Assisting Bug Report Assignment Using Automated Fault Localisation: An Industrial Case Study

Jeongju Sohn* KAIST Daejeon, Korea kasio555@kaist.ac.kr

Gabin An* KAIST Daejeon, Korea agb94@kaist.ac.kr

Jingun Hong SAP Labs Seoul, Korea jingun.hong@sap.com

Dongwon Hwang SAP Labs Seoul, Korea dong.won.hwang@sap.com

Shin Yoo KAIST Daejeon, Korea shin.yoo@kaist.ac.kr

***Abstract*—We present a case study of an industry scale application of automated fault localisation to SAP HANA2 database. When a test breaks in the Continuous Integration (CI) pipeline, the bug needs to be triaged and assigned to the appropriate development team. Given the scale and complexity of SAP HANA2, the assignment itself can be a challenging task. The current practice depends on the static mapping between test scripts and software components, as well as human domain knowledge. We apply automated fault localisation to aid the issue allocation in the CI pipeline: once a test failure is observed, the automated fault localisation technique identifies the suspicious software component using the information from the test failure. The localisation result can be used by the issue manager to allocate the incoming test failure issues more efficiently. We have analysed 137 CI test executions with at least one failing test script using Spectrum Based Fault Localisation. The results show that automated fault localisation can identify the faulty software component for 61 out of 137 studied test failures within top 10 places out of over 200 components. Out of the 61 faults, 36 faults were not identifiable based on the static mapping between test script and software components at all.**

***Index Terms*—fault localisation, spectrum based fault localisation, test suite diagnosability**

I. Introduction

As software systems grow in their size and complexity, potential faults also become increasingly complex and more difficult to fix. It has been reported that the debugging cost of software failures can take up to 80% of the entire development cost [25]. At the same time, a variety of factors including huge pressure on shorter time to market, and the decentralised organisational structure of software development, requires *continuous* testing and delivery [5]. Consequently, it is crucial that any defect is located, analysed, and patched as quickly as possible.

Fault localisation [29] aims to locate the part of the source code that is the root cause of an observed failure. Only after the fault is localised can an appropriate patch be designed, applied, and tested again. A precise and efficient fault localisation technique is essential for the productivity of the debugging pipeline. To address this issue, many automated fault localisation techniques have been widely developed and studied to precisely locate the cause of the failure without human intervention [29]. Some are static, i.e., do not require any information from actual test execution, and instead use the similarity between bug reports and source code locations [21], [13], [26]. Others are based on dynamic analysis, such as test results and coverage [10], [4], [1], [9], [16], [33], [31], or mutation analysis [18], [15], [8], [7]. However, while there have been controlled human experiments of fault localisation techniques [30], [19], [26], there has been little work on industrial evaluation of automated fault localisation techniques.

This paper aims to evaluate automated fault localisation technique in the context of Continuous Integration (CI) and deployment of a large industrial software system. We focus on the bug report assignment stage of the CI pipeline of SAP HANA2, a large scale in-memory relational database system. Currently, each test script in SAP HANA2 test suite is pre-mapped to a component it is supposed to test. When a test script fails, a QA manager manually assigns the resulting bug report to a team, after considering the mapping between test scripts and components. The manual assignment is both laborious and time consuming. Furthermore, due to the complex inter-dependency between components in SAP HANA2, the assignment based on fixed mappings can be inaccurate, resulting in the report being bounced back and forth between teams for long time. Such a ping pong session can significantly increase the time required to patch the bug [3], [12], [34].

We have extracted 137 known test executions that include at least one failing test script from the test database of SAP HANA2: for each of such failures, we also recover which component had actually caused the failure, based on the actual fix commit. Consequently, our ground truth is the actual root cause of the failures, rather than the reasoning based on the pre-existing mapping between test scripts and components. We have applied Spectrum-based Fault Localisation (SBFL) [29], an automated localisation technique that uses test results as well as test coverage information, to these known test executions and evaluated how precisely SBFL techniques can identify the components responsible for the failure in each test execution. To aid our analysis, we have also investigated the diagnostic capability of the SAP HAHA test suite for fault localisation, studied the correlation between the age of coverage information and the accuracy of fault localisation, and finally applied a voting-based aggregation to improve the accuracy of the localisation. Using coverage information that are collected weekly, the voting-based aggregation can narrow down the problematic component within top 10 out of over

*Both authors contributed equally to this work.

200 components for 61 out of 137 studied test executions. We also discuss future research direction that can improve the accuracy further.

Below are the main contributions of this paper.

- We present the first industrial scale case study of automated fault localisation techniques. The evaluation uses test information from SAP HANA2 in-memory relational database, whose code base includes more than $27K$ files and over 1,500 test scripts. Our aim is to identify the faulty component automatically to aid bug report assignment in the Continuous Integration (CI) pipeline.
- The empirical evaluation shows that simple Spectrum-based Fault Localisation (SBFL) can successfully narrow down the number of suspicious component to the maximum of 10, out of over 200 components, for 33 out of 137 studied test executions.
- We perform the Density, Diversity, and Uniqueness (DDU) analysis [20] to highlight the limits in the existing test suite for SAP HANA2. We also present how aggregation mechanisms based on code structure and voting can elevate automated fault localisation techniques to software component level.

The rest of the paper is organised as follows. Section II introduces background information about SBFL as well as SAP HANA2 test framework. Section III presents the result of the initial application of SBFL techniques. Section IV contains the results of test suite diagnosability evaluation, which provides further insights into the SBFL results. Section V introduces a voting-based ensemble approach to cope with the limitations in test suite composition. Section VI discusses the findings of this study as well as the future work, and Section VII concludes.

II. Background

In this section, we define some basic notation and describe SBFL based on those notations.

A. Basic Notation

Given a program P, following properties can be defined:

- $E = \{e_1, e_2, ..., e_n\}$ is a set of program components consist of a program P, e.g., lines or methods,
- $T = \{t_1, t_2, ..., t_m\}$ is a set of test cases of P,
- C is a $m \times n$ coverage matrix:

$$C = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m,1} & c_{m,2} & \cdots & c_{m,n} \end{bmatrix} = (c_{i,j}) \in \mathbb{B}^{m \times n}$$

, where $c_{i,j} = 1$ if t_i executes e_j, otherwise 0.
- $F(t)$ is a proposition that means a test t fails. If $\exists_{t \in T} F(t)$, the program P is regarded as *faulty*.

