# New Strategies for Automated Random Testing

Mian Asbat Ahmad

Department of Computer Science

The University of York

A thesis submitted for the degree of

*Doctor of Philosophy*

September 19, 2013

# Abstract

This is where you write your abstract ...

# Contents

# List of Figures

# List of Tables



.

# Acknowledgements

The years I spent working on my PhD degree at the University of York have undoubtedly been some of the most joyful and rewarding in my academic career. The institution provided me with everything I need to thrive: challenging research problems, excellent company, and a supportive environment. I am deeply grateful to the people who shared this experience with me and to those who made it possible.

Several people have contributed to the completion of my PhD dissertation. However, the most prominent personality deserving due recognition is my worthy advisor, Dr. Manuel Oriol. Thank you Manuel for your endless help, valuable guidance, constant encouragement, precious advice, sincere and affectionate attitude.

I thank my assessor prof. John Clark for his constructive feedback on my various reports and presentations. I am also thankful and highly indebted to Prof. Richard Paige for his generous help, cooperation and guidance during my research at the University of York.

Special thanks to my father prof. Mushtaq A. Mian who provided a conducive environment, valuable guidance and crucial support at all levels of my educational career and my very beloved mother whose love, affection and prayers have been my most precious assets. Also I am thankful to my brothers Dr. Ashfaq, Dr. Aftab, Dr. Ishaq, Dr. Afaq, Dr. Ilyas and my sister Dr. Haleema who have been the source of inspiration for me to pursue higher studies. Last but not the least I am very thankful to my dear wife Dr. Munazza for her company, help and cooperation throughout my stay at York.

I was funded by Departmental Overseas Research Scholarship (DORS), a financial support awarded to overseas students on the basis of outstanding academic ability and research potential. I am truly grateful to the Department of Computer Science for financial support that allowed me to concentrate on my research.

I feel it a great honour to dedicate my PhD thesis to my beloved parents for their significant contribution in achieving the goal of academic excellence.

# Chapter 1

# Introduction

This chapter includes motivation for the research study followed by the problems in random testing, the alternative approaches to overcome these problems, the research objectives and contributions of the study. At the end of the chapter, the structure of the remaining thesis is given.

## 1.1 Motivation

Today, software pervades every aspect of our life, from simple day to day operations to highly complex processes in specialized fields like research and education, business and finance, defence and security, health and medicine, science and technology, aeronautics and astronautics, commerce and industry, information and communication, environment and safety etc. The ever increasing dependency of software expect us to believe that the software in use is reliable, robust, safe and secure. Unfortunately the performance of software in general is not what is expected. According to the National Institute of Standards and Technology (NIST) US companies alone bear $60 billion loss every year due to software failures and one-third of that can be eliminated by an improved testing infrastructure [40]. Humans are prone to errors and programmers are no exceptions. Maurice Wilkes [97], a British computer pioneer, stated that:

> "As soon as we started programming, we found to our surprise that it wasnt as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

The margin of error in mission-critical and safety-critical systems is so small that a minor fault can lead to huge economic losses [50]. Therefore, software companies leave no stone

unturned to ensure the reliability and accuracy of the software. Software testing is by far the most recognized and widely used technique to verify the correctness and ensure quality of the software [65], [99], [82], [40]. According to Myers et al. some software companies spend up to 50% cost of the total cost of software development and maintenance on testing [65].

In the IEEE standard glossary of software engineering terminology [3], testing is defined as "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements and actual results. This dissertation is a humble contribution to the literature on the subject, with the aim to reduce the overall cost of software testing by devising new, improved and effective automated software testing techniques based on random strategy.

Random testing is a process (Figure 1.1) in which generation of test data is random but according to requirements, specifications or any other test adequacy criteria. The given SUT is executed against the test data and results obtained are evaluated to determine whether the output produced satisfies the expected results.

