DDU

Master's Thesis

Aaron Ang

DDU

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Aaron Ang
born in Amstelveen, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

PARC, a Xerox company
3333 Coyote Hill Road
Palo Alto, CA 94304
www.parc.com

DDU

Author: Aaron Ang
Student id: 4139194
Email: a.w.z.ang@student.tudelft.nl

Abstract

ABSTRACT

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Company supervisor:	Prof. Dr. R. Maranhao, University of Lisbon

Preface

This is where you thank people for helping you etc.

Aaron Ang
Delft, The Netherlands
February 12, 2018

Contents

List of Figures

Chapter 1

Introduction

Software systems are complex and error-prone, likely to expose failures to the end user. When a failure occurs, the developer has to debug the system to eliminate the failure. This debugging process can be described in three phases [11]. In the first phase, the developer has to pinpoint the fault, also known as the root cause, in code that causes the failure. In the second phase, the developer has to develop an understanding of the root cause and its context. Finally, in the third phase, the developer has to implement a patch that corrects the behavior of the system. This process is time-consuming and can account for 30% to 90% of the software development cycle [14, 4, 5].

Traditionally, developers use four different approaches to debug a software system, namely program logging, assertions, breakpoints and profiling [20]. Program logging is the act of inserting *print* statements in the code to observe program state information during execution. Assertions are constraints that can be added to a program that have to evaluate to true during execution time. Breakpoints allow the developer to pause the software system during execution, and observe and modify variable values. Profiling is used to perform runtime analysis and collect metrics on, for example, execution speed and memory usage. These techniques provide an intuitive approach to localize the root cause of a failure, but, as one might expect, are less effective in the massive size and scale of software systems today.

Therefore, in the last decades a lot of research has been performed on improving and developing *advanced* fault localization techniques [20] such that they are applicable to the software systems of today. Specifically, a prominent fault localization technique is spectrum-based fault localization (SBFL). SBFL techniques pinpoint faults in code based on execution information of a program, also known as a program spectrum [13]. The output of SBFL techniques is a list of components ranked by their fault probability, e.g. statements or methods. Intuitively, if a statement is executed primarily during failed executions, then this statement might be assigned a higher suspiciousness score. Similarly, if a statement is executed primarily during successful executions, then this statement might be assigned a lower suspiciousness score.

While SBFL techniques show promising results for debugging purposes, there are still aspects that prevent SBFL techniques from being used in practice. First, in one of the pioneering user studies performed by Parnin and Orso [11], the authors show that several assumptions made by SBFL techniques do not hold in practice. For example, the authors find no evidence that the effectiveness of SBFL techniques is affected by the rank of the faulty statement. Second, the diagnostic performance of SBFL techniques relies on the quality of the test suite. SBFL techniques cannot accurately pinpoint a fault with a single failing test that covers multiple components, e.g. statements, branches, methods. However, if the test suite consists of many tests with different coverage patterns, then SBFL might be able to pinpoint the fault more accurately because it can indict or exonerate components more precisely with more evidence, i.e. more execution traces.

The quality of test suites is commonly measured using an adequacy metric that focuses on error detection like statement coverage, branch coverage, method coverage. However, these metrics do not necessarily enforce the quality of a test suite with respect to diagnosability — the property of faults to be easily and precisely located [16]. For example, one could obtain a 100% test coverage with a relative small number of tests that each covers a big part of the codebase. In this case, when a test fails due to a fault, it might be difficult for SBFL techniques to pinpoint the root cause accurately since the failing test covers a large portion of the codebase, lowering the possibility for SBFL techniques to accurately indict or exonerate components from being faulty.

For this reason, Perez *et al.* [12] propose a new metric, called DDU, to quantify the diagnosability of test suites as opposed to adequacy metrics that focus on error detection. The objective of optimizing test suites with respect to DDU is to improve the diagnosability of test suites and consequently improve the diagnostic performance of SBFL techniques. The authors have shown empirical evidence that optimizing a test suite with respect to DDU improves the diagnostic performance of SBFL by 34% compared to a test suite optimized with respect to branch coverage.

1.1 Problem Definition

However, DDU is not yet usable in practice. Currently, when the DDU is computed for a given test suite, its value lies in the domain $[0, 1]$, where 0 suggests that the test suite's diagnosability is low, and 1 suggests that the test suite's diagnosability is high. The problem with this value is that the developer does not know how to extend or update the test suite given a DDU value. For example, when the test suite's DDU is equal to 0.1, the developer does not know how to write tests that improve the DDU. In other words, time spent on software debugging (using SBFL techniques) cannot be reduced using DDU because its practical implications are unclear to the developer.

1.2 Goal

Although DDU is currently not usable in practice, Perez *et al.* [12] have shown that optimizing a test suite with respect to DDU can yield a 34% gain in diagnostic performance using SBFL under constraint conditions. Having a test suite with a high diagnosability could possibly reduce the time spent debugging because the fault is easier to pinpoint using SBFL techniques. Therefore, the goal of this thesis is to explore how DDU behaves in practice and to find ways to make DDU usable in practice. In other words, we explore possibilities to convey DDU to the developer such that the developer knows what kind of tests to write to improve the system's diagnosability.

1.3 Structure of Report

The structure of this report is as follows. In Chapter 2, we explain the relevant topics to understanding this study, like spectrum-based fault localization, wasted effort, and DDU. In Chapter 3, we formalize the research questions and explain the motivation behind them. Then, in Chapters 4 to 6, we discuss the experimental design and results to answer the research questions. Finally, in Chapter 7, we conclude this study with a summary and recommendations for future work.

Chapter 2

Background

In this chapter, we discuss topics that are relevant to understanding this study. First, we discuss spectrum-based fault localization and spectrum-based reasoning, which are used in the experiments to perform fault diagnosis. Second, we discuss the metric used to evaluate the diagnostic performance of spectrum-based fault localization techniques. Then, we explain the definition of diagnosability and diagnosability assessment metrics. Finally, we discuss how DDU was evaluated in the study that proposed DDU.

2.1 Spectrum-Based Fault Localization (SBFL)

In spectrum-based fault localization, we define a finite set $\mathcal{C} = \langle c_1, c_2, \ldots, c_M \rangle$ of M system components, and a finite set $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N system transactions, i.e. test executions. The outcomes of all system tests are defined as an error vector $e = \langle e_1, e_2, \ldots, e_N \rangle$, where $e_i = 1$ indicates that test t_i has failed and $e_i = 0$ indicates that test t_i has passed. To keep track of which system components were executed during which test execution, we construct a $N \times M$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij} = 1$ indicates that component c_j was exercised during test t_i. The pair $(\mathcal{A}, e)$ is also known as a program spectrum, which was first coined by Reps *et al.* citereps1997use.

Given the program spectrum, SBFL techniques compute the suspiciousness scores of system components, resulting in a diagnostic report, which is a list of components ranked by their fault probability. The fault probabiliy is often computed using a similarity coefficient [8, 1, 10, 17, 19, 22, 18]. Intuitively, the similarity coefficient indicates the similarity between the component's activity and the error vector. When a component is more frequently exercised by test executions that fail, then the component is more likely to be faulty. Conversely, when a component is more frequently exercised by test executions that pass, then the component is more likely to be healthy.

2.2 Spectrum-Based Reasoning (SBR)

Spectrum-based reasoning distinguishes itself from SBFL techniques by leveraging a reasoning framework. The diagnostic report is generated by reasoning about the program spectrum instead of using a so-called similarity coefficient. The two main phases of SBR are candidate generation and candidate ranking:

1. In the candidate generation phase, a set $\mathcal{D} = \langle d_1, d_2, \ldots, d_k \rangle$ is constructed using a minimal hitting set (MHS) algorithm to cover all failing transactions, where each candidate d_i is a subset of $\mathcal{C}$. An MHS algorithm is used to prevent generating a large number of diagnostic candidates [3].

2. In the candidate ranking phase, the fault probability for each candidate d_i is computed using the Naive Bayes rule [3]:

$$P(d_i|(\mathcal{A},e)) = P(d_i) \cdot \prod_{j \in 1..N} \frac{P((\mathcal{A}_j, e_j)|d_i)}{P(\mathcal{A}_j)} \tag{2.1}$$

   $P(d_i)$ is the prior probability, i.e. the probability that d_i is faulty without any evidence. $P(\mathcal{A}_j)$ is a normalizing term that is identical for all candidates. $P((\mathcal{A}_j, e_j)|d_i)$ changes the prior probability with every new observation from the program spectrum. This term can be computed using maximum likelihood estimation.

In the experiments of this study, we make use of STACCATO [2] to generate candidates, and BARINEL [3], which implements spectrum-based reasoning, to rank candidates, i.e. perform software fault localization.

2.3 Evaluation of Diagnosis

Presently, cost of diagnosis C_d and wasted effort [3, 21, 6, 15, 12] are the most prevalent evaluation metrics for software fault localization (SFL) techniques. In essence, C_d computes the number of components that have to be inspected before the actual fault is investigated in the diagnostic report. When $C_d = 0$, it indicates that the actual fault is ranked first in the diagnostic report and, therefore, no effort is wasted investigating diagnosed components that are non-faulty. Wasted effort (or effort) is the cost of diagnosis normalized by the number of components in the diagnostic report.

Both evaluation metrics assume *perfect bug understanding*, which has been pointed out by Parnin and Orso [11] as a non-realistic assumption. However, cost of diagnosis and effort serve as an objective evaluation metric that can be used for comparison and therefore will also be used in this study.

2.4 Diagnosability

Diagnosability is the property of faults to be easily and precisely located [16]. In other words, given that a fault exists in a software system, if the test suite's diagnos-

ability is high and we would perform SFL using an automated debugging technique, then the faulty component would be ranked high in the diagnostic report, resulting in a low wasted effort. On the contrary, if the test suite's diagnosability is low and we would perform SFL using an automated debugging technique, then the faulty component would be ranked low in the diagnostic report, resulting in a high wasted effort.

2.5 Diagnosability Metric: Entropy

The optimal diagnosability is achieved by having an exhaustive test suite that would exercise any combination of software components. This way, any fault, whether it involves a single component or multiple components, can be diagnosed using an automated debugging technique with 100% accuracy. Perez *et al.* [12] find that Shannon's entropy accurately captures the test suite's exhaustiveness:

$$H(T) = -\sum_{i} P(t_i) \cdot \log_2(P(t_i)) \tag{2.2}$$

where T is the set of unique test activities, and $P(t_i)$ is the probability of test activity t_i occuring in the activity matrix. The optimal entropy for a system with M components is M shannons, and therefore we can compute the normalized entropy $\frac{H(T)}{M}$. SBFL techniques are able to diagnose faults with 100% accuracy when $\frac{H(T)}{M} = 1.0$.

However, an optimal normalized entropy would require $2^M - 1$ tests, which is difficult to achieve in practice. First, not all activity patterns can be generated from tests due to software topology, e.g. a basic block consisting of several statements — these statements will always be executed together, and therefore on a statement granularity it is not possible to achieve optimal entropy, see Listing 2.1. Second, systems of today can consist of millions of lines of code and would therefore require a non-realistic amount of effort to write the tests.

```
1  public FieldRotation(final T q0, final T q1, final T q2, final T q3, final
       boolean needsNormalization) {
2      if (needsNormalization) {
3          // normalization preprocessing
4          final T inv =
5                  q0.multiply(q0).add(q1.multiply(q1)).add(q2.multiply(q2)).
                       add(q3.multiply(q3)).sqrt().reciprocal();
6          this.q0 = inv.multiply(q0);
7          this.q1 = inv.multiply(q1);
8          this.q2 = inv.multiply(q2);
9          this.q3 = inv.multiply(q3);
10     } else {
11         this.q0 = q0;
12         this.q1 = q1;
13         this.q2 = q2;
14         this.q3 = q3;
15     }
16 }
```

Listing 2.1: Lines 4 - 9 will always be executed together and therefore no optimal entropy can be achieved on a statement granularity. Same holds for lines 11 - 14.

