A Theoretical and Empirical Analysis of Program Spectra Diagnosability

Perez, Alexandre; Abreu, Rui; Deursen, A. Van

DOI
10.1109/TSE.2019.2895640

Publication date
2019

Document Version
Final published version

Published in
IEEE Transactions on Software Engineering

A Theoretical and Empirical Analysis of Program Spectra Diagnosability

Alexandre Perez, *Member, IEEE*, Rui Abreu, *Senior Member, IEEE*, and Arie Van Deursen, *Member, IEEE*

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. We have performed a topology-based simulation of thousands of spectra and have found that DDU can effectively establish an upper bound on the effort to diagnose faults. Furthermore, our empirical experiments using the Defects4J dataset show that optimizing a test suite with respect to DDU yields a 34 percent gain in spectrum-based fault localization report accuracy when compared to the standard branch-coverage metric.

Index Terms—Testing, coverage, diagnosability

1 Introduction

This paper discusses the importance of measuring diagnosability of software, i.e., the ability of a program and its test suite to effectively and accurately locate faults when errors arise. It proposes DDU, a new metric for evaluating the diagnosability of a test-suite when applying spectrum-based fault localization approaches, and provides a thorough *theoretical* and *empirical* analysis of its effectiveness. Aimed at complementing adequacy measurements that focus on maximizing error detection of a suite, DDU provides an assessment on the effort required to pinpointing the root cause of potential failures. 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 can boost the accuracy of 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 to improve the diagnosability of the spectrum. 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 failing 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, one 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

- *A. Perez is with the Faculty of Engineering, University of Porto, Porto 4099-002, Portugal. E-mail: alexandre.perez@fe.up.pt.*
- *R. Abreu is with the INESC-ID and Instituto Superior Ténico, University of Lisbon, Lisbon 1649-004, Portugal. E-mail: rui@computer.org.*
- *A. van Deursen is with the Delft University of Technology, Delft 2628, The Netherlands. E-mail: arie.vandeursen@tudelft.nl.*

Manuscript received 14 Jan. 2018; revised 28 Dec. 2018; accepted 15 Jan. 2019. Date of publication 28 Jan. 2019; date of current version 11 Feb. 2021. (Corresponding author: Alexandre Perez.)
Recommended for acceptance by S. Kim.
Digital Object Identifier no. 10.1109/TSE.2019.2895640

1. Ice-cream cone software testing anti-pattern mentioned in Alister Scott's blog: http://goo.gl/bhXOrN (accessed January 2019).

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 value, 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.

We detail the optimal coverage matrix for achieving accurate spectrum-based fault localization. In this optimal scenario, 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: (1) density (ρ), ensuring components are frequently involved in tests; (2) test diversity ($\mathcal{G}$), ensuring components are tested in diverse combinations; and (3) 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 theoretical and empirical evaluations. The theoretical evaluation simulates a vast breadth of software systems and test suite compositions so that the range of DDU values can be effectively generated and analyzed in a holistic manner. Our simulation is built upon a tree-based representation of system structures—which we call topologies—that are randomly generated following phylogenetic processes. Topologies then guide the generation of multiple spectra, which are then fault-injected and diagnosed. This theoretical analysis reveals that DDU can effectively predict an upper-bound on the effort required to diagnose. We also empirically evaluate DDU by generating test suites for real-world 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 the state-of-the-art of diagnosability metrics such as density. The second empirical evaluation generates test-suites for a wide range of subjects in the Defects4J collection. We provide empirical evidence that optimizing a suite regarding DDU yields an increase of 34 percent in diagnostic accuracy when compared to test-suites that only consider branch-coverage as the optimization criterion and 17 percent when compared to optimizing mutation score.

This paper extends our previous work [10] by (1) providing a generalization to the information-theoretic reasoning behind targeting a certain optimal spectrum density value, (2) providing a large-scale evaluation of DDU through a topology-based program spectra simulation—so that we are able to generate and analyze a vast breadth of qualitatively distinct faulty spectra—, (3) expanding our evaluation by comparing the diagnostic effectiveness of DDU versus mutation coverage, and (4) expanding our discussion on the implications of using the DDU metric for assessing diagnosability.

2 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 Fig. 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 10 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 code path followed by t_1 to exonerate them from suspicion. When considering the test suite t_1 to t_4, the developer will have to 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 percent 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

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

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

		t_1	t_2	t_3	t_4	t'
1:	def groundDistance():	●	●		●	
2:	if underwater():	●	●		●	
3:	return surfaceDistance()		●		●	
4:	else:	●				
5:	return groundAltitude()	●				
6:	def groundAltitude ():	●		●	●	
7:	if landed():	●		●	●	
8:	return 0			●		
9:	else:	●				
10:	**return sub(GND, ALT)**	●				
11:	def sub (a,b):	●				●
12:	return a - b	●				●
	Pass/fail status:	✗	✓	✓	✓	✓

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

		t_1	t_2	t_3	t_4	t'
1:	def descend(increment):	●	●		●	●
2:	if landed():	●	●		●	●
3:	return Status.STOPPED	●			●	
4:	else:		●			●
5:	**descendMeters(increment)**		●			●
6:	return Status.DESCENDING		●			●
7:	def ascend(increment):	●		●		●
8:	if landed():	●		●		●
9:	liftoff()	●				
10:	return Status.LIFTOFF	●				
11:	else:			●		●
12:	**ascendFeet(increment)**			●		●
13:	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.

fairly large numbers in a real world scenario. So, even though test suite t_1 to t_4 has 100 percent branch-coverage, it does not provide many diagnostic clues. Adding test t' to the test suite will, in fact, not result in a change in coverage, but it will positively impact our proposed metric, as well as further isolate the fault.

The second example, depicted in Fig. 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. Even though test suite t_1 to t_4 has reached 100 percent 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. 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, as is depicted in test t', which also positively impacts our proposed metric.

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

Fig. 2. Spectrum of a system with 3 components and 4 transactions.

3 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 [11]—and detail previous attempts to define a diagnosability metric.

3.1 Spectrum-Based Reasoning (SR)

SR reasons about observed system executions and their outcomes to derive diagnoses that can explain faulty behavior in software [12]. 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 $e_j = 1$ if transaction t_j has failed and $e_j = 0$ otherwise;
- A $M \times N$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij}$ encodes the involvement of component c_i in transaction t_j.

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_i is involved in t_j and $\mathcal{A}_{ij} = 0$ otherwise. An example to be used throughout this section is shown in Fig. 2, which is analogous to the depiction of spectra from Fig. 1. This spectrum consists of four transactions (i.e., executions) of a system composed of three components. Transactions t_1, t_2 and t_3 fail, whereas in t_4 no error was observed.

Prior approaches using spectra were based on a so-called similarity coefficient to find a correlation between a component c_i's activity (i.e., $\langle \mathcal{A}_{ij} | j \in 1..N \rangle$) and the observed transaction outcomes encoded in error vector e [6], [7], [11], [13], [14]. 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 [12]. The two main steps of SR are candidate generation and candidate ranking:

3.1.1 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 [15], SAFARI [16] and MHS^2 [17].

In our example from Fig. 2, the collection of minimal diagnostic candidates that can explain the erroneous behavior is

- $d_1 = \langle c_1, c_2 \rangle$
- $d_2 = \langle c_1, c_3 \rangle$

3.1.2 Candidate Ranking

For each candidate d_k, their fault probability is calculated using the Naïve Bayes rule

$$\Pr(d_k \mid (\mathcal{A}, e)) = \Pr(d_k) \cdot \prod_{j \in 1..N} \frac{\Pr((\mathcal{A}_j, e_j) \mid d_k)}{\Pr(\mathcal{A}_j)}. \quad (1)$$

Let A_j be short for $\langle \mathcal{A}_{ij} | i \in 1..M \rangle$—i.e., the jth column of matrix $\mathcal{A}$, represented by a set encoding all component involvements in test t_j. The denominator $\Pr(\mathcal{A}_j)$ is a normalizing term that is identical for all candidates and is not considered for ranking purposes.

In order to define $\Pr(d_k)$, let p_i denote the prior probability[4] that a component c_i is at fault. The prior probability for a candidate d_k is given by

$$\Pr(d_k) = \prod_{i \in d_k} p_i \cdot \prod_{i \in C \setminus d_k} (1 - p_i). \quad (2)$$

$\Pr(d_k)$ estimates the probability that a candidate, without further evidence, is responsible for erroneous behavior.

$\Pr((\mathcal{A}_j, e_j) \mid d_k)$ is used to bias the prior probability taking observations into account. Let g_i (referred to as component goodness) denote the probability that a component c_i performs nominally

$$\Pr((\mathcal{A}_j, e_j) \mid d_k) = \begin{cases} \prod_{i \in (d_k \cap \mathcal{A}_j)} g_i & \text{if } e_j = 0 \\ 1 - \prod_{i \in (d_k \cap \mathcal{A}_j)} g_i & \text{otherwise} \end{cases}. \quad (3)$$

In cases where values for g_i are not available they can be estimated by maximizing $\Pr((\mathcal{A}, e) \mid d_k)$—i.e., maximum likelihood estimation (MLE) for the Naïve Bayes classifier—under parameters $\{g_i \mid i \in d_k\}$ [19]. This work uses MLE to estimate component goodness.

