Human-in-the-Loop Fault Localisation Using Efficient Test Prioritisation of Generated Tests

Gabin An
School of Computing
KAIST
Daejeon, Republic of Korea
agb94@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Many existing fault localisation techniques become less effective or even inapplicable when not adequately supported by a rich test suite. To overcome this challenge, we present a human-in-the-loop fault localisation technique, QFID, that works with only a small number of initial failing test cases. We augment the failing test cases with automatically generated test data and elicit oracles from a human developer to label the test cases. A new result-aware test prioritisation metric allows us to significantly reduce the labelling effort by prioritising the test cases to achieve maximum localisation accuracy. An evaluation with EvoSuite and our test prioritisation metric shows that QFID can significantly increase the localisation accuracy. After only ten human labellings, QFID can localise 27% and 66% of real-world faults in DEFECTS4J at the top and within the top ten, respectively. This is a 13 and 2 times higher performance than when using the initial test cases. QFID is also resilient to human errors, retaining 80% of its acc@1 performance on average when we introduce a 30% error rate to the simulated human oracle.**

***Index Terms*—Fault Localisation, Test Prioritisation, Diagnosability**

I. INTRODUCTION

Fault localisation techniques aim to reduce the cost of software debugging by automatically identifying the root cause of the observed test failures [1]. Some of the most widely studied fault localisation techniques are dynamic analyses that depend on test cases. For example, Spectrum Based Fault Localisation (SBFL) [2]–[4] exploits the differences in program spectrum between passing and failing test executions to identify the location of the fault, while Mutation Based Fault Localisation (MBFL) [5]–[8] focuses on how passing and failing test cases react differently to program mutations.

In order to achieve effective and efficient dynamic fault localisation, it is crucial to have a rich and strong test suite with a sufficient diagnostic capability [9]. The test suite used by fault localisation techniques should be executed in a way that extracts the maximum amount of information about the differences between passing and failing executions. The diagnostic capability of a test suite depends not only on its composition [9], [10] but also on the order of test execution [11]–[14].

A problem arises when no such test suite is available. A developer may have to debug a legacy system without a test suite, based on a recent bug report only; it is also possible that a test suite exists but is not strong or diverse enough. When given only a small number of failing test cases, most of the existing fault localisation techniques that depend on dynamic features become either much less effective or simply inapplicable. For example, the suspiciousness score of MUSE, an MBFL technique, ceases to be meaningful as it depends on how mutants affect passing test executions to narrow down the list of suspicious program elements. Similarly, Tarantula [2], an SBFL formula, is not computable by definition without any passing test executions. Automated test data generation alone is of little help, as it can only capture the current faulty behaviour with regression test oracles.

In this paper, we aim to solve the problem of weak or non-existent test cases using the combination of automated test data generation, human oracles, and highly effective test prioritisation metrics. We focus on prioritising test cases in a way that achieves maximum localisation accuracy with the minimum number of test cases. Minimising the number of test cases required for localisation also allows humans to act as test oracles for automatically generated regression test cases. We carefully analysed existing prioritisation techniques for fault localisation and designed new normalised prioritisation metrics, `Split` and `Cover`, based on the insights from the analysis as well as an empirical comparison of existing metrics. An empirical evaluation shows that the new metrics, `Split` and `Cover`, perform better or on par with all existing metrics as stand-alone prioritisation metrics for fault localisation. When hybridised, they outperform all existing prioritisation techniques.

Based on the hybrid prioritisation metric, we propose QFID, a framework for Query-based human-in-the-loop Fault localisation using Diagnosability-aware prioritisation of test cases. Starting from a small number of failing test cases, QFID first automatically generates a number of regression test cases to augment the inadequate test suite. Subsequently, QFID prioritises the generated test cases, aiming to elicit the fewest test labels from a human while achieving the highest localisation accuracy. We empirically evaluate QFID, using EvoSuite [15] as the test data generation tool to localise real-world faults in the DEFECTS4J benchmark [16], starting from only the failing test cases. When generating tests for ten minutes and simulating a perfect human oracle, QFID achieved acc@1 of 42 and acc@10 of 97, out of 138 studied

faults starting from only a single failing test case, within ten oracle elicitations. In total, QFID achieves acc@1 of 52 and acc@10 of 130, out of 196 studied faults when using only the failing test cases as the starting point.[1] We also show that QFID is resilient to human errors by introducing random error rates.

The main contributions of this paper are as follows:

- We survey test prioritisation and test suite diagnosability metrics for SBFL. We empirically compare how quickly existing prioritisation techniques can improve SBFL accuracy using the DEFECTS4J benchmark.
- We propose a novel coverage-based test prioritisation method for SBFL that significantly outperform existing techniques.
- We design a human-in-the-loop iterative fault localisation framework, QFID, that combines three key components: automated test data generation, test prioritisation, and fault localisation.
- We empirically investigate the feasibility of QFID using EvoSuite and DEFECTS4J. QFID can localise 27% of the studied faults at the top by querying the human oracle ten times, while the initial failing tests could localise only 2% of the faults at the top.

The rest of the paper is organised as follows. Section II presents the basic notations that will be used throughout the paper, as well as the fundamental concepts in SBFL. Section III surveys existing test prioritisation techniques for fault localisation and empirically compares their performance using the faults and human-written test cases in DEFECTS4J. Section IV presents our novel test prioritisation metrics, `Split` and `Cover`, as well as QFID, our human-in-the-loop iterative fault localisation framework. Section V contains the research questions and the results of the empirical evaluation of QFID. Section VI describes the related work, and Section VII considers threats to validity. Finally, Section VIII concludes.

II. BACKGROUND

This section introduces the basic notation and the background of SBFL techniques.

A. Basic Notation

Given a program P, let us define the following:

- Let $E = \{e_1, e_2, ..., e_n\}$ be the set of program elements that consist P, such as statements or methods.
- Let $T = \{t_1, t_2, ..., t_m\}$ be the test suite for P, and C_T the $m \times n$ coverage matrix of T:

$$C_T = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \in \mathbb{B}^{m \times n}$$

  where $C_T[i,j] = a_{ij} = 1$ if t_i covers e_j and 0 otherwise.
- Given an arbitrary test case t, let $c_t \in \mathbb{B}^{1 \times n}$ be the coverage vector of t, where $c_t[j] = 1$ if t executes e_j, and 0 otherwise.
- Let $R : T \to \mathbb{B}$ be the function that maps a test in T to its result: $R(t) = 0$ if t reveals a fault in P, and 1 otherwise. Subsequently, the set of failing tests, T_f, can be defined as $\{t \in T | R(t) = 0\}$. The program P is *faulty* when T_f is not empty.

[1]On average, the subject DEFECTS4J fault has 2.2 failing test cases.

