Fine-Tuning Spectrum Based Fault Localisation with Frequent Method Item Sets

Gulsher Laghari, Alessandro Murgia, Serge Demeyer
ANSYMO — Universiteit Antwerpen, Belgium
{gulsher.laghari, alessandro.murgia, serge.demeyer}@uantwerpen.be

ABSTRACT

Continuous integration is a best practice adopted in modern software development teams to identify potential faults immediately upon project build. Once a fault is detected it must be repaired immediately, hence continuous integration provides an ideal testbed for experimenting with the state of the art in fault localisation. In this paper we propose a variant of what is known as spectrum based fault localisation, which leverages patterns of method calls by means of frequent itemset mining. We compare our variant (we refer to it as patterned spectrum analysis) against the state of the art and demonstrate on 351 real faults drawn from five representative open source java projects that patterned spectrum analysis is more effective in localising the fault.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

Automated developer tests; Continuous integration; Spectrum based fault localisation; Statistical debugging

1. INTRODUCTION

Continuous integration is an important and essential phase in a modern release engineering pipeline [3]. The quintessential principle of continuous integration declares that software engineers should frequently merge their code with the project's codebase [14, 15]. This practice is helpful to ensure that the codebase remains stable and developers can continue further development, essentially reducing the risk of arriving in *integration hell* [16]. Indeed, during each integration step, a continuous integration server builds the entire project, using a fully automated process involving compilation, unit tests, integration tests, code analysis, security checks, ... When one of these steps fails, the build is said to be *broken*; development can then only proceed when the fault is repaired [31, 40].

The safety net on automated tests, encourages software engineers to write lots of tests — several reports indicate that there is more test code than application code [44, 11, 51]. Moreover, executing all these tests easily takes several hours [36]. Hence, it should come as no surprise that developers defer the full test to the continuous integration server instead of running them in the IDE before launching the build [5]. Occasionally, changes in the code introduce regression faults, causing some of the previously passing test cases to fail [43]. Repairing a regression fault seems easy: the most recent commits should contain the root cause. In reality it is seldom that easy [40]. There is always the possibility of lurking faults, i.e. faults in a piece of code which are revealed via changes in other parts of the code [46]. For truly complex systems with multiple branches and staged testing, faults will reveal themselves later in the life-cycle [17, 22].

Luckily, the state-of-the-art in software testing research provides a potential answer via *spectrum based fault localisation*. These heuristics compare execution traces of failing and passing test runs to produce a ranked list of program elements likely to be at fault. In this paper, we present a variant which leverages patterns of method calls by means of frequent itemset mining. As such, the heuristic is optimised for localising faults revealed by integration tests, hence ideally suited for serving in a continuous integration context.

In this paper, we make the following contributions.

1. We propose a variant of spectrum based fault localisation (referred to as patterned spectrum analysis in the remainder of this paper) which leverages patterns of method calls by means of frequent itemset mining.
2. We compare patterned spectrum analysis against the current state-of-the-art (referred to as raw spectrum analysis in the remainder of this paper) using the Defects4J dataset [21].
3. The comparison is inspired by a realistic fault localisation scenario in the context of continuous integration, drawn from a series of discussions with practitioners.

The remainder of this paper is organised as follows. Section 2 lists the current state-of-the-art. Section 3 presents a motivating example, followed by Section 4 explaining the inner details of our variant. Section 5 describes the case study set-up, which naturally leads to Section 6 reporting the results of the case study. After a discussion of potential improvements in Section 7, and the threats to validity in Section 8, we come to a conclusion in Section 9.

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970308

2. STATE OF THE ART

This section provides an overview of the current state-of-the-art in spectrum based fault localisation. In particular, we sketch the two dimensions for the variants that have been investigated: either the granularity (statement — block — method — class) or the fault locator function (Tarantula, Ochiai, T*, and Naish2). We also explain what is commonly used when evaluating the heuristics: the evaluation metric (*Wasted Effort*) and the available benchmarks and datasets. Finally, we list some common applications of fault localisation, which heavily influences the way people assess the effectiveness in the past research.

Automated Fault Localisation. To help developers quickly locate the faults, there exist two broad categories of automated fault localisation heuristics: (1) information retrieval based fault localisation [54, 38, 35, 25], and (2) spectrum based fault localisation [20, 19, 1, 41, 42]. Both of these categories produce a ranked list of program elements, indicating the likelihood of a program element causing the fault. While the former uses bug reports and source code files for analysis, the later uses program traces generated by executing failing and passing test cases. Since spectrum based fault localisation heuritsics only require traces from test runs— readily available after running the regression test suite— these heuritics are ideally suited for locating regression faults in a continuous integration context.

Spectrum based fault localisation is quite an effective heuristic as reported in several papers [19, 1, 41, 42]. Sometimes other names are used, namely coverage based fault localisation [12] and statistical debugging [53]. To understand how spectrum based fault localisation heuristics work, there are three crucial elements to consider: (1) the test coverage matrix; (2) the hit-spectrum; and the (3) fault locator. We explain each of them below.

1. All spectrum based fault localisation heuristics collect coverage information of the elements under test in a *test coverage matrix*. This is a matrix, where the rows 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 (i.e. covered) by the test case (marked as 1) or not (marked as 0).
2. Next, this test coverage matrix is transformed into the *hit-spectrum* (sometimes also called *coverage spectrum*) of a program. The hit-spectrum of an element under test is 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 and n_f and n_p are the numbers of failing and passing test cases that do *not* execute the element under test. Table 1 shows an example test coverage matrix and spectrum.
3. Finally, the heuristic assigns a suspiciousness to each element under test by means of a *fault locator*. This suspiciousness indicates the likelihood of the unit to be at fault. The underlying intuition is that an element under test executed more in failing tests and less in passing tests gets a higher suspiciousness and appears at top position in the ranking. Sorting the elements under test according to their suspiciousness in descending order produces the ranking. Many (if not all) variants of spectrum based fault localisation create a new fault locator; Table 2 gives an overview of the most popular ones.

Granularity. Other variants of spectrum based fault localisation concern the choice of the elements under test. Indeed, spectrum based fault localisation has been applied at different levels of granularity, including *statements* [20, 19, 7, 45, 30], *blocks* [2, 1, 28, 29, 24], *methods* [41, 42, 48], and *classes* [9, 23]. The seminal work on spectrum based fault localisation started off with statement level granularity [20]. As a result, most of the early research focussed at statement level, sometimes extended to basic blocks of statements. The effectiveness at the method level has been investigated in only a few cases and then even as part of a large-scale comparison involving several levels of granularity [41, 42, 48].

Today, the space of known spectrum based fault localisation heuristics is classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T, and Naish2). In this paper, we explore the hit-spectrum as a third dimension. We expand the four tuple (e_f, e_p, n_f, n_p) so that e_f and e_p incorporate patterns of method calls we extracted by means of frequent itemset mining.*
In the remainder of this paper we refer to the current state of the art as **raw spectrum analysis**, *while our variant will be denoted with* **patterned spectrum analysis**.