If we consider our example, the probabilities for both candidates are

$$\Pr(d_1 \mid (\mathcal{A}, e)) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1 \cdot g_2)}_{t_1} \times \underbrace{(1 - g_2)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1}_{t_4}}^{\Pr((\mathcal{A}, e) \mid d)} \quad (4)$$

$$\Pr(d_2 \mid (\mathcal{A}, e)) = \overbrace{\left(\frac{1}{1000} \cdot \frac{1}{1000} \cdot \left(1 - \frac{1}{1000}\right)\right)}^{\Pr(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \times \underbrace{(1 - g_3)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3} \times \underbrace{g_1 \cdot g_3}_{t_4}}^{\Pr((\mathcal{A}, e) \mid d)}. \quad (5)$$

By performing a MLE for both functions it follows that Equation 4 is maximized for $g_1 = 0.47$ and $g_2 = 0.19$. Equation 5 is maximized for $g_1 = 0.41$ and $g_3 = 0.50$. Applying the goodness values to both expressions, it follows that $\Pr(d_1 \mid (\mathcal{A}, e)) = 1.9 \times 10^{-9}$ and $\Pr(d_2 \mid (\mathcal{A}, e)) = 4.0 \times 10^{-10}$. It is customary to normalize fault probabilities over the set of candidates under consideration, producing: $\Pr(d_1 \mid (\mathcal{A}, e)) = 0.83$ and $\Pr(d_2 \mid (\mathcal{A}, e)) = 0.17$, entailing the ranking[5] (d_1, d_2).

3.2 Measuring Quality of Diagnosis

To measure the accuracy of fault-localization approaches, the cost of diagnosis C_d metric is often used [11], [12], [20], [21]. 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.[6] 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. The Wasted Effort metric (or merely Effort) 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.

Another widely used metric is Recall@N [22] (also referred to as Top@N [23] or Hit@N [24]), which computes the percentage of faults among the set of subjects that can detected by exclusively examining the top N (N=1,2,3,...) components of the ranked diagnostic report. Good fault localization techniques should allow developers to find more faults while inspecting less code, thus the higher the Recall@N value, the better the diagnostic performance.

Quality of diagnosis 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 [25], but there are approaches to mitigate it (e.g., [26]).

3.3 Diagnosability Assessment by Measuring Matrix Density

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

$$\rho = \frac{\sum_{i,j} A_{ij}}{N \times M}. \quad (6)$$

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

4. Component prior probabilities depend on the chosen granularity. For instance, if components are statements, one can approximate p_j as $1/1000$, i.e., 1 fault for each 1000 lines of code [18].

5. Also known as *diagnostic report*.

6. Or likelihood score, depending on the fault-localization approach used.

previously demonstrated that the information gain can be modeled as:

$$IG(t_g) = \begin{array}{l} -\Pr(e_g = 1) \cdot \log_2(\Pr(e_g = 1)) \\ -\Pr(e_g = 0) \cdot \log_2(\Pr(e_g = 0)), \end{array} \quad (7)$$

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 [20]. 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.

3.4 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* [27], 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

$$\mathcal{U} = \frac{|G|}{M}. \quad (8)$$

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

4 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.

4.1 Activity Matrix Entropy

To maximize the effectiveness of SR approaches, the ideal activity matrix is one that contains every combination of component activations—as depicted in Fig. 3 —, since it follows that every possible fault candidate in the system is exercised.

	t_1	t_2	t_3	$\cdots$	t_{2^M-1}
c_1	1	0	0	$\cdots$	0
c_2	0	1	0	$\cdots$	0
c_3	0	0	1	$\cdots$	0
c_4	0	0	0	$\cdots$	1
c_5	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_M	1	1	1	$\cdots$	1

Fig. 3. Ideal hit-spectra matrix for a system with M components.

A metric that accurately captures this exhaustiveness is entropy—the measure of uncertainty in a random variable. Shannon Entropy [28] is given by

$$H(X) = -\sum_i P(x_i) \cdot \log_2(P(x_i)) \quad (9)$$

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.[7]

4.2 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.

4.2.1 Density

As discussed in Section 3.3, the ρ metric captures the density of a system. Fig. 4 shows two activity matrices of different densities. A sparse activity matrix, depicted as a diagonal matrix in Fig. 4a, while achieving a high

7. An example of an ambiguity group is the set of statements in a basic block.

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

(a) Sparse activity matrix. $\rho = 0.25$

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

(b) Dense activity matrix. $\rho = 1.0$

Fig. 4. Sparse and dense activity matrices.

	t_1	$\cdots$	t_j	t'_1	$\cdots$	t'_j
c_1						
$\vdots$		$\mathcal{A}'$			$\mathcal{A}'$	
c_i						
c'	0	0	0	1	1	1

Fig. 5. Depiction of the optimal density proof.

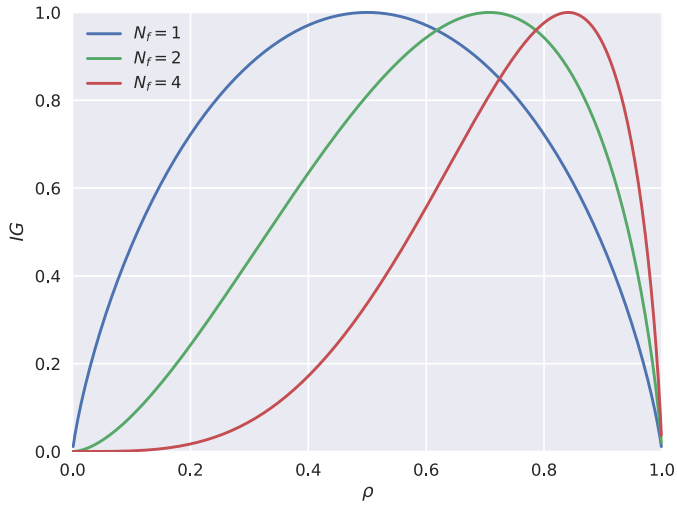

Fig. 6. ρ versus IG for different fault cardinalities.

component coverage due to the fact that every component is executed by the test suite, does not exercise components in tandem, and therefore many potential diagnostic candidates are left unexercised. Conversely, a dense activity matrix as depicted in Fig. 4b is unable to exonerate diagnostic candidates from suspicion as all components are active in all transactions. The ideal density value ($\rho = 0.5$) is in fact in between the two extremes depicted, as the theoretical work of González-Sanchez et al. [8] and the empirical work of Campos et al. [20] show.

It is also straightforward to show the optimality of the value of 0.5 for the density measurement by induction, as depicted in Fig. 5. Suppose that we have an activity matrix $\mathcal{A}'$, 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 optimal sub-matrix $\mathcal{A}'$ both when c' is active and when it is inactive. Therefore, the involvement rate of component c' would be 0.5.

Note that in the case of *dependent faults*—ones where multiple simultaneous components must be involved for the fault to trigger—the optimal value depends on the fault cardinality. Suppose that a system contains N_f dependent faults. The total number of fault candidates can then be expressed by the binomial coefficient $\binom{C}{N_f}$. If the system's coverage matrix density is ρ, tests that exercise it cover, on average, $\rho \cdot C$ components, and thus the number of candidates of cardinality N_f exercised by the test are $\binom{\rho \cdot C}{N_f}$. The probability of a test failing is then

$$\Pr(t_f) = \frac{\binom{\rho \cdot C}{N_f}}{\binom{C}{N_f}}. \tag{10}$$

A binomial coefficient can be expressed using Pochhammer's falling factorial[8]

$$\Pr(t_f) = \frac{\frac{(\rho \cdot C)_{N_f}}{N_f!}}{\frac{(C)_{N_f}}{N_f!}} = \frac{(\rho \cdot C)_{N_f}}{(C)_{N_f}}. \tag{11}$$

8. http://mathworld.wolfram.com/FallingFactorial.html

As the falling factorial $(x)_n$ is equal to $\prod_{i=1}^{n}(x - i + 1)$, Equation 11 can be rewritten as

$$\Pr(t_f) = \prod_{i=1}^{N_f} \frac{\rho \cdot C - i + 1}{C - i + 1}. \tag{12}$$

And since $C \gg N_f$, we can approximate the value of $\Pr(t_f)$

$$\Pr(t_f) \approx \lim_{C \to +\infty} \prod_{i=1}^{N_f} \frac{\rho \cdot C - i + 1}{C - i + 1} = \rho^{N_f}. \tag{13}$$

Then, the information gain from any given test case can be computed as demonstrated in Equation 7 from Section 3.3

$$\begin{aligned} IG &= -\Pr(t_f) \cdot \log_2(\Pr(t_f)) - \Pr(t_p) \cdot \log_2(\Pr(t_p)) \\ &= -\rho^{N_f} \cdot \log_2(\rho^{N_f}) - (1 - \rho^{N_f}) \cdot \log_2((1 - \rho^{N_f})). \end{aligned} \tag{14}$$

The optimal $IG = 1$ value corresponds to $\rho^{N_f} = 0.5$, which means that the optimal density is

$$\rho = \frac{1}{2^{\frac{1}{N_f}}}. \tag{15}$$

Fig. 6 shows the evolution of IG's value over the density for faults of cardinality 1, 2, and 4, where we can see a skew favoring higher densities the more components are involved in a fault. The reason for this behavior is that it is unnecessary to run sparse tests which execute less components than the number of components needed to trigger a failure. In the general case, since one does not know *a priori* about the cardinality of a failure, targeting a $\rho = 0.5$ is still the safest action in terms of covering all possible fault cardinalities. However, if one has a means of deducing the fault cardinality (for instance, using the defect prediction methodology as outlined in [29]), then such information can be exploited—e.g., by turning off sparse tests guaranteed to not trigger the complex fault and reduce the time to run the test suite.

