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

Summary of the Ph.D. Thesis

University of Szeged
Ph.D. School in Computer Science

Introduction

There is no software without bugs! Almost every week there are articles about bugs and vulnerabilities found in the most popular programs and operating systems. There are those that "only" cause inconvenience, but there have also been cases that involved very serious financial losses (for example, the destruction of the Ariane 5 rocket[1]). And that is just the tip of the iceberg!

These cases all highlight how important thorough and reliable testing is, even though it is a resource-intensive part of the development process (and the software life cycle). The sooner we detect the bugs, the less effort we can repair them, thereby saving a lot of time and resources.

There are many approaches and methods for the early automatic, semi-automatic, or manual detection of defects and for measuring the effectiveness of the testing, thus supporting thorough and continuous inspection. My research also proceeded along these two main directions: I presented more algorithms based on (unit)test results and their (source code) coverage obtained by automatic, dynamic execution, which helps to automatically determine the location of the bug in the software, and we produced a benchmark containing JavaScript programs, which can be used for example to measure the effectiveness of different fault localization algorithms. With the help of these, we can get a reliable picture of the performance of the algorithms, and these can be the starting point for further research, which can be used to further improve the performance of fault localization methods.

I Fault Localization Algorithms

One of the most popular and most researched algorithm families are the so-called *spectrum-based fault localization* algorithms (SBFL), whose essence is to assign a suspiciousness value to each code element based on the code coverage and the test results. Based on these values, it sets up a ranking list, which determines which code elements are potentially the most suspicious, that is, which code elements may contain bugs based on the concept. The code element can be a statement, branch, or method depending on the granularity, but most of the research seeks to determine the faulty method, that is, it conducts method-level debugging (and I used method-based resolution in my research).

The SBFL algorithms store the coverage data in a binary matrix *(coverage matrix)* and the test results in a binary vector *(result vector)*. If an element of the matrix is equal to 1, it means that the given code element (in this case, the method) was called during the execution of the given test, and if the given element of the result vector is 1, then the given test failed (if its value is 0 then it is successful, i.e. it passed)
A four-number (spectrum) can be extracted from these two data structures:

- m_{ef}: set of failed tests covered by m method
- m_{ep}: set of passed tests covered by m method
- m_{nf}: set of failed tests not covered by m method
- m_{np}: set of passed tests not covered by m method.

These are the basis of the SBFL formulae and on the basis of which the suspiciousness values (and ranks) are calculated. These are effective methods in many cases, but they do not take into account other relationships between the tests and the code element, which can be used to further increase the efficiency of the fault localization method.

[1] `https://homepages.inf.ed.ac.uk/perdita/Book/ariane5rep.html`

One such possible "extra information" is the frequency of calls. This topic has already been researched [7, 1], but they did not achieve the expected results, however, our approach changes that.
The basic idea was to use the number of occurrences in the dynamic call stacks generated during the execution of the test in the coverage matrix, that is, we store not only whether or not a method was called during the execution (of the test), but also how many times (it was called). Due to the fact that our matrix does not only store 0 and 1 values, we also had to redefine/adapt the $|m_{ef}|$, $|m_{ep}|$, $|m_{nf}|$, and $|m_{np}|$. In this new analogy, the $|m_{ef}|$ value of the given method will be the sum of the matrix elements for which the test result failed, and $|m_{ep}|$ will be the sum of the elements where the test was passed. The adaptation of $|m_{nf}|$ and $|m_{np}|$ is somewhat more complicated. In this case, we used the average coverage of the non-covered elements, i.e. we calculated the average coverage of the methods, that are not covered by the test (separating failing and non-failing tests) The new values thus obtained can be "substituted" into the SBFL formulas.

We can see through the example (Table 1) how the spectrum values are calculated for binary and frequency-based matrices. The cells marked in red determine the value of $|m_{ef}|$, those written in green indicate $|m_{ep}|$, orange indicates $|m_{nf}|$ and blue indicates the $|m_{np}|$ value. We can see that in the case of the binary matrix, these values depend only on the vector belonging to the given method (and the test results), whereas in the case of the frequency-based approach, the "full matrix" affects the values of the four variables.

Table 1: Example of a binary (hit) and frequency (count) matrix and the program spectra

	binary matrix				frequency matrix				test
	a	b	**c**	d	a	b	**c**	d	results
t1	1	0	1	1	2	0	2	1	0
t2	0	1	1	1	0	3	1	2	0
t3	1	1	0	0	2	2	0	0	0
t4	1	0	1	0	2	0	2	0	1
t5	1	1	0	1	3	2	0	1	1
t6	0	1	1	0	0	1	2	0	1
spectra of	ef	ep	nf	np	ef	ep	nf	np	
c method	2	2	1	1	4	3	3	4	

