Diagnosing Software Faults Using Multiverse Analysis

Prantik Chatterjee[1*] , **Abhijit Chatterjee**[1] , **José Campos**[2] , **Rui Abreu**[3] and **Subhajit Roy**[1]

[1]Indian Institute of Technology, Kanpur, India

[2]LASIGE, Faculdade de Ciências, University of Lisbon, Portugal

[3]INESC-ID and IST, University of Lisbon, Portugal

{prantik, abhijitc, subhajit}@cse.iitk.ac.in, jcmcampos@fc.ul.pt, rui@computer.org

Abstract

Spectrum-based Fault Localization (SFL) approaches aim to efficiently localize faulty components from examining program behavior. This is done by collecting the execution patterns of various combinations of components and the corresponding outcomes into a *spectrum*. Efficient fault localization depends heavily on the quality of the spectra. Previous approaches, including the current state-of-the-art Density- Diversity-Uniqueness (DDU) approach, attempt to generate "good" test-suites by improving certain structural properties of the spectra. In this work, we propose a different approach, *Multiverse Analysis*, that considers multiple hypothetical universes, each corresponding to a scenario where one of the components is assumed to be faulty, to generate a spectrum that attempts to reduce the *expected worst-case wasted effort* over all the universes. Our experiments show that the Multiverse Analysis not just improves the efficiency of fault localization but also achieves better coverage and generates smaller test-suites over DDU, the current state-of-the-art technique. On average, our approach reduces the developer effort over DDU by over 16% for more than 92% of the instances. Further, the improvements over DDU are indeed statistically significant on the paired Wilcoxon Signed-rank test.

1 Introduction

Spectrum-based Fault Localization (SFL) techniques [Abreu *et al.*, 2009b] have proved to be immensely helpful at localizing faults in large code-bases [Pearson *et al.*, 2017]. These techniques aim to identify the faulty components, e.g., faulty lines of code, based on a statistical analysis of the test *spectra*. A *spectra* captures information on the *activity pattern* of each test case (which components are executed) and the *resulting outcomes* (which tests fail).

Given a faulty program P with m components $\{c_1, c_2, \ldots, c_m\}$, the *quality of the test suite* is the key factor for SFL to produce accurate diagnostic reports [Campos *et al.*, 2013]. Test suites can be created either manually or using automatic test generation techniques [Campos *et al.*, 2014]. A test-suite generator can produce a test-suite T on P by optimizing a fitness function f. The function f is taken as a measure of the quality of a test-suite. The *activity pattern* of a test-case $t \in T$ can be represented as a m-dimensional binary vector where the i-th element is 1 if the corresponding component c_i was executed (activated) in test t. The behavior of a *test-suite* consisting of n such test-cases can, therefore, be captured by a $n \times m$-dimensional binary matrix A, where the element a_{ij} is set to 1 if the i-th test executed the j-th component of P. This matrix is referred to as an *activity matrix*; the rows of A correspond to the activity pattern of test-cases while the columns correspond to the *involvement pattern* of the corresponding components. The matrix A is complemented with a vector E, *error vector*, that captures whether a t_i is found to be a *passing* test, then the corresponding entity $E_i = 0$, and for *failing* tests, $E_i = 1$. The tuple (A, E) is referred to as the *spectrum*.

SFL techniques use this spectrum to rank the components of P by their *suspiciousness* of being faulty (several formulae to quantify suspiciousness exist [Pearson *et al.*, 2017; Lucia *et al.*, 2014; Wong *et al.*, 2016]). Developers are expected to examine the components in the (decreasing) order of their suspiciousness scores till the faulty component is identified; thus, one generally constructs an ordered list, L, by ranking the components in descending order of their suspiciousness.

As mentioned before, the effectiveness of the SFL technique heavily depends on the *quality* of the test-suite for diagnosabilty, and can be measured by the *rank* of the (ground-truth) faulty component in L: if the rank of the faulty component is lower, then a developer would need to examine a fewer number of components before she identifies the faulty one. This effort, termed as the *cost of diagnosis* ($\mathcal{D}$), is measured as $\mathcal{D} = \frac{r}{m}$, where r is the rank of the faulty component in L and m is the total number of components. We can also define the cost of diagnosis by the *wasted effort* ($\mathcal{W}$), that captures the effort that is wasted in examining non-faulty components before hitting the faulty one: $\mathcal{W} = \frac{r-1}{m-1}$.

A recent work [Perez *et al.*, 2017b] claims that test-suites with good scores on adequacy metrics (like coverage) do not necessarily imply that these test-suites offer good diagnosability. In fact, the authors performed rigorous experiments

[*]Contact Author

to prove the contrary and proposed a new metric, DDU, that attempts to capture the quality of test-suites for SFL. Interestingly, such metrics can be plugged as fitness functions within Search-Based Software Testing (SBST) [McMinn, 2011] to automatically generate test-suites with the desired properties. One such popular search-based unit test generator, EvoSuite [Fraser and Arcuri, 2011], accepts a program P, a fitness function f and employs a genetic algorithm to generate *good* test-suites by optimizing f. EvoSuite also provides fault oracles in the form of a set of assertions which model the behavior of P. By examining deviations from these assertions, EvoSuite can produce test outcomes and generate the error vector for a test-suite.