B. Spectrum-based Fault Localisation

Spectrum-based Fault Localisation (SBFL) is a statistical approach for automated software fault localisation [1]. It utilises *program spectrum*, a summary of the runtime information collected from the program executions, to pinpoint the faulty program elements. A representative program spectrum is based on the code coverage of test cases, which consists of four values: $(e_p, n_p, e_f, n_f) = (N_{11}, N_{01}, N_{10}, N_{00})$. Formally, given a test suite T, the spectrum values for a program element $e_j \in E$ can be defined as follows:

$$N_{ab} = |\{t_i \in T | C_T[i,j] = a \wedge R(t_i) = b\}|$$

SBFL statistically estimates the suspiciousness of each program element based on the following rationale: *"The more an element is correlated with failing tests and less with passing ones, the more suspicious the program element is, and vice versa"*. A risk evaluation formula $S : \mathbb{I}^4 \to \mathbb{R}$ is a formula that converts program spectra to suspiciousness scores. Many risk evaluation formulas [2]–[4], [17] have been designed to implement this core idea. For example, Ochiai [17], one of the most widely studied risk evaluation formulas, computes the suspiciousness of a program element e_j as follows:

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

where (e_p, n_p, e_f, n_f) is a program spectrum of e_j. Fig. 1 illustrates a concrete example of calculating Ochiai scores from the coverage matrix and test results.

C. Ambiguity Groups

An *ambiguity group* [18], [19] is a set of program elements that are only executed by the same set of test cases. Given a program P and its elements E, we define $AG(T)$ as a set of such ambiguity groups under the test suite T. The size of $AG(T)$ is equal to the number of unique columns in C_T. More formally, $AG(T)$ is a partition [20] of the set E that satisfies the following properties:

1) $\forall e \in E. \exists g \in AG(T). e \in g$
2) $\forall g \in AG(T). \forall e_i, e_j \in g. C_T[:,i] = C_T[:,j]$
3) $\forall g_1, g_2 \in AG(T). g_1 \neq g_2 \wedge e_i \in g_1 \wedge e_j \in g_2 \implies C_T[:,i] \neq C_T[:,j]$

Elements in the same ambiguity groups are assigned the same suspiciousness score, since their program spectra are identical. For example, in Fig. 1, $AG(\{t_1, t_2, t_3\})$ is $\{\{e_1\}, \{e_2\}, \{e_3, e_4\}, \{e_5\}\}$, and all the elements in an ambiguity group are tied in the final ranking and hence cannot be uniquely diagnosed as faulty. Therefore, the performance of an SBFL technique increases when there are more ambiguity groups, and their size is smaller.

Program Elements		e_1 (faulty)	e_2	e_3 (faulty)	e_4	e_5	
Tests	t_1					•	Pass
	t_2	•		•	•		Fail
	t_3		•	•	•		Fail
Spectrum	e_p	0	0	0	0	1	
	n_p	1	1	1	1	0	
	e_f	1	1	2	2	0	
	n_f	1	1	0	0	2	
Ochiai		0.71	0.71	1.00	1.00	0.00	
Rank		4	4	2	2	5	
Additional Tests	t'_1					•	?
	t'_2	•			•		?
	t'_3	•	•	•	•		?
	t'_4			•			?

Fig. 1. A simple motivating example (The dots (•) show the coverage relation, and the ranks are computed using the max tie-breaker.)

III. Study of Existing Test Prioritisation for Fault Localisation

Test case prioritisation identifies an order of test cases that achieves certain goals, such as code coverage, as early as possible [21]–[23]. Given a set of test case T, *coverage-based* test case prioritisation aims to find such optimal permutations of T using the test coverage data.

Our aim is to find an effective coverage-based test prioritisation technique for SBFL, i.e., permutate test cases to localise the faulty program elements as early as possible. For this, we should be able to measure the amount of diagnostic information a new test case can add to already executed tests. For example, in Fig. 1, consider the candidate additional test cases $\{t'_1, t'_2, t'_3, t'_4\}$. The execution of t'_3 does not alter the final ranking regardless of its result, as t'_3 does not increase the diagnosability of the executed subset of test cases. Although various test case prioritisation methods for fault localisation and test suite diagnosability metrics have been proposed, these techniques, to the best of our knowledge, have been extensively compared to each other using the same benchmark. This section briefly surveys the existing techniques and empirically compares them using DEFECTS4J, a collection of real-world faults in Java programs. We limit the scope to techniques that only use test coverage and results.

A. Test Prioritisation and Diagnosability Metrics for SBFL

We survey coverage-based test prioritisation techniques, as well as test suite diagnosability metrics that are designed to aid SBFL use. As a baseline, we include two widely studied test case prioritisation techniques, total and additional coverage [21], [22]. In addition to these two techniques, we identified seven different methods from the relevant literature published after 2010. We hereby introduce them using the unified metric notation, $f(T, t)$, which denotes the estimated diagnosability gain brought by a newly chosen single test case t to a set of already executed test cases T.

1) Test Case Prioritisation Metrics: Rothermel et al. [21] proposed two coverage-based test case prioritisation methods: total coverage and additional coverage, which greedily select the test case that has the highest coverage of all program elements and uncovered program elements, respectively. The rationale is that, by increasing coverage as early as possible, we hope to increase the amount of information provided to SBFL techniques.

$$\texttt{Total}(T, t) = |\{e_j \in E | c_t[j] = 1\}|$$

$$\texttt{Add}(T, t) = |\{e_j \in E | c_t[j] = 1 \wedge \forall t_i \in T. C_T[i, j] = 0\}|$$

Renieris and Reiss proposed Nearest Neighbours fault localisation [24], which selects the passing test cases whose execution trace is the closest to the given failing execution. They use a program dependency based metric to measure the proximity. Motivated by this work, Bandyopadhyay et al. put weights to test cases using the average of Jaccard similarity between the coverage of a given test case and the failing ones [25]. Although Bandyopadhyay et al. do not use this metric to prioritise test cases, we include the metric since it aims to measure the diagnosability of individual test cases. We call this metric as Prox hereafter.

$$\texttt{Prox}(T, t) = \frac{\sum_{t_i \in T_f} Jaccard(C_T[i, :], c_t)}{|T_f|}$$

Hao et al. [26] select a representative subset of test cases from a given test suite to reduce the oracle cost (i.e., to reduce the number of outputs to inspect). Among the three coverage based strategies introduced, S1, S2, and S3, the most effective one for fault localisation was S3: it takes into account the relative importance of ambiguity groups, which in turn is defined as the ratio of failing tests that execute each group. S3 assigns higher priority to tests that can more evenly split the more important ambiguity groups:

$$\texttt{S3}(T, t) = \sum_{g \in AG'(T)} pri(g) \cdot div(t, g)$$

AG' denotes a variation of ambiguity group, where each group includes program elements that share the same spectrum values; $pri(g)$ is the ratio of failing tests that cover g, i.e., $\frac{e_f}{e_f + n_f}$. Finally, when t is used to divide g based on coverage, $div(t, g)$ denotes the size of the smaller of the divided subsets, i.e., $min\left(|\{e_j \in g | c_t[j] = 1\}|, |\{e_j \in g | c_t[j] = 0\}|\right)$.

RAPTER [19] prioritises test cases by the amount of ambiguity group reduction achieved by individual test cases. While the aim of reducing ambiguity group is similar to S3, RAPTER considers only the size of ambiguity groups, and not the test results.[2] Here, $p(g) = |g|/n$ is the probability that g contains a faulty element, and $\frac{|g|-1}{2}$ refers to the expected wasted effort if a developer considers program elements in g randomly.[3]

$$\texttt{RAPTER}(T, t) = -\sum_{g \in AG(T \cup \{t\})} p(g) \cdot \frac{|g| - 1}{2}$$

FLINT [11] is an information-theoretic approach that formulates test case prioritisation as an entropy reduction process.

