Delft University of Technology
Software Engineering Research Group
Technical Report Series

A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches

Alexandre Perez, Rui Abreu, and Arie van Deursen

Report TUD-SERG-2017-004

TUD-SERG-2017-004

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Accepted for publication at the ACM/IEEE International Conference on Software Engineering (ICSE), held in Buenos Aires, May 2017.

A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches

Alexandre Perez*†, Rui Abreu*†, Arie van Deursen‡
*University of Porto & HASLab, INESC TEC, Portugal
†Palo Alto Research Center, USA
‡Delft University of Technology, The Netherlands
alexandre.perez@fe.up.pt, rui@computer.org, arie.vandeursen@tudelft.nl

***Abstract*—Current metrics for assessing the adequacy of a test-suite plainly focus on the number of components (be it lines, branches, paths) covered by the suite, but do not explicitly check how the tests actually exercise these components and whether they provide enough information so that spectrum-based fault localization techniques can perform accurate fault isolation. We propose a metric, called DDU, aimed at complementing adequacy measurements by quantifying a test-suite's diagnosability, *i.e.,* the effectiveness of applying spectrum-based fault localization to pinpoint faults in the code in the event of test failures. Our aim is to increase the value generated by creating thorough test-suites, so they are not only regarded as error detection mechanisms but also as effective diagnostic aids that help widely-used fault-localization techniques to accurately pinpoint the location of bugs in the system. Our experiments show that optimizing a test suite with respect to DDU yields a 34% gain in spectrum-based fault localization report accuracy when compared to the standard branch-coverage metric.**

***Keywords*-Testing; Coverage; Diagnosability.**

I. Introduction

This paper proposes DDU, a new metric for evaluating the diagnosability of a test-suite when applying spectrum-based fault localization approaches. Aimed at complementing adequacy measurements that focus on maximizing error detection of a suite, DDU provides an assessment on its efficiency at pinpointing the root cause of failure given that an error is detected. The proposed measurement increases the value of having a thorough test-suite, since an optimal suite with respect to DDU can not only act as an error detection tool but also as an aid to widely used fault localization approaches.

Current test quality metrics quantitatively describe how close a test-suite is to thoroughly exercising a system according to an adequacy criterion. Such criteria describe what characteristics of a program must be exercised. Examples of current metrics include branch and path coverage [1], modified decision/condition coverage [2], and mutation coverage [3]. According to Zhu *et al.*, such measurements can act as *generators*, meaning that they provide an intuition on *what* components to exercise to improve the suite [4]. However, this *generator* property does not provide any relevant, actionable information on *how* to test those components. These adequacy measurements abstract away the execution information of single test executions to favor an overall assessment of the suite, and are therefore oblivious to anti-patterns like the ice-cream cone.[1] The anti-pattern states that the vast majority of tests is written at the system level, with very few tests written at the unit granularity level. Even though high-coverage test-suites can detect errors in the system, it is not guaranteed that inspecting tests will yield a straightforward explanation for the cause of the observed failures, since fault isolation is not a primary concern. Our hypothesis is that a complementing metric that takes into account per-test execution information can provide further insight about the overall quality of a test-suite. This way, if a regression happens, we would have a test suite that is not only effective at *detecting* faults, but also aids spectrum-based techniques to *pinpoint* them among the code.

Previous test-suite diagnosability research has proposed measurements to assess diagnostic efficiency of spectrum-based fault localization techniques. One measurement uses the density (ρ) of a test-coverage matrix — also known as spectrum [5]: input to all spectrum-based fault localization techniques [6], [7] —, which encodes what software components have been involved in each test. González-Sanchez *et al.* have shown that when spectrum density approaches the optimal values, the effectiveness of spectrum-based approaches is maximal [8]. Another approach is one by Baudry *et al.*, that proposed a *test for diagnosis* criterion that attempts to reduce the size of *dynamic basic blocks* to improve fault localization accuracy [9].

Unfortunately, the existing diagnosability metrics rely on impractical assumptions that are unlikely to happen in the real world. The approach by Baudry *et al.* focuses on detection of single-faults in the system. The density approach assumes that all tests programmers write exercise a different path through the code and therefore produce different coverage patterns. In practice, it is common for tests to cover the same code. If one does not account for test diversity, it is possible to skew the test-coverage matrix to have a (supposedly) optimal density by repeating similar test cases. It also has the assumption that all tests cover, on average, the same number of code components. In reality, a test-suite can encompass tests ranging from a targeted, narrow unit test to a sweeping system test.

This paper details the optimal coverage matrix for achieving accurate spectrum-based fault localization. In this scenario,

[1] Ice-cream cone software testing anti-pattern mentioned in Alister Scott's blog: http://goo.gl/bhXOrN (accessed February 2017).

the test-suite contains a test case exercising every possible combination of components in the system, so that not only single-faults can be pinpointed but also allows for multiple-faults – which require simultaneous activations of components for the fault to manifest – can be isolated. Such a matrix is reached when its entropy is maximal. This is the theoretically optimal scenario. However, this entropy-maximization approach is intractable due to the sheer number of test cases required to exercise every combination of components in any real-world system.

Nevertheless, the entropy-optimal scenario helps elicit a set of properties coverage matrices need to exhibit for accurate spectrum-based fault localization. We leverage these properties in our proposed metric, coined DDU.[2] This metric addresses the related work assumptions detailed above, while still ensuring tractability, by combining into a single measurement the three key properties spectra ought to have for practical and efficient diagnosability: a) density (ρ), ensuring components are frequently involved in tests; b) test diversity ($\mathcal{G}$), ensuring components are tested in diverse combinations; and c) uniqueness ($\mathcal{U}$), favoring spectra with less ambiguity among components and providing a notion of component distinguishability. The metric addresses the quality of information gained from the test-suite should a program require fault-localization activities, and is intended as a complement to adequacy measurements such as branch-coverage.

To measure the effectiveness of the proposed metric, we perform two empirical evaluations of DDU by generating test suites for faulty software projects. Test generation, facilitated by the EVOSUITE tool, is guided to optimize test suites regarding a specific metric, and oracles are generated from correct project versions. The first empirical evaluation shows that generating tests that optimize DDU produces test-suites that require less diagnostic effort to find the faults compared to density. The second empirical evaluation generates test-suites for a wide range of subjects in the DEFECTS4J collection. It shows that optimizing a suite regarding DDU yields an increase of 34% in diagnostic accuracy when compared to test-suites that only consider branch-coverage as the optimization criterion.

This paper's contributions are:

- A description of the theoretically optimal test-suite for fault localization: one that generates a coverage matrix with maximal entropy. This optimal scenario is intractable due to the sheer number of test cases needed to be generated.
- We elicit from the optimal scenario three key properties matrices ought to exhibit to preserve high diagnostic accuracy: density, diversity and uniqueness.
- DDU, a new metric based on the aforementioned properties to assess a test-suite's diagnostic ability to pinpoint a fault in the system using spectrum-based techniques. The metric complements adequacy measurements such as branch-coverage.

