Monitoring Defects in Regression Testing via Spectra-Based Fault Localisation

Daniel Howden*, João F. Ferreira*†, Simon Lynch*
*School of Computing, Teesside University, England, UK
†HASLab/INESC TEC, Universidade do Minho, Portugal
dan@danhowden.co.uk, joao@joaoff.com, S.C.Lynch@tees.ac.uk

***Abstract*—Fault localisation is one of the most expensive and time-consuming tasks in program debugging. Therefore, there is a high demand for automated fault localisation methods that can reduce effort and time spent.**

This paper presents a new spectra-based fault localisation method that identifies root sources of software failures within sets of software tests and produces a summary of causative root defects that groups failures by cause. The method is implemented in a prototype tool which classifies test results based on historical failure reasons. In contrast to most fault localisation strategies, which tend to be granular and sensitive to the size of applications, this new approach uses high-level spectra data to locate multiple faults in software systems, allowing testers to observe defects both within individual tests and throughout the life cycle of an application.

We demonstrate, using three open-source applications, that this novel approach to fault localisation can eliminate the need for manual search effort in localising defects in up to 97% of cases with a single fault and up to 95% of cases with multiple faults. On average, the algorithm correctly ranks 80% of defects within its top 3 suspicious likely failure reasons where multiple failures are enabled and 61% for singularly induced failures. Furthermore, the classifications produced by the prototype tool qualitatively aid the debugging process in monitoring defect manifestation and progress over time, thus automating a previously expensive manual software verification process.

I. Introduction

Software verification is a costly but essential process where software testers normally use intuition and effort to debug root causes of failures during testing [1], [2], [3]. Often, failures are caused by a single causative root defect that manifests in multiple tests. This is particularly evident where integration testing is performed frequently as part of the regression testing process [4].

Significant effort is required to identify underlying causes and to quantify their impact [2], [3]. When this is performed manually, the causative defects can be found eventually, though it is a laborious process that must be repeated for *every* failure [5]. In an industry where software is becoming increasingly complex and fast development turnaround times are seen as essential, it is a priority that all processes involved become more assistive and automatic [6].

In this paper, we present a novel fault localisation method which assigns causative defects to failing tests based on spectra-level similarities and heuristic ratings of suspicion [5], [7], [8]. The method combines concepts ranging from spectra-based fault localisation [9], [10], [11], [12], [13], statement suspicion heuristics [5], [7], and similarity measures of execution segments [14]. In contrast to earlier work, the approach we describe here *condenses* failures into their causative root defects. Failures exhibiting similar causative defects are highlighted, reducing the effort needed to manually debug test failures.

The method consumes trace spectra to identify functional similarities between failing tests and, for any fault detected, it derives a *suspiciousness* metric for each functional 'phase' (a common sequence of execution), indicating the likelihood that the phase caused a given test failure. These metrics allow the likely causes of faults to be identified and make it possible to monitor manifestations of defects across future test executions. It is then possible to monitor tests for new, recurrent or unstable regressions.

Moreover, because the approach is based on trace spectra that collect information only at the method level — rather than at the statement level — significantly less spectra data is processed when compared with existing spectra-based fault localisation approaches. This reduces the time a debugger/tester will need to spend analysing, prioritising and understanding test results.

A. Example Scenario

Consider a system with two components (α and β) and with a test suite consisting of ten tests (testA, . . . , testJ) as shown in Fig. 1. The figure shows six failures originating from two different defects: DefectA and DefectB. It is a non-trivial task to debug the causes of these failures, since it is possible that failures exist from a previous test run. Discerning the cause of faults is demanding since the debugger can neither assume that any two faults are independent nor assume that they are caused

Test	Component	State	Causative Defect
testA	α	✔ PASS	
testB	α, β	✘ FAIL	DefectA
testC	β	✘ FAIL	DefectB
testD	β	✘ FAIL	DefectB
testE	α, β	✔ PASS	
testF	β	✘ FAIL	DefectA
testG	α, β	✘ FAIL	DefectA
testH	α	✔ PASS	
testI	α	✔ PASS	
testJ	α, β	✘ FAIL	DefectB

Fig. 1. Example scenario with multiple manifestations of multiple defects.

by a common defect. There may be up to six separate causative defects causing six unrelated failures, or one common defect causing them all. Recognising relationships between faults is further impeded when the tester is unfamiliar with the code, has limited experience in testing, or if time is constrained [15].

In Fig. 1, both DefectA and DefectB cause three failures. From this view it is impossible to observe that DefectA manifested in the previous test execution and that the *new* regression is DefectB. It becomes a decision based on development needs whether to investigate the *new, regressive* DefectB or the *existing* DefectA (i.e. if a 'known' defect would be less important that a new one).

Counter-intuitively, in this example, we *cannot* assume any relationship between defects and components. For example, a sensible inference from a tester manually debugging these results would be to assume some relation between DefectB and component α. However, such a relation may not exist. In general, it is non-trivial to verify or exclude these relationships without in-depth knowledge of the application under test; the experience of those testing software is a significant factor in the speed and effectiveness of this process [15].

Fig. 2 demonstrates the added complexity when observing failures over time. Note that even when teams record failures over time, it is often the case that *common* root causes are unknown, because as defects are debugged, they are normally fixed rapidly before their impact is understood. It is the unfixed defects that need to be tracked to identify the actual negative impact they have on the application or test results. For example, in Fig. 2 DefectA is ongoing over three iterations for testB, testF, testG and testJ. DefectB has re-occurred in testC and testD since being fixed. DefectC can be observed as an unstable test affecting the outcome of testH and testI. Even for such a small example, without the knowledge of causative defects, it is either guesswork or a laborious process to identify what defects lies behind each failure.