2.6 Diagnosability Metric: DDU

To elevate the problem with entropy, Perez *et al.* [12] propose a new diagnosability metric: DDU. DDU combines three diagnosability metrics that capture characteristics of the activity matrix, namely normalized density, diversity, and uniqueness.

2.6.1 Normalized Density

Prior work [7] has used density to assess the diagnosability of the activity matrix:

$$\rho = \frac{\sum_{i,j} \mathcal{A}_{ij}}{N \cdot M} \tag{2.3}$$

Gonzàles-Sanchez *et al.* [7] show by induction that the optimal density is obtained when $\rho = 0.5$. For DDU, Perez *et al.* [12] propose a normalized density ρ' where its optimal value is 1.0 instead of 0.5:

$$\rho' = 1 - |1 - 2\rho| \tag{2.4}$$

An optimal value for normalized density can be obtained without improving the diagnosability. For exampe, in Figure 2.1(a), we observe that four tests (t_1, t_2, t_3, t_4) have identical activation patterns, i.e. identical rows. Assuming that the tests are deterministic, if component c_1 is faulty, then all four tests will fail and therefore tests t_2, t_3, t_4 do not add any value from the diagnosability perspective, i.e. tests t_2, t_3, t_4 do not help SBFL techniques to further indict or exonerate components. For this reason, Perez *et al.* [12] propose two enhancements: diversity and uniqueness.

2.6.2 Diversity

The first enhancement to normalized density is diversity, which ensures test diversity among the activity matrix's rows. The test diversity is low when the test suite consists of many tests that have identical activity patterns, i.e. rows. Conversely, the test diversity is high when the test suite consists of many tests that have distinct activity patterns.

Perez *et al.* [12] use the Gini-Simpson index $\mathcal{G}$ [9] to capture test diversity:

$$\mathcal{G} = 1 - \frac{\sum_{i \in 1..|G|} |g_i| \cdot (|g_i| - 1)}{N \cdot (N-1)} \tag{2.5}$$

where $G = \langle g_1, \ldots, g_k \rangle$ is the set of ambiguity groups among the rows, $|g_i|$ is the number of test with identical activation patterns, and N is the number of tests. To clarify how this formula is used, we give two examples. In Figure 2.1(a), there is one ambiguity group $g_1 = \langle t_1, t_2, t_3, t_4 \rangle$, resulting in $\mathcal{G} = 1 - \frac{4 \cdot 3}{4 \cdot 3} = 0.0$. In Figure 2.1(b), there are four ambiguity groups, $g_1 = \langle t_1 \rangle, g_2 = \langle t_2 \rangle, g_3 = \langle t_3 \rangle, g_4 = \langle t_4 \rangle$, resulting in $\mathcal{G} = 1 - \frac{1 \cdot 0 + 1 \cdot 0 + 1 \cdot 0 + 1 \cdot 0}{4 \cdot 3} = 1.0$. As we can observe in Figure 2.1(b), when optimizing the activity matrix for test diversity, the shortcoming of normalized density, shown in Figure 2.1(a), is mitigated.

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

(a) No test diversity. $\rho' = 1.0$, $\mathcal{G} = 0.0$

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

(b) Component ambiguity. $\rho' = 1.0$, $\mathcal{G} = 1.0$, $\mathcal{U} = 0.75$

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

(c) No component ambiguity. $\rho' = 1.0$, $\mathcal{G} = 1.0$, $\mathcal{U} = 1.0$

Figure 2.1: The effect of diversity and uniqueness on diagnosability.

2.6.3 Uniqueness

The second enhancement to normalized density is uniqueness, which controls for the number of ambiguity groups among the activity matrix's columns. An ambiguity group is a set of components that have identical activation patterns across the test suite, i.e. identical columns in the activity matrix. Component ambiguity is undesirable because it prevents SBR from updating the fault probabilities of the individual components in the ambiguity group, resulting in a less accurate diagnosis. If test suite's uniqueness is low, then many components in the activity matrix have identical activity patterns, i.e. components are involved in the same test cases. Conversely, if the test suite's uniqueness is high, then many components in the activity matrix have distinct activity patterns.

Given the set of component ambiguity groups $G = \langle g_1, \ldots, g_l \rangle$, then the test suite's uniqueness is computed as follows:

$$\mathcal{U} = \frac{|G|}{M} \tag{2.6}$$

where $|G|$ is the number of ambiguity groups and M is the number of components, i.e. columns. To clarify how this formula is used, we give two examples. In Figure 2.1(b), there are 3 ambiguity groups: $g_1 = \langle c_1, c_2 \rangle$, $g_2 = \langle c_3 \rangle$, $g_3 = \langle c_4 \rangle$, resulting in $\mathcal{U} = \frac{3}{4} = 0.75$. In Figure 2.1(c), there are 4 ambiguity groups: $g_1 = \langle c_1 \rangle$, $g_2 = \langle c_2 \rangle$, $g_3 = \langle c_3 \rangle$, $g_4 = \langle c_4 \rangle$, resulting in $\mathcal{U} = \frac{4}{4} = 1$

We observe in Figure 2.1(b) that optimizing the test suite with respect to normalized density and diversity can still result in component ambiguity groups, namely $\langle c_1, c_2 \rangle$. When optimizing the test suite with respect to normalized density, diversity

and uniqueness, we observe in Figure 2.1(c) that there are no identical test activity patterns and no ambiguity groups, which results in a better diagnosability.

2.6.4 Combined

The DDU combines normalized density, diversity, and uniqueness as follows:

$$DDU = normalized\ density \cdot diversity \cdot uniqueness \tag{2.7}$$

If $DDU = 1$, then the test suite's diagnosability is high. Vice versa, if $DDU = 0$, then the test suite's diagnosability is low. Perez *et al.* [12] have shown in an experiment that optimizing a test suite with respect to DDU yields a 34% diagnostic performance compared to a test suite optimized for branch coverage.

2.7 Evaluation of DDU

In the study that proposed DDU, Perez *et al.* [12] performed two experiments to answer three 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 such as branch-coverage in traditional scenarios?

The goal of their study was to evaluate how DDU compares with the existing metrics: entropy, density, uniqueness, and branch coverage.

The empirical evaluation leveraged a test-generation approach using EVOSUITE[1], which employs search-based software testing approaches. EVOSUITE generates tests to minimize a fitness function using genetic algorithms. In essence, the fitness function is a measure that indicates the distance between a given solution and the optimal solution. For example, in the case of DDU, the optimal solution is $DDU = 1$ and, therefore, EVOSUITE generates tests to minimize the differene between the current test suite's DDU value and the optimal DDU value.

To answer **RQ1** and **RQ2**, the authors generated test suites for each class in the open source projects Commons Codec, Commons Compress, Commons Math and JodaTime that are optimized with respect to DDU, branch coverage, entropy, density and uniqueness. For each test suite, the activity matrix is constructed and, subsequently, the error vector is generated by artificially injecting faults — single-component and two-component faults. Then, given these program spectra, on each program spectrum SBR is performend to generate a diagnostic report. Finally, the

[1] `http://www.evosuite.org/`

wasted effort is computed for each diagnostic report, which enables the authors to compare the diagnostic performance of DDU against the diagnostic performance of branch coverage, entropy, density, uniqueness.

To answer **RQ3**, the authors make use of the DEFECTS4J database, which contains 357 real software bugs from 5 open source projects. The database contains a faulty and fixed software version for each bug. For each bug, the authors generate two test suites for the fixed software version — one test suite that optimizes DDU and one that optimizes branch coverage. Then, instead of artificially injecting faults, the authors run the generated test suite against the faulty software version and collect the program spectrum. Subsequently, SBR is performed on the collected program spectra, enabling the comparison between DDU and branch coverage.

Chapter 3

Research Questions

Ideally, we would like to propose an intuitive approach that allows developers to use DDU in their software development cycle. However, there has been no study yet on DDU that investigates how it actually behaves in practice. For this reason, the goal of this study is to explore how DDU behaves in practice and to find ways to make DDU usable in practice. To achieve this goal, we define three research questions.

RQ1: How do normalized density, diversity, uniqueness, and DDU vary in practice?

To be able to propose an intuitive approach that allows developers to use DDU in practie, we first need to understand how the metrics vary in practice. By analyzing how these metrics perform in practice, we are able to propose improvements for DDU, and potentially propose an approach that enables DDU in practice.

RQ2: What is the relation between normalized density, diversity, uniqueness, and DDU and diagnosability?

In Perez *et al.*'s work [12], the authors show that generating tests with respect to DDU yields a 34% diagnostic performance compared to a test suite optimized for branch coverage. However, this does necessarily imply that DDU is strongly correlated with diagnosability. Therefore, we would like to validate that DDU and diagnosability are strongly correlated, i.e. the higher DDU, the better the diagnosability, and vice versa. This question is important to answer because DDU was proposed as a metric to quantify the diagnosability.

RQ3: What is the relation between density, diversity, uniqueness, and DDU and test coverage?

The intention of test coverage is to optimize for error detection. Perez *et al.* [12] propose DDU as a complementary metric to test coverage because DDU is meant to capture the diagnosability and not error detection. However, if there is a strong correlation between DDU and test coverage, then DDU could possibly replace test coverage as a test adequacy metric, which is a use case that the authors have not thought of. Furthermore, assuming that DDU is strongly correlated with diagnosability and test coverage is representative for error detection, answering this research question will give us a better understanding on the relation between diagnosability and error detection, and could give us insight in how DDU and test coverage can be used together in practice.

Chapter 4

DDU in Practice

RQ1: How do normalized density, diversity, uniqueness, and DDU vary in practice?

In this chapter, the goal is to obtain a better intuition on what common values are for density, diversity, uniquness, and DDU by analyzing open source projects hosted on GitHub. First, we take a look at the distributions of normalized density, diversity, uniqueness, and DDU. Second, we give examples of classes that have a low or high value for any of the diagnosability metrics and analyze why these components have either a low or high value by investigating the class' test suite. Finally, we conclude this chapter with a list of observations.

4.1 Approach

To get a better understanding of how the values vary for normalized density, diversity, uniqueness, and DDU, we use `ddu-maven-plugin`[1], developed by Perez, to instrument Java code and construct the activity matrix. Once we obtain the activity matrices, we analyze the data using multiple Python scripts[2]. With these two tools we collect data such as density, normalized density, diversity, uniqueness, DDU, and the activity matrix.

Then, we analyze the collected data and show examples to illustrate how DDU and its individual terms vary as a consequence to particular kinds of tests or testing strategies. We are interested in what kinds of testing approaches result in a high or low value.

Note that `ddu-maven-plugin` is able to instrument the code for three different granularity levels, namely statements, branches, and methods. By default `ddu-maven-plugin` uses the method level granularity. In this study, we make use of the same

[1] `https://github.com/aperez/ddu-maven-plugin`

[2] `https://github.com/aaronang/ddu`

granularity used in the study performed by Perez *et al.* [12], namely branch granularity.

4.2 Selection

The selection of open source projects is done according to the following criteria.

- *The project must have an executable test suite.* To compute the DDU for a software project, we must construct an activity matrix. The activity matrix is constructed by running the test suite and instrumenting the code to keep track of which components are executed during a program execution. Since we are interested in the activity matrix $\mathcal{A}$, we do not care if a test suite has failing tests. A project meets this criterion as long as the test suite executes and does not crash.
- *The project must use Apache Maven, a software project management and comprehension tool.* The current tool that instruments the code to construct the activity matrix is implemented as a Maven plugin. Note that the current Maven plugin does not work for all Maven projects and, therefore, only projects that can be analyzed with this plugin are used.

Based on these requirements, we choose the following open source projects.