[2]S3, on the other hand, considers test results via $pri(g)$.

[3]Note that we take the multiplicative inverse of the original RAPTER and FLINT to make a higher score mean a higher diagnosability gain.

In FLINT, a test case is given higher priority when it is expected to reduce more entropy (H) [27] in the suspiciousness distribution across program elements. The expected entropy reduction of each test case is predicted based on the conditional probability of the test case failing; the probability of a new test case failing is approximated as the failure rate observed so far.

$$\texttt{FLINT}(T,t) = -\alpha \cdot H(P_f) - (1-\alpha) \cdot H(P_p)$$

Here, α is the observed failure rate, $|T_f|/|T|$, and P_p and P_f are the suspiciousness distributions of $T \cup \{t\}$, computed under the assumption that t passes and fails, respectively. The suspiciousness distributions are computed by normalising the Tarantula [2] scores.[3]

2) Test Suite Diagnosability Metrics: This section surveys test suite diagnosability metrics. By design, a test suite diagnosability metric measures the diagnostic capability of a test suite. However, a diagnosability metric F can also be used to quantify the diagnosability gain of an individual test, t, by computing the difference between an original test suite and the new test suite, i.e., $f(T,t) = F(T \cup \{t\}) - F(T)$. In this way, a test-suite-level diagnosability metric can be used to prioritise individual test cases for fault localisation.

Baudry et al. [10] analysed the features of a test suite that are related to the fault diagnosis accuracy and introduced the *Test-for-Diagnosis (TfD)* metric that measures the number of ambiguity groups (referred to as *Dynamic Basic Blocks (DBB)* in their paper). The authors proposed composing a test suite that maximises the number of DBBs to improve its diagnostic capability.

$$\texttt{TfD}(T) = |AG(T)|$$

EntBug [28] evaluates a test suite based on its coverage matrix density, which is defined as the ratio of ones in the coverage matrix. EntBug augments an existing test suite with additionally generated test cases with the goal of balancing the density of the coverage matrix to 0.5.

$$\texttt{EntBug}(T) = 1 - |1 - 2 \cdot \rho(T)|$$

where $\rho(T) = \sum C_T[i,j]/(|E| \cdot |T|)$. Note that this definition is the normalised version [9] of EntBug.

More recently, Perez et al. propose DDU [9], a test suite diagnosability metric for SBFL, that combines three key properties, **d**ensity, **d**iversity, and **u**niqueness, all being properties that a test suite should exhibit to achieve high localisation accuracy. The density component is identical to EntBug, while the uniqueness component is the ratio of the number of ambiguity groups over the number all program elements, i.e., $|AG(T)|/|E|$. Lastly, the diversity component is designed to ensure the diversity of test executions, i.e., the contents of the rows in the coverage matrix. Formally, it is defined as the Gini-Simpson index [29] among the rows.

$$\texttt{DDU}(T) = \texttt{density}(T) \times \texttt{diversity}(T) \times \texttt{uniqueness}(T)$$

3) Classification of Metrics: Among the aforementioned metrics, some can be calculated with only the test coverage, while others require the test results. Thereby, we broadly classify them into two categories based on the utilised information:

- **Result-Agnostic**: metrics utilising only coverage information; total coverage, additional coverage, RAPTER, TfD, EntBug, and DDU belong to this category.
- **Result-Aware**: metrics utilising coverage information and previous test results; Prox, S3, and FLINT belong here.

When we select tests sequentially according to the score metrics, the result-agnostic metrics are not affected by whether the previously chosen tests have failed or not, whereas the result-aware metrics are.

B. Prioritisation Metric Comparison Study

We compare how quickly each prioritisation metric finds test cases that accelerate fault localisation on real-world bugs with human-written test cases.

1) Implementation: All metrics described in Section III-A1 and III-A2 have been implemented in Python using PyTorch version 1.1.0. The implementation is publicly available.[4]

TABLE I
EXPERIMENTAL SUBJECT - DEFECTS4J

Subject	# Faults	kLoC	Avg. # Test cases		Avg. # Methods		
			Total	Failing	Total	Suspicious	Faulty
Commons-lang (Lang)	65	22	1,527	1.94	2,245	14.65	1.53
JFreeChart (Chart)	26	96	4,903	3.54	2,205	127.5	4.46
Joda-Time (Time)	27	28	1,946	2.81	4,130	375.37	2.04
Commons-math (Math)	106	85	2,713	1.66	3,602	53.24	1.73
Closure compiler (Closure)	133	90	5,038	2.63	7,927	855.44	1.80

2) Experimental Subject: We empirically evaluate the prioritisation metric on DEFECTS4J version 1.5.0 [16], a real-world fault benchmark of Java programs. Each fault in DEFECTS4J is in the program source code, not the configuration nor test files, and the corresponding patch is provided as a *fixing commit*. For each faulty program, human-written test cases, at least one of which is bound to fail due to the fault, are provided. The failing tests all pass once the fixing commit is applied to the faulty version. Table I shows the statistics of our subjects. The *Suspicious* column contains the average number of methods covered by at least one failing test case. Among the 357 subjects, we exclude two omission faults, Lang-23 and Lang-56, from our study, as they cannot be localised within the faulty version by SBFL techniques. We measure the statement coverage of the provided test suites using Cobertura version 2.0.3, which is part of DEFECTS4J by default.

3) Simulation and Evaluation Protocol: For every fault in our subjects, we set the initial test set to only a single failing test case among all failing test cases (consequently, we evaluate 816 prioritisation, as some of the 355 studied faults have more than one failing test cases). Next, we iteratively select ten test cases from the remaining tests, guided by the prioritisation metrics described in Section III-A. At each

[4]https://github.com/anonymous-icse21/test-prioritisation-metrics

iteration, we calculate the suspiciousness scores of each line using Ochiai and then aggregate them at the method level granularity using *max-aggregation*, which assigns each method the score of its most suspicious line. Finally, all methods in a program are ranked in decreasing order of suspiciousness scores with the max tie-breaker.

To evaluate the ranking results, we employ mean average precision (mAP), a widely used measure in the information retrieval field. mAP is defined as the average of the AP values across multiple queries (faulty programs), where AP stands for the average of precision values at each rank. For example, if a program contains two faulty methods, A and B, placed at the second and fifth places, respectively, the AP is 0.45, the average of $\frac{1}{2}$ and $\frac{2}{5}$. The higher the mAP value is, the better the overall ranking result is. The AP value of a fault with multiple failing test cases is computed as the mean AP of all prioritisations starting from each failing test case.

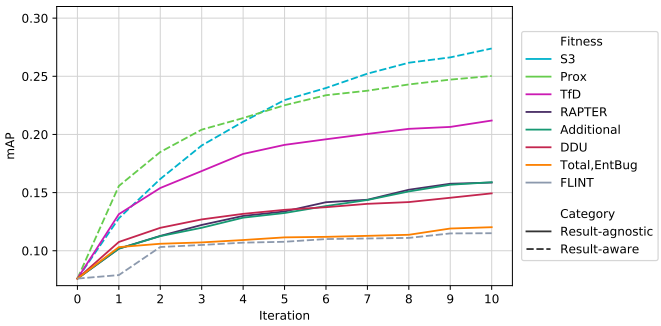

Fig. 2. The mAP values for each prioritisation metric (1st-10th iterations)

4) Results and Implications: Fig. 2 shows the mAP value at each iteration for each test prioritisation method. Each metric is colour-coded, while result-agnostic and result-aware metrics are shown in solid and dashed lines, respectively. While we only present the results up to the tenth iteration, the mAP values will eventually converge to the same value when all available test cases are used. For all prioritisation methods examined, the mAP values increase with every iteration. However, the speed of the convergence varies depending on how test cases are prioritised.