In this work, we propose a new metric, Ulysis, to capture the quality of test-suites for diagnosability. Instead of utilizing the structural properties of the activity matrix that are likely to be good proxies of test-case diagnosability (the road taken by prior efforts, like DDU [Perez *et al.*, 2017b]), our metric directly considers multiple hypothetical universe (collectively defined as a multiverse), each universe assuming a component to be faulty, and computes the *expected worst-case wasted effort* for each of these hypothetical universe. We implement our metric in EvoSuite and perform experiments on real-life software faults from the Defects4J benchmark (version 1.4.0). Our experiments demonstrate that Ulysis outperforms the current state-of-the-art metric, DDU, on all the relevant metrics: diagnosability, coverage and size of test-suites. The Wilcoxon signed-rank test showed that our fault localization improvements over DDU are indeed statistically significant.

The following are the contributions of this work:

- We propose a new metric, Ulysis, to measure the diagnosability of test-suites, essentially computing the expected worst-case wasted effort instead of using proxies for good diagnosability as used in previous works;
- We implement our metric as a fitness function in EvoSuite and evaluate the test-suites generated by our metric versus those by DDU and coverage.

2 Our Approach

2.1 Ulysis: Multiverse Analysis

The test-generation metrics accept an activity matrix A as an input and provide a score that quantifies the *goodness* of the test-suite[1]. Given an activity matrix A, as the faulty component is not known, we design a metric that attempts to reduce the worst-case wasted effort for all components. Given a program P with a set of m components $C = \{c_1, c_2, \ldots, c_m\}$, we consider m *hypothetical universe* (multiverse): the component c_i is assumed to be faulty in the i-th hypothetical universe. Hence, the hypothetical universe $\mathcal{Z}_i$ operates on a spectrum consisting of the activity matrix A, with a hypothetical error vector E_i according to *what the error vector would have been if c_i was (persistently) faulty*. This synthesized error vector for $\mathcal{Z}_i$ is, thus, nothing but the involvement pattern of c_i—a test passing whenever c_i is not activated and failing whenever it is.

For each such hypothetical universe $\mathcal{Z}_i$, we compute the *worst case wasted effort*. Worst-case wasted effort is nothing but the effort we waste to localize c_i as the faulty component in the hypothetical universe $\mathcal{Z}_i$ in the worst case. Clearly, c_i will be the most likely faulty candidate in $\mathcal{Z}_i$ as $c_i = E_i$, i.e., the involvement pattern matches perfectly with the error vector. However, all components from $\{c_1, c_2, \ldots, c_m\}$ that have the same involvement pattern as c_i will also have the same likelihood of being faulty in $\mathcal{Z}_i$. Assuming the number of such components is r, we will end up examining these r components before identifying c_i as the actual faulty component in the worst-case scenario. From this observation, we define a *highest ambiguity set* $\mathcal{L}_i$ as:

$$\mathcal{L}_i = \begin{cases} c_j \mid c_j \in C, j \neq i, & \text{if } c_i = \vec{0} \\ c_j \mid c_j \in C, c_j = c_i, j \neq i, & \text{otherwise.} \end{cases} \quad (1)$$

Hence, the *highest ambiguity set* $\mathcal{L}_i$ contains all these components that, due to having the same involvement pattern as c_i, cannot be distinguished from c_i by any similarity-based fault localization algorithm. As these elements in $\mathcal{L}_i$ are components which, in the worst case, will be examined before c_i, the cardinality of $\mathcal{L}_i$ can be taken as measure of the *worst-case wasted effort* for localizing c_i given $\mathcal{Z}_i$. We, thus, compute the worst case wasted effort in $\mathcal{Z}_i$ as: $\mathcal{W}_i = \frac{|\mathcal{L}_i|}{m-1}$.

In an ideal scenario, where we should be able to perfectly localize c_i as the faulty component with zero wasted effort, $\mathcal{W}_i = 0$ (the highest ambiguity group contains only c_i) and in the worst case scenario, where we will end up examining all other candidates before finally identifying c_i as the faulty component, $\mathcal{W}_i = 1$ (the highest ambiguity group contains all components other than c_i). Therefore, our objective is to minimize $\mathcal{W}_i$ for all components while generating test-suites. Hence, we can define the overall quality of the test-suite represented by A as the expectation over all $\mathcal{W}_i$: $\mathcal{W}_{Ulysis} = \sum_{i=1}^{m} p(c_i).\mathcal{W}_i$ where, $p(c_i)$ is our prior belief about c_i being the actual faulty component.

A previous work [Paterson *et al.*, 2019] has attempted to extract such possible distributions by past history of failures, number of repository commits etc. Without prior knowledge (as done in this work), we assume uninformed prior knowledge where all components are assumed equally likely to be faulty, i.e., $p(c_1) = p(c_2) = \cdots = p(c_m) = 1/m$. Thus $\mathcal{W}_{Ulysis}$ becomes: $\mathcal{W}_{Ulysis} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{W}_i$.