	Points in Time				
	t_5	t_4	t_3	t_2	t_1
testA					
testB	A	A	A		
testC	B			B	
testD	B			B	
testE					
testF	A	A	A		
testG	A	A	A		
testH	C		C		C
testI		C		C	
testJ	A	A	A		

Fig. 2. Defect progression over time. A, B and C denote defects. The point in time t_j is posterior to t_i whenever $j>i$.

The method we present in this paper tackles both issues shown in this example scenario: 1) it identifies causative root defects; and 2) it applies this knowledge to monitor such defect manifestation over time. As an example, for this particular scenario the method infers that DefectB is the actual new regression. By understanding what causes test failures and which tests are affected, the tester can spend more time on applying quality fixes to these defects rather than spending time understanding results.

In Section II we discuss related work. The fault localisation algorithm that identifies causative root defects is then presented in Section III. The algorithm is based on so-called *trace spectra* [14], [11] and consists of five sequential steps. The outcome is a set of prioritised defects; the debugger can use this information to determine the likelihood that a given defect caused a given test to fail. In Section IV, we propose a set of classifications, along with the rules used for evaluation, that can be applied to individual test results. We implemented the algorithm and the classification strategy into a prototype and we evaluated its accuracy and performance by using three open-source Java applications with supplied JUnit tests. The results are shown and discussed in Section V. Finally, we conclude the paper in Section VI, where we also provide some directions for future work.

II. Background and Related Work

This work builds on earlier fault localisation work but takes a new approach by combining concepts of spectra-based fault localisation, similarity measures and heuristic suspicion calculations.

There is a large body of work on fault localisation methods [16], [11], [8], [10], [13], [14], [9], [17], [18], [19], [20], [7], [12], [21], [22]. These works concentrate on identifying the likely locations of faults given a set of failures. This is a significant and well investigated area. For a thorough overview on the topic, we recommend the survey [23].

Given that many failures in a test suite may be caused by one defect [8], [10], [9], one must consider how to identify which tests are affected by common failure reasons, i.e. where multiple tests fail due to one defect.

Spectra-based localisation is a successful and applicable approach for fault localisation [8], [10], [13], [14], [9], [17], [20], [11], but there is little practical work into applying the information they produce other than proof of concepts such as Tarantula [8].

The inspiration for the method presented in this paper stems from the visualisation of execution traces in [14], where patterns are identified by comparing an execution trace against itself, identifying common features in phases of execution to see application flow. The properties of test executions that signal regressive faults are introduced in [9]. This work acknowledges program spectra as a significant indicator of regressions, primarily in identifying divergence between failing and passing executions. Spectra are similarly used in [17] to locate faults by selecting the most similar passing test and identifying differences that diverge from standard execution.

Another relevant and recent work is [5], where knowledge of program edits are used to present a list of these edits ranked by the suspicion of failures causation. This work differs from ours as it does not identify common causative defects within execution spectra; it assumes a program edit is at fault. The heuristic information obtained from program edit data proves to be of value in increasing the accuracy of fault localisation.

Execution slicing is used in [20] to identify differences in program executions using code statements. This method

involves considering differences between two execution slices that are then analysed to localise faults. The relevance of this work lies in the fact that it also considers the value of observing similarity/difference in execution.

Suspiciousness is calculated in [7] to discover informative metrics. Using dynamic program slicing to identify properties of execution, the method applies one or more heuristics, such as participation in failing tests, to identify suspiciousness.

The graphical representation of spectra differences are explored in [8] to highlight suspicious spectra. The method 'Tarantula' narrows the search for faults by highlighting suspicious statements. Suspicion per program statement is calculated based on its individual participation in passing/failing tests (in a similar way to [7]). This heuristic is used in other works that improve Tarantula [24], [13].

In [24], the issues in localising potential faults to individual statements within code are recognised; if several elements of code are ranked as suspicious as each other, the tester must investigate them all. By locating faults to their containing methods, less potential locations are ranked and manual effort is reduced.

Finally, spectra sequences are explored in [10] as a fault localisation method. They investigate the importance of sequences of method/statement manifestations in examining the flow of application execution. In contrast to the narrow views taken by [9], [17], [7], [12], it adopts a wider scope (as in [14], [13]) and is able to locate a defective class in 36% of cases.

III. Identification of Common Failure Causes

The algorithm that identifies common failure causes is based on *trace spectra* and consists of five sequential stages. We start by describing the requirements of *trace spectra*. The algorithm is then presented as a summary and each step is expanded in subsequent sections.

A. Trace Spectra

Trace spectra are sequences of events that occur during execution of a test [14], [11]. In this work we only consider trace spectra whose granularity is at the method level. We do not consider statement-level information such as assignments to variables [8], [5]. We assume trace spectra are ordered chronologically and contain events that have, at least, the following information:

- **Event type**: can be one of ENTRY, EXIT, CONSTRUCTOR, EXCEPTIONAL, and HANDLED EXCEPTION; these correspond to entries to methods, exits from methods, (static) constructors, uncaught exceptions and handled exceptions respectively.
- **Class name**: the canonical name of the class currently being executed.
- **Method name**: the short name of the method executed; for the event type CONSTRUCTOR, this is the same as the class name.
- **Line Number**: the line number of the method in the class currently being executed.
- **Arguments**: list of arguments that was passed to the method/constructor, if any; for the event types representing exceptions, this list would be undefined.

An example of a trace spectrum representing a call to a method $m_1()$ defined in line 42 of a class named C would be $[\text{ENTRY } C\ m_1\ 42\ null, \text{EXIT } C\ m_1\ 42\ null]$. This spectrum corresponds to a test whose control flow is restricted to method $C.m1()$.

We shall write *failing trace spectrum* to denote a spectrum associated with a failing test. Also, given a trace spectrum T_a, we write $|T_a|$ to denote its length and we write $T_a(n)$ to denote the n^{th} event in T_a.