B. Spectrum-based Fault Localisation

Spectrum-based fault localisation (SBFL) is a technique to find the faulty elements of a program using *program spectrum*, which is a summary of test coverage information. *Program spectrum* of a program element consists of four values: (e_p, n_p, e_f, n_f).

Formally, each spectrum value of $e_i \in E$ can be defined as follows:

- ${e_p}^1$ = $|\{1 \leq j \leq m | c_{i,j} = 1 \land \neg F(t_j)\}|$
- ${n_p}^2$ = $|\{1 \leq j \leq m | c_{i,j} = 0 \land \neg F(t_j)\}|$
- ${e_f}^3$ = $|\{1 \leq j \leq m | c_{i,j} = 1 \land F(t_j)\}|$
- ${n_f}^4$ = $|\{1 \leq j \leq m | c_{i,j} = 0 \land F(t_j)\}|$

The main rationale of SBFL is following: *"The more frequently covered by failing tests and the less frequently covered by passing tests, the more suspicious the program element is"*. Researchers have proposed many risk evaluation formulæ [2], [11], [17], [28] that concretise this fundamental idea of SBFL and estimates the suspiciousness of each program element. A formula takes the spectrum of a program element as input and returns the suspiciousness score.

		Program Elements							
		e₁	**e₂**	**e₃ (faulty)**	**e₄**	**e₅**	**e₆**	**e₇**	
Tests	**t₁**	0	0	0	1	0	1	0	PASS
	t₂	1	0	1	0	1	1	1	FAIL
	t₃	0	1	1	0	0	0	0	FAIL
Program Spectrum	e_p	0	0	0	1	0	1	0	
	n_p	1	1	1	0	1	0	1	
	e_f	1	1	2	0	1	1	1	
	n_f	1	1	0	2	1	1	1	
Score	Ochiai	0.71	0.71	1.00	0.00	0.71	0.50	0.71	
Ranking	dense	2	2	1	4	2	3	2	
	min	2	2	1	7	2	6	2	
	average	3.5	3.5	1	7	3.5	6	3.5	
	max	5	5	1	7	5	6	5	

Fig. 1: Example of calculating suspiciousness scores

For example, Ochiai [2], one of the state-of-the-art risk-evaluation formula, compute the suspiciousness score of a given program element e by:

$$Ochiai(e_p, n_p, e_f, n_f) = \frac{e_f}{\sqrt{(e_f + n_f) \times (e_f + e_p)}} \quad (1)$$

Figure 1 demonstrates the calculation of Ochiai scores from the coverage matrix and test results.

C. Overview of the SAP HANA2 Test Framework

SAP HANA2 project consists of several components; each file belongs to only one component, and each line belongs to only one file. The test-suite of SAP HANA2 is a collection of more than 1,500 test scripts, and each test script contains from 1 to 20,000 atomic test methods.[5] Each test script belongs to one of the components as like a normal file.

[1] the number passing tests that execute a program element
[2] the number passing tests that do not execute a program element
[3] the number failing tests that execute a program element
[4] the number failing tests that do not execute a program element
[5] In SAP HANA2, the test script and the test method are called *Test Case* and *Single Test*, respectively.

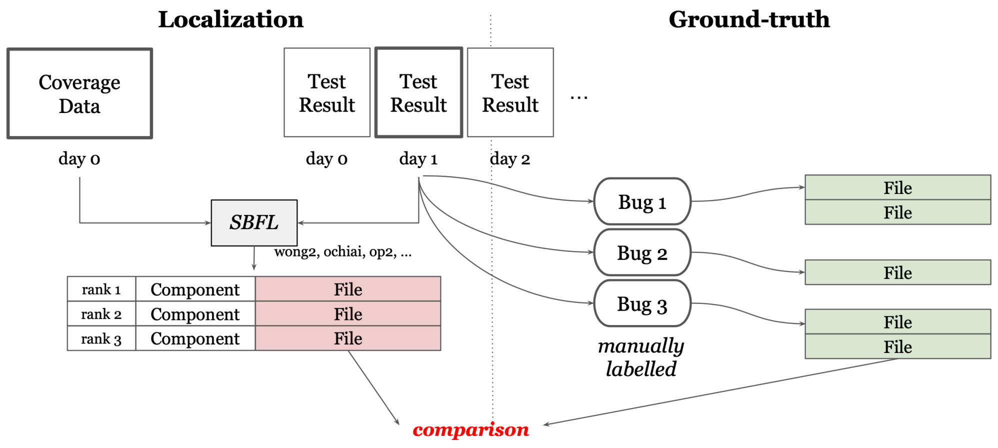


Fig. 2: Overview of SBFL experiment

SAP HANA2 test framework runs a daily test for steady quality assurance, and each daily test usually consists of executing more than $1,000$ test scripts. If some tests fail, the failure is reported immediately and passed on to the bug tracking system. If the failure is a previously discovered one, the same bug label is given. Otherwise, a new label is created. Each bug is manually assigned to a component that appears to be responsible for that bug. To aid this assignment process, there exists a mapping between test scripts and components they are supposed to test. However, as we show later in Section III-C, the mapping alone cannot accurately localise all failing test executions, hence the need for the manual assignment. Once a bug is resolved, the corresponding modifications are mapped to the bug as `fix commits`.

Despite the daily test runs, test coverage is measured only on weekly basis due to the huge project size. Coverage data is collected for each test script, not atomic test method, at a line-granularity level. As a result, we can get $m \times n$ line level coverage matrix C^l, where m is the number of test scripts, n is the number of Source Lines of Code (SLOC).

III. Fault Localisation on SAP HANA2

To analyse the feasibility of applying SBFL on SAP HANA2, we design a fault localisation experiment using reported-and-resolved bugs of SAP HANA2.