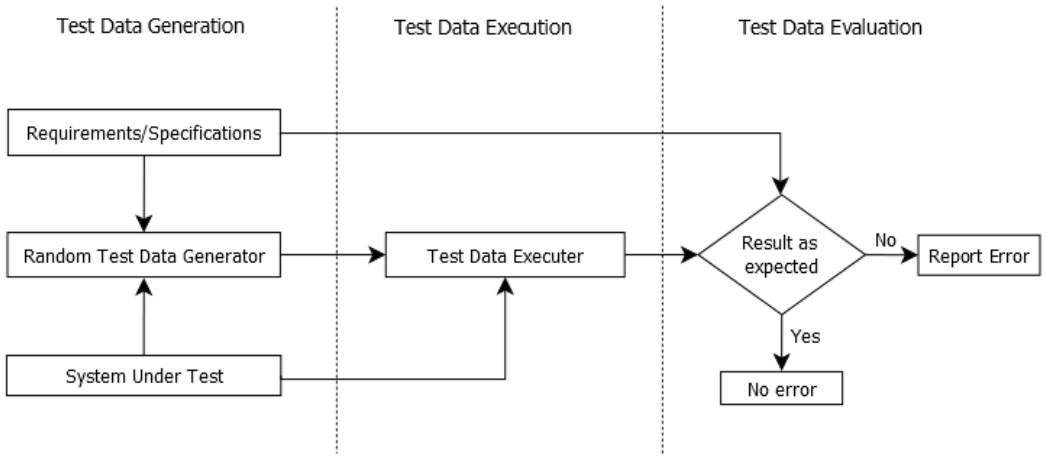



Figure 1.1: Three main phases of random testing

## 1.2 The Problems

Exhaustive testing of software is not always possible and the problem of selecting a test data set, from a large/infinite domain is often confronted. Test data set, as a subset of the whole domain, is carefully selected for testing the given software. Adequate test data set is a crucial factor in any testing technique because it represents the whole domain for evaluating the

structural and/or functional properties [49], [62]. Miller and Maloney were the first who comprehensively described a systematic approach of test data set selection known as path coverage. They proposed that testers select the test data so that all paths of the SUT are executed at least once [63]. The implementation of the strategy resulted in higher standards of test quality and a large number of test strategies were developed afterwords including boundary value analysis and equivalence class.

Generating test data set manually is a time-consuming and laborious exercise [54]; Therefore, automated test data set generation is always preferred. Data generators can be of different types i.e. Path-wise, Goal-Oriented, Intelligent or Random [96]. Random generator produces test data set randomly from the whole domain. Unlike other approaches random technique is simple, widely applicable, easy to implement, faster in computation, free from bias and costs minimum overhead [27]. According to Godefroid et al, "Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs" [45].

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [95]. Myers et al. [65] mentioned it as, "Probably the poorest methodology of all is random-input testing...". However, Ciupa et al. reported that the above stated statement is based on intuition and lacks any experimental evidence [28]. The criticism motivated the researchers to look into various aspects of random testing for evaluation and possible improvement. Adaptive random testing (ART) [15], Restricted Random Testing (RRT) [10], Feedback Directed Random Testing (FDRT) [80], Mirror Adaptive Random Testing (MART) [16] and Quasi Random Testing (QRT) [20] are a few of the enhanced random testing techniques reported in the literature.

Random testing is also considered weak in providing high code coverage [33], [68]. For example, in random testing when the conditional statement "*if (x == 25) then ...* " is exposed to execution then there is only one chance, of the "*then...*" part of the statement, to be executed out of $2^{32}$. If $x$ is an integer variable of 32 bit value [45].

Random testing is no exception when it comes to the complexity of understanding and evaluating test results. Modern testing techniques simplify results by truncating the lengthy log files and displaying only the fault revealing test cases in the form of unit tests. Further efforts are required to get the test results of random testing in more compact and user-friendly way.

## 1.3 Research Goals

The main goal of the research study is to develop new techniques for automated random testing with the aim to achieve the following objectives:

1. To develop a testing strategy with the capability to generate more fault-finding test data.
2. To develop a testing technique for finding faults, fault domains and presentation of results on a graphical chart within the specified lower and upper bound.
3. To develop a testing framework with focus on increase in code coverage along with generation of more faultfinding test data.

## 1.4 Contributions

The main contributions of the thesis research are stated below:

### 1.4.1 Dirt Spot Sweeping Random Strategy

The fault-finding ability of the random testing technique decreases when the failures lie in contiguous locations across the input domain. To overcome the problem, a new automated technique: Dirt Spot Sweeping Random (DSSR) strategy was developed. It is based on the assumption that unique failures reside in contiguous blocks and stripes. When a failure is identified, the DSSR strategy selects neighbouring values for the subsequent tests. Resultantly, selected values sweep around the failure, leading to the discovery of new failures in the vicinity. Results presented in Chapter 3 indicated higher fault-finding ability of DSSR strategy as compared with Random (R) and Random+ (R+) strategies.