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

$$\rho' = 1 - |1 - 2 \cdot \rho| \tag{16}$$

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

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

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

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

Fig. 7. Impact of diversity on ρ' and $\mathcal{G}$.

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

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

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

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

Fig. 8. Impact of component ambiguity on ρ', $\mathcal{G}$ and $\mathcal{U}$.

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 6). This means that a matrix with no diversity (depicted in the example from Fig. 7a) is able to reach the ideal value for the ρ' metric.

4.2.2 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. Ensuring diversity is also prone to minimize the impact of *coincidental correctness*—when a fault is executed but no failure is detected—as shown in the work by Masri and Assi, which remove passing tests which exhibit the same coverage pattern as failing tests, resulting in improved diagnostic accuracy [30]. 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 ($\mathcal{G}$) [31]. The $\mathcal{G}$ metric computes the probability of two elements selected at random being of different kinds:

$$\mathcal{G} = 1 - \frac{\sum n \times (n-1)}{N \times (N-1)}, \tag{17}$$

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. Fig. 7a and 7b 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.

4.2.3 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 Fig. 8a, then they form an *ambiguity group* (see Section 3.4), 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.[9]

We use a check for uniqueness ($\mathcal{U}$) as described in Equation 8 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.

4.2.4 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:

$$\text{DDU} = \rho' \times \mathcal{G} \times \mathcal{U} \tag{18}$$

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.

5 Theoretical Evaluation

A simulation approach to spectra generation enables us to consider an otherwise infeasible breadth of scenarios, so

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

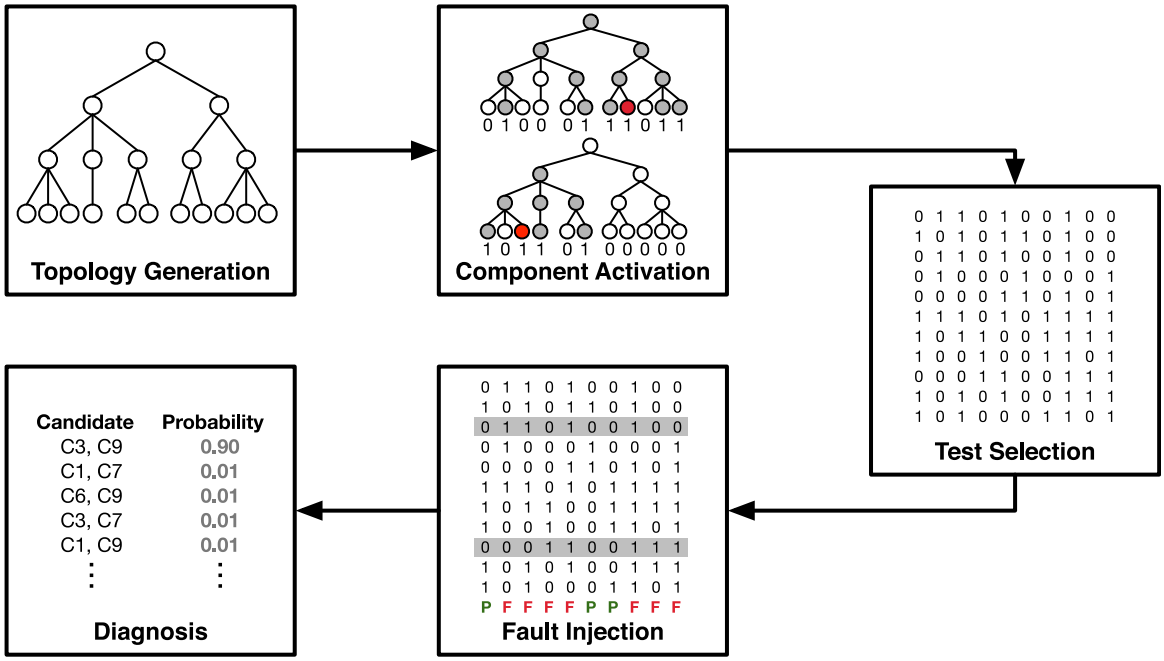

Fig. 9. Process followed by the spectra simulator.

that the metric's diagnosability performance can be analyzed from a holistic, theoretical standpoint—akin to related work on spectrum based fault localization [8], [12], [32]. Therefore, we first evaluate the DDU metric by generating a multitude of program spectra via simulation to further confirm the claims we make while devising the DDU metric in the previous section. This section (1) describes the topology-based spectra simulator and fault injector we created for this theoretical analysis; (2) details the experimental setup, where thousands of qualitatively distinct spectra were automatically generated by the simulator; and (3) presents an assessment on the correlation of DDU—and coverage—with diagnostic effort, as well as an assessment on the influence of a system's topology on its diagnosability, based on the simulated data. Afterwards, in Section 6, we empirically evaluate the DDU metric.

5.1 Spectra Simulator

The spectra simulator we built for this theoretical assessment is able to generate a breadth of qualitatively distinct coverage matrices. It uses *topology*-based[10] policies to select which components are active on each test, and relies on component goodnesses—as described in Section 3.1.2—to inject test failures. Fig. 9 depicts the overall process followed by the simulator to generate a set of faulty program spectra and their respective diagnoses. The following subsections detail each step of the simulation process.

5.1.1 Topology Generation

The first step in the simulation process is to generate a random tree with as many leaves as components to be simulated. Tree generation follows a uniform *birth-death* process, commonly used to simulate *phylogenetic trees* [33], in which *lineages* (or tree paths) have a constant probability of *speciating* (splitting into multiple branches), and a constant probability of going *extinct*, per time unit. The generated tree acts as the system *topology*, and is predicated on the fact that, in most programming paradigms, source code is structured in a hierarchical fashion—especially in the case of object-oriented languages. Specifically, tree leaves correspond to the components in the spectrum abstraction—the units of computation used to diagnose the system, which can be branches, statements, *etc*—, and inner nodes correspond to hierarchical source code artifacts of coarser granularity such as methods, classes and subclasses, and package folders. We note that, much like system topologies, our generated trees are not necessarily balanced.

5.1.2 Component Activation

After generating a topology, the component activation step generates a vast amount of test cases by activating components and propagating these activations through the topology. This step starts with the selection of a component (which we call the *anchor*) and setting it as active in a newly created test-case. Anchor components are shown as red tree nodes in the Component Activation step depicted in Fig. 9. With the selection of an anchor, we randomly activate other components based on their distance to the anchor—following the assumption that the farther away two components are, the less related they are and hence less likely to be covered in the current test-case being generated. To confirm our assumption, we have constructed a topology tree for each subject in the Defects4J catalog (further described in Section 6.1) and measured the frequency with which distances[11] between any two covered components appear in test cases. Fig. 10 depicts these findings, which indicate that, indeed, the further two components are from each other,

10. The use of topologies to generate spectra is inspired by SERG-Delft's simulator: https://github.com/SERG-Delft/sfl-simulator

11. i.e., the minimal number of edges one needs to traverse to go from a given node in the tree to another given node.

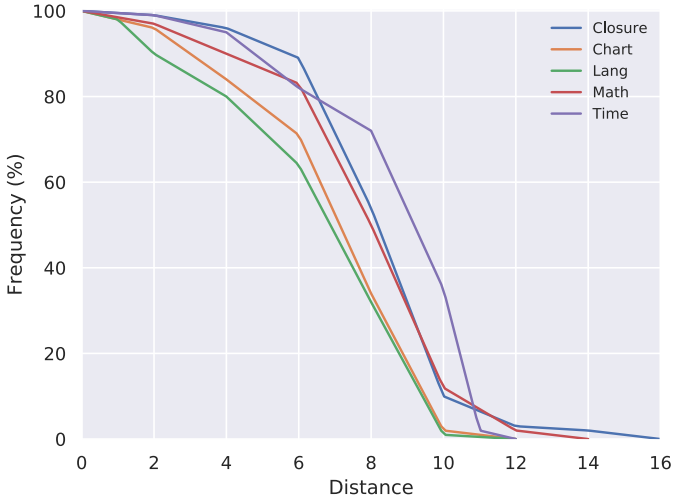

Fig. 10. Frequency of component distances in test cases for each Defects4J subject.

