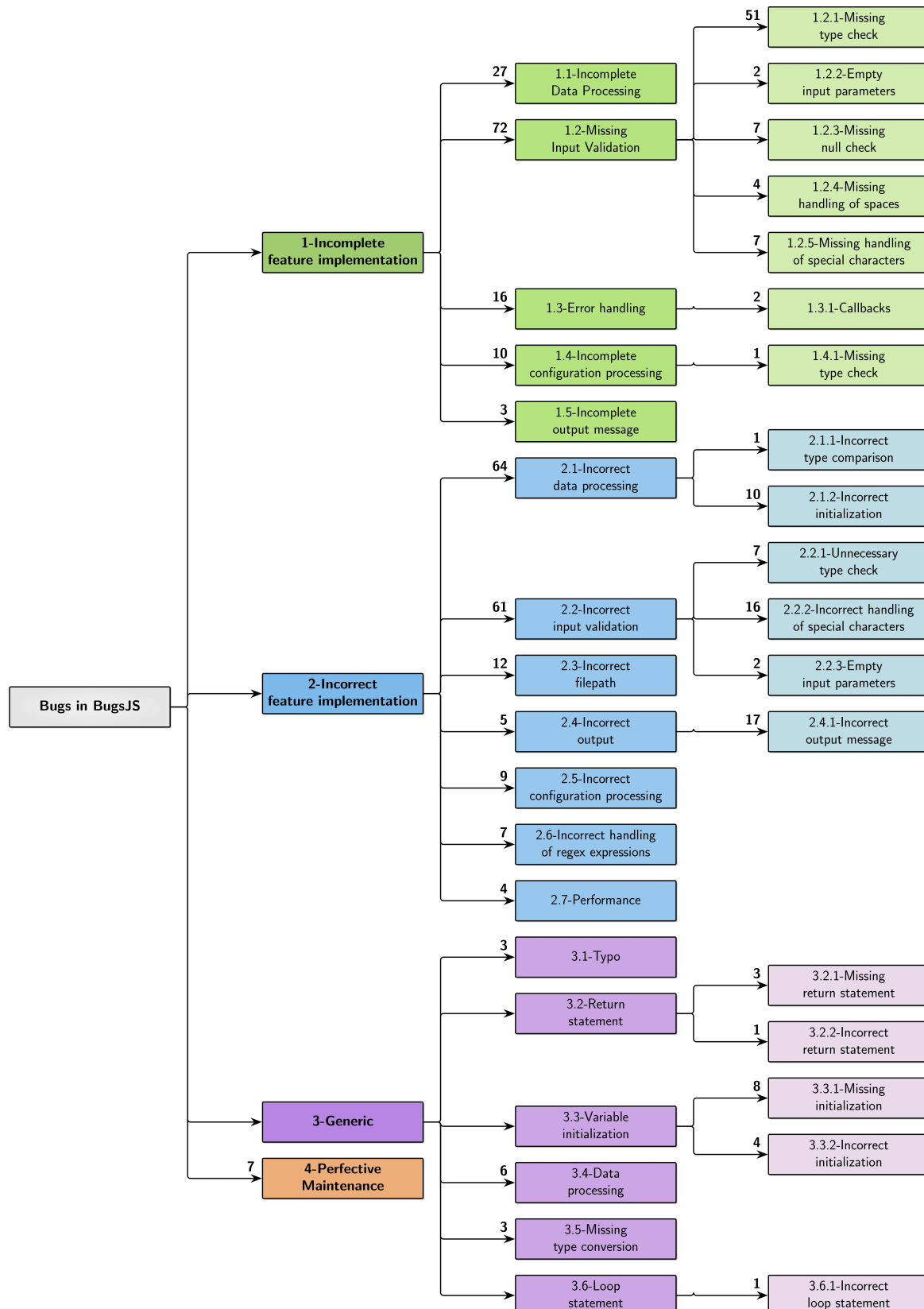


Figure 3.5 presents a graphical view of our taxonomy of bugs in the JavaScript benchmark. Nodes represent classes and subclasses of bugs, and edges represent specialization relationships. Specializations are not complete, disjoint relationships. Even though during labeling we tried assigning the most specific category, we found out during taxonomy creation that we had to group together many app-specific corner cases. Thus, some bugs pertaining to inner nodes were not further specialized to avoid creating an excessive number of leaf nodes with only a few corner cases.

At the highest level, we identified four distinct categories of causes of bugs, as follows:

1. Causes related to *incomplete feature implementation*. These bugs are related either to an incomplete understanding of the main functionalities of the considered application, or a refinement in the requirements. In these cases, the functionalities have already been implemented by developers according to their best knowledge, but over time, users or other developers found out that they do not consider all aspects of the corresponding requirements. More precisely, given a requirement r , the developer implemented a program feature f' which corresponds to only a subset $r' \subset r$ of the intended functionality. Thus, the developer has to adapt the existing functionality $f' \subset f$ to f , in order to satisfy the requirement in r . Typical instances of this bug category are related to one or more specific corner cases that were unpredictable at the time in which that feature was initially created, or when the requirements for the main functionalities are changed or extended to some extent.
2. Causes related to *incorrect feature implementation*. These bugs are also related to the mainstream functionalities of the application. As opposed to the previous category, the bugs in this category are related to a wrong implementation by the developers, for instance, due to an incorrect interpretation of the requirements. More precisely, suppose that given a requirement r , the developer implemented a program feature f' , to the best of their knowledge. Over time, other developers found out, through the usage of the program, that the behavior of f' does not reflect the intended behavior described in r , and opened a dedicated issue in the GitHub repository (and, eventually, a pull request with a first fix attempt).
3. Causes related to *generic* programming errors. Bugs belonging to this category are typically not related to an incomplete/incorrect understanding of the requirements by developers but rather to common coding errors, which are also important from the point of view of a taxonomy of bugs.
4. Causes related to *perfective maintenance*. Perfective maintenance involves making functional enhancements to the program in addition to the activities to increase its performance even when the changes have not been suggested by bugs. These can include completely new requirements for the functionalities, or improvements to other internal or external quality attributes not affecting existing functionalities. When composing BUGSJS, we aimed at excluding such cases from the candidate bugs (see Section 3.3), however, the bugs that we classified in the taxonomy with this category were labeled as bugs by the original developers, so we decided to retain them in the benchmark.

We will now discuss each of these categories in turn, in each case considering the subcategories beneath them.

1 - Incomplete feature implementation

This category contains 45% of the bugs overall and has five subcategories, which we describe in the following subsection.

1.1 - Incomplete data processing

The bugs in this category are related to an incomplete implementation of a feature's logic, i.e., the way in which the input is consumed and transformed into output.

Overall, 27 bugs were found to be of this type.

An example is **Bug#7** of Hexo, in which an HTML anchor was undefined unless a correct escaping of markdown characters is used.

```
1 - var text = $(this).html();  
2 + var text = _.escape($(this).text());
```

1.2 - Missing input validation

The bugs in this category are related to incomplete input validation, i.e., the way in which the program checks whether a given input is valid, and can be processed further.

Overall, 16% of the bugs were found to be of this type, and a further 16% in more specialized instances. This prevalence was mostly due to the nature of some of our programs. For instance, ESLint provides linting utilities for JavaScript code, and it is the most represented project in BUGSJS (73%). Therefore, being its main scope to actually validate code, we found many cases related to invalid inputs being unmanaged by the library, even though we also found instances of these bugs in other projects. For instance, in **Bug#4** of Karma, a file parsing operation should not be triggered on URLs having no line number. As such, in the bug-fixing commit, the proposed fix adds one more condition.

```
1 - if (file && file.sourceMap) {  
2 + if (file && file.sourceMap && line) {
```

Another prevalent category is due to missing type checks on inputs (11%), whereas less frequent categories were missing check of null inputs, empty parameters, and missing handling of spaces or other special characters (e.g., in URLs).

1.3 - Error handling

The bugs in this category are related to the incomplete handling of errors, i.e., the way in which the program manages erroneous cases, i.e., exception handling.

Overall, 3% of the bugs were found to be of this type. For instance, in **Bug#14** of Karma, the program does not throw an error when using a plugin for a browser that is not installed, which is a corner case missed in the initial implementation. Additionally, we found two cases specific to callbacks.

1.4 - Incomplete configuration processing

The bugs in this category are related to an incomplete configuration, i.e., the values of parameters accepted by the program.

Overall, 2% of the bugs were found to be of this type. For instance, in **Bug#10** of ESLint, an invalid configuration is used when applying extensions to the default configuration object. The bug fix updates the default configuration object's constructor to use the correct context, and to make sure the config cache exists when the default configuration is evaluated.

1.5 - Incomplete output message

The last subcategory pertains to bugs related to incomplete output messages by the program.

Only three bugs were found to be of this type. For instance, in **Bug#8** of Hessian.js, the program casts the values exceeding `Number.MAX_SAFE_INTEGER` as string, to allow safe readings of large floating point values.

2 - Incorrect feature implementation

This category contains 48% of the bugs overall and has seven subcategories, which we describe in the following subsections.

2.1 - Incorrect data processing

The bugs in this category are related to a wrong implementation of a feature's logic, i.e., the way in which the input is consumed and transformed into output.

Overall, 75 bugs were found to be of this type, with two subcategories due to wrong type comparison (1 bug), or incorrect initialization (10 bugs). An example of this latter category is **Bug#238** of ESLint, in which developers remove the default parser from CLIEngine options to fix a parsing error.

```
1 - parser: DEFAULT_PARSER
2 + parser: ""
```

2.2 - Incorrect input validation

The bugs in this category are related to wrong input validation, i.e., the way in which the program checks whether a given input is valid, and can be processed further.

Overall, 19% of the bugs were found to be of this type, with three subcategories due to unnecessary type checks (7 bugs), incorrect handling of special characters (16 bugs), or empty input parameters given to the program (2 bugs). As an example of this latter category, in **Bug#171** of ESLint, where the rule did not check for all spaces between the arrow character (`=>`) within a given code. Therefore, it is updated as follows:

```
1 - while (t.type !== "Punctuator" || t.value !== ">") {
2 + while (arrow.value !== ">") {
```

2.3 - Incorrect filepath

The bugs in this category are related to wrong paths to external resources necessary for the program, such as files. For instance, in Bug#6 of ESLint, developers failed to check for configuration files within sub-directories. Therefore, the code was updated as follows:

```
1 - if (!directory)
2 + if (directory) directory = path.resolve(this.cwd, directory);
```

2.4 - Incorrect output

The bugs in this category are related to incorrect output by the program. For instance, in Bug#7 of Karma, the exit code is wrongly replaced by null characters (`\0x00`), which results in squares (`□□□□□□`) being displayed in the standard output.

```
1 - return exitCode
2 + return {exitCode: exitCode, buffer: buffer.slice(0, tailPos)}
```

2.5 - Incorrect configuration processing

The bugs in this category are related to an incorrect configuration of the program, i.e., the values of parameters accepted.

Nine bugs were found to be of this type. For instance, in Bug#145 of ESLint, a regression was accidentally introduced where parsers would get passed additional unwanted default options even when the user did not specify them. The fix updates the default parser options to prevent any unexpected options from getting passed to parsers.

```
1 - let parserOptions = Object.assign({}, defaultConfig.parserOptions);
2 + let parserOptions = {};
```

2.6 - Incorrect handling of regex expressions

The bugs in this category are related to incorrect use of regular expressions.

Seven bugs were found to be of this type. For instance, in Bug#244 of ESLint, a regular expression is wrongly used to check that the function name starts with `setTimeout`.

2.7 - Performance

The bugs in this category caused the program to use an excessive amount of resources (e.g., memory). Only four bugs were found to be of this type. For instance, in Bug#85 of ESLint, a regular expression susceptible to catastrophic backtracking was used. The match takes quadratic time in the length of the last line of the file, causing Node.js to hang when the last line of the file contains more than 30,000 characters. Another representative example is Bug#1 of Node-Redis, in which parsing big JSON files takes substantial time due to an inefficient caching mechanism which makes the parsing time grow exponentially with the size of the file.

3 - Generic

This category contains 6% of the bugs overall and has six subcategories, which we describe next.

3.1 - Typo

This category refers to typographical errors by the developers.

We found three such bugs in our benchmark. For instance, in **Bug#321** of ESLint, a rule is intended to compare the *start* line of a statement with the end line of the previous token. Due to a typo, it was comparing the *end* line of the statement instead, which caused false positives for multiline statements.

3.2 - Return statement

The bugs in this category are related to either missing return statements (3 bugs), or incorrect usage of return statements (1 bug). For instance, in **Bug#8** of Mongoose, the fix involves adding an explicit return statement.

```
1 - this.constructor.update.apply(this.constructor, args);
2 + return this.constructor.update.apply(this.constructor, args);
```

3.3 - Variable initialization

The bugs in this category are related to either missing initialization of variables statements (8 bugs), or an incorrect initialization of variables (4 bugs). For instance, in **Bug#9** of Express, the fix involves correcting a wrongly initialized variable.

```
1 - mount_app.mountpath = path;
2 + mount_app.mountpath = mount_path;
```

3.4 - Data processing

The bugs in this category are related to the incorrect processing of information.

Six bugs were found to be of this type. For instance, in **Bug#184** of ESLint, developers fixed the possibility of passing negative values to the `string.slice` function.

```
1 - currentText.slice(node.range[0] - (beforeCount || 0)
2 + currentText.slice(Math.max(node.range[0] - (beforeCount || 0), 0)
```

3.5 - Missing type conversion

The bugs in this category are related to missing type conversions.

Three bugs were found to be of this type. For instance, in **Bug#4** of Shields, developers forgot to convert labels to strings prior to applying the uppercase transformation.

```
1 - data.text[0] = data.text[0].toUpperCase();
2 + data.text[0] = ('' + data.text[0]).toUpperCase();
```

3.6 - Loop statement

We found only one bug of this type—Bug#304 of Shields, —related to the incorrect usage of loop statements.

```
1 - while ((currentAncestor = currentAncestor.parent))
2 - if (isConditionalTestExpression(currentAncestor)) {
3 -   return currentAncestor.parent;
4 - }
5 - }
6
7 + do {
8 +   if (isConditionalTestExpression(currentAncestor)) {
9 +     return currentAncestor.parent;
10 +   }
11 + } while ((currentAncestor = currentAncestor.parent));
```

4 - Perfective maintenance

This category contains only 1% of the bugs. For instance, in Bug#209 of ESLint, developers fix JUnit parsing errors which treat no test cases having an empty output message as a failure.