### 1.4.2 Automated Discovery of Failure Domain

The existing random strategies of software testing discover the faults in the SUT but lack the capability of locating the fault domains. In the current research study, a fully automated testing strategy named, "Automated Discovery of Failure Domain (ADFD)" was developed with the ability to find the faults as well as the fault domains in a given SUT and provides visualization of the identified pass and fail domains in the form of a chart. The strategy is described, implemented in YETI, and practically illustrated by executing several programs of one and two dimensions in the Chapter 4. The experimental results proved that ADFD strategy automatically performed identification of faults and fault domains along with graphical representation in the form of chart.

### 1.4.3 Invariant Guided Random+ Strategy

Another random test strategy named, "Invariant guided Random+ Strategy (IGR+S)" was developed in the current research study. IGR+S is an extended form of Random+ strategy guided

by software invariants. Invariants from the given SUT are collected by Daikon— an automated invariant detector for reporting likely invariants and adding them to the SUT as assertions. The IGR+S is implemented in YETI and generates values in compliance with the added assertions. Experimental results presented in Chapter 5 indicated improved features of IGR+S in terms of higher code coverage and identification of subtle errors that R, R+ and DSSR strategies were either unable to accomplish or required larger duration.

## 1.5 Structure of the Thesis

The rest of the thesis is organized as follows: In Chapter 2, a thorough review of the relevant literature is given. It includes a brief introduction of software testing techniques followed by automated random testing tools. Chapter 3 describes Dirt Spot Sweeping Random (DSSR) strategy, which is based on sweeping of fault clusters in the input domain. Chapter 4 presents the newly developed Automated Discovery of Fault Domains (ADFD) strategy, which focuses on dynamically finding the faults and domains along with their graphical representation. Chapter 5 presents the new strategy Invariant Guided Random+ Strategy (IGR+S) developed with the focus on quick identification of faults and increase in code coverage with the help of assertions. Chapter 6 summarizes contributions of the thesis research, discusses the strength and weaknesses of the study, gives conclusion and suggests avenues for future work. Chapter 7 ?

# Chapter 2

# Literature Review

The very famous quote of, "To err is human, but to really foul things up you need a computer", is quite relevant to the software programmers. Programmers being humans are prone to errors. Errors are not tolerated, particularly in software because a single error may cause a large upset in the system. According to the National Institute of Standard and Technology 2002, software errors cost an estimated $59.5 billion loss to US economy annually. The destruction of the Mariner 1 rocket (1962) costing $18.5 million was due to a small error in formula coded incorrectly by programmer. The Hartford Coliseum Collapse (1978) costing $70 million, Wall Street crash (1987) costing $500 billion, failing of long division by Pentium (1993) costing $475 million, Ariane 5 Rocket disaster costing $500 million are a few examples of losses caused by minor errors in the software. To achieve high quality, the software has to satisfy rigorous stages of testing. The more critical and complex the software, the higher the requirements for software testing and the larger the extent of damage caused as a result of an error in the software.

## 2.1 Software Testing

In the IEEE standard glossary of software engineering terminology [3], testing is defined as "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements and actual results. A successful test is one that finds a fault [**?** ], where fault is defined as, the error made by people during software development [3].

Being an integral part of Software Development Life Cycle (SDLC), the testing process starts from requirement phase and continues throughout the life of the software. In traditional testing when tester finds a fault in the given SUT, the software is returned to the developers for

rectification and is consequently given back to the tester for retesting. It is important to note that, "program testing can be used to show the presence of bugs, but never to show the absence of bugs" [36]. In other words, a SUT that passes all the tests without giving a single error is not guaranteed to contain no error. However, the testing process increases the reliability and confidence of users in the tested product.

Table 2.1: Parts of Software Testing [1], [24], [43], [85], [90]

| Levels | Purpose | Perspective | Execution |
|---|---|---|---|
| 1. Unit<br>2. Integration<br>3. System | 1. Functionality<br>2. Structural<br>3. Robustness<br>4. Stress<br>5. Compatibility<br>6. Performance | 1. White Box<br>2. Black Box<br>3. Grey Box | 1. Static<br>2. Dynamic |

### 2.1.1 Software Testing Levels

The three main levels of software testing defined in the literature include unit testing, integration testing and system testing [24]. Unit testing involves evaluation of piece-by-piece code and each piece is considered as independent unit. Units are combined together to form components. Integration testing ensures that the integration of units in a component is working properly. System testing is called out to make sure that the system formed by combination of components performs correctly to give the desired output.