We refer to $\mathcal{W}_{Ulysis}$ as the Ulysis score. Since, enhancing the quality of a test-suite can be expressed in terms of minimizing the Ulysis score, we can plug in Ulysis score as a fitness function in any SBST tool which aim to generate test-suites by optimizing the given fitness function.

There is one major advantage of using $\mathcal{W}_{Ulysis}$ to measure the quality of A instead of DDU. Consider a situation where a particular component c_k was never executed in any test case. In such cases, the k-th column of A will contain all 0 values. If c_k is a 0 vector, then the corresponding imaginary error vector E_k in the hypothetical universe $\mathcal{Z}_k$ will also be a 0 vector. In that case, following Eqn. 1, $\mathcal{L}_k$ will contain all the $(m-1)$ components from C other than c_k. Consequently,

[1] Note that the test-generation metrics do not have access to the error vector as the test-generation phase does not have access to the fault oracles. Fault localization is performed post test-generation.

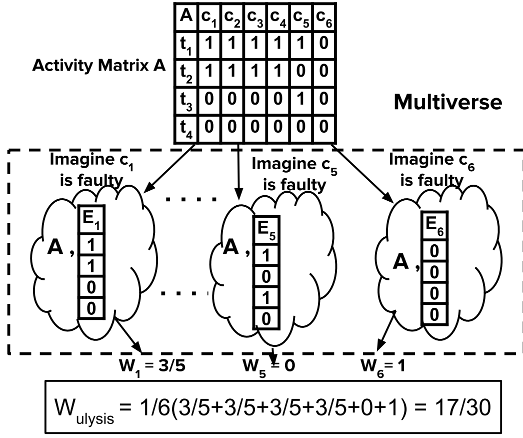

Figure 1: Multiverse analysis.

the value of $\mathcal{W}_k$, i.e., the worst-case wasted effort to identify c_k as faulty, will be 1. Therefore, to minimize $\mathcal{W}_k$, the k-th component c_k must be executed at least once in any test case. This is important because, if c_k is the faulty component and it is never executed, then there is no way for us to identify c_k as the faulty component. Therefore, optimizing our proposed metric will result in higher coverage of a program as well.

We give a brief demonstration of how to compute $\mathcal{W}_{Ulysis}$ using an example shown in Figure 1. We start by assuming a hypothetical universe $\mathcal{Z}_1$ where c_1 is the faulty component. Then, the corresponding imaginary error vector will have the same pattern as c_1. This imaginary error vector in $\mathcal{Z}_1$ is shown by E_1. Since, components $\{c_2, c_3, c_4\}$ share the same involvement patterns as c_1, $\mathcal{L}_1 = \{c_2, c_3, c_4\}$. Therefore, $\mathcal{W}_1 = \frac{|\mathcal{L}_1|}{m-1} = \frac{3}{5}$. Similarly, $\mathcal{W}_2 = \mathcal{W}_3 = \mathcal{W}_4 = \frac{3}{5}$. Now, when we assume c_5 to be the faulty component in a hypothetical universe $\mathcal{Z}_5$, E_5 becomes the corresponding imaginary error vector and $\mathcal{L}_5 = \phi$ as no other component shares the same involvement pattern as c_5. Therefore, $\mathcal{W}_5 = 0$. When we assume c_6 to be faulty, the corresponding imaginary error vector E_6 in $\mathcal{Z}_6$ is a 0 vector as c_6 is never executed in any test-case. Therefore, $\mathcal{W}_6 = 1$. Hence, $\mathcal{W}_{Ulysis} = \frac{1}{6}(\frac{3}{5} + \frac{3}{5} + \frac{3}{5} + \frac{3}{5} + 0 + 1) = \frac{17}{30}$. Our formulation makes two simplifying assumptions:

- **Single fault**: We assume that the program has a fault in a single component.
- **Perfect detection**: We assume that the tests do not exhibit flakiness [Bell *et al.*, 2018], i.e., the outcome of a test case is failure if and only if the faulty component is triggered in that particular test case.

Note that, *single fault assumption* is a common assumption in most SFL-based works; further, Perez et al. [Perez *et al.*, 2017a] show that it is a fair assumption. The single fault assumption makes our model simple and efficient. Also, we assumed *perfect detection* instead of inserting random noise as: (1) we lose predictable behavior of a deterministic fitness function, (2) with no knowledge of the magnitude and direction of the noise in the output, the noise introduced in the input will often turn additive, deteriorating the performance further. Our evaluations on the DEFECTS4J benchmark (which has real programs with multiple faults) show that Ulysis outperforms the state-of-the-art, even when the assumptions (including perfect detection) are violated.