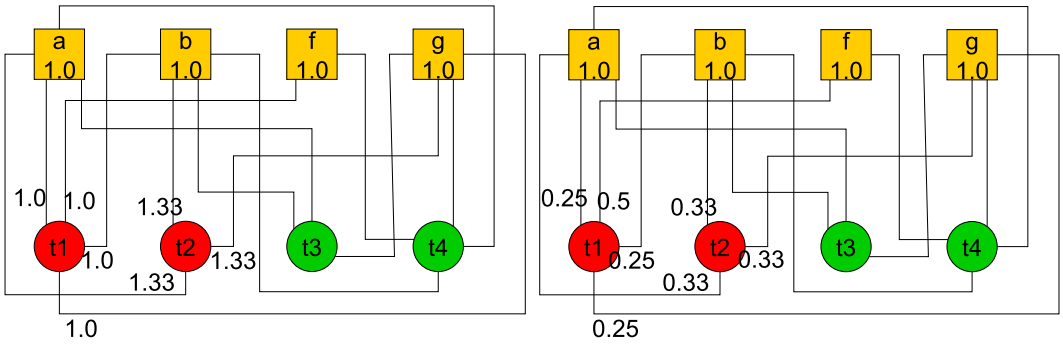
The SBFL formulas use this spectrum to determine the suspiciousness value of each method. For example, this is the value for c for the Ochiai [4] algorithm: $\frac{|m_{ef}|}{\sqrt{(|m_{ef}|+|m_{nf}|)\cdot(|m_{ef}|+|m_{ep}|)}} \rightarrow \frac{2}{\sqrt{(2+1)\cdot(2+2)}} = 0.577$, and $\frac{4}{\sqrt{(4+3)\cdot(4+3)}} = 0.571$, depending on which type of matrix we are talking about.

We have the option to replace only certain components of the formula, that is, some elements of the spectrum are calculated based on the binary, and the other elements are calculated based on the frequency matrix. Based on this, I distinguished four scenarios: (a) $\Delta^U_{ef^{num}}$: I use the frequency-based $|m_{ef}|$ value only in the nominator instead of the binary-based $|m_{ef}|$, (b) Δ^U_{ef} for all $|m_{ef}|$ values are "replaced" in the formula, (c) Δ^U_e: I replace the $|m_{ef}|$ and $|m_{ep}|$ values and (d) Δ^U_{all}: I use only the frequency-based spectra (where Δ denotes an arbitrary SBFL formula). I compared the above four approaches with the results of the "original" SBFL formulas.

It is also possible to extract additional information from the structure of the matrix. I created graph-based concepts that use the coverage graph formed from the coverage matrix (Figure 1), where there is an edge between a test and a method if the method is covered by the test.

With the help of features extracted from this graph, two new algorithms (*NFL* and *ENFL*) were created, which determine the suspiciousness values in 4 and 4+3 steps, respectively. The following four steps are the basis of the *NFL* algorithm (Fig. 1):

- the determination of the weight of edges related to faulty tests (Fig. 1(a)): this is nothing but the quotient of the number of methods and the number of methods covered by the failed test. The larger this value is, the more "test-oriented" the given method is (since only a small part of the methods is affected by the test)
- proportioning of edge weights (Fig. 1(b)): the (edge weight) values obtained in the first step are divided by the number of tests covered by the given method, thereby expressing that methods covered by fewer tests are more suspicious than those affected by more tests.
- aggregation (Fig. 1(c)): sums the weights on the edges and aggregates them into the "method-nodes"
- final proportionality (Fig. 1(d)): the value obtained in the previous step is multiplied by the quotient of the number of failed tests covered by the method and the number of edges coming out of the failed tests, thereby rewarding if a method is "responsible" for the failure for the majority of calls related to tests.



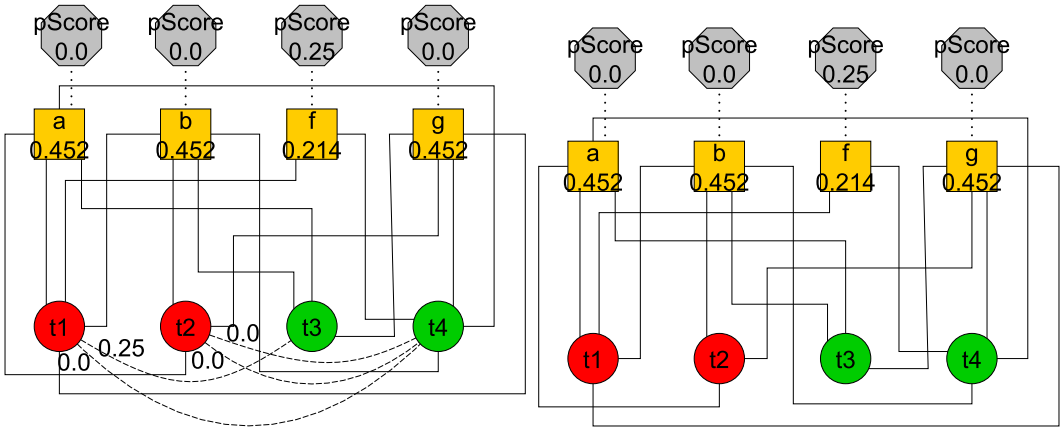
(a) 1. step (b) 2. step

(c) 3. step (d) 4. step

Figure 1: The steps of the *NFL* algorithm

The *NFL* algorithm can be extended with three additional steps (Fig. 2), which try to determine the location of the bug as precisely as possible using the coverage of the failed-passed test pairs.

- comparison of test pairs (Fig. 2(a)): all test pairs were created where one of the tests passed and the other failed, then the quotient was calculated for each pair, which is obtained by dividing the number of methods covered only by failed test (i.e. not covered by the passed pair) by the number of methods covered by the failed test. These quotients are aggregated for the methods that were only covered by the failed test (and not by the passed pair)
- averaging (Fig. 2(b)): the value obtained in the previous step is averaged, i.e. divided by the number of failed tests covered by the method
- combining (Fig. 2(c)): the values given by *NFL* and calculated after the two steps above are multiplied together, that is, the results obtained by examining the relationships between test-method and between test-test are combined.