A. Experiment Design

Figure 2 describes the overview of our experiment. We first collect the daily test result that consists of test failure. If there exist more than one execution for one test, we only take the last execution. For each daily test, we localise faults using the recent coverage data [6] and SBFL techniques. Since only weekly coverage data are available, we use the latest recent coverage data that is six days old at maximum. After applying an SBFL formula to the coverage data, we obtain the suspiciousness score for each line. However, since line level is too fine-grained in this scale of the project, we aggregate the results to file and component level using *max aggregation scheme*, which assigns each file with the highest score among the lines in that file, and, in turn, each component with the highest score among its files. The final localisation result is a ranked list of files and components, ordered by their suspiciousness scores respectively. Note that each belongs to exactly one component, as mentioned in Section II-C.

To evaluate the localisation results, we collect resolved bugs that are explicitly linked to fix commits. Assuming that every file in a fix commit contains the root cause of the test failures, we call them ground-truth faulty files. Finally, we compare the localisation results with the ground-truth to assess the performance of SBFL on the SAP HANA2 project.

[6]Based on make_id, which is assigned to each build, not time

B. Details of Implementation and Evaluation

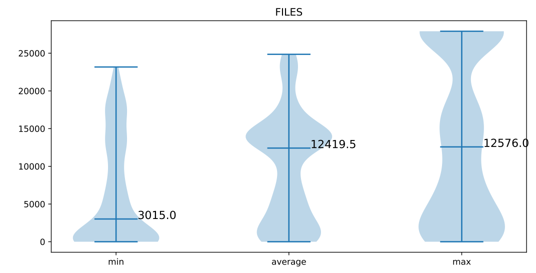
As an evaluation subject, We collect six coverage data between 2019/12/08 and 2020/02/02, and also the results of all test runs executed within the same time period, of which there are 229. We discarded 91 out of 229 test executions whose fix commits include no modification of any files or components existing in the coverage data, e.g., omission faults, as those are out-of-scope for SBFL. This leaves 138 faulty test executions for the file level analysis. In the component level, we excluded one more test execution where the file-component mappings at the test run and coverage measurement was inconsistent. This leaves 137 faulty test executions for the component level analysis.

As a risk evaluation formula, we use Ochiai that is previously described in Section II-B. The ranks of each file are calculated using the following three tie-breaking methods:

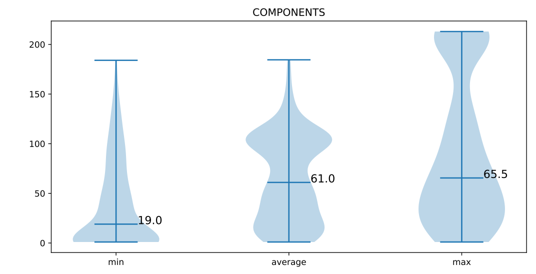
- **min** assigns the minimum of the ranks (the highest) to all the tied values.
- **average** assigns the average of the ranks to all the tied values.
- **max** assigns the maximum of the ranks (the lowest) to all the tied values.

The example of ranking calculation is presented in Fig. 1. It is worth noting that the more the tied values there are, the larger the gap between the ranks with different tie-breaking schemes become.

To evaluate the localisation results, we use *acc@n*. For every test execution in our subject, *acc@n* measures the number of test runs where at least one of faulty program elements are ranked within the top n location at both file and component level.



(a) Ranking of faulty files



(b) Ranking of faulty components

Fig. 3: Ranking of faulty files and components with different tie-Breaking schemes (showing medians)

C. Results

The results of our experiment are shown in Table I and Figure 3. We present the ranking of both components and files.

As shown in the *acc@1* column of Table I, Ochiai is able to rank a faulty file at the top for 59 out of 138 test executions with the min tie-breaker. However, if the max-tie breaker is used, the number becomes 1, which means that there exists a large number of tied scores. Also, from the distribution of ranks of faulty files and components shown in Figure 3, we can observe that the ranking varies greatly depending on the tie-breaker. Similar results are observed using other formulæ, Jaccard [2] and Tarantula [11].

At the component level, we can localize 100 out of 137 (79%) of components within the ranking 10 with Ochiai and min-tie breaker. However, there are also quite large gaps between results of the different tie-breakers. Since we use the max-aggregation scheme for the calculation of component scores, the components would inherit the ties in file level.

The component ranking results of SBFL are complementary to the mapping between failing tests and their components.

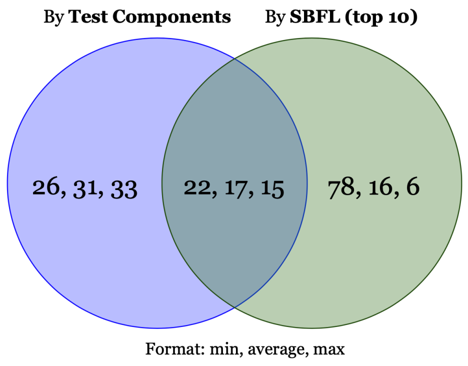


Fig. 4: Comparison between the component mapping and SBFL within the top 10 places

In Figure 4, the the right circle represents the set of failing test executions for which Ochiai ranks the ground truth faulty components within the top ten places, whereas left circle represents the set of failing test executions for which the component mapped to the failing tests are actually the ground truths. The Venn Diagram shows that there are 16 failing test executions whose root causes cannot be found by using the mapping between test scripts and components; SBFL successfully localise these failures using the average tie-breaker. Note that the test-component mapping based localisation is only correct for 58 of the failures constantly: the sets of three numbers are due to the changes in the SBFL results due to different tie breakers. The Venn Diagram also shows that the test-component mapping cannot precisely localise all failures, as it is only correct for 48 out of 137 faults.

D. Correlation Between the Age of Coverage Information and Fault Localisation Performance

As explained in Section III-A, fault localisation takes the latest coverage data as inputs, under the assumption that the more recent the coverage is, the accurate the localisation result will be as the used coverage is likely to be closer to the actual coverage at the time of testing. To validate this assumption, we investigate the correlation between the age of coverage and the performance of fault localisation (SBFL) using Spearman correlation analysis.