Evaluation metric: wasted effort. Fault localisation heuristics produce a ranking 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 fault [42]. Despite providing a good summary of the accuracy of a heuristic, absolute measures are currently deemed better for comparison purposes [33, 41, 42]. Today, the *wasted effort* metric is commonly adopted [41, 42, 48]. Consequently, we will rely on the *wasted effort* when comparing raw spectrum analysis against patterned spectrum analysis. (The exact formula for wasted effort is provided in Section 5 — Equation 8).

Data Set. The early evaluations on the effectiveness of raw spectrum analysis heuristics were done by means of small C programs, taken from the `Siemens` set and `Space` [4]. Despite having industrial origins, the faults used in the experiments were manually seeded by the authors [18, 20]. The next attempt at a common dataset for empirical evaluation of software testing and debugging is the Software-Artifact Infrastructure Repository (SIR) [13]. Unfortunately, most of the faults in this dataset are manually seeded as well. Consequently, Dallmeier et. al created the iBugs dataset containing real faults drawn from open source Java projects [10]. iBugs contains 223 faults all accompanied with at least one failing test case to reproduce the fault. The last improvement on fault datasets is known as Defects4J [21]. Defects4J has a few advantages over iBugs: all the faults in Defects4J are isolated— the changes in V_{fix} for corresponding V_{bug} purely represent the bug fix. Unrelated changes —such as adding features or refactorings— are isolated. Defects4J also provides a comprehensive test execution framework, which abstracts away the underlying build system and provides a uniform interface to common build tasks — compilation, test runs, etc.... To the best of our knowledge, the Defects4J has not yet been used for evaluating raw spectrum analysis. Hence, we will adopt the Defects4J dataset for our comparison.

Table 1: An Example Test Coverage Matrix and Hit-Spectrum

element under test	Failing test cases			Passing test cases			e_f	e_p	n_f	n_p
	t_1	$\dots$	t_m	t_{m+1}	$\dots$	t_n				
$unit_i$	$X_{i,1}$	$\dots$	$X_{i,m}$	$X_{i,m+1}$	$\dots$	$X_{i,n}$	$\sum_{j=1}^{m} X_{i,j}$	$\sum_{j=m+1}^{n} X_{i,j}$	$m - e_f$	$(n-m) - e_p$

t_i denotes i_{th} test case $X_{j,j}$ takes the binary value 0 or 1

Table 2: Popular Fault Locators

Faul Locator	Definition
Tarantula [19]	$\dfrac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$
Ochiai [1]	$\dfrac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$
T* [42]	$\left(\dfrac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}\right).max\left(\frac{e_f}{e_f+n_f},\frac{e_p}{e_p+n_p}\right)$
Naish2 [32]	$e_f - \frac{e_p}{e_p+n_p+1}$

The current state of the art relies on wasted effort to evaluate fault localisation heuristics mainly via the SIR *and* iBugs *datasets. When comparing* raw spectrum analysis *against* patterned spectrum analysis, *we rely on wasted effort as well, yet adopt the more recent* Defects4J *dataset.*

Applications. In the initial research papers, the main perspective for spectrum based fault localisation was to assist an individual programmer during debugging [20, 19, 2, 1, 28, 7, 30, 29, 24]. The typical scenario was a debugging window showing not only the stack trace but also a ranked list of potential locations for the fault, hoping that the root cause of the fault appears at the top of the list. This explains why the accuracy of these heuristics was mainly evaluated in terms of percentage of code that needs to be inspected. Recently, another application emerged: automated fault repair [26, 43, 47]. The latter techniques repair a fault by modifying potentially faulty program elements in brute-force manner until a valid patch —i.e. one that makes the tests pass— is found. The first step in automated repair is fault localisation, which in turn resulted in another evaluation perspective, namely whether it increases the effectiveness of automated fault repair [34].

The two commonly used applications for fault localisation are debugging *and* automated fault repair. *Up until now, continuous integration has never been considered. We will present the implications of broken builds within continuous integration in Section 3.*

3. MOTIVATING SCENARIO

Since we propose continuous integration as a testbed for validating patterned spectrum analysis, it is necessary to be precise about what exactly constitutes a continuous integration tool and what kind of requirements it imposes on a fault localisation heuristic. As commonly accepted in requirements engineering, we specify the context and its requirements by means of a *scenario*. The driving force underlying the scenario is the observation that if a build is broken, it should be repaired immediately hence the root cause should be identified as quickly as possible.

Note that at a first glance this scenario may seem naive. Nevertheless, it is based on a series of discussions with software engineers working with the agile development process SCRUM and who rely on a continuous integration server to deploy their software on a daily basis. The discussions were held during meetings of the steering group of the Cha-Q project (http://soft.vub.ac.be/chaq/), where we confronted practitioners with the scenario below and asked for their input on what a good fault localisation method should achieve. Therefore, we can assure the reader that the scenario represents a real problem felt within today's software teams.

Prelude: GeoPulse GeoPulse[1] is an app which locates the nearest location of an external heart defibrillator so that in case of an emergency one can quickly help an individual suffering from a cardiac arrest. The software runs mainly as a web-service (where the database of all known defibrillators is maintained), yet is accessed by several versions of the app running on a wide range of devices (smart phones, tablets and even a mini-version for smart watches).

Software Team. There is a 12 person team responsible for the development of the GeoPulse app; 10 work from the main office in Brussels while 2 work from a remote site in Budapest. The team adopts a SCRUM process and uses continuous integration to ensure that everything runs smoothly. It's a staged build process, where the build server performs the following steps: (1) compilation; (2) unit tests; (3) static code analysis; (4) integration tests; (5) platform tests; (6) performance tests; (7) security tests. Steps (1) — (3) are the level 1 tests and fixing problems there is the responsibility of the individual team members; steps (4) — (7) are the level 2 defence and the responsibility of the team.

Scene 1: Unit Testing. Angela just resolved a long standing issue with the smart-watch version of the app and drastically reduced the response time when communicating with the smart-phone over bluetooth. She marks the issue-report as closed, puts the issue-ID in the commit message and sends everything off to the continuous integration server. A few seconds later, the lava-lamp in her cubicle glows orange, notifying a broken build. Angela quickly inspects the build-log and finds that one unit-test fails. Luckily, the guidelines for unit tests are strictly followed within the GeoPulse team (unit-tests run fast, have few dependencies on other modules and come with good diagnosing messages). Angela can quickly pinpoint the root cause as a missing initialisation routine in one of the subclasses she created. She adds the initialiser, commits again and this time the build server finds no problems and commits her work to the main branch for further testing during the nightly build. The lava-lamp turns green again and Angela goes to fetch a coffee before starting her next work item.

Purpose. This scene illustrates the importance of the Level 1 tests and the role of unit tests in there. Ideally, running the whole suite of unit tests takes just a few seconds and if one of the unit tests fails, it is almost straightforward to locate the fault. Moreover, it is also clear who should fix

[1]The name and description of the app is fictitious.

the fault, as it is the last person who made a commit on the branch. Thus, fault localisation in the context of unit tests *sensu stricto* is pointless: the fault is located within the unit by definition and the diagnosing messages combined with the recent changes is usually sufficient to repair efficiently.