(a) 5. step

(b) 6. step

(c) 7. step

Figure 2: The steps of the *ENFL* algorithm

One of the advantages of the *NFL* and *ENFL* methods is that they can be interpreted not only on an unweighted graph but also on a weighted one, that is, we have the possibility to combine graph-based and frequency-based methods, where the weight of the edges will be the call frequency value.

In order to get the objective picture of the efficiency of algorithms, three things are necessary: "reference algorithms" to which we compare the performance of new methods, a reliable data

set (benchmark) on which to evaluate the performance, and evaluation metrics that quantify the efficiency.

Due to reliable assessment, I selected eight SBFL formulae (Table 2), which are regularly used as state-of-the-art in research on a similar topic.

Table 2: Details of the hit-based SBFL formulae used in the experiment

Barinel (B) [2]: $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{ep}\rvert}$	*DStar* (D) [10]: $\frac{\lvert m_{ef}\rvert^2}{\lvert m_{ep}\rvert + \lvert m_{nf}\rvert}$
GP13 (G) [12]: $\lvert m_{ef}\rvert \cdot \left(1 + \frac{1}{2 \cdot \lvert m_{ep}\rvert + \lvert m_{ef}\rvert}\right)$	*Jaccard* (J) [4]: $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$
Ochiai (O) [4] : $\frac{\lvert m_{ef}\rvert}{\sqrt{(\lvert m_{ef}\rvert + \lvert m_{nf}\rvert) \cdot (\lvert m_{ef}\rvert + \lvert m_{ep}\rvert)}}$	*Russell-Rao* (R) [3]: $\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert + \lvert m_{np}\rvert}$
Sørensen-Dice (S) [8] : $\frac{2 \cdot \lvert m_{ef}\rvert}{2 \cdot \lvert m_{ef}\rvert + \lvert m_{nf}\rvert + \lvert m_{ep}\rvert}$	*Tarantula* (T) [5] : $\frac{\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert}}{\frac{\lvert m_{ef}\rvert}{\lvert m_{ef}\rvert + \lvert m_{nf}\rvert} + \frac{\lvert m_{ep}\rvert}{\lvert m_{ep}\rvert + \lvert m_{np}\rvert}}$

For analysis and evaluation, I selected Defects4J [6], a widely used collection of Java programs and curated bugs in FL research. This benchmark contains seventeen open-source Java projects with manually validated, non-trivial real bugs. The original dataset contains 835 bugs, however, there were cases that I had to exclude from the study due to instrumentation errors or unreliable test results. A total of 786 defects were included in the final dataset.

For the comparability of the suspiciousness values, I used the *ranks*, which indicate how many positions the examined element is in the suspiciousness ranking list. The rank is defined as follows:

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

where s_i and s_f are the suspicion values for the not faulty (i) and faulty methods (f). If several methods have the same value, the average of these ranks will be assigned to each such method. Note that, if there are multiple bugs for a program version, I will use the highest rank of buggy methods. In other words, the rank of the bug shows how many methods on average the developer needs to examine to find the location of the bug.

The possible aspects of the comparison, on the basis of which I examined the results, is how many cases resulted in a higher (better) or lower (worse) rank of the new algorithm than the SBFL methods (Fig. 3), what the average ranks of the bugs were(Fig. 4) and how great the difference between ranks was.

Several studies(e.g. [11]) showed that developers only examine the first 5 or 10 most suspicious elements during bug fixing, the rest of the ranking elements are not relevant to them. Therefore, I examined how many errors are among the 5 most suspicious elements for each algorithm (*Top-5*)

We can see that in most cases, the new methods a give better rank than the “classic” SBFL formulae, and it can be read that for all SBFL algorithms *WENFL* “won” it most of the time (that is, in the row of Table 3, *WENFL* achieves the highest win-value).

Table 3: Basic statistics of formulae. Losses are in favor of the hit-based formulae

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

Table 4 shows the average of the ranks achieved on 786 bigs. We can see that the best result, i.e. the highest average rank was achieved by the *WENFL* method (20.59), producing much better results than the reference SBFL algorithms (33.98-135.96). Furthermore, for all formulas except for *Tarantula* and *GP13*, the frequency-based approach performed better (at least in one concept) than the binary one, that is, it was (on average) more efficient at finding the location of the bug by taking the call frequency into account. For example $B^U_{ef^{num}}$ and B^U_{ef} both performed better (24.68 and 24.55) than the binary approach (36.01), but B^U_e and B^U_{all} were worse (38.63).

Table 4: Average ranks (E)

	hit	*NFL*	*ENFL*	$\Delta^U_{ef^{num}}$	Δ^U_{ef}	Δ^U_e	Δ^U_{all}	*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		

Table 5 and Table 6 show the number of bugs with a rank less than or equal to 5 for each algorithm (# column), what part of the total data set this is (% column), how many bugs there were that, according to the reference SBFL algorithm, belong to the Top-5 category, but based on the new algorithms, their ranked lower (*Det.* column), and the number of cases, where the rank was reduced by the new method and thus got positioned in the Top-5 category (*E.im.* column). In this case too, we see similar results as in the case of the average rank. It can also be established that the number of bugs in the Top-5 category can be increased by using the frequency-based spectrum. The *WENFL* method achieves better Top-5 values (389 – 49.5%) than methods using the binary approach and it also produces high $E.Im.$ values (97-282), that is, there were more cases where the reference formulae gave a rank lower than 5, but the new algorithm reduced this

to Top-5. It is also interesting that both *NFL* and *ENFL* achieve larger # values than any SBFL formula, all with very low *Det.* values (that is, there are few cases where the bug is "removed" from the Top-5 category as a result of the new algorithm) and in all cases (similar to *WENFL*) it is included in the Top-5 more times than it is removed from it (i.e. $Det - E.Im. < 0$).