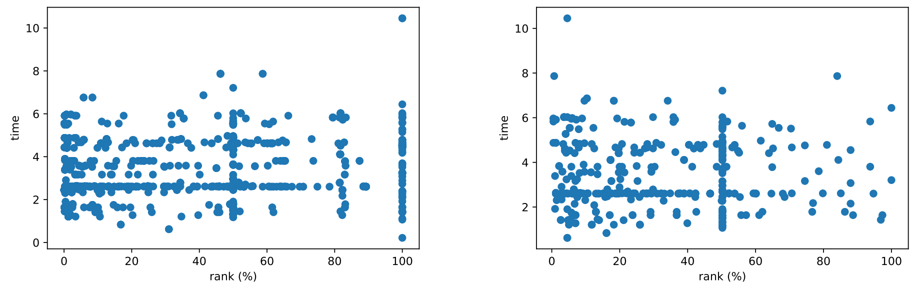
Table II presents the coefficients and p-values from this analysis. There is a weak correlation between coverage age and localisation accuracy at the file level, but the correlation becomes both weaker and not statistically significant at the component level. The correlation is the weakest when the min tie-breaking scheme is used. We suspect that this is due to the overestimation made by the min tie-breaking scheme as it assigns the highest rank to all tied program elements. Fig. 5 shows the scatterplot of coverage age and localisation accuracy.

IV. Test Diagnosability Assessment

The analysis in Section III has shown that there were many ties in the SBFL results on SAP HANA2, which are usually

TABLE I: Results of fault localisation with Tarantula and Jaccard

Formula	Granularity	Tie-Breaker	Total	acc@1	acc@3	acc@5	acc@10
Ochiai	File	min	138	59	60	62	64
		average	138	1	1	2	3
		max	138	1	1	1	3
	Component	min	137	85	94	97	100
		average	137	5	16	19	33
		max	137	5	11	16	21
Tarantula	File	min	138	68	68	69	70
		average	138	1	1	1	2
		max	138	1	1	1	2
	Component	min	137	104	105	105	106
		average	137	1	5	8	21
		max	137	1	3	5	8
Jaccard	File	min	138	56	57	59	61
		average	138	1	1	2	3
		max	138	1	1	1	3
	Component	min	137	80	86	90	98
		average	137	4	12	17	32
		max	137	4	8	14	20



(a) Files (b) Components

Fig. 5: Scatter plots of Spearman correlation analysis between coverage age (days) and performance

TABLE II: Results of Spearman correlation analysis between the coverage ages and the performance

Tie-Breaker	File		Component	
	corr	pval	corr	pval
min	0.1079	0.0001	0.0542	0.2818
max	0.2768	0.0000	-0.0702	0.1631
average	0.2825	0.0000	0.0339	0.5007

	e1	e2	e3	e4	
t1	●	●	●	●	PASS
t2	●	●	●	●	FAIL
t3	●	●	●	●	PASS
t4	●	●	●	●	PASS

Fig. 6: Example of the coverage matrix with a low diagnosability. t_i refers to each test case, and e_i refers to each program element. The black circle means that a program element is coverage by a test.

due to the structural characteristics of the test-suite. In general, to successfully apply SBFL to a project, the coverage matrix of the test-suite must be suitable for diagnosing faults, i.e. a test suite should have an ability to effectively locate faults given a failure. We call this property the *diagnosability* of the test-suite. For example, if every test in the test-suite covers all program elements as the illustrative example in Figure 6, it will be difficult to isolate the faulty program elements even though a fault is revealed by the test failure.

To analyse the cause of frequent ties, we conduct a diagnosability analysis on SAP HANA2 to determine whether its test-suite is suitable for applying SBFL. We use DDU [20], a state-of-the-art test-suite diagnosability metric for SBFL. We briefly describe the metric in Section IV-A and present the assessment result in Section IV-C.

A. Introduction of DDU

Perez et al. [20] proposed a metric DDU to quantify a diagnosability of a test-suite by complementing three adequacy metrics: ***Density***, ***Diversity***, and ***Uniqueness***. Each has the following meaning:

- *Density* ensures that program components are frequently involved in tests so that the coverage matrix have a optimal density, 0.5,
- *Diversity* ensures that tests cover diverse combinations of components,
- *Uniqueness* ensures program components are distinguishable, i.e., covered by different sets of tests.

A DDU value is then defined as the multiplication of all three metrics. More details and the formal definition of each metric can be found in the original work [20].

B. File level Diagnosability Assessment

As mentioned in Section II-C, test coverage C^l is measured at line level. However, due to the enormously large size of coverage matrix, we deduce file level coverage matrix C^f from C^l which is more coarse-grained and relatively small. Using C^f, we approximate the diagnosability of SAP HANA2 test-suite by computing its DDU value. We use a recent coverage matrix measured on December 22nd, 2019 (make id = 6248833), where the number of test scripts is $1,558$, and the number of files is $27,907$.

C. Result and Implication

A assessment result is summarised as follows:

- density = 0.869
- diversity = 0.999 ($1,543$ unique tests found among $1,558$ tests)
- uniqueness = 0.199 ($5,549$ unique coverage spectrum found among $27,907$ files)
- As a result, DDU = 0.173

At the file level, the test-suite of SAP HANA2 shows a low DDU score resulting from low uniqueness value. Although this is an approximated version of original line level coverage data, the low uniqueness implies that there are many and large ambiguity groups, the set of program elements coverage by same test cases, in SAP HANA2 test suite. The files in the same ambiguity groups may be assigned similar[7] suspiciousness scores which may harm the performance of SBFL.

Figure 7 shows the size of 10 biggest ambiguity groups ordered by their sizes (the number of files). Interestingly, the largest group with size $2,875$ is run by the same $1,444$ tests, 92% of all test scripts, and the third largest group with size 868 was run by no tests. Theoretically, when the number of test is $1,543$, the combination of tests by which a program element can be executed is 2^{1543}. That means even if we have far more program elements than test scripts, every program element can be executed by different set of tests. However, in SAP HANA2, a large amount of program files are executed by the exact same set of test cases. Writing tests that can break the ambiguity groups or splitting existing test cases would be helpful for improving the effectiveness of SBFL on SAP HANA2.

