Diagnosing Software Faults Using Multiverse Analysis

Abstract

Spectrum-based Fault Localization approaches aim to efficiently localize faulty components within a buggy program by collecting the execution patterns of various combinations of components and the corresponding outcomes into a *spectrum* that models the program behavior. Efficient fault localization, i.e., locating the faulty component with less effort, depends heavily on the quality of the spectrum. Previous approaches, including the current state-of-the-art *DDU* approach, attempt to generate "good" test-suites for efficient fault localization by improving certain structural properties of the spectra.

In this work, we propose a different approach, *Multiverse Analysis*, that considers multiple hypothetical universe, each universe 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 expected effort* over all the universe. 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. Further, we found that the improvements over DDU for fault localization are indeed statistically significant on the paired Wilcoxon Signed-rank test.

1 Introduction

Spectrum-based fault localization (SFL) techniques (Abreu et al. 2009) have proved immensely helpful for localizing faults in large code-bases (Pearson et al. 2017). The SFL techniques attempt to identify the faulty components based on a statistical analysis of the test *spectra*, that captures information on the *activity pattern* of each test in the test-suite (which test executes which components) and the *resulting outcomes* (which tests fail).

The workflow of SFL is demonstrated in Figure 1. Given a faulty program P with m components $\{c_1, c_2, \ldots, c_m\}$, 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 an $n \times m$-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.

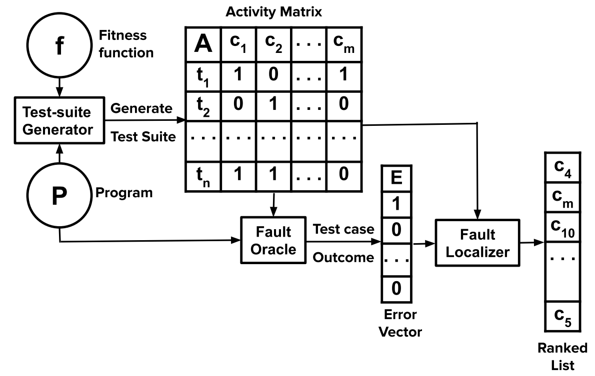


Figure 1: SFL approach: The test-suite generator takes as input a program P and a fitness function f and generates a test-suite in the form of an activity matrix A. The fault oracle takes A and P and generates an error vector E by executing A on P. The fault localizer takes A and E and ranks the components of P in descending order of suspiciousness

After T has been generated, a fault oracle executes T on P and provides the outcome of each test-case t. The fault oracles have access to a set of assertions which model correct program behaviour. By examining deviations from the assertions, the oracle can deduce whether a test-case is passing or failing. The outcome of the test-cases is captured by an n-dimensional binary *error vector*, E; if a test t_i is found to be a *passing* test, then the corresponding entity $E_i = 0$, and for *failing* tests, $E_i = 1$. The activity matrix, A, and the error vector, E, are together referred to as the *spectrum* over the program P.

SFL techniques use this spectrum to rank the components of P by their *suspiciousness* of being faulty. Over the years,

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

researchers have proposed various statistical metrics for the above ranking: Ochiai (Abreu, Zoeteweij, and Van Gemund 2009a), Tarantula (Jones and Harrold 2005), Zoltar (Janssen, Abreu, and Van Gemund 2009), Ample (Dallmeier, Lindig, and Zeller 2005), and Jaccard (Chen et al. 2002) are some of the popular ones. 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 (see Figure 1).

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 we need to examine fewer number of components before we identify 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, Abreu, and van Deursen 2017) 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 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 Test-generators (SST) (McMinn 2011) to automatically generate test-suites with the desired properties. One such popular search-based test-generator, Evosuite (Fraser and Arcuri 2011), accepts a program P, a fitness function f and employs genetic algorithms to generate good test-suites by optimizing on 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, Abreu, and van Deursen 2017)), 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 benchmarks (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 (Woolson 2007) 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 agrees 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:

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

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 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:

$$\mathcal{W}_i = \frac{|\mathcal{L}_i|}{m-1} \quad (2)$$

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

[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. Localization is performed post test-generation.

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 $\mathcal{W}_i$:

$$\mathcal{W}_{Ulysis} = \sum_{i=1}^{m} p(c_i).\mathcal{W}_i \tag{3}$$

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 a uninformed prior 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 Eqn. 3 reduces to:

$$\mathcal{W}_{Ulysis} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{W}_i \tag{4}$$

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 Eqn. 4 as a fitness function in any Search-based Software Testing (SST) tool which aim to generate test-suites by optimizing the given fitness function.