- Commons Codec[3]: a package that contains simple encoders and decoders for various formats such as Base64 and Hexadecimal.
- Commons Compress[4]: an API for working with compression and archive formats.
- Commons Math[5]: a library of lightweight, self-contained mathematics and statistics components addressing the most common practical problems.
- Guice[6]: a lightweight dependency injection framework for Java 6 and above.
- Jsoup[7]: an API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods.

Note that some of the projects are also used in Perez *et al.*'s study [12]. In Table 4.1, we show some statistics for each project. The number of classes, methods, and lines were obtained by running the complete test suites in IntelliJ and collecting the coverage report. The number of branches was obtained by running the complete test suite while the DDU-MAVEN-PLUGIN instruments the code using the branch granularity and consequently counting the number of components in the program spectrum.

[3] https://github.com/apache/commons-codec
[4] https://github.com/apache/commons-compress
[5] https://github.com/apache/commons-math
[6] https://github.com/google/guice
[7] https://github.com/jhy/jsoup

Table 4.1: Statistics of open source projects.

Subject	Classes	Methods	Branches	Lines
Commons Codec	87	743	2901	3827
Commons Compress	261	2143	8650	12182
Commons Math	977	7406	33614	45004
Guice	332	2039	9134	7102
Jsoup	232	1343	5088	6875

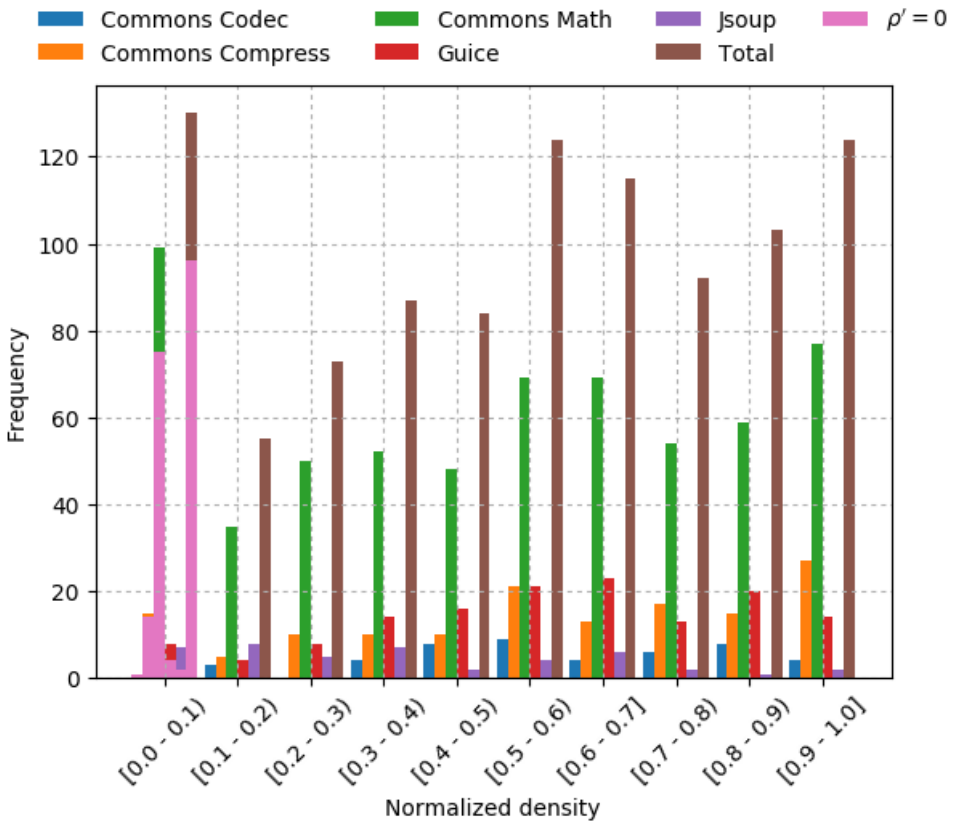


Figure 4.1: Normalized density distribution.

4.3 Normalized Density

In Figure 4.1, we show the distribution of normalized densities for all classes of the five open source projects mentioned before. The average equals to 0.5145. The peak for the interval [0, 0.1) is primarily caused by classes that are exercised by tests that involve all components. In the interval [0, 0.1), 43 classes consist of only one branch. A class with one branch will always have a density of 1.0 and therefore a normalized density of 0.

Table 4.2: Partial activity matrix of the `com.google.inject.internal.ProvidesMethodScanner` class where $\rho' = 0.111$.

transaction	c_1	...	c_{22}	c_{23}	c_{24}	...	c_{35}
`BinderTest#testUntargettedBinding`	0	...	0	1	0	...	0
`BinderTest#testMissingDependency`	0	...	0	1	0	...	0
`BinderTest#testProviderFromBinder`	0	...	0	1	0	...	0
`BinderTest#testToStringOnBinderApi`	0	...	0	1	0	...	0
`BinderTest#testUserReportedError`	0	...	0	1	0	...	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.3: Activity matrix of the `org.apache.commons.math4.fitting.leastsquares.CircleProblem` class where $\rho' = 0$ because every transaction covers every component.

transaction	c_1	c_2	...	c_{26}
`LevenbergMarquardtOptimizerTest#testParameterValidator`	1	1	...	1
`LevenbergMarquardtOptimizerTest#testCircleFitting2`	1	1	...	1

The normalized density value is low when a class is tested by many tests that only cover a couple of branches, or covered by tests that involve all components. For example, the class `ProvidesMethodScanner` has 35 branches and its partial activity matrix is shown in Table 4.2, where the columns represent branches of `ProvidesMethodScanner`. In the first row, we observe that the transaction `testUntargettedBinding` only hits one branch c_{23}, indicated by a 1. We observe that all transactions in the activity matrix have a similar coverage. Since every transaction is only hitting a few components, the normalized density is low. Moreover, `ProvidesMethodScanner` has 35 components and if a transaction only hits a few components, it results in the activity matrix to become sparse.

In the activity matrix of `CircleProblem`, shown in Table 4.3, we observe a high density; all branches of `CircleProblem` are hit in every single test. This results in a low normalized density: 0.0, because the density is high: 1.0.

Thus, ideally, to obtain a high value for the normalized density, we need a good balance between tests that cover many components and tests that cover a few. In Commons Math, the `ElitisticListPopulation` class has a nearly optimal normalized density. In its activity matrix, see Table 4.4, we observe that there is a good balance between tests that cover a few components and tests that cover many components, resulting in a normalized density of 0.979.

4.4 Diversity

In Figure 4.2, the distribution of diversity of classes is shown. The average is 0.588. The peak for the interval [0, 0.1) occurs for various reasons. The first reason is that there are 43 classes with only one method and therefore every row is identical, resulting in a diversity of 0. The second reason is that for 52 classes there exists only

Table 4.4: Activity matrix of `org.apache.commons.math4.genetics.ElitisticListPopulation` where $\rho' = 0.979$.

transaction	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
`ElitisticListPopulationTest#testChromosomeListConstructorTooLow`	1	0	0	1	0	0	0	0
`ElitisticListPopulationTest#testSetElitismRateTooLow`	1	0	0	1	0	0	1	0
`ElitisticListPopulationTest#testConstructorTooHigh`	1	0	0	1	0	0	0	0
`ElitisticListPopulationTest#testConstructorTooLow`	1	0	0	1	0	0	0	0
`ElitisticListPopulationTest#testSetElitismRateTooHigh`	1	0	0	1	0	0	1	0
`ElitisticListPopulationTest#testChromosomeListConstructorTooHigh`	1	0	0	1	0	0	0	0
`ElitisticListPopulationTest#testSetElitismRate`	1	0	0	0	0	1	1	0
`ElitisticListPopulationTest#testNextGeneration`	1	1	1	0	1	1	1	1
`FitnessCachingTest#testFitnessCaching`	1	1	1	0	1	1	1	1
`GeneticAlgorithmTestBinary#test`	1	1	1	0	1	1	1	1
`GeneticAlgorithmTestPermutations#test`	1	1	1	0	1	1	1	1
`TournamentSelectionTest#testSelect`	1	0	0	0	0	0	1	0

Table 4.5: Partial activity matrix of `org.apache.commons.math4.analysis.function.Power` where $\mathcal{G} = 0.077$.

transaction	c_1	c_2	c_3
`FunctionUtilsTest#testFixingArguments`	1	1	0
`FunctionUtilsTest#testMultiplyDifferentiable`	1	0	1
`FunctionUtilsTest#testComposeDifferentiable`	1	1	1
`FunctionUtilsTest#testCompose`	1	1	0
`FunctionUtilsTest#testMultiply`	1	1	0
`ArrayRealVectorTest#testMap`	1	1	0
`ArrayRealVectorTest#testMapToSelf`	1	1	0
`RealVectorTest#testMap`	1	1	0
⋮	⋮	⋮	⋮

one test, and in the current Python script the diversity defaults to 0 when there is only one test. The third reason is that there are test suites where all the test cases have identical activity patterns. In total there are 176 classes with a diversity of 0, where 36 classes only have one method, 45 classes only have one test, and 7 classes only have one method and one test.

Intuitively, the diversity has a low value when the number of identical transactions is high, i.e. identical rows in the activity matrix. Conversely, the diversity is high when the number of identical transactions is low.

In the partial activity matrix of `Power`, shown in Table 4.5, we observe that almost every transaction has an identical activity and therefore the diversity is low: 0.077. Another reason for the low diversity of `Power` is that it is covered by 102 test cases, while there are only $2^3 - 1 = 7$ possible different tests for two components. After 7 unique tests every additional test will have a negative effect on the diversity because it will share an identical activity with an existing test.

In the activity matrix of `AbstractParameterizable`, shown in Table 4.6, we observe that almost every transaction has a unique activity and therefore its diversity is high: 0.933. Note that the diversity suffers when there are too many test cases, but does not suffer from a low number of test cases.

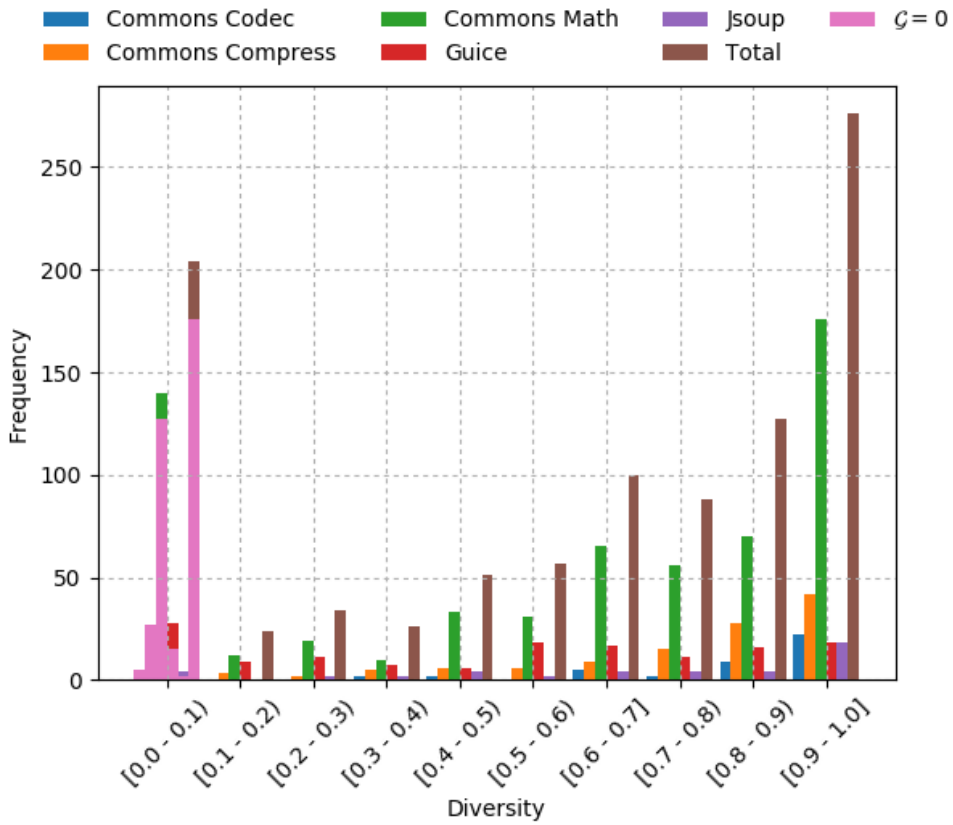


Figure 4.2: Diversity distribution.