[7]Not exactly the same, because scores will be calculated at line level and then aggregated

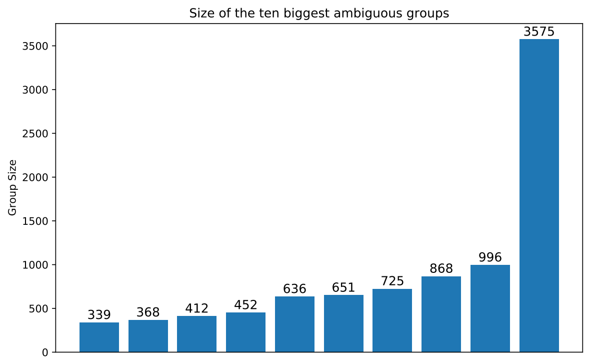


Fig. 7: The size of ten biggest ambiguity groups

We now focus on ways to improve the component level localisation accuracy, as the bug report assignment task is directly coupled to the component level localisation. Additional analysis in Figure 8 also shows that, although it is difficult to pinpoint faulty files due to these structural limitations, if we know exactly which faulty components are faulty, the search space for faulty files is much less than looking at the entire project. Figure 8 shows that the distribution of ranking of faulty files within their components: the ranking of faulty files are much higher compared to the previous results Fig. 3a. This suggests that it may be possible to provide file level localisation for developers, once we achieve sufficiently accurate component level localisation. Therefore, in the following section, we propose a more effective localisation method for finding faulty components by breaking ties using voting-based aggregation.

V. Voting-based Aggregation for Component Level Fault Localisation

Section III-C shows that SBFL can effectively localise some faults that can not be found by test-component mappings. However, the use of the max aggregation scheme results in too many ties between program elements at both file and component granularity. To alleviate the tie issue, we adopt a voting based aggregation scheme. Voting has been used to improve fault localisation in the context of ensemble learning [24]. Here, we use voting to replace max aggregation at the component level. The intuition is that the component that contains more files that are deemed suspicious is also more likely to be the root cause of the failing test execution, when compared to those with fewer suspicious files.

Under the voting based aggregation scheme, each file votes for the component it belongs to; the more suspicious the file is, the more votes the file casts. The suspiciousness of a component is determined by the amount of votes it has

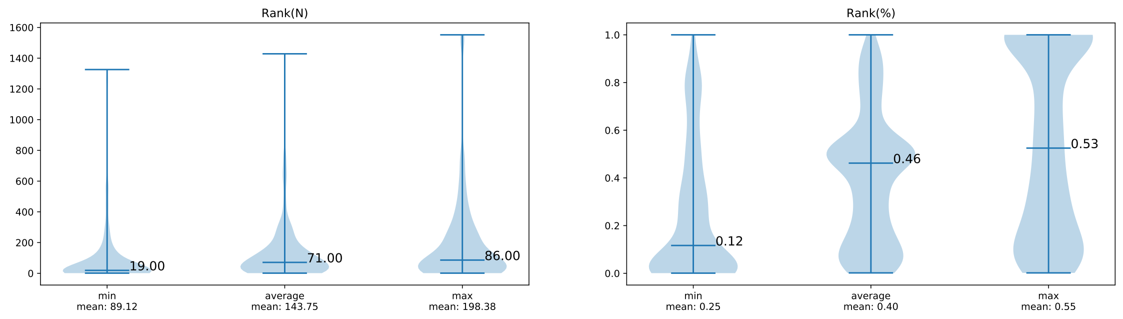


Fig. 8: Ranking of faulty files in their components (showing medians)

received. As such, the suspiciousness score of a component can be considered as a summary of the suspiciousness of the files included in the component. The total vote a component receives is defined as Equation 2, in which F_c denotes the set of files in a component c, and $vote(f)$ denotes the amount of vote a file f casts:

$$NV_c = \sum_{f}^{F_c} vote(f) \tag{2}$$

A. Voting-based Aggregation

The total votes for a component, shown in Equation 2, depends on how $vote(f)$ is defined. This section introduces four variations of $vote(f)$: they differ from each other by the tie-breaking scheme used at the file level ranking, as well as extra factors used to compute the amount of votes each file can cast.

1) Dense Rank Based Voting (V_D): Dense rank based voting, V_D, uses a dense tie-breaking scheme for the ranking at the file level, and uses the resulting dense rank to compute the amount of vote cast by each file. Dense tie breaker ranks files as tightly as possible, i.e., without any skipped rank between files. For example, suppose there are files A, B, C, and D, with suspiciousness scores 0.8, 0.8, 0.6, and 0.6. respectively. Under the dense tie-breaking scheme, both file A and B are ranked at one, and file C and D are ranked immediately after the rank of A and B at rank two. With V_D, the amount of votes a file cast is inversely correlated to its dense rank; it does not consider the number of tied elements.

$$V_D(f) = \frac{1}{dr_f} \tag{3}$$

2) Dense Rank Based Tie Aware Voting (V_{DNT}): V_{DNT} is a variation of V_D that considers the number of tied elements in addition to the ranks of files. Since V_D does not consider the number of tied files, all files tied at the same dense rank will cast the same amount of vote. Consequently, some components may receive a large amount of vote not because its constituent files were ranked higher, but because many of its files were tied. To alleviate this issue, V_{DNT} decreases the amount of vote a file f can cast in proportion to the number of files tied with f. Equation 4 shows the details of V_{DNT}; here, $N_{tie}(f)$ is the number of files tied to a file f.

$$V_{DNT}(f) = \frac{1}{dr_f * N_{tie}(f)} \tag{4}$$

3) Minimum Rank Based Voting: V_M: V_M is similar to V_D but uses minimum tie breaker to form the file level ranking. The minimum tie-breaker considers the number of tied files in its resulting ranking. Let us consider the previous example of files A, B, C, and D with suspiciousness scores 0.8, 0.8, 0.6, and 0.6. The dense ranking will be A and B at rank one and C and D at rank two. In this case, the minimum tie breaker will skip the rank two and place A and B at rank one, C and D at rank *three*. Compared to V_D, V_M penalises files that are ranked lower by making the divisor larger, as can be seen in Equation 5 (note that the minimum rank mr_f is equal to or lower than the dense rank, dr_f).

$$V_M(f) = \frac{1}{mr_f} \tag{5}$$

4) Dense Rank Based Suspiciousnesss Aware Voting (V_{DS}): Finally, V_{DS} extends V_D using the suspiciousness score of each file, as shown in Equation 6: $susp_f$ denotes the raw suspiciousness score of the file f. The intuition is that, by reflecting the actual suspiciousness score in the amount of vote, we will be able to break further ties.

$$V_{DS}(f) = \frac{susp_f}{dr_f} \tag{6}$$

5) Utilising Test-Component Mappings: Results in Section II-C suggest that the test-component mapping can be complementary to SBFL results; Figure 4) shows that the mapping based localisation can be accurate for a different set of faults from the set for which SBFL is accurate. To exploit this, we propose another voting scheme that allow failing test scripts to vote for the components they are mapped to. Equation 7 shows how the votes from test scripts can be added to other voting schemes described earlier: $C_{failing}$ denotes the set of components mapped to failing test scripts, and $alpha$ is a control parameter that balances two sources of voting:

$$NV_c = \alpha \times [c \in C_{failing}] + \sum_{f}^{F_c} vote(f) \tag{7}$$

B. Voting Results

In this section, we evaluate the voting-based aggregation in terms of factors that may affect the performance and compare its performance to both SBFL without voting and test-component mappings.