[2]DDU is an acronym for Density-Diversity-Uniqueness.

		t_1	t_2	t_3	t_4
1:	`method groundDistance () {`	●	●		●
2:	`if (underwater()) {`	●	●		●
3:	`return surfaceDistance();`		●		●
4:	`} else {`	●			
5:	`return groundAltitude();}}`	●			
6:					
7:	`method groundAltitude () {`	●		●	●
8:	`if (landed()) {`	●		●	●
9:	`return 0;`			●	●
10:	`} else {`	●			
11:	**`return sub(GND, ALT);`**`}}`	●			
12:					
13:	`method sub (a,b) {`	●			
14:	`return a - b;}`	●			
	Pass/fail status:	✗	✓	✓	✓

(a) Per-test coverage of a single-faulted system.

		t_1	t_2	t_3	t_4
1:	`method descend (increment) {`	●	●		●
2:	`if (landed()) {`	●	●		●
3:	`return Status.STOPPED;`	●			●
4:	`} else {`		●		
5:	**`descendMeters(increment);`**		●		
6:	`return Status.DESCENDING;}}`		●		
7:					
8:	`method ascend (increment) {`	●		●	
9:	`if (landed()) {`	●		●	
10:	`liftoff();`	●			
11:	`return Status.LIFTOFF;`	●			
12:	`} else {`			●	
13:	**`ascendFeet(increment);`**			●	
14:	`return Status.ASCENDING;}}`			●	
	Pass/fail status:	✓	✓	✓	✓

(b) Per-test coverage of a multiple-faulted system.

Fig. 1: Code snippets showing test and coverage information. Test passes and failures are represented by ✓and ✗. ● indicates that the component in the respective row was exercised.

- Empirical evidence that DDU is more accurate at assessing diagnostic ability than the state-of-the-art.
- Empirical evidence that optimizing a test-suite with respect to DDU yields a 34% gain in diagnostic efficiency when compared to similarly adequate suites.

II. MOTIVATION

We present two code snippets along with runtime information of several test cases as a motivational example demonstrating the need for a new metric that accurately describes the diagnostic ability of a test-suite.[3]

The first example, depicted in Figure 1a, shows a snippet of code from a sensor array capable of measuring distance to the ground both when submerged and airborne. The purpose of `groundAltitude` is to measure distance to the ground using the internal altitude sensor (`ALT`) and the ground elevation sensor (`GND`). This method has a bug: it will produce negative values if `ALT` is greater than `GND`. Line 11 should then read `return sub(ALT, GND);`. Test t_1 does indeed detect the error in the system. But the problem is that no other test also exercises the branches followed by t_1 to exonerate them from suspicion. This results in the developer having to

[3]We use line of code as the component granularity throughout the motivation section.

manually inspect all components that do not appear in passing tests. Six lines out of a total of 12 will have to be inspected, corresponding to nearly 50% of the total code in the snippet. In this small example, it is feasible to inspect all components, but component inspection slices can grow to fairly large numbers in a real world scenario. So, even though this test-suite has 100% branch-coverage, it does not provide many diagnostic clues.

The second example, depicted in Figure 1b, contains a snippet of code for controlling the ascent and descent of a drone. The `descend` method uses meters to quantify the amount of descent, while the `ascend` method uses feet. Assuming there is no explicit check for altitude available, testing these methods independently will not reveal the failure. In fact, only a test that covers both methods' else branches may reveal it if, for instance, there is an unexpected liftoff after a descent. Even though we have reached 100% branch coverage, this test-suite has not managed to expose the fault in the code. Also note that even satisfying a stronger coverage criterion like the modified condition/decision coverage or even a stronger intra-procedural analysis will not expose the fault. To expose the fault in this example one would need to exercise combinations of decisions from different methods.

III. Background

This section describes the background work on which the metric proposed on this paper is inspired. Namely, we cover the concept of Spectrum-based Reasoning (SR) — which is amongst the best performing spectrum-based fault localization approaches [10] —and detail previous attempts to define a diagnosability metric.

A. Spectrum-based Reasoning (SR)

SR reasons about observed system executions and their outcomes to derive diagnoses that can explain faulty behavior in software [11]. In SR, the following is given:

- A finite set $\mathcal{C} = \langle c_1, c_2, ..., c_M \rangle$ of M system components. Components can be any source code artifact of arbitrary granularity such as a class, a method, a statement, or a branch [5];
- A finite set $\mathcal{T} = \langle t_1, t_2, ..., t_N \rangle$ of N system transactions, which can be seen as records of a system execution, such as, *e.g.*, test cases;
- The outcome of system transactions is encoded in the error vector $e = \langle e_1, e_2, ..., e_N \rangle$, where $e_i = 1$ if transaction t_i has failed and $e_i = 0$ otherwise;
- A $N \times M$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij}$ encodes the involvement of component c_j in transaction t_i.

The pair $(\mathcal{A}, e)$ is commonly referred to as spectrum [5]. Several types of spectra exist. The most commonly used is called hit-spectrum, where the activity matrix is encoded in terms of binary *hit* (1) and *not hit* (0) flags, i.e., $\mathcal{A}_{ij} = 1$ if c_j is involved in t_i and $\mathcal{A}_{ij} = 0$ otherwise.

Prior approaches using spectra were based on a so-called similarity coefficient to find a correlation between a component c_j's activity (*i.e.*, $\langle \mathcal{A}_{ij} | i \in 1..N \rangle$) and the observed transaction outcomes encoded in error vector e [6], [7], [10], [12], [13]. SR relies instead on a reasoning approach that leverages a Bayesian reasoning framework to diagnose the system. SR was also shown to outperform similarity-based approaches [11]. The two main steps of SR are candidate generation and candidate ranking:

1) Candidate Generation: The first step in SR is to generate a set $\mathcal{D} = \langle d_1, d_2, ..., d_k \rangle$ of diagnosis candidates. Each diagnosis candidate d_k is a subset of $\mathcal{C}$, and d_k is said to be valid if every failed transaction involved at least one component from d_k. A candidate d_k is *minimal* if no valid candidate d' is contained in d_k. We are only interested in minimal candidates, as they can subsume others of higher cardinality. Heuristic approaches to finding these minimal candidates, which is an instance of the *minimal hitting set* problem, thus NP-hard, include STACCATO [14], SAFARI [15] and MHS^2 [16].

2) Candidate Ranking: For each candidate d_k, their fault probability is calculated using the Naïve Bayes rule

$$\Pr(d_k \mid (A, e)) = \Pr(d_k) \cdot \prod_{i \in 1..N} \frac{\Pr((A_i, e_i) \mid d_k)}{\Pr(A_i)} \quad (1)$$

$\Pr(d_k)$ estimates the probability that a candidate, without further evidence, is responsible for the observed, erroneous behavior. Typically, candidates of higher cardinality have a lower *prior probability* of being faulty, since conditional independence is assumed throughout the process [11]. The denominator $\Pr(A_i)$ is a normalizing term that is identical for all candidates. Lastly, $\Pr((A_i, e_i) \mid d_k)$ is used to bias the prior probability taking observations from the program spectrum into account. One approach to computing this term is to use Maximum Likelihood Estimation (MLE) to maximize the probability that each component involved in a transaction i behaves nominally. Further details on the inner workings of the candidate ranking step are detailed in [11].