3.4.5 Analysis of Bug-Fixes Patterns

To gain a better understanding of the characteristics of bug-fixes of the bugs included in BUGSJS, we have performed two analyses to quantitatively and qualitatively assess the representativeness of our benchmark. This serves as an addition to the taxonomy presented in Section 3.4.4 which, by connecting the bug types to the bug-fix types, can support applications, such as automated fault localization and automated bug repair.

We further analyzed the bugs in BUGSJS to observe the occurrence of low-level *bug fixes* for recurring patterns. Previous works [91, 144, 17] have studied patterns in bug-fixing changes within Java programs. They suggest that the existence of patterns in fixes reveals that specific kinds of code constructs (*e.g.*, if conditionals) could signal weak points in the source code where developers are consistently more prone to introducing bugs [91].

Methodology. Four authors of this paper manually investigated all 453 bug-fixing commits in BUGSJS and attempted to assign the bug-fixing changes to one of the predefined categories suggested in previous studies. In particular, we used the categories proposed by Pan et al. [91]. These categories, however, are related to Java bug fixes. Our aim is to assess whether they generalize to JavaScript, or whether, in contrast, JavaScript-specific bug fix patterns would emerge.

Following the original category definitions [91], we assigned each *individual* bug-fix to exactly one category. Disagreements concerning classification or potential new categories were resolved by further discussion between the authors. To identify the occurrences of such patterns, we opted for manual analysis to ensure covering potential new patterns, and to add an extra layer of validation against potential misclassifications (*e.g.*, false positives).

Table 3.5 shows the number of bug fix occurrences following the categories by Pan et al. [91]. (For fixes spanning multiple lines, we possibly assigned more than one category to a single bug-fixing commit, hence, the overall number of occurrences is greater than the number of bugs.)

Table 3.5: Bug-fixing change types (Pan et al. [91])

	Category	Example	#
EXISTING	if -related (IF)	Changing if conditions	291
	Assignments (AS)	Modifying the RHS of an assignment	166
	Function/Method calls (MC)	Adding or modifying an argument	151
	Class fields (CF)	Adding/removing class fields	25
	Function/Method declarations (MD)	Modifying a function’s signature	94
	Sequences (SQ)	Adding a function call to a sequence of calls	42
	Loops (LP)	Changing a loop’s predicate	7
	switch blocks (SW)	Adding/removing a switch branch	6
	try blocks (TY)	Introducing a new try-catch block	1
NEW	return statements	Changing a return statement	40
	Variable declaration	Declaring an existing variable	2
	Initialization	Initializing a variable with empty object literal/array	3

Note that, since the categories proposed by Pan et al. have been derived from Java programs, we had to make sure to match them correctly onto JavaScript code. In particular, until ECMAScript 2015, JavaScript did not include syntactical support for classes. Classes were *emulated* using functions as constructors, and methods/fields were added to their prototypes [113, 104, 33]. In addition, *object literals* could represent imaginary classes: the comma-separated list of name-value pairs enclosed in curly braces, where the name-value pairs declare the class fields/methods. We have taken all these aspects into account during the assignment task, to avoid misclassifications.

Our analysis revealed that, in 88% of bugs in BUGSJS, the fix includes changes falling into one of the proposed categories. The most prevalent bug fix patterns involve changing an **if** statement (*i.e.*, modifying the **if** condition or adding a precondition), changing assignment statements, and modifying function call arguments (Table 3.5). The same three categories have been also found to be most recurring in Java code, but with a different ordering: Pan et al. [91] report that the most prevalent fix patterns are changes done on method calls, **if** conditions, and assignment expressions. This can be explained by the fact that in JavaScript, object literals are frequently created without the need for defining a class or function constructor, and, as far as fixing bugs is concerned, updating their attributes (*i.e.*, fields) is a common practice.

JavaScript-related Bug-Fixing Patterns

We found three *new* recurring patterns in our benchmark, which we describe next.

Changes to the **return statement’s expression.** We found a recurring bug-fixing pattern involving changing the return statement’s expression of a function.

Variable declaration. In JavaScript, it is possible to use a variable without declaring it. However, this has implications that might lead to subtle *silent bugs*. For example, when a variable is used inside a function without being declared, it is “hoisted” to the top of the global scope. As a consequence, it is visible to all functions, *outside its original lexical scope*, which can lead to name clashes. This fix pattern essentially

includes declaring a variable that has already been in use.

Initialization of empty variables. This bug-fixing pattern category corresponds to Hanam et al.’s first bug pattern, *i.e.*, Dereferenced non-values. To avoid this type of bug, developers can add additional `if` statements, comparing values against “falsy” values (“`undefined`” type, or “`null`”). This bug fix pattern provides a shortcut to using an `if` statement, by using a logical “or” operator, *e.g.*, `a = a || {}`, which means that the value of `a` will remain intact if it already has a “non-falsy” value, or it will be initialized with an empty object otherwise.

Table 3.6 gives a description of the bug-fixing pattern abbreviations and how many of these cases occurred in the framework. Our results are also very similar to what Pan *et al.* experienced [91]. They analyzed 7 Java programs (Eclipse, Columba, JEdit, Scarab, ArgoUML, Lucene, and MegaMek) and came to the conclusion that the most common individual bug-fixing patterns are MC-DAP (method call with different actual parameter values), IF-CC (change in *if* conditional), and AS-CE (change of assignment expression).

The most common pattern in BUGSJS is AS-CE (166), which changes the expression on the right-hand side of an assignment statement (while the left-hand side remains unchanged in the buggy and fixed versions). This can be seen in Figure 3.6, which shows part of the ESLint-11 bug-fixing.

```

@@ -290,7 +290,9 @@ module.exports = {
 290 290     }
 291 291     } else if (multiOrNest) {
 292 292         if (hasBlock && body.body.length === 1 && isOneLiner(body.body[0])) {
 293 +         const leadingComments = sourceCode.getComments(body.body[0]).leading;
 294 +
 293 -         expected = false;
 295 +         expected = leadingComments.length > 0;
 296 296     } else if (!isOneLiner(body)) {
 297 297         expected = true;
 298 298     }
  
```

Figure 3.6: AS-CE bug-fixing pattern (bug: ESLint-11)

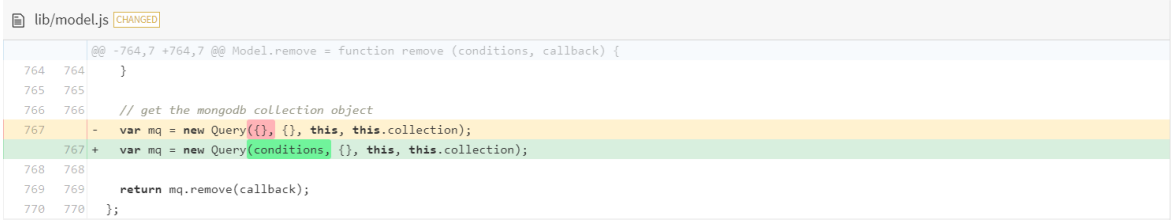
The second most common pattern (not far behind AS-CC) is IF-CC (152), which refers to some error in the *if* condition, and the condition had to be modified during correction. The `job_runner.js` modification, which is part of Pencilblue-6’s bug fixing, falls into this category (Figure 3.7).

```

@@ -237,7 +237,7 @@ module.exports = function JobRunnerModule(pb) {
 237 237     var query = pb.DAO.getIdWhere(this.getId());
 238 238     var updates = {};
 239 239
 240 -     if (pb.validation.isFloat(progressIncrement, true, true)) {
 240 +     if (pb.validation.isNum(progressIncrement, true) && progressIncrement > 0) {
 241 241         updates.$inc = {progress: progressIncrement};
 242 242     }
 243 243     if (pb.validation.isNonEmptyStr(status, true)) {
  
```

Figure 3.7: IF-CC bug-fixing pattern (bug: Pencilblue-6)

The third most common “patch” is MC-DAP (a method call with different actual parameter values: 93 cases), which changes the expression passed into one or more parameters of a method call. An example of this is the Mongoose-22 bug, which was fixed by changing the parameter of a function call (Figure 3.8)



```

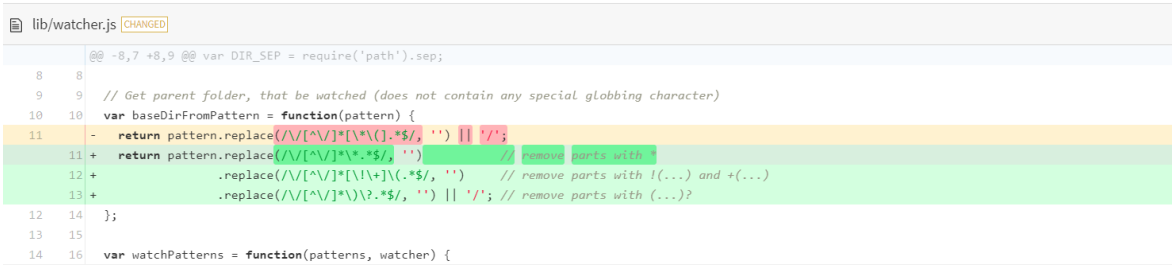
lib/model.js CHANGED
@@ -764,7 +764,7 @@ Model.remove = function remove (conditions, callback) {
764 764 }
765 765
766 766 // get the mongodb collection object
767 - var mq = new Query({}, {}, this, this.collection);
767 + var mq = new Query(conditions, {}, this, this.collection);
768 768
769 769 return mq.remove(callback);
770 770 };

```

Figure 3.8: MC-DAP bug-fixing pattern (bug: Mongoose-22)

Furthermore, the number of fixes was significant, where method declaration was added in the fix commit (MD-ADD: 70), and where precondition was added with jump statement such as break or return (IF-APCJ: 58). Interestingly, there were patterns for which we did not find examples in the data set (SQ-RFO and TY-ARCB).

Among the groups we created, JS-Return (changing a return statement) was the one that included many (40) bug fixes (for example Figure 3.9).



```

lib/watcher.js CHANGED
@@ -8,7 +8,9 @@ var DIR_SEP = require('path').sep;
8 8
9 9 // Get parent folder, that be watched (does not contain any special globbing character)
10 10 var baseDirFromPattern = function(pattern) {
11 - return pattern.replace(/\/[^\|\/]*[^\|\/]*$/g, '');
11 + return pattern.replace(/\/[^\|\/]*[^\|\/]*$/g, '') // remove parts with /
12 + .replace(/\/[^\|\/]*[^\|\/]*\(\.*/g, '') // remove parts with !(...) and +(...)
13 + .replace(/\/[^\|\/]*[^\|\/]*\?.*$/g, '') // remove parts with (...)?
14 };
15
16 var watchPatterns = function(patterns, watcher) {

```

Figure 3.9: JS-Return bug-fixing pattern (bug: Karma-9)

Overall, it can be concluded that we obtained similar results to other researchers, but further thorough examination of the results can help improve the results of other research areas (for example increasing the performance of fault localization and bug prediction algorithms).

3.5 Discussion

Our results might drive devising novel software analysis and repair techniques for JavaScript, with BUGSJS being a suitable real-world bug benchmark for their evaluation, as well as inform developers of the most error-prone constructs.

In the rest of this section, we discuss some of the potential uses for our taxonomy, together with possible use cases of our benchmark in supporting empirical studies in software analysis and testing, as well as its limitations and threats to the validity of our study.

3.5.1 Directing Research Efforts

Our taxonomy and associated data can be useful in several contexts related to JavaScript analysis and testing contexts. Our study reveals that the majority of bugs are related to mistakes made by developers. This finding is in line with those of the previous study by Ocariza et al. [85] on client-side JavaScript programs. Overall, the bug fixes included in BUGSJS cover a diverse range of categories, some of which are specific

Table 3.6: Bug-fixing patterns (*Pan* [91])

Category	Pattern name	#
Assignment (AS)	Change of assignment expression (AS-CE)	166
Class Field (CF)	Addition of a class field (CF-ADD)	5
	Change of class field declaration (CF-CHG)	18
	Removal of a class field (CF-RMV)	2
If-related (IF)	Addition of an else branch (IF-ABR)	13
	Addition of precondition check (IF-APC)	45
	Addition of precondition check with jump (IF-APCJ)	58
	Addition of post-condition check (IF-APTC)	8
	Change of if condition expression (IF-CC)	152
	Removal of an else branch (IF-RBR)	6
	Removal of an if predicate (IF-RMV)	9
Loop (LP)	Change of loop condition (LP-CC)	6
	Change of the expression that modifies the loop variable (LP-CE)	1
Method Call (MC)	Method call with different actual parameter values (MC-DAP)	93
	Different method call to a class instance (MC-DM)	26
	Method call with different numbers or types of parameters (MC-DNP)	32
Method Declaration (MD)	Change of method declaration (MD-CHG)	18
	Addition of a method declaration (MD-ADD)	70
	Removal of a method declaration (MD-RMV)	6
Sequence (SQ)	Addition of operations in an operation sequence of field settings (SQ-AFO)	18
	Addition of operations in an operation sequence of method calls to an object (SQ-AMO)	12
	Addition or removal method call operations in a short construct body (SQ-AROB)	7
	Removal of operations from an operation sequence of field settings (SQ-RFO)	0
	Removal of operations from an operation sequence of method calls to an object (SQ-RMO)	5
Switch (SW)	Addition/removal of switch branch (SW-ARSB)	6
Try (TY)	Addition/removal of a catch block (TY-ARCB)	0
	Addition/removal of a try statement (TY-ARTC)	1
<i>BugsJS</i>		
JavaScript (JS)	Changing a return statement (JS-Return)	40
	Initializing a variable with empty object literal/array (JS-Initialization)	3
	Declaring an existing variable (JS-Declaration)	2

only to JavaScript (Table 3.4.5). For instance, Incorrect/missing input validation and Incorrect/incomplete data processing caused the majority of bugs we observed, 50%¹ and 23%,² respectively. It has to be said that this prevalence is due to the ESLint project, which is a linting tool, essentially a code validator. While ESLint may not be representative of the majority of JavaScript web services, it is a very popular JavaScript linting tool, being recommended by multiple comparative studies above other

¹(72+51+2+7+4+7+61+7+16+2)/453%

²(27+64+1+10)/453%

tools like JSLint, JSHint, and JSCS.³⁴ Moreover, ESLint is supported by JetBrains in the WebStorm IDE by Vue.js to validate their templates, and by Facebook’s React to help enforce their coding rules. This adoption suggests that major companies recognize input validation/data processing tasks as vital throughout software development.