Algorithm 1: Ulysis

1 population ← Initialize();
2 **while** *not converged and not timedout* **do**
3   $T \leftarrow \emptyset$;
4   **for** *each test-suite t in population* **do**
5   **for** *each row i in the activation matrix A of t* **do**
6       $$\mathcal{W}_i = \begin{cases} 1, & \text{if } \vec{A[i]} = \vec{0} \\ \frac{1}{m-1} \sum_{j=1:m, j\neq i} (\vec{A[j]} = \vec{A[i]}), & \text{otherwise} \end{cases}$$
7     $\mathcal{W}_t = \frac{1}{m} \sum_{i=1:m} \mathcal{W}_i$;
8     $T \leftarrow T \cup \langle t, W_t \rangle$;
9   population ← SelectTopTest-suites(T);
10 **return** *population*

2.2 Algorithm

Algorithm 1 describes our test-suite generation in EVOSUITE using Ulysis. The underlying evolutionary algorithm in EVOSUITE starts by initializing a population of test-suites (Line 1). Then, the algorithm is guided through the search-space by our fitness function (Lines 2-9), i.e., the Ulysis approach. In detail, for each test suite t in the population (Line 4), the algorithms uses the $t's$ activation matrix A (Line 5) to compute the worst-case wasted effort $\mathcal{W}_i$ by assuming each component j is faulty one at a time. Then, it computes the Ulysis score $\mathcal{W}_t$ of t by averaging all worst-case wasted efforts $\mathcal{W}_i$ (Line 7). Finally, it collects all Ulysis scores of all test suites (Line 8). On each iteration of the evolutionary algorithm, the population is refined by selecting the top test-suites based on w_t from T (Line 9) until the fitness function converges or the time budget is exceeded.

3 Ulysis versus DDU

Density- Diversity-Uniqueness (DDU) [Perez *et al.*, 2017b], the state-of-the-art metric for diagnosability, uses three structural properties of an activity matrix A to generate good test-suites in terms of their fault localization capability.

Density. Given a program P with m components and a test-suite with n tests, the *density* of an activity matrix A is defined as: $\rho = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} A_{ij}}{n \times m}$. This metric essentially attempts to improve the entropy of the activity matrix, and hence, the ideal value of density is 0.5.

Diversity. Test-cases having the same activity pattern are redundant, only increasing the size of the test-suite. Test-cases should be diverse, i.e., execute different combinations of components. The diversity measure [Perez *et al.*, 2017b] tries to ensure that each test pattern in A (rows of A) is unique. Mathematically this is expressed as the Gini-Simpson index [Jost, 2006]: $\mathcal{G} = 1 - \frac{\sum_{i=1}^{k} n_i \times (n_i - 1)}{n \times (n-1)}$, where k represents the number of groups of test cases having unique activity patterns, n_i is the number of test cases having the same

Activity Matrix / Imaginary Error Vectors (a)

A	c_1	c_2	c_3	c_4	c_5	c_6	E_1	E_2	E_3
t_1	1	1	1	1	0	0	1	0	0
t_2	1	1	1	1	0	1	1	0	1
t_3	1	1	1	1	1	0	1	1	0
t_4	0	0	0	0	0	1	0	0	1
t_5	0	0	0	0	1	0	0	1	0
t_6	0	0	0	0	1	1	0	1	1

Activity Matrix / Imaginary Error Vectors (b)

A	c_1	c_2	c_3	c_4	c_5	c_6	E_1	E_2	E_3
t_1	0	0	0	0	1	1	0	0	1
t_2	0	0	1	1	0	0	0	1	0
t_3	1	1	0	0	0	0	1	0	0
t_4	0	0	1	1	1	1	0	1	1
t_5	1	1	1	1	0	0	1	1	0
t_6	1	1	0	0	1	1	1	0	1

Figure 2: This example demonstrate a scenario where Ulysis is able to judge (b) as a better test-suite than (a) whereas DDU considers them both to be of the same quality.

activity pattern belonging to group i and n is the total number of test cases. Essentially, $\mathcal{G}$ measures how likely is it for two test cases, chosen at random from A, to have the same activity pattern. If all test cases are unique, then the value of $\mathcal{G}$ is 1.

Uniqueness. The uniqueness measure [Baudry *et al.*, 2006] ensures that the number of components having the same involvement pattern (columns of A) is reduced. To formulate uniqueness, we first define *ambiguity groups*: if one or more components share the same involvement pattern then we say that these components form an ambiguity group. Uniqueness of a test-suite is measured as: $\mathcal{U} = \frac{l}{m}$, where l represents the number of ambiguity groups in A and m is the total number of components. For a good test-suite, the value of $\mathcal{U}$ should be 1, i.e., all component patterns should be unique.

Finally, DDU is defined as: $\boxed{(1 - |1 - 2\rho|) \times \mathcal{G} \times \mathcal{U}.}$

Let us analyze DDU for diagnosability: Figures 2(a) and (b) demonstrate two test-suites from a program having six components where the value of $\mathcal{U}$ is less than 1 (generally $\mathcal{U} = 1$ is infeasible due to branch correlations). For both of the test-suites, density (ρ) is 0.5 and diversity ($\mathcal{G}$) is 1 as all the test cases have unique activation patterns. For the test-suite in Figure 2(a), there are three ambiguity groups: $\{c_1, c_2, c_3, c_4\}$, $\{c_5\}$ and $\{c_6\}$. Therefore, the uniqueness score $\mathcal{U}$ is $\frac{3}{6} = 0.5$. Similarly, the test-suite in Figure 2(b) contains three ambiguity groups as well: $\{c_1, c_2\}$, $\{c_3, c_4\}$ and $\{c_5, c_6\}$. Hence, the value of $\mathcal{U}$ is again 0.5 in this case. Therefore, according to the DDU metric, both of these test-suites are equally good in terms of effort spent to localize the faulty component in the corresponding program, and hence, any of these test-suites are equally likely to be selected.