### 2.1.2 Software Testing Purpose

The primary purpose of software testing is identification of faults in the given SUT for necessary correction in order to achieve high quality. Maximum number of faults can be identified if software is tested exhaustively. In exhausting testing SUT is checked against all possible combinations of input data, and the results obtained are compared with the expected results for assessment. Exhaustive testing is not always possible in most scenarios because of limited resources and infinite number of input values that software can take. Therefore, the purpose of testing is generally directed to achieve confidence in the system involved from a specific point of view. For example, functionality testing is performed to check functional aspect for working correctly. Structural testing analyses the code structure for generating test cases in order to evaluate paths of execution and identification of unreachable or dead code. In robustness testing the software behaviour is observed in the case when software receives input outside the

expected input range. Stress and performance testing aims at testing the response of software under high load and its ability to process different nature of tasks [34]. Finally, compatibility testing is performed to see the interaction of software with underlying operating system.

### 2.1.3 Software Testing Perspective

Testing activities can be split up into white-box and black-box testing on the basis of perspective taken.

#### 2.1.3.1 White-box testing

In white-box or structural testing, the testers must do need to know about the complete structure of the software and can do necessary modification, if so required. Test cases are derived from the code structure and test passes only if the results are correct and the expected code is followed during test execution [76]. Some of the most common White-box testing techniques are briefly defined:

**2.1.3.1.1 Data Flow Analysis** Data Flow Analysis is a testing technique that focuses on the input values by observing the behaviour of respective variables during the execution of the SUT [32]. In this technique a control flow graph (CFG), graphical representation of all possible states of program, of a SUT is drawn to determine the paths that might be traversed by a program during its execution. Test cases are generated and executed to verify its conformance with CFG on the basis of data.

Normally, program execution implies input of data, operations on it according to the defined algorithm, and output of results. This process can be viewed as a flow of data from input to output in which data may transform into several intermediate results before reaching its final state. In the process several errors can occur e.g. references may be made to variables that dont exist, values may be assigned to undeclared variables or the value of variables may be changed in an unexpected and undesired manner. It is the ordered use of data implicit in this process that is the central object of the technique to ensure that none of the aforementioned errors occur [42].

**2.1.3.1.2 Control Flow Analysis** Control flow Analysis is a testing technique which takes into consideration the control structure of a given SUT. Control structure is the order in which the individual statements, instructions or function calls are executed. In this technique a control flow graph (CFG), similar to the one required in data flow analysis, is drawn to determine the paths that might be traversed by a program during its execution. Test cases are generated and

executed to verify its conformance with CFG on the basis of control. For example to follow a specific path (also known as branch) between two or more choices at specific state. Efforts are made to ensure that the set of selected test cases execute all the possible control choices at least once. The effectiveness of the testing technique depends on controls measurement. Two of the most common measurement criteria defined by Vilkomir et al. are Decision/Branch coverage and Condition coverage [92].

#### 2.1.3.1.3 Code based Fault Testing

#### 2.1.3.2 Black-box testing

In black-box or functional testing, the testers do not need to know about internal code structure of the SUT. Test cases are derived from the specifications and test passes if the result is according to expected output. Internal code structure of the SUT is not taken into any consideration [7]. Some of the most common black-box testing techniques are briefly defined:

#### 2.1.3.2.1 Use Case based Testing

#### 2.1.3.2.2 Partition Testing

#### 2.1.3.2.3 Boundary Value Analysis

#### 2.1.3.2.4 Formal Specification Testing

### 2.1.4 Software Testing Execution

Testing process can be divided into static and dynamic phases on the basis of test execution. In static testing test cases are analysed statically for checking errors without test execution. In addition to software code, high quality softwares are accompanied by necessary documentation. It includes requirements, design, technical, user manual marketing information. Reviews, walkthroughs or inspections are most commonly used techniques for static testing. In dynamic testing the software code is executed and input is converted into output. Results are analysed against expected outputs to find any error in the software. Unit testing, integration testing, system testing, and acceptance testing are most commonly used as dynamic testing methods [39].

