On the Use of Sequence Mining within Spectrum Based Fault Localisation

Gulsher Laghari
AnSyMo — Universiteit Antwerpen België
gulsher.laghari@uantwerpen.be

Serge Demeyer
AnSyMo — Universiteit Antwerpen België
Flanders Make — Belgium

ABSTRACT

Spectrum based fault localisation is a widely studied class of heuristics for locating faults within a software program. Unfortunately, the current state of the art ignores the inherent dependencies between the methods leading up to the fault, hence having a limited diagnostic accuracy. In this paper we present a variant of spectrum based fault localisation, which leverages series of method calls by means of sequence mining. We validate our variant (we refer to it as *sequenced spectrum analysis*) on the Defects4J benchmark, demonstrating that sequenced spectrum analysis gains a 12% points improvement against the state of the art.

KEYWORDS

Debugging; Spectrum based fault localisation; Method call sequences

1 INTRODUCTION

Software defects remain a primary concern within the software engineering community [39]. Since the source code provides the ultimate description of the behaviour of the system, it is there that software engineers search for the the root cause of a defect— the so-called *fault*— and fix it subsequently. Fault localisation is widely acknowledged to be one of the more difficult and time consuming steps while fixing defects and it is, therefore, a heavily investigated research topic [33].

In this paper, we focus on what is known as *spectrum based fault localisation* [3, 11, 30]. Spectrum based fault localisation, a lightweight automated fault localisation technique, statistically compares executions traces from both failing and passing test cases to pinpoint a faulty program element. It produces a ranked list of program elements, indicating the likelihood of a program element being at fault. Spectrum based fault localisation comprises three steps: (i) creating a *test coverage matrix*; (ii) deducing the *hit-spectrum*, and (iii) applying a *fault locator*.

Today, most variants of spectrum based fault localisation focus on step (iii), and experiment with different fault locator functions (e.g., Ochiai [2] and Naish2 [21]). Recently, however, a new branch of research investigated variations of step (ii) providing alternative ways for deducing the hit-spectrum (e.g., time spectra [36], frequent itemsets [15, 16], and method invariants [4]). This paper investigates one other alternative for deducing the hit-spectrum, namely *sequence mining*. As such, we make the following contributions.

(1) We present a variant of spectrum based fault localisation (referred to as *sequenced spectrum analysis* in the remainder of this paper) which leverages series of method calls by means of sequence mining.

(2) We use 47 known fault locators to create a suite of spectrum based fault localisation heuristics— the current state-of-the-art (referred to as *raw spectrum analysis*)— and evaluate them on real faults. This sets the baseline for the comparison.

(3) We compare sequenced spectrum analysis against the raw spectrum analysis using the Defects4J dataset [13].

(4) We use several evaluation metrics during that comparison, effectively adhering to the concerns of absolute measure [22, 29], early precision [27], and total recall [27] which implies a stringent comparison.

The remainder of this paper is organised as follows. Section 2 lists the known variants of spectrum based fault localisation for comparison, while Section 3 explains the details of the variant proposed here. Section 4 describes the set-up of the comparison, which naturally leads to Section 5 reporting the results. After a discussion on the related work in Section 6 and the threats to validity in Section 7, we come to a conclusion in Section 8.

2 BACKGROUND

Spectrum based fault localisation takes as input the faulty program and a test-suite where at least one test case exposes the defect. It produces, as the output, a ranked list of program elements where the most suspicious program elements appear at the top of the list.

There are several concerns to take into account when applying spectrum based fault localisation on a faulty program. First, it involves the decision to select a *granularity* level of a program element. The choices for granularity include from fine-grained statements to course-grained classes.

Second, the coverage of selected program element is collected by running the test-suite for the faulty program. This coverage is organised into a data structure called the *test coverage matrix*. The rows in this matrix correspond to elements under test and the columns represent the test cases. Each cell in the matrix marks whether a given element under test is executed by the test case (marked as 1) or not (marked as 0).

Third, the test coverage matrix is summarised into the *hit- spectrum*— a summarised abstract behaviour of the program. The hit-spectrum of an element under test is a tuple of four values (e_f , e_p, n_f , n_p). Where e_f and e_p are the numbers of failing and passing test cases that execute the element under test respectively and n_f and n_p are the numbers of failing and passing test cases that do not execute the element under test respectively.

Fourth, the *fault locator function* translates the hit-spectrum into suspiciousness of the element under test. This suspiciousness, which is function of the fault locator, indicates the likelihood of the element under test to be at fault. Most fault locator functions have a range in interval [0, 1]. Thus, the suspiciousness value for

an element under test may have the lowest value 0 (not suspicious at all) to 1 (highly suspicious). The underlying intuition is that an element under test which is executed more in failing tests and less in passing tests gets a higher suspiciousness and appears at the top location in the ranked list. Sorting the elements under test according to their suspiciousness in descending order produces the ranked list. We refer to this kind of fault localisation analysis as *raw spectrum analysis*.

When applying spectrum based fault localisation, there are three avenues to improve the effectiveness of the heuristic. The first is exploring the different levels of granularity of program elements. The state of the art has almost explored all possible granularity levels from *statements* [6, 11, 23, 31], *blocks* [3, 17, 19, 20], *methods* [4, 16, 30, 35], and to *classes* [9, 15]. In this paper, we select method-level granularity for four reasons. (1) In object oriented testing a method is the smallest element under test [5]. (2) Objects interact through methods by following a certain protocol on the calling sequence of its methods [32]. For complicated protocols this is a source of subtle bugs which are notoriously difficult to resolve [12]. (3) A method often provides sufficient context needed to help developers understand a bug [4]. (4) Developers expressed a slight preference for method-level granularity [14].

The second dimension is to optimise the fault locator function which was untill now the primary avenue for improvement. The efforts include applying functions used in the molecular biology domain for fault localisation [3], applying association measures from data mining for fault localisation [19], proposing fault locators based on a theoretical model [21], and evolving a completely new set of fault locators through genetic programming [38].

The third dimension, which has remained largely unexplored up until now, is to try a variation of the hit-spectrum— the input to the fault locator. Yilmaz et. al. for example adopted traces of method execution times instead of a mere count of passing and failing tests [36]. Dallmeier et. al. extracted sequence of method calls by sliding a window over execution traces of classes to identify faulty classes [9]. Similarly, we adopted itemset mining to pinpoint faulty classes [15]. Later, we explored a variant of the hit-spectrum adopting frequent itemset mining to localise faulty methods [16]. In this paper, we present another variant of the hit-spectrum adopting sequence mining to locate faulty methods.