Scene 2: Integration Testing. Bob arrives in his office in the morning and sees that the lava-lamp is purple, signifying that the nightly build broke. He quickly inspects the server logs and sees that the team resolved 9 issues yesterday, resulting in 8 separate branches merged into the main trunk. There are three seemingly unrelated integration tests which fail, thus Bob has no clue on the root cause of the failure. During the stand-up meeting the team discusses the status of the build, and then suspends all work to fix the broken build. Team members form pairs to get rapid feedback, however synchronising with Vaclav and Ivor (in the Budapest office) is cumbersome — Skype is not ideal for pair programming. It takes the team the rest of the morning until Angela and Vaclav eventually find and fix the root cause of the fault — there was a null check missing in the initialisation routine Angela added yesterday.

Purpose. This scene illustrates the real potential of fault localisation during continuous integration. Faults in integration tests rarely occur, but have a big impact because they are difficult to locate hence difficult to assign to an individual. Moreover, software engineers must analyse code written the day before and integration tests written by others: the mental overhead of such context switches is significant. Finally, since these faults block all progress, team members must drop all other tasks to fix the build.

3.1 Requirements

From the above scenario, we can infer a few requirements that should hold for a fault localisation heuristic integrated in a continuous integration server.

Method Level Granularity. The seminal work on raw spectrum analysis (named Tarantula) was motivated by supporting an individual test engineer, and chose statement level granularity [20]. However, for fault localisation within integration tests, method level granularity is more appropriate. Indeed, the smallest element under test in object oriented testing is a method [6]. This also shows in modern IDE, where failing tests and stack traces report at method level. Last but not least, objects interact through methods, thus integration faults appear when objects don't invoke the methods according to the (often implicit) protocol.

Top 10. A fault localisation heuristic produces a ranked list of program elements likely to be at fault, thus the obvious question is how deep in the ranking the correct answer should be to still be considered acceptable. In the remainder of the paper we set the threshold to 10, inspired by earlier work from Lucia et. al [29]. 10 is still an arbitrary number but was confirmed to be a good target during our interviews with real developers.

Fault localisation is applicable for complex systems with multiple branches and staged testing. Faults in integration tests in particular are very relevant: they seldom occur, but when they do, they have a big impact on the team productivity. Thus, to compare raw spectrum analysis against patterned spectrum analysis we should treat integration tests differently than unit tests.

4. PATTERNED SPECTRUM ANALYSIS

As explained earlier, current raw spectrum analysis heuristics comprise several variants, typically classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T*, and Naish2). In this paper, we explore the hit-spectrum as a third dimension, incorporating patterns of method calls extracted by means of frequent itemset mining.

Below, we explain the details of the patterned spectrum analysis variant. We run the test suite and for each test case, collect the trace (Cf. Section 4.1), slice the trace into individual method traces (Cf. Section 4.2), reduce the sliced traces into call patterns for a method (Cf. Section 4.3), calculate the hit-spectrum by incorporating frequent itemset mining (Cf. Section 4.4), and finally rank the methods according to their likelihood of being at fault (Cf. Section 4.5).

4.1 Collecting the Trace

We maintain a single trace per test case. When a test runs, it invokes methods in the project base code. We intercept all the method calls originating from the base code method. We do not intercept calls in test methods, since we assume that the test oracles themselves are correct. The trace is collected by introducing logger functionality into the base code via AspectJ[2]. More specifically, we use a method call join point with a pointcut to pick out every call site. For each intercepted call, we collect the called method identifier, caller object identifier, and the caller method identifier. These identifiers are integers uniquely associated with a method name.

Listing 1: Code snippet for a sample method

```
1  public class A {
2   B objB;
3   C objC;
4   .....
5   public void collaborate() {
6    b.getData();
7    while(...) {
8     if(...)
9      c.getAttributes();
10    if(...)
11     c.setAttributes(...);
12    if(...)
13     c.processData(...);
14   } // while
15   b.saveData();
16  } // method
17 }
```

As an example, assuming the test case instantiates three objects of class A and calls method `collaborate()` (Listing 1) for each instance. A sample trace in a test case, specifically highlighting the method calls originating from the `collaborate()` method in Listing 1, is shown in Table 3. The three instances of class A are shown (id 1, 2, and 3) which each received a separate call to `collaborate()`. The execution of `collaborate()` on object id 1 resulted into a call to `getData()` (line 6), `getAttributes()` (line 9), `setAttributes()` (line 11), and finally `saveData()` (line 15). Execution of `collaborate()` on object id 2 and 3 results in a slightly different calls.

The 'caller object id' is the identifier of the caller object which calls the method, the 'caller' is the method from which

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

Table 3: A Sample Trace Highlighting Calls in Listing 1

caller object id	caller id[†]	callee id[‡]
1	5	6
1	5	9
1	5	11
1	5	15
2	5	6
2	5	11
2	5	13
3	5	6
3	5	11
3	5	15
...	...	...

[†] caller id 5 indicates method `collaborate()` in Listing 1

[‡] callee id is the line number in Listing 1

the call is made and the 'callee' is the called method. When a method is executed in a class context (static methods can be executed without instantiating a class), there is no caller object, hence we mark the 'object caller id' as -1.

Considering the intercepted call `getData()` (line 6), the "caller object id" is the id of the class A object instantiated in the test case, the "caller id" is the id of method `collaborate()`, and the "callee id" is the id of method `getData()`. In a similar manner, calls originating from other methods such as method `getData()` of class B invoked from method `collaborate()` (line 6) are recorded in the trace.

4.2 Slicing the Trace

Once a trace for a test case is obtained, we slice the trace into individual method traces. Each sliced trace represents the trace for each executed method in the test case.

The sliced 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 represents the method calls invoked from method `m()` through the same caller object. If the method `m()` is static, the calls appear in a single trace for 'caller object id' -1.

Referring to Table 3, $t_1 = \langle 6, 9, 11, 15\rangle$ for the calls of method `collaborate()` (id 5) with caller object id 1, $t_2 = \langle 6, 11, 13\rangle$ with caller object id 2, and $t_3 = \langle 6, 11, 15\rangle$ with caller object id 3. Therefore, the sliced trace τ_5 for method `collaborate()` (id 5) is as follows.

$$\mathcal{T}_5 = \{\langle 6, 9, 11, 15\rangle, \langle 6, 11, 13\rangle, \langle 6, 11, 15\rangle\} \qquad (1)$$

4.3 Obtaining Call Patterns

We reduce the sliced trace $\mathcal{T}_m$ of a method `m()` coming from a test case $\mathcal{T}$ into a set of call patterns $\mathcal{S}_{\mathcal{T}_m}$. To arrive at set of call patterns $\mathcal{S}_{\mathcal{T}_m}$, we adopt the *closed itemset mining algorithm* [52]. Given the sliced trace $\mathcal{T}_m$ of method `m()` in a test case $\mathcal{T}$, we define:

- $\mathcal{X}$ —*itemset*— a set of method calls.
- $\sigma(\mathcal{X})$ —support of $\mathcal{X}$— the number of t_i in $\mathcal{T}_m$ that contain this itemset $\mathcal{X}$.
- *minsup* —minimum support of $\mathcal{X}$— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset $\mathcal{X}$ is frequent when $\sigma(\mathcal{X}) \geq$ *minsup*.
- *closed itemset* — a frequent itemset $\mathcal{X}$ is closed if there exists no proper superset $\mathcal{X}'$ whose support is the same as the support of $\mathcal{X}$ (i-e. $\sigma(\mathcal{X}') = \sigma(\mathcal{X})$).