B. Algorithm Summary

The algorithm requires a set of trace spectra, obtained from a test suite, as its input. The format of each spectrum may depend on implementation, but it should at least contain the information described in the Section III-A above. The five stages of the algorithm are as follows.

Stage 1) Build a *similarity matrix* $S_{a,b}$ for each failing spectra pair (T_a, T_b). Each $S_{a,b}$ is a matrix of size $|T_a|\times|T_b|$, where each entry $S_{a,b}(i,j)$ corresponds to a *similarity score* given to the events $T_a(i)$ and $T_b(j)$. Expanded in Section III-C.

Stage 2) Identify the *phases* of execution for each matrix. A phase is a list of consecutive trace spectra events, all with the same risk classification. The risk classification is defined as 'risky' if the event has *not* participated in a passing execution result, otherwise the event is classified as 'safe'. Our notion of phase is similar to the concept of *Dynamic Basic Block* (DBB) [10], [13]; the main difference is that DBBs consider atomic functional blocks of execution, whereas phases cannot be guaranteed to be atomic (and likely will not be). Expanded in Section III-D.

Stage 3) Merge identified phases so that each phase is unique and associated with any tests that it occurs in. Normalise phases by analysing the differences of those which are of matching length but are not identical. Expanded in Section III-E.

Stage 4) Assign a *suspicion score* to each phase/test pair using heuristics. The score represents the likelihood that the phase caused the given test to fail. Expanded in Section III-F.

Stage 5) Prioritise phases by suspiciousness: the most suspicious phase for a test is then regarded as the likely failure reason. Any other tests failing for the same reason (same phase) are considered to be failing due to a common causative defect (*reason*). Expanded in Section III-G.

C. Stage 1: Building Similarity Matrices

The algorithm must identify functional similarities in otherwise independent trace spectra. It achieves this by comparing every spectrum against every other spectrum. This builds a similarity matrix that captures the similarity between two failing trace spectra.

To create a similarity matrix $S_{a,b}$ for a given pair of spectra (T_a, T_b), each line within the spectrum T_a is compared to each other line of spectrum T_b, producing a similarity score based

on a *similarity function*. Any function that takes two spectra events and produces a real number can be used as a similarity function. An example of such a function is Algorithm 1, where method and class equality are prioritised. In practice this parametric line-scoring function can be changed depending on need so, for example, if line numbers are irrelevant, the parameters can be modified appropriately (e.g. by removing lines 20–22 in Algorithm 1).

Algorithm 1 Example of similarity function

Require: $L1$ and $L2$ are spectra events

```
1: if type(L1) == EXCEPTIONAL then
2:     if L1 === L2 then
3:         return 1
4:     end if
5:     return 0
6: end if
7: if type(L1) == type(L2) then
8:     score ← score + 2
9: else
10:    return 0
11: end if
12: if method(L1) == method(L2) then
13:    score ← score + 3
14: else
15:    score ← score − 2
16: end if
17: if class(L1) == class(L2) then
18:    score ← score + 4
19: end if
20: if lineNum(L1) == lineNum(L2) then
21:    score ← score + 2
22: end if
23: if argVals(L1) == argVals(L2) then
24:    score ← score + 2
25: end if
26: return score/13
```

The matrix creation phase produces $\frac{n\times(n-1)}{2}$ similarity matrices, where n is the number of spectra. Each matrix contains $|T_a|\times|T_b|$ entries. This stage of the algorithm is thus quadratic both in time and space, affecting the performance of the method to some extent. This is discussed in more detail in Section V.

After the similarity matrix is constructed, the matrix is pre-processed to allow extraction of phases. The goal of this pre-process step is to identify the best match per event pair to allow the likely path of execution to be extracted from the matrix. More than one likely pair may exist within the matrix, so we assume that the *first* unpaired pair is the correct pairing. Fig. 3 demonstrates this; although entry $(4,1)$ is the most similar pair on row 4, the column has already been marked in $(1,1)$ and therefore the *next highest* pair is $(4,3)$ which is marked appropriately.

The matrix pre-processing can be achieved as follows: for each column j, detect the highest scoring *unmarked* entry. Let us say that this entry is in row i; then, if the entry on row i is the highest unmarked entry, we mark this entry (i,j) as an identified pair/match. Otherwise we attempt the next highest scoring entry on the column j until these conditions are satisfied. Fig. 3 illustrates the outcome of this process. In the figure, the outlier at $(3,5)$ is marked because it is the highest value on line 5 of trace T_2 and line 3 of trace T_1. Entry $(1,4)$ is a joint highest value on the row 1 and higher than the marked entry $(5,4)$ of 0.80; however, it cannot be marked because row 1 already has an entry marked at $(1,1)$.

		$\mathbf{T_2}$				
		1	**2**	**3**	**4**	**5**
$\mathbf{T_1}$	**1**	(0.95)	0.50	0.45	0.95	0.90
	2	0.75	(0.70)	0.10	0.25	0.4
	3	0.75	0.50	0.10	0.30	(0.90)
	4	0.95	0.15	(0.80)	0	0
	5	0.75	0.50	0.80	(0.80)	0

Fig. 3. Pre-processed similarity matrix for two spectra with five lines each.

D. Stage 2: Identifying Phases

With the similarity matrix $S_{a,b}$ pre-processed, phases of execution common to traces T_a and T_b can be extracted. A phase is defined as a list of consecutive trace spectra events, all with the same risk classification. The number of phases will be no more than smallest dimension of the matrix, i.e., $\min(|T_a|,|T_b|)$.

The first step is to mark the pre-processed similarity matrix (e.g. Fig. 3) with risk classifications (as shown in Fig. 4). Recall that if a given event pair exists within the spectra of a test that *passed* then it is classified as 'safe'; otherwise it is 'risky' [7].