Now, let us analyze two possible scenarios:

The test-suite in Figure 2(a) is chosen. If any of the components ($\{c_1, c_2, c_3, c_4\}$) is faulty, we would end up examining 60% of the program components before we are able to identify the actual fault. For real programs, which may contain thousands of components, this would be disastrous. However, if the component c_5 is the faulty component, c_5 will be identified with zero wasted effort—a *lucky* situation!

The test-suite in Figure 2(b) is chosen. In this case, regardless of whichever component is faulty, we will never have to examine more than 20% of the program components before discovering the actual fault *even in the worst case scenario*.

Hence, DDU is incapable of discriminating test-suites based on their *worst-case behavior*. Given that we have no prior knowledge about which component is faulty, it is therefore far more reasonable to select the second test-suite for efficient fault localization.

On the other hand, the Ulysis scores of the test-suites in Figures 2(a) and (b), are $\frac{2}{5}$ and $\frac{1}{5}$ respectively. This clearly demonstrates that, unlike DDU, Ulysis is capable of discriminating test-suites based on *worst-case scenarios*. Note that, optimizing to improve the worst case, reduces the chances of riding on *lucky* situations (as the case with DDU picking the first test-case and the component c_5 being faulty). This is seen in our experiments (Figure 3) where DDU performs exceedingly better in a few (7% instances) but the average decrease in effort while using Ulysis rather than Coverage is over 13% for more than 95% of all faults and the same over DDU is over 16% for more than 92% instances.

4 Experiments

Test-suites can be evaluated on three criteria:

- **Coverage**: Though not a good diagnosability metric [Staats *et al.*, 2012], coverage is still an important metric that allows faults to be triggered. Note that, diagnosability metrics are helpless unless failing tests are found.
- **Diagnosability**: This metric captures low wasted effort (or high suspiciousness scores) for ground-truth faults, given fault triggering tests are available.
- **Cost**: This captures the runtime cost of testing. Smaller test-suites are preferred over bigger test-suites.

We pose three research questions to evaluate the performance of our proposal.

RQ1 What is the saving of developer effort by our proposal over prior techniques?

RQ2 Is the improvement in the ranking of the faulty component by our proposal indeed statistically significant over prior techniques like DDU and coverage?

RQ3 Is the quality of test-suites (size and coverage) produced by our technique better than existing techniques?

We have performed our experiments on DEFECTS4J version 1.4.0 [Just *et al.*, 2014] which is a benchmark suite consisting of six diverse Java project repositories. DEFECTS4J contains 395 real-life software faults. Given that our experimental evaluations show an improvement in fault localization over the DEFECTS4J benchmark suite, the results should generalize.

We have implemented Ulysis[2] as a fitness function within EVOSUITE [Fraser and Arcuri, 2011] and compared it with other state-of-the-art fitness functions such as DDU and coverage [Fraser and Arcuri, 2015] (available within EVOSUITE). DDU [Perez *et al.*, 2017b] was shown to be better than current SFL techniques. So, we restricted our comparison to only DDU and Coverage. To take into account the randomization within EVOSUITE, for each fault, we have generated 5 test-suites using a time limit of 600 seconds on each fitness function. Post test-suite generation, we perform

[2]Ulysis is available in EVOSUITE as part of pull request #293, https://github.com/EvoSuite/evosuite/pull/293.

$\Delta\mathcal{W}$	cov	DDU	top-k	*Ulysis*	DDU
>0	59.05%	54.96%	5	34.11%	23.88%
$=0$	12.38%	13.51%	10	44.19%	35.07%
<0	28.57%	31.53%	50	70.54%	66.42%

Table 1: Comparisons w.r.t. wasted effort and top-k.

fault localization with the Ochiai metric using the GZOLTAR tool [Campos *et al.*, 2012]. The reason we use Ochiai is because it usually outperforms other similarity based metrics [Abreu *et al.*, 2006], [Pearson *et al.*, 2017] and it is as good as Bayesian Reasoning techniques, if we assume single faults [Abreu *et al.*, 2009c]. Both Ochiai and Spectrum-based Reasoning approaches like Barinel [Abreu *et al.*, 2009a] produce the theoretically optimal ranking under perfect detection and single-fault assumptions (assumption in most SFL-work, including ours). Finally, it is known that other approaches like Model Based Reasoning are computationally prohibitive on large codebases [Wong *et al.*, 2016] such the ones used in our experiments.