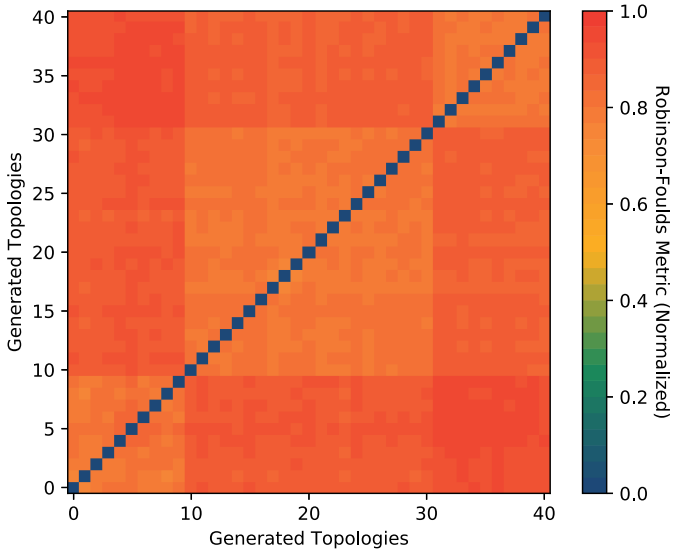

Fig. 11. Robinson-Foulds metric values for every generated topology.

the less frequently both of them are covered in the same execution.

It is worth noting that coverage density of a test can be manipulated by multiplying the activation probability by a *density term*. If this term is < 1, then sparse test cases are generated. Conversely, a value > 1 yields denser test cases. We generate test cases using a wide spectrum of *density terms*. This component activation process is repeated numerous times for each component in the system, so that a large collection of test cases is available to the next steps in the simulation.

5.1.3 Test Selection

This step consists of selecting a set of test cases out of the test case pool generated in the previous step. We have chosen to select as many tests as there are components in the system—yielding square coverage matrices. Having the test suite depend on the number of components allows it to grow with program size, with the assumption that the larger the code base is, the more tests are created.

5.1.4 Fault Injection

For each matrix that the previous step produces, we inject it with: (1) a single fault, (2) multiple independent faults, and (3) multiple dependent faults. In the first case, we randomly assign a component from the system as the faulty one, and set each test which covers the faulty component as having a failing outcome. In scenario (2), multiple components are considered as being faulty, and thus tests that cover any non-empty subset of faulty components are set to failing. In the last scenario, only tests that cover the conjunction of all failing components are set to failing. We include multiple-faulted scenarios in our analysis since, as studied in previous work [34], such scenarios account for a non-trivial portion (20 percent) of bug-fixing tasks in open-source projects.

The fault injection step is also able to consider component goodnesses, which, as described in Section 3.1.2, describe the probability of a faulty component exhibiting nominal behavior (and thus not triggering a test failure). For instance, in a single-faulted scenario modeled with 0.25 goodness, a test case that covers the fault has a 75 percent chance to be labeled as failing. Such a component goodness modeling therefore allows us to replicate *coincidentally correct* scenarios.

5.1.5 Diagnosis

We diagnose the faulty spectra generated in the previous step using the reasoning-based fault-localization technique described in Section 3.1.

5.2 Setup

We have run our simulation 40 times so that 40 distinct topologies ranging from 100 to 500 components were considered. To assess if we generate distinct topologies, we measure the Robinson-Foulds distance metric [35], [36] for every pair of generated topology topologies. Robinson-Foulds measures the minimal number of operations (such as adding or removing nodes and edges) that are required to transform a given tree A into tree B. The metric's lower bound is zero and it corresponds to the case when the two trees under consideration are identical. The upper bound is equal to the sum of all edges among both trees, and it means that the entirety of tree A has to be reconstructed to obtain tree B, thus the two trees do not share any similar structure. Since the upper bound depends on the sizes of the two trees under consideration, the metric's value can be normalized (dividing by the upper bound value) such that it ranges between 0 and 1. Fig. 11 shows the normalized Robinson-Foulds metric values for every pair of topology trees we generate. Results show that any tree exhibits high Robinson-Foulds values when compared to all other generated trees, which leads us to conclude that all our generated topologies are different and qualitatively distinct from each other.

For each topology, all components acted as anchors, generating a test-case pool using several *density terms*. Each test-case pool produced 100 matrices, which were fault-injected—with a single fault, two/three independent faults, and two/three dependent faults. We have used the following goodness values for our simulation: 0.0, 0.25, 0.50, and 0.75. Regarding metrics, we have gathered coverage, DDU,

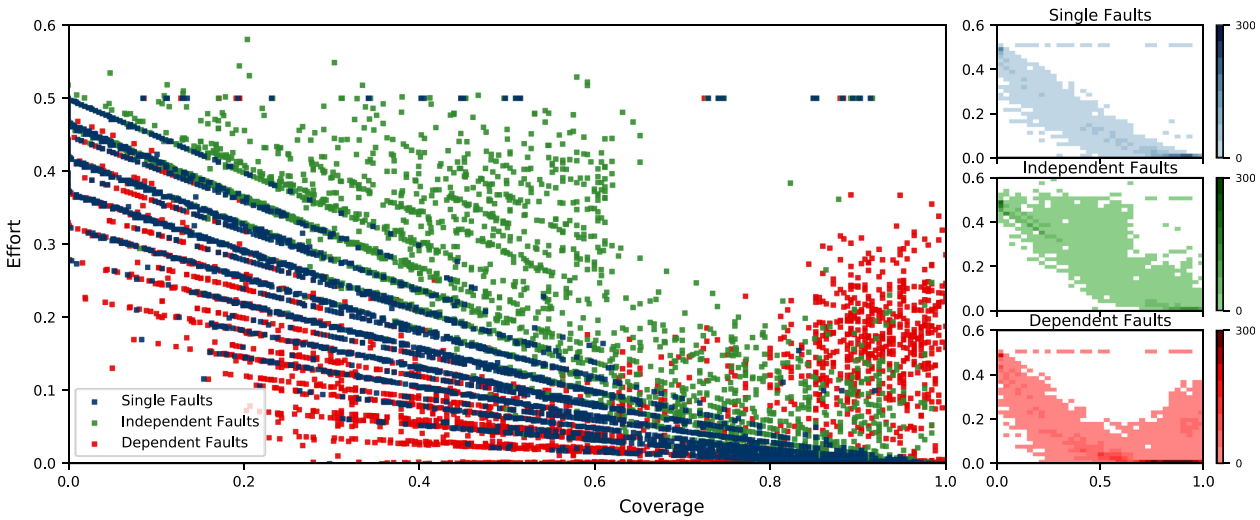

Fig. 12. Relation between diagnostic effort and coverage.

entropy, and effort to diagnose for every faulty spectra generated by the simulator.

To ensure reproducibility, our spectra simulator, and its respective configuration file describing this experiment, are made available.[12] In total, more than half a million spectra were simulated, fault injected, and diagnosed in this experiment.

5.3 Results

Below we present and discuss the spectra simulation results by (1) evaluating the diagnostic quality using the effort metric as described in Section 3.2, and by (2) evaluating the simulated spectra's propensity for error detection.

5.3.1 Diagnostic Quality

Diagnostic effort results for every spectrum generated in this experiment are shown in Figs. 12, 13, 14, 15. Each figure shows a scatter plot portraying the relation of diagnostic effort[13] with different metrics—namely coverage, DDU, entropy, and the average of density, diversity and uniqueness. Points in the scatter plot represent simulated spectra. Beside each scatter plot are three two-dimensional histograms depicting the distribution of spectra containing each fault type described in Section 5.1.4.

Fig. 12 portrays the relation between coverage and diagnostic effort for all simulated spectra. Regarding spectra that were injected with a single fault, their diagnosability improves by increasing coverage. Note that single-faulted spectra seem to form several downward lines in the scatter plot—each of these lines corresponds to a different topology used as the basis for emulating software structure. We can therefore make two observations. The first is that, for a given topology, the selection and composition of the test suite influences not only coverage but also the effort to diagnose. The second is that the choice of base topology also influences diagnostic quality.

While single-fault diagnostic effort mostly decreases with coverage, the same cannot be said for scenarios with multiple faults, especially ones where dependent faults were injected, since several spectrum instances with high coverage are not in the bottom-right of the plot. For these scenarios, high coverage is not a good indicator of diagnosability. An illustrative example of such phenomenon is as follows. Consider a spectrum that resembles a diagonal matrix, where each test exercises a single distinct component. Such a spectrum has high coverage—because every component is exercised—and, at the same, is *sparse*—since all tests contain a single component activation. In effect, this is analogous to a high-coverage *unit* test suite with no integration tests exercising multiple components. For single fault scenarios, this suite is very likely to find and accurately isolate faults. However, in cases where a fault requires set of component activations for an error to be triggered, this suite cannot provide enough evidence for fault localization algorithms to pinpoint faults.