There is one other advantage of using $\mathcal{W}_{Ulysis}$ to measure the quality of A. 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, the value of $\mathcal{W}_k$, following Eqn. 2 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 2. 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, following Eqn. 1, $\mathcal{W}_6 = 1$. Hence, following Eqn. 4,

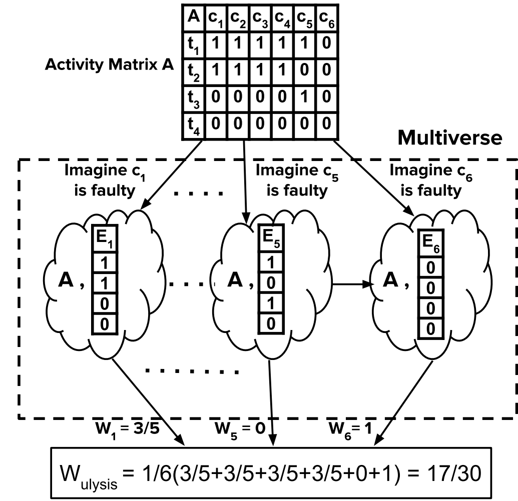


Figure 2: Multiverse analysis example: Form a multiverse, i.e., a set of hypothetical universe where we assume each component of a program to be faulty and synthesize an imaginary error vector in each universe. We compute the worst-case wasted effort in each universe and finally take an expectation of worst-case wasted effort over the multiverse as a measure of the quality of A

$\mathcal{W}_{Ulysis} = \frac{1}{6}(\frac{3}{5} + \frac{3}{5} + \frac{3}{5} + \frac{3}{5} + 0 + 1) = \frac{17}{30}$. 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.

Our experimental results demonstrate that our model works well for real-life programs even when the above assumptions do not hold.

2.2 Algorithm

Algorithm 1 takes an activity matrix A having n test cases and m components as input and computes the value of the fitness function Ulysis from Eqn. 4.

In Algorithm 1, line 1 initializes $\mathcal{W}$ to 0. Line 2 loops over each component c_i and imagines it to be the faulty component in a hypothetical universe $\mathcal{Z}_i$. Line 3 initializes the cardinality *h* of the highest ambiguity set $\mathcal{L}_i$ in $\mathcal{Z}_i$ to 0. Lines 4 and 5 sets the value of *h* to $(m-1)$ if the involvement pattern of the component c_i represented by the *i*-th column vector $\vec{A_i}$ is $\vec{0}$ (i.e. the respective component was never activated). Otherwise, lines 6 – 12 count the number of column vectors in *A* which has the same pattern as $\vec{A_i}$ and stores the count in *h*. Line 13 computes the value of worst case wasted effort $\mathcal{W}_i$ from Eqn. 2. At line 14, the value of $\mathcal{W}_i$ for each hypothetical universe is accumulated at $\mathcal{W}$ and finally at line 16, the expected value of the worst case wasted effort for the input activity matrix A is returned following Eqn. 4.

Algorithm 1 Ulysis

Input: Activity Matrix $A_{n \times m}$
Output: $\mathcal{W}_{Ulysis}$ **for** A

1: $\mathcal{W} \leftarrow 0$
2: **for** $i = 1$ to m **do**
3: $\quad h \leftarrow 0$
4: $\quad$ **if** $\vec{A[i]} == \vec{0}$ **then**
5: $\quad\quad h = m - 1$
6: $\quad$ **else**
7: $\quad\quad$ **for** $j = 1$ to m **do**
8: $\quad\quad\quad$ **if** $\vec{A[j]} == \vec{A[i]}$ $\&$ $j \neq i$ **then**
9: $\quad\quad\quad\quad h \leftarrow h + 1$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end if**
13: $\quad \mathcal{W}_i \leftarrow h/(m-1)$
14: $\quad \mathcal{W} \leftarrow \mathcal{W} + \mathcal{W}_i$
15: **end for**
16: **return** $\mathcal{W}/m$

3 Discussion

3.1 Overview of Density-Diversity-Uniqueness metric (DDU)

As the name suggests, the DDU metric uses the three following 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, Abreu, and van Deursen 2017) 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 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, Fleurey, and Le Traon 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.