A phase is constructed from the pre-processed similarity matrix by iteratively considering consecutive event pairs. We assume consecutive application operation and that events are in such an order; each event entry in the matrix must have an immediate predecessor in both spectra to be in the same phase. Against this definition, in Fig. 4, entry $(3,3)$ is not a pair, and entry $(4,3)$ is not consecutive in T_1 (even though it is in T_2) so $(4,3)$ cannot be added to P_1. When a pair is not consecutive in either spectra, the current phase is concluded and a new phase is constructed. The outlier at $(3,5)$ will be a single-event phase, as no event consecutively precedes nor succeeds it.

Fig. 4 shows that the phases are constructed according to these rules: phase P_1 contains consecutive event pairs with a 'safe' classification. On the other hand, although P_3 and P_4 have consecutive event pairs, P_4 is a new phase and not included in P_3 because it has a 'risky' classification that does not match that of the preceding pair in P_3.

E. Stage 3: Merging Identified Phases

After Stage 2, there may be some phases that are functionally identical. These phases can be merged to produce a condensed set of phases. Any phases that match exactly in

		$\mathbf{T_2}$				
		1	**2**	**3**	**4**	**5**
$\mathbf{T_1}$	**1**	✔ - P_1				
	2		✔ - P_1			
	3					✔ - P_2
	4			✔ - P_3		
	5				✕ - P_4	

Fig. 4. Marked similarity matrix with risk classifications (✔ is used for 'safe' and ✕ for 'risky').

length and content are treated as identical. In this case, only one phase containing a link to all tests it affects is kept.

Where phases are identical in length but differ in content, they are evaluated to judge their *similarity*. If, according to the similarity function, all events meet a *similarity threshold*, then these phases should be merged. In that case, the difference between the components must be cancelled or normalised. To achieve this, the conflicting information is simply removed and is therefore irrelevant for future comparisons.

F. Stage 4: Calculating Suspiciousness Score

When Stage 3 is complete there will be a set of dissimilar phases of execution. This may contain all functional aspects of the system, though it is possible that some functional aspects did not result in a discoverable phase. Therefore, for each phase P and test t affected by P, the *suspiciousness heuristic* is calculated; if it reaches a given threshold value, it is recorded in the list of scored potential causative reasons. Any function that takes a phase and a test affected by that phase, and that produces a real number can be used as a *suspiciousness heuristic function*. An example of such a function is Algorithm 2, in which case *depthInSpectra* is the percentage of execution completed before this phase began.

To improve accuracy, the suspiciousness heuristic function can be provided with a list of class or method names that have been modified since the last test execution ($modifiedComponents$ in Algorithm 2). This information is often included in source control repositories, so may be available as input to the suspicion heuristic.

G. Stage 5: Prioritising Suspicious Phases

At this stage, each phase/test pair has a suspiciousness score, representing the likelihood that the phase caused the test to fail. It is now possible to prioritise defects based on the suspicion ratings. However, for an effective ranking of potential defects in any one execution, it is not enough to prioritise based only on individual suspiciousness scores as multiple tests may be affected by a single defect. It is possible that any one suspiciousness score for a phase is too low for correct prioritisation, so a combination of suspicion and the number of affected tests is used to rank based on likelihood and impact. For example, one way of calculating the priority of a defect is by using a function similar to

$$priority(P) = \frac{affectedTests(P)}{avgSuspicion(P)}$$

Algorithm 2 Example of suspiciousness heuristic function

Require: P is the phase; T is the trace from test affected by P; $modifiedComponents$ is a list of class or method names that have been modified since the last test execution

```
score ← 0
if isSafePhase(P) then
    score ← score + 4
end if
if linesAreModified(P, modifiedComponents) then
    score ← score + 1
    score ←
      count(modifiedComponents)/numLines(P) ∗ 2.25
end if
if anyLineInExceptionStacktrace(P, T) then
    score ← score + 2.25
end if
score ← score + depthInSpectra
if score > SUSPICIONTHRESHOLD then
    return score/10.5
else
    return 0
end if
```

where $affectedTests(P)$ is the number of tests affected by phase P and $avgSuspicion(P)$ is the average of all the suspicion scores associated with P.

IV. Classification of Result Types

The algorithm considers a set of failures for each execution. Over time, each execution and analysis will add or remove failing tests and causative defects. As new failures emerge it is possible that defects which previously occurred will re-emerge or identified defects will continue unfixed. Based on this historical knowledge it is possible to classify what each individual test result represents, as an extension to a simple PASS or FAIL binary classification.

This paper proposes a set of classifications, along with the rules used for evaluation, that can be applied to individual test results given information delivered by the fault localisation algorithm. The rules are presented in a prescribed, necessary, order. This order exists to ensure the correct application of classification. For example, the *existing* classification is also valid for *ongoing* tests, but the *ongoing* rule must be applied before the *existing* rule for it to make sense.

1) **System Failure -** *This reason has previously been classified as a failure due to runtime, temporal or machine issues given a pre-defined list of criteria.* In the case that the test should still be run (if it offers some business value) it is appropriate to classify the test as a known system failure and defer results indicating that failure of the test does not require immediate investigation. In situations where the system can *detect* that a failure is due to a system runtime reason, or such a defect has been manually flagged as a system issue, it prevents expending unnecessary effort on further investigation.