We refer to closed itemset $\mathcal{X}$ as a call pattern. We set *minsup* to 1 to include call patterns for the methods executed with one object only or for those executed in a class context. The set of call patterns $\mathcal{S}_{\mathcal{T}_m}$ for method `collaborate()` (id 5) from sliced trace $\mathcal{T}_5$ (Equation 1) is as follows.

$$\mathcal{S}_{\mathcal{T}_5} = \{\{6, 9, 11, 15\}, \{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \qquad (2)$$

4.4 Calculating the Hit-Spectrum

Unlike raw spectrum analysis, where there is a single test coverage matrix per program, patterned spectrum analysis creates a test coverage matrix for each executed method. In the raw spectrum analysis, a row of test coverage matrix corresponds to a method, which is a program element per se, and the hit-spectrum (e_f, e_p, n_f, n_p) indicates whether or not the method is involved in test cases. In patterned spectrum analysis, there is a separate test coverage matrix for each method and a row corresponds to a call pattern (itemset $\mathcal{X}$) of the method. Here the call pattern ($\mathcal{X}$) is not a program element anymore. The hit-spectrum (e_f, e_p, n_f, n_p) of $\mathcal{X}$ not only indicates whether or not the method is involved in a test case, but also summarises its run-time behaviour.

The call patterns of a method `m()` in patterned 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 patterns $\mathcal{S}_m$ (Equation 3) for each method `m()` — which is the union of (i) the call patterns of a method resulting from the failing test cases ($\mathcal{S}_{\mathcal{T}_m} \in \mathbb{T}_F$) and (ii) the call patterns resulting from the passing test cases ($\mathcal{S}_{\mathcal{T}_m} \in \mathbb{T}_P$).

$$\mathcal{S}_m = \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \ \mathcal{T} \in \mathbb{T}_F\} \cup \{\mathcal{X} | \mathcal{X} \in \mathcal{S}_{\mathcal{T}_m} \wedge \ \mathcal{T} \in \mathbb{T}_P\} \qquad (3)$$

The set $\mathcal{S}_m$ (Equation 3) is used to construct the test coverage matrix for a method.

As an example, consider the set of call patterns for $\mathcal{S}_{\mathcal{T}_5}$ (Equation 2) of the method `collaborate()`. Assuming, this call pattern results from a failing test case it will end up in $\mathcal{T} \in \mathbb{T}_F$. However, the same method `collaborate()` is also executed in a passing test case (i-e $\mathcal{T} \in \mathbb{T}_P$) and will result in another set of call patterns, shown in Equation 4.

$$\mathcal{S}_{\mathcal{T}_5} = \{\{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \qquad (4)$$

Then, the call pattern set $\mathcal{S}_5$ for the method `collaborate()` becomes the union of Equation 2 and Equation 4.

$$\mathcal{S}_5 = \{\{6, 9, 11, 15\}, \{6, 11, 13\}, \{6, 11, 15\}, \{6, 11\}\} \qquad (5)$$

The hit spectrum is then calculated for each call pattern in the call pattern set which ultimately results in a test coverage matrix for each method. As an example, we show the test coverage matrix for `collaborate()` in Table 4.

4.5 Ranking Methods

Based on the test coverage matrix of call patterns for each method, each pattern in the call pattern set $\mathcal{S}_m$ (Equation 3) gets a suspiciousness score. This suspiciousness is calculated by using a fault locator [1, 41, 42]. Then, we set the suspiciousness of the method as the maximum suspiciousness of its constituting patterns.

Suspiciousness per call pattern. Each call pattern $\mathcal{X} \in S_m$ (Equation 3) gets a suspiciousness $W(\mathcal{X})$ calculated with a fault locator. In principle, any fault locator can be chosen from the literature. However, for our comparison purpose

Table 4: An Example Test Coverage Matrix for Method `collaborate()`

Call pattern $\mathcal{X}$	Failing test cases ($\mathcal{T} \in \mathbb{T}_F$)	Passing test cases ($\mathcal{T} \in \mathbb{T}_P$)	$e_f(\mathcal{X})$	$e_p(\mathcal{X})$	$n_f(\mathcal{X})$	$n_p(\mathcal{X})$	$W(\mathcal{X})$
	t_1	t_2					
$\{6, 9, 11, 15\}$	1	0	1	0	0	1	1.0
$\{6, 11, 13\}$	1	1	1	1	0	0	0.7
$\{6, 11, 15\}$	1	1	1	1	0	0	0.7
$\{6, 11\}$	1	1	1	1	0	0	0.7

we tested all four fault locators mentioned in Table 2 in patterned spectrum analysis and Ochiai (Equation 6) came out as the best performing one. For our running example, the suspiciousness $W(\mathcal{X})$ for each call pattern of method `collaborate()` is given in Table 4.

$$W(\mathcal{X}) = \frac{e_f(\mathcal{X})}{\sqrt{(e_f(\mathcal{X}) + n_f(\mathcal{X})) * (e_f(\mathcal{X}) + e_p(\mathcal{X}))}} \quad (6)$$

Suspiciousness per method. Each method `m()` gets a suspiciousness $W(m)$ which is the suspiciousness of the call pattern $\mathcal{X}$ with the highest suspiciousness (Equation 7). We choose the maximum (instead of average) for the suspiciousness score because the technique is looking for exceptional traces: one unique and highly suspicious pattern is more important than several unsuspicious ones. Those methods without call patterns set have suspiciousness 0. The suspiciousness for method `collaborate()` $W(5)$ in our running example is 1.0, which is the suspiciousness of the call pattern ($\{6, 9, 11, 15\}$)— with highest suspiciousness (Table 4).

$$W(m) = \max_{\mathcal{X} \in S_m} \Big(W(\mathcal{X}) \Big) \quad (7)$$

Ranking. Finally, a ranking of all executed methods is produced using their suspiciousness W(m). The suspiciousness of the method indicates its likelihood of being at fault. Those methods with the highest suspiciousness appear a the top in the ranking.

5. CASE STUDY SETUP

Given the current state of the art (referred to as raw spectrum analysis) and the variant proposed in this paper (referred to as patterned spectrum analysis), we can now compare the effectiveness of these two heuristics from the perspective of a continuous integration scenario. We give some details about the dataset used for the comparison (Defects4J), the evaluation metric (Wasted Effort), to finish with the research questions, and protocol driving the comparison.