Perez et al. (Perez, Abreu, and van Deursen 2017) have used a combination of the three measures stated above to define their own fitness function DDU as: $DDU = (1 - |1 - 2\rho|) \times \mathcal{G} \times \mathcal{U}$.

3.2 Ulysis versus DDU

As we have established in the previous section, DDU improves the efficiency of fault localization by optimizing three structural properties of the activity matrix. Optimizing density reduces the size of the test-suite while maintaining the quality, optimizing diversity reduces redundancy and ensures that the test cases explore different combinations of components and optimizing uniqueness ensures that the actual faulty component receives the highest possible suspiciousness score. We have seen from our previous discussion that if the value of the uniqueness measure ($\mathcal{U}$) is 1, i.e., each component pattern is unique, then the faulty component will surely have a high suspiciousness score. However, is it actually possible to reach this ideal scenario where we are able to generate a test-suite with a uniqueness value of 1? If we want to generate a test-suite where each component pattern is unique, then we must be able to execute each component independently of each other in the corresponding program. However, in real-life scenarios, this may not always be the case. Program components may be dependant on each other, such that if we execute one component, we may invariably end up executing some other component. In such cases, it will not possible to distinguish between these dependent component patterns and we will get an activity matrix containing ambiguity groups of size two or more. We

(a)

Activity Matrix

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

Imaginary Error Vectors

E_1	E_2	E_3
1	0	0
1	0	1
1	1	0
0	0	1
0	1	0
0	1	1

(b)

Activity Matrix

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

Imaginary Error Vectors

E_1	E_2	E_3
0	0	1
0	1	0
1	0	0
0	1	1
1	1	0
1	0	1

Figure 3: 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.

have also observed similar cases in our experiments where we generated test-suites on actual faulty programs (detailed descriptions of these faulty programs and the experimental setup is provided in the experiments section) by maximizing the DDU metric. The median value of the uniqueness measure $\mathcal{U}$ for these test-suites were 0.5 and the mean value was 0.47 with a standard deviation of 0.32. Since, it is usually not possible to get a test-suite with $\mathcal{U} = 1$, let us instead

observe the quality of test-suites where $\mathcal{U} < 1$.

Figures 3(a) and (b) demonstrate two such test-suites where the value of $\mathcal{U}$ is less than 1. Assume that we have a program having six components and we have generated two test suites on this particular program. For both of the test-suites in Figures 3(a) and (b), the value of density (ρ) is 0.5 and the value of diversity ($\mathcal{G}$) is 1 as all the test cases have unique activation patterns. For the test-suite in Figure 3(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 3(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.

Now, consider the simple fact that while performing fault localization we are unaware of which component is actually faulty. Therefore, to begin with, any of the six components is equally likely to be faulty. So, we will assume each component c_j to be faulty in turn and try to measure the wasted effort $\mathcal{W}_j$ needed to locate c_j as the faulty component in the worst case scenario for each of the test-suites shown in Figures 3(a) and (b).

For, the test-suite in Figure 3(a), if c_1 is the faulty component, then the outcome of each of the test-case where c_1 is executed, would be failure and the outcomes of the remaining test cases would be success. Again we assume that the outcome of a test case is always failure if the faulty component is executed and success otherwise. Therefore, if c_1 is the faulty component, then E_1 would be the corresponding error vector. Now, since components $\{c_1, c_2, c_3, c_4\}$ share the same involvement pattern and this involvement pattern is exactly the same as the error vector E_1, all these four components will have the highest suspiciousness score of 1 based on their similarity with E_1. Since, all of them has the same suspiciousness score, in the worst case scenario, we will end up examining $\{c_2, c_3, c_4\}$ before we arrive at the actual faulty component c_1. Therefore, from Eqn. 2, the worst-case wasted effort given c_1 is faulty can be measured as $\mathcal{W}_1 = \frac{3}{5} = 0.6$. Similarly, if we assume any of $\{c_2, c_3, c_4\}$ to be faulty, the worst-case wasted effort is going to be 0.6 in each case. However, for components c_5 and c_6, the scenario is different. The error vectors corresponding to the cases where we assume either c_5 or c_6 to be faulty, are shown as E_2 and E_3 respectively. Since, the involvement patterns of both c_5 and c_6 are unique, the worst-case wasted effort, if either of these components were faulty, would be 0.