2) **Unstable (New Reason) -** *This test matches the* unstable *classification, but a* different *reason for failure has emerged.*
3) **Unstable -** *This test fluctuated between* PASS *and* FAIL *states more than 4 times in the previous 8 runs, for a* consistent *reason.* In some situations tests will be executed on systems that have temporal or environmental influences. It is also possible that such tests have been written badly and fail randomly (e.g., if a test generates a random input). This classification highlights tests that are known to be unstable and therefore are *less likely* to be a new regression. The numbers 4 and 8 can be adapted to specific contexts.
4) **Fixed -** *This test failed in the immediately previous run but now is in a* PASS *state.*
5) **OK -** *This test is in the* PASS *state.*
6) **Ongoing -** *This test failed sequentially in the previous 3 runs for the same reason.* A failure that persists for a long period is arguably of less value to a development team, given that it will be a 'known problem'. A classification of 'ongoing' represents a failure that is less recent than an 'existing' failure.
7) **Existing -** *This test failed in the immediately previous run for the same reason.* Failures are not likely to be rectified as soon as they are discovered. A classification of 'existing' highlights that this failure is not new but still requires investigation.
8) **New Reason -** *This test failed in the immediately previous run, but for a different reason.* A test may continue to fail when the *reason* for the failure is new; an example is when an attempt to rectify a known failure has resulted in further defects.
9) **Recurrent -** *This test failed previously for this reason after since being fixed.* This classification indicates a test is failing for a reason that is not new, but is a re-emergence of a defect that was fixed previously. This may highlight that a fix has uncovered a further known defect.
10) **New -** *This test has never failed for this reason.* It is important to observe when a failure is truly 'new' in a given execution result. In regression testing is it common to find test failures which have not yet been fixed but are nevertheless known (and ignored) failures. Given that the purpose of regression testing is to discover if new failures have been introduced, classifying results as *truly new* failures is of significant value.

V. EVALUATION

We have developed a prototype tool which implements the approach described in earlier sections and have evaluated its accuracy and performance by experimentation. The tool manages both the execution of tests and the algorithm to condense and visualise the state of a regression test suite. The tool *(1)* executes tests, *(2)* uses the algorithm shown in Section III to group causative defects, *(3)* classifies test results as discussed in Section IV, and *(4)* visualises these results.

Built in Java, the prototype uses the JUnit test infrastructure to augment the execution of tests and to acquire test data. AspectJ is used to instrument applications to acquire trace spectra [25]. AspectJ is used to insert trace logging information into existing applications, adding debugging capabilities.

A. Test Setup

We used Algorithm 1 as the similarity function and Algorithm 2 as the suspiciousness heuristic function. To acquire sensible test data for the algorithm and classification strategy, three open-source Java applications with supplied JUnit tests were sourced. These non-trivial applications were chosen at a variety of test suite sizes to allow sensible practical evaluation (see Fig. 5). The applications were augmented with the same AspectJ instrumentation to acquire consistent formatted trace spectra (as specified in III-A).

Project Name	Total #Tests	Existing Failures
JGAP (v. 3.63)	1,366	12
MARC4J (v. 2.6.0)	41	0
Egothor (v. 3.1.8)	231	0

Fig. 5. Open-source applications used in the evaluation. JGAP contains 12 failures due to the execution platform used.

The first application is the *Java Genetic Algorithms Package* (JGAP) which contains computationally complex and deeply iterative functionality tested by 1,366 tests [26]. *MARC4J* [27] is a small parsing application with 41 JUnit tests. The third application is *Egothor*, which contains 231 JUnit tests [28]. This parsing application provided significant amounts of repetitive trace useful as an intensive performance test.

MARC4J and Egothor provided tests that had a 100% passrate. Tests for JGAP have 12 failures by default due to the execution platform used; these failures are denoted as *JGAP i* and provide a useful real-world test.

For each of the applications, three defects were induced: **induced defect *i*** introduced an incorrect operand, **induced defect *ii*** inserted an un-handled exception, and **induced defect *iii*** simulated missing/deleted code. As JGAP contained an existing failure, this was left in place and an incorrect operand and un-handled exception were induced as *JGAP ii* and *JGAP iii* respectively.

For each application, induced defects were tested in two different ways. First, each induced defect was tested individually for a single-failure accuracy measure. Second, for each application *all* defects were activated to test the accuracy of diagnosing multiple defects within one execution. In addition, the priority of the diagnosed defects was recorded to test the prioritisation strategy.

B. Fault Localisation Accuracy

In *all* cases, the algorithm highlighted that the faulty class was executed either (a) as a probable reason or (b) included in the list of reported prioritised probable causes within a phase (observed as a phase of execution relevant to a test). This demonstrates that the algorithm extracts relevant aspects of execution[1].

1) Single Defects: Fig. 6 shows results from running the prototype against a version of the applications with only one manually induced failure at a time. In situations where the

[1]All the percentages shown in this section are based on the average of the results shown in the tables.

algorithm did not report the likely causative reason(s) with 100% accuracy, the algorithm was repeated with additional knowledge about program edits in class/method components to further increase the accuracy of the suspicion heuristic (as discussed in III-F).

Given the three induced defects for each application, it can be seen that on average, the algorithm predicted the actual causative defect as the *most likely reason* in 55% of the tests. It assigned the actual causative defect within the top 3 likely failure reasons for a test in 61% of cases. Given prior knowledge about modified classes and methods, the algorithm was accurate in calculating the most likely causative defect in 55% and 97% of cases, respectively.

For test *JGAP iii*, the 2 of 5 failures not correctly classified were higher-class methods that called the faulty method (it instantiated the faulty object) and therefore would likely lead to discovery by a tester, though it is not measurable against the criterion here.

For test *Egothor ii*, the causative defect was listed second in suspiciousness for each test failure it caused, demonstrating that even when not perfectly identifying causes, the algorithm offers significant value.

2) Multiple Defects: In this test, the defects were enabled simultaneously to evaluate the algorithm's capability in detecting the presence of multiple defects in one execution.