B. Measuring Quality of Diagnosis

To measure the accuracy of fault-localization approaches, the cost of diagnosis C_d metric is often used [10], [11], [17], [18]. It measures the number of candidates that need to be inspected until the real faulty candidate is reached, given that the candidates are being inspected by descending order of probability.[4] A value of 0 for C_d indicates an ideal diagnostic report where the faulty candidate is at the top of the ranking and thus no spurious code inspections will occur. Another common metric is Wasted Effort (or merely Effort), that normalizes C_d over the total number of components in the system so that the metric ranges from 0 (optimal value – no developer time wasted chasing wrong leads) to 1 (worst value – states that the whole system will be inspected until the fault is reached) in all cases.

Effort measurements assume perfect fault understanding, meaning that when the real faulty candidate is inspected, it is correctly identified as such. This assumption may not always hold [19], but there are approaches to mitigate it (e.g., [20]).

[4]Or likelihood score, depending on the fault-localization approach used.

C. Diagnosability Assessment by Measuring Matrix Density

Previous work [8] has used matrix density (ρ) as a measure for diagnosability:

$$\rho = \frac{\sum_{i,j} A_{ij}}{N \times M} \tag{2}$$

The intuition is to find an optimal matrix density such that every transaction observed reduces the entropy of the diagnostic report set $\mathcal{R} = \langle \Pr(d_k|(A,e))|d_k \in \mathcal{D}\rangle$. It has been previously demonstrated that the information gain can be modeled as:

$$\begin{aligned} IG(t_g) = &- \Pr(e_g = 1) \cdot \log_2(\Pr(e_g = 1)) \\ &- \Pr(e_g = 0) \cdot \log_2(\Pr(e_g = 0)) \end{aligned} \tag{3}$$

where $\Pr(e_g = 1)$ is the probability of observing an error in transaction t_g, conversely $\Pr(e_g = 0)$ is the probability of observing nominal behavior. Optimal information gain ($IG(t_g) = 1$) is achieved when $\Pr(e_g = 1) = \Pr(e_g = 0) = 0.5$. With the assumption that transaction activity is normally distributed, then it follows that a transaction's average component activation rate equals the overall matrix density. Thus, it can be said that $\Pr(e_g = 1) = \rho$, yielding $\rho = 0.5$ as the ideal value for diagnosis using SR approaches [8]. Density was also leveraged by Campos *et al.* to guide automated test generation [17]. This work shows that density-guided test-suites managed to reduce diagnostic effort when compared to using branch coverage as the fitness function for the generation.

D. Diagnosability Assessment by Measuring Uniqueness

Baudry *et al.* propose a diagnosability metric that tracks the number of *dynamic basic blocks* in a system [9]. Dynamic basic blocks, which other authors also call *ambiguity groups* [21], correspond to sets of components that exhibit the same involvement pattern across the entire test-suite. For diagnosing a system, the more ambiguity groups there are, the less accurate the diagnostic report can be, because one cannot distinguish among components in a given ambiguity group, as they all show the same involvement pattern across every transaction.

This metric, that we call uniqueness, can be used to ensure that the test-suite is able to break as many ambiguity groups as possible. A matrix $\mathcal{A}$ decomposes the system into a partition $G = g_1, g_2, ..., g_L$ of subsets of all components with identical columns in $\mathcal{A}$. Then, measuring the uniqueness $\mathcal{U}$ of a system can be done by

$$\mathcal{U} = \frac{|G|}{M} \tag{4}$$

When $\mathcal{U} = {}^1/_M$ all components belong to the same ambiguity group. When $\mathcal{U} = 1$, all components can be uniquely identified.

IV. Diagnosability Metric

This section presents the DDU metric. First, we detail a method for quantifying the exhaustiveness of a test suite using the notion of entropy, motivated by the optimal diagnosability scenario. Although we use SR in our motivation, the entropy approach can be applied to other spectrum-based fault localization strategies as well, because it focuses on isolating diagnostic candidates. We show that entropy may not be suitable in practice due to the number of transactions needed to reach an ideal spectrum. Finally, we propose the DDU metric as a relaxed alternative, based on previous work that uses density as an indicator for diagnosability.

A. Activity Matrix Entropy

To maximize the effectiveness of SR approaches, the ideal activity matrix is one that contains every combination of component activations, since it follows that every possible fault candidate in the system is exercised. A metric that accurately captures this exhaustiveness is entropy – the measure of uncertainty in a random variable. Shannon Entropy [22] is given by

$$H(X) = -\sum_i P(x_i) \cdot \log_2(P(x_i)) \tag{5}$$

in this context, X is the set of unique transaction activities in the spectrum matrix. $P(x_i)$ is the probability of selecting a transaction $t \in \mathcal{T}$ and it having the same activity pattern as x_i. When $H(X)$ is maximal, it means that all possible transactions are present in the spectrum. For a system with M components, maximum entropy is M *shannons* (*i.e.*, number of bits required to represent the test suite). Therefore, we can normalize it to ${}^{H(X)}/_M$. Matrices with a normalized entropy of 1.0 would, then, be able to efficiently diagnose any fault (single or multiple) provided that the error detection oracles that classify transactions as faulty are sufficiently accurate.

The main downside of using entropy as a measure of diagnosability is that one would need $2^M - 1$ tests to achieve this ideal spectrum (and thus a normalized entropy of 1.0). In practice, some transaction activities are impossible to be generated, either due to the system's topology or due to the existence of ambiguity groups: a set of components that always exhibit the same activity pattern.[5]

B. DDU

Our DDU is detailed next. Its goal is to capture several structural properties of the activity matrix that make it ideal for diagnosing, while avoiding the combinatorial explosion of the optimal entropy approach. We start by considering activity matrix density as the basis for our approach, and then propose the diversity and uniqueness enhancements so that the impractical assumptions of the base approach can be lifted.

1) Density: The ρ metric captures the density of a system. Its ideal value for minimizing the diagnostic report entropy is 0.5, as shown in the work of González-Sanchez *et al.* [8]. It is also straightforward to show the optimality of the value of 0.5 for the density measurement by induction. Suppose that we have an activity matrix $\mathcal{A}_1$, which is optimal for diagnosis. Suppose also that we want to add a new component c' to our system. To preserve optimality, we would need to repeat the

[5] An example of an ambiguity group is the set of statements in a basic block.

optimal sub-matrix $\mathcal{A}_1$ both when c' is active and when it is inactive. Therefore, c' involvement rate will be 0.5. Since $\rho = 0.5$ is our optimal target value, we propose a normalized metric ρ' where its upper bound (1.0) is the actual target

$$\rho' = 1 - |1 - 2 \cdot \rho| \tag{6}$$

and the lower bound 0 means that every cell in the matrix contains the same value. However, this optimal target is only valid assuming that all transactions in the activity matrix are *distinct*. Such assumption is not encoded in the metric itself (see Equation (2)). This means that a matrix with no diversity (depicted in the example from Figure 2a) is able to reach the ideal value for the ρ' metric.