We observe that the two result-aware prioritisation methods, S3 and Prox, outperform all other result-agnostic metrics. With S3, the initial mAP value, 0.076, is increased to 0.274 (an increase of 261%) after the ten test cases are selected, whereas TfD, the best performing result-agnostic metric, improves the mAP value only by 179% within the ten iterations. Prox, which sorts test cases by their similarity to failing tests, shows faster initial convergence, while S3 achieves higher localisation performance after the fifth iteration. Both S3 and TfD, metrics that aim to split ambiguity groups, show the best result at the tenth iteration among the result-aware and result-agnostic metrics, respectively.

Compared to S3 and Prox, FLINT does not perform as well, despite being a result-aware metric. FLINT is originally designed to initially prioritise for coverage and switch to prioritisation for fault localisation once a test fails, whereas our evaluation scenario starts with a failing test case. Consequently, the observed failure rate starts at 1.0 for FLINT, which significantly skews its following analysis.

Among the result-agnostic methods, Total and EntBug behave identically on our subjects. Total gives higher priority to tests that cover more program elements. Similarly, EntBug also takes into account the number of covered program elements, while aiming to reach the optimal density 0.5. However, since the level of coverage achieved by individual human-written test cases in our subject is fairly low, EntBug ends up trying to increase the coverage for all ten iterations, which in turn increases the density. Another interesting observation is that TfD outperforms DDU, even though TfD is conceptually identical to the uniqueness component of DDU. DDU gives high priority to test cases that are either not similar to existing failing test cases or cover more elements, thanks to the diversity and the density metrics. However, the results of Prox show that choosing tests similar to failing tests can be effective, while Total and EntBug show that relying on coverage density alone may not be effective. Based on this, we conclude that, when a failing test already exists and is known, it is better to focus on tests that are similar to the failing test than to increase the overall diversity.

To summarise, result-agnostic metrics may be insufficient to efficiently prioritise test cases when diagnosing faults that have been already observed by failing executions, as they cannot distinguish the more suspicious elements from the ones that are less so. On the other hand, result-aware metrics can give higher priority to tests that will increase the diagnosability for currently suspicious program elements. More detailed comparison data using acc@n metrics can be found in the results for RQ1 (Table II).

IV. User-Guided Iterative Fault Localisation

This section proposes new prioritisation metrics, and introduces QFID, which is a human-in-the-loop iterative fault localisation framework with diagnosability-aware automated test generation and prioritisation.

A. Novel Test Prioritisation Metrics

Section III-B shows that result-aware prioritisation metrics outperform the result-agnostic ones, when starting with a failing test case. This is because the failing test cases tell us which program elements we should focus more on. Based on the observations from Section III-B, we propose two novel result-aware prioritisation metrics, `Split` and `Cover`, designed to better measure the diagnosability gain for more suspicious program elements. `Split` is designed to break ambiguity groups, while `Cover` is designed to focus test executions on more suspicious program elements.

1) Breaking the suspicious ambiguity groups: `Split` measures the expected wasted localisation effort when a new test t is added to T. It extends RAPTER by weighting each ambiguity group using the previous test results as in S3, instead of the size of the ambiguity group. However, while S3 only considers the number of failing test cases, `Split`

uses suspiciousness scores, which take into account both the number of failing test cases and passing ones.

Let us first define the probability of an ambiguity group containing the fault, $p(g)$, as the sum of the probabilities of its elements being faulty:

$$p(g) = \sum_{e_j \in g} p(e_j)$$

where $p(e_j)$ refers to the probability of e_j being faulty. We define $p(e_j)$ as the normalised suspiciousness score of e_j: if w_j is the suspiciousness score of e_j calculated using T, then $p(e_j) = w_j / \sum_i w_i$.[5] Using $p(g)$, `Split` is defined as follows:

$$\begin{aligned}\texttt{Split}(T,t) &= 1 - \sum_{g \in AG(T \cup \{t\})} p(g) \cdot \left(\frac{|g|-1}{2}\right) / \left(\frac{n-1}{2}\right) \\ &= 1 - \frac{1}{n-1} \cdot \sum_{g \in AG(T \cup \{t\})} p(g) \cdot (|g|-1)\end{aligned}$$

Here, $(n-1)/2$ is the normalisation constant, which is the maximum localisation effort when all program elements belong to a single ambiguity group. In general, if $T \cup \{t\}$ produces larger and more suspicious ambiguity groups, `Split` will penalise t more heavily.

2) Covering the suspicious program elements: `Cover` quantifies how much a new test case covers current suspicious elements. More formally, it measures weighted coverage, where the weights of program elements are simply their suspiciousness scores.

$$\texttt{Cover}(T,t) = \frac{\sum_{j=1}^{n} w_j \cdot c_t[j]}{n}$$

`Cover` shares movitation with Prox [24], [25], which directly uses coverage similarity to failing tests. However, this direct comparison is its downfall: if there are multiple failing tests with significantly different coverage patterns, averaging similarities may not be the best way to measure the similarity between test cases. We instead use the suspiciousness score, which can be thought of as an aggregation of failing and passing test executions, to weight coverage.

3) Combining metrics: As both metrics are normalised, it is possible to hybridise by taking the weighted sum:

$$\texttt{FDG}(T,t) = \beta \cdot \texttt{Split}(T,t) + (1-\beta) \cdot \texttt{Cover}(T,t)$$

where $0 \leq \beta \leq 1$ determines the relative weights of metrics.

B. QFID: A Framework for HITL Iterative Fault Localisation

SBFL techniques require a test suite with sufficient diagnostic capabilities [9]. However, when a program fault is observed, there may be only a few other, or even no other, test cases except for the ones that revealed the initial failure, which makes it challenging for SBFL to localise the fault accurately. For example, a program actively being developed may not yet be equipped with a fully adequate test suite; even for a mature project, there may be parts of the system for which test cases are inadequate. To address such cases, we propose QFID, a framework for user-driven iterative fault localisation, which helps to localise faults in the absence of sufficient test cases. QFID uses test case prioritisation to elicit human labels as efficiently as possible for automatically generated test data, with the aim of achieving the most accurate localisation using the labels.

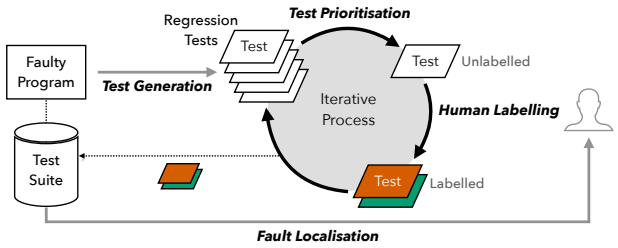

Fig. 3. Overview of QFID

An overview of QFID is presented in Fig. 3. QFID uses an automatic test generation tool, such as EvoSuite [15] or Randoop [30], to produce regression tests for the given faulty program (**Test Generation**). Since regression tests simply record the current behaviour of the program, some generated assertions in the tests may capture the faulty behaviour of the program [31].[6] Consequently, unless the failure is detectable by implicit oracles [33] (such as program crashes or uncaught exceptions), human judgement on the correctness of the captured program behaviour is required (**Human Labelling**). However, this, in turn, raises a couple of issues:

1) If the program size is large, the number of tests generated may be too numerous for manual inspection.
2) The labelling budget may be limited, so the order in which tests are presented to a user should be efficient.