Table 5: Number of bugs in Top-5 category (using frequency-based approaches)

	hit		Δ^U_{efnum}				Δ^U_{ef}				Δ^U_e				Δ^U_{all}			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)	373	(47.5%)	78	94	376	(47.8%)	65	84	354	(45.0%)	45	42	354	(45.0%)	45	42
D	366	(46.6%)	327	(41.6%)	128	89	327	(41.6%)	128	89	391	(49.7%)	86	111	388	(49.4%)	88	110
G	361	(45.9%)	-	-	-	-	266	(33.8%)	172	77	269	(34.2%)	170	78	269	(34.2%)	170	78
J	358	(45.5%)	372	(47.3%)	81	95	380	(48.3%)	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
O	367	(46.7%)	323	(41.1%)	135	91	380	(48.3%)	74	87	363	(46.2%)	51	47	361	(45.9%)	54	48
R	111	(14.1%)	249	(31.7%)	12	150	248	(31.6%)	12	149	380	(48.3%)	6	275	356	(45.3%)	10	255
S	358	(45.5%)	366	(46.6%)	87	95	380	(48.3%)	69	91	357	(45.4%)	47	46	356	(45.3%)	48	46
T	357	(45.4%)	-	-	-	-	351	(44.7%)	12	6	258	(32.8%)	111	12	254	(32.3%)	117	14

Table 6: Number of bugs in Top-5 category (using graph-based approaches)

	hit		***NFL***				***ENFL***				***WNFL***				***WENFL***			
	#	%	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.	#	%	Det.	E. Im.
B	357	(45.4%)			14	26			15	26			127	96			69	101
D	366	(46.6)%			12	15			13	15			133	93			74	97
G	361	(45.9)%			16	24			19	26			132	97			76	104
J	358	(45.5)%	369	(46.9%)	10	21	368	(46.8%)	11	21	326	(41.5%)	128	96	389	(49.5%)	70	101
O	367	(46.7)%			11	13			12	13			134	93			75	97
R	111	(14.1)%			2	260			2	259			6	221			4	282
S	358	(45.5)%			10	21			11	21			128	96			70	101
T	357	(45.4)%			14	26			15	26			127	96			69	101

Based on the above comparison, we can state that (i) the performance of SBFL algorithms can be improved by using frequency information, (ii) the graph-based approach produces better (or almost as good) results as the reference algorithms in many cases, and (iii) the weighted, graph-based approach using call frequency finds the faulty methods more efficiently than the other presented algorithms.

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, he was actively involved in processing the literature, planning the experiments and measurements, and writing the publications.

Publications

- **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes. Fault Localization Using Function Call Frequencies Journal of Systems and Software (JSS), Volume: 193, Elsevier, 2022.
- **B. Vancsics** NFL: Neighbor-Based Fault Localization Technique In Proceedings of the 1th International Workshop on Intelligent Bug Fixing (IBF), Hangzhou, China, pp. 17-22, IEEE, 2019
- **B. Vancsics** Graph-Based Fault Localization In Proceedings of the International Conference on Computational Science and Its Applications (ICCSA), Saint Petersburg, Russia, pp. 372-387, Springer, 2019
- **B. Vancsics**, F. Horváth, A. Szatmári and Á. Beszédes Call Frequency-Based Fault Localization In Proceedings of the 28th International Workshop on on Software Analysis, Evolution, and Reengineering (SANER), Honolulu, HI, USA, pp. 365-376, IEEE, 2021

II BugsJS: a Benchmark of JavaScript Bugs

A high-quality bug data set is essential for the reliable evaluation of fault localization algorithms. There are several benchmarks (for example *Defects4J* [6] – Java), but until now no such data set was available for JavaScript programs. We wanted to fill this gap.

As a first step, we determined the criteria on the basis of which the programs are selected, then we automatically collected the bugs in them, manually and automatically (dynamically) validated them, and then loaded them into the established infrastructure.

During the selection of the projects, we took into account several relevant aspects. One of these was to be available in the GitHub version control system, as well as a server-side Node.js application. In order to identify and validate bugs, it was essential that the project uses GitHub's *issue-tracker* system and that it contains commits with the *bug* tag. In addition, it had to be popular (Stargazers count ≥ 100), "mature" (number of commits > 200), and active (year of the latest commit ≥ 2017) to be considered potential candidates can be included.

For each selected project, we first checked GitHub's official API to see if it was a closed bug with a specific bug tag or not. For each closed bug entry, we automatically searched for the commit that fixed it and the one that was the commit before the fix. We ignored cases where two or more changes fixed the bug. In this way, the status of both the project containing the bug and the one no longer containing the bug were identified.

In the case of manual validation, we set five conditions and in the case of dynamic validation four conditions against the error and commits. In the case of manual validation, the following had to be met:

- 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 the refactoring of the production code.