Fig. 7 shows the results from a test where all three failures were induced in each application. The algorithm correctly identified the causative defect as the *most likely* in 68% of cases and in the top 3 likely causative defects for 80% of induced defects. With class or method program edit information, the algorithm was accurate in identifying the most likely reason in 68% and 86% of cases respectively. The algorithm respectively ranked the actual causative defect within the top three likely reasons in 80% and 95% with known program edits.

Note that there is not a one-to-one relationship between failures and induced defects. One defect may prevent another from manifesting (as in MARC4J) and defects may cause the same test(s) to fail (as in Egothor). Therefore the total number of failures caused in Fig. 6 may not match that in Fig. 7.

The real value of providing the algorithm with information of program edits is when multiple changes have taken place. This is because it will preferentially score the *disparate* phases that have been identified should phases common to all failures exist; there will be many more phases identified so the ability to assign a higher suspicion to these *known changes* will increase the priority within all scoring functions. This is why Fig. 7 shows an improvement in the Egothor application when known changes to methods are passed to the suspiciousness heuristic function (in this case, Algorithm 2).

C. Defect Prioritisation Accuracy

Fig. 8 shows the accuracy of the algorithm in prioritising identified causative defects in an appropriate order. The optimal priority for any induced defect is clearly 1. However, the algorithm can significantly reduce search space (and required analysis time for testers) when defects are successfully identified even when their prioritised value is less than 1.

For test *MARC4J iii*, the different configurations of the tests (with/without known program edits) improved the overall ranking of the causative defect within the 'global' defect priority list. This serves to highlight the significance of the suspicion heuristic in deriving accurate priorities.

For the cumulative test on MARC4J, defects ii and iii did not manifest as i terminated test execution, so the prioritisation of this defect was nevertheless correct.

The prioritisation is not optimal for the cumulative test of Egothor, with ii and iii ranking as 10th and 32nd priority, respectively. This is due to the complex iterative nature of the application producing many separate phases that scored higher on both suspicion and priority. This demonstrates a lack of direct correlation between the accuracy in identifying the most likely failure cause and the prioritisation within the overall test execution.

D. Result Classification Accuracy

To verify the correctness of the classification strategy, the failures of the JGAP application (i, ii and iii) were enabled simultaneously. A selection of failing tests were observed and recorded in Fig. 10 with their identified causative defects. Failed defect identifications were not used. Also, to achieve classifications such as *Fixed* or *Recurrent*, the manifestation of defects within individual tests was manually induced.

Based on the classification rules described in Section IV, there is a set of classifications that can be deterministically predicted. There should be no margin of error within these classifications if we assume the implementation is sound. The classifications in Fig. 11 are the results presented by the classification strategy described. For each test, the correct classification that we expected was assigned. Recall that test *JGAP i* was present by default as a platform-related failure. As the algorithm was instructed to classify this defect as a system failure, that classification was applied for *testZ*.

The classification strategy is rule-based and does not change depending on the application or environment under test. As such, the rules that have been implemented are static and are perfectly accurate in applying the correct rules as described in Section IV. This accuracy may lessen should adaptive rules be used. Figure 12 shows a screenshot of the current prototype result view. The screenshot corresponds to the situation shown in Fig. 11.

E. Speed of Analysis

The additional instrumentation to acquire spectra is not part of the algorithm and thus is not considered as a measurable aspect of the algorithm. However, the additional time necessary to perform the analysis will define the applicability of such a method.

It is difficult to quantify the time it takes a human tester to reach similar conclusions to the algorithm, as this varies by expertise, complexity of software and experience. This is nevertheless directly correlated to the number of failures, number of tests and complexity of the software under test. Fig. 9 shows the time to execute the algorithm within the prototype. It compares the time taken for test execution and spectra acquisition without running the algorithm ('without

				Causative defects assigned correctly	
Application	**Induced fault**	**Failures caused**	**Known program edits**	*Within top 3*	*1st most Suspicious*
JGAP	i	12		100%	100%
JGAP	ii	5		100%	100%
JGAP	iii	5		60%	60%
JGAP	iii	5	at class level	60%	60%
JGAP	iii	5	at method level	100%	100%
MARC4J	i	8		100%	100%
MARC4J	ii	27		100%	100%
MARC4J	iii	6		34%	0%
MARC4J	iii	6	at class level	100%	0%
MARC4J	iii	6	at method level	100%	100%
Egothor	i	11		0%	0%
Egothor	i	11	at class level	55%	0%
Egothor	i	11	at method level	100%	82%
Egothor	ii	3		0%	0%
Egothor	ii	3	at class level	100%	0%
Egothor	ii	3	at method level	100%	100%
Egothor	iii	16		56%	38%
Egothor	iii	16	at class level	56%	38%
Egothor	iii	16	at method level	88%	88%

Fig. 6. Fault localisation accuracy with single induced failures.

				Causative defects assigned correctly	
Application	**Induced faults**	**Failures caused**	**Known program edits**	*Within top 3*	*1st most Suspicious*
JGAP	i, ii, iii	22		91	91
MARC4J	i, ii, iii	27		100	100
Egothor	i, ii, iii	27		48	14
Egothor	i, ii, iii	27	at class level	48	14
Egothor	i, ii, iii	27	at method level	93	67

Fig. 7. Fault localisation accuracy with multiple induced failures.

Application	**Induced fault**	**Priority**
JGAP	nil	1
JGAP	ii	1
JGAP	iii	4
JGAP	all	1, 2, N/A
MARC4J	i	1
MARC4J	ii	1
MARC4J	iii	6
MARC4J	all	1
Egothor	i	3
Egothor	ii	6
Egothor	iii	1
Egothor	all	1, 10, 32

Fig. 8. Defect prioritisation accuracy.