Now, let us examine the test-suite in Figure 3(b). This test-suite contains three ambiguity groups of size two each: $\{c_1, c_2\}$, $\{c_3, c_4\}$ and $\{c_5, c_6\}$. E_1 represents the error vector if either c_1 or c_2 is faulty. Similarly, E_2 and E_3 represents the error vectors corresponding to the scenarios if we assume the components from the other two ambiguity groups to be faulty respectively. Again, using Eqn. 2, we can see that the worst-case wasted effort if c_1 is buggy, will be $\mathcal{W}_1 = \frac{1}{5} = 0.2$ since there is only one component c_2 which has the same involvement pattern as c_1 and therefore, in the worst case, may be examined before c_1. Similarly, we can see that the worst case wasted effort for all other scenarios where each of $\{c_2, c_3, c_4, c_5, c_6\}$ is buggy, will be 0.2 if we use this test-suite.

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. However, after examining them in detail we are inclined to ask if that is really the case. If we use the DDU metric to measure the quality of a test-suite, we may choose any of these two test-suites to localize the faulty component with the program. Let us assume, we choose the test-suite given in Figure 3(a) and component c_5 is the actual faulty component. Then we are in luck! As evident from our earlier discussion, we will be able to identify c_5 with zero wasted effort. However, if any of $\{c_1, c_2, c_3, c_4\}$ is faulty, then 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 will be disastrous.

On the other hand, if we choose the test-suite in Figure 3(b), then regardless of whichever component is faulty, we will never have to examine more than 20% of the program components before identifying the actual fault even in the worst case scenario. Given that we have no prior knowledge about which component is buggy, it is therefore far more reasonable to select this particular test-suite for our purpose of efficient fault localization.

Using our metric, the Ulysis scores of the test-suites in Figures 3(a) and (b) following Eqn. 4, are $\frac{2}{5}$ and $\frac{1}{5}$ respectively. This clearly demonstrates that, unlike DDU, our metric is able to make a finer distinction between these two cases and select the test-suite in Figure 3(b) as a better choice as this will result in a lesser wasted effort overall regardless of whichever component is faulty.

4 Experiments

Test-suites are typically evaluated on three criterion:

- **Coverage**: Though coverage is not a good diagnosability metric (Staats et al. 2012), it is still an important metric that allows faults to be triggered. Note that diagnosability metrics are helpless unless failing tests are found.
- **Diagnosability**: This represents the test-suites that show low wasted effort (or high ranks) for ground-truth faults on SFL techniques, given fault triggering tests are available.
- **Cost**: This captures the 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?

Table 1: Comparison of Ulysis, DDU and Coverage

	$\Delta\mathcal{W}_{cov}$	$\Delta\mathcal{W}_{DDU}$
$\Delta\mathcal{W} > 0$	59.05%	54.96%
$\Delta\mathcal{W} = 0$	12.38%	13.51%
$\Delta\mathcal{W} < 0$	28.57%	31.53%

We have performed our experiments on Defects4J version 1.4.0 (Just, Jalali, and Ernst 2014) which is a collection of Java project repositories. Defects4J contains 395 real-life software bugs.

We have implemented Ulysis as a fitness function within Evosuite (Fraser and Arcuri 2011)[2] and compared it with other state-of-the-art fitness functions such as DDU and coverage (Fraser and Arcuri 2015) (available within Evosuite). 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 (total time taken is more than 60 hours). Post test-suite generation, we perform 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, Zoeteweij, and Van Gemund 2006) and it is as good as Bayesian Reasoning techniques, if we assume single faults (Abreu, Zoeteweij, and Van Gemund 2009a). 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 buggy 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. 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.

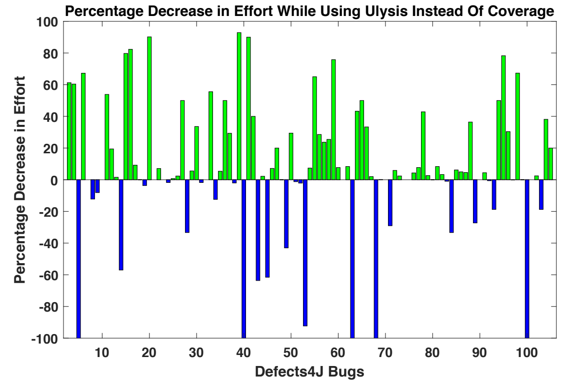
4.1 RQ1: Fault Localization Performance

We quantify *goodness* of test-suites by their *wasted effort* $\mathcal{W}$ (Eqn. 2). 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 $\Delta\mathcal{W} > 0$.