Another relevant source of bugs is due to missing type checks, a construct that is particularly problematic due to the dynamically-typed nature of JavaScript, which makes it easier for developers to introduce bugs if they are, for instance, more familiar with strongly typed languages [85, 45]. Researchers have proposed approaches for addressing this class of errors and finding ways to prevent them [10, 119, 74, 99, 8, 9], which suggests that this class of errors deserves considerable attention.

Benchmark for Testing Techniques

Various fields of testing research can benefit from BUGSJS. First, our benchmark includes more than 25*k* JavaScript test cases, which makes it a rather large dataset for different regression testing studies (*e.g.*, test prioritization, test minimization, or test selection). Second, BUGSJS can play a role in supporting research into software oracles (*e.g.*, automated generation of semantically-meaningful assertions), as it contains all test suites’ evolution as well as examples of real fixes made by developers. Additionally, these can be used to drive the design of automated test repair techniques [117, 44]. Finally, test generation or mutation techniques for JavaScript can be evaluated on BUGSJS at a low cost, since pre-computed coverage information is available for use.

Bug prediction using static source code analysis

To construct reliable bug prediction models, training feature sets are extracted from the source code, consisting of instances of buggy and healthy code, and static metrics. BUGSJS can support these studies since it streamlines the hardest part of constructing the training and testing datasets, that is, determining whether a given code element is affected by a bug. As such, the cleaned fixes included in BUGSJS make this task much easier. Also, the availability of both uncleaned and cleaned bug-fixing patches in the dataset can allow for assessing the sensitivity of the proposed models to the noise. In addition, some of the most important static code metrics are readily available as pre-computed data. Several studies have attempted to devise reliable static bug prediction models. These models are usually trained and tested on bug datasets containing examples of both buggy and healthy code elements (*e.g.*, functions), and often static measures (*e.g.*, source code metrics) are used as training features. Such datasets can be easily constructed using BUGSJS, since it streamlines the hardest part of constructing the training and testing datasets; that is, determining whether a given code element is buggy or not. The cleaned patches included in BUGSJS make this task rather straightforward.

Bug localization

BUGSJS can support the devising of novel bug localization techniques for JavaScript. Approaches that use NLP can take advantage of our benchmark since bugs are readily available to be processed. Indeed, text retrieval techniques are used to formulate a

³<https://www.sitepoint.com/comparison-javascript-linting-tools/>

⁴<https://codekitapp.com/help/jshint/>

natural language query that describes the observed bug. To this aim, BUGSJS contains pointers to the natural language bug description and discussions for several hundreds of real-world bugs. Similarly, BUGSJS will be of great benefit to other popular bug localization approaches, *e.g.*, the spectrum-based techniques [29, 98, 96, 97] because all the necessary data—test case outcomes, code coverage, and bug information—are readily available.

Automated program repair

Automated program repair techniques aim at automatically fixing bugs in programs, by generating a large pool of candidate fixes, to be later validated. The manually cleaned patches available in BUGSJS can be used as learning examples for patch generation in novel automated program repair for JavaScript. Also, BUGSJS provides an out-of-the-box solution for automatic dynamic patch validation. Detailed classification of the bugs according to our bug taxonomy and the bug-fix types provides additional useful information for this kind of research.

3.5.2 Limitations

BUGSJS only includes server-side JavaScript applications developed with the Node.js framework. As such, experiments evaluating the client-side (*e.g.*, the DOM) are not currently supported. While our survey revealed a large number of subjects being used for evaluating such techniques, the majority of these programs could not be directly included in our proposed benchmark.

Indeed, in the JavaScript realm, the availability of many implementations, standards, and testing frameworks poses major technical challenges with respect to devising a uniform and cohesive bug infrastructure. Similar reasoning holds for selecting Mocha as a reference testing framework.

Running tests for browser-based programs may require complex and time-consuming configurations. When dealing with a large and diverse set of applications, achieving isolation would require automating every single configuration for all possible JavaScript development and testing frameworks, which is a cumbersome task. Clearly, this is a potential limitation and bias for experiments that use BUGSJS, and we are considering a future revision of the benchmark to support other environments as well. We must note, however, that due to the mentioned specialties of the JavaScript ecosystem, we do not expect to be able to fully cover the plethora of different execution environments. Nevertheless, all the subjects included in BUGSJS have been previously used by at least one work in our literature survey (*e.g.*, BOWER, SHIELDS, KARMA, NODE-REDIS, and MONGOOSE are all used in bug-related studies).

3.5.3 Threats to Validity

The main threat to the *internal validity* is the bugs included in the benchmark. The manual classification of the bugs was carried out by the researchers (including the author of the dissertation), which includes the risk of subjective judgment. We tried to reduce this by having several researchers analyze the source code, the bug issue, and the fix commit independently (individually) and perform this classification. In those cases where there was no consensus, the researchers conducted further dialogue and agreed on the controversial issues.

A similar threat existed during the creation of the taxonomy, which the researchers (including the author of the dissertation) tried to reduce with systematic and structured procedures, multiple interactions, and further cooperation and discussion.

There are other, *external threats* that can be reduced by further analysis and by expanding the benchmark. One aspect is that the dataset currently contains only 10 web applications, which does not cover all relevant programs, so some generalizations may not hold for other projects. Also, other relevant classes of bugs might be unrepresented or underrepresented within our benchmark because the rate of ESLint-bugs is very large compared to the size of the benchmark (and this can bias the result). Nevertheless, we tried to mitigate this threat by selecting applications with different sizes and different domains. Despite all this, I feel (along with my other colleagues) that the results are relevant and can serve as a starting point for further research.

With respect to the *reproducibility* of our results, all classifications, subjects, and experimental data are available online, making the analysis reproducible.

3.6 Potential Uses of Benchmark: Bug Localization

The goals of this section were to demonstrate the usefulness of the framework/benchmark and examine the performance of different SBFL algorithms using BugsJS. In particular, my goal was to find out if any of the most popular algorithms (Tantulum [54], Ochiai [3], and DStar [130]) produce better overall localization efficiency than the others.

Furthermore, I was interested in the different types of bug fixes and if these affect the algorithms' fault localization efficiency. Simply by looking at how these algorithms compute the fault localization ranks (they look at code coverage details of passed and failed test cases), one could not easily predict if the bug (and the corresponding code fix) type could have any significant influence on the algorithms' performance. However, if we take into account that the developers have a tendency to make mistakes to various extents for different code constructs (more often in conditional statements than assignment expressions [91], for instance), knowledge about how automated SBFL behaves in different situations could help design more specialized algorithms that could better serve the programmers.

Despite the fact that JavaScript is a popular programming language, automated fault localization in this language is less researched than in Java or C/C++, for instance. Hence, my work is a novel contribution in this regard as well.

3.6.1 Overview

Figure 3.10 contains the overview of my process to obtain empirical measurement data. For each bug, it includes several related code revisions and sets of test cases and enables the individual execution of these versions. Execution information from related test cases can be obtained including per-test code coverage and test results.

Three code revisions were used ("bug-tags") for each bug in the benchmark:

- a) *buggy*: the parent commit of the revision in which the bug was fixed,
- b) *fixed* contains only the production code changes introduced to fix the bug, applied to the buggy revision, and

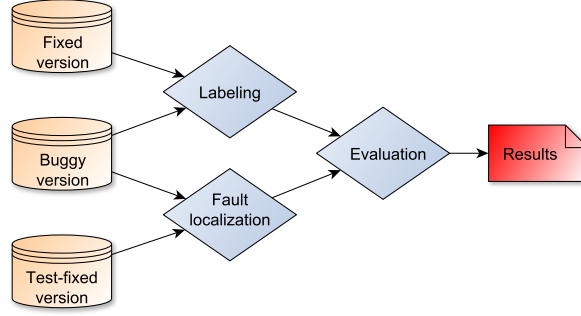


Figure 3.10: Experiment overview

- c) *only test-fix* contains only the tests introduced in the bug-fixing commit, applied to the buggy revision.

Using this set of data, three steps were necessary to obtain the final results for answering our research questions:

- a) Bug labeling, in which we calculated the function change sets between the buggy and fixed revisions for all bugs and determined the respective bug-fix labels.
- b) Coverage measurement and fault localization score calculation, in which we collected coverage data and test results, and calculated the fault localization rank values for each source code element (function).
- c) Data evaluation.

Figure 3.11 shows the overall process of assigning bug-fix types to the benchmark bugs. Following the preliminary classification done by the benchmark authors [42], I started with the bug-fix types of Pan *et al.* [91]. Starting from the *buggy* and *fixed* code revisions, I used GitHub’s *split diff* view to identify the changed code lines and the corresponding JavaScript functions with the help of the code coverage data. More information on the labeling process and the labels themselves are found in Section 3.4.5.

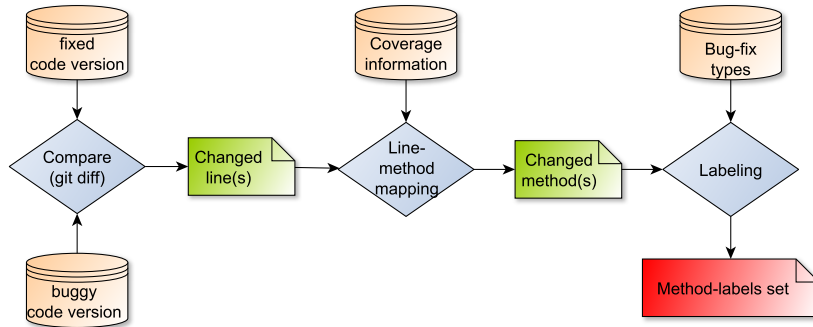


Figure 3.11: Bug labeling process

Finally, Figure 3.12 shows how I calculated the fault localization results for the benchmark bugs. I used the “per-test” measurement feature of the benchmark that allows computing function-level code coverage data for each test case separately along with the test case outcomes. I ran the tests separately in the two tagged versions for each bug (*buggy* and *only test-fix*) and then compared the test results (pass or fail) of these two versions. I omitted those tests that failed in both cases, because these failed

tests were most probably not related to the relevant bugs. The inputs to the SBFL algorithms were these filtered test results and the function-level code coverage data of the *buggy* versions.

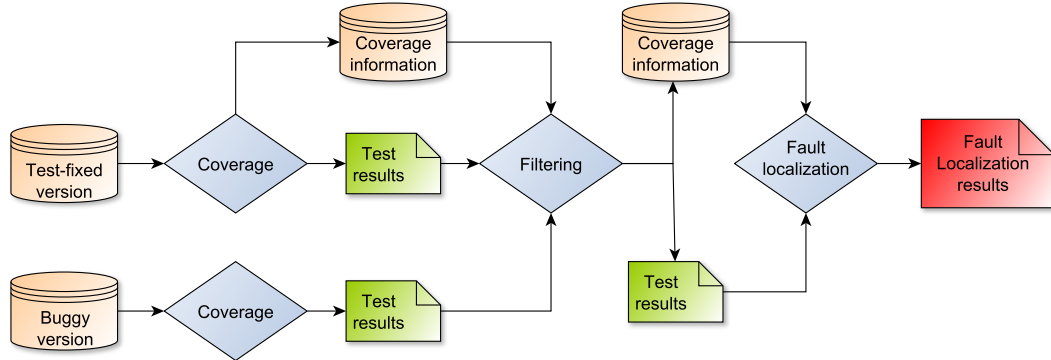


Figure 3.12: Fault Localization process

To store the program execution information, we calculated the coverage matrix and the spectrum metrics, which are the same as those presented in Section 2.4. There have been numerous formulae proposed in the literature [29, 94, 24, 96, 6], but some of the most often used ones are Tarantula [54], Ochiai [3], and DStar [130], so we implemented these to be used in our experiments (see Section 2.2).

I wrote in detail about how to compare the results in the previous thesis point (Sec. 2.8). In the case of BUGSJS, I took into account two aspects in the evaluation: absolute average rank (wasted effort) and Top-N. I distinguished between bugs where the minimum of faulty methods' rank is equal to 1 (*Top-1*), it is less or equal to three (*Top-3*), less or equal to five (*Top-5*), less or equal to ten (*Top-10*), and when it is over ten (*Other*), commonly referred to as *Top-N* [95]. The variant of this is the non-accumulating variant, where there are no overlaps between the categories, that is, a defect can belong to a non-overlapping interval of [1], (1, 3], (3, 5], (5, 10], or (10, ...].

These evaluation metrics help judge the efficiency of the three algorithms and highlight the differences (in efficiency) between them.

3.6.2 Results

Table 3.7 shows the *average ranks* obtained per project and per algorithm. First, we can observe that the three approaches typically produced similar results, apart from some outlier cases, most notably, DStar on the Hexo project. This is not so surprising as other studies have already found that this algorithm can produce extreme ranks in some cases. It can be stated that, overall, Ochiai obtained the best results, i.e. the lowest average ranks. It is interesting to note that there were no cases where the average rank of Tarantula was lower than that of Ochiai, but surprisingly DStar gave worse results than Ochiai only on Hexo and in all other cases it was better. Without the outlier, DStar would have been the clear winner among the three methods. Other works that analyzed these algorithms but on different languages, obtained similar results [96, 130, 2, 83].

Table 3.7: Average ranks

Project	Bower	Shields	Hexo	Hessian.js	Express	Pencilblue	Eslint	All
Tarantula	25.83	5.83	3.25	4.81	8.10	1.83	20.39	18.24
Ochiai	19.17	5.83	3.00	3.88	7.94	1.67	19.90	17.73
DStar	17.50	5.17	80.88	3.00	7.94	1.67	19.90	20.47

To verify if there is a statistically significant difference between the algorithms, I used the Wilcoxon signed-rank test on the fault localization ranks [128]. This is a non-parametric statistical hypothesis test, which is used to compare two samples to assess whether their population mean ranks differ. I applied it to determine whether these two dependent samples were selected from populations having the same distribution. In particular, I wanted to know if any of the three algorithms is significantly better than the others. The significance level was chosen to be $\alpha = 0.05$, and we set the null hypothesis (H_0) to be so that a fault localization algorithm A is not significantly better than algorithm B .

Wilcoxon signed-rank test gave us a $p = 0.00001$ value for the Tarantula-Ochiai pair, so we reject H_0 and accept H_1 , that is, there is a difference in efficiency between the two algorithms. The same goes for Tarantula and Dstar, the test gave us the value of $p = 0.00043$. For the Ochiai - DStar pair, the p value was bigger than α ($p = 0.70975$), therefore, H_0 was adopted in this case, that is, the two approaches produce similar results (despite some of DStar's extreme rankings).