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	1	1	0	0
t_3	1	1	0	0
t_4	1	1	0	0

(a) No Test Diversity. $\rho' = 1.0 \quad \mathcal{G} = 0.0$

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	0	1	1
t_3	1	1	1	0
t_4	0	0	0	1

(b) Test Diversity. $\rho' = 1.0 \quad \mathcal{G} = 1.0$

Fig. 2: Impact of diversity on ρ' and $\mathcal{G}$.

2) Diversity: The first enhancement we propose to the ρ' analysis is to encode a check for test diversity. In a diagnostic sense, the advantage of having considerable variety in the recorded transactions is related to the fact that each diagnostic candidate's posterior probabilities of being faulty are updated with each observed transaction. If a given transaction is failing, it means that the diagnostic candidates whose components are active in that transaction are further indicted as being faulty – so their fault probability will increase. Conversely, if the transaction is passing, then it means that the candidates that are active in the transaction will be further exonerated from being faulty – and their fault probability will decrease. Having such diversity means that more diagnostic candidates will have their fault probabilities updated so that they are consistent with the observations, leading to a more accurate representation of the state of the system.

We use the Gini-Simpson index to measure diversity ($\mathcal{G}$) [23]. The $\mathcal{G}$ metric computes the probability of two elements selected at random being of different kinds:

$$\mathcal{G} = 1 - \frac{\sum n \times (n-1)}{N \times (N-1)} \tag{7}$$

where n is the number of tests that share the same activity. When $\mathcal{G} = 1$, every test has a different activity pattern. When $\mathcal{G} = 0$, all tests have equal activity. Figures 2a and 2b depict examples of repeated and diverse test cases, respectively. We can see that the ρ' metric by itself cannot distinguish between the two matrices, as they have the same density. If we also account for diversity, the two matrices can be distinguished.

3) Uniqueness: The second extension we propose has to do with checking for ambiguity in component activity patterns. If two or more components are ambiguous, like components c_1 and c_2 from the example in Figure 3a, then they form an

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	0	1	1
t_3	1	1	1	0
t_4	0	0	0	1

(a) Component Ambiguity. $\rho' = 1.0 \quad \mathcal{G} = 1.0 \quad \mathcal{U} = 0.75$

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	1	1	0
t_3	1	0	1	1
t_4	0	0	0	1

(b) No Component Ambiguity. $\rho' = 1.0 \quad \mathcal{G} = 1.0 \quad \mathcal{U} = 1.0$

Fig. 3: Impact of component ambiguity on ρ', $\mathcal{G}$ and $\mathcal{U}$.

ambiguity group (see Section III-D), and it is impossible to distinguish between these components to provide a minimal diagnosis if tests t_1 and t_3 fail. As finding potential diagnostic candidates can be reduced to a set-cover/minimal-hitting-set problem, then two things may happen as a result of breaking an ambiguity group and having those components being tested independently. One is that some diagnostic candidates containing components from that ambiguity group can become inconsistent with the observations and thus would be removed from the set of possible diagnostic candidates, improving the tractability of the bayesian update step of the SR approach. The other is that diagnostic candidates will be of lower cardinality, thus improving our confidence in the accuracy of diagnosis. This happens because, as faults are considered to be independent, then the probability of having multiple faults as the explanation for the system's behavior is generally several orders of magnitude lower when compared to low-cardinality candidates.[6]

We use a check for uniqueness ($\mathcal{U}$) as described in Equation (4) to quantify ambiguity. Uniqueness is also used by Baudry *et al.* to measure diagnosability [9]. However, we argue that uniqueness alone does not provide sufficient insight into the suite's diagnostic ability. Particularly, it does not guarantee that component activations are combined in different ways to further exonerate or indict multiple-fault candidates. In that aspect, information regarding the diversity of a suite provides further insight.

4) Combining Diagnostic Predictors: Our last step is to provide a relaxed version of entropy (which we call DDU) by combining the three aforementioned metrics that assess the key properties (i.e., necessary and sufficient) a coverage matrix ought to have to ensure proper diagnosability:

$$\text{DDU} = \rho' \times \mathcal{G} \times \mathcal{U} \tag{8}$$

and its ideal value is 1.0. We reduce ρ', $\mathcal{G}$ and $\mathcal{U}$ into a single value by means of multiplication. The reason being that since in each term the value of 0.0 corresponds to the worst-case and 1.0 to the ideal case, we are able to leverage properties of multiplication such as multiplicative identity and the zero property.

V. Empirical Evaluation

To evaluate the proposed metric with regard to its ability to diagnose faults, we aim to address the following research

[6]Thus having to be supported by many observations for our confidence on that diagnosis to increase.

questions:

RQ1: Is the DDU metric more accurate than the state-of-the-art in diagnosability assessment?

RQ2: How close does the DDU metric come to the (ideal yet intractable) full entropy?

RQ3: Does optimizing a test-suite with regard to DDU result in better diagnosability than optimizing adequacy metrics such as branch-coverage in traditional scenarios?

RQ1 asks if there is a benefit in utilizing the proposed approach as opposed to density and uniqueness – which have been used in related work. **RQ2** is concerned with assessing if DDU shares a statistical relationship with entropy – the measurement whose maximal value describes an optimal (yet intractable and impractical) coverage matrix. **RQ3** asks if using DDU as an indicator of the diagnostic ability of a test-suite is more accurate than using standard adequacy measurements like branch-coverage in a setting with real faults.

A. Experimental Setup

Our empirical evaluation compares DDU to several metrics in use today. To effectively compare the diagnosability of test-suites of a given program that maximize a specific metric, we leverage a test-generation approach. EvoSuite[7] is a tool that employs Search-based Software Testing (SST) approaches to create new test cases. It applies Genetic Algorithms (GAs) to minimize a *fitness function* which describes the distance to an optimal solution. The metrics to be compared are DDU – our proposed measurement; density and uniqueness to be able to answer **RQ1**; entropy to answer **RQ2** and lastly branch-coverage for **RQ3**. These metrics were encoded as *fitness functions* in the EvoSuite framework. As the GA in EvoSuite tries to minimize the value of a function over a test suite TS, the *fitness functions* for each metric $\mathcal{M}$ are as follows

$$f_{\mathcal{M}}(TS) = |\mathcal{O}_{\mathcal{M}} - \mathcal{M}(TS)| \tag{9}$$

where $\mathcal{O}_{\mathcal{M}}$ is the optimal value of metric M (*e.g.,* 1.0 for the case of branch-coverage, and 0.5 for density), and $M(TS)$ is the result of applying metric M to test suite TS. To account for the randomness of EvoSuite's GA, we repeated each test-suite generation experiment 10 times. EvoSuite's maximum search time budget was set to 600 seconds, which follows the setup of previous studies also using the tool [17].