Fig. 13 depicts the relation between DDU and effort. We can tell that this metric upper bounds the effort to diagnose—the higher the DDU, the lower the maximal diagnostic effort—providing a more accurate expectation of diagnosability when compared with coverage. As opposed to coverage, multiple faults do not negatively influence the DDU's diagnostic accuracy.

Fig. 14 shows test entropy—as described in Section 4.1—in the x-axis. Note that entropy values range from 0 to 1, but to improve legibility, we are showing a partial range of entropy values up to 0.08 as no spectra in our simulation exceeded this value. In effect, the number of test cases generated by the simulator (set to be the same as the number of components in every generated spectra) is insufficient to significantly explore the entire range of entropy values. Limiting the number of tests was an intentional way to model how developers test in practice, and therefore it leads us to conclude that optimizing for entropy is infeasible with a reasonable number of tests.

We discuss in Section 4.2.4 the reasons for choosing multiplication as a way of reducing the composing terms of DDU (namely density, diversity and uniqueness) into a single value that represents the system's diagnosability. While we explain why each term is important for the overall diagnosability, it might be the case that there is a better way to

12. Available at https://github.com/aperez/sfl-simulator.

13. Normalized over the number of components, so that spectra of systems with a different number of components can effectively be compared.

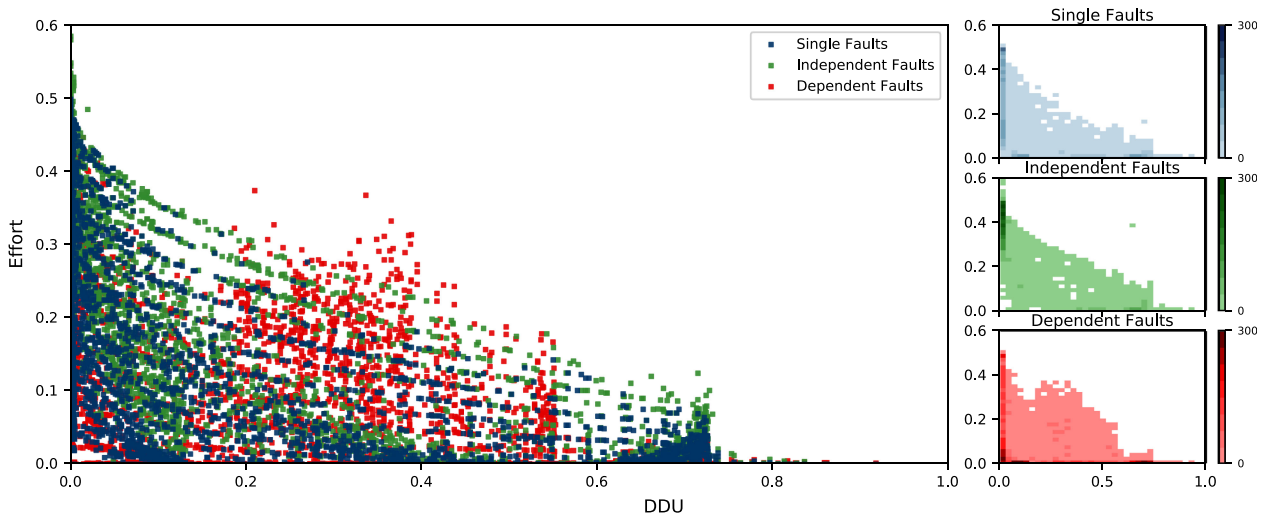

Fig. 13. Relation between diagnostic effort and DDU.

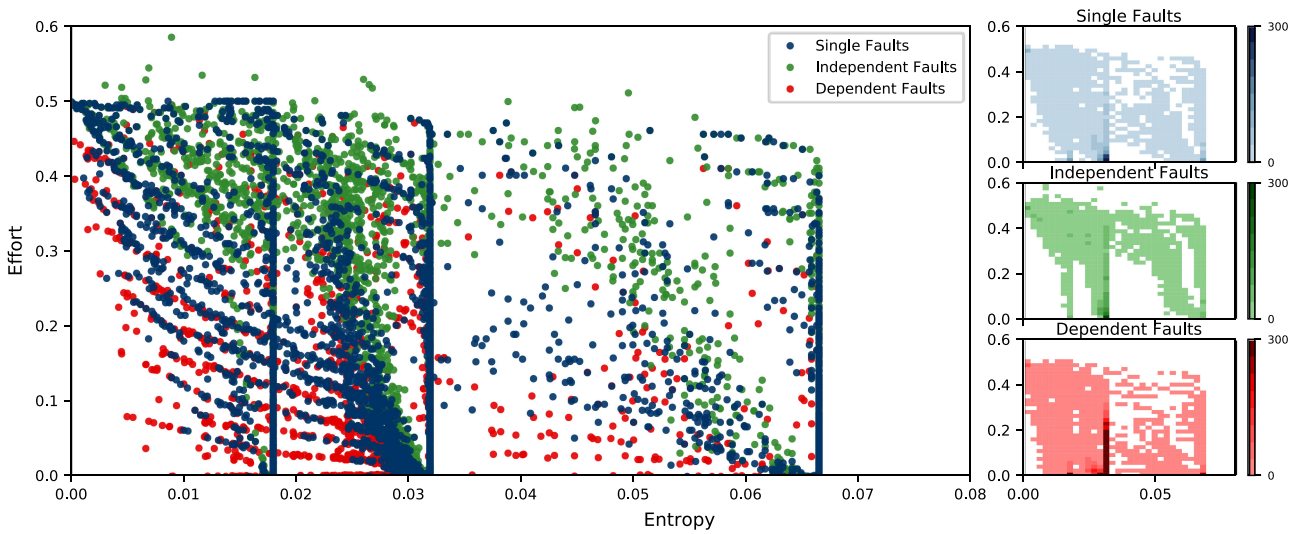

Fig. 14. Relation between diagnostic effort and entropy.

reduce them into a one-dimensional value. Fig. 15 depicts using the average of density, diversity and uniqueness values as the measure for diagnosability—as opposed to their multiplication, which is depicted in Fig. 13. We can conclude that the multiplication of density, diversity and uniqueness more accurately predicts the diagnostic performance of the test suite.

5.3.2 Error Detection

Besides investigating diagnostic quality, which relates to the actual effort bugs take to be located, we have also recorded the *error detection rate*. This evaluates the propensity for faults in a given coverage matrix to induce test errors, and is achieved by keeping track of the frequency in which errors are detected in faulty spectra. As readers may recall from Section 5.1.4, each coverage matrix we generate is subject to multiple rounds of fault injection. As a result, we generate a sets of spectra that exhibit the same coverage matrix and different error vectors. Error detection rate is then the frequency by which these sets of spectra exhibit failing error vectors.

Fig. 16a and 16b show two-dimensional histograms depicting the error detection frequency of coverage matrices along coverage values and DDU values, respectively. Fig. 16a tells us that the majority of high coverage spectra are able to produce test failures when faults are injected, as portrayed by the intensity of the top-right portion of the histogram. However, we still observe a significant portion of cases with low error detection despite their coverage value, as evidenced by the intensity of the bottom row in the histogram. Such spectra do not have adequate test cases that *detect* the injected faults. In contrast, we see that when DDU is considered—Fig. 16b —, there are considerably less cases of high-DDU spectra yielding low error detection rates. This is initial evidence that DDU may be suited for measuring the *adequacy* of test suites, besides simply measuring diagnosability.

6 Empirical Evaluation

Results obtained by simulating a breadth of program spectra seem to indicate that, from a theoretical standpoint, DDU effectively estimates the diagnostic effort required to pinpoint bugs, regardless of fault type. However, these promising results do not exclude the need to evaluate the metric against real-world subjects. This section details our

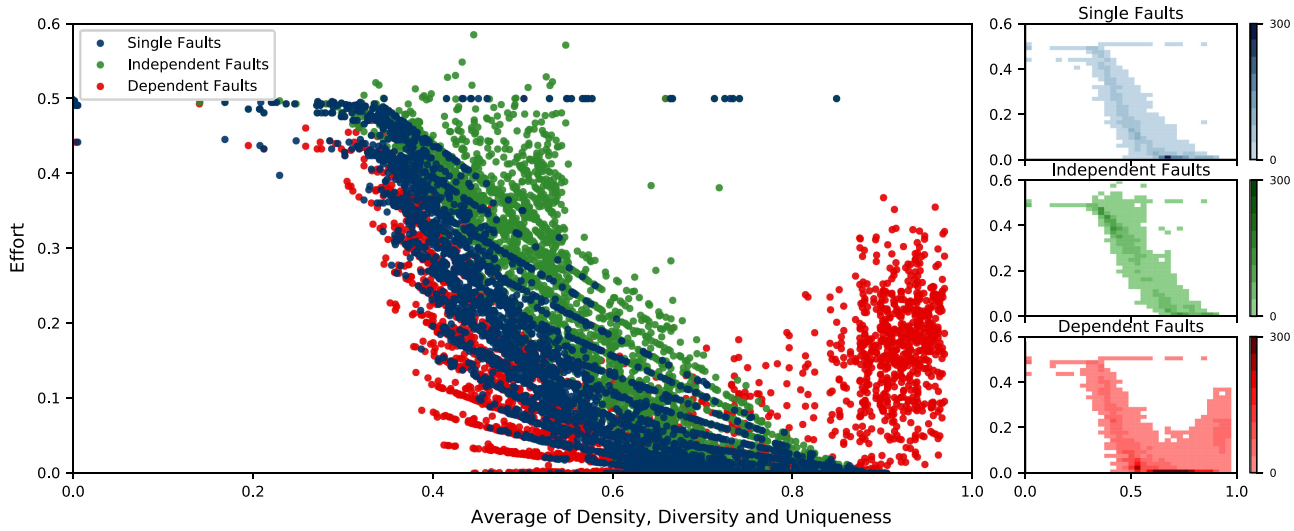

Fig. 15. Relation between diagnostic effort and the average of density, diversity and uniqueness.