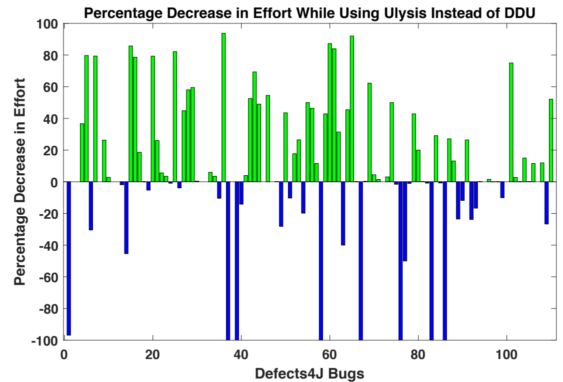
We denote $\Delta\mathcal{W}_{cov} = \mathcal{W}_{Coverage} - \mathcal{W}_{Ulysis}$ and $\Delta\mathcal{W}_{DDU} = \mathcal{W}_{DDU} - \mathcal{W}_{Ulysis}$, where $\mathcal{W}_{Coverage}$, $\mathcal{W}_{DDU}$ and $\mathcal{W}_{Ulysis}$ represents the *wasted effort* needed to localize the fault on test-suites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

In Table 1, rows 1, 2 and 3 record the number of instances where $\Delta\mathcal{W}$ values are positive (Ulysis is better), zero (both metrics are equivalent) and negative (Ulysis is worse). We find Ulysis is better than both the competing fitness functions in more than about 55% instances while being better or equivalent in about 68% instances.

[2]We plan to release our code post publication.



(a)



(b)

Figure 4: Percentage decrease of effort in fault localization while using Ulysis instead of DDU and Coverage.

Figures 4 (a) and (b) shows a detailed report on the percentage decrease in the fault localization effort while using $\mathcal{W}_{Ulysis}$ rather than $\mathcal{W}_{cov}$ and $\mathcal{W}_{DDU}$ on all of our 111 (faults) instances. It is seen that Ulysis reduces effort in most instances. In some cases, the competing metrics get "lucky" and are able to significantly decrease effort (the graphs are truncated at -100%); few such cases are expected as per our discussion in Section 3.2. Overall, the median of the percentage decrease in effort while using Ulysis rather than Coverage is 5.17% and the same over DDU is 3.48%.

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 localizing 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 (Razali, Wah, and others 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

Table 2: Comparison between the quality of test-suites generated by Coverage, DDU and Ulysis

Function	Median of Metrics				
	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

not from a normal distribution, we perform a paired Wilcoxon Signed-rank test (Woolson 2007) 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 diagnosabilty for these test-suites is poor. Ulysis is comparable to *coverage* in terms of coverage, albeit with slightly larger test-suites. 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.

5 Related work

Related studies on fault localization primarily focus on two key aspects, test-suite generation and fault localization (using the generated test-suites). Test-suite generation approaches can be broadly categorized into two types: approaches that seek to improve test-suite adequacy and approaches that seek to improve test-suite diagnosability.

The objective of the test-suite generation approaches, that fall into the first category, is to increase the adequacy of a generated test-suite. Adequacy of a test-suite refers to how thorough a test-suite is. Maximization of coverage criterions such as branch coverage (Fraser and Arcuri 2011) are examples of test-suite generation strategies using adequacy measures. Although, maximizing the coverage score of a test-suite ensures that the test-suite covers an adequate number of corresponding program components, it does 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). As we have demonstrated in Sections 2.1 and 4.3, our approach, while focused on enhancing the diagnosability of test-suites, can also improve the coverage of test-suites.

The second category of test-site generation approaches focus on improving the diagnosability, i.e., fault localization effectiveness of test-suites. One way to construct an ideal test-suite for fault localization is to maximize the entropy of the activity matrix (Abreu, Zoeteweij, and Van Gemund 2009b). Maximizing the entropy would produce a test-suite with all possible combinations of components, however, as a result, the test-suite may become prohibitively large. To circumvent this problem, another approach suggests optimizing the density of a test-suite to 0.5 which will result in maximization of entropy as well as a reduction in the size of a test-suite (Gonzalez-Sanchez, Gross, and van Gemund 2011). Other metrics, such as Uniqueness (Baudry, Fleurey, and Le Traon 2006) and DDU (Perez, Abreu, and van Deursen 2017) 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 ignore such proxies and instead focus on generating test-suites in such a way that the fault localization performance is improved.