### 2.1.5 Manual Testing

Manual testing is the technique of finding faults in software in which the tester writes the code by hand to create test cases and test oracle [29]. Manual testing may be effective in some cases but it is generally laborious, time consuming and error-prone [91]. Additionally, it requires that the testers must have appropriate skills, experience and sufficient knowledge of the SUT for evaluation from different perspectives.

### 2.1.6 Automated Testing

Automated testing is the technique of finding faults in a software in which the test cases or test oracle are generated automatically by a testing tool [55]. There are tools which can automate part of a test process like generation of test cases or execution of test cases or evaluation of results while other tools are available which can automate the whole testing process. The use of automated testing made it possible to test large volumes of code, which would have been impossible otherwise [84].

### 2.1.7 Test Oracle

Test oracles set the acceptable behaviour for test executions [5]. All software-testing techniques depend on the availability of test oracle [43]. Designing test oracle for ordinary software may be simple straightforward. However, for relatively complex software designing of oracle is quite cumbersome and requires special ways to overcome the oracle problem. Some of the common issues in the oracle problem include:

1. The assumption that the test results are observable and can be compared with the oracle.
2. An ideal test oracle would satisfy desirable properties of program specifications [5].
3. A specific oracle to satisfy all conditions is seldom available as rightly pointed out by Weyuker who ?states that truly general test oracles are often unobtainable Weyuker [94]

## 2.2 Random Testing

Random testing is a dynamic black-box testing technique in which the software is tested with non-correlating or unpredictable test data from the specified input domain [10]. The input domain is a set of all possible inputs to the software under test. According to Richard [48], in random testing, input domain is first identified, then test points are randomly taken from the whole input domain by means of random number generator. The program under test is executed

on these points and the results obtained are compared with the program specifications. The test fails if the results are not according to the specifications and vice versa. Failure of any test results reflects error in the SUT.


Figure 2.1: Random Testing

Generating test data by random generator is quite economical and requires less intellectual and computational efforts [28]. This feature makes it an ideal choice for use in automated testing tools [—]. Moreover, the no human intervention is involved in data generation that ensures an unbiased testing process.

However, generating test cases with out using any background information makes random testing susceptible to criticism. It is criticized for generating many test cases ending up at the same state of software. It is further stated that, random testing generates test inputs that violates requirements of the given SUT making it less effective [86], [77]. Myers mentioned random testing as one of the least effective testing technique [64].

It is stated by ——— that Myers statement was not based on any experimental evidence. Later experiments performed by several researchers [48], [29], [56] and [37] confirmed that random testing is as effective as any other testing technique. It was found experimentally [37] that random testing can also discover subtle faults in a given SUT when subjected to large number of test cases. It was pointed out that the simplicity and cost effectiveness of random testing makes it more feasible to run large number of test cases as opposed to systematic testing techniques which require considerable time and resources for test case generation and execution. The empirical comparison proves that random testing and partition testing are equally effective [47] A comparative study conducted by Ntafos [66]. concluded that random testing is more effective as compared to proportional partition testing.

## 2.3 Various versions of random testing

Researchers have tried various approaches to bring about improvement in the performance of random testing. The prominent modifications in random strategy are stated below:

### 2.3.1 Adaptive Random Testing

Adaptive random testing (ART), proposed by Chen et al. [15] is based on the previous work of Chan et al. [9]regarding the existence of failure patterns across the input domain. Chan et al observed that failure inducing inputs in the whole input domain form certain geometrical patterns and divided these into point, block and strip patterns described below.

1. Point pattern: In the point pattern inputs inducing failures are scattered across the input domain in the form of stand-alone points. Example of point pattern is the division by zero in a statement total = num1/num2; where num1, num2 and total are variables of type integer.

2. Block pattern: In the block pattern inputs inducing failures lie in close vicinity to form a block in the input domain. Example of block pattern is failure caused by the statement if ( (num >10) && (num <20) ). Here 11 to 19 are a block of faults.

3. Strip pattern: In the strip pattern inputs inducing failures form a strip across the input domain. Example of strip pattern is failure caused by a statement num1 + num2 = 20. Here multiple values of num1 and num2 can lead to the fault value 20.