The motivation for exploring sequence mining in the hit-spectrum is that in the traditional fault localisation, the hit spectrum only tells whether or not the method is involved in a test case. However, it ignores the inherent dependencies between the calls leading up to the fault. In particular, branch conditions, data inputs, or exceptions thrown may be the real cause for deviating behaviour of the failing test [9]. Since fault localisation has access to the complete call trace anyway, it is relatively easy to incorporate information about the call sequences itself. Consequently, we modify the hit-spectrum by adopting sequence mining, referring to this kind of fault localisation analysis as *sequenced spectrum analysis* compared to the traditional approach referred to as *raw spectrum analysis*.

The hit-spectrum in raw spectrum analysis ignores the inherent dependency relationships between the calls leading up to the fault. In sequenced spectrum analysis, we modify the hit-spectrum by incorporating series of method calls mined from the execution traces.

3 SEQUENCED SPECTRUM ANALYSIS

Here, we briefly describe the steps in our sequenced spectrum analysis. We run the test cases on a faulty program and in each test case, (1) collect the traces for each executed method of the project (Section 3.1), (2) mine the call sequences from these traces (Section 3.2), (3) calculate the hit-spectrum (Section 3.3), and finally (4) rank the methods (Section 3.4) according to their likelihood of being at fault.

3.1 Collecting the Trace

In each test case, during the execution of a method, we intercept each method call directly originating from the method and record it into the trace. The intercepted call can be a call to a project method or to the external library method. Note that we incorporate calls to the constructors, hence have knowledge about the creation of objects as well.

A trace is collected by introducing the logger functionality into the base code via AspectJ[1]. More specifically, we use method execution and method call join points. For a method execution join point, there are two advices (*before* and *after*), while there is one advice for method call join point. In the *before execution* advice, a trace is initiated for the executed method. In the *before call* advice, which picks out every call site, we collect the names of both callee and the caller method and add the called method into the trace of the caller method. Finally, in the *after execution* advice, the current trace for the method is closed. To save the memory, we assign unique integer identifier to each executed method and add the identifier to the trace instead of the name.

As a method can also execute one or more times in a test case, we separate the call traces for each execution. Thus, the complete trace for a method `m()` in a test case $\mathcal{T}$ is represented as a set $\mathcal{T}_m = \{t_1, t_2, ..., t_n\}$. Where t_i is a list of the method calls invoked directly from method `m()` during its i^{th} execution. This implies that the calling relation is not followed transitively but is terminated after one level. If method `m()` in Listing 1 executed twice in a test case $\mathcal{T}$, its trace would be $\mathcal{T}_m = \{\langle m2, m3\rangle, \langle m2, m3\rangle\}$.

Listing 1: Example method

```
1  public void m()
2    { m2(); m3(); }
```

3.2 Obtaining Call Sequences

Once the call traces are collected, we mine the sequence of method calls from the traces of a method. Normally, a method executes only once in a test case resulting into only a single trace for a method ($|\mathcal{T}_m| = 1$).

Hence, we adopt the *MARBLES algorithm* to mine the call sequences from the method call traces [8]. This algorithm mines general, parallel, and serial episodes (subsequences) from a large sequence (*a single call trace in our case*) sliding a window of fixed size. Since we are interested in an order-preserving sequence of method calls, we only use the serial episodes. From here on, we refer to these episodes simply as sequences. In this experiment, we use a window size of 8. We restrict the window size to 8 inspired by Dallmeier et. al. who found that increasing the window size beyond

[1] http://www.eclipse.org/aspectj/

8 did not increase effectiveness of fault localisation [9]. Also for window sizes greater than 8 when applied on long traces, MARBLES takes a long time to produce a result. We apply the algorithm to translate the complete trace $\mathcal{T}_m$ for a method `m()` obtained in a test case $\mathcal{T}$ into a set of call sequences $S_{\mathcal{T}_m}$. Thus, for each call trace $ti \in \mathcal{T}_m$ as input, the algorithm produces s_i a set of call sequences as output. Note that the method trace may comprise a call (or a series of calls) to a single method. In this case, MARBLES outputs an empty sequences set since a sequence must contain at least two distinct items. However, we record the single call as sequence, since it can be useful to tackle API violations like *open-close principle* [25]. The s_i is a set of unique call sequences $\mathcal{X}$. The final set of the call sequences $S_{\mathcal{T}_m}$ (Equation 1) for the method `m()` in test case $\mathcal{T}$ is the union of the set of call sequences s_i.

$$S_{\mathcal{T}_m} = \bigcup_{i=1}^{n} s_i \tag{1}$$

3.3 Calculating the Hit-Spectrum

The call sequences of a method `m()` in sequenced spectrum analysis are obtained by running the set of failing test cases (denoted as $\mathbb{T}_F$) and the set of passing test cases (denoted as $\mathbb{T}_P$). We obtain a set of call sequences $\mathcal{S}_m$ (Equation 2) for each method. The call sequences set $\mathcal{S}_m$ is the union of (i) the call sequences of a method resulting from the failing test cases ($\mathcal{S}_{\mathcal{T}_m} : \mathcal{T} \in \mathbb{T}_F$) and (ii) the call sequences resulting from the passing test cases ($\mathcal{S}_{\mathcal{T}_m} : \mathcal{T} \in \mathbb{T}_P$).

$$\mathcal{S}_m = \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_F\} \cup \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \mathcal{T} \in \mathbb{T}_P\} \tag{2}$$

The set $\mathcal{S}_m$ (Equation 2) is used to construct the test coverage matrix for a method. The hit spectrum is then calculated for each call sequence $\mathcal{X}$ in set $\mathcal{S}_m$ from the test coverage matrix.

3.4 Ranking Methods

To produce a ranked list of methods, first each call sequence gets a suspiciousness score. Then, a method gets the suspiciousness which is the maximum suspiciousness of its constituting call sequences. The details for these steps follow.