The average of the ranks can be easily misleading as high average rank values can be caused by some extremely bad values and they can influence the overall results greatly (such as the case with DStar and Hexo). So, I also investigated a set of rank positions that we believe are particularly important. As mentioned earlier, the practical usability of SBFL depends on the position of the faulty element (in absolute terms, not in relative), and developers tend to investigate only the top 5 or at most the top 10 elements. I used the top-N ranking scheme in this set of experiments.

The top-N values for all the bugs are shown in Table 3.8. Overall, almost 30% of bugs have a rank of 1 and 90% of them have ranks of 10 or less. It can be observed that Ochiai and DStar continue to produce similar results (and this is especially true for the top-1, top-3, and top-5 categories). This also indicates that DStar is not performing as badly as the average rank showed above, which was due to an outlier.

Table 3.8: Top-N ranks

	Tar.	(%)	Ochiai	(%)	DStar	(%)
top-1	96	28.6	101	30.1	100	29.8
top-3	206	61.3	214	63.7	214	63.7
top-5	255	75.9	264	78.6	264	78.6
top-10	300	89.3	306	91.1	303	90.2
other	36	10.7	30	8.9	33	9.8

Figure 3.13 shows this data in a slightly different manner. Here, I did not use the top-N values as defined above, but their non-accumulating variant. This means that I counted the cases where a particular bug fell into a non-overlapping interval of $[1]$, $(1, 3]$, $(3, 5]$, $(5, 10]$, or $(10, \dots]$. We see from Figure 3.13, for example, that there are 48 bugs that rank between the 3 to 5 interval (right closed) based on Tarantula.

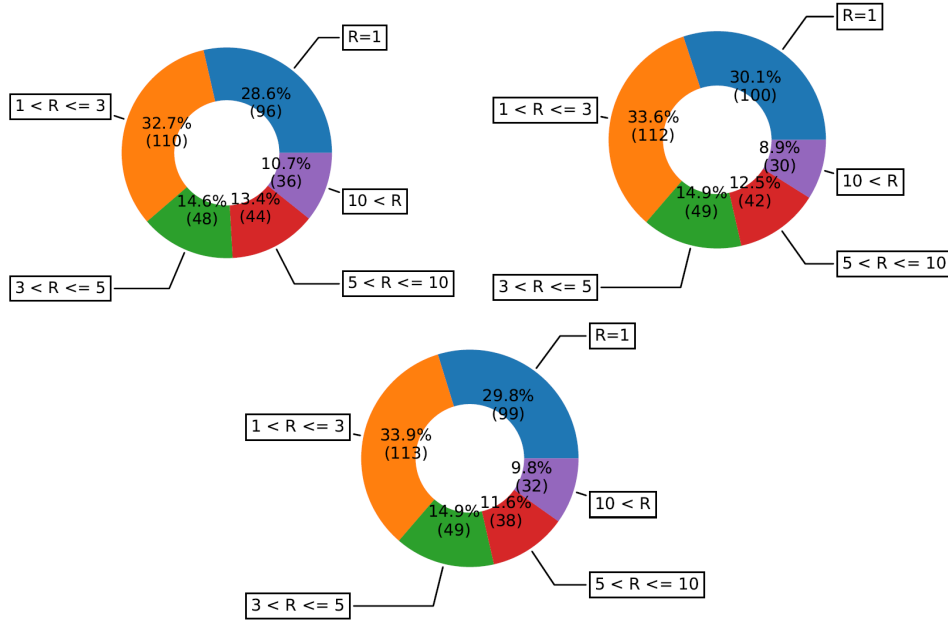


Figure 3.13: Interval of algorithms (*Tarantula*, *Ochiai*, and *DStar*)

Overall, $\approx 30\%$ of bugs belong to the best [1] category, but most elements are found in interval (1, 3] (nearly 33%). It can be observed that each algorithm produces similar results, however, *Ochiai* and *DStar* contain more elements in the segments [1], (1, 3], (3, 5] than *Tarantula*. In addition, *Tarantula* assigned most of the elements to the two worst groups: (5, 10]: 44 functions, 13.4% and (10, ...]: 36 functions, 10.7%. This information confirms my earlier findings based on average ranks (Table 3.7) that the results are similar, but *Tarantula* slightly lags behind.

Concluding these results, I found that there is no significant difference between the *Ochiai* and *DStar* fault localization algorithms, but *Tarantula* is different (slightly worse) than the other two. Both the average rankings (Table 3.7) and the results of the top-N analysis (Table 3.8 and Figure 3.13) support this conclusion. In terms of the top-N positions, in about 30% of the cases, perfect localization could be achieved, and in about 90% of the cases, the fault was among the top-10 positions.

Based on the bug-fix categories, I assigned labels to each of the JavaScript functions participating in the investigated bugs. Table 3.9 shows the associated statistics for *Tarantula*: for each bug category and subject system, I give the number of functions that got assigned to that category.

Note that the number of labels (412) is greater than the number of bugs (336). This is because there are several functions that have multiple tags associated with them. The three algorithms produced very similar results (hence I only give numbers for one), with one difference (and it is marked with * in Table 3.9): in the ESLint project, 75 of the modified functions have an AS bug-fix type based on *Tarantula*, but it is 76 based on *Ochiai* and *DStar*. This may occur because it is not guaranteed that all fault localization algorithms will be assigned the lowest rank for the same method and this can cause the labels to differ. This difference also affects aggregate values, and these are also marked in the table.

The numbers of IF (176) and AS (102 or 103) appear to be prominent, and MC (65) is quite high as well, but there are also very rare types (e.g. SW - 5 or TY - 1).

Table 3.9: Overall bug-fix type statistics based on Tarantula

Bug-fix type	Bower	Shields	Hexo	Hessian	Express	Pencilblue	Eslint	Total
IF	2	0	5	4	12	2	151	176
AS	2	3	9	2	9	2	75*	102*
MD	0	0	0	0	1	0	23	24
LP	0	0	0	0	2	0	6	8
MC	0	0	1	0	3	1	60	65
SQ	0	0	1	3	6	0	19	29
SW	0	0	0	0	1	0	4	5
TY	0	0	0	0	0	0	1	1
CF	0	1	0	0	0	0	1	2
Total	4	4	16	9	34	5	340*	

The goal of the experiments presented in this section was to find out if there are any bug-fix types for which the SBFL algorithms' efficiency is significantly different (either better or worse) than for the others. Similar to the overall rank analysis from the previous section, I counted the number of bugs belonging to the different top-N categories and separated them according to the bug labels. That is, I wanted to see what kinds of labels were assigned to the lowest-ranked (modified) functions for each bug, and how they were distributed among the top-N categories. If more than one function was associated with a bug only the lowest rank (and labels) were considered, and if there was more than one label associated with a function then that function was accounted for each label.

Table 3.10 shows the distribution of the labels obtained for the three algorithms by category. An element in the table tells us how many bugs there are where the least-ranked modified function contains the given bug-fix type and the rank falls within a given top-N range. For example, there are 115 bugs where the lowest Tarantula rank is 3 or less and this function has an IF bug-fix label.

The relative versions of these numbers are provided in Table 3.11. Here, we can see what percentage of items with the specified label are in the top-N category. For example, 65.3% of least-ranked modified functions with the IF tag have a rank less than or equal to 3. When analyzing the most common labels, it is interesting to note that IF is slightly better, and AS is slightly worse than the top-N results in Table 3.8. Also, the results of SQ are interesting because the number (and proportion) of labels in the top-1 category is very low while its *other* category is high. Other labels produced low element numbers, so their general interpretation is less relevant.

As in the previous section, I counted the number of labels occurring in the non-overlapping rank ranges as well. The associated results can be seen in Figure 3.14. The more common labels (IF and AS) also show that there are nearly as many labels in [1] interval as labels with a value rank of 2 to 3. In addition, Tarantula is a bit less efficient at finding buggy functions than Ochiai or DStar. For example, the number of labels in (1, 3] is 131 for Tarantula, 136 for Ochiai, and 141 for DStar. We can again see that the occurrence of SQ labels in the lower ranges is quite low compared to the others, while IF is the opposite: it has a high proportion of labels in the top categories and fewer in the worse rank positions.

Figure 3.14 shows that there are certain bug types that Tarantula finds easier.

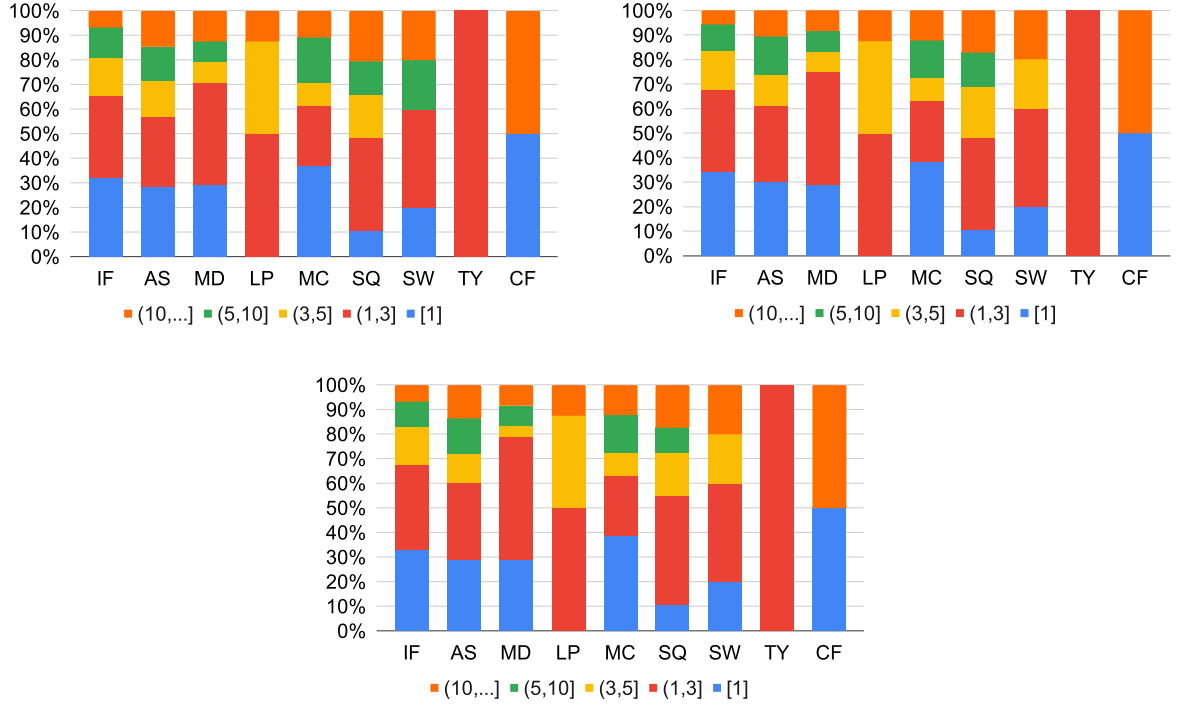


Figure 3.14: Interval statistics (*Tarantula*, *Ochiai* and *DStar*)

Although this can be misleading if there is not enough data, e.g. on TY *Tarantula* performs really well, however, as Table 3.10 shows, there is only one bug-fix with this label.

As we can see in Figure 3.14, the labels IF, AS, and MD in the [1] interval have similar results as R=1 (Top 1) in Figure 3.13.

To verify if this data shows any statistically significant trends, I used Fisher's exact test. It is a statistical significance test, which is one of the non-parametric methods and is used in the analysis of contingency tables [7]. I counted the number of labels per metric provided by the three SBFL algorithms in Table 3.10, and in order to perform the test, I created contingency tables for each (*bug-fix types*, *non-overlapping interval*, *algorithm*) configuration.

Table 3.12 shows the p values that the labels get in the different intervals beside the given algorithm for Fisher's exact test. The null hypothesis is that the ratio of belonging to a top-N range is not higher for one label than for others. If this value is less than the chosen significance level, 0.05, then I reject the null hypothesis and I can say that the ratio of belonging to a range is different (higher or lower) for a given label than the others, *i.e.* the proportion of the labels in the range is different. This test only determines whether there is a difference in probability, it does not determine its direction.

In Table 3.12, I highlighted the cases where the difference was significant according to the test. I can make the following observations based on this data. First, in the top-1 category, SQ is significantly different (worse) with all algorithms (this type seems to be more difficult to localize), which we observed from previous data as well. In the top-5, top-10, and other categories the label IF is significantly different from the other labels; in the top-5 and top-10 they are likely to be found by the SBFL algorithms. Also, in the *other* section, it is significantly different in the opposite direction (there are

Table 3.10: Number of labels (per metric)

Name	Top-1 (#)			Top-3 (#)			Top-5 (#)			Top-10 (#)			Other (#)		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	57	60	58	115	119	119	142	147	146	164	166	164	12	10	12
AS	29	31	30	58	63	62	73	76	74	87	92	89	15	11	14
CF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
MD	7	7	7	17	18	19	19	20	20	21	22	22	3	2	2
MC	24	25	25	40	41	41	46	47	47	58	57	59	7	8	8
SQ	3	3	3	14	14	16	19	20	21	23	24	24	6	5	5
SW	1	1	1	3	3	3	3	4	4	4	4	4	1	1	1
LP	0	0	0	4	4	4	7	7	7	7	7	7	1	1	1
TY	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0
All	122	128	125	253	264	266	311	323	321	366	374	369	46	39	44

Table 3.11: Percents of labels (per metrics)

Name	Top-1 (%)			Top-3 (%)			Top-5 (%)			Top-10 (%)			Other (%)		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	32.4	34.1	33.0	65.3	67.6	67.6	80.7	83.5	83.0	93.2	94.3	93.2	6.8	5.7	6.8
AS	28.4	30.1	29.1	56.9	61.2	60.2	71.6	73.8	71.8	85.3	89.3	86.4	14.7	10.7	13.6
CF	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0
MD	29.2	29.2	29.2	70.8	75.0	79.2	79.2	83.3	83.3	87.5	91.7	91.7	12.5	8.3	8.3
MC	36.9	38.5	38.5	61.5	63.1	63.1	70.8	72.3	72.3	89.2	87.7	87.7	10.8	12.3	12.3
SQ	10.3	10.3	10.3	48.3	48.3	55.2	65.5	69.0	72.4	79.3	82.8	82.8	20.7	17.2	17.2
SW	20.0	20.0	20.0	60.0	60.0	60.0	60.0	80.0	80.0	80.0	80.0	80.0	20.0	20.0	20.0
LP	0.0	0.0	0.0	50.0	50.0	50.0	87.5	87.5	87.5	87.5	87.5	87.5	12.5	12.5	12.5
TY	0.0	0.0	0.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0

fewer labels here). Table 3.11 can be used to determine the direction of significance: for SQ, the number of elements in the top-1 category/section is low as opposed to IF where its ratio is high. Summarized: there are two bug-fix types that are significantly different from the others in terms of how successfully the SBFL algorithms can locate them. Faults that require modifications of SQ type are less likely to be successfully localized at very high-rank positions by the algorithms, while faults belonging to the IF category are ranked higher than other types in the top-5 and top-10 ranges.