Table 4.6: Activity matrix of `org.apache.commons.math4.ode.AbstractParameterizable` where $\mathcal{G} = 0.933$.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
`JacobianMatricesTest#testHighAccuracyExternalDifferentiation`	1	0	1	0	1	0	1	0
`JacobianMatricesTest#testAnalyticalDifferentiation`	1	0	1	0	1	0	1	0
`JacobianMatricesTest#testInternalDifferentiation`	0	0	0	0	1	0	1	0
`JacobianMatricesTest#testParameterizable`	1	0	0	0	1	0	1	1
`JacobianMatricesTest#testWrongParameterName`	0	0	1	1	1	1	0	1
`JacobianMatricesTest#testFinalResult`	1	0	1	0	1	0	1	1

Table 4.7: Partial activity matrix of `org.apache.commons.math4.analysis.integration.gauss.SymmetricGaussIntegrator`, where parameterized testing exhibits identical activity patterns.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8
`HermiteParametricTest#testAllMonomials[0]`	0	1	1	0	0	0	0	0
`HermiteParametricTest#testAllMonomials[1]`	1	0	1	1	1	1	1	0
`HermiteParametricTest#testAllMonomials[2]`	1	0	1	1	1	1	1	1
`HermiteParametricTest#testAllMonomials[3]`	1	0	1	1	1	1	1	0
`HermiteParametricTest#testAllMonomials[4]`	1	0	1	1	1	1	1	1
`HermiteParametricTest#testAllMonomials[5]`	1	0	1	1	1	1	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
`HermiteParametricTest#testAllMonomials[26]`	1	0	1	1	1	1	1	1
`HermiteParametricTest#testAllMonomials[27]`	1	0	1	1	1	1	1	0
`HermiteParametricTest#testAllMonomials[28]`	1	0	1	1	1	1	1	1
`HermiteParametricTest#testAllMonomials[29]`	1	0	1	1	1	1	1	0

Table 4.8: Activity matrix of `org.apache.commons.codec.digest.Crypt` where $\mathcal{U} = 0.916$.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}
`CryptTest#testDefaultCryptVariant`	0	0	0	1	1	0	0	1	0	1	0	0
`CryptTest#testCryptWithEmptySalt`	1	1	0	0	0	0	1	1	0	1	1	0
`CryptTest#testCryptWithBytes`	0	1	1	1	0	1	0	1	0	1	0	0
`Md5CryptTest#testMd5CryptBytes`	0	1	0	0	0	0	1	1	1	1	1	0
`Md5CryptTest#testMd5CryptLongInput`	0	1	0	0	0	0	1	1	1	1	1	0
`Md5CryptTest#testMd5CryptStrings`	0	1	0	0	0	0	1	1	1	1	1	0
`Sha256CryptTest#testSha256CryptBytes`	0	1	0	0	0	0	0	1	0	1	1	1
`Sha256CryptTest#testSha256CryptStrings`	0	1	0	0	0	0	0	1	0	1	1	1
`Sha512CryptTest#testSha512CryptBytes`	0	1	1	0	0	0	0	1	0	1	0	0
`Sha512CryptTest#testSha512CryptStrings`	0	1	1	0	0	0	0	1	0	1	0	0
`UnixCryptTest#testUnixCryptStrings`	1	1	0	0	0	0	1	1	0	1	1	0
`UnixCryptTest#testUnixCryptBytes`	1	1	0	0	0	0	1	1	0	1	1	0

An interesting case for diversity is parameterized testing. Although parameterized testing is a common practice to test different inputs for a unit, it can have a negative effect on the diversity if the various parameters chosen exhibit identical activity patterns. An example is the class `SymmetricGaussIntegrator` with a diversity of 0.571, shown in Table 4.7.

4.5 Uniqueness

The distribution of uniqueness of classes is shown in Figure 4.3. The average is 0.477. The peak for the interval [0.9, 1.0] is partially caused by classes that only have one component; activity matrices that consist of one component always have a uniqueness of 1.0. There are 130 classes that have a uniqueness of 1.0 and 43 out of the 130 classes only have one branch.

The uniqueness of a class is high when there is a high number of unique columns in the activity matrix. Conversely, the uniqueness is low when there is a low number of unique columns in the activity matrix.

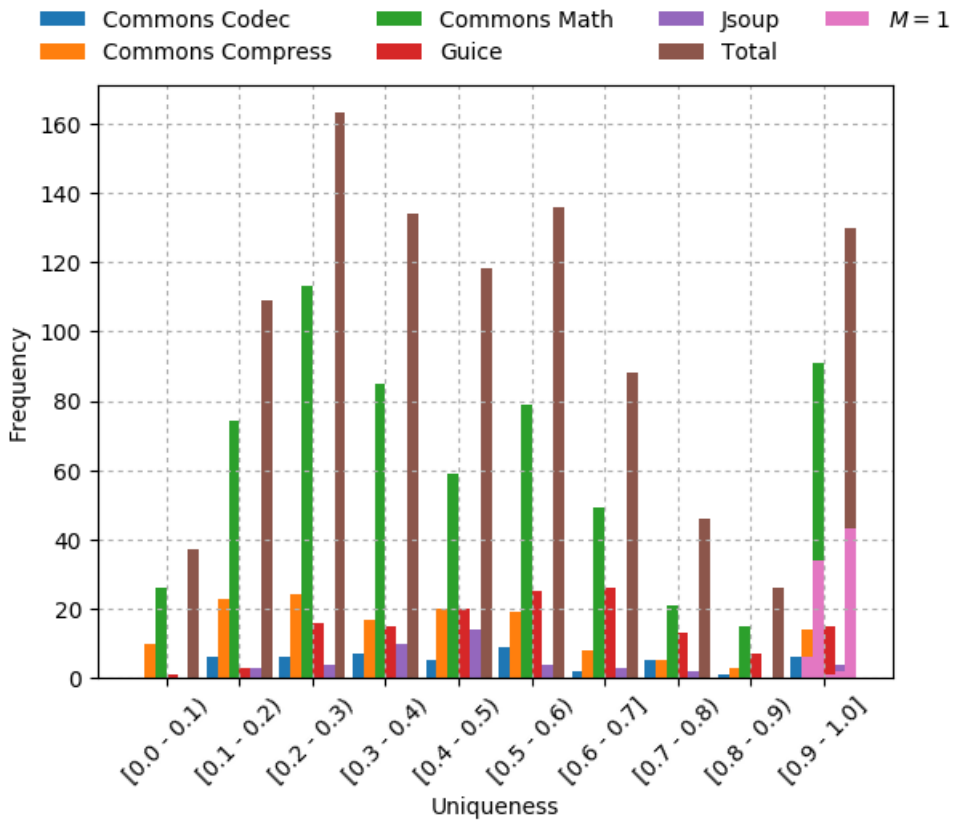


Figure 4.3: Distribution of uniqueness.

Table 4.9: Partial activity matrix of `org.apache.commons.math4.random.UnitSphereRandomVectorGenerator` where $\mathcal{U} = 0.142$.

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
`MicrosphereProjectionInterpolatorTest#testLinearFunction2D`	1	1	1	1	1	1	1
`FieldRotationDfpTest#testDoubleVectors`	1	1	1	1	1	1	1
`FieldRotationDfpTest#testDoubleRotations`	1	1	1	1	1	1	1
`FieldRotationDSTest#testDoubleVectors`	1	1	1	1	1	1	1
`FieldRotationDSTest#testDoubleRotations`	1	1	1	1	1	1	1
`SphereGeneratorTest#testRandom`	1	1	1	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

An example of a class with a high uniqueness is the `Crypt` class of Commons Codec, see Table 4.8. The `Crypt` class has a uniqueness of 0.916 because it only has one ambiguity group $\langle c_8, c_{10} \rangle$.

In the activity matrix of `UnitSphereRandomVectorGenerator`, see Table 4.9, we observe that all components have identical activity patterns and, therefore, the uniqueness is low: 0.142. Note that the uniqueness does not equal zero although

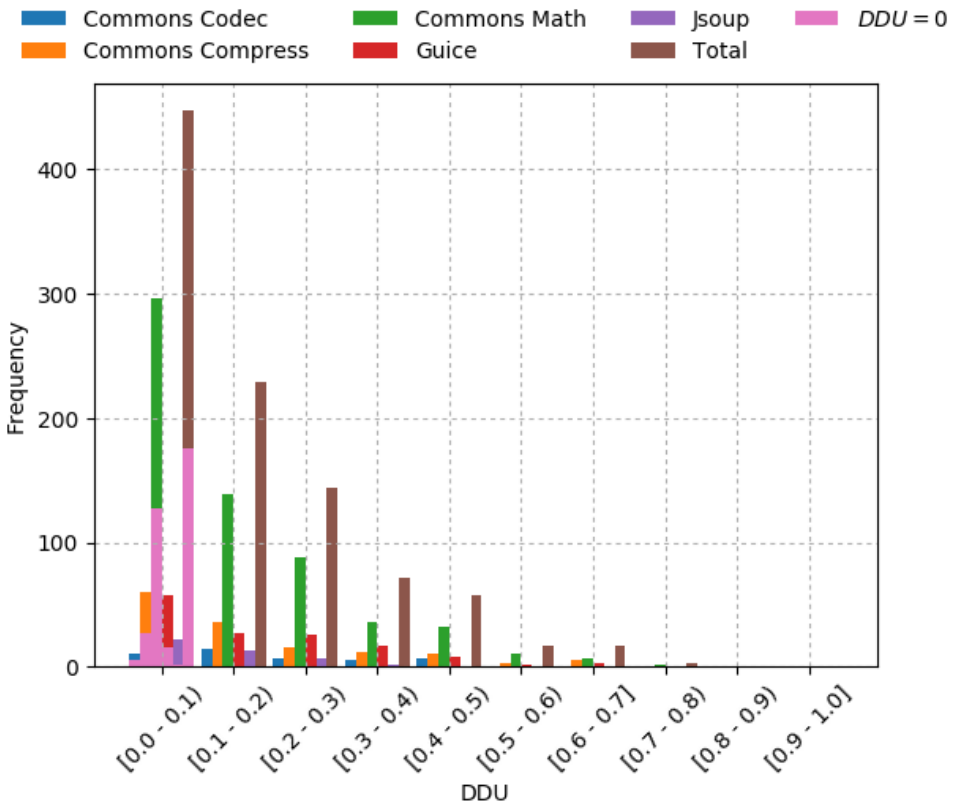


Figure 4.4: DDU distribution.

there is no component with a unique activity.

4.6 DDU

The distribution of DDU of classes is shown in Figure 4.4. The average is 0.157. We observe that 176 out of 987 classes have a DDU of zero due to either normalized density, diversity, or uniqueness being equal to zero. Additionally, we observe a right-tailed distribution due to the nature of how DDU is computed, a product of three metrics within the domain [0, 1].

In Table 4.10, we observe a class with a high DDU: 0.809. Its DDU is high because it has an optimal normalized density, uniqueness, and an almost optimal diversity.

In Table 4.11, we observe a class with a low DDU: 0.082. Eventhough the normalized density and uniqueness are above average, the DDU is low due to the low diversity.