EvoSuite by itself does not generate fault-finding *oracles* – otherwise, a model of correct behavior would have to be provided. Instead, it creates assertions based on static and dynamic analyses of the project's source code. This means that if we run the generated test-suite against the same source code used for said generation, all tests will pass (provided the code is deterministic[8]). Thus, if the source code submitted for test-generation contains faults, no generated test oracle will expose them.

For the experiments comparing with the state-of-the-art and the idealistic approach (to answer **RQ1** and **RQ2**, respectively), we need a controlled environment so that oracle quality (which in itself is an orthogonal factor) does not affect results. Therefore, the experiment described in Section V-B mutates the program spectrum of generated test-suites to contain seeded faults and seeded failing tests. In each experiment a set of components were considered as faulty, and tests that exercise them were set as failing according to an *oracle quality probability* – in our experiments, the *oracle quality* is 0.75, meaning that whenever a faulty component is involved in a test, there is a 75% chance that the test will be set as failing. The chosen value is a compromise between perfect error detection (*i.e.*, *oracle quality* of 1) and essentially random error detection (*oracle quality* of 0.5) This fault injection approach is common practice among controlled, theoretical evaluations of spectrum-based diagnosis [8], [24].

For assessing the applicability in real world scenarios and to answer **RQ3**, we need real life bugs and fixes. Therefore, in Section V-C we make use of Defects4J[9] – a software fault catalog – to generate test-suites from fixed versions of a program and then gather program spectra by testing the corresponding faulty version.

Spectrum gathering was performed at the branch granularity for both experiments, so every component in our subjects' coverage matrices corresponds to a method branch – this way we can fairly compare our approach to branch coverage. Each program spectrum gathered in the previous step is then diagnosed using the automated diagnosis tool Crowbar.[10] This tool implements the Spectrum-based Reasoning approach described in Section III-A, and generates a ranked list of diagnostic candidates for the observed failures.

For a given subject program, to compare the diagnosability of a test-suite generated by the DDU criterion with the one generated by a criterion C, we use the following metric

$$\Delta_{\text{Effort}}(C) = \text{Effort}_C - \text{Effort}_{\text{DDU}} \tag{10}$$

where $\text{Effort}_{\text{DDU}}$ is the effort to diagnose using the test-suite generated with the DDU criterion and Effort_C is the effort to diagnose with the test suite by maximizing some criterion C. Effort takes as input the ranked list of diagnostic candidates from Crowbar and estimates quality of diagnosis as described in Section III-B. The $\Delta_{\text{Effort}}(C)$ metric ranges from -1 to 1. Positive values of $\Delta_{\text{Effort}}(C)$ mean that the bug is found faster in diagnoses that use the DDU generated

[7] EvoSuite tool is available at http://www.evosuite.org. Version 1.0.2 was used for experiments (accessed February 2017).

[8] EvoSuite also tries to replicate the state of the environment at each test-run so that even some non-deterministic functionality such as random number generation can be tested.

[9] Defects4J tool is available at https://github.com/rjust/defects4j. Version 1.0.1 was used for experiments (accessed February 2017).

[10] Crowbar tool is available at https://github.com/TQRG/crowbar-maven-plugin (accessed February 2017).

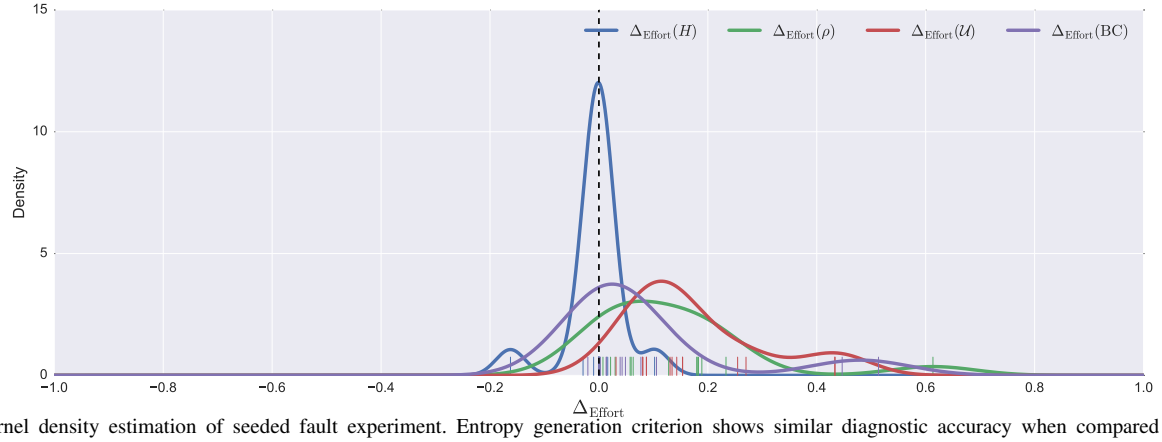

Fig. 4: Kernel density estimation of seeded fault experiment. Entropy generation criterion shows similar diagnostic accuracy when compared DDU. The remaining generation criteria exhibit worse diagnostic performance than DDU.

test suite. Negative values mean that the faulty component is ranked higher in the C-generated test-suite than the DDU one, thus requiring less spurious diagnostic inspections. $\Delta_{\text{Effort}}(C)$ of value 0 means that the faulty component is ranked with the same priority in both test generations.

We make use of kernel density estimation plots to show the $\Delta_{\text{Effort}}(C)$ values in Figures 4 and 5. Such plots estimate the probability density function of a variable, *i.e.*, they describe the relative likelihood (y-axis) for a random variable ($\Delta_{\text{Effort}}(C)$ in our case) to take on a given value (x-axis). In our experiments, the higher the density value at a certain value in the x-axis, the more instances with $\Delta_{\text{Effort}}(C)$ near that value were observed. Note that the observed data is shown as a *rug plot*, with tick marks along the x-axis (reminiscent of the tassels on a rug).

B. Diagnosing Seeded Faults

Our first experiment attempts to answer **RQ1** and **RQ2** by generating test-suites and seeding faults in their spectra in a controlled way. We same set of subjects as empirical evaluations from related work [17]. Namely, we use the open-source projects Apache Commons-Codec, Apache Commons-Compress, Apache Commons-Math and JodaTime. For each subject, we generate test-suites that optimize DDU, branch-coverage, entropy, density, and uniqueness. In total, 1050 program spectra were generated and diagnosed.

Experimental results are shown in Figure 4. When we consider the entropy generation, we can say that the resulting test-suites are very similar in terms of diagnosability compared to DDU, since $\Delta_{\text{Effort}}(H)$ is denser at the origin. For the remaining generation criteria, their respective Δ_{Effort} probability masses are shifted to $\Delta_{\text{Effort}} > 0$, so their diagnostic reports perform worse at diagnosing the faults than when DDU is utilized. In fact, our inspection of experimental results reveals that, when optimizing branch-coverage, 78% of scenarios showed lower diagnostic accuracy when compared to DDU. For both the density-optimized and uniqueness-optimized test generations – which are the state-of-the-art measurements for test-suite diagnosability – this figure rises to 100% of scenarios.

TABLE I: Metric results for the seeded faults experiment.