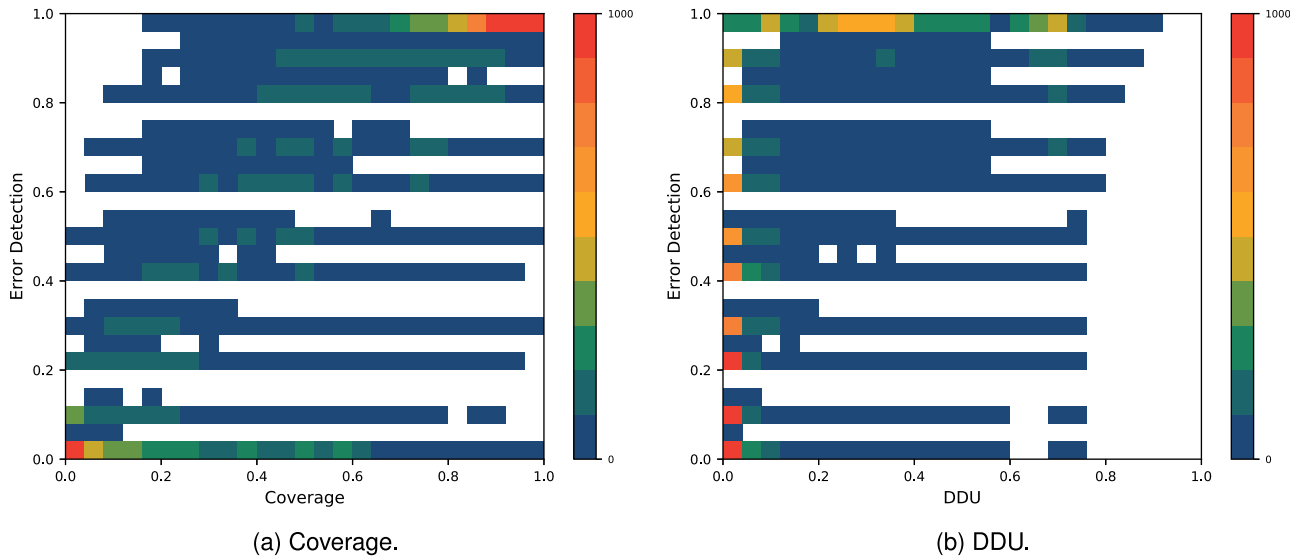

Fig. 16. Two-dimensional histograms depicting the number of simulated matrices along with the relation between error detection and several metrics.

following experiment, in which we empirically evaluate the proposed metric in regard to its ability to assess diagnostic quality. We aim to address the following research 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?

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.

6.1 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 [14] is a tool that employs Search-based Software Testing (SST) approaches to create new test cases. It applies Genetic Algorithm (GAs) to minimize a *fitness function* which describes the distance to an optimal solution. The metrics to be compared are DDU—our proposed measurement;

14. EvoSuite tool is available at http://www.evosuite.org. Version 1.0.2 was used for experiments (accessed January 2019).

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

$$f_{\mathcal{M}}(TS) = |\mathcal{O}_{\mathcal{M}} - \mathcal{M}(TS)|, \tag{19}$$

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 [20].

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[15]). 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 6.2 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 percent 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], [19].

For assessing the applicability in real world scenarios and to answer *RQ3*, we need real life bugs and fixes. Therefore, in Section 6.3 we make use of Defects4J [16]—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.[17] This tool implements the approach described in Section 3.1, 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

$$\Delta_{\text{Effort}}(C) = \text{Effort}_C - \text{Effort}_{\text{DDU}}, \tag{20}$$

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 3.2. 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 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 consider that the use of the normalized effort to create the paired $\Delta_{\text{Effort}}(C)$ provides an adequate means of comparing diagnostic quality that captures the magnitude of effort differences over distinct subjects. Conversely, such magnitude could be incorrectly measured using other quality of diagnosis metrics described in Section 3.2, such as Recall@N—due to the N threshold —, or C_d—due to different program sizes among subjects.

We make use of kernel density estimation plots to show the $\Delta_{\text{Effort}}(C)$ values in Figs. 17 and 18. 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). Thus, these plots help visualize the distribution of data over a continuous interval and can be considered as smoothed, continuous histograms. 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). Also, the dashed vertical line at $\Delta_{\text{Effort}}(C) = 0$ is present to aid the interpretation of results. $\Delta_{\text{Effort}}(C) = 0$ is an important landmark to take into consideration because for positive values of $\Delta_{\text{Effort}}(C)$ it means that the test generation using the DDU yielded better diagnostic reports that the C criterion. Vice versa for negative values of $\Delta_{\text{Effort}}(C)$.

It is worth noting that the setup of our empirical evaluation differs from that of the theoretical evaluation. In the theoretical evaluation we simulate a multitude of qualitatively distinct spectra ranging the entire range of DDU and coverage values to observe how changing these variables impacts diagnosability. Repeating that evaluation on an empirical setting would mean devising a (reasonably small) windowed stopping criterion so that the test generation

15. 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.

16. Defects4J tool is available at https://github.com/rjust/defects4j. Version 1.0.1 was used for experiments (accessed January 2019).

17. Crowbar tool is available at https://github.com/TQRG/crowbar-maven-plugin (accessed January 2019).

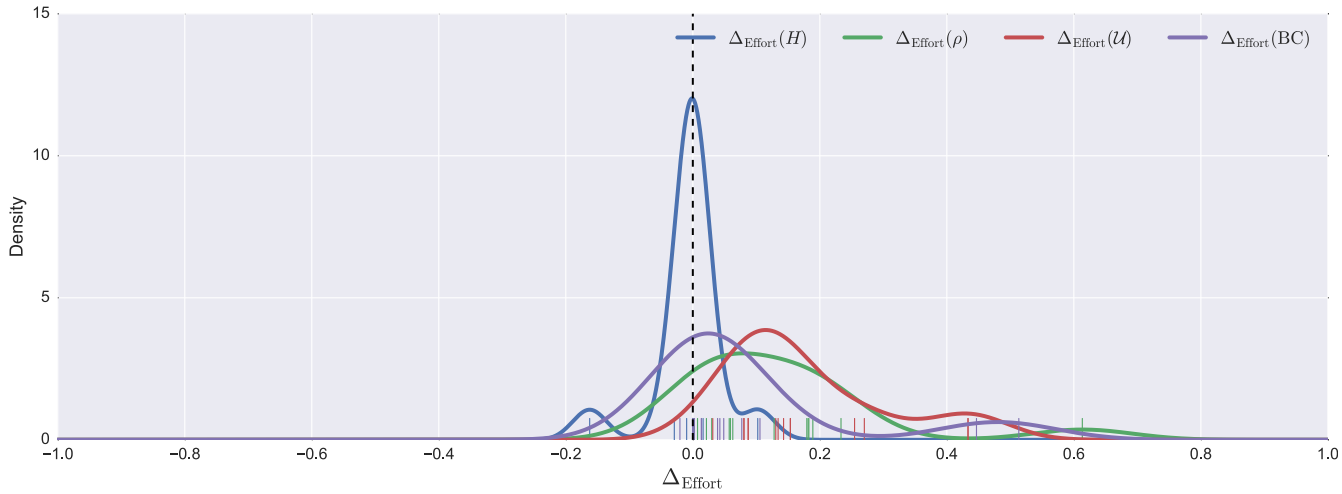

Fig. 17. Kernel density estimation of seeded fault experiment. Entropy generation criterion shows similar diagnostic accuracy when compared DDU. The remaining generation criteria exhibit worse diagnostic performance than DDU.