Suspiciousness per call pattern. Based on the hit-spectrum calculated from the test coverage matrix for each method, each call sequence $\mathcal{X} \in S_m$ (Equation 2) gets a suspiciousness score $Susp(\mathcal{X})$ calculated by using a fault locator.

Suspiciousness per method. Each method `m()` gets a suspiciousness $Susp(m)$ which is the suspiciousness of the call sequence $\mathcal{X}$ with the highest suspiciousness (Equation 3). We choose the maximum (instead of average) for the suspiciousness score because the technique is looking for exceptional sequences: one unique and highly suspicious sequence is more important than several unsuspicious ones. The methods with an empty call sequence get suspiciousness 0.

$$Susp(m) = \max\left(\{Susp(\mathcal{X}) \mid \mathcal{X} \in S_m\}\right) \tag{3}$$

Ranked list. Finally, a ranked list of all executed methods is produced by sorting the methods on their suspiciousness $Susp(m)$ such that methods with the highest suspiciousness appear at the top.

Table 1: Descriptive Statistics for the Projects Used in Our Experiments

Project	# Bugs	Source KLoC	Test KLoC	# Tests
Math	106	85	19	3,602
Lang	65	22	6	2,245
Time	27	28	53	4,130
Chart	26	96	50	2,205
Closure	133	90	83	7,927

4 EVALUATION

In this section, we provide the details of empirical evaluation on how far the two variants (raw spectrum analysis and sequenced spectrum analysis) can improve the fault localisation.

4.1 Dataset

For this empirical evaluation, we use real defects which have been collected by Just et. al. into a database called Defects4J[2] (a **d**atabase of **e**xisting **f**aults to **e**nable **c**ontrolled **t**esting **s**tudies for **J**ava programs) [13]. Defects4J version 1.1.0 contains defects from 6 open source java projects: Apache Commons **Math**, Apache Commons **Lang**, Joda-**Time**, JFree**Chart**, Google **Closure** Compiler, and **Mockito**. In our study, our tracing system could not be used with Mockito, hence this project was excluded from our study. The descriptive statistics of 5 projects are reported in Table 1.

The database contains meta info about each defect including the source classes modified to fix the defect, the test cases that expose the defect, and the test cases that trigger at least one of the modified classes. In this evaluation, we use 346 defects which are located inside methods or constructors.

4.2 Evaluation Metrics

Fault localisation heuristics produce a ranked list of elements under test; in the ideal case the faulty unit appears on top of the list. Several ways to evaluate such rankings have been used in the past, including relative measures in relation to project size, such as the percentage of units that need or need not be inspected to pinpoint the defect [30]. However, absolute measures are currently deemed better for comparison purposes [22, 30]. The most commonly adopted metrics are *wasted effort*, *acc@n*, and *mean average precision* [4, 27, 30, 35]. Consequently, we will use these metrics for our comparisons.

To deal with defects spread over multiple locations, we evaluate from the perspective of a *best-case debugging* scenario as argued by Pearson et. al. [23]. In such a scenario identifying one of the possible locations is good to understand and consequently repair the defect. Indeed, once the first faulty element is located it will help developers to find the remaining ones [27].

Mean Wasted Effort (MWE) – **Smaller is better.** The *mean wasted effort* is the simply the mean of the *wasted effort* in all ranked lists. The *wasted effort* is an absolute measure which indicates the number of non-faulty methods to inspect in vain before reaching the first faulty method. It is computed as follows:

$$\text{wasted effort} = m + \frac{n}{2} \tag{4}$$

Where m is the number of non-faulty methods ranked strictly higher than the faulty method; and n is the number of non-faulty methods

[2] http://defects4j.org

with equal rank in the ranked list to the faulty method. This deals with ties in the ranked list.

acc@n – **Higher is better.** This is the count of all the faults successfully localised in top-n positions in the ranked list. Inspired by Le et al. [4], we also choose n ∈ {1, 3, 5}, thus effectively creating three variants of the *acc@n* namely *acc@1*, *acc@3*, and *acc@5*. It is not uncommon for two methods in a ranked list sharing the same suspiciousness scores. Hence, while computing the *acc@n*, we break the ties randomly.

Mean Average Precision (MAP) – **Higher is better.** The *mean average precision* has traditionally been used in information retrieval to evaluate the ranked lists and is also adopted for studying fault localisation. It takes all faulty elements into account and emphasises recall over precision. Thus, it is suitable in scenarios where developers search deep in the ranked list to find more relevant faulty elements [27]. The *mean average precision* is the mean of *average precision* in all ranked lists. The *average precision* in a single ranked list is calculated as following:

$$\text{average precision} = \sum_{i=1}^{M} \frac{P(i) \times pos(i)}{number\ of\ faulty\ methods} \tag{5}$$

Where: i indicates the position of a method in the ranked list; M is size of the ranked list (number of methods ranked); $pos(i)$ is a boolean indicating whether or not the method at i^{th} position in the ranked list is faulty; $P(i)$ is the precision at i^{th} position in the ranked list, computed as $P(i) = \frac{\#\ faulty\ methods\ in\ top\ i}{i}$.

Our use of several metrics together evaluates fault localisation in several contexts. *Wasted effort* does not normalise the rank of faulty methods with respect to total number of methods in the program. Thus, it is inline with recommendations of Parnin et. al. [22] that for the fault localisation to be useful for developers the aim should be to improve absolute rank rather than percentage rank. In their study, they found that developers switched to other means of debugging when they did not find faulty statements within the first few top positions in the ranked list. The same concerns are also addressed by *acc@n*. However, when developers want to search deep in the ranked list to find more relevant faulty methods, *mean average precision* is suitable in this context [27].

4.3 Experimental Protocol

To compare the two variants, we use faulty version of each project. Then, we run each spectrum based fault localisation for all *relevant* test cases, i.e. all test classes which trigger at least one of the source classes modified to fix the fault as recorded in the Defects4J dataset. As such, we obtain ranked lists for each of the 346 defects in the dataset altogether and also organised by project (see Table 1). We first compare the 47 fault locators within each spectrum analysis before going into a more detailed analysis. We use five different metrics for this comparison: Mean Average Precision (MAP), Mean Wasted Effort (MWE) and acc@1, acc@3, and acc@5.