1) How does voting-based aggregation contribute to the performance?: Compared to the max aggregation based localisation in Section III-C, our voting-based aggregation method could not only break the tie, but also localise more faults. Fig. 9e shows the distribution of rankings using V_D with different component level tie-breaker. From this figure, we observe that the type of the tie-breaker has little effect on the final rankings of V_D. Other voting schemes, i.e., V_{DNT}, V_M, and V_{DS}, have also shown the similar result in breaking the ties. From these results, we can conclude that our voting schemes successfully breaks most of the ties between components in rankings. Hence, we decide to present only the result with the max tie-breaker, which is the most conservative tie-breaking scheme, hereafter.

Table IV shows the evaluation results of the voting-based aggregation in terms of $acc@n$ with the max-tie breaker. Compared to the initial results with max-aggregation in Table I, the voting-based aggregation can localise from 220% to 340% more faults at the top and from 95% to 190% more faults within the top ten depending on the voting scheme. Overall, the voting-based aggregation successfully localises faulty components at the top for up to 16% of all faulty executions (22 out of 137) and within the top ten for up to 44% of all faulty executions (61 out of 137).

In Section III-C, we have observed that SBFL and the mapping between components and test scripts are complementary to each other. We investigate how the voting-based aggregation affect this complementary relation. The Venn Diagram in Fig. 10 shows that there are 36 out 57 failing test executions whose faulty components cannot be found by the mappings between test scripts and components but can be localised by V_D in top ten places. Compared to Fig. 4, with the max-tie breaker, V_D is able to find 30 more faults that cannot be localised by test-component mappings.

2) Does using different voting schemes affect the performance?: Among the four voting methods introduced in Section V-A, V_D achieves the best performance in terms of $acc@1$, while V_{DNT} is the most effective in terms of $acc@10$. As V_{DNT} is a variation of V_D that takes the number of tied files, N_{tie}, into account, the use of N_{tie} decreases the amount of votes from tied files, e.g., if a file is ranked at the top ($dr_f = 1$) but tied to two other files ($N_{tie}(f) = 3$), $V_{DNT}(f)$ will be $\frac{1}{3}$ in V_{DNT} while $V_D(f) = 1$.

To understand the impact of N_{tie} on $acc@1$ and $acc@10$ from V_D and V_{DNT} respectively, let us consider the following example. Files a_1, a_2, b, c_1, and c_2 belong to component A, A, B, C, and C, respectively. For the sake of clarity, let us assume that no other files are mapped to components A, B, and C. Suppose a_1, a_2, b, and c_1 are tied at rank n, followed by c_2 at rank $n+1$. Table III shows the resulting component level rankings generated by V_D and V_{DNT}, respectively. The expressions inside parentheses show the amount of vote each component receives from files. We show that, the component level rank of C is always higher than that of A when using V_{DNT}.

Ranking	Files	Components (V_D)	Component (V_{DNT})
n	a_1, a_2, b, c_1	A ($\frac{2}{n}$)	C ($\frac{1}{4n} + \frac{1}{n+1}$)
$n+1$	c_2	C ($\frac{1}{n} + \frac{1}{n+1}$)	A ($\frac{2}{4n}$)
$n+2$	-	B ($\frac{1}{n}$)	B ($\frac{1}{4n}$)

TABLE III: An example explaining shifts in ranking between V_D and V_{DNT}: ties at the file level can either increase or decrease the ranking at the component level depending on the choice of voting scheme.

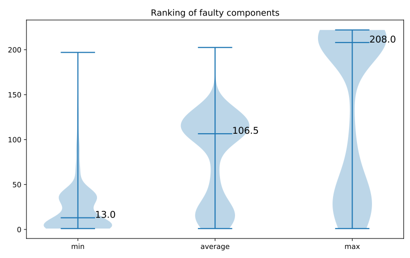
Theorem 1. *Given the file level ranking in Table III, the component level rank of C is always lower than A under V_D, but always higher than A under V_{DNT}.*

Proof. First, we show that rank of C is always lower than that of A under V_D, i.e., $\frac{2}{n} > \frac{1}{n} + \frac{1}{n+1}$. The amount of vote for C is reduced to $\frac{2n+1}{n(n+1)}$, whereas the amount for A is $\frac{2n+2}{n(n+1)}$. Since n is a rank, $n >= 1$. Therefore the inequality holds. Second, we show the opposite, i.e., that the rank of C is always higher than that of A under V_{DNT}, i.e., $\frac{1}{n+1} + \frac{1}{4n} > \frac{2}{4n}$. By rewriting, we get $\frac{5n+1}{4n(n+1)} > \frac{2n+2}{4n(n+1)}$. This holds whenever $n > \frac{1}{3}$. Since n is a rank, $n >= 1$. Therefore, the inequality holds. □

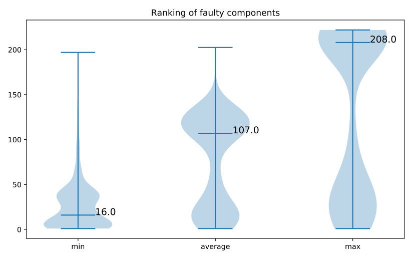
Theorem 1 shows that, depending on the identity of the actual faulty component, the choice between V_D and V_{DNT} can change the ranking of the faulty component in either direction, resulting in changes in $acc@n$. This explains the observed changed between $acc@1$ and $acc@10$ in Table IV. However, overall, results of all voting schemes can outperform the max aggregation results, even with the worst case which is V_{DS}. From these, we conclude that voting-based aggregation can effectively break ties in components, improving the overall accuracy of fault localisation. Since V_D shows the best performance in terms of $acc@1$, we will focus only on the results with V_D through the following sections.