Table 3.12: Significance in top-N based on Fisher’s exact test

Name	top-1			top-3			top-5			top-10			other		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	0.326	0.237	0.330	0.406	0.214	0.254	0.037	0.030	0.031	0.017	0.027	0.036	0.017	0.027	0.036
AS	0.804	0.902	0.806	0.157	0.554	0.342	0.291	0.217	0.103	0.206	0.697	0.272	0.206	0.697	0.272
MD	1.000	1.000	1.000	0.516	0.281	0.131	0.809	0.799	0.619	0.741	1.000	1.000	0.741	1.000	1.000
LP	0.112	0.113	0.113	0.471	0.467	0.463	0.686	1.000	0.691	1.000	0.551	0.597	1.000	0.551	0.597
MC	0.183	0.145	0.141	0.780	0.889	0.888	0.348	0.251	0.258	1.000	0.362	0.661	1.000	0.362	0.661
SQ	0.019	0.012	0.019	0.109	0.074	0.316	0.261	0.242	0.489	0.118	0.176	0.218	0.118	0.176	0.218
SW	1.000	1.000	1.000	1.000	1.000	1.000	0.600	1.000	1.000	0.448	0.393	0.432	0.448	0.393	0.432
TY	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
CF	0.505	0.519	0.514	1.000	1.000	1.000	0.431	0.389	0.396	0.211	0.180	0.202	0.211	0.180	0.202

To summarize, I found that there are certain bug-fix types that seem to be harder to localize by the current algorithms (for instance, operation sequence change), while

some others are easier than the rest (*if* condition-related bugs). The analysis of bug-fix types is a first step and will be extended to include other bug categorizations that include bug causes as well, and hence contribute more directly to designing better fault localization algorithms.

There is a lower-level grouping of bug-fixing types. In this case, each group is divided into several smaller categories. For example: $IF \rightarrow IF - CC$: changing the condition expression of an if condition, $IF - RBR$: removing an else branch, etc.

In one of our analyses, we looked for the answer to the question of whether there are any bug-fix types within the if-related (IF) or sequence-modification (SQ) classes on which any of the three most popular SBFL algorithms perform significantly better or worse. I used *Tarantula*, *Ochiai*, and *DStar* algorithms, but I present only *Ochiai* because these give very similar results.

The following categories are analyzed in detail [91]: a) **IF-ABR**: adding an *else* branch b) **IF-APC**: addition of precondition check c) **IF-APCJ**: addition of precondition check with jump d) **IF-APTC**: addition of post-condition check e) **IF-RMV**: removing an if predicate f) **IF-CC**: changing the condition expression of an if condition g) **IF-RBR**: removing an *else* branch h) **SQ-AMO**: addition of operations in an operation sequence of method calls to an object i) **SQ-RMO**: removal of operations from an operation sequence of method calls to an object j) **SQ-AFO**: adding one or more operations in a sequence of setting the object fields of the same object k) **SQ-RFO**: removing one or more operations from a sequence of setting the object fields of the same object l) **SQ-AROB**: adding or removing method calls from a construct body.

If we take a look at Figure 3.15 and Table 3.13 we can see that bug-fixes labeled IF-RMV are more likely to be put in the top-3 category, i.e. have a rank of 1,2, or 3. Furthermore, bugs that have their fixes labeled with IF-CC are found easier by the algorithms and they are more likely to be put in Top-5 or Top-10.

Although if we take a look at Figure 3.16 and Table 3.13 we can observe that only SQ-AMO and SQ-AFO were found in the [1] interval, which means they are easier to find for *Ochiai*. Therefore, bugs that were fixed with setting object fields (SQ) are still significantly worse than other types in terms of ranking, though SQ-AMO and SQ-AFO seem to be better than other SQ types.

Table 3.13: Number of labels in Top-N categories (*T* – *Tarantula*, *O* – *Ochiai*, *D* – *DStar*)

Types	Top-1 (#)			Top-3 (#)			Top-5 (#)			Top-10 (#)			Other (#)		
	T	O	D	T	O	D	T	O	D	T	O	D	T	O	D
IF-ABR	3	3	3	4	5	6	10	11	11	12	13	13	1	0	0
IF-APC	10	11	11	20	21	21	28	28	28	30	30	30	2	2	2
IF-APCJ	14	13	12	28	27	27	32	33	33	34	35	34	5	3	4
IF-APTC	0	0	0	0	0	0	1	1	1	2	2	2	0	0	0
IF-CC	29	32	30	61	64	63	80	82	80	95	96	94	5	5	7
IF-RBR	1	1	1	1	1	1	1	2	2	3	3	3	1	1	1
IF-RMV	8	8	8	16	17	17	20	20	20	20	20	20	1	1	1
SQ-AFO	3	3	3	6	6	6	7	7	7	8	8	8	4	4	4
SQ-AMO	1	1	1	1	1	1	1	2	2	2	2	2	0	0	0
SQ-AROB	0	0	0	5	5	6	8	8	9	10	11	11	2	1	1
SQ-RFO	0	0	0	3	3	4	4	4	4	4	4	4	1	1	1
SQ-RMO	0	0	0	2	2	2	2	2	2	2	2	2	0	0	0

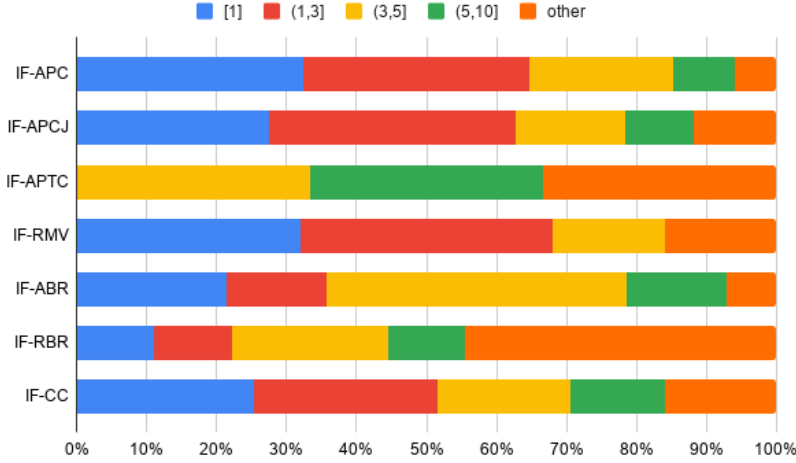


Figure 3.15: Ochiai Ranks in IF labels

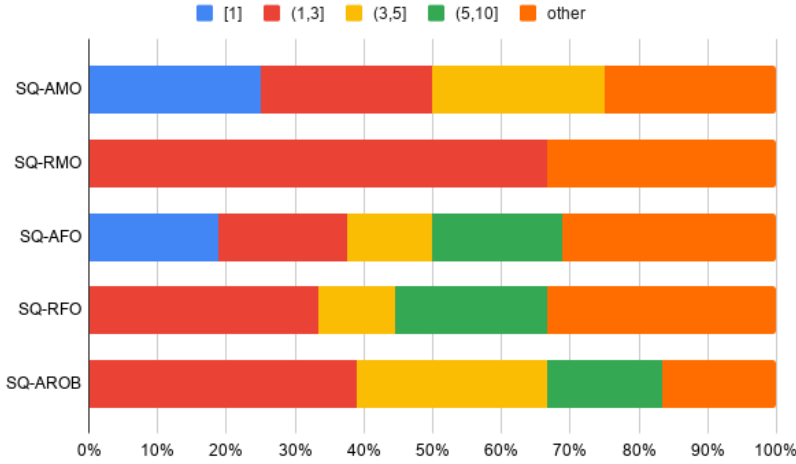


Figure 3.16: Ochiai Ranks in SQ labels

I used Fisher’s exact test to decide if there is any significant difference between these bug-fixes types. Let the null hypothesis H_0 be that fault localization algorithms produce similar results using these bug-fix labels. Let the H_1 hypothesis be that one of them is significantly different.

Table 3.14 shows that IF-ABR in Top-3 and IF-RBR in Top-5 are significantly different, hence in these two cases, we reject the null hypothesis. Therefore, IF-RBR and IF-ABR are worse in terms of SBFL algorithm efficiency and as we can see, none of the SQ-cases is under the significance level ($\alpha = 0.005$).

Summarized, within the IF bug-fix type, two types are significantly different from the other IF types. We can conclude that these are less likely to be successfully localized in Top-3 and Top-5 categories. On the other hand, there is no significant difference in the SQ category. In conclusion, bugs with sequence-related bug fixes are difficult to find for SBFL algorithms regardless of their subtypes.

Investigating the possible explanations for these phenomena is among my avenues for future work. Once we understand the underlying causes of the different behaviors of the algorithms on different bug-fix types (and eventually other bug categories), we

Table 3.14: Significance in Top-N within the IF and SQ categories based on Fisher exact test

Name	Top-1	Top-3	Top-5	Top-10	Other
IF-ABR	0.7583	0.0182	0.7083	0.6033	0.6033
IF-APC	0.8366	1.0000	0.3099	0.6984	0.6984
IF-APCJ	0.4501	0.2716	1.0000	0.1445	0.1445
IF-APTC	1.0000	0.1327	0.3299	1.0000	1.0000
IF-CC	0.6542	0.6666	0.5909	0.4156	0.4156
IF-RBR	1.0000	0.1399	0.0194	0.2427	0.2427
IF-RMV	0.4629	0.2369	0.1342	1.0000	1.0000
SQ-AFO	0.1250	0.7241	0.4713	0.3774	0.3774
SQ-AMO	0.2311	1.0000	1.0000	1.0000	1.0000
SQ-AROB	0.2713	0.4670	1.0000	1.0000	1.0000
SQ-RFO	1.0000	1.0000	0.6431	1.0000	1.0000
SQ-RMO	1.0000	0.4882	0.5417	1.0000	1.0000

may be able to design improved algorithms in place of these very generic ones we use currently, which would perform better on multiple types of bugs. Other possible implications of the results include useful insights for researchers working in related fields, such as automated program repair, test generation, and bug prediction.

3.7 Related Works

3.7.1 C, C++, and C# Benchmarks

The Siemens benchmark suite [51] was one of the first datasets of bugs used in testing research. It consists of seven C programs, containing manually seeded faults. The first widely used benchmark of real bugs and fixes is the SIR (Software-artifact Infrastructure Repository) [28], which includes the Siemens benchmark, and extends it with nine additional large C programs and seven Java programs. SIR also features test suites, bug data, and automation scripts. The benchmark contains both real and seeded faults, the latter being more frequent.

Le Goues et al. [62] proposed two benchmarks for C programs called ManyBugs and IntroClass,¹ which include 1,183 bugs in total. The benchmarks are designed to support the comparative evaluation of automatic repair, targeting large-scale production (ManyBugs) as well as smaller (IntroClass) programs. ManyBugs is based on nine open-source programs (5.9M LOC and over 10k test cases) and it contains 185 bugs. IntroClass includes 6 small programs and 998 bugs.

Rahman et al. [101] examined the OpenCV project mining 40 bugs from seven out of 52 C++ modules into the benchmark *Pairika*. The seven modules analyzed contain more than 490k LOC, and about 11k test cases, and each bug is accompanied by at least one failing test.

Lu et al. [70] proposed *BugBench*, a collection of 17 open-source C/C++ programs containing 19 bugs pertaining to memory and concurrency issues.

¹<http://repairbenchmarks.cs.umass.edu/>

Codeflaws [110] contains nearly 4k bugs in C programs, for which annotated ASTs with annotated syntactic differences between buggy and patch code are provided.

3.7.2 Java Benchmarks

Just et al. [55] presented Defects4J, a bug database and extensible framework containing 357 validated bugs from five real-world Java programs. BUGSJS shares with Defects4J the idea of mining bugs from the version control history. However, BUGSJS has some additional features: subject systems are accessible in the form of *git* forks on a central GitHub repository, which maintains the whole project history. Further, all programs are equipped with pre-built environments in form of the Docker containers. Moreover, I also provide a more detailed analysis of subjects, tests, and bugs.

Bugs.jar [105] is a large-scale dataset intended for research in the automated debugging, patching, and testing of Java programs. Bugs.jar consists of 1,158 bugs and patches, collected from eight large, popular open-source Java projects.

iBugs [22] is another benchmark containing real Java bugs from bug-tracking systems originally proposed for bug localization research. It is composed of 390 bugs and 197k LOC coming from three open-source projects.

3.7.3 Multi-language Benchmarks

QuixBugs [68] is a benchmark suite of 40 confirmed bugs used in program repair experiments targeting Python and Java with passing and failing test cases.

BugSwarm [27] is a recent dataset of real software bugs and bug fixes to support various empirical testing experiments, such as test generation, mutation testing, and fault localization.

Code4Bench [73] is another cross-language benchmark comprising C/C++, Java, Python, and Kotlin programs among others. Code4Bench also features a coarse-grained bug classification based on an automatic fault localization process for which faults were classified only into three groups, namely addition, modifications, and deletion. In contrast, BUGSJS focuses on JavaScript bugs, for which we provide a fine-grained analysis based on a rigorous manual process.

3.7.4 Benchmarks Comparison

We summarize the related benchmarks and compare their main features to BUGSJS in Table 3.15. The table includes the language(s) in which the programs were written and the kinds of bugs the benchmarks contain. Further, the table indicates whether the modified versions have been cleaned from irrelevant changes, e.g., achieving the isolation property, and whether the benchmark includes quantitative or qualitative analyses of the faults. This information was retrieved from the papers in which the benchmarks were proposed first.