Dataset. We use 351 real faults from 5 open source java projects: Apache Commons Math, Apache Commons Lang, Joda-Time, JFreeChart, and Google Closure Compiler. The descriptive statistics of these projects are reported in Table 5. These faults have been collected by Just et. al. into a database called Defects4J[3] (a **d**atabase of **e**xisting **f**aults to **e**nable **c**ontrolled **t**esting **s**tudies for **J**ava programs) [21]. The database contains meta info about each fault including the source classes modified to fix the fault, the test cases that expose the fault, and the test cases that trigger at least one of the modified classes. Although, the framework does not explicitly list the modified methods, we could reverse engineer those by means of the patches that come with the framework. Note that we excluded 3 faults of Apache Commons Lang, 2 faults of Apache Commons Math, and 1 fault of Joda-Time since the fault was not located inside a method.

[3]http://defects4j.org

Unfortunately, the Defects4J dataset does not distinguish between unit tests or integration tests. As argued in the Scenario (Section 3), this is a crucial factor when assessing a fault localisation heuristic in a continuous integration context. We, therefore, manually inspected a sample of test methods and noticed that four projects (Apache Commons Math, Apache Commons Lang, Joda-Time, and JFreeChart) mainly contain unit tests: they have a small (often empty) set-up method, and test methods contain only a few `asserts`. One project however (Closure Compiler) relies on integration tests. The test cases there, are a subclass of `CompilerTestCase` that defines a few template methods, which are the entry point to several classes in the base code of the project.

To corroborate this manual inspection, we calculated the number of methods triggered in each fault spectrum analysis. The assumption here is that integration tests exercise several methods in various classes, consequently the fault spectrum analysis should trigger many methods as well. Thus, projects which gravitate towards integration testing should trigger many methods while projects gravitating towards unit tests should trigger far fewer. The results are shown in the last two columns (μ and σ) of Table 5; listing the average and standard deviation per project respectively. The high number of μ for the Closure project is an indication that the Closure tests exercise a lot of the base code, yet the high standard deviation σ signals the presence of unit tests as well. On the other hand, the low number of μ for the other project hints at mostly unit tests, yet Chart has a standard deviation σ of 407 (compared to an average of 306), indicating a few outlier tests which cover a lot of the base code.

The Defects4J dataset does not distinguish between unit tests or integration tests. However, one project (Closure Compiler) gravitates towards integration tests. Therefore, the results of the Closure Compiler should serve as circumstantial evidence during the comparison.

Wasted Effort. As mentioned earlier, we compare by means of the *wasted effort* metric, commonly adopted in recent research [41, 42, 48]. The wasted effort indicates the number of non-faulty methods to inspect in vain before reaching the faulty method.

$$\text{wasted effort} = m + (n + 1)/2 \quad (8)$$

Where

- m is the number of non-faulty methods ranked strictly higher than the faulty method;
- n is the number of non-faulty methods with equal rank to the faulty method. This deals with ties in the ranking.

The comparison is driven by the following research questions.

Table 5: Descriptive Statistics for the Projects Used in Our Experiments — Defects4J [http://defects4j.org]

Project	# of Bugs	Source KLoC	Test KLoC	# of Tests	Age (years)	# Methods triggered (μ†)	# Methods triggered ($\sigma^‡$)
Math (^{1})	106	85	19	3,602	11	153.1	140.8
Lang (^{2})	65	22	6	2,245	12	89.3	55.2
Time (^{3})	27	28	53	4,130	11	586.0	209.5
Chart (^{4})	26	96	50	2,205	7	306.9	407.5
Closure (^{5})	133	90	83	7,927	5	2043.0	1228.9

† Average number of methods triggered by the Spectrum based fault localisation—— ‡ Standard deviation

(^{1}) Apache Commons Math – http://commons.apache.org/math
(^{2}) Apache Commons Lang – http://commons.apache.org/lang
(^{3}) Joda-Time – http://joda.org/joda-time
(^{4}) JFreeChart – http://jfree.org/jfreechart
(^{5}) Google Closure Compiler – http://code.google.com/closure/compiler/

RQ1 – *Which ranking results in the lowest wasted effort: raw spectrum analysis or patterned spectrum analysis?*
Motivation: This is the first step of the comparison; assessing which of the two fault localisation methods provides the best overall ranking.

RQ2 – *How often do raw spectrum analysis and patterned spectrum analysis rankings result in a wasted effort ≤ 10 ?*
Motivation: Based on the scenario (Section 3), we investigate how many times the location of the fault is ranked in the first 10 items.

RQ3 – *How does the number of triggered methods affect the wasted effort of raw spectrum analysis and patterned spectrum analysis?*
Motivation: Again, based on the scenario (Section 3) we gauge the impact of integration tests. The number of methods triggered by the fault spectrum analysis acts as a proxy for the degree of integration tests in the test suite.

Fault Locator. One dimension of variation in spectrum based fault localisation is the fault locator; Table 2 lists the most popular ones. As explained in Section 4.5, for comparison purpose we use Ochiai for patterned spectrum analysis. However, for the optimal configuration of raw spectrum analysis, we actually tested all four fault locators (Table 2). Naish2 performed the best on the Defects4J dataset with method level granularity as can be seen in Table 6. There, we compare the wasted effort of Naish2 against the wasted effort of other fault locators, using the 133 defects in the Closure project. For most defects, Naish2 results in a better or equal ranking; only for a few defects is the ranking with other locators better. For space reasons we do not show the comparison on other projects, but there as well Naish2 was the best. Hence, we choose Ochiai for patterned spectrum analysis and Naish2 for raw spectrum analysis in the case study.

Table 6: Naish within Raw Spectrum Analysis vs. Tarantula, Ochiai and T*

Faul Locator	<	>	=
Tarantula	50 (38%)	8 (6%)	75 (56%)
Ochiai	46 (35%)	6 (5%)	81 (61%)
T*	20 (15%)	4 (3%)	109 (82%)

Protocol. To run the fault spectrum analysis, we check out a faulty version (V_{fault}) for each project. Then we run the actual 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. Given the continuous integration context for this research, this is the most logical way to minimise the number of tests which are fed into the spectrum based fault localisation. Note that this explains why the number of methods triggered by a fault spectrum is a good indicator for the integration tests; since the tests are chosen such that they cover all changes made to fix the defect.

6. RESULTS AND DISCUSSION

In this section, we address the three research questions introduced in Section 5. This allows for a multifaceted comparison of the effectiveness of patterned spectrum analysis against the state of the art raw spectrum analysis.