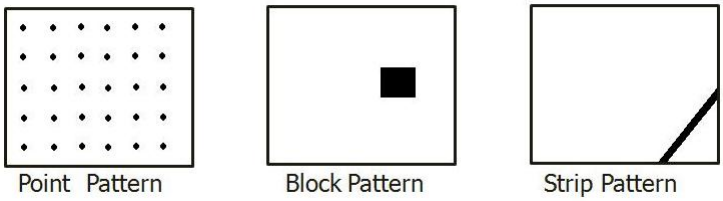



Figure 2.2: Patterns of failure causing inputs

In figure 2.2 the square boxes indicate the whole input domains. The white space in each box shows legitimate and faultless values while the black colour in the form of points, block and strip inside the respective box each box indicates the point, block and strip fault patterns.

Chen et al. [15] argued that ordinary random testing might generate test inputs lurking too close or too far from the input inducing failure and thus fails to discover the fault. To generate more faults targeted test inputs they proposed Adaptive Random Testing (ART) as a modified version of ordinary random testing where test values are selected at random as usual but are evenly spread across the input domain by using two sets. The executed set comprises the test cases executed by the system and the candidate set includes the test cases to be executed by the system. Initially both the sets are empty. The first test case is selected at random from the

candidate set and stored in executed set after execution. The second test case is then selected from the candidate set which is far away from the last executed test case. In this way the whole input domain is tested with greater chances of generating test input from the existing fault patterns.

In the experiments conducted by Chen et al. [15] the number of test cases required to detect first fault (F-measure) was used as a performance matrix instead of the traditional matrix P-measure and E-measure. Experimental results using ART showed up to 50% increase in performance compared to ordinary random testing. The authors admitted that the issues of increase overhead, spreading test cases across the input domain for complex objects and efficient ways of selecting candidate test cases still exist. Chen et al continued their work on ART to address some of these issues and proposed its upgraded version in [17] and [20].

### 2.3.2 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) [16] is an improvement on ART by using mirror-partitioning technique to decrease the extra computation involved in ART and reduce the overhead.

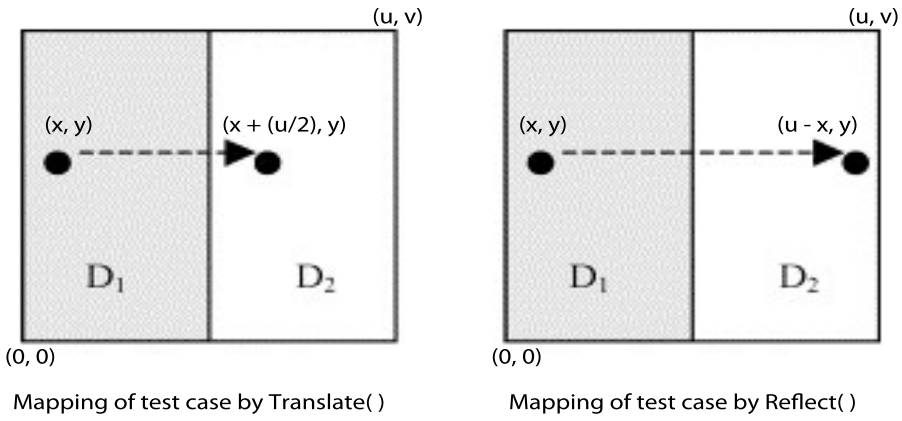



Figure 2.3: Mirror Adaptive Random Testing [16]

In this technique, the input domain of the program under test is divided into n disjoint sub-domains of equal size and shape. One of the sub-domains is called source sub-domain while all the others are termed as mirror sub-domains. ART is then applied only to the source sub-domain to select the test cases while from all other sub-domains test cases are selected by using mirror function. In MART (0, 0), (u, v) are used to represent the whole input domain where (0, 0) is the leftmost and (u, v) is the rightmost top corner of the two dimensional rectangle.

On splitting it into two sub-domains we get (0, 0), (u/2, v) as source sub-domain and (u/2, 0), (u, v) as mirror sub-domain. Let suppose we get x and y test cases by applying ART to source sub-domain, now we can linearly translate these test cases to achieve the mirrored effect, i.e. (x + (u/2), y) as shown in the figure 2.3. Comparative study of MART with ART provide evidence of equally good results of the two strategies with MART having the added advantage of using only one quarter of the calculation as compared with ART and thereby reduces the overhead.