Table 4.10: Activity matrix of `org.apache.commons.math4.analysis.function.Min`, $\rho' = 1.0$, $\mathcal{G} = 0.809$, $\mathcal{U} = 1.0$, and $DDU = 0.809$.

transaction	c_1	c_2	c_3	c_4
`UnivariateDifferentiableFunctionTest#testMinus`	1	0	1	1
`FunctionUtilsTest#testCollector`	0	1	0	0
`FunctionUtilsTest#testAdd`	1	0	1	0
`FunctionUtilsTest#testComposeDifferentiable`	0	0	1	1
`FunctionUtilsTest#testCombine`	1	0	1	0
`FunctionUtilsTest#testCompose`	1	0	1	0
`FunctionUtilsTest#testAddDifferentiable`	0	0	1	1

Table 4.11: Partial activity matrix of `org.jsoup.parser.ParseSettings`, $\rho' = 0.727$, $\mathcal{G} = 0.204$, $\mathcal{U} = 0.555$, and $DDU = 0.082$.

transaction	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
`ParseTest#testBaidu`	0	1	1	0	0	1	1	1	1
`AttributesTest#html`	0	0	1	0	0	0	1	1	0
`HtmlParserTest#canPreserveAttributeCase`	0	1	1	0	0	0	1	1	1
`HtmlParserTest#handlesBaseTags`	0	1	1	0	0	1	1	1	1
`SelectorTest#descendant`	0	1	1	0	0	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

4.7 Observations

In this section, we summarize our findings with regards to normalized density, diversity, uniqueness, and DDU.

The optimal normalized density is 1.0 and the optimal density is 0.5 for a test suite. It is difficult to utilize normalized density to recommend the developer what kind of test to write — a test that covers a few components or a test that covers many components. For example, if $\rho' = 0.6$, then the density can either be $\rho = 0.3$ or $\rho = 0.7$. Therefore, the density can be used to guide the developer in writing tests. For example, when the test suite's density is less than 0.5, it is recommended to write tests that cover many components. Conversely, when the test suite's density is greater than 0.5, it is recommended to write tests that cover a few components.

In practice, we write tests to account for many corner cases. From the diagnostic perspective, adding tests, that do not improve the information gain, is useless. For example, the diversity can be negatively impacted when we write parameterized tests with certain inputs that cause identical activation patterns. Another example is property-based testing, where the outputs are checked against a so-called *property* that should hold true given arbitrary inputs that meet certain criteria. Therefore, we pose the question whether we need a metric that penalizes tests that have identical component coverage.

We observe in Figure 4.4, that the DDU distribution is right-tailed. This can be explained by the fact that the DDU is the product of normalized density, diversity, and uniqueness, which all operate in the domain [0, 1]. Even when a test suite has a high normalized density and diversity but a low uniqueness, then the value for DDU

can only be as high as the uniqueness. In other words, the DDU can only be as high as the term with the lowest value. A possible solution to the right-tailed distribution is to use geometric mean.

Revisiting the first research question:

> **RQ1:** How do normalized density, diversity, uniqueness, and DDU vary in practice?

A: We observed that the average for normalized density, diversity, and uniqueness are 0.5145, 0.588, 0.477, respectively. However, the values for DDU have an average of 0.157. In addition, the distribution of DDU is right-tailed which could be explained by the fact that DDU can only be as high as the lowest component — normalized density, diversity, or uniqueness.

Chapter 5

DDU vs. Diagnosability

RQ2: What is the relation between density, diversity, uniqueness, and DDU and diagnosability?

In prior work, Perez *et al.* [12] show that optimizing test suite generation with respect to DDU results in better fault diagnosis. Optimizing test suite generation with respect to DDU yields a 34% diagnostic performance compared to a test suite optimized for branch coverage. Therefore, in this chapter, we perform experiments to verify the correlation between DDU and diagnosability.

5.1 Experimental Setup

To verify the correlation between DDU and diagnosability, we generate 10 artificial multiple components faults of cardinality 2 — that is, faults that are caused by two components — for each class that has at least 8 components, i.e. branches. We do not generate single component faults because in this case the optimal activity matrix for diagnosability is an identity matrix, i.e. each component is tested individually by a test.

For each injected fault, we generate an error vector corresponding to the activity matrix and the fault. We determine for each test that exercises the faulty components whether it is failing according to an *oracle quality probability* of 0.75, which was also used in prior work [12]. Note that we only generate faults that result in at least one failing test in the error vector because in this experiment we are interested in diagnosability and not error detection. For some classes, it is not possible to generate 10 faults that result in a failing test in the error vector and, therefore, we do not include these classes in the experiment.

Then, for each generated program spectrum, we use STACCATO to generate fault candidates and BARINEL to rank these candidates into a diagnostic report, see Section 2.2 for a more detailed explanation. Based on the diagnostic report we compute

Table 5.1: Classes that were not included in the analysis due to STACCATO taking longer than 10 seconds.

Class	Number of tests	Number of branches
com.google.inject.AbstractModule	643	27
com.google.inject.internal.InjectorImpl	601	223
com.google.inject.spi.Elements	706	109
org.apache.commons.math4.analysis.differentiation.DSCompiler	278	338
org.apache.commons.math4.linear.AbstractFieldMatrix	140	295
org.apache.commons.math4.linear.AbstractRealMatrix	559	262
org.apache.commons.math4.linear.BlockRealMatrix	221	374
org.apache.commons.math4.linear.QRDecomposition	108	101
org.apache.commons.math4.util.Decimal64	222	147
org.apache.commons.math4.util.FastMath	3646	702
org.apache.commons.math4.util.MathArrays	1261	306
org.jsoup.nodes.Element	426	233
org.jsoup.parser.HtmlTreeBuilder	406	941
org.jsoup.parser.HtmlTreeBuilderState	392	673
org.jsoup.parser.TokeniserState	406	439

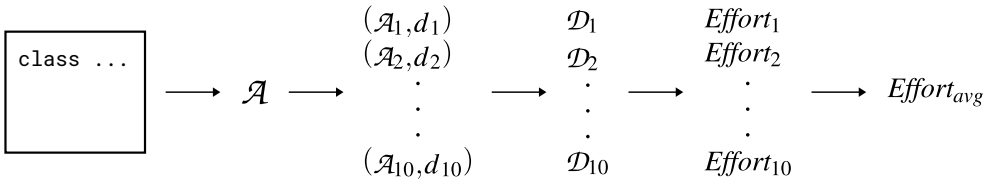


Figure 5.1: An activity matrix $\mathcal{A}$ is constructed from a particular class. Then, 10 fault candidates of cardinality 2 are generated with a corresponding activity matrix $\mathcal{A}_k$. For each generated matrix, we perform fault diagnosis with BARINEL resulting in diagnostic report $\mathcal{D}_k$ and compute the wasted effort. Finally, we compute the average wasted effort. This process is repeated 10 times.

the wasted effort which is a measurement for diagnosability. This whole process of constructing an activity matrix, injecting artificial faults, performing SFL, and diagnosis evaluation is shown in Figure 5.1.

To account for randomness of generating fault sets, we repeat this process 10 times. Additionally, STACCATO can sometimes take hours or days to generate fault candidates. Hence, we discard classes when the generation of fault candidates takes longer than 10 seconds; this resulted in 15 classes being discarded. In Table 5.1, we observe that the discarded classes are covered by a relative big number of tests and consist of many branches, potentially causing STACCATO's computation to take longer than 10 seconds.

In the construction of the activity matrix we use the branch granularity, that is,

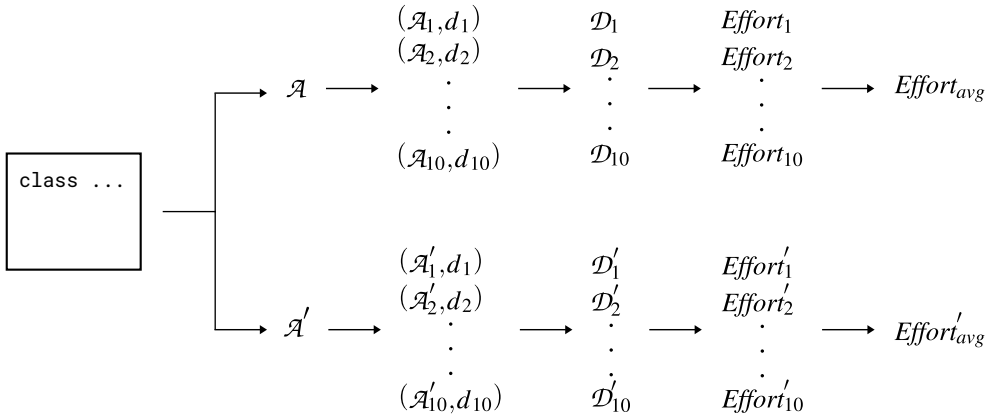


Figure 5.2: Two activity matrices $\mathcal{A}$ and $\mathcal{A}'$ are generated for a particular class based on two different test suites. We generate 10 fault candidates of cardinality 2 and accordingly generate 10 activity matrices. Then, we use BARINEL to perform fault diagnosis and compute the wasted effort.

every column in the activity matrix represents a method branch. This granularity is also used by Perez *et al.* [12]. To construct the activity matrix of a class we use Perez' DDU Maven plugin[1] using the `basicblock` granularity, which represents branch granularity. The steps after obtaining the activity matrix in Figure 5.1 are performed using Python scripts[2].

This experiment is different from Perez *et al.*'s work because we do not improve the DDU of a fixed system. In Perez *et al.*'s study, the authors improved the DDU of a fixed system under test by generating tests using *EvoSuite*. However, in this experiment, we compute the DDU for each class and measure for each class its diagnosability using the aforementioned approach.

For this reason, we perform another experiment where we generate two test suites for each class with at least 8 components and at least 10 tests. We generate two test suites by enabling the first 50% of the tests and 100% of the tests. For both test suites we compute the DDU and randomly generate 10 multiple components faults of cardinality 2 to compute the wasted effort. Similar to previous experiment we perform this process 10 times to account for randomness of generating fault sets. The intuition behind this experiment is when we improve the DDU of a fixed system, its diagnosability should improve too. The setup of this experiment is illustrated in Figure 5.2.

In the experiments, we compute the wasted effort using the following approach. Given a diagnostic report where the candidates are ranked by their fault probabilities in descending order, shown in Table 5.2, and the injected fault set: $\langle c_0, c_1 \rangle$, the wasted effort is computed by, first, flattening the ranked list to single components and recomputing the probabilities for each component using addition, e.g. fault probability of

[1] `https://github.com/aperez/ddu-maven-plugin`

[2] `https://github.com/aaronang/ddu`

Table 5.2: Computation of wasted effort.

(a) Diagnostic report computed by BARINEL.

candidate	fault probability
$\langle c_0, c_1 \rangle$	0.5
$\langle c_2 \rangle$	0.3
$\langle c_3 \rangle$	0.15
$\langle c_0 \rangle$	0.05

(b) Flatten candidates and rerank.

candidate	fault probability
c_0	0.55
c_1	0.5
c_2	0.3
c_3	0.15

$\langle c_0 \rangle = 0.5 + 0.05 = 0.55$. Then, the wasted effort is computed for each component in the generated fault set: $\mathit{effort}_{c_0} = \frac{0}{4} = 0$, $\mathit{effort}_{c_1} = \frac{1}{4} = 0.25$. Finally, the wasted effort of the generated multi-component fault is computed by averaging the computed wasted efforts: $\frac{0+0.25}{2} = 0.125$. This approach is used because of its simplicity. It's not always guaranteed that the multi-component fault is found as a candidate in the diagnostic report. For example, there are cases that STACCATO does not generate c_0 and c_1 as one candidate set but as separate candidates.

5.2 Experimental Results

In the first experiment, we measure for each class the normalized density, diversity, uniqueness, DDU, and effort. The results of this experiment are shown in Figure 5.3. In Figure 5.3, the population comprises all classes of all projects. Each datapoint in Figure 5.3 represents a class for which 100 fault candidates are generated in (potentially overlapping) sets of 10 fault candidates as described in Figure 5.1.