Our experimental methodology of generating tests on the golden version is a standard setup for all SFL-related work. Of course, this methodology assumes a test-oracle to capture the program specification. Generation of the test-oracle is orthogonal, and a concern of the base framework (EVOSUITE, in our case). In real-world scenarios, for maintenance projects, the previous version is often taken as the golden version for regression, and for others, oracles can be constructed from programmer annotations.

We did not consider all test-suites for fault localization as in some cases either EVOSUITE generated an empty test-suite or no failing test cases were present in the spectrum when executed on the faulty version for either Ulysis or the state-of-the-art approaches such as DDU and coverage. We found 111 such valid instances, on which we compare the median effort over 5 test-suite generation attempts with each of Ulysis, DDU and coverage. Note that, in our experiments, both Ulysis and DDU detect similar number of faults (in DEFECTS4J), showing that Ulysis is comparable to the state-of-the-art in fault detection. This also demonstrates that generation of either empty test-suites or test-suites without fault-triggering test cases is a limitation of EVOSUITE and not of the fitness functions [Shamshiri *et al.*, 2015]. All experiments are performed at branch granularity, i.e., the program components are branches. We have done these experiments on a 16 core virtual machine with Intel Xeon processors having 2.1 GHz core frequency and 32 gigabytes of RAM.

Threats to validity. In this study, we used faults taken from only six Java open source projects. Although our results may not generalize to other Java projects with different characteristics, we followed the setup proposed by others to ease the comparison with previous works. Also, our study only considered one test generation tool. There are, however, other tools that may generate test cases that improve even further the performance of the underline fault localization technique. We leave such analysis for future work.

4.1 RQ1: Fault Localization Performance

We quantify *goodness* of test-suites by their *wasted effort* $\mathcal{W}$. Given a fault, two fitness functions, say A and B, are compared by the *sign of the difference in wasted effort*, $\Delta\mathcal{W} = \mathcal{W}_A - \mathcal{W}_B$; of course, as a lower value of wasted effort is better, B is a better performing metric if $\Delta\mathcal{W} > 0$.

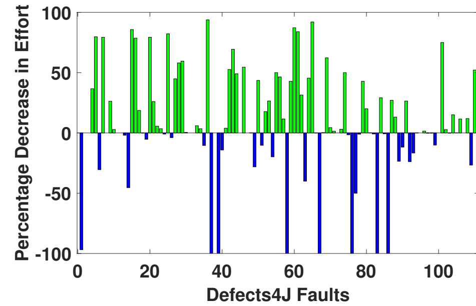

Figure 3: Percentage decrease of effort in fault localization while using Ulysis instead of DDU.

We denote $\Delta\mathcal{W}_{cov} = \mathcal{W}_{Coverage} - \mathcal{W}_{Ulysis}$ and $\Delta\mathcal{W}_{DDU} = \mathcal{W}_{DDU} - \mathcal{W}_{Ulysis}$, where $\mathcal{W}_{Coverage}$, $\mathcal{W}_{DDU}$ and $\mathcal{W}_{Ulysis}$ represents the *wasted effort* needed to localize the fault on test-suites generated by Coverage, DDU and Ulysis (respectively).

Rows 1, 2 and 3 of Table 1 show the number of instances where $\Delta\mathcal{W}$ values are positive (Ulysis is better), zero (both metrics are equivalent ,i.e., difference within $1e-15$) and negative (Ulysis is worse). Ulysis is better than both the competing fitness functions in more than about 55% instances while being better or equivalent in about 68% instances.

In Table 1, we compare Ulysis and DDU using the $top-n$ metric which is a standard practice [Pearson *et al.*, 2017]. The $top-n$ metric shows the percentage of cases for each metric where the faulty component appears within the top-n positions of the ranked list after performing fault localization on the test-suites produced by the corresponding metric. As we can see, test-suites produced by Ulysis are usually superior with respect to the $top-n$ metric. The comparison between Ulysis and Coverage w.r.t $top-n$ metric is also similar.

Figure 3 details the percentage decrease in the fault localization effort while using $\mathcal{W}_{Ulysis}$ rather than $\mathcal{W}_{DDU}$ on all of our 111 (faulty) instances: Ulysis reduces effort in most instances. In some cases (around 7% instances), the competing metrics get "lucky" and are able to significantly decrease effort; few such cases are expected as per our discussion in Section 3. Since Ulysis optimizes fault localization efficiency over all components, we treat such instances as outliers. For fairness, we assume that any instances where one metric outperforms another by over 100%, are outliers and are not considered while summarizing the result (the graphs are truncated at 100% and -100% on both ends of the y-axis). Our results against coverage is similar (omitted for brevity).

In summary, the average decrease in effort while using Ulysis rather than Coverage is over 13% for more than 95% of all faults and the same over DDU is over 16% for more than 92% of all faults in DEFECTS4J.