Subject	Median / Size / Correlation / Correlation **p-value**				
	H	**DDU**	ρ	$\mathcal{U}$	**BC**
Apache Commons-Codec	2.65×10^{-2}	0.620	0.476	0.669	0.910
	177	170	126	81	177
	N.A.	0.957	0.658	0.902	0.793
	N.A.	2.71×10^{-3}	1.98×10^{-2}	3.58×10^{-2}	2.08×10^{-3}
Apache Commons-Compress	4.66×10^{-2}	0.962	0.510	0.669	0.825
	108	108	30.5	29.5	126
	N.A.	0.999	0.999	0.873	0.968
	N.A.	1.08×10^{-6}	7.51×10^{-7}	1.47×10^{-3}	9.62×10^{-4}
Apache Commons-Math	4.36×10^{-2}	0.818	0.424	0.659	0.922
	497	467	402	246	528.5
	N.A.	0.989	0.905	0.725	0.885
	N.A.	4.68×10^{-4}	1.85×10^{-2}	4.79×10^{-2}	2.31×10^{-2}
JodaTime	1.580×10^{-2}	0.582	0.369	0.417	0.790
	265	265	267	171	267
	N.A.	0.976	0.674	0.921	0.654
	N.A.	8.54×10^{-4}	1.60×10^{-2}	2.59×10^{-2}	2.09×10^{-2}

We show in Table I the dominant metric median values for each generation criterion along with the median number of tests generated. By dominant metric we mean the metric which that particular test generation was trying to optimize. Along with the median value we also show (where available) the metric's Pearson correlation with entropy (denoted by r_H) and the p-value of the correlation. With 95% confidence, we can say that the correlation values shown are statistically significant. DDU exhibits a high correlation with entropy, having $r_H > 0.95$ for all subjects. In all other generation criteria, the correlation with entropy fluctuates considerably between subjects. Also, note that for both ρ and branch-coverage criteria, their dominant mean values approach the theoretical optima (at 0.5 and 1.0, respectively) while Δ_{Effort} still shows that DDU test generation was able to produce suites with better diagnostic accuracy.

Revisiting the first research question:

> **RQ1:** Is the DDU metric more accurate than the state-of-the-art in diagnosability assessment?

A: There is a clear benefit in optimizing a suite with regard to DDU compared to density if we consider the effort of finding faults in a system. This is evidenced by the fact that 100% of scenarios in our seeded fault experiment show improved diagnostic accuracy when using DDU when compared to the state-of-the-art density and uniqueness measurements.

If we look at the second research question:

> **RQ2:** How close does the DDU metric come to the (ideal yet intractable) full entropy?

A: Table I shows a strong correlation between entropy and DDU, with a Pearson correlation value above 0.95 for all subjects. Correlation of other metrics is much lower and varies greatly across subjects. Thus, we can conclude that DDU closely captures the characteristics of entropy.

The reader might then pose the question: if maximal entropy does indeed correspond to the optimal coverage matrix, why should one avoid using it as the diagnosability metric? While we agree that in automated test generation settings entropy can be plugged as the fitness function to optimize,[11] for manual test generation entropy will yield very small values for any complex system, as one can see from Table I. In fact, for a system composed of only 30 components, the number of tests needed to reach entropy of 1.0 surpasses the billion mark. This makes it difficult for developers to leverage information out of their test-suite's entropy value to gauge when can one confidently stop writing further tests.

C. Diagnosing Real Faults

We used the DEFECTS4J database [25] for sourcing the experimental subjects. DEFECTS4J is a database and framework that contains 357 real software bugs from 5 open source projects. For each bug, the framework provides faulty and fixed versions of the program, a test suite exposing the bug, and the fault location in the code. The idea behind DEFECTS4J is to allow for reproducible research in software testing using real-world examples of bugs, rather than using the more common hand-seeded faults or mutants. In our evaluation, we generate test suites for each of DEFECTS4J's 357 catalogued bugs, using both branch-coverage and DDU as EVOSUITE's fitness functions, and then compare the two generated suites with regard to their diagnosability and adequacy. The experiments' methodology is as follows. For every bug in DEFECTS4J's catalog, we use EVOSUITE to generate test suites for the fixed version of the program. The test suites are executed against the faulty program versions. This means that any test failure is due to the bug – which is the delta between the faulty and fixed program versions.

Out of the 357 catalogued bugs in DEFECTS4J, not all were considered for analysis. Scenarios were discarded due to the following reasons:

- EVOSUITE returned an empty suite;
- The generated suite did not compile or produced a runtime error;
- No failing tests were present in either DDU or branch-coverage criteria for generating test suites.

In total, 171 scenarios were filtered out. The remaining 186 listed in Table II are fit for analysis and their results are used throughout this section.

TABLE II: DEFECTS4J Projects.

Identifier	Project Name	# Scenarios	Considered
Chart	JFreechart	26	1, 4, 6, 8–11, 13–15, 18, 20, 22, 24, 26
Closure	Closure Compiler	133	3, 4, 7, 9, 12, 14–17, 19, 20–28, 30 33–35, 39, 43, 44, 46–49, 51, 52 54–56, 58, 63, 65, 66, 67, 69, 71–74 76–78, 82, 85, 87, 107, 108, 110–113 115, 116, 118, 119, 124, 126, 127, 129–132
Lang	Apache Commons-Lang	65	1–7, 9–14, 16, 17, 19, 21, 22, 24–28, 30, 31, 33, 36, 38–42, 46, 47, 49, 50–57, 59–61, 65
Math	Apache Commons-Math	106	1–10, 14–16, 18–20, 24–27, 29, 30, 32, 34, 35, 37–42, 44–46, 48–56, 100, 101, 103, 105, 106
Time	JodaTime	27	6, 8, 12, 15, 21, 22, 26, 27

TABLE III: Metric medians and statistical tests.

	Branch-Coverage Generation	DDU Generation
Branch Coverage	0.85	0.75
DDU	0.10	0.42
Suite Size	291	374
Effort	0.31	0.10
Shapiro-Wilk	$W = 0.92$ p-value $= 1.70\times10^{-8}$	$W = 0.85$ p-value $= 1.05\times10^{-12}$
Wilcoxon Signed-rank	$Z = 2335.0$ p-value $= 3.50\times10^{-13}$	

Experimental results are shown in Figure 5. Results are shown per-subject. We can see that for every subject in the DEFECTS4J catalog, all their estimated probability density funtions are shifted towards $\Delta_{\text{Effort}}(\text{BC}) > 0$, meaning that the majority of instances have better diagnostic accuracy when test generation optimizes DDU. In fact, our experiments reveal that 77% of scenarios (144 in total) yield a positive $\Delta_{\text{Effort}}(\text{BC})$.

We performed statistical tests to assess whether the gathered metrics yielded statistically significant results. Table III shows the relevant statistics. The first four rows show the median values for branch-coverage, DDU, generated suite size and diagnosis effort for both EVOSUITE test generations. As to be expected, the median branch-coverage is higher in the branch-coverage-maximizing generation. Conversely, the DDU criterion yields the higher DDU. Results in the effort row corroborate our observations from Figure 5 – the test suites optimizing DDU take on average less effort to diagnose the fault. In fact, our results show that the effort reduction when considering DDU over branch-coverage is 34% on average. However, this fact alone does not guarantee that the results are

[11] Because tools like EVOSUITE can be configured with a time budget as another stopping criteria.