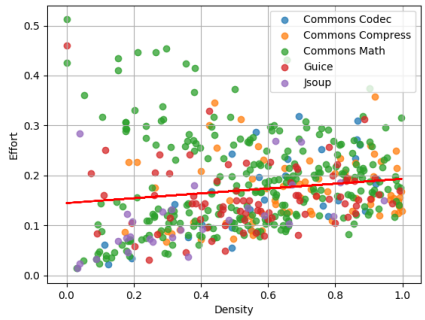
Using a significance level of 5%, we observe the that there is a weak positive correlation between density and effort, a weak negative correlation between diversity and effort, a weak negative correlation between uniqueness and effort, and a statistically non-significant weak correlation between DDU and effort.

To investigate the relations between these metrics in more detail, we display the correlation values *per project* in Table 5.3. For three projects we can say with 95% confidence that normalized density is correlated with effort. However, the Pearson correlation for Commons Compress is negative while the Pearson correlation values for Commons Math and Commons Codec are positive. Hence, the results show no strong evidence that density is strongly correlated with effort.

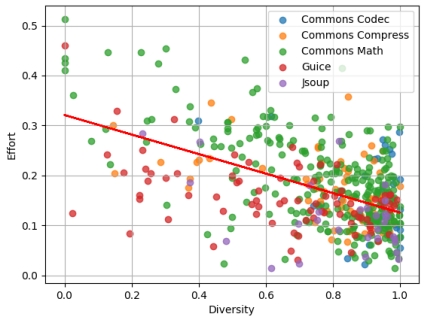
Regarding diversity and uniqueness, we observe in Table 5.3 and Figure 5.3 that both metrics have a weak negative correlation with effort, and that the correlation values in 4 out of 5 projects are statistically significant.

Regarding DDU, for two projects the results show statistical significance that DDU has a weak negative correlation with effort. However, for three projects there is no evidence that DDU is correlated to effort. Therefore, there is no strong evidence that DDU is negatively correlated to effort.

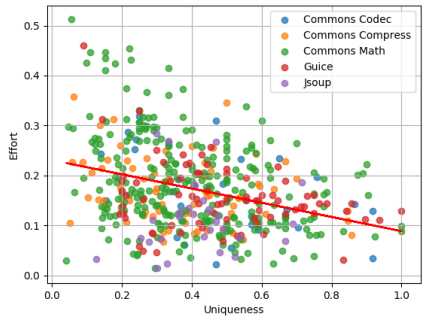
It is unexpected that the normalized density tends to be positively correlated to wasted effort. The normalized density is a diagnosability assessment metric and is supposed to be negatively correlated to wasted effort, that is, the higher the nor-



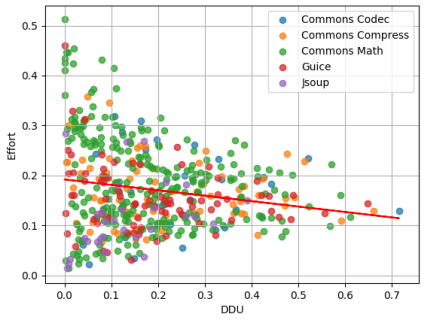
(a) Normalized density, $r = 0.183$, $p < 0.001$.



(b) Diversity, $r = -0.326$, $p < 0.001$.



(c) Uniqueness, $r = -0.120$, $p < 0.001$.



(d) DDU, $r = -0.046$, $p = 0.224$.

Figure 5.3: Scatterplot of density, diversity, uniqueness, and DDU against effort.

Table 5.3: Correlations between density, diversity, uniqueness, DDU, and effort for each project.

Subject	Number of classes	Pearson correlation / Correlation p-value			
		Density	Diversity	Uniqueness	DDU
Commons Codec	34	**0.63** 5.880×10^{-5}	-0.33 0.057	**-0.65** 3.713×10^{-5}	-0.23 0.197
Commons Compress	104	**-0.22** 0.027	**-0.45** 1.348×10^{-6}	**-0.40** 2.083×10^{-5}	**-0.37** 1.121×10^{-4}
Commons Math	420	**0.20** 4.982×10^{-5}	**-0.36** 1.782×10^{-14}	**-0.19** 7.553×10^{-5}	-0.03 0.572
Guice	94	0.01 0.935	**-0.31** 0.002	**-0.29** 0.005	**-0.22** 0.031
Jsoup	37	0.29 0.085	**-0.37** 0.024	0.16 0.337	0.20 0.229

Table 5.4: Partial activity matrix of `com.google.inject.internal.AbstractProcessor`, $\rho = 1$, $\rho' = 0$.

transaction	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}
com.google.inject.BinderTest#testUntargettedBinding	1	1	1	1	1	1	1	1	1	1	1
com.google.inject.BinderTest#testMissingDependency	1	1	1	1	1	1	1	1	1	1	1
com.google.inject.BinderTest#testProviderFromBinder	1	1	1	1	1	1	1	1	1	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
com.googlecode.guice.Jsr330Test#testInjecting...	1	1	1	1	1	1	1	1	1	1	1
com.googlecode.guice.Jsr330Test#testScopeAnnotation	1	1	1	1	1	1	1	1	1	1	1
com.googlecode.guice.Jsr330Test#testInject	1	1	1	1	1	1	1	1	1	1	1

Table 5.5: Partial activity matrix of `org.apache.commons.math4.ode.nonstiff.DormandPrince853FieldStepInterpolator`, $\rho' = 902$.

transaction	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}
testModelsMerging	1	1	1	0	1	1	1	1	0	1	1	1	1	1
testStartFailure	0	0	0	0	0	0	1	0	0	0	0	0	0	0
testEvents	1	1	1	1	1	1	1	1	1	1	1	1	1	1

malized density, the better the diagnosability and, thus, the lower the wasted effort. In Figure 5.3(a), the three datapoints in the top-left corner ($\rho' = 0 \wedge \mathit{effort} > 0.4$) are classes — `AbstractProcessor`, `IterativeLegendreGaussIntegrator`, `CircleVectorial` — that have an activity matrix where $\rho = 1$ resulting in a normalized density $\rho' = 0$. In Table 5.4, the partial activity matrix of `AbstractProcessor` is shown. During the candidate generation phase using STACCATO, the following fault candidates are generated: $\langle c_1 \rangle$, $\langle c_2 \rangle$, $\langle c_3 \rangle$, $\langle c_4 \rangle$, $\langle c_5 \rangle$, $\langle c_6 \rangle$, $\langle c_7 \rangle$, $\langle c_8 \rangle$, $\langle c_9 \rangle$, $\langle c_{10} \rangle$, $\langle c_{11} \rangle$. BARINEL will rank these candidates randomly because each component is involved in all transactions. Hence, the diagnostic performance of these three classes is as good as randomly investingating components and, thus, the wasted effort for these classes is around 0.5.

The datapoints in the bottom-left corner ($\rho' < 0.1 \wedge \mathit{effort} < 0.05$), in Figure 5.3(a) are classes that have a low density. Since we are only including classes for which we can generate 10 faults that result in a failing test in the error vector, these classes have a high diagnosability for the generated faults and, thus, a low wasted effort.

In the top-right corner of Figure 5.3(a) ($\rho' > 0.8 \wedge \mathit{effort} > 0.3$), the classes have a higher wasted effort than the classes in the bottom-right corner ($\rho' > 0.8 \wedge \mathit{effort} < 0.15$) because the activity matrices consist mostly of tests that cover many components. For example, in Table 5.5, we observe that most transactions cover all components making it difficult for SBR to diagnose the faults accurately. In Table 5.6, we observe the class `BaseOptimizer`'s activity matrix which mostly consists of tests that cover a few components, enabling SBR to diagnose faults better. Note that there are classes in the top-right corner of Figure 5.3(a) for which the activity matrix looks similar to Table 5.6 and, therefore, it is unclear what exactly causes the diagnosability to be good or bad for classes where $\rho' > 0.8$.

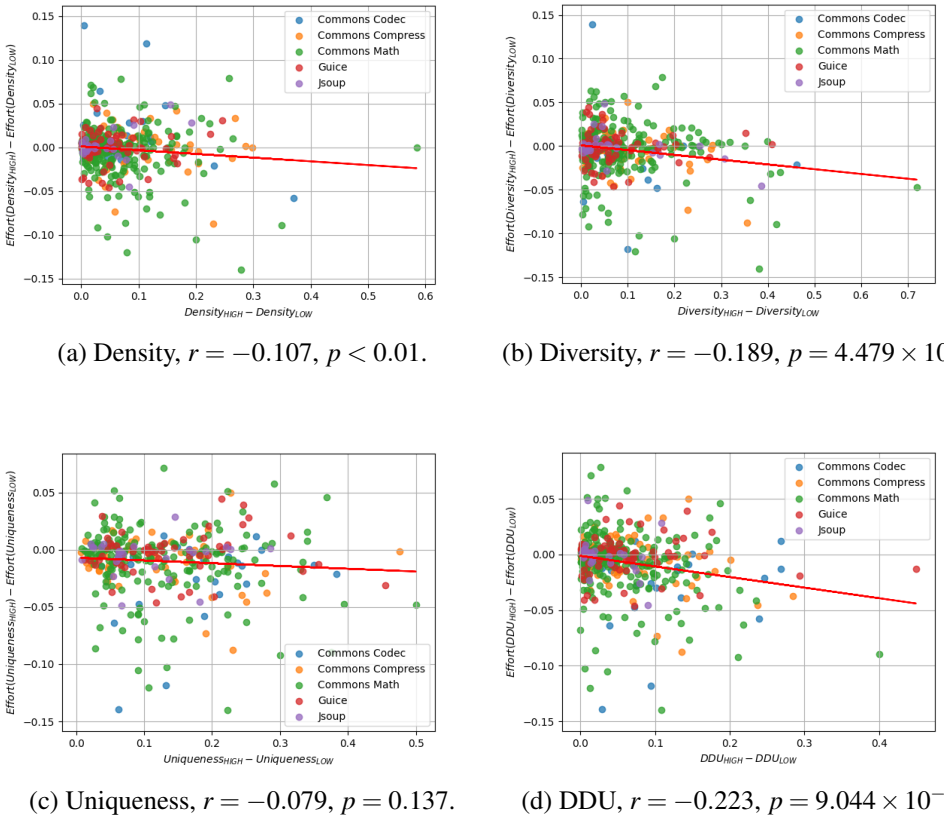
In the second experiment, we generate two test suites for a given class: a test

Table 5.6: Partial activity matrix of `org.apache.commons.math4.optim.BaseOptimizer`, $\rho' = 0.976$.

transaction	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}
`testDimensionMatch`	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1
`testBoundariesDimensionMismatch`	0	0	0	0	1	0	0	0	0	1	1	0	0	1	0	1
`testMissingMaxEval`	0	0	0	0	1	0	0	0	0	1	1	0	1	0	0	1
`testMissingSearchInterval`	0	0	0	0	1	0	0	0	0	1	1	0	1	1	0	1
`testMath290GEQ`	0	0	0	0	1	0	0	0	0	1	1	1	0	0	0	1
`testAckley`	0	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1
`testBadFunction`	0	0	1	0	1	0	1	0	1	1	1	0	1	1	0	1
`testMaxEvaluations`	0	0	1	1	1	0	1	0	0	1	1	0	0	1	0	1
`testMath842Cycle`	1	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1
`testDegeneracy`	1	0	0	0	1	0	0	0	0	1	1	1	0	0	0	1
`testSolutionCallback`	1	0	0	0	1	0	0	1	0	1	1	1	0	0	0	1
`testBoundaries`	1	0	0	0	1	0	1	0	0	1	1	0	1	1	0	1
`testAckley`	1	0	0	0	1	1	1	0	0	1	1	0	1	1	0	1
`testQuinticMax`	1	0	0	1	1	0	1	0	0	1	1	0	1	1	0	1
`testMaxIterations`	1	0	0	1	1	1	1	0	0	1	1	0	1	1	0	1
`testQuinticMin`	1	0	1	0	1	0	1	0	0	1	1	0	1	1	0	1
`testQuinticMin`	1	0	1	0	1	0	1	0	1	1	1	0	1	1	0	1
`testLeastSquares1`	1	0	1	0	1	1	1	0	0	1	1	0	1	1	0	1
`testCircleFitting`	1	0	1	0	1	1	1	0	1	1	1	0	1	1	0	1
`testSinMin`	1	0	1	1	1	0	1	0	1	1	1	0	1	1	0	1

Table 5.7: Correlations between delta normalized density, delta diversity, delta uniqueness, delta DDU, and delta effort for each project.