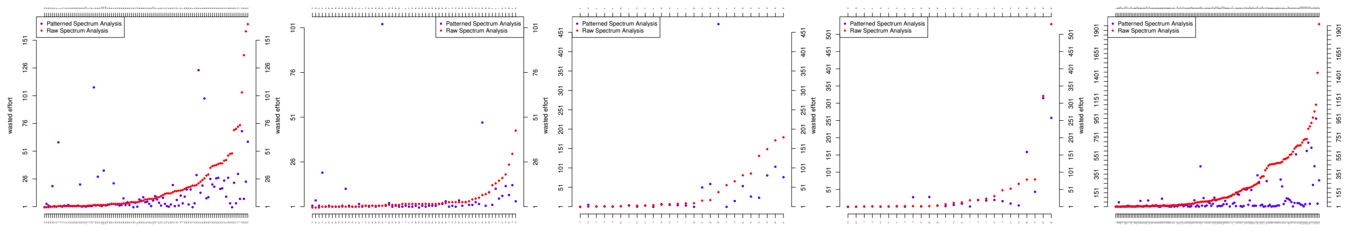
RQ1 – *Which ranking results in the lowest wasted effort: raw spectrum analysis or patterned spectrum analysis?*

To determine the best performing heuristic, we plot the wasted effort for all of the faults for both heuristics. To allow for an easy exploration of the nature of the difference, we sort the faults according to the wasted effort of raw spectrum analysis and plot the wasted effort for patterned spectrum analysis accordingly. The result can be seen in (Figures 1a, 1b, 1c, 1d, and 1e). Next, we count all the faults for which the wasted effort (in patterned spectrum analysis) is strictly less (<), strictly more (>), or the same (=) and list the absolute numbers per project (See Table 7).

Table 7: Comparing Wasted Effort: Patterned Spectrum Analysis vs Raw Spectrum Analysis

Project	<	>	=	Total
Math	69 (66%)	22 (21%)	13 (13%)	104
Lang	36 (58%)	14 (23%)	12 (19%)	62
Time	16 (62%)	7 (27%)	3 (12%)	26
Chart	16 (62%)	7 (27%)	3 (12%)	26
Closure	101 (76 %)	30 (23 %)	2 (2%)	133
Total	238 (68 %)	80 (23 %)	33 (9%)	351

To illustrate how the rankings of the heuristics differ, we inspect fault 40 of the Closure project where the wasted effort for patterned spectrum analysis is 0.5 (the faulty method is ranked first), while for raw spectrum analysis the wasted effort is 183. This is due to the fact that the faulty method has a call pattern which is unique in all failing test cases, hence is easily picked up by patterned spectrum analysis. On the other hand, just marking whether or not the method is executed, is not discriminating in raw spectrum analysis. The number of failing test cases covering the faulty method and non-faulty methods, is the same 169. Yet, the non-faulty methods have



(a) Math (b) Lang (c) Time (d) Chart (e) Closure

Figure 1: The comparison plots of all the rankings in each Lang

more suspiciousness than faulty method because the number of passing test cases covering the non-faulty methods is less. Since more passing test cases cover the faulty method (high value of e_p), it renders the faulty method less suspicious.

> *For 68% faults in the dataset, the wasted effort with patterned spectrum analysis is lower than raw spectrum analysis. Moreover, this improvement is a lot better for the Closure project (the one system in the data set which gravitates towards integration tests), where we see an improvement for 76% of faults (101 out of 131).*

RQ2 – *How often do raw spectrum analysis and patterned spectrum analysis rankings result in a wasted effort ≤ 10 ?* Inspired by the scenario in Section 3, we count how many times the location of the fault is ranked in the top 10. To deal with ties in the ranking (especially at position 10), we identify these as having a wasted effort ≤ 10.

Table 8: # Faults where Wasted Effort is ≤ 10

Project	PSA†	RSA‡	† − ‡	Total
Math	73 (70%)	59 (57%)	14	104
Lang	55 (89%)	54 (87%)	1	62
Time	16 (62%)	14 (54%)	2	26
Chart	16 (62%)	13 (50%)	3	26
Closure	56 (42%)	30 (23%)	26	133
Total	216 (62%)	170 (48%)	46	351

† PSA = patterned spectrum analysis.
‡ RSA = raw spectrum analysis.

Table 8 shows, for each project, the number of faults where the wasted effort is within the range of 10 with both heuristics. For three projects (Lang, Time, and Chart), the performance of the patterned spectrum analysis is comparable but still better than the one of the raw spectrum analysis. Whereas, for the remaining two projects (Math and Closure) the performance of the patterned spectrum analysis is noticeably better. These findings confirm that patterned spectrum analysis ranks more faults in the top 10. However, there are still a large amount of faults where the ranking is poor (wasted effort > 10). Especially, for the Closure project less than half (42%) of the faults are ranked in the top 10. Hence, there is still room for improvement, which we will cover in Section 7.

> *The patterned spectrum analysis succeeds in ranking the root cause of the fault in the top 10 for 62% of the faults, against 48% for raw spectrum analysis.*

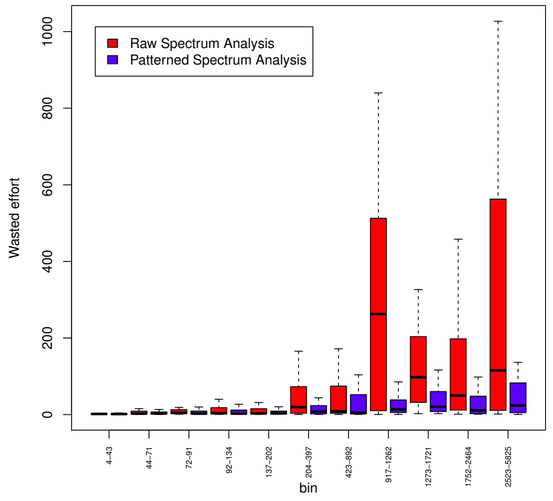


Figure 2: # Triggered Methods vs. Wasted Effort

RQ3 – *How does the number of triggered methods affect the wasted effort of raw spectrum analysis and patterned spectrum analysis?*

In Section 5, we argued that the number of methods triggered by the fault spectrum analysis is an indicator of the gravitation towards integration tests (see also the last two columns in Table 5). If that is the case, a good spectrum based fault localisation heuristic should obtain a good ranking for a particular fault regardless of the number of triggered methods. Again, based on the scenario (Section 3), we gauge the impact of integration tests. Therefore, for each fault, we calculate the number of methods triggered by the fault spectrum analysis. We then sort the faults according to the number of methods and inspect the trend with respect to the number of triggered methods. Unfortunately, the standard deviation for the number of triggered methods is really high (see the σ column in Table 5) and a normal scatterplot mainly showed the noise. Therefore, we group the faults according to the triggered methods into 11 bins of 32 elements. (As these numbers did not divide well, there were two bins having 30 and 33 triggered methods respectively.) This binning was decided as a trade-off for having an equal number of elements per bin and enough bins to highlight a trend in the

Table 9: # Triggered Methods vs. Wasted Effort

Bin	PSA†			RSA‡		
	Q1	Median	Q3	Q1	Median	Q3
4-43	1.0	1.5	2.5	1.0	1.8	2.9
44-71	1.5	3.0	6.8	2.2	2.8	8.5
72-91	1.5	2.8	9.1	2.4	5.2	13.0
92-134	1.5	2.8	11.5	1.5	3.8	17.6
137-202	1.5	3.2	9.1	1.5	3.2	15.5
204-397	2.0	8.0	23.5	3.5	20.0	73.0
423-892	1.9	5.0	51.4	3.5	9.0	70.8
917-1262	5.8	14.0	38.5	10.4	263.0	511.6
1273-1721	8.2	20.8	56.4	33.9	97.8	203.1
1752-2464	2.5	11.2	40.9	12.4	50.0	196.0
2523-5825	5.0	24.0	77.5	11.0	115.5	561.1

† patterned spectrum analysis. ‡ raw spectrum analysis.

number of triggered methods, if any. For each of the bins, we calculated the first quartile, median, and the third quartile, listing them all in Table 9 and plotting them in a series of boxplots (Figure 2)

Table 9 and Figure 2 illustrate that the number of methods triggered has little effect on patterned spectrum analysis, however, quite a lot on raw spectrum analysis. The last four bins, in particular, contain faults which trigger more than thousand methods. The median wasted effort for patterned spectrum analysis is four to eighteen times lower than raw spectrum analysis.