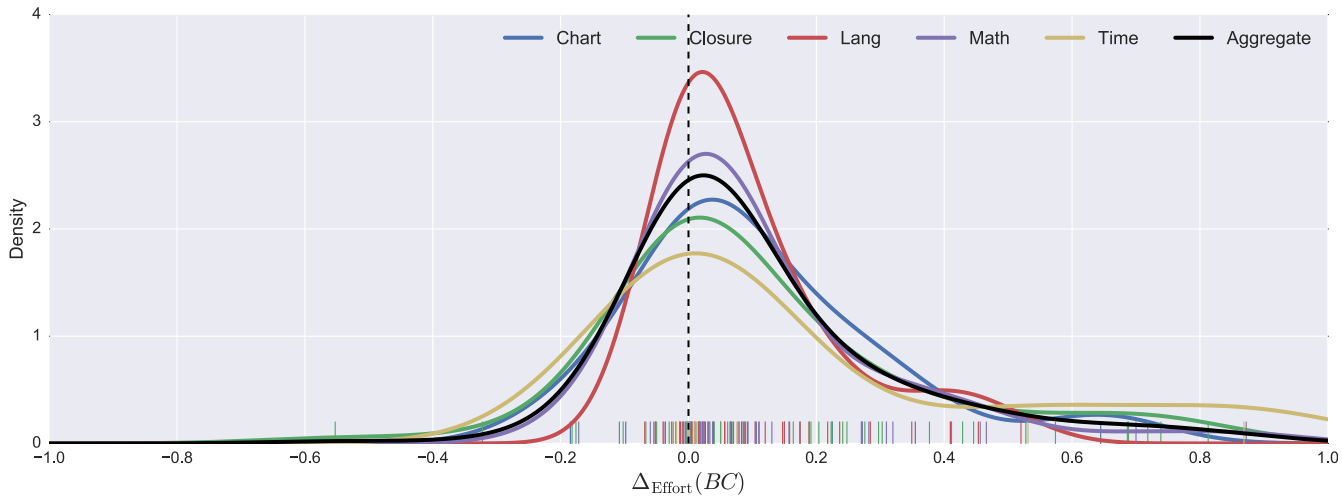

Fig. 18. Kernel density estimation of the $\Delta_{\text{Effort}}(\text{BC})$ metric for DEFECTS4J subjects. 77 percent of instances have a positive $\Delta_{\text{Effort}}(\text{BC})$, meaning that branch-coverage generations perform worse than DDU generations.

process could be ran across the entire metric range. In practice, such an approach is not guaranteed to ever terminate, because of, e.g., local maxima near the stopping window. Instead, in this empirical experiment, we are generating test cases to maximize a given metric and comparing test suites generated by distinct test generation strategies.

6.2 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 use the same set of subjects as empirical evaluations from related work [20]. Namely, we use the open-source projects Apache Commons-Codec, Apache Commons-Compress, Apache Commons-Math and Joda-Time. 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 Fig. 17. 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 percent 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 percent of scenarios.

We show in Table 1 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 percent 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

TABLE 1
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^{-2s}	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}

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 percent 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 1 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,[18] for manual test generation entropy will yield very small values for any complex system, as one can see from Table 1. 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.

18. Because tools like EvoSuite can be configured with a time budget as another stopping criteria.

TABLE 2
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

6.3 Diagnosing Real Faults

We used the Defects4J database [37] 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 2 are fit for analysis and their results are used throughout this section.

TABLE 3
Measurements and Statistical Tests Comparing Coverage and DDU Test Generations

	Branch-Coverage Generation	DDU Generation
Branch Coverage	0.85	0.75
DDU	0.10	0.42
Suite Size	291	374
Recall@1	6.3%	26.3%
Recall@10	23.7%	46.2%
Recall@25	35.6%	58.3%
Effort	0.31	0.10
Shapiro-Wilk	$W = 0.92$ $p\text{-value} = 1.70\times10^{-8}$	$W = 0.85$ $p\text{-value} = 1.05\times10^{-12}$
Wilcoxon Signed-rank	$Z = 2335.0$ $p\text{-value} = 3.50\times10^{-13}$	

TABLE 4
Measurements and Statistical Tests Comparing Strong Mutation and DDU Test Generations

	Strong Mutation Generation	DDU Generation
Branch Coverage	0.68	0.75
DDU	0.08	0.42
Suite Size	71	374
Recall@1	18.0%	26.3%
Recall@10	32.6%	46.2%
Recall@25	50.0%	58.3%
Effort	0.26	0.10
Shapiro-Wilk	$W = 0.93$ $p\text{-value} = 5.82\times10^{-7}$	$W = 0.85$ $p\text{-value} = 1.05\times10^{-12}$
Wilcoxon Signed-rank	$Z = 3227.5$ $p\text{-value} = 4.03\times10^{-3}$	

Experimental results are shown in Fig. 18. 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 percent of scenarios (144 in total) yield a positive $\Delta_{\text{Effort}}(\text{BC})$.

We performed several measurements and statistical tests to assess whether the gathered metrics yielded statistically significant results. Table 3 shows the relevant statistics. The first three 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. The following three rows display Recall@N figures for N=1, N=10 and N=25, respectively, which show that the DDU criterion is more effective at prioritizing the inspection of the real fault even for small (and practical) values for N. Results in the effort row corroborate our observations from Fig. 18—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 percent on average. However, this fact alone does not guarantee that the results are 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 3, tell us that the distributions are not normal, with confidence of 99 percent.

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 3 shows the resulting Z statistic and p-value. With 99 percent confidence, we can refute the null-hypothesis.

We have also repeated this experiment using EVOSUITE 's strong mutation criterion for test generation. For the same set of DEFECTS4J subjects in Table 2, we generate, using each subject's *fixed version*, a test suite that maximizes strong mutation score. This EVOSUITE criterion creates a set of program mutations by applying mutation operators such as statement deletion, negation of conditions, unary operator insertion, operator replacement, variable replacement, among others. Subsequently, it generates test cases that not only cover the mutants, but also that yield different outcomes between the original program instance and its mutated counterpart. We have run the generated test suites against the faulty DEFECTS4J subjects, and obtained the results depicted in Table 4. Our results show a median effort of 0.26 using the strong mutation, and an effort reduction of 17 percent when considering DDU over strong mutation. Results also show that the mutation criterion yields better Recall@N performance compared to coverage, but they also show that this criterion is not as effective as the DDU criterion at prioritizing the real fault. Furthermore, we show that the effort distributions are not normal (with a confidence of 99 percent) by performing the Shapiro-Wilk test. Again, by performing the Wilcoxon signed-rank test, we can, with 99 percent confidence, refute the null-hypothesis which stated that the median difference between the to observations is zero.

Revisiting *RQ3*:

> *RQ3*: Does optimizing a test-suite with regard to DDU result in better diagnosability than optimizing adequacy metrics?

A: Since the median effort in the DDU generation is lower we can say that optimizing for DDU produces better, statistically significant, diagnoses when compared to test suites that optimize for branch-coverage or mutation score.

6.4 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 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 [38]) 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 3.2 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.

7 DISCUSSION

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.

7.1 Composition of a Test Suite

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.[19] Is this split indeed ideal in terms of diagnostic accuracy? We believe a DDU analysis can provide guidance as to what the answer is, as evidenced in the theoretical evaluation. Our simulation of spectra shows that changing the composition of a test suite through test selection does impact the diagnostic effectiveness for a given base topology, and as such, an optimal selection can be achieved through minimization of the DDU metric.

7.2 Test Design Strategy

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 [39].

Additionally, an *ensemble* of strategies that individually improve DDU's density, diversity, and uniqueness terms could also be considered. Density-based test strategies would focus on selecting the optimal test scope. Diversity-based strategies would focus on identifying and exercising untested code paths. Uniqueness-based strategies would focus on decoupling component executions. Tying into genetic-algorithm-based automated test generation tools such as the one used in our evaluation, these three strategies could serve as the cornerstone for a multi-objective approach to test generation, that maximizes the ability of a test suite to further isolate faults, similar to *test suite amplification* strategies [9], [40].

At a broader scope, our simulation experiment also tells us that system structure or architecture—which we call topology—also has an influence on diagnosability. Test design strategies will necessarily need to utilize such structural information to provide better assessments as to what tests should be performed to improve diagnostic quality. Conversely, it is also not unreasonable to expect that a change in the structure could yield considerable gains in diagnosability.

7.3 Visualization

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? In our opinion, the challenge for creating such visualization would be conveying the three diferent properties that the DDU metric captures in such a way that would elucidate the user regarding what his/hers best next action is in order to increase the system's diagnosability. We envision that visualization approaches for program comprehension, such as EXTRAVIS [41] and PANGOLIN [42], will constitute a solid starting point for a study on visual, interactive and actionable ways to improve the system.

7.4 Generalization to Other Debugging Techniques

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. For example, *program repair* approaches that require a diagnostic report to guide the program synthesis process, such as SemFix [43], will benefit from the fact that there is less wasted effort in highly diagnosable test suites.

We plan to investigate the hypothesis that DDU applies to other runtime-based diagnostic techniques as future work.