Best Performing Fault Locator. While comparing raw spectrum analysis against sequenced spectrum analysis we use the best performing fault locator for each case. To identify the best performing fault locator, we first rank all 47 fault locators on each of the five evaluation metrics and then compute the mean of the ranks. Thus, the fault locator with the lowest mean rank— performing best in all evaluation metrics— is selected as the best performing one.

Significance Tests. We perform statistical tests of significance for raw spectrum analysis and sequenced spectrum analysis on the evaluation metrics *wasted effort* (*WE*) and *average precision* (*AP*). Since we have a matched pair design and we compare whether one variant is better than its counterpart, we choose the Wilcoxon signed rank test and run as paired one-tailed test. We favour the non-parametric Wilcoxon signed rank test over parametric t-test owing to small sample sizes and non-normal distribution of scores for both *wasted effort* as well as *average precision*. As is common practice in software engineering research, we set the significance level α of 0.05 (there is 5% risk of concluding that the two distributions are different when in fact they are not).

Research Questions. In this evaluation, we address following research questions.

RQ1. **What is the baseline performance of raw spectrum analysis?**
Motivation. We establish the best performing fault locator and obtain the rankings for the 346 faults in the Defects4J. This sets the baseline against which we compare.

RQ2. **How much can sequenced spectrum analysis improve upon raw spectrum analysis?**
Motivation. We establish the best performing fault locator for sequenced spectrum analysis and obtain the rankings for the same 346 faults. We compare these rankings against the baseline obtained in RQ1.

RQ3. **Are there project specific differences between the rankings ?**
Motivation. Inspired by the work of Zeller et. al. [40], we investigate whether the results obtained for the whole Defects4J data set apply to the various projects within that dataset. This is to assess the robustness of our findings.

5 RESULTS

In this section, we discuss the answers to three research questions introduced in Section 4.

RQ1 – To answer this question, we apply raw spectrum analysis with 47 known fault locators on all defects together, thus aggregating the results over all projects in the dataset. This allows for a sufficiently rigorous analysis of the current state of the art and as such establishes the baseline performance of raw spectrum analysis. As mentioned in the protocol, we rank the fault locators from the top with the best performance to the bottom with the worst.

Table 2 lists the fault locators along with their scores for five evaluation metrics sorted on their rank (rightmost column entitled R). Fault locators with the same rank are highlighted in the same background colour. We observe in the table that GP13 and Naish2 have good scores for acc@1, acc@3, acc@5, and a better mean average precision than M2 and Goodman. M2 and Goodman, on the other hand, are slightly better in terms of mean wasted effort. Studying the performance on each of the evaluation metrics, the value of 63 for acc@1 tells us that for 63 out of 346 (18%) raw spectrum analysis has an exact hit: the method containing the fault is the first one in the ranking. Similarly, acc@3 (120 out of 346 is 35%) and acc@5 (142 out of 346 is 41%) demonstrates that the fault locators perform reasonably well in many cases. The *mean*

Table 2: Establish the baseline performance for raw spectrum analysis over the 346 defects in the dataset.

MAP = Mean Average Precision, @1 = acc@1, @3 = acc@3, @5 = acc@5, MWE = Mean Wasted Effort, R = Rank.

Fault locator	@1	@3	@5	MAP	MWE	R
GP13 [38]	63	120	142	0.2780349	96.73	1
Naish2 [21]	63	120	142	0.2776756	96.64	1
M2 [21]	62	118	141	0.2753030	96.32	2
Goodman [21]	61	116	138	0.2695181	16.68	3
Ample2 [21]	64	120	140	0.2764775	101.24	3
T* [30]	62	119	139	0.2744910	96.37	4
Zoltar [28]	61	118	138	0.2735461	96.14	5
Kulczynski2 [21]	61	116	137	0.2718006	96.56	6
Ochiai [3]	61	116	138	0.2693963	98.02	7
Jaccard [7]	61	116	138	0.2695181	104.20	8
Dice [21]	61	116	138	0.2695181	104.20	8
Kulczynski1 [21]	61	116	138	0.2695181	104.20	8
Anderberg [21]	61	116	138	0.2695181	104.20	8
Sørensen -Dice [21]	61	116	138	0.2695181	104.20	8
GP19 [38]	62	118	139	0.2729868	125.20	9
Arithmetic Mean [21]	61	118	138	0.2677694	102.93	9
Cohen [21]	61	118	138	0.2678460	105.66	10
Harmonic Mean [21]	60	115	135	0.2637967	82.19	10
Geometric Mean [21]	60	115	135	0.2620774	83.72	11
Fleiss [21]	58	107	123	0.2471480	36.29	12
Scott [21]	57	107	125	0.2446414	37.93	13
CBIInc [21]	60	118	135	0.2669550	107.01	13
Barinel [1]	60	118	135	0.2669550	107.11	14
Tarantula [11]	60	118	135	0.2668479	107.03	14
CBISqrt [21]	62	115	136	0.2657316	125.09	15
Rogot2 [21]	60	115	136	0.2642806	139.41	16
Ochiai2 [21]	60	115	136	0.2634117	130.00	17
Wong3′ [21]	42	72	86	0.1785421	22.64	18
Wong3 [34]	42	72	86	0.1785421	22.64	18
Wong2 [34]	41	66	77	0.1582824	16.04	19
Rogot1 [21]	57	107	125	0.2446351	338.34	20
Ample [3]	54	93	116	0.2297198	214.02	21
CBILog [21]	15	28	36	0.0757946	33.22	22
Overlap [21]	41	77	100	0.1857891	178.36	23
Russell & Rao [21]	38	73	97	0.1802516	177.87	24
Wong1 [34]	38	73	97	0.1802516	177.87	24
Binary [21]	37	69	92	0.1714134	188.00	25
Hamann [21]	41	66	77	0.1582824	367.43	26
Hamming etc. [21]	41	66	77	0.1582824	437.74	27
Rogers & Tanimoto [21]	41	66	77	0.1582824	437.74	27
GP02 [38]	23	48	63	0.1268758	277.10	27
M1 [21]	41	66	77	0.1582824	437.74	27
Simple Matching [21]	41	66	77	0.1582824	437.74	27
Sokal [21]	41	66	77	0.1582824	437.74	27
GP03 [38]	12	23	32	0.0656960	342.97	28
Euclid [21]	7	14	27	0.0521883	428.46	29
Naish1 [21]	1	3	7	0.0200899	419.36	30

wasted effort (MWE), however, reveals that for many cases the fault localisation is unsatisfactory: a MWE of 96 implies that on average 96 methods need to be inspected before one arrives at the correct location. This suggests a long tail distribution, where for many cases the first faulty method is ranked quite low. The value for *mean average precision* (MAP) reinforces the problem: a low value of 0.27 implies that the relative location of other faulty methods (in cases where fault expands to multiple methods) is also quite low.