The better rankings for Closure in Table 7 and Table 8 are inconclusive, as one case is not enough to generalise upon. Yet, based on an analysis of the number of methods triggered by the fault spectrum, there is at least circumstantial evidence that patterned spectrum analysis performs better for integration tests.

7. POSSIBLE IMPROVEMENTS

Upon closer inspection of those faults ranked high by the patterned spectrum analysis heuristic, we can infer some suggestions for improvement regarding future variations.

First of all, an inherent limitation is that a faulty method which does not call any other methods will always be ranked at the bottom. Indeed, such methods don't have a call pattern (which is the primary coverage element appearing in the test coverage matrix), thus the method gets suspiciousness 0. In our case study, we noticed a few cases where none of the faulty methods had any call pattern. More specifically, there are 4 such cases in the Math project, 3 in the Chart project, 2 in the Time and Lang projects, and only 1 in the Closure project. The best example corresponds to the highest wasted effort on fault 60 of the Lang project (See Listing 2). Indeed, the faulty method "`contains(char)`" in class "`org.apache.commons.lang.text.StrBuilder`" gets suspiciousness 0 because the for loop only performs direct accesses to memory and never calls any methods.

Similarly, the highest wasted effort for fault 22 in the Math project is due to the faulty method "`isSupportUpperBoundInclusive()`" in class "`distribution.UniformRealDistribution`" which again never calls any other methods. In this case, the method body contained a single statement "`return false;`"; the bug fix replaced it by "`return true;`". A last example is fault 22 in Time project; where the fault resided in a faulty constructor, hence did again not have any method call pattern.

Listing 2: Code snippet for a sample method

```
public boolean contains(char ch) {
 char[] thisBuf = buffer;
 // Correct code
//for (int i = 0; i < this.size; i++) {
 // Incorrect code
for (int i = 0;i < thisBuf.length;i++) {
  if (thisBuf[i] == ch) {
   return true;
  }
 }
}
```

Listing 3: Unique call sequence in faulty method `tryMinimizeExits(Node,int,String)`

```
1 Node.getLastChild()
2 NodeUtil.getCatchBlock(Node)
3 NodeUtil.hasCatchHandler(Node)
4 NodeUtil.hasFinally(Node)
5 Node.getLastChild()
6 tryMinimizeExits(Node,int,String)
```

Second, patterned spectrum analysis is often able to push the faulty method high in the ranking, however there are several cases where it never reaches the top 10. A nice example is fault 126 in Closure, where the wasted effort for patterned spectrum analysis is 85.5. This value is still lower than the one given by raw spectrum analysis (532.5), yet it is too high to ever be considered in a realistic scenario. Manually analysing the traces of the faulty method `tryMinimizeExits(Node,int,String)` in class `com.google.javascript.jscomp.MinimizeExitPoints`, we found a unique call pattern (Listing 3) which is only called in the failing tests. The bug fix[4] reveals that the developers removed the "if check" with a `finally` block. This "if check" involves the last 3 calls in Listing 3 (lines 4-6). Despite being unique, the reason why this call pattern was not picked up by patterned spectrum analysis is because the order of method calls is crucial. Indeed, the call pattern in patterned spectrum analysis is an itemset, hence the call pattern is not order preserving and has no repetitive method calls. Note that the importance of the call-order was also pointed out by Lo et. al. [27].

As a future improvements of patterned spectrum analysis, we might incorporate statements or branches into the hit-spectrum. The call-order of methods, as well, is relevant information to incorporate into the hit-spectrum.

8. 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 [37, 50]), we organise them into four categories.

Construct validity – do we measure what was intended ?

Wasted Effort. In this research, we adopted the wasted effort metric to compare raw spectrum analysis against patterned spectrum analysis. However, in information retrieval rankings where users do not want to inspect all outcomes other measures are considered, such as Mean Reciprocal

[4] https://github.com/google/closure-compiler/commit/bd2803

Rank (MRR) or Mean Average Precision (MAP) [39, 25]. It is unclear whether the use of these relative evaluation metrics would alter the results. Nevertheless, the use of an absolute metric alleviates other concerns [33, 41]. Therefore, the impact is minimal.

Fault Masking. One particular phenomenon which occurs in a few faults in the Defects4J dataset is "fault masking" [41]. This is a fault which is spread over multiple locations and where triggering one location already fails the test. The fix for fault 23 of project Chart for instance, comprises two changes in two separate methods of the class "`renderer.category.MinMaxCategoryRenderer`". The first change is to override "`equals(Object)`" method and the second involves changes in method "`setGroupStroke(Stroke)`". The test case which exposes the defect calls both methods, yet the test case fails on the first assertion calling the "`equals(Object)`" method thereby masking the "`setGroupStroke(Stroke)`" method. The question then is what a fault localisation should report: one location or all locations ? Furthermore, how should we assess the ranking of multiple locations. In this research, inspired by earlier work [39, 25], we took the assumption that reporting one location is sufficient and use the highest ranking of all possible locations. However, one could make other assumptions.

Internal validity – are there unknown factors which might affect the outcome of the analyses ?

Multiple faults. One often heard critique on fault localisation heuristics in general and spectrum based fault localisation in particular is that when multiple faults exist, the heuristic will confuse their effects and its accuracy will decrease. Two independent research teams confirmed that multiple faults indeed influence the accuracy of the heuristic, however it created a negligible effect on the effectiveness [12, 49]. We ignore the potential effect of multiple faults in this paper. Nevertheless, future research should study the effect of multiple faults.

Correctness of the Oracle. The continuous integration scenario in Section 3 makes the assumption that the test oracle itself is infallible. However this does not hold in practice: Christophe et. al. observed that functional tests written in the Selenium library get updated frequently [8]. We ignore the effects of the tests being at fault in this paper, but here as well point out that this is something to be studied in future work.

External validity – to what extent is it possible to generalise the findings ? In our study, we experimented with 351 real faults drawn from five representative open source object oriented projects from Defects4J dataset; the most recent defect dataset currently available. Obviously, it remains to be seen whether similar results would hold for other defects in other systems. In particular, there is a bias towards unit test in the Defects4J dataset, with only the Closure project gravitating towards integration tests. Further research is needed to verify whether the patterned spectrum analysis is indeed a lot better on integration tests in other systems.