3) How does the number of files participating in voting affect the performance?: We assume that some files might not be suspicious enough to affect the final result. Based on this assumption, we set a hypothesis that excluding these less suspicious files will improve the result by eliminating the noises in data introduced by these files. To validate this hypothesis, we vary the number of files participating in the voting N_{top} and select these files from the top. For example, if N_{top} is five, only the files located within the top five will participate in the voting.

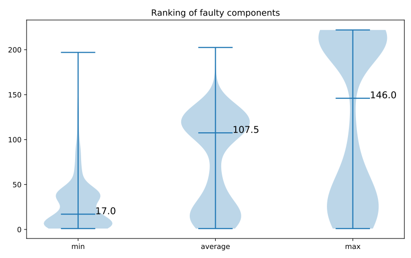
Fig. 9 describes the impact of using different N_{top} values in the effectiveness of the voting. The gap between the results of using different tie-breakers gradually decreases as more and more files participate in the voting. Fig. 9 shows that the median ranking of faulty components becomes higher alongside N_{top}: from 208 with $N_{top} = 1$ to 40 with $N_{top} = N_f$. The voting-based aggregation achieves the best



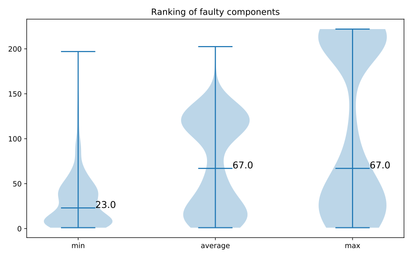
(a) $N_{top} = 1$



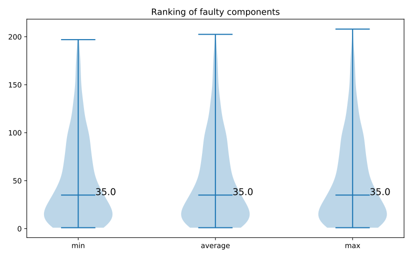
(b) $N_{top} = 3$



(c) $N_{top} = 5$



(d) $N_{top} = 10$



(e) $N_{top} = N_f$

Fig. 9: Violin plots of ranking of faulty components generated by the voting among files ranked within the top N_{top} : $N_{top} = 1, 3, 5, 10, N_f$, N_f = the number of files. Here, V_D is used

performance when all files participate, which is contradictory to our hypothesis that eliminating less suspicious files from voting would improve the localisation results by reducing noises.

We may find a potential explanation for this result from the initial experimental results in Section III-C. In Fig. 3, the median ranking of faulty files is $3,015$ with the min tie-breaker, which means that the ground-truth faulty files might not be located near the top. Thus, eliminating the low-rank files may result in excluding the actual faulty files from the voting. As a result, in SAP HANA2, utilising every file in voting is more effective than eliminating the low-rank files in terms of breaking the ties and localising faulty components.

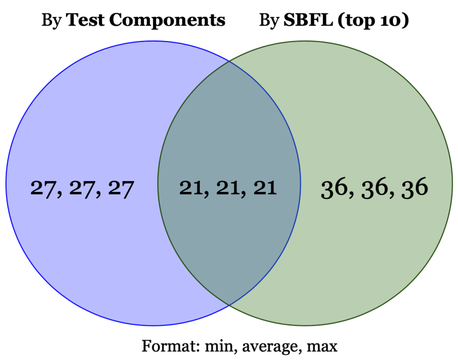


Fig. 10: Comparison between the component mapping and SBFL with voting V_D within the top 10: V_D located 9 more faults compared to the baseline. Among 57 faults localised by V_D, 36 of them are newly localised.

4) How does the test-component mapping affect the performance of the voting-based aggregation?: In Section V-A5, we extended NV_c in Section 7 to allow failing test scripts to vote for their mapped component. To evaluate the impact of utilising the test-component mappings, we compare the localisation performance of V_D varying the control parameter α. Table V presents the localisation results for $\alpha = 1, 3, 5, 7, 10$; the row with $\alpha = 0$ refers to the baseline without utilising test-component mappings. By leveraging the test-component mappings, V_D can localise up to one more fault at the top ($\alpha = 1$) and up to five more faults ($\alpha = 7$) within the top ten. However, when we compare the Venn-diagram in Fig. 11 to the one in Fig. 10, the number of faults localised by both the test-component mapping and the voting-based aggregation increases from 21 to 25 while the number of faults localised only by the voting remains almost the same. Along with the decrease in the number of faults located only by the mapping, we conclude that the voting-based aggregation successfully localise new faults by exploiting the test-component mappings.

TABLE IV: Voting results with four different voting schemes: V_D,V_{DNT},V_M,V_{DS}

Voting Scheme	Total	acc@1	acc@3	acc@5	acc@10
V_D	137	22	31	42	57
V_{DNT}	137	17	32	42	61
V_M	137	19	31	40	58
V_{DS}	137	16	27	32	41

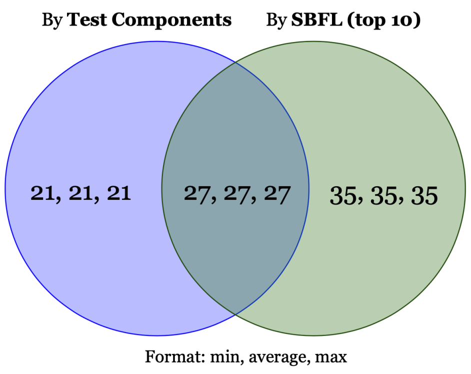


Fig. 11: Comparison between the component mapping and SBFL with the voting scheme V_D ($\alpha = 7$) within the top 10: V_D combined with the test-component mappings now can localise five more faults that only cannot be localised by using only V_D

TABLE V: Voting results using V_D with test-component mappings

α	Total	acc@1	acc@3	acc@5	acc@10
0	137	22	31	42	57
1	137	23	32	41	60
3	137	22	34	41	61
5	137	22	35	41	61
7	137	22	35	41	62
10	137	22	34	41	60

VI. Discussions

A. Improving Test Suites for Better Diagnosability

We suspect that the fundamental issue that currently limits the accuracy of localisation is the low diagnosability of the test suite, especially its low uniqueness (see Section IV) in the coverage matrix. A large number of test scripts tend to also cover a large number of files in SAP HANA2, resulting in decreased capability to distinguish them. This is partly because of the way test cases and their execution units are defined for SAP HANA2. A test script for SAP HANA2 actually contains multiple test *methods*, each of which can be also considered as an individual test case. Although the CI pipeline executes, and reports the results of test methods, the coverage is measured at the test script level, not the test methods level. Thus, the uniqueness of each test methods is lost when we perform the fault localisation. For example, let us suppose test methods f_a and f_b both belong to the same test case t. If the case f_a fails but f_b passes, or vice versa, we consider t fails when applying SBFL as we do not distinguish f_a and f_b in the coverage data.