Table 3: Performance improvement for sequenced spectrum analysis over the 346 defects in the dataset.

MAP = Mean Average Precision, @1 = acc@1, @3 = acc@3, @5 = acc@5, MWE = Mean Wasted Effort, R = Rank.

Fault locator	@1	@3	@5	MAP	MWE	R
Ample2	103	159	191	0.3925699	25.88	1
Fleiss	103	157	191	0.3874628	16.01	2
T*	102	160	190	0.3936098	31.18	3
M2	102	160	189	0.3933028	31.10	4
Arithmetic Mean	103	157	189	0.3875244	26.67	5
Goodman	101	157	190	0.3854730	9.83	6
Naish2	101	159	187	0.3918473	29.12	7
Geometric Mean	102	156	188	0.3866599	24.31	8
Kulczynski2	101	157	189	0.3893806	30.53	9
Ochiai	101	158	191	0.3883824	31.42	10
GP19	103	159	190	0.3929647	34.38	10
GP13	101	159	188	0.3927791	33.46	11
Harmonic Mean	102	154	186	0.3859800	23.83	12
Scott	101	156	189	0.3821700	16.73	12
Cohen	101	157	189	0.3851854	26.74	13
Jaccard	101	157	190	0.3866061	32.17	14
Dice	101	157	190	0.3866061	32.17	14
Anderberg	101	157	190	0.3866061	32.17	14
Sørensen -Dice	101	157	190	0.3866061	32.17	14
Ochiai2	101	157	192	0.3863531	35.48	15
Kulczynski1	101	154	187	0.3848006	32.52	16
Rogot2	102	154	186	0.3860920	37.45	17
Wong3′	89	139	167	0.3391367	12.15	18
CBIInc	96	147	182	0.3738493	28.32	18
Wong3	89	139	167	0.3391367	12.15	18
CBISqrt	99	154	189	0.3817832	34.71	19
Rogot1	101	156	189	0.3820858	61.86	20
Wong2	87	132	157	0.3222868	9.79	21
Zoltar	96	153	183	0.3739687	32.60	22
Tarantula	96	147	182	0.3750226	34.01	23
Barinel	96	147	182	0.3750091	34.03	24
Ample	94	146	176	0.3612457	44.78	25
CBILog	38	63	82	0.1650905	8.73	26
Hamming etc.	87	132	157	0.3222977	91.93	27
Rogers & Tanimoto	87	132	157	0.3222977	91.93	27
Hamann	87	132	157	0.3222868	83.37	27
Simple Matching	87	132	157	0.3222977	91.93	27
Sokal	87	132	157	0.3222977	91.93	27
M1	87	130	155	0.3211702	92.57	28
Russell & Rao	51	96	118	0.2328541	113.18	29
Wong1	51	96	118	0.2328541	113.18	29
Binary	45	88	109	0.2098345	125.15	30
Overlap	43	80	104	0.2018613	114.95	31
GP02	31	69	88	0.1702866	176.30	32
GP03	31	69	89	0.1632317	175.70	32
Naish1	16	28	45	0.0900345	142.08	33
Euclid	8	18	30	0.0681149	268.55	34

As baseline performance, we deduce that 18% of faults correspond with an exact hit (acc@1) while for many faults the heuristic performs reasonably well (acc@3 for 35% of the faults; acc@5 for 41% of the faults). However, the mean wasted effort is 96.73 which implies that for many cases the fault localisation is unsatisfactory and suggests a long tail distribution.

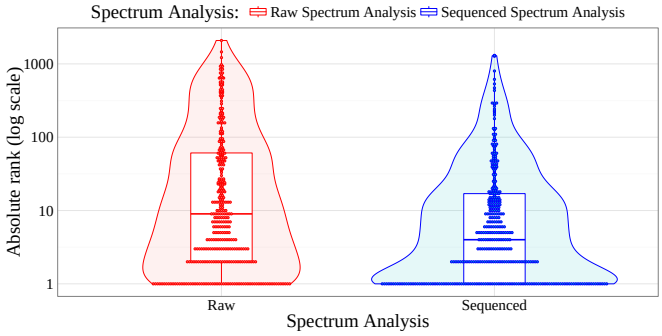

Figure 1: Comparison of the distribution of absolute rankings for faulty methods.

RQ2 – To answer this question, we apply sequenced spectrum analysis over the same dataset using the same protocol. Thus, we use the fault locators on all 346 defects and rank them to identify the best performing one. This allows for a fair comparison between raw spectrum analysis and sequenced spectrum analysis in the sense that we choose the optimal configuration for both of them. Table 3 lists the fault locators along with their scores for five evaluation metrics sorted according to their rank; highlighting fault locators with the same rank in the same background colour.

Here, we see that Ample2 is the best performing fault locator with Fleiss, T*, M2, and Arithmetic Mean as close seconds. However, the scores of Ample2 with sequenced spectrum analysis are better for all evaluation metrics compared to the raw spectrum analysis. The value for acc@1 is 103, thus for 103 out of 346 (30%) faults sequenced spectrum analysis has an exact hit — an absolute 12% improvement (30% vs 18%). Similar improvements can be seen for acc@3 (159 out of 346 is 46% compared to 35%) and acc@5 (191 out of 346 is 55% compared to 41%). Also, the mean wasted effort has reduced from 96.73 to 25.88, thus on average the fault is now located on the 25th position in the ranking. The mean average precision has risen from 0.27 to 0.39 implying that besides the location of the first faulty methods, the location of other faulty methods has also improved— the faulty methods are ranked higher in the list .