Reliability – is the result dependent on the tools ? All the tools involved in this case study (i.e. creating the traces, calculating the raw spectrum analysis, and patterned spectrum analysis rankings) have been created by one of the authors. They have been tested over a period of 2 years; thus the risk of faults in the tools is small. Moreover, for the calculation of the raw spectrum analysis rankings we compared as best as possible against the results reported in earlier papers. The algorithm for frequent itemset mining was adopted from open source library SPMF[5], hence there as well the risk of faults is small.

9. CONCLUSION

Spectrum based fault localisation is a class of heuristics known to be effective for localising faults in existing software systems. These heuristics compare execution traces of failing and passing test runs to produce a ranked list of program elements likely to be at fault. The current state of the art (referred to as raw spectrum analysis) comprises several variants, typically classified according to two dimensions: the granularity (statement — block — method — class) and the fault locator function (Tarantula, Ochiai, T*, and Naish2). In this paper, we explore a third dimension: the hit-spectrum. More specifically, we propose a variant (referred to as patterned spectrum analysis) which extends the hit-spectrum with patterns of method calls extracted by means of frequent itemset mining.

The motivation for the patterned spectrum analysis variant stems from a series of contacts with software developers working in Agile projects and relying on continuous integration to run all the tests. Complex systems with multiple branches and staged testing could really benefit from fault localisation. Faults in integration tests, in particular, are very relevant: they seldom occur, but if they do, they have a big impact on the team productivity.

Inspired by the continuous integration motivational example, we compare patterned spectrum analysis against raw spectrum analysis using the Defects4J dataset. This dataset contains 351 real faults drawn from five representative open source java projects. Despite a bias towards unit tests in the dataset, we demonstrate that patterned spectrum analysis is more effective in localising the fault. For 68% faults in the dataset, the wasted effort with patterned spectrum analysis is lower than raw spectrum analysis. Also, patterned spectrum analysis succeeds in ranking the root cause of the fault in the top 10 for 63% of the defects, against 48% for raw spectrum analysis. Moreover, this improvement is a lot better for the Closure project; the one system in the data set which gravitates towards integration tests. There, we see an improvement for 76% defects (101 out of 131). The better rankings for Closure are inconclusive (one case is not enough to generalise upon), yet based on an analysis of the number of methods triggered by the fault spectrum, there is at least circumstantial evidence that patterned spectrum analysis performs better for integration tests. Despite this improvement, we collect anecdotal evidence from those situations where the patterned spectrum analysis ranking is less adequate and derive suggestions for future improvements.

10. ACKNOWLEDGMENTS

Thanks to prof. Martin Monperrus for reviewing an early draft of this paper. This work is sponsored by (i) the Higher Education Commission of Pakistan under a project titled "Strengthening of University of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled "Change-centric Quality Assurance (CHAQ)" with number 120028.

[5]http://www.philippe-fournier-viger.com/spmf/

11. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, Nov. 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. 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, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[3] B. Adams and S. McIntosh. Modern release engineering in a nutshell – why researchers should care. In *Leaders of Tomorrow: Future of Software Engineering, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Osaka, Japan, March 2016.

[4] P. Agarwal and A. P. Agrawal. Fault-localization techniques for software systems: A literature review. *SIGSOFT Softw. Eng. Notes*, 39(5):1–8, Sept. 2014.

[5] M. Beller, G. Gousios, A. Panichella, and A. Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

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

[7] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 378–381, New York, NY, USA, 2012. ACM.

[8] L. Christophe, R. Stevens, C. D. Roover, and W. D. Meuter. Prevalence and maintenance of automated functional tests for web applications. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 141–150, Washington, DC, USA, 2014. IEEE Computer Society.

[9] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM.

[11] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the Int'l Conference on Automated Software Engineering (ASE)*, pages 433–444. IEEE CS, 2009.

[12] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 210–220, New York, NY, USA, 2011. ACM.

[13] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[14] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[15] S. Elbaum, G. Rothermel, and J. Penix. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 235–245, New York, NY, USA, 2014. ACM.

[16] M. Fowler and M. Foemmel. Continuous integration (original version). http://http://www.martinfowler.com/, Sept. 2010. Accessed: April, 1st 2016.

[17] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy. The art of testing less without sacrificing quality. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 483–493, Piscataway, NJ, USA, 2015. IEEE Press.

[18] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[20] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[21] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[22] F. Khomh, B. Adams, T. Dhaliwal, and Y. Zou. Understanding the impact of rapid releases on software quality: The case of firefox. *Empirical Software Engineering*, 20(2):336–373, 2015.

[23] G. Laghari, A. Murgia, and S. Demeyer. Localising faults in test execution traces. In *Proceedings of the 14th International Workshop on Principles of Software Evolution*, IWPSE 2015, pages 1–8, New York, NY, USA, 2015. ACM.

[24] T.-D. B. Le, D. Lo, and F. Thung. Should i follow this fault localization tool's output? *Empirical Softw. Engg.*, 20(5):1237–1274, Oct. 2015.

[25] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM.

[26] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 3–13, Piscataway, NJ, USA, 2012. IEEE Press.

[27] D. Lo, S.-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 460–469, New York, NY, USA, 2007. ACM.

[28] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[29] Lucia, D. Lo, and X. Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM.

[30] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang. Slice-based statistical fault localization. *J. Syst. Softw.*, 89:51–62, Mar. 2014.

[31] A. Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE '08. Conference*, pages 289–293, Aug 2008.

[32] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, Aug. 2011.

[33] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011*

International Symposium on Software Testing and Analysis, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[34] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 191–201, New York, NY, USA, 2013. ACM.

[35] S. Rao, H. Medeiros, and A. Kak. Comparing incremental latent semantic analysis algorithms for efficient retrieval from software libraries for bug localization. *SIGSOFT Softw. Eng. Notes*, 40(1):1–8, Feb. 2015.

[36] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

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

[38] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013.

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

[40] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87(0):48 — 59, 2014.

[41] F. Steimann and M. Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 121–130, Nov 2012.

[42] F. Steimann, M. Frenkel, and R. Abreu. 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, pages 314–324, New York, NY, USA, 2013. ACM.

[43] S. H. Tan and A. Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 471–482, Piscataway, NJ, USA, 2015. IEEE Press.

[44] N. Tillmann and W. Schulte. Unit tests reloaded: parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, July 2006.

[45] J. Tu, L. Chen, Y. Zhou, J. Zhao, and B. Xu. 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, pages 1–8, Washington, DC, USA, 2012. IEEE Computer Society.

[46] A. Wasylkowski, A. Zeller, and C. Lindig. 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, pages 35–44, New York, NY, USA, 2007. ACM.

[47] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering*, 2016.

[48] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200, Sept 2014.

[49] X. Xue and A. S. Namin. How significant is the effect of fault interactions on coverage-based fault localizations? In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 113–122, Oct. 2013.

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

[51] A. Zaidman, B. V. Rompaey, van Arie van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[52] M. J. Zaki and C. J. Hsiao. CHARM: an efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, pages 457–473, 2002.

[53] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 1105–1112, New York, NY, USA, 2006. ACM.

[54] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.