### 2.3.3 Restricted Random Testing

Restricted Random Testing [11] is another approach to overcome the problem of extra overhead in ART. Restricted Random Testing (RRT) achieves this by creating a circular exclusion zone around the executed test case. A candidate is randomly selected from the input domain for the next test case. Before execution the candidate is checked and discarded if it lies inside the exclusion zone. This process repeats until a candidate laying outside the exclusion zone is selected. This ensures that the test case to be executed is well apart from the last executed test case. The radius of exclusion zone is constant around each test case and the area of each zone decreases with successive cases.

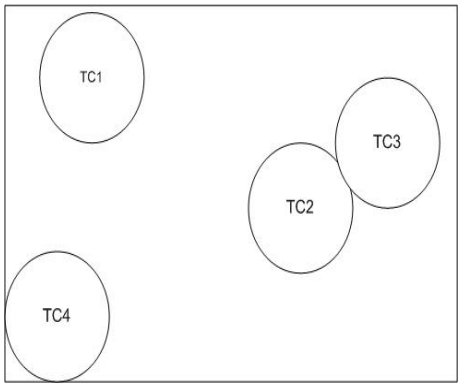



Figure 2.4: Input domain with exclusion zone around the selected test case

To find the effectiveness of RRT, the authors compared it with ART and RT on 7 out of the 12 programs evaluated by ART and MART. The experimental results showed that the performance of RRT increases with the increase in the size of the exclusion zone and reaches the maximum level when the exclusion zone is raised to largest possible size. They further found that RRT is up to 55% more effective than ordinary random testing in terms of F-measure (Where F-measure is the total number of test cases required to find the first failure).

### 2.3.4 Directed Automated Random Testing

Godefroid et al., [45] proposed Directed Automated Random Testing (DART). The main features of DART can be divided into the following three parts:

1. Automated Interface Extraction: DART automatically identifies external interfaces of a given SUT. These interfaces include external variables, external functions and the user-specified main function, which initializes the program execution.

2. Automatic Test Driver: DART generate test drivers to run the test cases. All the test cases are randomly generated according to the underlying environment.

3. Dynamic Analysis of execution: The DART instruments the given SUT at the start of the process in order to track its behaviour dynamically at run time. The results obtained are analysed in real time to systematically direct the test case execution along alternative path for maximum code coverage.

The DART algorithm is implemented in the tool which is completely automatic and accepts the test program as input. After the external interfaces are extracted it then use the pre-conditions and post-conditions of the program under test to validate the test inputs. For languages that do not support contracts inside the code (like C), they used public methods or interfaces to mimic the scenario - to be continued

### 2.3.5 Quasi Random Testing

Quasi-random testing (QRT) [20] is a testing technique which takes advantage of failure region contiguity by distributing test cases evenly with decreased computation. To achieve even spreading of test cases, QRT uses a class with a formula, that forms an s-dimensional cube in s-dimensional input domain and generates a set of numbers with small discrepancy and low dispersion. The set of numbers is then used to generate random test cases that are permuted to make them less clustered and more evenly distributed. An empirical study was conducted to compare the effectiveness of QRT with ART and RT. The empirical results of the experiments showed that in 9 out of 12 programs QRT found a fault quicker than ART and RT while there was no significant improvement in the remaining three programs.

### 2.3.6 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) [81] is a technique that generate unit test suite at random for object-oriented programs. As the name implies FDRT uses the feedback received from the execution of first batch of randomly selected unit test suite to generate next batch

of more directed unit test suite. In this way redundant and illegal unit tests are eliminated incrementally from the test suite with the help of filtration and application of contracts. For example unit test that produce IllegalArgumentException on execution is discarded, because, selected argument used in this test was not according to the type of argument the method required.