Subject	Size / Pearson correlation / Correlation p-value			
	Density	Diversity	Uniqueness	DDU
Commons Codec	29	24	26	29
	-0.277	-0.229	0.230	0.011
	0.145	0.281	0.256	0.950
Commons Compress	74	73	55	74
	-0.098	**-0.372**	-0.217	**-0.297**
	0.401	0.001	0.111	0.010
Commons Math	251	256	186	262
	-0.142	**-0.162**	-0.109	**-0.280**
	0.023	0.009	0.135	3.855×10^{-6}
Guice	78	78	57	78
	0.103	0.001	0.047	-0.006
	0.368	0.995	0.728	0.952
Jsoup	29	29	27	29
	0.452	**-0.465**	0.012	-0.273
	0.013	0.011	0.951	0.150



(a) Density, $r = -0.107$, $p < 0.01$.

(b) Diversity, $r = -0.189$, $p = 4.479 \times 10^{-5}$.

(c) Uniqueness, $r = -0.079$, $p = 0.137$.

(d) DDU, $r = -0.223$, $p = 9.044 \times 10^{-7}$.

Figure 5.4: Scatterplot of delta density, delta diversity, delta uniqueness, and delta DDU against delta effort.

suite with the first 50% of the test cases enabled, and a test suite with 100% of the test cases enabled. This naturally results in two test suites with two different DDU values. For each class, we compare the effort of a test suite with a lower DDU value with a test suite with a higher DDU value. Identical approach is used for normalized density, diversity, and uniqueness. Note that we exclude classes where the two test suites do not result in a metric difference. The results of this experiment are shown in Figure 5.4 and Table 5.7. In Figure 5.4, we observe that an increase in any metric — normalized density, diversity, uniqueness, DDU — results in a lower required effort to diagnose mistakes. Although the results are statistically significant except for uniqueness, see Figure 5.4, the correlations are weak.

Revisiting the second research question:

RQ2: What is the relation between density, diversity, uniqueness, and DDU and diagnosability?

A: In the first experiment, we observe that diversity, uniqueness and DDU have a negative weak correlation with diagnosability. The normalized density showed an unexpected weak positive correlation with diagnosability. In the second experiment, we observe that an improvement in normalized density, diversity, uniqueness and DDU have a negative weak correlation with an improvement in diagnosability. In conclusion, there is no strong evidence that indicates that the normalized density, diversity, uniqueness, DDU are strongly correlated with the diagnosability. Consequently, this means that these diagnosability assessment metrics are, currently, only useful for generating test suites to improve diagnosability and SBFL techniques.

Chapter 6

DDU vs. Test Coverage

RQ3: What is the relation between density, diversity, uniqueness, and DDU and test coverage?

In this chapter, we investigate the relation between DDU and test coverage. The reason for researching the relation between DDU and test coverage is to investigate whether DDU should be used as a complementary metric to test coverage as Perez *et al.* proposed. If there is a strong correlation between DDU and test coverage, then it means that DDU might function as an error detection metric too. Furthermore, investigating the relation between DDU and test coverage might give us insight in how DDU and test coverage could work together in practice.

To answer the research question we will perform several experiments. First, we would like to confirm that test coverage is strongly correlated with error detection since test coverage presumably optimizes the test suite for error detection. Second, we would like to investigate the relation between DDU and test coverage. Third, we would like to examine the relation between DDU and error detection.

6.1 Experimental Setup

We have to measure the test coverage and error detection to analyze the relation between test coverage and error detection. Test coverage can be determined by dividing the number of components hit in the activity matrix by the total number of components in the activity matrix. Given the acitity matrix in Table 6.1, we observe that 3 out of the 4 components are hit and therefore the test coverage is $\frac{3}{4} = 0.75$. The error detection is computed by generating 10 artificial faults — in the experiments we generate multiple components faults with a cardinality 2. For each fault, we compute the error vector using an *oracle probability* of 0.75, similar to Perez *et al.*'s study, and check whether the vector contains an error, i.e. $1 \in e_i$. Finally, the error detection is computed by dividing the sum of faults that were detected by 10, the number

Table 6.1: Example of activity matrix with a test coverage of 75%. Bolded components are hit by one of the tests.

transactions	**c_1**	**c_2**	c_3	**c_4**
t_1	0	0	0	1
t_2	1	0	0	1
t_3	1	1	0	0

of generated faults. To account for randomness of generating fault candidates, we repeat this process 10 times. We obtain the activity matrices by using Perez *et al.*'s `ddu-maven-plugin`, similar to the experiments in Chapter 5. The computation of error detection is illustrated in Figure 6.1.

For the second experiment, in which we investigate the relation between DDU and test coverage, we compute the normalized density, diversity, uniqueness, DDU, and test coverage for each activity matrix, and compute the correlation. Furthermore, we use a similar approach as the experiments in Chapter 5, see Figure 6.2, where we create two test suites: (1) a test suite that consists of 50% of the available tests and (2) a test suite that consists of 100% of the available tests. This will most likely result in a system with two test suites with different DDU values, allowing us to investigate whether improving the DDU for a fixed system results in a test coverage improvement.

For the third experiment, we generate two test suites with different DDU values

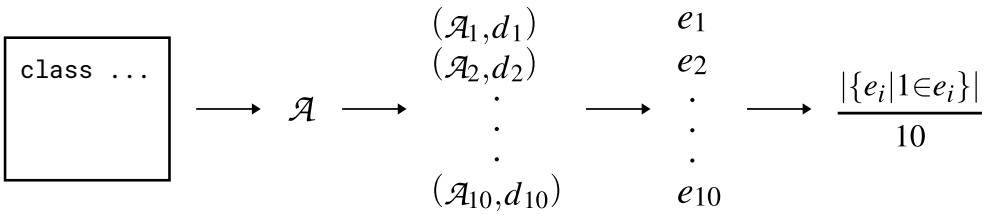


Figure 6.1: We compute the activity matrix for a given class and generate 10 artificial fault candidates d_1, d_2, d_{10} of cardinality 2. For each pair $(\mathcal{A}_i, d_i)$, the error vector e_i is computed. The error detection is computed by dividing the sum of faults that were detected $|\{e_i | 1 \in e_i\}|$ by 10, the number of generated faults.

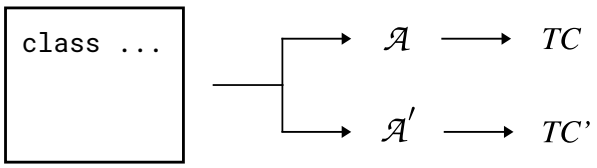


Figure 6.2: For each class, two test suites are generated, resulting in two activity matrices $\mathcal{A}$ and $\mathcal{A}'$. For each activity matrix, we compute test coverage TC, DDU, normalized density, diversity, uniqueness.

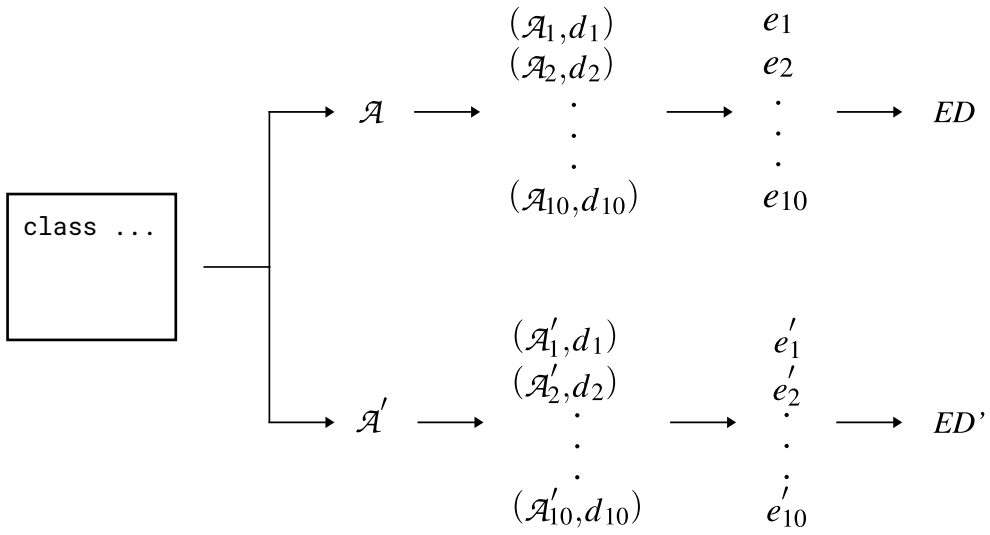


Figure 6.3: For each class, we generate two different test suites, resulting in a two activity matrices. For each matrix, we generate 10 fault candidates of cardinality 2 and compute the error vector. Finally, we compute the error detection and repeat this process 10 times.

and investigate whether the error detection improves when DDU improves, see Figure 6.3. Similar to the first experiment, we generate 10 fault candidates of cardinality 2 and compute the error vector with an *oracle probability* of 0.75 to determine the error detection. To account for randomness of generating fault candidates, we repeat this process 10 times.

Note that for the experiments we use the same projects used in previous experiments, namely Commons Codec, Commons Compress, Commons Math, Guice, Jsoup.

6.2 Experimental Results

In the first experiment, we confirm whether test coverage is representative for error detection. In Figure 6.4, we observe that there is a strong correlation between branch coverage and error detection. Furthermore, we observe that branch coverage puts an upper bound on error detection. The upper bound can be explained with the following example. Assuming that 60% of the components are tested, faults that involve untested components (in the remaining 40%) can never be detected. Therefore, the error detection can only be 100% if the test coverage is 100%.

In the second experiment, we examine the correlation between normalized density, diversity, uniqueness, and DDU and branch coverage. The results are shown in Figure 6.6 and Figure 6.7. We observe in Figure 6.6(a) and Figure 6.6(b) that normalized density is uncorrelated to branch coverage, and that diversity is weakly

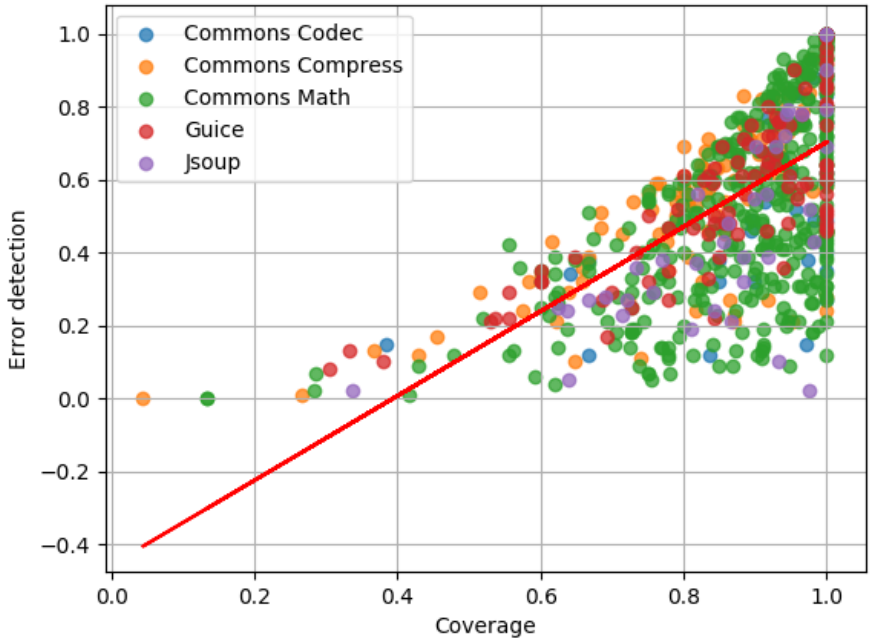


Figure 6.4: Scatterplot of coverage and error detection, $r = 0.628$, $p < 0.01$.

correlated to branch coverage. In Figure 6.6(c) and Figure 6.6(d), we observe that uniqueness and DDU are weakly correlated to branch coverage.