The four conditions necessary for dynamic validation were established as follows:

- test does not fail: let V_{bug} be the version of the source code that contains a bug b, and let V_{fix} be the version in which b is fixed. The existing test cases in V_{bug} do not fail due to b, however, at least one test of V_{fix} should fail when executed on V_{bug}
- correct dependencies: it could not happen that the tests could not be run due to missing dependencies
- error does not exist in tests: no errors occurred while running the tests
- Mocha: the project uses the Mocha (test)framework

Overall, 795 commits were manually validated, of which 542 (68.18%) fulfilled the criteria. Table 7 (Manual) illustrates the results 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), where the fix did not involve production code (29), or where it 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.

After the dynamic analysis, 453 bug candidates were ultimately retained for inclusion in BugsJS (84% of the 542 bug candidates from the previous step). Table 7 (Dynamic) reports the results of the dynamic validation phase. 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. In 12 cases, tests were failing during the execution, whereas in 40 cases no tests failed when executed on V_{bug}. We excluded all such bug candidates from the benchmark.

Table 7: Manual and dynamic validation statistics per application for all considered commits

		Bower	Eslint	Express	Hessian.js	Hexo	Karma	Mongoose	Node-redis	Pencilblue	Shields	**Total**
	Initial number of bugs	10	559	39	17	24	37	56	25	18	10	**795**
Manuális	✘ Fixes multiple issues	0	18	1	0	1	5	2	5	0	0	32
Manuális	✘ Too complex	0	94	0	4	8	4	8	7	9	2	136
Manuális	✘ Only dependency	1	9	0	0	1	0	2	0	0	0	13
Manuális	✘ No production code	0	20	4	0	1	1	2	0	0	1	29
Manuális	✘ No tests changed	1	0	1	0	0	0	0	1	1	0	4
Manuális	✘ Refactoring	0	36	0	0	0	1	1	1	0	0	39
	After manual validation	8	382	33	13	13	26	41	11	8	7	**542**
Dinamikus	✘ Test does not fail at V_{bug}	1	11	6	4	1	2	8	3	1	3	40
Dinamikus	✘ Dependency missing	3	17	0	0	0	1	1	0	0	0	22
Dinamikus	✘ Error in tests	1	7	0	0	0	0	3	1	0	0	12
Dinamikus	✘ Not Mocha	0	14	0	0	0	1	0	0	0	0	15
	✔ ***Final Number of Bugs***	3	333	27	9	12	22	29	7	7	4	**453**

We performed manual cleaning on the bug-fixing patches, to make sure they only include changes related to bug fixes. In particular, we removed the irrelevant changes (i.e., source code comments, when only comments changed, and comments unrelated to bug-fixing code changes, as well as changes solely pertaining to whitespaces, tabs, or newlines). Furthermore, for easier analysis, we separated the patches into two separate files, the first one including the modifications to the tests, and the second one pertaining to the production code fixes.

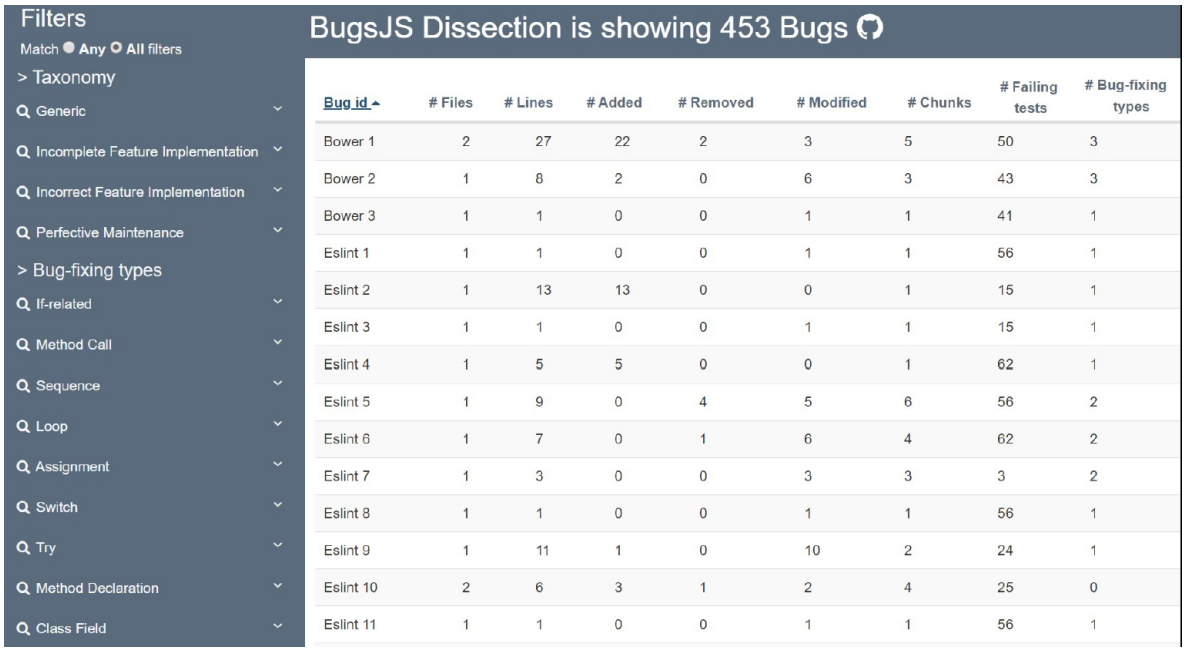
We added the resulting changes to the BugsJS infrastructure and provided users with a command-line interface that (currently) can receive four commands. The interface includes the following commands: (i) info: prints out information about a given bug (ii) check out: checks out the source code for a given bug (iii) test: runs all tests for a given bug and measures the test coverage (iv) per-test: runs each test individually and measures the per-test coverage for a given bug