This suggests that the distribution of the rankings is better for sequenced spectrum analysis, which is confirmed in Figure 1. In these violin plots we juxtapose the distributions of absolute ranks (i.e. the location of the first faulty method) for both variants. There are some interesting observations in these plots. First, the density curve in the plot for sequenced spectrum analysis is wide for lower ranks and quickly narrows for higher ranks, indicating that most of the faulty methods are located on top of the ranked list. The density curve for raw spectrum analysis on the other hand narrows slowly indicating that many faulty methods are also located deeper in the ranked list — the ranks are spread. Second, the median in the box plot for sequenced spectrum analysis shows that for 50% of the faults the faulty methods are located within location 4 in the ranked lists, whereas for raw spectrum analysis this median is at 9 implying the faulty methods are located more deeply. Finally, the third quartile for sequenced spectrum analysis is nearly same as the median for raw spectrum analysis— the highest location where half of the faults are ranked in raw spectrum analysis, with sequenced spectrum analysis about 75% of the faults are located at the same location.

Table 4: Project specific comparison of sequenced spectrum analysis (SS) versus raw spectrum analysis (RS).

P = Project, SA = Spectrum Analysis (SS vs RS), Flt. Lctr. = Fault Locator, @1 = acc@1, @3 = acc@3, @5 = acc@5, MAP = Mean Average Precision, MWE = Mean Wasted Effort.

P	SA	Flt. Lctr.	@1	@3	@5	MAP	MWE
Closure	SS	Fleiss	17	37	47	0.2236320	30.61
	RS	GP13	7	15	20	0.1085065	222.89
Math	SS	Goodman	33	54	69	0.4458375	4.79
	RS	Goodman	21	45	51	0.3349293	7.92
Lang	SS	Geometric Mean	41	50	52	0.7294623	1.167
	RS	GP13	21	43	49	0.5238196	3.78
Time	SS	Ample2	6	10	15	0.2938999	16.15
	RS	GP13	5	8	9	0.2201962	39.38
Chart	SS	CBISqrt	9	14	16	0.4632694	13.2
	RS	Tarantula	10	15	16	0.4986104	27.16

Table 5: Significance tests for sequenced spectrum analysis vs. raw spectrum analysis.

AP = Average Precision, WE = Wasted Effort.

Project	Comparison	p-value (AP)	p-value (WE)
Closure	SS > RS	**7.849e-10**	**< 2.2e-16**
Math	SS > RS	**6.983e-05**	**9.915e-07**
Lang	SS > RS	**2.87e-05**	**2.096e-05**
Time	SS > RS	**0.01442**	**0.0009032**
Chart	SS > RS	—	0.2764
	RS > SS	0.3638	—

Significance tests for sequenced spectrum analysis versus raw spectrum analysis on both average precision and wasted effort metrics have p-values < **2.2e-16**, show that sequenced spectrum analysis is not only better, but that it is *significantly* better in the statistics sense of the word.

When compared to raw spectrum analysis, sequenced spectrum analysis gains 12% improvement for exact hit (acc@1) and reduces the average wasted effort from 96.73 to 25.88. The distribution of the fault locations is better which results in statistically significant improvements.

RQ3 – While answering the previous questions, we generalised the comparison on all the defects together irrespective of the project. However, as noted by Zeller et. al. there are project-specific variations that might provide valuable insights [40]. Therefore, we compare the two spectrum analyses on defects for each individual project. As done in previous subsections, we first select the best performing fault locator for each project and for each variant. Due to space limitations, we omit the results for the selection of the best performing fault locator and immediately move towards the actual comparison in Table 4. This table lists the project specific scores for the five evaluation metrics for the best performing fault locator for that project. Table 5 also lists the project specific comparison of p-values for both average precision and wasted effort.

The first interesting observation to make concerns the best performing fault locators: they vary a lot across projects. Naish2 (the

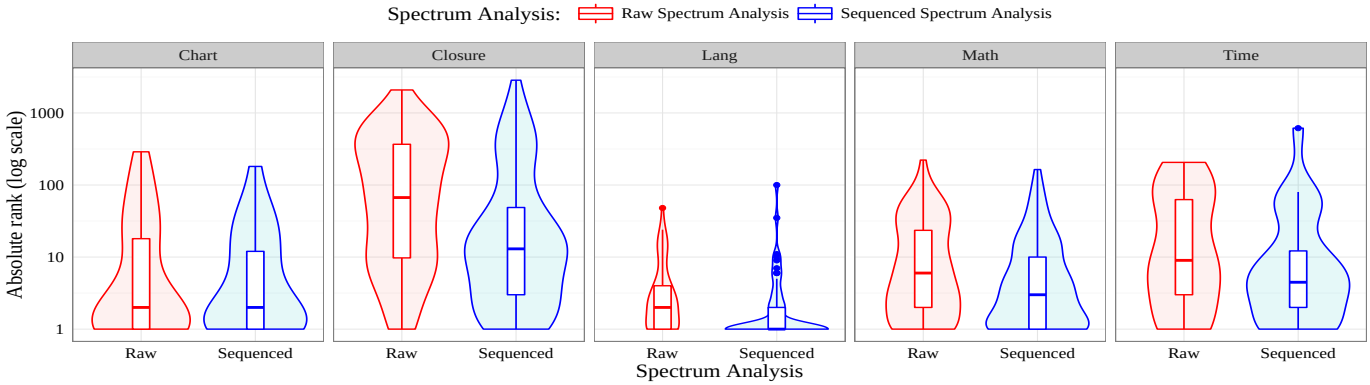

Figure 2: Distributions of absolute ranks of faulty methods for both spectrum analyses for each project.

best performing fault locator for raw spectrum analysis when applied on all projects together) is never the best performing project-specific fault locator. And Ample2 (the best performing fault locator for sequenced spectrum analysis when applied on all projects together) only appears as the best for the `Time` project. Thus, a new set of best performing fault locators has emerged for both raw spectrum analysis and sequenced spectrum analysis on project by project basis, illustrating that great care is needed when configuring such tools.

The second interesting observation in Table 4 is that there is positive change for the values of acc@1 for both spectrum analyses. With a new set of best performing fault locators, overall exact hit (acc@1) for sequenced spectrum analysis has now increased from 103 to 106, while for raw spectrum analysis it has increased from 63 to 64.