For higher diagnosability, we need to either refine and split existing test scripts, or execute tests and collect coverage by smaller units (i.e., test methods). Refinement of test cases has been studied as test purification [32]: we may consider permanently splitting test scripts into smaller groups of test methods, or dynamically executing subsets of existing test scripts for localisation only. Collecting coverage by smaller units of execution will require significant change to the existing CI pipeline infrastructure for SAP HANA2, which we can also consider for better localisation.

B. Advanced Fault Localisation Models

While this work adopted Spectrum Based Fault Localisation for the pilot study, there are more advanced learn-to-rank approaches for fault localisation that combines multiple lower level localisation techniques using learn-to-rank machine learning models [22], [23], [14]. For future work, we will consider adopting learn-to-rank fault localisation approaches, while looking for additional features that we can extract from the CI pipeline and add in order to improve the accuracy (such as the existing mapping between test scripts and HANA components, or code and change metrics used by Sohn and Yoo [22], [23]).

C. Handling Multiple and Residual Faults

Most of the existing fault localisation work makes the single fault assumption, i.e., there is a single fault to be localised. In the context of CI pipeline, this is no longer realistic. Even if we can accurately localise the very first single fault, the second one may occur while the first fault is not patched, adding noise to the localisation of the second fault. Existing approaches towards handling multiple faults are mostly based on clustering of test cases based on their execution traces [27], [6], but mostly using benchmarks that are smaller than SAP HANA2 and not in the context of continuous integration. Future work will consider how to effectively isolate the residual faults from localisation of newly observed faults in the context of CI pipeline.

VII. Conclusion

We present a case study of automated fault localisation in the context of bug report assignment in the CI pipeline. As far as we know, this paper reports the first industrial scale application and evaluation of fault localisation. Based on statement level coverage data that are collected weekly, we compute and aggregate SBFL scores for files and components. Using 137 test executions that included at least one failing test script, we compare the result of the SBFL to the ground truth (i.e., the location of the actual fix commit). With the help of voting-based aggregation, we can localise the root cause of up to 22 failures to the exact component, and 61 failures down to 10 components, out of 137 failures studied. The results suggest that automated fault localisation can effectively aid bug report assignment in a large industrial CI pipeline.

Acknowledgment

Gabin An, Jeongju Sohn, and Shin Yoo have been supported by SAP Labs, as well as the Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7068179), and National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (Grant No. NRF-2020R1A2C1013629).

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC van Gemund. An evaluation of similarity coefficients for software fault localization. In *The proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC 2006, pages 39–46. IEEE, 2006.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[3] John Anvik. Automating bug report assignment. In *Proceedings of the 28th International Conference on Software Engineering*, pages 937–940, 2006.

[4] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight bug localization with ample. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 99–104, New York, NY, USA, 2005. ACM.

[5] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. Devops. *IEEE Software*, 33(3):94–100, 2016.

[6] Ruizhi Gao and W Eric Wong. Mseeran advanced technique for locating multiple bugs in parallel. *IEEE Transactions on Software Engineering*, 45(3):301–318, 2017.

[7] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Museum: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology*, 82:80–95, 2017.

[8] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 464–475, 2015.

[9] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 662–664, Washington, DC, USA, 2009. IEEE Computer Society.

[10] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE2005)*, pages 273–282. ACM Press, 2005.

[11] 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*, pages 273–282, 2005.

[12] Dongwon Kang, Jinhwan Jung, and Doo-Hwan Bae. Constraint-based human resource allocation in software projects. *Software: Practice and Experience*, 41(5):551–577, 2011.

[13] Tien-Duy B. Le, Ferdian Thung, and David Lo. Predicting effectiveness of ir-based bug localization techniques. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, pages 335–345, 2014.

[14] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 169–180, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, ICST 2014, pages 153–162, 2014.

[16] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering Methodology*, 20(3):11:1–11:32, August 2011.

[17] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[18] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test., Verif. Reliab.*, 25(5-7):605–628, 2015.

[19] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA 2011, pages 199–209, New York, NY, USA, 2011. ACM.

[20] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664. IEEE, 2017.

[21] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355. IEEE, 2013.

[22] Jeongju Sohn and Shin Yoo. Fluccs: Using code and change metrics to improve fault localisation. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 273–283, 2017.

[23] Jeongju Sohn and Shin Yoo. Empirical evaluation of fault localisation using code and change metrics. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[24] Jeongju Sohn and Shin Yoo. Why train-and-select when you can use them all? Ensemble model for fault localisation. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation*, GECCO 2019, pages 1408–1416, 2019.

[25] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3.2002, National Institute of Standards and Technology, 2002.

[26] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 1–11, 2015.

[27] Yabin Wang, Ruizhi Gao, Zhenyu Chen, W Eric Wong, and Bin Luo. Was: A weighted attribute-based strategy for cluster test selection. *Journal of Systems and Software*, 98:44–58, 2014.

[28] 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.

[29] W. E. Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707, August 2016.

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

[31] 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 Transactions on Software Engineering Methodology*, 22(4):31:1–31:40, October 2013.

[32] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

[33] Shin Yoo. Evolving human competitive spectra-based fault localisation techniques. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 244–258. Springer Berlin Heidelberg, 2012.

[34] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E Hassan. An empirical study on factors impacting bug fixing time. In *2012 19th Working Conference on Reverse Engineering*, pages 225–234. IEEE, 2012.