The table highlights that BUGSJS is the only benchmark that contains JavaScript programs. This paper also provides both a quantitative analysis of the benchmark, and a qualitative analysis of the bugs (from which a taxonomy was derived) and the bug fixes (by comparing them with existing taxonomies). For instance, in the case of Defect4J, the original paper proposed only the benchmark [55], whereas a quantitative analysis was added in a subsequent paper by Sobreira et al. [115]. More qualitative analyses

Table 3.15: Properties of the benchmarks

Benchmark	Language(s)	Fault Type	# Bugs	Isolation	Quantitative An.	Qualitative An.
Siemens/SIR [28]	C/Java	Real/seeded	662	✓	✗	✗
ManyBugs [62]	C	Real	185	✓	✓	✓
IntroClass [62]	C	Real	998	✓	✓	✓
Pairika [101]	C++	Real	40	✓	✓	✗
BugBench [70]	C/C++	Real	19	✗	✗	✓
Defects4J [55]	Java	Real	357	✓	✓	✓
Bugs.jar [105]	Java	Real	1,158	✓	✗	✗
iBugs [22]	Java	Real	390	✗	✓	✗
QuixBugs [68]	Java/Python	Seeded	40	✓	✓	✓
Codeflaws [110]	C	Real	3,902	✓	✗	✓
BugSwarm [27]	Java/Python	Real	3,165	✗	✓	✗
Code4Bench [73]	Multiple	Real	N/A	✗	✓	✗
BugsJS	JavaScript	Real	453	✓	✓	✓

were also made by Sobreira et al. [115] and Motwani et al. [82], who independently propose two orthogonal classifications of repair.

To summarize, BUGSJS is the first benchmark of bugs and related artifacts (e.g., source code and test cases) that targets the JavaScript domain. In addition, BUGSJS can be differentiated from the previously-mentioned benchmarks in the following aspects: (1) the subjects are provided as *git* forks with complete histories, (2) a framework is provided with several features enabling convenient usage of the benchmark, (3) the subjects and the framework itself are available as GitHub repositories, (4) Docker container images are provided for easier usage, (5) the bug descriptions are accompanied by their natural language discussions, as well as (6) a manually-derived bug taxonomy and a comparison with an existing bug-fixes taxonomy.

3.8 Conclusions

JS continues to be very popular among developers, so JavaScript programs are becoming the focus of more and more research, and the number of publications dealing with their analysis is constantly increasing. However, reproducing and comparing them without a benchmark is very difficult task.

We wanted to make up for this insufficiency of not having a benchmark, therefore, the BUGSJS bug data set was created, which contains 10, manually validated, real, and popular JavaScript projects (with 453 real bugs). Furthermore, the taxonomy of the bugs was created, which sheds light on their diversity and enables a new research direction in the future (for example the relationship between fault localization efficiency and taxonomy). In addition, individual bug fixes have been classified into bug-fixing pattern categories, the analysis of which can help in the “development” of other research areas (for example bug prediction).

In order to ensure reproducibility and usability, we made the entire data set (including the infrastructure and documentation) and the associated website (which contains the most important general information about the bugs) available for further investigation.

Furthermore, the “use of the data set” was presented through a possible use case (fault localization), we came to the following conclusions from the experiment. I used

three traditional SBFL algorithms in the experiments and found that they produce similar results to each other, but differences could also be observed that are aligned with previous studies for other languages. I examined whether there are differences in fault localization effectiveness between different types of bugs based on the associated bug fixes. I found that there are certain bug-fix types that seem to be harder to localize by the current algorithms (for instance, operation sequence change), while some others are easier than the rest (if condition-related bugs). The analysis of bug-fix types is a first step and will be extended to include other bug categorizations such as bug causes, and hence contribute more directly to designing better fault localization algorithms.

Appendices



Summary in English

There is no program without bug, at least the error has not been revealed yet! That is why testing is an extremely important and resource-intensive part of the software development. The sooner we detect the bugs, the less effort we can repair it, thereby saving a lot of time and resource.

There are many approaches and methods for early automatic, semi-automatic or manual detection of the bugs. In first part of my thesis, I deal with the spectrum-based fault localization (SBFL) algorithms, which is one of the most well-known and popular family of FL concept and in the second part I present a new benchmark, with the help of which the efficiency of the algorithms can be objectively measured and thus compared.

1. part: Fault Localization Algorithms

In this thesis point, I present in detail the operating principle of coverage-based fault localization procedures (so-called *SBFL* algorithms), the most popular methods / formulae, and other relevant literature.

After that, I present two approaches based on graph features, which try to determine the location of the bugs as precisely as possible with input data similar to SBFL. For this, it builds a so-called coverage graph, from which it identifies the most suspicious elements by extracting various neighborhood information.

Then the concept of *unique, deepest call stacks* (UDCS) was introduced, which was illustrated by a small example. It is described in detail how the call frequency information (that can be extracted from the UDCS) can be adapted in the formulas used by the SBFL, which can be used to further increase the efficiency of fault localization algorithms. Finally, I will show how the two concepts described above (graph and frequency-based algorithms) can be combined, thus making use of the potential opportunities of them.

I performed the quantitative evaluation on the *Defects4J* (Java) benchmark, which showed that 1) the two graph-based methods are in many cases more efficient than the examined SBFL methods, 2) the procedures using frequency information give signifi-

cantly better results are achieved than methods based on binary coverage and 3) the best result was obtained by combining the two concepts.

The graph-based approaches gave “mixed” results, there are some SBFL procedures in which these performed better (*e.g.* average ranks: *ENFL*– 34.05 vs. *Barinel*– 36.01) but there were some cases where they achieved minimally worse results (*e.g.* *Ochiai*– 33.98). This resulted in an improvement between -7% and +5%. By using frequency, the improvement in the average rank is between 5 and 101 (for 6 out of 8 algorithms), which represents a relative efficiency increase of 14-74%. The *Jaccard*, *Ochiai* and *Sørensen-Dice* algorithms performed the best with the Δ_{ef}^U configuration. The best result was achieved by “hybrid” *WENFL*, the average rank was 20.59, which achieved a 39-84% improvement compared to SBFL.

Similar results can also be seen when examining the 5 most suspicious elements (Top-5). Among the “classical” SBFL methods, *Ochiai* achieved the best result (367, 46.7%), however, better results are available with both graph-based, frequency-based, and combined methods (eg: *Ochiai* $_{ef}^U$: 380 – 48.3%, *NFL*: 369 – 46.9%, *WENFL*: 389 – 49.5%) Based on these, it can be stated that a significant improvement in the performance of the fault localization algorithms can be achieved by using graph features and call frequencies.

The contributions of the author to this thesis point

The author of the dissertation created the algorithm using a graph-based approach, which works on “traditional” coverage matrices, and he performed the quantitative evaluation of the method. He had a decisive role in the development of the approach based on call chains, its adaptation and the quantification of the results. Combining graph and chain-based methods is also the author’s work, as is the comparison and evaluation of the results. In addition to these, the he was actively involved in processing the literature, planning the experiments and measurements, and writing the publications.

Publications

1. **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes: Fault Localization Using Function Call Frequencies (*The Journal of Systems & Software*, 2022)
2. **B. Vancsics**: NFL: Neighbor-Based Fault Localization Technique (*Proceedings of the 1th International Workshop on Intelligent Bug Fixing*, IEEE, 17-22, 2019)
3. **B. Vancsics**: Graph-Based Fault Localization (*Proceedings of the Computational Science and Its Applications*, Springer International Publishing, 372-387, 2019)
4. **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes: Call Frequency-Based Fault Localization (*Proceedings of the 28th International Workshop on on Software Analysis, Evolution, and Reengineering*, IEEE, 365-376, 2021)

Fault Localization-Related Publications

1. **B. Vancsics**, A. Szatmári and Á. Beszédes: Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs (*Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering*, IEEE, 308-319, 2020)

2. A. Szatmári, **B. Vancsics** and Á. Beszédes: Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? (*Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 16-23, 2020)
3. **B. Vancsics**, T. Gergely and Á. Beszédes: Simulating the Effect of Test Flakiness on Fault Localization Effectiveness (*Proceedings of the 3rd Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 28-35, 2020.)
4. F. Horváth, G. Balogh, A. Szatmári, Q. Idrees Sarhan, **B. Vancsics** and Á. Beszédes: Interacting with interactive fault localization tools (*Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, ACM, 61-63, 2022)
5. F. Horváth, Á. Beszédes, **B. Vancsics**, G. Balogh, L. Vidács and T. Gyimóthy: Using contextual knowledge in interactive fault localization (Empirical Software Engineering, volume: 27, number: 6, 1-69, 2022)
6. F. Horváth, Á. Beszédes, **B. Vancsics**, G. Balogh, L. Vidács and T. Gyimóthy: Experiments with Interactive Fault Localization Using Simulated and Real Users (*Proceedings of the 36th International Conference on Software Maintenance and Evolution*, IEEE, 290-300, 2020)
7. Q. Idrees Sarhan, **B. Vancsics** and Á. Beszédes: Method Calls Frequency-Based Tie-Breaking Strategy For Software Fault Localization (*Proceedings of the 21th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 103-113, 2021)

2. part: BugsJS: a Benchmark of JavaScript Bugs

For the reliable evaluation of fault localization algorithms, a high quality bug data set is essential. There are several similar collections (for example: *Defects4J* – Java), but until now no such data set was available for JavaScript programs.

In the second half of the dissertation, the author outlines how this new data set (BUGSJS) was created, describes in detail the steps involved in the selection of projects (10 projects), the detection and validation of the (453) bugs, its “cleaning” and its analysis according to different aspects.

At the end of the research, a framework was created, using which the different (buggy and fixed) version of the bugs in the data set became easily accessible and reproducible, thereby helping researchers to be able to use JavaScript language programs for their various researches. In addition to these, we created the systematic taxonomy of the bugs in the data set (which covers a hierarchical error categorization) and presented the resulting groups in detail, through real examples.

Furthermore, we analyzed and categorized the errors according to how these were fixed, as well as examined the relationship between the categories of taxonomy and bug-fixing types. Overall, the most common bug-fixes types are if-related(291), the second most common are assignment related (166), and the third most common are method call related (152) bug-fixes. These bug-fixes types are mostly related to the most prominent bug categories, namely missing input validation, incorrect input validation, and incorrect data processing. Another correlation is that assignment related fixes are also the preferred way to fix regexes.

Finally, in a possible use case, we demonstrated how the BUGSJS can be used by researchers through the examination of fault localization algorithms. We found that the results the examined three SBFL algorithms are similar to each other, but differences could also be observed that are aligned with previous studies for other languages and benchmarks. We came to the conclusion that there is a difference between the bugs in the bug-fixing groups based on how efficiently the algorithms find them. There are certain bug-fix types that are harder to localize (for instance, operation sequence change) and some others are easier (if condition related bugs). This information can help to design and develop more efficient algorithms in the future.

The contributions of the author to this thesis point

The author of the dissertation carried out the mapping and analysis of the existing bug data sets and he was actively involved in the design and implementation of the new benchmark and the new framework (including the validation of bugs and the construction of the software architecture). Both the creation of the taxonomy and the categorization of bugs (into the bug-fixing type groups) required the consensus decision of several researchers, in which the author also took an active role. Furthermore, he was the one who designed the experiment (and implemented the programs) through which it was demonstrated how the BUGSJS can be used to evaluate and compare the effectiveness of fault localization algorithms.

Publications

1. P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah: BugsJS: a Benchmark and Taxonomy of JavaScript Bugs. (*Software Testing, Verification and Reliability*, 31, 2021)
2. P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah: BugsJS: a Benchmark of JavaScript Bugs (*Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, IEEE, 90-101, 2019)
3. **B. Vancsics**, A. Szatmári and Á. Beszédes: Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs (*Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering*, IEEE, 308-319, 2020)
4. A. Szatmári, **B. Vancsics** and Á. Beszédes: Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? (*Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 16-23, 2020)
5. **B. Vancsics**, P. Gyimesi, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah: Supporting JavaScript Experimentation with BUGSJS (*Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, IEEE, 375-378, 2019)



Magyar nyelvű összefoglaló

Nincsen hiba nélküli program, legfeljebb még nem derült rá fény! Éppen ezért a szoftverfejlesztésnek egy roppant fontos és erőforrás-igényes része a tesztelés. Minél előbb sikerült "elkapnunk a hibát", annál kisebb ráfordítással javíthatjuk azt, ezáltal rengeteg időt és pénzt takaríthatunk meg.

Számos megközelítés és módszer létezik a hibák mielőbbi automatikus, félig automatikus vagy manuális detektálására. Dolgozatomban a unit tesztekre épülő hibadetektáló algoritmusok egyik legismertebb és legnépszerűbb családjával, a lefedettség/spektrum-alapú eljárásokkal foglalkozok, a dolgozat második felében pedig bemutatom az általunk megalkotott benchmark-t, amely segítségével az algoritmusok hatékonysága objektíven mérhetővé és ezáltal összehasonlíthatóvá válik.

1. rész: Hibalokalizációs algoritmusok

Ebben a tézispontban részletesen bemutatom a lefedettség-alapú hibakereső eljárások (úgynevezett *SBFL* algoritmusok) működési elvét, a legismertebb módszereket valamint az egyéb releváns szakirodalmakat.

Ezt követően bemutatok kettő, gráf-jellemzőkre épülő megközelítést, amelyek az SBFL-hez hasonló bemeneti adatok mellett próbálják minél precízebben meghatározni a hiba helyét. Ehhez egy úgynevezett lefedettségi-gráfot (coverage graph) épít, amelyből különböző szomszédsági információkat kinyerve azonosítja a leggyanúsabb elemeket.

Ezután bevezetésre került az *egyedi, leghosszabb hívássorozatok* fogalma (unique, deepest call stack - UDCS), amelyet kis példán keresztül szemléltetek. Részletesen leírásra kerül, hogy a UDCS-ből kinyerhető hívási gyakoriság-információk (call frequency) hogyan adaptálhatóak az SBFL által használt, korábban leírt formulákban, amelyek segítségével tovább növelhető a hibalokalizációs algoritmusok hatékonysága. Végezetül pedig bemutatom, hogy a fentebb leír két (gráf- és gyakoriság-alapú) koncepció hogyan ötvözhető, ezáltal kihasználva a két módszerben levő potenciális lehetőségeket.

A kvantitatív kiértékelést a *Defects4J* (Java) benchmark-on végeztem, amelyek azt mutatták, hogy 1) a két gráf-alapú módszer sok esetben hatékonyabb, mint a vizsgált SBFL módszerek, 2) a gyakorisági információkat felhasználó eljárások jelentősen

jobb eredményt érnek el, mint a bináris lefedettségére épülő módszerek és 3) a legjobb eredményt a kettő koncepció ötvöztetésével kapott, *WENFL* algoritmus eredményezte.