To mitigate these problems, QFID first tries to reduce the total number of generated tests by limiting the scope of regression tests to only the suspicious program elements that are covered by the initial failing tests, rather than the entire program. Second, using a test prioritisation technique, QFID presents more relevant tests earlier to the developer, to facilitate effective and efficient fault localisation (**Test Prioritisation**).

Algorithm 1 formally describes the workflow of QFID. It generates regression tests T' for the suspicious part P' of a given faulty program P within the test generation budget tb. Then, until either the oracle querying budget is exhausted or T' becomes empty, it iteratively selects the test case t_s that has the maximum value of the prioritisation metric, f, from T'. If a test t_s reveals faulty behaviour, the user labels it as *Failing* (i.e., $R(t_s) = 0$), or otherwise as *Passing* (i.e., $R(t_s) = 1$). Once labelled, the test is moved from T' to T, and the test result R is updated. Finally, when the loop terminates, QFID returns the final fault localisation results.

[5]Note that we use min-max scaled Ochiai scores throughout this paper.

[6]Note that the regression test cases generated by EvoSuite always capture the behaviour of the current system as identity assertions [32]. As such, labelling whether these assertions are correct or not is essentially to judge whether the captured output value is correct or not.

Algorithm 1: QFID

Input: Faulty program P, Initial test suite T, Test results R, Test generation tool G, Test generation budget tb, Prioritisation metric f, Querying budget qb, SBFL formula S

Output: The fault localisation result F

```
assert ∃t ∈ T.R(t) = 0;
P' ← GetSuspiciousParts(T, R);
T' ← G(P', tb);                          // test generation
iter ← 1;
while iter <= qb and T' ≠ ∅ do
    t_s ← argmax_{t∈T'} f(T, t);         // test selection
    R(t_s) ← HumanLabel(t_s);            // human labelling
    T ← T ∪ {t_s};                       // added to the test suite
    T' ← T' \ {t_s};
    iter ← iter + 1;
end
F ← S(T, R);                             // fault localisation
```

V. Evaluation

This section introduces research questions for the empirical evaluation of our novel prioritisation metrics and QFID, describes the experimental setting, and discusses the results.

A. Research Questions

1) ***RQ1*** *(Prioritisation Effectiveness):* We ask how effective our novel prioritisation metric is when compared to the existing approaches. To evaluate our prioritisation metrics, we use the same protocol described in Section III-B3. For FDG, we set the parameter β to 0.5.

2) ***RQ2*** *(Localisation Effectiveness):* We investigate how accurate QFID is for the localisation of real-world faults under limited test adequacy. For this, we try to localise the faults in DEFECTS4J using QFID and only the failing test cases. As a test data generation tool we use EvoSuite [15], and for the test prioritisation our proposed metric FDG ($\beta = 0.5$) is used. Faults are localised at the method level as described in Section III-B3.

For each faulty program, we use EvoSuite version 1.0.7[7] to generate the additional regression tests. To run the most recent version of EvoSuite, we use DEFECTS4J version 2.0.0, unlike the metric comparison study in Section III-B that used 1.5.0. For each fault, we run EvoSuite[8] for each suspicious class[9], limiting the target methods to only the suspicious methods. We use two time budgets, 3 and 10 minutes, and allocate the physical time budget to a class proportionally to the number of suspicious methods in the class. For example, under the 3-minute budget, suppose suspicious classes A and B contain 5 and 10 suspicious methods, respectively: we allocate $3 \times \frac{5}{5+10} = 1$ minute to class A, and $3 \times \frac{10}{5+10} = 2$ minutes to class B. We generate 20 test suites for each fault, using the 3- and 10-minute time budgets and ten random seeds.

The coverage of the tests is measured using Cobertura[10]. We simulate a perfect human oracle by running the generated test suite on the fixed version. If a regression test case fails on the fixed version (due to oracle violation or compile error), we regard the test as a failing test.

3) ***RQ3*** *(Robustness against Human Errors):* We evaluate how robust QFID is against labelling error, as expecting the human engineer to act as a perfect oracle without mistake is not realistic. To simulate human labelling mistakes, we artificially inject labelling noises by introducing various error rates into the same experiment as the one for RQ2. We control the error rate, i.e., the probability p of flipping the correct judgement, to $\{0.1, 0.3, 0.5, 0.7, 0.9\}$ and see how much the localisation performance deteriorates.

B. Excluded Faults

In the experiments for RQ2 and RQ3, the 133 faults in Closure were excluded from the evaluation of QFID as there were too many suspicious classes and methods to distribute the given time budget. In addition, among the subjects listed in Table I, we exclude two faults deprecated in DEFECTS4J v2.0.0, Lang-2 and Time-21, as well as two omission faults, Lang-23 and -56. Due to an unknown reason, the list of relevant (suspicious) classes of Time-5 provided by DEFECTS4J does not contain the actual faulty class: we also exclude this fault instead of modifying the list of relevant classes arbitrarily. We also could not compile the failing tests for 23 faults in Math (IDs 17-35, 98, 100-102) due to an encoding problem, which prevented us from measuring the coverage.[11] Overall, 161 out of 357 faults have been excluded.

C. Evaluation of Localisation Performance

To evaluate the results, we use the following metrics:

- **Top-n Accuracy** (acc@n): For every fault in our subject, acc@n measures the number of faults where at least one of the faulty program elements are ranked within the top n locations. When there are several initial test sets for one faulty subject, the acc@n value for that subject is averaged over the whole initial test suite.
- **Mean Average Precision** (mAP): See Section III-B3.

D. Experimental Results

1) ***RQ1*** *(Prioritisation Effectiveness):* Table II shows the prioritisation performance of each method on DEFECTS4J human-written tests. Each row presents acc@n values ($n \in [1, 3, 5, 10]$) of fault localisation results at each iteration, and the *Initial* row and *Full* row represent the results when using

[7]This is not an official release, but is the most recent version on GitHub (commit no. 800e12).

[8]We set the maximum number of tests per class to 200. To avoid long and complex test cases being generated, the length of test case is limited to 20. Other than that, the default configuration is used.

[9]We use the *classes.relevant* property in DEFECTS4J.

[10]EvoSuite also reports the line level coverage of test cases, but since it only includes the coverage within the target class, Cobertura is used to measure the coverage for all suspicious classes. Modifying EvoSuite would eliminate the cost of measuring coverage separately.

[11]This is a fixable problem that can be addressed in time if accepted: we do not expect these faults to change the overall trend in the results.