Application	**Failures**	**Time without analysis (sec)**	**Time with analysis (sec)**	**Difference (sec)**
JGAP i	12	67.6	86.0	+ 18.3
JGAP ii	5	62.4	66.1	+ 3.7
JGAP iii	5	62.1	79.3	+ 17.1
JGAP all	22	70.6	179.7	+ 109.0
MARC4J i	8	13.5	29.4	+ 15.9
MARC4J ii	27	1.2	206.2	+ 205.0
MARC4J iii	6	10.3	77.8	+ 67.6
MARC4J all	27	15.1	196.2	+ 181.1
Egothor i	11	10.5	11.4	+ 0.9
Egothor ii	3	8.3	22.5	+ 14.2
Egothor iii	16	10.4	15.2	+ 4.8
Egothor all	27	9.0	302.4	+ 293.5

Fig. 9. Time taken for additional analysis.

analysis'); to localise faults or classify results against the time taken to run analysis ('with analysis'). 'Difference' shows the *additional* length of time before the test results are consumable when compared to not executing this algorithm.

The algorithm adds a noticeable amount of time onto the test execution. It is possible to create this analysis step as a background task to actual test execution, which would augment additional data once it has been calculated. This would allow testers to access and view results immediately should that be necessary. The additional time taken is specific to this prototype implementation and may be reduced with further optimisation.

Fig. 9 shows that the additional analysis time rises significantly when all defects are enabled. Note the number of test failures for such tests (22, 27 and 27 respectively). The nature of the algorithm means that the number of failures dictates the number of test comparisons that take place (231, 351 and 351 similarity matrices, respectively). The *length* of spectra within these traces also affects the computational effort needed to perform the comparison, with a spectra with 483,839 lines in MARC4J ii being compared with another spectra of 478,021 lines resulting in 2.6 billion line similarity scoring operations.

	Points in Time							
	t_8	t_7	t_6	t_5	t_4	t_3	t_2	t_1
Tests	E	D				E	E	
	A	D	A	A	A	A	A	A
		D						
	B	D		B		B		B
		D					D	
	F	D					D	
		D					D	
	A	D	A	A	C	C	C	C
		D						
testZ	G	G	G	G	G	G	G	G

Fig. 10. Actual manifestation of defects within a test set over time for the application JGAP. Test *testZ* corresponds to test *JGAP i*.

	Points in Time							
	t_8	t_7	t_6	t_5	t_4	t_3	t_2	t_1
Tests	Rec	Sys			Fixed	New	New	
	Ong	Sys	Ong	Ong	Ong	Ong	Ong	Ong
		Sys						
	Uns	Sys		Uns		Uns		Uns
		Sys					Sys	
	New	Sys					Sys	
		Sys					Sys	
	Exi	Sys	Exi	New	Ong	Ong	Exi	Exi
		Sys						
testZ	Sys	Sys	Sys	Sys	Sys	Sys	Sys	Sys

Fig. 11. Classification of defects within a test set over time for the application JGAP (see Fig. 10). **Key: Rec**urrent, **Sys**tem, **Ong**oing, **New**, **Exi**sting, and **Fixed**.

F. Qualitative Benefits

We have demonstrated that our approach can identify causative root defects of a set of failures to a high accuracy. The algorithm can calculate three causative root defects in an application with over 1,300 tests, with 22 failures, in 3 minutes. It is highly unlikely that even an experienced tester could perform as well.

The utility of the method nonetheless increases rapidly as the experience of a tester falls and/or the complexity of an application increases. There is no correlation in the results between accuracy, speed and size of an application. In fact, it emerged that medium-sized Egothor produced the least accurate results due its *iterative complexity* of computation, not application size or test suite size. Additionally, the size of the application bears little correlation to the additional time required to perform the analysis; on average, analysis of MARC4J was slower than Egothor.

When the algorithm correctly identifies the correct causative root defect as a likely failure cause, it has at best reduced the search space to a single method within the application. At worst, it has reduced the search space to all methods that occur in the phase. This reduces the search space almost completely by pinpointing in many cases the method-level location of a causative defect or defects.

VI. Conclusion and Future Work

The method presented in this paper is proven to localise regressive failures in a software test environment. In some cases the algorithm is able to pinpoint a method-level location of an induced defect with a 100% accuracy, both with and without prior knowledge about changes to the source code. In other cases it is able to prioritise potential faults as issues to investigate with useful accuracy.

The method is parametric, since the line similarity, suspicion heuristic, and defect prioritisation functions can be altered depending on development needs (i.e. if the definitions of 'priority', 'similarity' or 'suspicion' change). The parametric nature of the method also lends itself for continued refinement. It is observed that accuracy falls when the tested application produces repetitive trace (e.g. Egothor). It is hypothesised that by reducing the number of similar execution phases, the accuracy would be further improved.

The major novel outcome of this work is an ability to condense a set of given failures and to identify common causative root defects, allowing the classification of execution results. Debugging efforts can be guided and prioritised based on this information. In many cases the algorithm pinpoints the method-level location of a defect, qualitatively aiding a software tester by automating a laborious part of the testing process. Integration with further tools, such as software versioning and change management can aid this process. By utilising and comparing *all* available trace spectra, the method is implicitly context-aware of execution flow, therefore supplementing the accuracy afforded by other fault localisation techniques. This knowledge allows application of the presented result classification strategy to better define the state of a test suite by monitoring defect manifestation over the development life cycle. By reducing the search space significantly for real failures in real applications (e.g. *JGAP i*), we have presented a practical method that can automate an expensive task in software regression testing.

A. Future Work

While the algorithm has been tested on a variety of different sized applications (from 10-line spectra to 600-line spectra) further testing is desirable to improve comparison with alternative methods and corroborate findings. A greater range of open-source applications provided with unit test frameworks would facilitate this.