7.5 Adequacy Assessment

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 evaluating the fault finding effectiveness. Thus, gathering more empirical data on the development of real systems and expanding previous assessments on the usefulness of the DDU metric [10], [44] is a particularly interesting 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. Namely, we argue that developers, when writing new test cases that either exercise uncovered branches or live mutants within the code, should do so with the understanding that test suite diagnosability is of critical importance to ensure an effortless debugging experience in the event that a failure is detected in the system, and should therefore follow a test design strategy that also takes into account the DDU measurement.

8 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.

19. Google Testing blog: Just Say No to More End-to-End Tests. http://goo.gl/S5HhZ7 (accessed January 2019).

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 [45]. 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 [46] 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 [47]. This approach, since it separates tests into small fractions, has the potential to optimize component uniqueness and therefore can be a crucial tool to minimize ambiguity grouping. As new (qualitatively distinct) tests are added, this approach also has the potential to improve test diversity. However, since it will create several test cases which cover small portions of code, resulting in a decrease in overall density, it potentially overlooks the case where a specific combination of components need to be involved in a test for a failure to occur, much like in the second example of our motivation. Nevertheless, we believe that a combination of *test-case purification* and *test-case amplification* approaches, that break tests into smaller fragments, and then generate combinations of such fragments can be a way to extend the approach proposed by Xuan et al., in such a way that improves DDU.

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., [48], [49]) 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 [20], 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 6 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 [50].

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 [51], [52]. *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* by analyzing data and control-flow patterns [53]. Just et al. investigated the use of mutants to estimate oracle quality, and compared their performance against the use of real faults [54]. 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.

9 Conclusion

This paper highlights the importance of diagnosability—the ability to effectively locate potential faults in the code—as a criterion for assessing the quality of a test suite, and proposes DDU as a measurement of program spectra 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. As opposed to adequacy measurements such as coverage which mainly tackle the issue of error detection, a diagnosability measurement like DDU analyses how combinations of components are exercised in tandem in order to maximize the usefulness of fault localization techniques at pinpointing the *cause* of any error that may occur.

Our topology-based simulation of program spectra was able to reveal that DDU effectively establishes an upper-bound on the maximal effort required to diagnose faults, regardless of fault type or cardinality. We also performed an empirical evaluation 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 percent when locating faults by optimizing DDU compared to branch-coverage.

Besides paving the way for a more comprehensive use of test-suites, we also consider this study on the diagnosability of software to have broader implications. Namely, that the relative amount of system tests, unit tests and end-to-end tests that compose a test-suite is critical for its diagnostic effectiveness and that the structure of systems directly influences their diagnosability.

Acknowledgments

We thank Maurício Aniche, Lara Crawford and Alexey Zagalsky for the useful feedback on previous versions of this paper. The work reported in this article was partially supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2019, the FaultLocker Project (ref. PTDC/CCI-COM/29300/2017), by the FCT scholarship number SFRH/BD/95339/2013, 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," *Commun. 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," *Softw. Eng. J.*, 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 Comput. 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 *Proc. SIGPLAN/SIGSOFT Workshop Program Anal. Softw. Tools Eng.*, 1998, pp. 83–90.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 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," *J. Syst. Softw.*, 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 *Proc. 4th IEEE Int. Conf. Softw. Testing Verification Validation Workshop*, 2011, pp. 634–643.

[9] B. Baudry, F. Fleurey, and Y. L. Traon, "Improving test suites for efficient fault localization," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 82–91.

[10] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 654–664.

[11] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey of software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[12] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *Proc. 24th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 88–99.

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

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

[15] R. Abreu and A. J. C. van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," in *Proc. 8th Symp. Abstraction Reformulation Approximation*, 2009, pp. 2–9.

[16] A. Feldman, G. M. Provan, and A. J. C. van Gemund, "Computing minimal diagnoses by greedy stochastic search," in *Proc. 23rd AAAI Conf. Artif. Intell.*, 2008, pp. 911–918.

[17] N. Cardoso and R. Abreu, "MHS^2: A map-reduce heuristic-driven minimal hitting set search algorithm," in *Proc. Int. Conf. Multicore Softw. Eng. Perform. Tools*, 2013, pp. 25–36.

[18] J. Carey, N. Gross, M. Stepanek, and O. Port, "Software hell," in *Proc. Bus. Week*, 1999, pp. 391–411.

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "A new bayesian approach to multiple intermittent fault diagnosis," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 653–658.

[20] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 257–267.

[21] 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 *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 314–324.

[22] R. Wu, H. Zhang, S. Cheung, and S. Kim, "Crashlocator: locating crashing faults based on crash stacks," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 204–214.

[23] M. Wen, R. Wu, and S. Cheung, "Locus: locating bugs from software changes," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 262–273.

[24] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proc. ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 127–138.

[25] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.

[26] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," in *Proc. 1st IEEE Working Conf. Softw. Vis.*, 2013, pp. 1–10.

[27] A. González-Sanchez, R. Abreu, H. Gross, and A. J. C. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proc. 26th IEEE/ACM Int Conf. Automated Softw. Eng.*, 2011, pp. 83–92.

[28] C. E. Shannon, "A mathematical theory of communication," *Mobile Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, 2001.

[29] R. Abreu, A. González-Sanchez, and A. J. C. van Gemund, "A diagnostic reasoning approach to defect prediction," in *Proc. 24th Int. Conf. Ind. Eng. Other Appl. Appl. Intell. Syst. Part II*, 2011, pp. 416–425.

[30] W. Masri and R. A. Assi, "Cleansing test suites from coincidental correctness to enhance fault-localization," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation*, 2010, pp. 165–174.

[31] 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

[32] N. Cardoso and R. Abreu, "A kernel density estimate-based approach to component goodness modeling," in *Proc. 27th AAAI Conf. Artif. Intell.*, 2013, pp. 152–158.

[33] K. Hartmann, D. Wong, and T. Stadler, "Sampling trees from evolutionary models," *Systematic Biol.*, vol. 59, no. 4, pp. 465–476, 2010.

[34] A. Perez, R. Abreu, and M. d'Amorim, "Prevalence of single-fault fixes and its impact on fault localization," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation*, 2017, pp. 12–22.

[35] D. Robinson and L. Foulds, "Comparison of phylogenetic trees," *Math. Biosciences*, vol. 53, no. 1, pp. 131–147, 1981.

[36] N. D. Pattengale, E. J. Gottlieb, and B. M. E. Moret, "Efficiently computing the Robinson-Foulds metric," *J. Comput. Biol.*, vol. 14, no. 6, pp. 724–735, 2007.

[37] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[38] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Proc. Companion 22nd Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2007, pp. 815–816.

[39] R. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Reading, MA, USA: Addison-Wesley, 2000.

[40] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 309–319.

[41] 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, May/Jun. 2011.

[42] A. Perez and R. Abreu, "Framing program comprehension as fault localization," *J. Softw.: Evolution Process*, vol. 28, no. 10, pp. 840–862, 2016.

[43] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 772–781.

[44] A. Ang, "Exploring DDU in Practice," Department of Software Technology Master's thesis, Delft Univ. Technol., Delft, The Netherlands, 2018.

[45] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 201–210.

[46] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 30–39.

[47] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 52–63.

[48] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *J. Syst. Softw.*, vol. 83, no. 4, pp. 689–701, 2010.

[49] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, "Evaluating non-adequate test-case reduction," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 16–26.

[50] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 49–60.

[51] D. Schuler and A. Zeller, "Assessing oracle quality with checked coverage," in *Proc. 4th IEEE Int. Conf. Softw. Testing Verification Validation*, 2011, pp. 90–99.

[52] D. Schuler and A. Zeller, "Checked coverage: An indicator for oracle quality," *Softw. Testing, Verification Rel.*, vol. 23, no. 7, pp. 531–551, 2013.
[53] X. Wang, S. Cheung, W. K. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 45–55.
[54] 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 *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 654–665.

Alexandre Perez received the MSc degree in informatics and computing engineering from the University of Porto, Portugal, and the PhD degree in informatics engineering from the University of Porto, Portugal. He has a background in software engineering and artificial intelligence research, with a focus on the diagnosis of software systems. His research interests include fault localization, code analysis, and language design. He is a member of the IEEE.

Rui Abreu received the MSc degree in computer and systems engineering from the University of Minho, Portugal, and the PhD degree in computer science—software engineering from the Delft University of Technology, The Netherlands. His research revolves around software quality, with emphasis in automating the testing and debugging phases of the software development lifecycle as well as self-adaptation. He has extensive expertise in both static and dynamic analysis algorithms for improving software quality. Before joining IST, ULisbon as an associate professor and INESC-ID as a senior researcher, he was a member of the Model-Based Reasoning group at PARC's System and Sciences Laboratory and an assistant professor with the University of Porto. He is a senior member of the IEEE.

Arie van Deursen received the PhD degree from the University of Amsterdam, in 1994. He is a professor in software engineering with Delft University of Technology, where is presently head of the Department of Software Technology. His research interests include software architecture, software testing, human aspects of software engineering, and AI-based software engineering. He will serve as program co-chair for ICSE 2021, the 43rd International Conference on Software Engineering. He is a member of the IEEE.