We have assigned five states to each bug, depending on the changes (bug fixes) it contains:

- Bug-X: The parent commit of the revision in which the bug was fixed (i.e., the buggy revision)
- Bug-X-original: A revision with the original bug-fixing changes (including the production code and the newly added tests)
- Bug-X-test: A revision containing only the tests introduced in the bug-fixing commit, applied to the buggy revision

- Bug-X-fix: A revision containing only the production code changes introduced to fix the bug, applied to the buggy revision
- Bug-X-full: A revision containing both the cleaned fix and the newly added tests, applied to the buggy revision.

Furthermore, BugsJS Dissection (Fig.3 and Fig. 4) was created based on Defects4J Dissection[2], which contains basic information about each bug and is available via the following link: `https://bugsjs.github.io/dissection/`.



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: BugsJS Dissectiont (overview page)

Not only the infrastructure was created, but the bugs in it were also categorized according to their nature (taxonomy) and the bug-fix type. 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 that categories and their sub-categories follow a specialization relationship.

Furthermore, we have categorized the bug fixes according to the code change. For this, we used the change patterns created by Pan et al. [9] and expanded it with 3 new categories.

After the grouping according to the two criteria was done and every bug received at least one taxonomy label and the bug-fix type label, we examined the relationship between them. Based on this, the stronger and looser relationships between the groups could be determined. For example: the connection between the *missing input validation* and the *if-related fix* is strong, but it is weak between the *incomplete data processing* and the *sequence-related fix* groups.

In addition, we demonstrated the usability of BugsJS through a possible area of use, fault localization. We examined 7 projects that contained 336 bugs and evaluated them using three, previously described algorithms (*Tarantula*, *Ochiai*, and *DStar*). The results confirmed what the researchers experienced on the other benchmarks (Table 8): the methods performed similarly, in

[2] `http://program-repair.org/defects4j-dissection/`

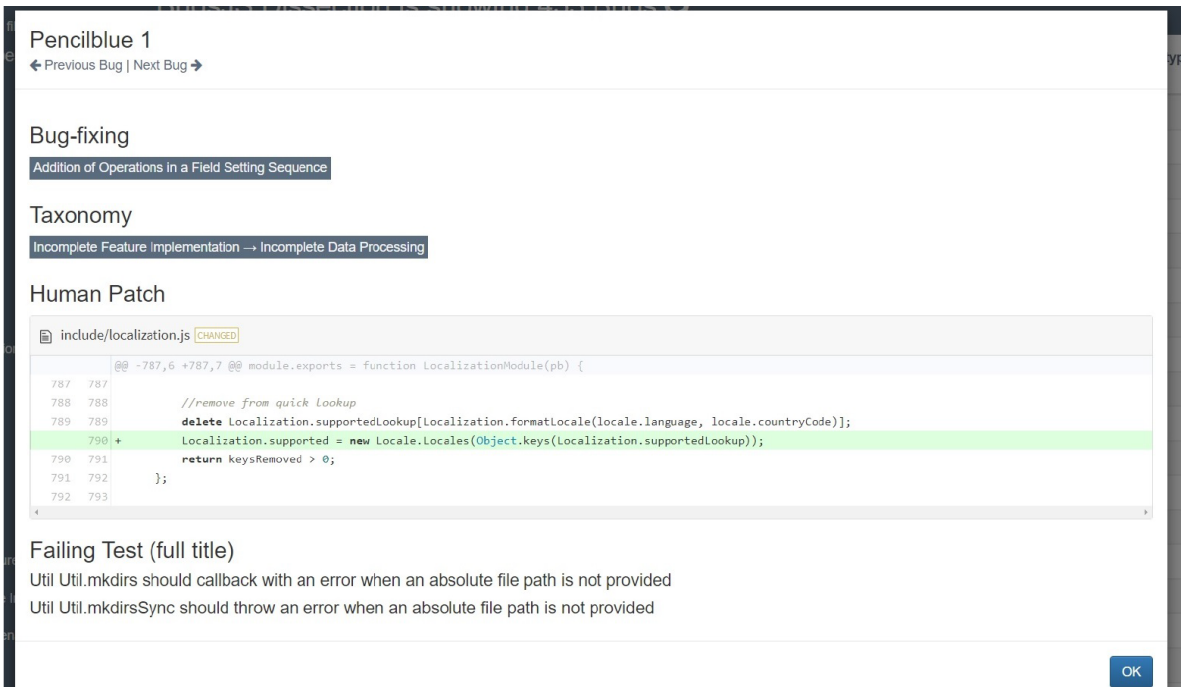


Figure 4: BugsJS Dissectiont (page for one bug)

6 cases out of 7 projects *DStar* performed best average ranks (with the exception of Hessian.js), but overall the best result was achieved by *Ochiai* (column All).