Moreover, the method presented is designed to work with Object-Oriented (OO) applications. In the situation that similar levels and structures of execution spectra is available, it is likely that the solution would be of some value in non-OO applications though this requires further investigation.

Finally, the classification method in use is currently a pseudo-adaptive strategy. It is possible that a machine learning strategy could classify both machine failures and unstable test-cases based on learned characteristics of test runs and the environment(s) in which they are run. This would produce a truly adaptive solution that would evolve to be specific to any given test scenario.

Acknowledgements

Thanks to Jonathon Read for valuable feedback on this paper.

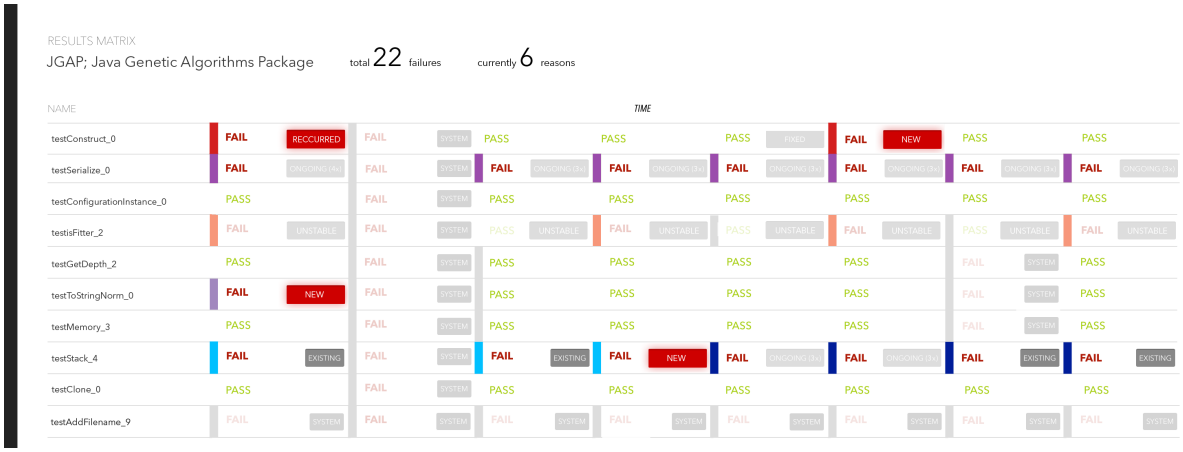

Fig. 12. Screenshot of the current prototype result view.

REFERENCES

[1] E. Dustin, T. Garrett, and B. Gauf, *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education, 2009.

[2] G. Myers, C. Sandler, T. Badgett, and T. Thomas, *The Art of Software Testing*. Business Data Processing: A Wiley Series, Wiley, 2004.

[3] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology, RTI Project*, vol. 7007, no. 011, 2002.

[4] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 119–129, ACM, 2002.

[5] L. Zhang, M. Kim, and S. Khurshid, "Localizing failure-inducing program edits based on spectrum information," in *27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 23–32, 2011.

[6] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering*, pp. 85–103, IEEE Computer Society, 2007.

[7] H. Pan and E. H. Spafford, "Heuristics for automatic localization of software faults," Tech. Rep. SERC-TR-116-P, Purdue University, 1992.

[8] 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, (New York, NY, USA), pp. 467–477, ACM, 2002.

[9] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, pp. 171–194, 2000.

[10] 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, (Berlin, Heidelberg), pp. 528–550, Springer-Verlag, 2005.

[11] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, (New York, NY, USA), pp. 342–351, ACM, 2005.

[12] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *ACM Software Engineering Notes*, pp. 432–449, Springer-Verlag, 1997.

[13] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), pp. 82–91, ACM, 2006.

[14] B. Cornelissen and L. Moonen, "Visualizing similarities in execution traces," in *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pp. 6–10, 2007.

[15] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance release," in *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, (Washington, DC, USA), pp. 464–474, IEEE Computer Society, 1996.

[16] L. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with Tarantula," in *The 18th IEEE International Symposium on Software Reliability (ISSRE)*, pp. 137–146, 2007.

[17] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th International Conference on Automated Software Engineering (ASE 2003)*, pp. 30–39, IEEE Computer Society, 2003.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 15–26, ACM, 2005.

[19] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, (New York, NY, USA), pp. 1–10, ACM, 2002.

[20] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault localization using execution slices and dataflow tests," in *Sixth International Symposium on Software Reliability Engineering*, pp. 143–151, 1995.

[21] T. Ball and J. R. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, (Washington, DC, USA), pp. 46–57, IEEE Computer Society, 1996.

[22] J. S. Collofello and L. Cousins, "Towards automatic software fault location through decision-to-decision path analysis," in *AFIPS Conference Proceedings. 1987 National Computer Conference*, pp. 539–544, 1987.

[23] W. E. Wong and V. Debroy, "A survey of software fault localization," Tech. Rep. UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, 2009.

[24] H. de Souza and M. Lordello Chaim, "Adding context to fault localization with integration coverage," in *28th International Conference on Automated Software Engineering (ASE 2013)*, pp. 628–633, Nov 2013.

[25] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *ECOOP 2001 Object-Oriented Programming* (J. Knudsen, ed.), vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–354, Springer Berlin Heidelberg, 2001.

[26] K. Meffert, "JGAP - Java Genetic Algorithms and Genetic Programming package." Online, Last accessed: 6 May 2014. Available online at http://jgap.sf.net.

[27] B. Peters, "MARC4J - MAchine Readable Cataloguing For Java." Online, Last accessed: 6 May 2014. Available online at https://github.com/marc4j/marc4j.

[28] L. Galambos, "Egothor." Online, Last accessed: 6 May 2014. Available online at http://www.egothor.org/product/egothor2/index.html.