4.2 RQ2: Statistical Significance

Having seen that Ulysis indeed seems to improve fault localization, we question if the improvement is indeed statistically significant? We take the effort needed for localiz-

Function	Cov	Uniq	Size	DDU	Ulysis
DDU	0.88	0.69	25	0.65	0.13
Coverage	**0.93**	0.35	**10**	0.14	0.16
Ulysis	0.92	**0.76**	18	0.33	0.08

Table 2: Comparison between the quality of test-suites generated by Coverage, DDU and Ulysis. Each value represents the median of each function.

ing each fault by coverage, DDU and Ulysis as individual data columns ($\mathcal{W}_{Coverage}, \mathcal{W}_{DDU}, \mathcal{W}_{Ulysis}$) and perform a Shapiro-Wilk test [Mohd Razali and Yap, 2011] for normality on each of these columns. The test refutes the null hypothesis that any of these data columns are from a normal distribution with 99% confidence. The corresponding p-values are $5.20e-14$, $7.02e-14$ and $1.98e-14$ respectively.

Having concluded that the effort data columns are not from a normal distribution, we perform a paired Wilcoxon Signed-rank test on $\Delta\mathcal{W}_{cov} = (\mathcal{W}_{Coverage} - \mathcal{W}_{Ulysis})$ and $\Delta\mathcal{W}_{DDU} = (\mathcal{W}_{DDU} - \mathcal{W}_{Ulysis})$ respectively. Our null hypothesis is that the medians of both $\Delta\mathcal{W}cov$ and $\Delta\mathcal{W}_{DDU}$ are 0, while the alternate hypothesis is that the medians are greater than 0. In both cases, we are able refute the null hypothesis with 99% confidence, with corresponding p-values being 0.0015 for $\Delta\mathcal{W}_{cov}$ and 0.0017 for $\Delta\mathcal{W}_{DDU}$.

4.3 RQ3: Quality of Test-suites (Size and Coverage)

In Table 2, we show the median values of coverage (Cov), DDU, uniqueness (Uniq), test-suite sizes in the number of test cases (Size) and the Ulysis score of the test-suites generated by DDU, Coverage and Ulysis respectively (with the best values set in bold). Not surprisingly, the test-suites generated by *coverage* attain the highest coverage with the least number of tests; however, the diagnosability for these test-suites is poor. Ulysis is comparable to the *coverage* metric in terms of percentage of component covered. Ulysis beats DDU, the current state-of-the-art fitness function for diagnosabilty on uniqueness, coverage and test-suite size. As discussed previously, uniqueness is a very important metric. Diagnosability of a test-suite is directly correlated with the uniqueness score and Ulysis scores higher than all the other metrics in this regard, being even higher than DDU that includes it as part of its fitness function. This indicates that optimizing on the expected worst-case wasted effort automatically optimizes this very important metric. Note that, the numbers are similar if we use mean instead of median.

5 Related Work

Related studies on fault localization primarily focus on two key aspects, test-suite generation and fault localization. Test-suite generation approaches can be broadly categorized into approaches that improve test-suite adequacy (such as branch coverage [Fraser and Arcuri, 2011]) versus test-suite diagnosability (e.g., [Perez *et al.*, 2017b]). It has been shown that test-case adequacy measures do not have a direct correlation with the effectiveness of fault localization [Staats *et al.*, 2012]. Other studies have demonstrated that coverage and size of test-suites together exhibit a stronger non-linear relation with the fault localization capabilities of a test-suites [Namin and Andrews, 2009]. Our approach, while focused on enhancing the diagnosability of test-suites, can also improve adequacy of test-suites.

Approaches directed at improving diagnosability have attempted to maximize the entropy of the activity matrix [Abreu *et al.*, 2009a], which, however, can lead to prohibitively large test-suites. Subsequent works [Gonzalez-Sanchez *et al.*, 2011] have attempted to contain this explosion in size by optimizing the density of the activity matrix (to an ideal 0.5). Other metrics, such as Uniqueness [Baudry *et al.*, 2006] and DDU [Perez *et al.*, 2017b] focus on enhancing certain structural properties of the activity matrix in order to improve fault localization performance of test-suites. While the objective of our approach is also to improve the diagnosability of test-suites, we choose to directly attack the fault localization performance instead of optimizing proxies for it. The test-suites generated by our approach are smaller than those generated by diagnosability and entropy optimization metrics such as DDU while providing comparable adequacy to those generated by adequacy enhancement metrics such as coverage. Since the fault localization performance of our approach is also better, this ensures that our test-suites are both more efficient and faster on virtue of being smaller and therefore, less computationally expensive and taking lesser time to execute. Another advantage is the interpretability of the Ulysis score of a test-suite. Since we assume that each component may be faulty while computing the score, it can also be interpreted as an estimate of the fault localization performance of any test-suite even prior to its execution. No other test-suite generation metric can provide such an estimate. Also, recent approaches which are orthogonal to the test-suite generation but use SFL [Liu *et al.*, 2019] can benefit from our work.

The diagnostic accuracy of similarity-based fault localization is dictated not just by the quality of the test suite, but also at the efficiency of the metric to compare component's involvement patterns with error vectors. There are many such metrics [Lucia *et al.*, 2014]; a recent study [Pearson *et al.*, 2017] has identified Ochiai to be amongst the best metrics. Note that, although we have chosen to perform fault localization using the Ochiai score, our approach is not dependant on any particular fault localization method.