Table 8: Average ranks

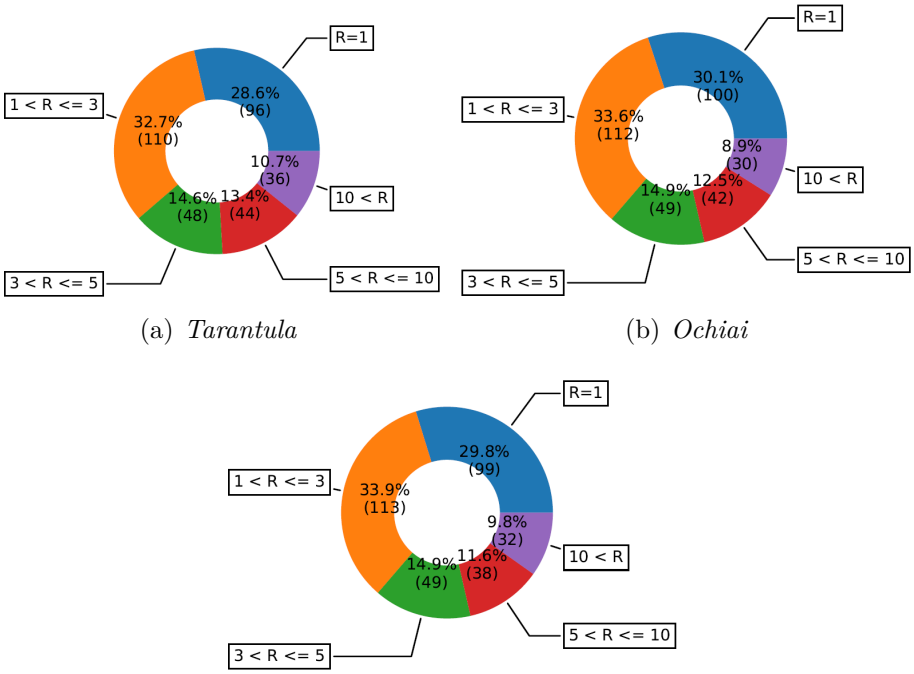
Project	Bower	Shields	Hexo	Hessian.js	Express	Pencilblue	Eslint	All
Tarantula	25.83	5.83	3.25	4.81	8.10	1.83	20.39	18.24
Ochiai	19.17	5.83	3.00	3.88	7.94	1.67	19.90	17.73
DStar	17.50	5.17	80.88	3.00	7.94	1.67	19.90	20.47

If we look at the distribution of the ranks, we reach a similar conclusion (Fig. 5): in 28-30% of the cases, the rank value is 1 (that is, the most suspicious element is the faulty method), almost two-thirds of the bugs get 3 or higher rank (61-63%) and only 9-11% of them for which the rank is lower than 10.

Furthermore, we wondered whether there were bug-fix types that could be found more efficiently by the algorithms mentioned above. Table 9 shows in what percentage of cases the faulty method was the most suspicious (*Top-1*), how many times they resulted in 3 or better (*Top-3*), 5 or better (*Top-5*), 10 or better (*Top-10*) or worse than 10 (*Other*) rank.

It can be seen that in the case of *if-related* types (IF), the faulty methods are found more efficiently than average, more often than average, the faulty method will be at the top of the ranked list for *method call-related* fixes (MC), and in cases related to *method declarations* (MD), the algorithms perform somewhat better. The opposite conclusion can be drawn for *sequence-related*s (SQ). These errors are more difficult to find by the methods and are given a lower rank. Due to the low number of cases (column # in Table 9), it is difficult to draw reliable conclusions about the other categories.

In summary, we presented BugsJS, a benchmark of 453 real, manually validated JS bugs from 10 popular JavaScript programs. The quantitative and qualitative analyses, including a categorization of bugs in a dedicated taxonomy, show the diversity of the bugs included in BugsJS



(a) *Tarantula* (b) *Ochiai*

(c) *DStar*

Figure 5: The distribution of ranks (R) for the algorithms.

Table 9: The distribution of ranks depending on the algorithms and the bug-fixing types

Type	#	**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	176	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	102	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	2	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	24	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	65	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	29	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	5	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	8	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	1	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

that can be used for conducting highly reproducible empirical studies in software analysis and testing research related to, among others, regression testing, bug prediction, and fault localization for JavaScript.

The contributions of the author to this thesis point

The author of the dissertation was actively involved in the investigation of existing bug data sets and 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 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 BugsJS can be used to evaluate and compare the effectiveness of fault localization algorithms and he evaluated the results.

Publications

- 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 (STVR), Volume 31, Issue 4, John Wiley & Sons, 2021

- P. Gyimesi, **B. Vancsics**, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah BugsJS: a Benchmark of JavaScript Bugs In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 90-101, IEEE, 2019

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

- A. Szatmári, **B. Vancsics** and Á. Beszédes Do Bug-Fix Types Affect Spectrum-Based Fault Localization Algorithms' Efficiency? In Proceedings of the 3th International Workshop on Validation, Analysis and Evolution of Software Tests (VST), pp. 16-23, IEEE, 2020

- **B. Vancsics**, P. Gyimesi, A. Stocco, D. Mazinanian, Á Beszédes, R. Ferenc and A. Mesbah Supporting JavaScript Experimentation with BUGSJS In Proceedings of the 12th International Conference on Software Testing, Verification and Validation (ICST), pp. 375-3378, IEEE, 2019