A gráf-alapú megközelítések vegyes eredményeket adott, vannak olyan SBFL eljárások amelyeknél ezek jobban teljesítettek (pl. átlagos rangok: *ENFL*– 34.05 vs. *Barinel*– 36.01) de volt ahol minimálisan ugyan, de rosszabb eredményt értek el (pl: *Ochiai*– 33.98). Ez -7% és +5% közötti javítást eredményezett. A gyakoriság felhasználásával elérhető javulás az átlagos rangban 5-101 közötti (8-ból 6 algoritmus esetében), amely 14-74%-os relatív hatékonyságnövekedést jelent. A legjobban a *Jaccard*, *Ochiai* és a *Sørensen-Dice* algoritmusok teljesítettek a Δ_{ef}^U konfigurációban. A legjobb eredményt a “hibrid” *WENFL* érte el a maga 20.59 átlagos rangjával, amely a SBFL-hez képest 39-84 százalékos javulást ért el. Hasonló eredményeket tapasztalhatóak akkor is, amikor a leggyanúsabb 5 elemet vizsgáljuk (Top-5). Az “klasszikus” SBFL módszerek közül az *Ochiai* érte el a legjobb eredményt (367, 46.7%), azonban mind a gráf-alapú, mind a gyakoriság-alapú, mind a hibrid módszerekkel elérhető ennél jobb eredmény (pl: *Ochiai*_{ef}^U: 380 – 48.3%, *NFL*: 369 – 46.9%, *WENFL*: 389 – 49.5%) Ezek alapján kijelenthető, hogy a gráf-jellemzők és a hívási gyakoriságok használatával jelentős javulás érhető el a hibakereső algoritmusok teljesítményében.

A szerző hozzájárulása

A disszertáció szerzője alkotta meg a gráf-alapú megközelítést használó, “hagyományos” lefedettség-mátrixokon működő algoritmust, valamint ő végezte el a módszer kvantitatív kiértékelését. Meghatározó szerepe volt a hívási láncokon alapuló megközelítés kidolgozásában, annak adaptálásában valamint az eredmények számszerűsítésében. A gráf- és a lánc alapú módszerek ötvöztetése szintén a szerző munkája, ahogy az eredmények összehasonlítása és kiértékelése is. Ezek mellett a disszertáció szerzője aktívan részt vett a szakirodalom feldolgozásában, a kísérletek valamint a mérések megtervezésében és a publikációk megírásában egyaránt.

Publikációk

1. **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes: Fault Localization Using Function Call Frequencies (*The Journal of Systems & Software*, 2022)
2. **B. Vancsics**: NFL: Neighbor-Based Fault Localization Technique (*Proceedings of the 1th International Workshop on Intelligent Bug Fixing*, IEEE, 17-22, 2019)
3. **B. Vancsics**: Graph-Based Fault Localization (*Proceedings of the International Conference on Computational Science and Its Applications*, Springer International Publishing, 372-387, 2019)
4. **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes: Call Frequency-Based Fault Localization (*Proceedings of the 28th International Workshop on Software Analysis, Evolution, and Reengineering*, IEEE, 365-376, 2021)

Hibalokalizációval kapcsolatos egyéb publikációk

1. Q. Idrees Sarhan, **B. Vancsics** and Á. Beszédes: Method Calls Frequency-Based Tie-Breaking Strategy For Software Fault Localization (*Proceedings of the 21th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 103-113, 2021)

2. **B. Vancsics**, A. Szatmári and Á. Beszédes: Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs (*Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering*, IEEE, 308-319, 2020)
3. A. Szatmári, **B. Vancsics** and Á. Beszédes: Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? (*Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 16-23, 2020)
4. **B. Vancsics**, T. Gergely and Á. Beszédes: Simulating the Effect of Test Flakiness on Fault Localization Effectiveness (*Proceedings of the 3rd Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 28-35, 2020.)
5. F. Horváth, G. Balogh, A. Szatmári, Q. Idrees Sarhan, **B. Vancsics** and Á. Beszédes: Interacting with interactive fault localization tools (*Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*, ACM, 61-63, 2022)
6. F. Horváth, Á. Beszédes, **B. Vancsics**, G. Balogh, L. Vidács and T. Gyimóthy: Using contextual knowledge in interactive fault localization (Empirical Software Engineering, volume: 27, number: 6, 1-69, 2022)
7. F. Horváth, Á. Beszédes, **B. Vancsics**, G. Balogh, L. Vidács and T. Gyimóthy: Experiments with Interactive Fault Localization Using Simulated and Real Users (*Proceedings of the 36th International Conference on Software Maintenance and Evolution*, IEEE, 290-300, 2020)

2. rész: BugsJS: JavaScript nyelven írt programok hiba-adatbázisa

A hibalokalizációs algoritmusok megbízható kiértékeléséhez elengedhetetlen a megfelelő minőségű hiba-adatbázis. Több, hasonló hibagyűjtemény létezik (például: *Defects4J* - Java), azonban ezidáig JavaScript nyelven írt programokra nem állt rendelkezésre ilyen adathalmaz. A disszertáció második felében felvázolja a szerző, hogy hogyan is történt ennek az új adathalmaznak (BUGSJS) a megalkotása, részletesen leírja, hogy milyen lépések mentén zajlott a projektek kiválasztása (10 projekt), a bennük levő hibák detektálása, validálása, "tisztítása" valamint különböző szempontok szerinti elemzése (453 hiba). A kutatás végére megalkotásra került egy olyan keretrendszer amely használatával könnyen elérhetővé és reprodukálhatóvá váltak az adathalmazban levő hibáknak az egyes (javítás előtti ill. a javítás utána) állapotai, ezáltal segítve a kutatókat, hogy JavaScript nyelvű programokat is tudjanak használni a különböző kutatásaikhoz.

Ezek mellett megalkottuk az adathalmazban levő hibáknak a rendszertanát (amely egy hierarchikus hiba-kategorizálást takar) és az így kapott taxonómiában létrehozott hiba-csoportokat részletesen, valós példákon keresztül bemutattuk. Továbbá elemeztük és kategorizáltuk az egyes hibákat aszerint is, hogy hogyan történt azok javítása, valamint megvizsgáltuk, hogy milyen összefüggés van a taxonómia és a hibajavítás kategóriái között. Összességében a leggyakoribb hibajavítási típusok az *if*-re vonatkozó változtatásokat tartalmazó kategóriához kapcsolódóak (291), a második leggyakoribb a hozzárendeléssel (166), a harmadik pedig a metódushívással kapcsolatos (152) hibajavítások. Ezek a típusok többnyire a legjelentősebb hibakategóriákhoz kapcsolódnak,

nevezetesen a hiányzó vagy helytelen bemeneti adat ellenőrzéshez és a helytelen adatfeldolgozáshoz.

Végezetül, pedig egy lehetséges felhasználási területen, a hibalokalizációs algoritmusok vizsgálatán keresztül bemutattuk, hogy a BUGSJS hogyan használható a kutatók számára. Megállapítottuk, hogy a vizsgált három SBFL algoritmus eredményei hasonlóak egymáshoz, de olyan eltérések is megfigyelhetők, amelyek összhangban vannak más programozási nyelvekre és adathalmazokra vonatkozó korábbi tanulmányokkal. Arra a következtetésre jutottunk, hogy egyes (a hiba javítása alapján létrehozott) csoportokban lévő hibák között különbség van az alapján, hogy az algoritmusok milyen hatékonyan találják meg őket. Vannak bizonyos (hibajavítási) típusok, amelyeket nehezebb lokalizálni (például a műveleti sorrend megváltoztatása), mások pedig könnyebbek (*if* elágazáshoz kapcsolódó hibák). Ezek az információk a jövőben segíthetnek hatékonyabb algoritmusok tervezésében és fejlesztésében.

A szerző hozzájárulása

A disszertáció szerzője végezte a már létező hiba-adatbázisok feltérképezést és elemzését, valamint aktívan részt vett az új data set megtervezésében és kivitelezésében (beleértve egyaránt a hibák validálását és a szoftveres architektúra kiépítését). Mind a taxonómia megalkotása, mind a hibajavítások kategorizálása több kutató konszenzusos döntését követelte, amiben a szerző is aktív szerepet vállalt. Továbbá ő volt az, aki megtervezte azt a kísérletet (és implementálta a szükséges programokat), amin keresztül bemutatásra került, hogy a BUGSJS hogyan használható hibalokalizációs algoritmusok hatékonyságának kiértékelésére és összehasonlítására.

Publikációk

1. P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc and A. Mesbah: BugsJS: a Benchmark and Taxonomy of JavaScript Bugs. (*Software Testing, Verification and Reliability*, 31, 2021)
2. P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc and A. Mesbah: BugsJS: a Benchmark of JavaScript Bugs (*Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, IEEE, 90-101, 2019)
3. **B. Vancsics**, A. Szatmári and Á. Beszédes: Relationship Between the Effectiveness of Spectrum-Based Fault Localization and Bug-fix Types in JavaScript Programs (*Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering*, IEEE, 308-319, 2020)
4. A. Szatmári, **B. Vancsics** and Á. Beszédes: Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? (*Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests*, IEEE, 16-23, 2020)
5. **B. Vancsics**, P. Gyimesi, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc and A. Mesbah: Supporting JavaScript Experimentation with BUGSJS (*Proceedings of the 12th International Conference on Software Testing, Verification and Validation*, IEEE, 375-378, 2019)

Acknowledgement

The dissertation emphasizes the work of the author and focuses on his results, but it could not have been created without the help of others. First of all, I would like to thank my supervisor, Dr. Árpád Beszédes, for the many years of joint work. I learned a lot from him during the year. I would not have gotten this far without his help, advice, and constant motivation.

I also owe a debt of gratitude to Tamás Gergely, Ferenc Horváth, Gergő Balogh, and Attila Szatmári, with whom I have worked for years and am still working. They are not only excellent colleagues but also excellent researchers who have continuously supported me during the research of the past years. I would like to thank Edit Szűcs for reviewing and correcting my thesis from a linguistic point of view.

And, of course, I would like to thank my family for supporting and trusting me all the way and for creating the conditions for me to focus only on my research. I cannot thank them enough for that!

The research was supported by the 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.

In addition, 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.

Bibliography

- [1] Rui Abreu, Alberto Gonzalez-Sanchez, and Arjan JC van Gemund. Exploiting count spectra for bayesian fault localization. *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–10, New York, NY, USA, 2010. ACM, Association for Computing Machinery.
- [2] Rui Abreu, Peter Zoetewij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, ASE '09*, pages 88–99. IEEE Computer Society, 2009.
- [3] Rui Abreu, Peter Zoetewij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.
- [4] Rui Abreu, Peter Zoetewij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. *Proceedings of the 2007 Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, IEEE Press, 2007.
- [5] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling non-determinism. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.
- [6] Pragya Agarwal and Arun Prakash Agrawal. Fault-localization techniques for software systems: A literature review. *SIGSOFT Softw. Eng. Notes*, 39(5):1–8.
- [7] Alan Agresti et al. A survey of exact inference for contingency tables. *Statistical science*, 7(1):131–153, 1992.
- [8] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid DOM-sensitive change impact analysis for JavaScript. In *Proc. of European Conference on Object-Oriented Programming (ECOOP)*, 2015.
- [9] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack JavaScript. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.
- [10] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 428–452, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

- [11] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Comput. Surv.*, 50(5):66:1–66:36, September 2017.
- [12] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of International Conference on Software Engineering*, 2005.
- [13] S. Artzi, J. Dolby, S. H. Jensen, A. Moller, and F. Tip. A framework for automated testing of JavaScript web applications. In *33rd International Conference on Software Engineering (ICSE)*, 2011.
- [14] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, ACM, 2016.
- [15] Árpád Beszédes, Ferenc Horváth, Massimiliano Di Penta, and Tibor Gyimóthy. Leveraging contextual information from function call chains to improve fault localization. *Proceedings of 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–479. IEEE, IEEE Press, 2020.
- [16] Marina Billes, Anders Møller, and Michael Pradel. Systematic black-box analysis of collaborative web applications. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.
- [17] E. C. Campos and M. d. A. Maia. Common bug-fix patterns: A large-scale observational study. In *Proc. of ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2017.
- [18] Heling Cao, Shujuan Jiang, Xiaolin Ju, Zhang Yanmei, and Guan Yuan. Applying association analysis to dynamic slicing based fault localization. *IEICE Transactions on Information and Systems*, E97.D:2057–2066, 08 2014.
- [19] J. S. Collofello and L. Cousins. Towards automatic software fault location through decision-to-decision path analysis. *Proceedings of the 1987 International Workshop on Managing Requirements Knowledge (AFIPS)*, page 539. IEEE, 12 1987.
- [20] William Jay Conover. *Practical nonparametric statistics*, volume 350. John Wiley & Sons, 1998.
- [21] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *European conference on object-oriented programming*, pages 528–550. Springer, 2005.
- [22] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of International Conference on Automated Software Engineering*.
- [23] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *Proc. of 12nd European Conference on Computer Systems (EuroSys)*, 2017.

-
- [24] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *ArXiv*, abs/1607.04347, 2016.
 - [25] Vidroha Debroy, W Eric Wong, Xiaofeng Xu, and Byoungju Choi. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 13–22. IEEE, 2010.
 - [26] Monika Dhok, Murali Krishna Ramanathan, and Nishant Sinha. Type-aware concolic testing of JavaScript programs. In *Proc. of 38th International Conference on Software Engineering*, 2016.
 - [27] Naji Dmeiri, David A. Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar Devanbu, Bogdan Vasilescu, and Cindy Rubio-Gonzalez. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *Proc. of 41st International Conference on Software Engineering (ICSE)*, 2019.
 - [28] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.
 - [29] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42:1–1, 08 2016.
 - [30] W. Eric Wong and Yu Qi. Bp neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 19, 11 2011.
 - [31] Markus Ermuth and Michael Pradel. Monkey see, monkey do: Effective generation of GUI tests with inferred macro events. In *Proc. of 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
 - [32] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE transactions on software engineering*, 27(2):99–123, 2001.
 - [33] Laleh Eshkevari, Davood Mazinanian, Shahriar Rostami, and Nikolaos Tsantalis. JSDeodorant: Class-awareness for JavaScript Programs. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 2017.
 - [34] A. M. Fard, A. Mesbah, and E. Wohlstadter. Generating fixtures for JavaScript unit testing. In *Proc. of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.
 - [35] Amin Milani Fard and Ali Mesbah. JavaScript: The (un)covered parts. In *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.
 - [36] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *Proc. of 34th International Conference on Software Engineering (ICSE)*, 2012.