Next, we see that —with the exception of project `Chart`— sequenced spectrum analysis outperforms raw spectrum analysis. Moreover, the p-values in Table 5 confirm that sequenced spectrum analysis is statistically significantly better for these four projects, however the p-value for average precision in project `Time` is insignificant. A deeper analysis of the project `Chart` reveals that sequenced spectrum analysis is better for mean wasted effort while raw spectrum analysis is better for mean average precision and acc@1. However, as seen in Table 5 the better score for mean wasted effort with sequenced spectrum analysis is statistically significant while the better score for mean average precision with raw spectrum analysis is statistically insignificant.

We again turn to violin plots to explore these differences in more detail. Figure 2 provides distributions of absolute ranks of first faulty method in the ranked lists for both spectrum analyses for each project. We readily observe that shapes of the two distributions for the project `Chart` are nearly the same, suggesting that sequenced spectrum analysis slightly improves upon raw spectrum analysis. However, sequenced spectrum analysis improves upon raw spectrum analysis for the remaining four projects— with significant improvement for the project `Lang`. Yet, we observe that the distribution of absolute ranks with sequenced spectrum analysis for project `Time` has some outliers.

On project by project basis, sequenced spectrum analysis performs better than raw spectrum analysis for four projects. For the fifth project the results are, for practical purposes, the same. Moreover, the best performing fault locator varies a lot across the projects.

6 RELATED WORK

The Tarantula tool provided the foundation for research on spectrum based fault localisation [11]. Afterwards, several researchers made attempts to increase the effectiveness of spectrum based fault localisation, including work on (a) finding the optimal *fault locators*, (b) changing the *spectrum analysis*, (c) using state-of-the-art information retrieval techniques to *learn to rank*, and (d) *test-suite reduction and diagnosability*.

Fault locators. Abreu et. al. introduced Ochiai, used in the molecular biology domain, into spectrum based fault localisation and demonstrated better performance [3]. Steimann et. al. defined and evaluated T* (a variant of Tarantula) and there as well demonstrated better performance [30]. Lucia et. al. applied 20 well-known association measures from data mining on fault localisation and concluded that 10 out of 20 association measures were comparable to Tarantula and Ochiai [19]. Naish et. al. proposed a couple of fault locators through a theoretical model and established that they performed better than existing ones [21]. Later studies confirmed that one (Naish2) is among the best performing fault locators [4, 23], which is corroborated in this comparison. Yoo evolved an entirely different set of fault locators (GP01 ...GP30) that performed better than existing ones [38]. Work by Le et. al. finds that GP13 and GP19 perform better [4].

Hit-Spectra. Yilmaz et. al. proposed time-spectrum as a variation for spectrum analysis [36]. Instead of coverage of methods, time-spectrum uses traces of method execution times collected from both passing and failing tests. The potential causes of faults are identified as deviations of failing tests from behaviour models created from time spectra collected in passing test runs. Dallmeier et. al. extracted sequence of method calls by sliding a window of fixed size over execution traces of classes to identify faulty classes [9]. Likewise, Laghari et. al. pinpoint faulty classes but adopting itemset mining [15]. However, in our approach, we use sequence mining

to mine the sequences from call traces of methods to locate faulty methods and not classes.

Learning to rank. Xuan et. al. proposed MULTRIC, a learning-based approach which combines multiple ranking metrics to learn and then rank [35]. They demonstrated on seeded faults that MULTRIC improved upon existing fault locators. Similarly, Le et. al. [4] propose *Savant*, a learning to rank approach which exploited inferred likely method invariants mined from passing and failing test cases. They find that *Savant* is more effective than state of the art on real faults.

Specification mining. Runtime traces have been used to learn API specifications such as legal method call sequences. These specifications are used for purposes including documentation, learning the APIs, and also for bug detection. OCD learns and enforces temporal specifications over method call sequence [10]. The algorithm uses a predefined template which specifies a sequence of only two method calls and operates over a finite window on the trace. The tool is reported to have detected anomalies as violations of inferred sequences in Eclipse and Ant, though the anomalies did not result in program crashes. Pradel and Gross infer specification for Java standard library from method traces. They use method calls as object collaborations to infer API specifications as finite state machines which model the legal method call sequence [24]. JMiner traces Java programs to generate parametric specifications. The specifications produced with JMiner are reported to have detected a few bugs in open source Java programs [18]. Similarly, we mine call sequences for methods from both passing and failing tests and statistically compare these sequences to pinpoint faulty methods.

7 THREATS TO VALIDITY

As with all empirical research, we identify those factors that may jeopardise the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see [26, 37]), we organise them into four categories.

Construct validity. In this research, we compare sequenced spectrum analysis against raw spectrum analysis. To reduce the risk on construct validity, we evaluated with five different metrics assessing different perspectives on what is deemed better. The use of an absolute metric (wasted effort) and also acc@n alleviates concerns on relative measures [22]. While the evaluation of fault localisation on mean average precision has implication for developers who search deep in the ranked list to find more relevant faulty methods and for automated fault repair techniques.

Internal validity. When selecting the best fault locator which performs better in all five evaluation metrics, we use simplified method of first ranking the fault locators on individual metrics, then computing the mean of their ranks, and finally rank them on their mean rank. This may not be the best solution but we ensured that it was better than simple aggregation method of summation of metric scores.

External validity. We use several evaluation metrics together which implies a stringent comparison. Evaluation on a single metric alone may result in a different interpretations. A notable example in this paper is comparison on project Chart (see Table 4). If only evaluated on *Mean Wasted Effort*, the sequenced spectrum analysis is better than raw spectrum analysis. However, when comparing on several metrics together the result is different. This observation signals that the evaluation metric used to evaluate the fault localisation has an effect on its accuracy. Thus, it is also unwise to generalise the findings, but instead metric-specific insights should be explored.

Reliability. All the tools involved in this case study (i.e. creating the traces, calculating the ranked lists etc.) have been implemented and tested for three years by the first author. Moreover, for sequenced spectrum analysis we used the MARBLES [8] algorithm which has been already tested by the creators of MARBLES. However, lurking faults in any of the tools may affect the precise rankings.

8 CONCLUSION

In this paper, we presented sequenced spectrum analysis— a class of spectrum based fault localisation which modifies the hit-spectrum by incorporating series of method calls mined from the execution traces. To compare sequenced spectrum analysis against the state of the art, we created a suite of fault localisation heuristics with 47 known fault locators and evaluated them to establish a baseline with the best performing one. Then, we compared sequenced spectrum analysis against raw spectrum analysis and conclude that sequenced spectrum analysis is better than raw spectrum analysis, regardless of whether we evaluate for the whole dataset or whether we evaluate on a project specific basis.