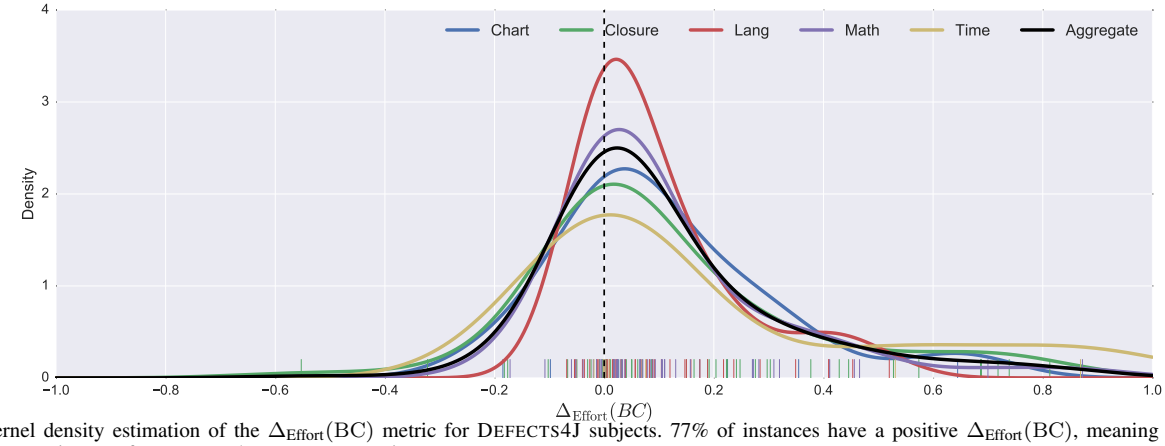

Fig. 5: Kernel density estimation of the $\Delta_{\text{Effort}}(\text{BC})$ metric for DEFECTS4J subjects. 77% of instances have a positive $\Delta_{\text{Effort}}(\text{BC})$, meaning that branch-coverage generations perform worse than DDU generations.

significant, which prompted us to perform statistical tests. The first test performed was the Shapiro-Wilk test for normality of effort data for both generations. The results, which can be seen in the fourth row of Table III, tell us that the distributions are not normal, with confidence of 99%.

Given that the effort data is not normally distributed and that each observation is paired, we use the non-parametrical statistical hypothesis test Wilcoxon signed-rank. Our null-hypothesis is that the median difference between the two observations (*i.e.*, Δ_{Effort}) is zero. The fifth row in Table III shows the resulting Z statistic and p-value. With 99% confidence, we can refute the null-hypothesis. Revisiting **RQ3**:

> **RQ3:** Does optimizing a test-suite with regard to DDU result in better diagnosability than optimizing adequacy metrics such as branch-coverage in traditional scenarios?

A: Since the median effort in the DDU generation is lower – the reduction amounting to 34% on average – we can say that optimizing for DDU produces better, statistically significant, diagnoses when compared to test suites that optimize for branch-coverage.

VI. Discussion

We discuss the practical implications of our findings from the previous section, as well as outline the potential threats to their validity.

A. Practical Implications

DDU was shown to be useful for evaluating the quality of a test-suite. But what are the practical implications of this finding? We outline such assessments next.

- We argue that the DDU analysis can suggest an ideal balance between unit tests and system tests (*i.e.,* when DDU reaches its optimal value) due to its density term. We are then able to compare the balanced suites to ones created following testing practices currently established at software development companies. For instance, Google suggests a 70%/20%/10% split between unit, system and end-to-end tests in a suite.[12] Is this split indeed ideal in terms of diagnostic accuracy? We believe a DDU analysis can provide guidance as to what the answer is.
- We expect the DDU analysis to be used as the first step of a test design strategy that aims to increase diagnostic accuracy of a suite. For that, we envision that new test patterns that focus on optimizing diagnosability will need to be researched and incorporated in established test strategy corpora such as [26].
- In coverage metrics, it is straightforward to visualize the analysis of a system so that users know what code components were left untested, highlighting where to focus when writing new test cases. Is there a way to visualize DDU analysis in a similar way? We envision that visualization approaches for program comprehension, such as EXTRAVIS [27] and PANGOLIN [28], will constitute a solid starting point for a study on visual, interactive and actionable ways to convey DDU information.
- We show that DDU depicts the diagnosability of spectrum-based fault localization approaches. However, our intuition is that DDU is general and applies to any diagnosis technique that uses a failing test suite as the basis for locating faults. We plan to investigate this hypothesis as future work.
- DDU provides an assessment of the diagnostic effectiveness of a given test suite. It remains to be seen if that can also be said for assessing the fault finding effectiveness, which is also a good avenue for future work. In the meantime we consider our metric to be a complement to adequacy metrics, and envision that testers will employ a hybrid approach that relies on branch coverage and DDU to assess adequacy and diagnosability, respectively.

B. Threats to Validity

The main threats to validity of this study are related to external validity. When choosing the projects for our study, our aim was to opt for projects that resemble a general large-sized application being worked on by several people. To reduce

[12]Google Testing blog: Just Say No to More End-to-End Tests. http://goo.gl/S5HhZ7 (accessed February 2017).

selection bias and facilitate the comparison of our results, we decided to use the real-world scenarios described in the DEFECTS4J database. Another threat to external validity relates to the choice of test suites generated by EVOSUITE. Additional research is needed to see how the metric behaves both with different test-generation frameworks (such as RANDOOP [29]) and with hand-written test cases.

A potential threat to construct validity relates to the choice of effort as indicator for diagnosability. However, as argued in Section III-B this choice reflects the effort that a programmer with minimal knowledge about the system would require to effectively pinpoint all the faults that explain the observed failures.

The main threat to internal validity lies in the complexity of several of the tools used in our experiments, most notably the EVOSUITE test generator and our diagnosis tool.

VII. RELATED WORK

Related work in the assessment of the diagnosability of a test suite has focused on three key areas: test-suite minimization and generation strategies, and assessing oracle quality.

The topic of test-suite minimization is a prime candidate for our approach, since it has been shown that there is a tradeoff between reducing tests and the suite's fault localization effectiveness [30]. In minimization settings, one tries to reduce the number of tests (and thus its overall running time) while still ensuring that an adequacy criterion – usually branch coverage – is not greatly affected. Current minimization strategies can often improve the diversity score of a coverage matrix by removing tests with identical coverage patterns [31] at the cost of overlooking density and uniqueness, which we argue are of key importance to assess diagnosability. The uniqueness property is also exploited by Xuan *et al.*, with a *test-case purification* approach that separates a test-case into multiple smaller tests [32]. This approach overlooks the fact that density will decrease, along with the ability to diagnose a multiple-fault scenario.