6 Conclusion

We propose a test-suite diagnosability metric that, unlike previous state-of-the-art approaches, directly attacks the fault localization metric via a multiverse model, instead of using structural properties of the activity matrix as proxies (like density or uniqueness). The test-suites generated by our method are not only statistically better in terms of diagnostic accuracy, but they also provide comparable or better code coverage.

Acknowledgements

This material is based upon work supported by Fundação para a Ciência e Tecnologia (FCT), through project with ref. PTDC/CCI-COM/29300/2017, and by the research units UIDB/50021/2020, UIDB/00408/2020 and UIDP/00408/2020.

References

[Abreu *et al.*, 2006] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proc. of the 12th PRDC*, page 39–46. IEEE Computer Society, 2006.

[Abreu *et al.*, 2009a] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-Based Multiple Fault Localization. In *Proc. of the 2009 IEEE/ACM ASE*, page 88–99. IEEE Computer Society, 2009.

[Abreu *et al.*, 2009b] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A Practical Evaluation of Spectrum-Based Fault Localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[Abreu *et al.*, 2009c] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. A New Bayesian Approach to Multiple Intermittent Fault Diagnosis. In *Proc. of the 21st IJCAI*, page 653–658. Morgan Kaufmann Publishers Inc., 2009.

[Baudry *et al.*, 2006] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving Test Suites for Efficient Fault Localization. In *Proc. of the 28th ICSE*, page 82–91. ACM, 2006.

[Bell *et al.*, 2018] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. DeFlaker: Automatically Detecting Flaky Tests. In *Proc. of the 40th ICSE*, page 433–444. ACM, 2018.

[Campos *et al.*, 2012] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proc. of the 27th IEEE/ACM ASE*, page 378–381. ACM, 2012.

[Campos *et al.*, 2013] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. Entropy-based test generation for improved fault localization. In *Proc. of the 28th IEEE/ACM ASE*, page 257–267. IEEE Press, 2013.

[Campos *et al.*, 2014] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. Continuous test generation: Enhancing continuous integration with automated test generation. In *Proc. of the 29th ACM/IEEE ASE*, page 55–66. ACM, 2014.

[Fraser and Arcuri, 2011] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proc. of the 19th ACM ESEC/FSE*, page 416–419. ACM, 2011.

[Fraser and Arcuri, 2015] Gordon Fraser and Andrea Arcuri. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Softw. Engg.*, 20(3):611–639, 2015.

[Gonzalez-Sanchez *et al.*, 2011] Alberto Gonzalez-Sanchez, Hans-Gerhard Gross, and Arjan JC van Gemund. Modeling the Diagnostic Efficiency of Regression Test Suites. In *Proc. of the 4th IEEE ICSTW*, pages 634–643, 2011.

[Jost, 2006] Lou Jost. Entropy and diversity. *Oikos*, 113(2):363–375, 2006.

[Just *et al.*, 2014] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of the 23rd ISSTA*, page 437–440. ACM, 2014.

[Liu *et al.*, 2019] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proc. of the 12th IEEE ICST*, pages 102–113, 2019.

[Lucia *et al.*, 2014] Lucia Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended Comprehensive Study of Association Measures for Fault Localization. *J. Softw. Evol. Process*, 26(2):172–219, 2014.

[McMinn, 2011] Phil McMinn. Search-Based Software Testing: Past, Present and Future. In *Proc. of the 4th IEEE ICSTW*, pages 153–163, 2011.

[Mohd Razali and Yap, 2011] Nornadiah Mohd Razali and Bee Yap. Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests. *J. Stat. Model. Analytics*, 2:21–33, 2011.

[Namin and Andrews, 2009] Akbar Siami Namin and James H. Andrews. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proc. of the 18th ISSTA*, page 57–68. ACM, 2009.

[Paterson *et al.*, 2019] David Paterson, José Campos, Rui Abreu, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *Proc. of the 12th IEEE ICST*, pages 346–357, 2019.

[Pearson *et al.*, 2017] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and Improving Fault Localization. In *Proc. of the 39th ICSE*, page 609–620. IEEE Press, 2017.

[Perez *et al.*, 2017a] Alexandre Perez, Rui Abreu, and Marcelo d'Amorim. Prevalence of Single-Fault Fixes and Its Impact on Fault Localization. In *Proc. of the 10th IEEE ICST*, pages 12–22, 2017.

[Perez *et al.*, 2017b] Alexandre Perez, Rui Abreu, and Arie van Deursen. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. In *Proc. of the 39th ICSE*, page 654–664. IEEE Press, 2017.

[Shamshiri *et al.*, 2015] Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proc. of the 30th IEEE/ACM ASE*, pages 201–211, 2015.

[Staats *et al.*, 2012] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. On the danger of coverage directed test case generation. In *FASE*, pages 409–424. Springer Berlin Heidelberg, 2012.

[Wong *et al.*, 2016] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A Survey on Software Fault Localization. *IEEE TSE*, 42(8):707–740, 2016.