During this comparison, we also learned that the best performing fault locator varies quite a lot depending on the project, the variant of spectrum analyses (raw spectrum analysis and sequenced spectrum analysis), and even the experimental setting (all defects, defects per project basis). Finally, the evaluation metrics do also play a role: we observed some cases where the best performing fault locator varies with the evaluation metric used (Mean Average Precision, Mean Wasted Effort, acc@n).

These observations have few important consequences for future research in fault localisation. First, choosing the best fault locator is highly context specific, depending on both the project and the experimental set-up, therefore these factors need to be considered before generalising the conclusions. Second, the evaluation metric used to assess the performance of fault localisation does also matter when drawing the conclusions.

Acknowledgments. Thanks to Boris Cule for helping with MARBLES algorithm. This work is sponsored by (a) the Higher Education Commission of Pakistan under a project titled “Strengthening of University of Sindh (Faculty Development Program)"; (b) Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 88–99. https://doi.org/10.1109/ASE.2009.25

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *Journal of Systems and Software* 82, 11 (Nov. 2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION '07)*. IEEE Computer Society, Washington, DC, USA, 89–98. http://dl.acm.org/citation.cfm?id=1308173.1308264

[4] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049

[5] Robert V. Binder. 1999. *Testing Object-oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 378–381. https://doi.org/10.1145/2351676.2351752

[7] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*. IEEE Computer Society, Washington, DC, USA, 595–604. http://dl.acm.org/citation.cfm?id=647883.738238

[8] Boris Cule, Nikolaj Tatti, and Bart Goethals. 2014. Marbles: Mining association rules buried in long event sequences. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 7, 2 (2014), 93–110.

[9] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight Defect Localization for Java. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*. Springer-Verlag, Berlin, Heidelberg, 528–550. https://doi.org/10.1007/11531142_23

[10] Mark Gabel and Zhendong Su. 2010. Online Inference and Enforcement of Temporal Properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 15–24. https://doi.org/10.1145/1806799.1806806

[11] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[12] Jonathan Aldrich Joshua Sushine, James D. Herbsleb. 2015. Searching the State Space: A Qualitative Study of API Protocol Usability. In *Proceedings of the International Conference on Program Comprehension (ICPC)*.

[13] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[14] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 165–176. https://doi.org/10.1145/2931037.2931051

[15] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. 2015. Localising Faults in Test Execution Traces. In *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2804360.2804361

[16] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. 2016. Fine-tuning Spectrum Based Fault Localisation with Frequent Method Item Sets. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 274–285. https://doi.org/10.1145/2970276.2970308

[17] Tien-Duy B. Le, David Lo, and Ferdian Thung. 2015. Should I Follow This Fault Localization Tool's Output? *Empirical Softw. Engg.* 20, 5 (Oct. 2015), 1237–1274. https://doi.org/10.1007/s10664-014-9349-1

[18] Choonghwan Lee, Feng Chen, and Grigore Roşu. 2011. Mining Parametric Specifications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1985793.1985874

[19] Lucia, D. Lo, Lingxiao Jiang, and A. Budi. 2010. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. 1–10. https://doi.org/10.1109/ICSM.2010.5609542

[20] Lucia, David Lo, and Xin Xia. 2014. Fusion Fault Localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 127–138. https://doi.org/10.1145/2642937.2642983

[21] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (Aug. 2011), 32 pages. https://doi.org/10.1145/2000791.2000795

[22] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445

[23] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *ICSE 2017, Proceedings of the 39th International Conference on Software Engineering*. Buenos Aires, Argentina.

[24] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 371–382. https://doi.org/10.1109/ASE.2009.60

[25] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Trans. Softw. Eng.* 39, 5 (May 2013), 613–637. https://doi.org/10.1109/TSE.2012.63

[26] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engineering* 14, 2 (2009), 131–164.

[27] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry. 2013. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. 345–355. https://doi.org/10.1109/ASE.2013.6693093

[28] Alberto González Sánchez. 2007. *Automatic Error Detection Techniques Based on Dynamic Invariants*. Master's thesis. Delft University of Technology, the Netherlands. http://swerl.tudelft.nl/twiki/pub/Main/AlbertoGonzalezSanchez/thesis_gonzalez.pdf

[29] F. Steimann and M. Frenkel. 2012. Improving Coverage-Based Localization of Multiple Faults Using Algorithms from Integer Linear Programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. 121–130. https://doi.org/10.1109/ISSRE.2012.28

[30] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage- based Fault Locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. ACM, New York, NY, USA, 314–324. https://doi.org/10.1145/2483760.2483767

[31] Jingxuan Tu, Lin Chen, Yuming Zhou, Jianjun Zhao, and Baowen Xu. 2012. Leveraging Method Call Anomalies to Improve the Effectiveness of Spectrum-Based Fault Localization Techniques for Object-Oriented Programs. In *Proceedings of the 2012 12th International Conference on Quality Software (QSIC '12)*. IEEE Computer Society, Washington, DC, USA, 1–8. https://doi.org/10.1109/QSIC.2012.30

[32] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1287624.1287632

[33] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (Aug. 2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[34] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization Using Code Coverage. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '07)*. IEEE Computer Society, Washington, DC, USA, 449–456. https://doi.org/10.1109/COMPSAC.2007.109

[35] J. Xuan and M. Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. 191–200. https://doi.org/10.1109/ICSME.2014.41

[36] Cemal Yilmaz, Amit Paradkar, and Clay Williams. 2008. Time will tell: fault localization using time spectra. In *In ICSE Conference Proceedings*. 81–90.

[37] Robert K. Yin. 2002. *Case Study Research: Design and Methods, 3 edition*. Sage Publications.

[38] Shin Yoo. 2012. Evolving Human Competitive Spectra-based Fault Localisation Techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering (SSBSE'12)*. Springer-Verlag, Berlin, Heidelberg, 244–258. https://doi.org/10.1007/978-3-642-33119-0_18

[39] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann.

[40] Andreas Zeller, Thomas Zimmermann, and Christian Bird. 2011. Failure is a Four-letter Word: A Parody in Empirical Research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (Promise '11)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/2020390.2020395