TABLE II
THE ACC@N VALUES WITH THE SELECTED HUMAN-WRITTEN TEST CASES OF DEFECTS4J AT EACH ITERATION (# TOTAL SUBJECTS = 355)

n		1	3	5	10	1	3	5	10	1	3	5	10	1	3	5	10	1	3	5	10	1	3	5	10
Initial		4	30	45	66	4	30	45	66	4	30	45	66	4	30	45	66	4	30	45	66	4	30	45	66
Metric		EntBug				Total				DDU				Additional				RAPTER				TfD			
Iteration	1	9	40	62	82	9	40	62	82	10	43	65	87	9	39	60	82	9	39	60	82	22	52	68	86
	2	9	42	64	87	9	42	64	87	12	51	76	102	10	44	68	97	10	44	68	97	29	68	82	101
	3	9	41	64	92	9	41	64	92	14	54	77	110	11	47	72	102	12	49	74	103	33	73	90	114
	4	9	41	66	96	9	41	66	96	15	55	78	115	12	53	78	107	13	54	78	107	35	83	97	129
	5	9	43	69	98	9	43	69	98	15	55	80	118	12	55	82	115	13	54	83	115	37	84	100	132
	6	9	43	69	99	9	43	69	99	15	56	80	121	14	57	84	118	16	57	85	120	37	89	102	137
	7	9	43	70	99	9	43	70	99	15	57	82	126	15	58	87	122	16	58	86	120	38	92	108	145
	8	9	43	70	102	9	43	70	102	15	57	82	128	17	61	88	124	18	60	87	126	39	92	112	148
	9	11	45	70	102	11	45	70	102	16	59	83	130	19	63	89	128	20	62	89	129	39	94	114	152
	10	12	46	70	104	12	46	70	104	17	60	85	130	19	66	90	130	20	64	89	130	41	96	114	152
Metric		FLINT				Prox				S3				**Split**				**Cover**				**FDG** ($\beta = 0.5$)			
Iteration	1	4	31	47	70	**28**	72	**98**	**131**	23	52	70	93	23	52	70	93	26	**76**	95	123	25	66	89	111
	2	8	46	67	80	35	88	119	146	33	71	94	126	32	71	93	123	36	94	114	146	**42**	**101**	**126**	**157**
	3	9	48	69	80	41	98	123	159	41	89	109	139	42	93	114	148	47	106	131	165	**49**	**109**	**137**	**177**
	4	10	49	70	82	43	108	131	169	48	98	125	153	48	108	128	155	**54**	115	146	182	53	**117**	**147**	**192**
	5	10	49	72	83	46	110	138	175	56	107	130	160	54	116	137	162	55	**122**	156	190	**58**	**122**	**160**	**206**
	6	10	50	73	84	48	113	141	183	58	115	137	162	**61**	124	143	166	56	130	157	194	**61**	**131**	**166**	**214**
	7	10	50	73	87	48	116	145	187	**65**	120	143	166	62	127	146	173	59	133	164	200	**65**	**137**	**172**	**223**
	8	10	51	74	87	49	122	154	190	**68**	122	148	173	64	131	153	178	63	135	169	206	67	**141**	**176**	**222**
	9	12	53	76	87	50	123	156	192	**68**	126	151	181	66	132	155	180	64	135	177	211	**68**	**147**	**180**	**222**
	10	12	53	76	87	51	126	161	197	**72**	129	155	182	70	136	159	191	65	141	172	214	70	**150**	**181**	**224**
Full		112	210	254	281	112	210	254	281	112	210	254	281	112	210	254	281	112	210	254	281	112	210	254	281

a single initial failing test case and full test suite (4,207 tests on average), respectively. The numbers in bold are the highest values in the iteration for the corresponding n value.

Results show that our method, FDG, outperforms all other prioritisation methods by achieving the highest acc@3, 5, 10 values in all iterations except for the first. In particular, in the 10th iteration, the tests selected by FDG achieve 17.5x acc@1 and 3.4x acc@10 compared to the initial test suite. Also, although the proportion of tests is only $\frac{1+10}{4207} = 0.26\%$ of the full test cases, FDG achieves 62.5% of acc@1 and 80.0% of acc@10 compared to the full test suite. Furthermore, we report the performance of Split and Cover as stand-alone metrics. Cover both performs better and converges faster than Split. Although their performances are lower than that of the FDG, the new metrics overall perform better than or equal to all existing metrics as a stand-alone metric.

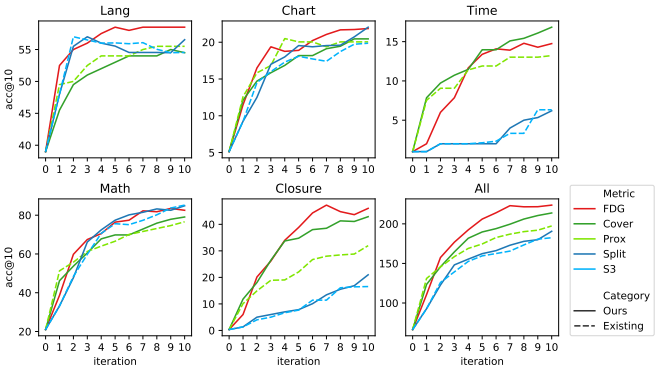

Fig. 4. The acc@10 values per projects during the ten iterations

Fig. 4 shows the acc@10 values per projects for our and best-performing existing methods. Each metric is colour-coded: our new metrics are shown in solid lines, and existing ones in dashed lines. FDG performs the best when results from all projects are aggregated and, in particular, it significantly outperforms the existing methods for Closure, achieving acc@10 values higher by 15 and 30 when compared to Prox and S3, respectively. The performance trends of Split and Cover are similar to those of S3 and Prox, which reflects our design intention: these two strategies are complementary to each other, as the better performance of FDG shows.

Based on the results, we answer RQ1 that FDG can significantly outperform existing coverage-based test prioritisation techniques on real-world faults. For a detailed discussion of other metrics, please refer to Section III-B.

TABLE III
THE AVERAGE NUMBER OF TOTAL TEST CASES GENERATED BY EVOSUITE (THE NUMBERS IN PARENTHESES MEAN THE NUMBER OF FAILING TESTS.)

Time	Project			
Budget	Lang	Chart	Time	Math
3 mins	32.4 (1.0)	225.2 (2.4)	441.8 (0.7)	89.8 (0.8)
10 mins	34.2 (1.1)	232.6 (2.5)	475.3 (0.8)	93.5 (0.9)

*2) **RQ2** (Localisation Effectiveness):* Table III shows the average number of total and failing test cases generated by EvoSuite for the studied subjects, using 3 and 10 minutes as budgets, respectively (hereafter denoted by T3 and T10). These tests correspond to T' in Line 3 of Algorithm 1. Using these generated test cases, we evaluate localisation performance after a different number of human oracle queries.