Fault localization approaches use the test-suites produced by the above approaches and tries to assign suspiciousness scores to the program components based on how likely they are to be faulty. Similarity based approaches work by computing the similarity between the component involvement patterns and the error vector pattern. There exist several types of similarity scores, such as Ochiai (Meyer et al. 2004), Tarantula (Jones and Harrold 2005), Zoltar (Janssen, Abreu, and Van Gemund 2009), Ample (Dallmeier, Lindig, and Zeller 2005), Jaccard (Chen et al. 2002) etc., although a recent study has demonstrated that Ochiai usually outperforms others in this regard. Spectrum-based Reasoning approaches (SR) (Abreu, Zoeteweij, and Van Gemund 2009b) work by initially assigning a prior probability (likelihood of being faulty) to each of the components and then updating the posterior probabilities of each component using the Bayesian update rule.

6 Conclusion

We propose a test-suite diagnosability metric that generates test-suites with good fault-localization capability. Unlike previous state-of-the-art approaches, instead of using structural properties of the activity matrix as proxies (like density or uniqueness) for the "goodness" of test-suites, we improve diagnosability of test-suites by imagining multiple hypothetical universe in which we assume individual components to be faulty, and then try to reduce wasted effort to identify the fault over this multiverse. The test-suites generated by our method are not only statistically better in terms of fault localization than the diagnostic approaches, but they also provide comparable or better component coverage.

References

Abreu, R.; Zoeteweij, P.; Golsteijn, R.; and Van Gemund, A. J. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82(11):1780–1792.

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

Abreu, R.; Zoeteweij, P.; and Van Gemund, A. J. 2009a. A new bayesian approach to multiple intermittent fault diagnosis. In *Twenty-First International Joint Conference on Artificial Intelligence*.

Abreu, R.; Zoeteweij, P.; and Van Gemund, A. J. 2009b. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, 88–99. IEEE Computer Society.

Baudry, B.; Fleurey, F.; and Le Traon, Y. 2006. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, 82–91. ACM.

Bell, J.; Legunsen, O.; Hilton, M.; Eloussi, L.; Yung, T.; and Marinov, D. 2018. D e f laker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*, 433–444. ACM.

Campos, J.; Riboira, A.; Perez, A.; and Abreu, R. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 378–381. ACM.

Chen, M. Y.; Kiciman, E.; Fratkin, E.; Fox, A.; and Brewer, E. 2002. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, 595–604. IEEE.

Dallmeier, V.; Lindig, C.; and Zeller, A. 2005. Lightweight defect localization for java. In *European conference on object-oriented programming*, 528–550. Springer.

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

Fraser, G., and Arcuri, A. 2015. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering* 20(3):611–639.

Gonzalez-Sanchez, A.; Gross, H.-G.; and van Gemund, A. J. 2011. Modeling the diagnostic efficiency of regression test suites. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 634–643. IEEE.

Janssen, T.; Abreu, R.; and Van Gemund, A. J. 2009. Zoltar: a spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution@ runtime*, 23–30. ACM.

Jones, J. A., and Harrold, M. J. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 273–282. ACM.

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

Just, R.; Jalali, D.; and Ernst, M. D. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 437–440. ACM.

McMinn, P. 2011. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, 153–163. IEEE.

Meyer, A. d. S.; Garcia, A. A. F.; Souza, A. P. d.; and Souza Jr, C. L. d. 2004. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology* 27(1):83–91.

Namin, A. S., and Andrews, J. H. 2009. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, 57–68. ACM.

Paterson, D.; Campos, J.; Abreu, R.; Kapfhammer, G. M.; Fraser, G.; and McMinn, P. 2019. An empirical study on the use of defect prediction for test case prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 346–357. IEEE.

Pearson, S.; Campos, J.; Just, R.; Fraser, G.; Abreu, R.; Ernst, M. D.; Pang, D.; and Keller, B. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, 609–620. IEEE Press.

Perez, A.; Abreu, R.; and van Deursen, A. 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*, 654–664. IEEE Press.

Razali, N. M.; Wah, Y. B.; et al. 2011. Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics* 2(1):21–33.

Staats, M.; Gay, G.; Whalen, M.; and Heimdahl, M. 2012. On the danger of coverage directed test case generation. In *International Conference on Fundamental Approaches to Software Engineering*, 409–424. Springer.

Woolson, R. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials* 1–3.