- [37] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in JavaScript. In *Proc. 39th International Conference on Software Engineering*, 2017.
- [38] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. VulinOSS: A dataset of security vulnerabilities in open-source systems. In *Proc. of 15th International Conference on Mining Software Repositories*, 2018.
- [39] Liang Gong, Hongyu Zhang, Hyunmin Seo, and Sunghun Kim. Locating crashing faults based on crash stack traces. *CoRR*, abs/1404.4100, 04 2014.
- [40] R. Gopinath, C. Jensen, and A. Groce. Mutations: How close are they to real faults? In *Proc. of International Symposium on Software Reliability Engineering*, 2014.
- [41] Robert J Grissom and John J Kim. *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers, 2005.
- [42] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinianian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. BugJS: A benchmark of javascript bugs. In *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation, ICST 2019*, page 12 pages. IEEE, 2019.
- [43] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinianian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Bugsjs: a benchmark of javascript bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 90–101, 2019.
- [44] Mouna Hammoudi, Gregg Rothermel, and Andrea Stocco. WATERFALL: An incremental approach for repairing record-replay tests of web applications. In *Proc. of 24th International Symposium on Foundations of Software Engineering (FSE)*, 2016.
- [45] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in JavaScript. In *Proc. of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.
- [46] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
- [47] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, page 83–90, New York, NY, USA, 1998. ACM, Association for Computing Machinery.
- [48] Hongdou He, Jiadong Ren, Guyu Zhao, and Haitao He. Enhancing spectrum-based fault localization using fault influence propagation. *IEEE Access*, 8:18497–18513, 2020.

-
- [49] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience*, 49(8):1197–1224, 2019.
 - [50] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side JavaScript web applications. In *Proc. of IEEE 7th International Conference on Software Testing, Verification and Validation*, 2014.
 - [51] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proc. of 16th International Conference on Software Engineering*, 1994.
 - [52] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5), 2011.
 - [53] Shujuan Jiang, Wei Li, Haiyang Li, Zhang Yanmei, Hongchang Zhang, and Yingqi Liu. Fault localization for null pointer exception based on stack trace and program slicing. Proceedings of the 2012 12th International Conference on Quality Software, pages 9–12, USA, 08 2012. IEEE, IEEE Computer Society.
 - [54] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 273–282, New York, NY, USA, 2005. ACM.
 - [55] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014*, pages 437–440. ACM, 2014.
 - [56] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.
 - [57] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pages 114–125. IEEE, IEEE Press, 2017.
 - [58] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners’ expectations on automated fault localization. Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 165–176. ACM, ACM, 2016.
 - [59] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. Localising faults in test execution traces. Proceedings of the 14th International Workshop on Principles of Software Evolution, pages 1–8. ACM, ACM, 08 2015.
 - [60] T. B. Le, F. Thung, and D. Lo. Theory and practice, do they match? a case with spectrum-based fault localization. Proceedings of the 2013 IEEE International Conference on Software Maintenance, pages 380–383. IEEE, IEEE Press, 2013.

- [61] Tien-Duy B Le, David Lo, and Ferdian Thung. Should i follow this fault localization tool's output? *Empirical Software Engineering*, 20(5):1237–1274, 2015.
- [62] Claire Le Goues, Neal Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015.
- [63] H. J. Lee, L. Naish, and K. Ramamohanarao. Effective software bug localization using spectral frequency weighting function. Proceedings of the 34th Annual Computer Software and Applications Conference, pages 218–227. IEEE, IEEE Press, 2010.
- [64] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 169–180. ACM, ACM, 2019.
- [65] Yihan Li and Chao Liu. Effective fault localization using weighted test cases. *J. Softw.*, 9(8):2112–2119, 2014.
- [66] Zheng Li, Yonghao Wu, Haifeng Wang, and Yong Liu. Test oracle prediction for mutation based fault localization. In Zheng Li, He Jiang, Ge Li, Minghui Zhou, and Ming Li, editors, *Software Engineering and Methodology for Emerging Domains*, pages 15–34. Springer Singapore, Singapore, 2019.
- [67] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, June 2005.
- [68] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proc. of International Conference on Systems, Programming, Languages, and Applications: Software for Humanity: Companion*.
- [69] Chao Liu, Xifeng Yan, Hwanjo Yu, Jiawei Han, and Philip S Yu. Mining behavior graphs for “backtrace” of noncrashing bugs. In *Proceedings of the 2005 SIAM international conference on data mining*, pages 286–297. SIAM, 2005.
- [70] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bug-bench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [71] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of software: Evolution and Process*, 26(2):172–219, 2014.
- [72] Magnus Madsen, Frank Tip, Esben Andreasen, Koushik Sen, and Anders Møller. Feedback-directed instrumentation for deployed JavaScript applications. In *Proc. of 38th International Conference on Software Engineering (ICSE)*, 2016.
- [73] Amirabbas Majd, Mojtaba Vahidi-Asl, Alireza Khalilian, Ahmad Baraani-Dastjerdi, and Bahman Zamani. Code4bench: A multidimensional benchmark of

- codeforces data for different program analysis techniques. *Journal of Computer Languages*, 53:38–52, 2019.
- [74] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: Inferring javascript function types from natural language information. In *Proceedings of the 41st International Conference on Software Engineering, ICSE '19*, pages 304–315, Piscataway, NJ, USA, 2019. IEEE Press.
 - [75] Xiaoguang Mao, Yan Lei, Ziyang Dai, Yuhua Qi, and Chengsong Wang. Slice-based statistical fault localization. *Journal of Systems and Software*, 89:51–62, 2014.
 - [76] A. Mesbah, A. van Deursen, and D. Roest. Invariant-based automatic testing of modern web applications. *IEEE Transactions on Software Engineering*, 2012.
 - [77] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Efficient JavaScript mutation testing. In *Proc. of 6th International Conference on Software Testing, Verification and Validation (ICST)*, 2013.
 - [78] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Guided mutation testing for JavaScript web applications. *IEEE Transactions on Software Engineering*, 41(5):429–444, May 2015.
 - [79] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. JSEFT: Automated JavaScript unit test generation. In *Proc. of 8th International Conference on Software Testing, Verification and Validation (ICST)*, 2015.
 - [80] S. Mirshokraie, A. Mesbah, and K. Pattabiraman. Atrina: Inferring unit oracles from GUI test cases. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2016.
 - [81] Shabnam Mirshokraie and Ali Mesbah. JSART: JavaScript assertion-based regression testing. In *Web Engineering (ICWE)*, pages 238–252, 2012.
 - [82] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. Do automated program repair techniques repair hard and important bugs? *Empirical Softw. Engg.*, 23(5):2901–2947, October 2018.
 - [83] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectral-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.
 - [84] N Neelofar, Lee Naish, and Kotagiri Ramamohanarao. Spectral-based fault localization using hyperbolic function. *Software: Practice and Experience*, 48(3):641–664, 2018.
 - [85] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah. A Study of Causes and Consequences of Client-Side JavaScript Bugs. *IEEE Transactions on Software Engineering*, 43(2):128–144, Feb 2017.
 - [86] Frolin S. Ocariza, Guanpeng Li, Karthik Pattabiraman, and Ali Mesbah. Automatic fault localization for client-side JavaScript. *Softw. Test. Verif. Reliab.*, 26(1), January 2016.

- [87] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Vejovis: Suggesting fixes for JavaScript faults. In *Proc. of 36th International Conference on Software Engineering*.
- [88] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Detecting inconsistencies in JavaScript MVC applications. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, 2015.
- [89] Frolin S Ocariza Jr, Karthik Pattabiraman, and Ali Mesbah. Detecting unknown inconsistencies in web applications. In *Proc. of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.
- [90] Cherry Oo and Hnin Min Oo. Spectrum-based bug localization of real-world java bugs. In Roger Lee, editor, *Software Engineering Research, Management and Applications*, pages 75–89. Springer International Publishing, Cham, 2020.
- [91] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, Jun 2009.
- [92] Mike Papadakis and Yves Le Traon. Effective fault localization via mutation analysis: A selective mutation approach. Proceedings of the 29th Annual ACM Symposium on Applied Computing, page 1293–1300, New York, NY, USA, 2014. ACM, Association for Computing Machinery.
- [93] Mike Papadakis and Yves Le Traon. Metallaxis-fl: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5–7):605–628, August 2015.
- [94] Priya Parmar and Miral Patel. Software fault localization: A survey. *International Journal of Computer Applications*, 154(9), 2016.
- [95] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? Proceedings of the 2011 International Symposium on Software Testing and Analysis, pages 199–209. ACM, ACM, 2011.
- [96] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.
- [97] A. Perez, R. Abreu, and M. D’Amorim. Prevalence of single-fault fixes and its impact on fault localization. In *Proc. of IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.
- [98] A. Perez, R. Abreu, and A. van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proc. of 39th International Conference on Software Engineering (ICSE)*, 2017.
- [99] M. Pradel, P. Schuh, and K. Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 314–324, May 2015.

-
- [100] Christoffer Quist, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In *Proc. of International Symposium on Software Testing and Analysis*.
 - [101] Md. Rezaur Rahman, Mojdeh Golagha, and Alexander Pretschner. Pairika: A failure diagnosis benchmark for C++ programs. In *Proc. of 40th International Conference on Software Engineering: Companion*, 2018.
 - [102] Xiaoxia Ren and Barbara G. Ryder. Heuristic ranking of java program edits for fault localization. In *ISSTA '07*, 2007.
 - [103] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM SIGSOFT Software Engineering Notes*, 22(6):432–449, November 1997.
 - [104] S. Rostami, L. Eshkevari, D. Mazinianian, and N. Tsantalis. Detecting function constructors in JavaScript. In *Proc. of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.
 - [105] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.Jar: A large-scale, diverse dataset of real-world Java bugs. In *Proc. of 15th International Conference on Mining Software Repositories (MSR)*, 2018.
 - [106] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 139–152, 2013.
 - [107] R. Santelices, J. A. Jones, Yanbing Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 56–66. IEEE, IEEE Press, 2009.
 - [108] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? *Proceedings of the 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 118–121. IEEE, 05 2010.
 - [109] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Trans. Softw. Eng.*, 25(4):557–572, July 1999.
 - [110] Shin Hwei Tan, Jooyong Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182, 2017.
 - [111] G. Shu, B. Sun, A. Podgurski, and F. Cao. Mfl: Method-level fault localization with causal inference. *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 124–133. IEEE, IEEE Press, 2013.
 - [112] Ting Shu, Tiantian Ye, Zuohua Ding, and Jinsong Xia. Fault localization based on statement frequency. *Information Sciences*, 360:43 – 56, 2016.

- [113] Leonardo Humberto Silva, Marco Tulio Valente, and Alexandre Bergel. Refactoring legacy JavaScript code to use classes: The good, the bad and the ugly. In *Mastering Scale and Complexity in Software Reuse*, 2017.
- [114] Takao Simomura. Critical slice-based fault localization for any type of error. *IEICE Transactions on Information and Systems*, 76:656–667, 1993.
- [115] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *Proceedings of SANER*, 2018.
- [116] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, page 314–324, New York, NY, USA, 2013. ACM, Association for Computing Machinery.
- [117] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proc. of 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.
- [118] Attila Szatmári, Béla Vancsics, and Árpád Beszédes. Do bug-fix types affect spectrum-based fault localization algorithms’ efficiency? In *2020 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 16–23, 2020.
- [119] Peter Thiemann. Towards a type system for analyzing javascript programs. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 408–422, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.
- [120] B. Vancsics. NFL: Neighbor-based fault localization technique. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pages 17–22, Feb 2019.
- [121] Béla Vancsics. Graph-based fault localization. In *Computational Science and Its Applications – ICCSA 2019*, pages 372–387, Cham, 2019. Springer International Publishing.
- [122] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Call frequency-based fault localization. *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER’21)*, pages 365–376. IEEE, IEEE Press, March 2021.
- [123] Béla Vancsics, Péter Gyimesi, Andrea Stocco, Davood Mazinianian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. Poster: Supporting javascript experimentation with bugsjs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 375–378, 2019.
- [124] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. Fault localization using function call frequencies. *Journal of Systems and Software*, 193:111429, 2022.

- [125] Béla Vancsics, Attila Szatmári, and Árpád Beszédes. Relationship between the effectiveness of spectrum-based fault localization and bug-fix types in javascript programs. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 308–319, 2020.
- [126] J. Wang, W. Dou, C. Gao, Y. Gao, and J. Wei. Context-based event trace reduction in client-side JavaScript applications. In *Proc. of International Conference on Software Testing, Verification and Validation (ICST)*, 2018.
- [127] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei. A comprehensive study on real world concurrency bugs in Node.js. In *Proc. of International Conference on Automated Software Engineering*, 2017.
- [128] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
- [129] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proc. of EASE '14*, pages 1–10, 2014.
- [130] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2014.
- [131] W Eric Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45*, 9, 2009.
- [132] W Eric Wong and Yu Qi. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891–903, 2006.
- [133] W Eric Wong, Tatiana Sugeta, Yu Qi, and Jose C Maldonado. Smart debugging software architectural design in sdl. *Journal of Systems and Software*, 76(1):15–28, 2005.
- [134] Xin Xia, Lingfeng Bao, David Lo, and Shanping 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. *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, pages 267–278. IEEE, IEEE Press, 2016.
- [135] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.*, 22(4):31:1–31:40, October 2013.
- [136] Xiaofeng Xu, Vidroha Debroy, W Eric Wong, and Donghui Guo. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering*, 21(06):803–827, 2011.
- [137] Jifeng Xuan and Martin Monperrus. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 191–200. IEEE, 2014.

- [138] Cemal Yilmaz, Amit Paradkar, and Clay Williams. Time will tell. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 81–90. IEEE, 2008.
- [139] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. *Proceedings of the 2012 International Symposium on Search Based Software Engineering*, pages 244–258. Springer, Springer, 2012.
- [140] Abubakar Zakari, Sai Peck Lee, and Ibrahim Abaker Targio Hashem. A single fault localization technique based on failed test input. *Array*, 3:100008, 2019.
- [141] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 261–272. ACM, ACM, 2017.
- [142] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging, AADEBUG 2005*, pages 33–42, New York, NY, USA, 09 2005. ACM, ACM.
- [143] Guyu Zhao, Hongdou He, and Yifang Huang. Fault centrality: boosting spectrum-based fault localization via local influence calculation. *Applied Intelligence*, pages 1–23, 2021.
- [144] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proc. of 37th International Conference on Software Engineering (ICSE)*, pages 913–923, 2015.
- [145] Hui Zhu, Tu Peng, Ling Xiong, and Daiyuan Peng. Fault localization using function call sequences. *Procedia Computer Science*, 107:871–877, 12 2017.
- [146] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 2019.