Fig. 5 shows how the acc@n values change as the query budget increases for T3 and T10 scenarios. Overall, T10 has slightly better localisation results and less standard deviation in performance than T3, although the difference is not significant. This is as expected, for T10 is more likely to produce larger and more diverse sets of test cases. Through ten oracle queries, we can achieve about 13 times acc@1 and 2 times acc@10 values compared to the initial test set. We expect faults with a single failing DEFECTS4J test case to be more difficult to localise and hence analyse these separately. In total, 138 out of

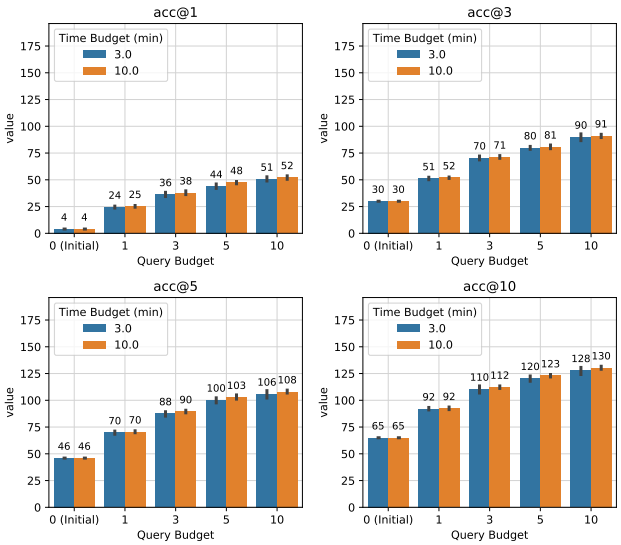

Fig. 5. The acc@n values for each time and query budgets. The error bars show the standard deviation of the observations from different seeds. (# total subjects = 196)

196 studied faults have a single failing test case (the average is 2.2). For these faults, ten human queries can increase the acc@1 from 4 to 42 (10.5x) and the acc@10 from 50 to 97 (1.9x) with T10, which is an almost equivalent improvement compared to the overall results. Interestingly, it is not required to generate many additional failing test cases to localise faults accurately. The localisation results are still promising despite the small number of generated failing test cases (see Table III). This is because generated passing test cases can still effectively decrease the suspiciousness of non-faulty program elements.

We answer RQ2 by noting that QFID can effectively localise real-world faults, even when there are only a few failing test cases. Using EvoSuite and FDG, QFID can localise 27% (52 out of 196) of the studied faults at the top, and 66% (130 out of 196) faults within the top ten, while requiring only ten human oracle queries.

*3) **RQ3** (Robustness against Human Errors):* Fig. 6 shows how localisation performance varies against different labelling error rates. The results show that localisation performance gradually deteriorates as the error rate increases. We observe that T10 is slightly more robust than T3 against the same error rate, although it is not statistically significant. Based on the acc@1 values, the localisation results are fairly robust up to a noise probability of 0.3; however, above 0.3, the amount of improvement compared to the initial test is reduced to almost half (e.g., from the improvement of 12.25 times from 4 to 49 when error rate is 0.1, to 7.5 times from 4 to 30 when the error rate is 0.5). The results show that QFID can still perform significantly better than the initial test suite even under noisy human oracles. This is because the correctly labelled subsequent test cases can mitigate the impact of the error and adjust the results. The richer the generated test suite is, the more likely it contains tests that can adjust the errors caused by the incorrect labels: this may explain why T10 is slightly more robust than T3 in our results.

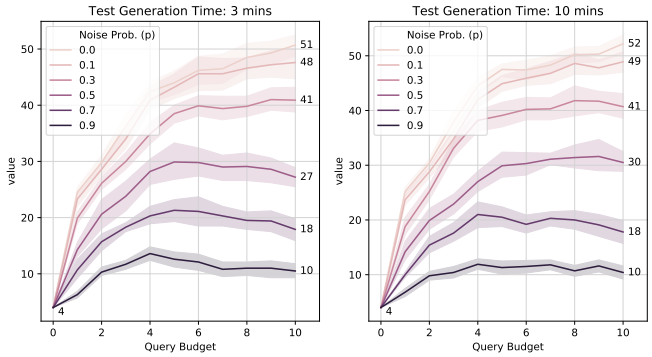

Fig. 6. The acc@1 values for each labelling noise probabilities and query budgets. The error bands show the standard deviation of the observations from different seeds.

Existing related work [31] shows that a qualified crowd can perform well as a human oracle, i.e., do a good job in classifying the wrong assertions in EvoSuite tests, with accuracy ranging from 0.69 to 0.90 depending on the level of difficulty. Based on the results, we answer RQ3 by showing that QFID can be resilient to human errors in oracle judgement.

E. Impact of Test Granularity

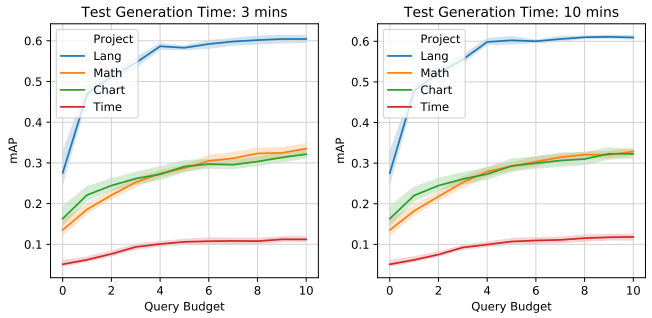

Fig. 7. The mAP values of fault localisation results for each projects and query budgets. The error bands show the standard deviation of the observations from different seeds.

Fig. 7 shows that the localisation mAP values, as well as their convergence rate that varies greatly between projects. QFID performs the best for Lang and the worst for Time. Investigating the root cause of this performance gap may help us decide when to recommend the use of QFID. Previous work [34] has shown that the granularity of tests may have an effect on the fault localisation effectiveness. Even if a test is created with the same tool, the granularity of tests can be significantly different due to the internal dependency structure of the target program.

The boxplot in Fig. 8 shows the number of methods covered by all generated test cases per projects. While the test cases of Lang cover only 10.1 methods on average, those of Time cover 323.1 methods on average. Since EvoSuite only targets a single class at a time, we assume that the high number of covered methods observed in Time is due to the interdependency in the code structure. This is reflected in the low mAP value of the initial failing test set (i.e., mAP values with no query in Fig. 7): coarse-grained test cases result in larger initially suspi-

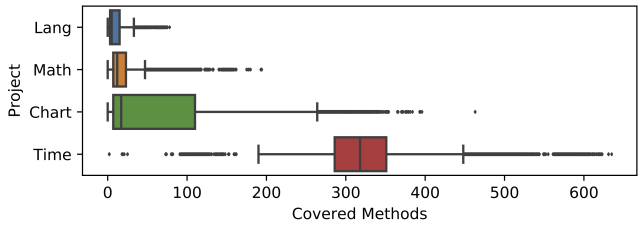

Fig. 8. The distribution of the # of methods covered by generated tests

cious program elements. While further investigation is needed to thoroughly understand the condition under which QFID performs well, we posit that finer test granularity is important for QFID as with other fault localisation techniques [34].

VI. Related Work

A. Human-in-the-Loop Debugging

Many debugging techniques utilised user feedback [35]–[44]. Hao et al. proposed an interactive debugging framework that recommends breakpoints in programs based on the suspiciousness of statements [39]: a developer examines the program state at each point, and the resulting feedback is used to modify the suspiciousness. ENLIGHTEN [41] asks the developer to investigate input-output pairs of the most suspicious method invocations: the feedback is then encoded as a virtual test and used to update the localisation results. QFID differs from ENLIGHTEN in that it does not require a full test suite. Recently, Böhme et al. proposed Learn2Fix [42], a human-in-the-loop program repair technique. Learn2Fix generates test cases by mutating the failing test and allowing the user label them in the order of their likelihood of failing. Learn2Fix trains an automatic bug oracle using user feedback, which is in turn used to amplify the test suite for better patch generation. While Learn2Fix only supports programs with numerical input/output due to the need of an SMT solver in learning the bug oracle, QFID has no limitations on variable types.