Nevertheless, we observe that uniqueness and DDU put a lower bound on branch coverage. The branch coverage is lower bounded by uniqueness since uniqueness enforces the number of unique columns in the activity matrix. This means that if the uniqueness is low, then the number of unique columns is low and therefore the branch coverage is potentially low. In Figure 6.5(a) and Figure 6.5(b), we observe that branch coverage can be high or low given an activity matrix with a low uniqueness. If the uniqueness is high, then the number of unique columns is high and therefore the branch coverage has to be high too. In Figure 6.5(c) and Figure 6.5(d), we observe two activity matrices with optimal uniqueness, resulting in a relative high or optimal coverage. The DDU lower bounds coverage since uniqueness is a component of DDU.

In the third experiment, we analyze the relation between DDU and error detection. In Figure 6.8, we observe that there is no strong correlation between normalized density, diversity, uniqueness, DDU, and error detection. In Figure 6.9, we observe similar results; no strong correlation between delta normalized density, delta diversity, delta uniqueness, delta DDU, and delta error detection, i.e. improving the components of DDU does not result in an improvement in the error detection. The results in Figure 6.8(c) and Figure 6.9(c) are unexpected because we have seen in

	c_1	c_2	...	c_{10}
t_1	1	0	...	0
t_2	1	0	...	0
t_3	1	0	...	0
t_4	1	0	...	0

(a) Low uniqueness, $\mathcal{U} = 0.2$, $coverage = 0.1$.

	c_1	c_2	...	c_{10}
t_1	1	1	...	1
t_2	1	1	...	1
t_3	1	1	...	1
t_4	1	1	...	1

(b) Low uniqueness, $\mathcal{U} = 0.1$, $coverage = 1.0$.

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

(c) Optimal uniqueness, $\mathcal{U} = 1.0$, $coverage = 1.0$.

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

(d) Optimal uniqueness, $\mathcal{U} = 1.0$, $coverage = 0.75$.

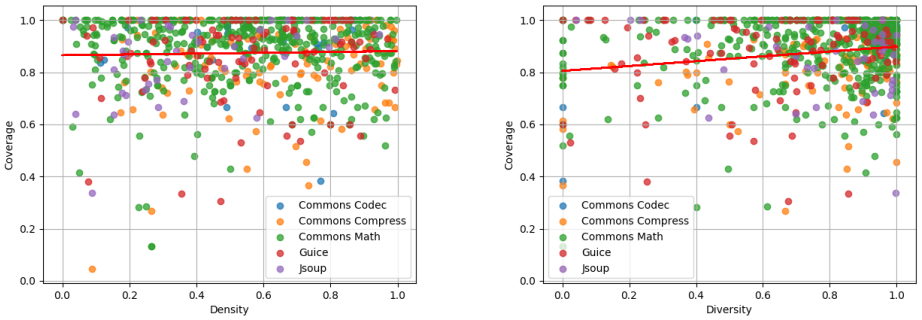
Figure 6.5: Examples of activity matrices with a low or high uniqueness.

previous experiments that branch coverage is strongly correlated to error detection, see Figure 6.4, and that an improvement in uniqueness is strongly correlated to an improvement in branch coverage, see Figure 6.7(c). However, we do not observe any correlation between uniqueness and error detection.

Revisiting the third research question:

RQ3: What is the relation between density, diversity, uniqueness, and DDU and test coverage?

A: In the first experiment, we confirmed that branch coverage is strongly correlated to error detection. In the second experiment, in which we analyze the relation between the diagnosability metrics and branch coverage, we observed that only an improvement in uniqueness is strongly correlated to an improvement in branch coverage. The other metrics did not show any significant correlation. Further, we observed that uniqueness and DDU put a lower bound on branch coverage. In final experiment, we found no evidence that uniqueness and the other diagnosability metrics are correlated to error detection.



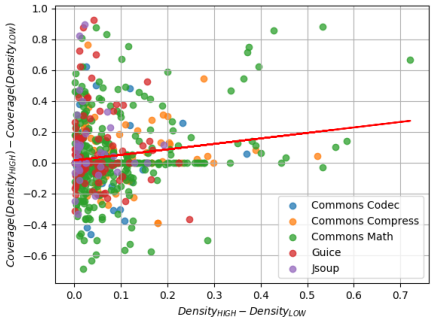
(a) Density, $r = 0.029$, $p = 0.44$.

(b) Diversity, $r = 0.173$, $p < 0.01$.

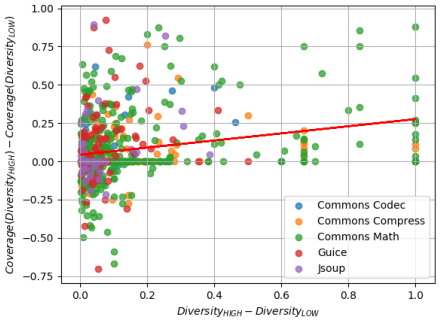
(c) Uniqueness, $r = 0.237$, $p < 0.01$.

(d) DDU, $r = 0.228$, $p < 0.01$.

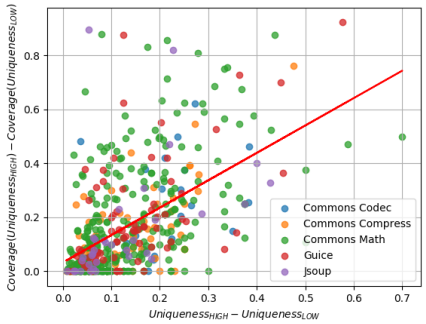
Figure 6.6: Scatterplot of normalized density, diversity, uniqueness, and DDU against branch coverage.



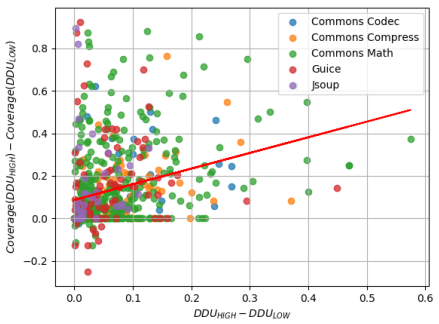
(a) Density, $r = 0.143$, $p < 0.01$.



(b) Diversity, $r = 0.230$, $p < 0.01$.

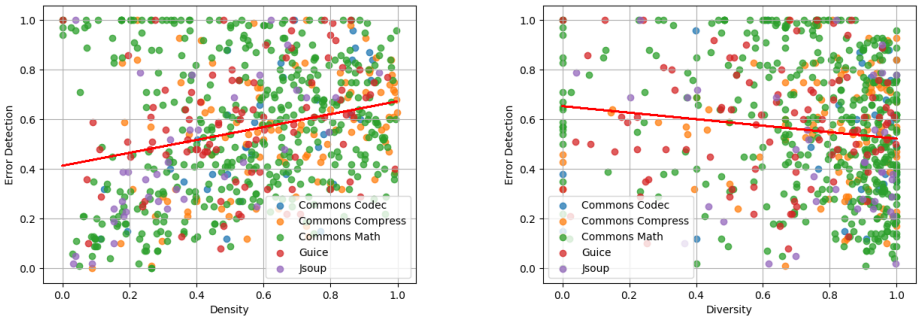


(c) Uniqueness, $r = 0.546$, $p < 0.01$.



(d) DDU, $r = 0.290$, $p < 0.01$.

Figure 6.7: Scatterplot of delta normalized density, delta diversity, delta uniqueness, and delta DDU against delta branch coverage.



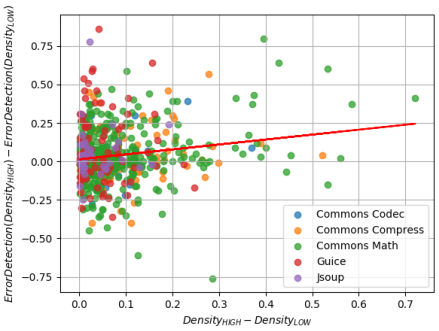
(a) Density, $r = 0.256$, $p < 0.01$.

(b) Diversity, $r = -0.132$, $p < 0.01$.

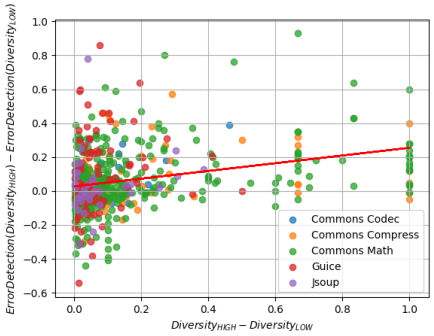
(c) Uniqueness, $r = -0.080$, $p < 0.05$.

(d) DDU, $r = 0.142$, $p < 0.01$.

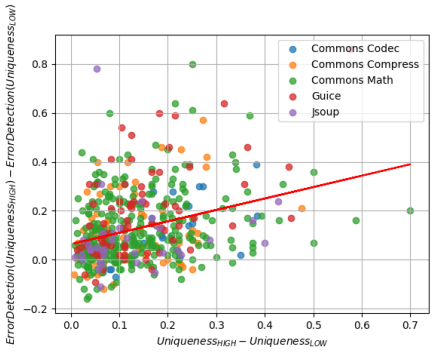
Figure 6.8: Scatterplot of normalized density, diversity, uniqueness, and DDU against error detection.



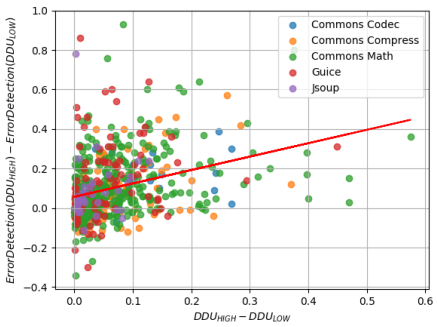
(a) Density, $r = 0.165$, $p < 0.01$.



(b) Diversity, $r = 0.275$, $p < 0.01$.



(c) Uniqueness, $r = 0.323$, $p < 0.01$.



(d) DDU, $r = 0.324$, $p < 0.01$.

Figure 6.9: Scatterplot of delta normalized density, delta diversity, delta uniqueness, and delta DDU against delta error detection.

Chapter 7

Conclusion

Bibliography

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

[2] Rui Abreu and Arjan J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In Vadim Bulitko and J. Christopher Beck, editors, *SARA*. AAAI, 2009.

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[5] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

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

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

[8] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization for fault localization. In *in Proceedings of ICSE 2001 Workshop on Software Visualization*, pages 71–75, 2001.

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

[10] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, August 2011.

[11] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[12] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 654–664, 2017.

[13] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software EngineeringESEC/FSE'97*, pages 432–449. Springer, 1997.

[14] John Robbins. *Debugging Applications for Microsoft .NET and Microsoft Windows*. Microsoft Press, 2003.

[15] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 314–324, New York, NY, USA, 2013. ACM.

[16] Y. Le Traon, F. Ouabdesselam, and C. Robach. Software diagnosability. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 257–266, Nov 1998.

[17] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, March 2014.

[18] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham. Effective software fault localization using an rbf neural network. *IEEE Transactions on Reliability*, 61(1):149–169, March 2012.

[19] W. E. Wong, V. Debroy, and D. Xu. Towards better fault localization: A crosstab-based statistical approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):378–396, May 2012.

[20] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[21] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[22] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188 – 208, 2010. Computer Software and Applications.