Current test-suite minimization frameworks that take adequacy criteria into account could also benefit from our approach to preserve diagnostic accuracy if a multi-objective optimization (such as, *e.g.*, [33], [34]) to also account for DDU is employed. This paves an interesting avenue for future work.

On the test-suite generation front, previous work has also started considering diagnosability as a generation criterion. The work of Campos *et al.*, which generated tests that would converge towards coverage matrix densities of 0.5 [17], has paved the way for creating improved measurements like DDU. Checks for diversity and uniqueness were not explicitly added, and we show when we answer **RQ1** in Section V that the density criterion produces results that are less diagnostically accurate. Another approach to suite generation is one by Artzi *et al.*, that proposes an online approach that leverages *concolic analysis* to generate tests that are similar to existing failing tests in a system [35].

Lastly, we highlight some of the work targeting diagnosability by improving test oracle accuracy. Schuler *et al.* propose *checked coverage* as a way of assessing oracle quality [36], [37]. *Checked coverage* tries to gauge whether the computed results from a test are actually being checked by the oracle. Wang *et al.* have proposed a way of addressing *coincidental correctness* – when a fault is executed but no failure is detected – by analyzing data and control-flow patterns [38]. Just *et al.* investigated the use of mutants to estimate oracle quality, and compared their performance against the use of real faults [39]. Their results suggest that a suite's mutation score is a better predictor of fault detection than code coverage. We consider this topic of assessing and improving oracle quality of critical importance towards test-suite diagnosability, but also orthogonal to DDU in that the two would complement each other.

VIII. CONCLUSION

This paper proposes a new metric to assess a test suite's diagnostic ability to detect fault using spectrum-based approaches given that there are failing tests. The intuition is that a test suite where each test is diverse in that it exercises as many combinations of components as possible is more exhaustive than one that merely focuses on maximizing code coverage. In fact, a variable number of covered elements is crucial for good fault isolation when tests fail, as it helps techniques like Spectrum-based Reasoning indict and exonerate both single-component faults and multiple-component faults.

The metric, coined DDU, tries to emulate the properties of calculating per-test coverage entropy, to ensure accurate diagnosability. Ideal diagnostic ability can be proved to exist when a suite reaches maximum entropy, however, the number of tests required to achieve that is impractical as the number of components in the system increases. DDU focuses on three particular properties of entropy: a) ensures that test cases are diverse; b) ensures that there are no ambiguous components; c) ensures that there is a proportional number of tests of distinct granularity; while still ensuring tractability.

An experiment was performed to assess DDU as a metric for diagnosability. It used the EVOSUITE tool to generate test suites for faulty programs from the DEFECTS4J catalog that would optimize different metrics. We observed a statistically significant increase in diagnostic performance of about 34% when locating faults by optimizing DDU compared to branch-coverage.

DDU paves the way for a more comprehensive use of test-suites, making them not only a great tool for error detection and requirements elicitation, but also an effective diagnostic aid when failures arise.

SOFTWARE ARTIFACTS

We provide a fork of EVOSUITE 1.0.2 implementing every generation criterion used in Section V at https://github.com/aperez/evosuite. A MAVEN plugin for determining the DDU value for Java projects is available at https://github.com/aperez/ddu-maven-plugin.

ACKNOWLEDGMENTS

We thank Maurício Aniche, Lara Crawford and Alexey Zagalsky for the useful feedback on previous versions of this paper. This material is based upon work supported by the scholarship number SFRH/BD/95339/2013 and project POCI-01-0145-FEDER-016718 from Fundação para a Ciência e Tecnologia (FCT), by ERDF COMPETE 2020 Programme, by EU Project STAMP ICT-16-10 No.731529 and by 4TU project "Big Software on The Run".

REFERENCES

[1] J. C. Miller and C. J. Maloney, "Systematic mistake analysis of digital computer programs," *Communications of the ACM*, vol. 6, no. 2, pp. 58–63, 1963.

[2] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[3] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University New Haven CT USA, 1980.

[4] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[5] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE) June 16, 1998*, 1998, pp. 83–90.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2005, pp. 273–282.

[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[8] A. González-Sanchez, H. Gross, and A. J. C. van Gemund, "Modeling the diagnostic efficiency of regression test suites," in *4th IEEE International Conference on Software Testing, Verification and Validation (ICST), Workshop Proceedings*, 2011, pp. 634–643.

[9] B. Baudry, F. Fleurey, and Y. L. Traon, "Improving test suites for efficient fault localization," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, 2006, pp. 82–91.

[10] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey of software fault localization," *IEEE Transactions on Software Engineering*, 2016.

[11] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2009, pp. 88–99.

[12] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[13] 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.

[14] R. Abreu and A. J. C. van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," in *8th Symposium on Abstraction, Reformulation, and Approximation (SARA)*, 2009.

[15] A. Feldman, G. M. Provan, and A. J. C. van Gemund, "Computing minimal diagnoses by greedy stochastic search," in *23rd AAAI Conference on Artificial Intelligence (AAAI)*, 2008, pp. 911–918.

[16] N. Cardoso and R. Abreu, "MHS^2: A map-reduce heuristic-driven minimal hitting set search algorithm," in *Multicore Software Engineering, Performance, and Tools*, ser. Lecture Notes in Computer Science, 2013, vol. 8063, pp. 25–36.

[17] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-Based Test Generation for Improved Fault Localization," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2013, 2013, pp. 257–267.

[18] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2013, pp. 314–324.

[19] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[20] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," in *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 2013, pp. 1–10.

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

[22] C. E. Shannon, "A mathematical theory of communication," *Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3–55, 2001.

[23] L. Jost, "Entropy and diversity," *Oikos*, vol. 113, no. 2, pp. 363–375, may 2006. [Online]. Available: http://dx.doi.org/10.1111/j.2006.0030-1299.14714.x

[24] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "A new bayesian approach to multiple intermittent fault diagnosis," in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009, pp. 653–658.

[25] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2014, pp. 437–440, tool demo.

[26] R. Binder, *Testing Object-oriented Systems: Models, Patterns, and Tools*, ser. Object Technology Series. Addison-Wesley, 2000.

[27] B. Cornelissen, A. Zaidman, and A. van Deursen, "A controlled experiment for program comprehension through trace visualization," *IEEE Trans. Software Eng.*, vol. 37, no. 3, pp. 341–355, 2011.

[28] A. Perez and R. Abreu, "Framing program comprehension as fault localization," *Journal of Software: Evolution and Process*, 2016.

[29] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, 2007, pp. 815–816.

[30] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 201–210.

[31] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, 2012, pp. 30–39.

[32] 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, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 52–63.

[33] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, vol. 83, no. 4, pp. 689–701, 2010.

[34] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating non-adequate test-case reduction," in *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[35] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, 2010, pp. 49–60.

[36] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, 2011, pp. 90–99.

[37] ——, "Checked coverage: an indicator for oracle quality," *Softw. Test., Verif. Reliab.*, vol. 23, no. 7, pp. 531–551, 2013.

[38] X. Wang, S. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 45–55.

[39] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 654–665.

TUD-SERG-2017-004
ISSN 1872-5392