Further Realted Publications

- Horváth, F., Beszédes, Á., **Vancsics, B.**, Balogh, G., Vidács, L. and Gyimóthy, T. Using Contextual Knowledge in Interactive Fault Localization Empirical Software Engineering (EMSE), Volume: 27, Number: 150, Published by Springer, 2022
- Sarhan, Q. I., **Vancsics, B.** and Beszédes, Á Method Calls Frequency-Based Tie-Breaking Strategy For Software Fault Localization In Proceedings of the 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 103-113, IEEE, 2021
- Horváth, F., Beszédes, Á., **Vancsics, B.**, Balogh, G., Vidács, L. and Gyimóthy, T. Experiments with Interactive Fault Localization Using Simulated and Real Users In Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME), Adelaide, Australia, pp. 290-300, IEEE, 2020
- **Vancsics, B.**, Gergely, T. and Beszédes, Á. Simulating the Effect of Test Flakiness on Fault Localization Effectiveness In Proceedings of the 3rd International Workshop on Validation, Analysis and Evolution of Software Tests (VST), London, Ontario, Canada, pp. 28-35, IEEE, 2020
- Gergely, T., Balogh, G., Horváth, F., **Vancsics, B.**, Beszédes, Á. and Gyimóthy, T. Differences Between a Static and a Dynamic Test-to-Code Traceability Recovery Method Software Quality Journal (SQJ), Volume 27, Number 2, pp. 797-822, Springer, 2018
- Gergely, T., Balogh, G., Horváth, F., **Vancsics, B.**, Beszédes, Á. and Gyimóthy, T. Analysis of Static and Dynamic Test-to-code Traceability Information Acta Cybernetica, Volume 23, Number 3 (Special Issue In Memoriam Csanád Imreh), pp. 903-919, Institute of Informatics, University of Szeged, 2018
- Tengeri, D., Vidács, L., Beszédes, Á., Jász, J., Balogh, G., **Vancsics, B.** and Gyimóthy, T. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density In Proceedings of the International Conference on Software Testing, Verification and Validation Workshops: 11th International Workshop on Mutation Analysis (MUTATION), pp. 174-179, IEEE, 2016
- Horváth, F., **Vancsics, B.**, Vidács, L., Beszédes, Á., Tengeri, D., Gergely, T. and Gyimóthy, T. Test Suite Evaluation using Code Coverage Based Metrics In Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST), CEUR Workshop Proceedings. Vol. 1525., pp. 46–60, 2015
- Binkley, D., Beszédes, Á., Islam, S., Jász, J. and **Vancsics, B.** Uncovering Dependence Clusters and Linchpin Functions In Proceedings of the 31th International Conference on Software Maintenance and Evolution (ICSME), pp. 141-150, IEEE, 2015
- Vidács, L., Horváth, F., Mihalicza, J., **Vancsics, B.** and Beszédes, Á. Supporting Software Product Line Testing by Optimizing Code Configuration Coverage In Proceedings of the 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 1-7, IEEE, 2015

Summary

In my thesis, I focused on a very important subtask of software development, software testing, including a small segment of it, fault localization. Iin my dissertation I deal with one of the most well-known and popular families of fault localization (FL) algorithms, which is based on the source code coverage information, and in the second part, I present the benchmark we created, with the help of which the efficiency of the algorithms can be objectively measured and thus compared.

I present in detail the concept of coverage-based debugging procedures (so-called *spectrum-based fault localization (SBFL)* algorithms), the best-known methods, and other relevant literature. After that, I present two approaches based on graph attributes/properties, which try to determine the location of the bugs as precisely as possible. For this, it builds (from the SBFL input data) a so-called coverage graph, from which it identifies the most suspicious elements by extracting different neighborhood information.

Then the concept of UDCS (unique, deepest call stack) was introduced, which is illustrated by a small example. It is described in detail how the method call frequency information that can be extracted from the UDCS can be adapted in the previously described formulae used by SBFL algorithms.

Finally, I showed how the two concepts described above (graph- and frequency-based) can be combined, thereby making use of the potential opportunities of the two methods.

I performed the quantitative evaluation on the *Defects4J* (Java) benchmark, which showed that (i) the two graph-based methods are in many cases more efficient than the examined SBFL methods, (ii) methods using frequency information achieve significantly better results than methods based on binary coverage, and (iii) the best result was obtained by combining the two concepts, the *WENFL* algorithm.

A high-quality bug data set is essential for the reliable evaluation of fault localization algorithms. There are several benchmarks (for example, *Defects4J* – Java), but until now no such data set was available for JavaScript programs. In the second half of the dissertation, I presented how this new data set (BugsJS) was created, I describe in detail the steps involved in the selection of the projects (10 projects), the detection, validation, and "cleaning" of the 453 bugs in them, and their analysis according to different aspects.

In addition to these, we created the grouping of the bugs in the data set (which covers a hierarchical categorization), and the groups in the taxonomy were presented in detail, using real examples. Furthermore, we analyzed and categorized the bugs according to what (code)changes occurred during the fix, and we examined the relationship between the taxonomy categories and bug-fixing types. Finally, in a possible use case, we demonstrated how BugsJS can be used by researchers through the examination of fault localization algorithms.

I hope that these results will contribute to the creation of more efficient fault localization algorithms and that the evaluation of performances will be more reliable in the future.

Acknowledgements

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 additional, 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.

Béla Vancsics, 2023

References

[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 Zoeteweij, 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 Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11), 2009.

[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE, 2007.

[5] 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, New York, NY, USA, 2005. ACM.

[6] 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, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[7] 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.

[8] 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.

[9] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[10] 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.

[11] 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.

[12] 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.