B. Automated Test Generation for Fault Localisation

Automated test generation has been widely used to support fault localisation. Artzi et al. [45] use concolic execution to generate test cases similar to failing executions [24]: in comparison, QFID simply aims to generate diverse test cases that cover the faulty program element using a search-based approach, and depends on a human oracle and test prioritisation for effective localisation. BUGEX [46] generates additional test cases similar to a given failing one using EvoSuite and uses an oracle to differentiate between passing and failing executions. Finally, BUGEX identifies the runtime properties that are relevant to the failure by comparing passing and failing executions. Compared to BUGEX, QFID does not assume the existence of an automated bug oracle and instead uses test prioritisation to reduce the number of oracle invocations. F^3 [47] is a fault localisation technique for field failures, which extends a bug reproduction technique, BugRedux [48]: it synthesises failing and passing executions similar to the field failure and uses them for their customised fault localisation technique. While F^3 can only debug program crashes, which can be detected implicitly, QFID aims to localise faults where no such automatic oracle is available. Xuan et al. [49] split test cases into finer pieces to increase the fault diagnosability of a test suite, instead of generating test cases from scratch. In comparison, QFID targets a scenario in which little or no additional test cases are available apart from the failing ones.

VII. Threats to Validity

Threats to internal validity regard factors that may influence the observed effects, such as the integrity of the coverage and test results data, as well as the test data generation and fault localisation. We have used the widely studied Cobertura and EvoSuite tools, as well as the publicly available scripts in DEFECTS4J, to collect or generate data.

Threats to external validity concern any factor that may limit the generalisation of our results. Our results are based on DEFECTS4J, a fault benchmark against which many of fault localisation techniques are evaluated. However, only further experimentations using more diverse subject programs and faults can strengthen the generalisability of our claims. We also consider the additive prioritisation scenario, in which test cases are chosen one by one. It is possible that non-constructive heuristics such as Genetic Algorithm may choose different and potentially better sets of test cases. Our choice of constructive prioritisation is based on the assumption that an iterative process will be easier for the human engineer to make the oracle judgement.

Finally, threats to construct validity concerns situations where used metrics may not reflect the true potential properties they claim to measure. All evaluation metrics that we used for our claims are intuitive and all widely used in fault localisation literature, leaving little room for misunderstanding. It is possible that Coincidental Correctness (CC) [50] has interfered with our measurements, as it is known to exist in DEFECTS4J [51]. However, it is theoretically impossible to entirely filter out CC from test results, and all coverage-based fault localisation techniques are equally affected by CC.

VIII. Conclusion

We first propose novel coverage-based test prioritisation metrics for SBFL, `Split` and `Cover`. When combined, our technique significantly outperforms existing methods: when choosing only ten test cases based on our metric for fault localisation, the acc@10 values are 14% and 23% higher than state-of-the-art metrics, S3 and Prox, respectively. We also introduce a human-in-the-loop iterative fault localisation framework, QFID, that localises faults when only a small number of failing tests is available. It uses an automated test generation tool to augment existing tests and requires the least test labels from a human oracle while achieving high localisation accuracy. Empirical evaluation using EvoSuite and DEFECTS4J shows that QFID can achieve an acc@1 of 51 out of 196 studied faults requiring three minutes of test data generation using EvoSuite and only ten human oracle elicitations.

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[2] 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*, 2005, pp. 273–282.

[3] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[4] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[5] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation*, ser. ICST 2014, 2014, pp. 153–162.

[6] S. Hong, T. Kwak, B. Lee, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Museum: Debugging real-world multilingual programs using mutation analysis," *Information and Software Technology*, vol. 82, pp. 80–95, 2017.

[7] M. Papadakis and Y. L. Traon, "Metallaxis-fl: mutation-based fault localization," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 605–628, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1509

[8] M. Papadakis and Y. Le-Traon, "Using mutants to locate "unknown" faults," in *Proceedings of the 5th IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. Mutation 2012, 2012, pp. 691–700.

[9] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 654–664.

[10] 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*, 2006, pp. 82–91.

[11] S. Yoo, M. Harman, and D. Clark, "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 1–29, 2013.

[12] A. Gonzalez-Sanchez, E. Piel, H.-G. Gross, and A. J. van Gemund, "Prioritizing tests for software fault localization," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 42–51.

[13] D. Hao, L. Zhang, and H. Mei, "Test-case prioritization: achievements and challenges," *Frontiers of Computer Science*, vol. 10, no. 5, pp. 769–777, 2016.

[14] A. Gonzalez-Sanchez, É. Piel, R. Abreu, H.-G. Gross, and A. J. C. van Gemund, "Prioritizing tests for software fault diagnosis," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1105–1129, 2011. [Online]. Available: http://dx.doi.org/10.1002/spe.1065

[15] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[16] 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*, 2014, pp. 437–440.

[17] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[18] G. Stenbakken, T. Souders, and G. Stewart, "Ambiguity groups and testability," *IEEE Transactions on Instrumentation and Measurement*, vol. 38, no. 5, pp. 941–947, 1989.

[19] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 83–92.

[20] P. R. Halmos, *Naive set theory*. Courier Dover Publications, 2017.

[21] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[22] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE transactions on software engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[23] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[24] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 30–39.

[25] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 2011, pp. 420–423.

[26] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun, and H. Mei, "Test input reduction for result inspection to facilitate fault localization," *Automated software engineering*, vol. 17, no. 1, p. 5, 2010.

[27] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[28] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 257–267.

[29] L. Jost, "Entropy and diversity," *Oikos*, vol. 113, no. 2, pp. 363–375, 2006.

[30] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, 2007, pp. 815–816.

[31] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 342–351.

[32] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2012.

[33] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[34] G. Laghari and S. Demeyer, "Poster: Unit tests and component tests do make a difference on fault localisation effectiveness," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 280–281.

[35] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158.

[36] A. Ko and B. Myers, "Debugging reinvented," in *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, pp. 301–310.

[37] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Interactive fault localization leveraging simple user feedback," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 67–76.

[38] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "Vida: Visual interactive debugging," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 583–586.

[39] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive fault localization using test information," *Journal of Computer Science and Technology*, vol. 24, no. 5, pp. 962–974, 2009.

[40] X. Li, M. d'Amorim, and A. Orso, "Iterative user-driven fault localization," in *Haifa Verification Conference*. Springer, 2016, pp. 82–98.

[41] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 82–92.

[42] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," *arXiv preprint arXiv:1912.07758*, 2019.

[43] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 393–403.

[44] A. Bandyopadhyay and S. Ghosh, "Tester feedback driven fault localization," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 41–50.

[45] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*, 2010, pp. 49–60.

[46] J. Röβler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 international symposium on software testing and analysis*, 2012, pp. 309–319.

[47] W. Jin and A. Orso, "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 213–223.

[48] ——, "Bugredux: reproducing field failures for in-house debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 474–484.

[49] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 52–63.

[50] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, 2009, pp. 1–5.

[51] R. Abou Assi, C. Trad, M. Maalouf, and W. Masri, "Coincidental correctness in the defects4j benchmark," *Software Testing, Verification and Reliability*, vol. 29, no. 3, p. e1696, 2019.