#### 2.3.6.1 Randoop: Feedback-directed Random Testing

The FDRT technique is implemented in Random tester for Object Oriented Programs (RANDOOP) tool [79]. RANDOOP is a fully automatic tool, capable of testing Java classes and .Net binaries which takes a set of classes (java or .Net executable), contracts, filters and time limit as input and gives output as a suite of JUnit and NUnit for Java and .Net programs respectively. Each unit test in a test suite is a sequence of method calls (hereafter referred as sequence). RANDOOP build the sequence incrementally by randomly selecting a public method from the class under test and arguments for these methods are selected from the predefined pool in case of primitive types and a sequence or null value in case of reference type. RANDOOP maintains two sets called ErrorSeqs and NonErrorSeqs to record the feedback. It extends ErrorSeqs set in case of contract or filter violation and NonErrorSeqs set when no violation is recorded in the feedback. The use of this dynamic feedback evaluation at runtime brings an object to very complex and interesting state. On test completion it produce ErrorSeqs and NonErrorSeqs as JUnit/NUnit test suite. In terms of coverage and number of faults discovered, RANDOOP implementing FDRT was compared with random testing of JCrasher and JavaPathFinder [93]. In the experiments 14 libraries of both Java and .Net were evaluated. The results showed that RANDOOP achieved more coverage than JCrasher in behavioural, branch coverage and faults detection. It can achieve on par coverage with systematic approaches like JavaPathFinder. RANDOOP also has an edge over model checking for its ability to easily search large input domains.

### 2.3.7 Object Distance and its application

To improve the performance of random testing the emphasis of ART was on the distance between the test cases. But this distance was defined only for primitive data types like integers and other elementary input. Ciupa et al defined the parameters that can be used to calculate distance between the composite programmer-defined types so that ART can be applicable to testing of todays object-oriented programs [26]. Two objects have more distance between them if they have more dissimilar properties. The parameters to specify the distance between the objects are dynamic types, values of its primitive and reference fields. Strings are treated as

a directly usable values and Levenshtein distance [58] that is also known as edit distance is used as a distance criteria between the two strings. To implement object distance first all the distances of the objects are measured. Then two sets candidate- objects containing the all the objects ready to be run by the system and the used-objects set, which is initially empty. First object is selected randomly from the candidate-object set and is moved to used-object set when executed by the system. Now the second object selected from the candidate set for execution is the one with the biggest distance from the last executed object present in the used-object set. This process is continuing until the bug is found or the objects in the candidate-object set are finished.

#### 2.3.7.1 ARTOO Tool

After the criteria to calculate the distance between the objects is defined [26], the same team implemented that model and performed several experiments to evaluate the proposed model. Adaptive Random Testing for Object Oriented (ARTOO) is a testing strategy, based on object distance, implemented in AutoTest tool [**?** ]. ARTOO was implemented as a plug-in strategy in AutoTest. It only deals with creating and selecting inputs and all other functionality of the AutoTest was the same. Since ARTOO is based on object distance therefore the method for test input selection is to pick that object from the candidate set (A pool of objects that is a potential candidate to be executed by the system) that has the highest average distance in comparison to the objects already executed. In the experiments classes from EiffelBase library [**?** ] were used. To evaluate ARTOO the same tests were also applied to directed random strategy (RAND). The outcome of the experiments showed that ARTOO finds the first bug with fewer test cases than RAND. The computation to select test case in ARTOO is more than RAND and therefore ARTOO takes more time to generate a test input. The experiments also found few of the bug found by ARTOO were not pointed out by RAND furthermore ARTOO is less sensitive to the variation of seed value than RAND.

#### 2.3.7.2 Experimental Assessment of RT for Object-Oriented Software

In this research the effect of various parameters involved in random testing and its effect on efficiency is evaluated by performing various experiments on Industrial-grade code base. Large-scale clusters of computers were used for 1500 hours of CPU time which resulted in 1875 test sessions for 8 classes under test. [27] The finding of the experiments are 1. Version of random testing algorithm that is efficient for smaller testing timeout is equally efficient for higher testing timeouts. 2. The value of seed for random testing algorithm plays a vital role in finding the number of bugs in specific time. 3. Most of the bugs are found in the first few minutes of the

testing sessions.

## 2.4 Tools for Automated Random Testing

From the literature we can find a number of open source and commercial testing tools that automatically generate unit tests. Each tool utilize different generation technique but the one we are interested in is random technique. We present the most well known tools.

### 2.4.1 JCrasher

JCrasher is first of the three automatic testing tools developed by Csallner C. and Smaragadakis Y. [79]. As the name suggests JCrasher tries to crash the Java program with random input and any exceptions thrown during the process are recorded. All exceptions are then compared to the list of acceptable exception, which are defined in advance as heuristics; any undefined/undeclared runtime exceptions are considered errors. Since programs interact with the world through its public methods and they are also exposed to different kind of inputs, therefore, JCrasher tests only these methods with random inputs.

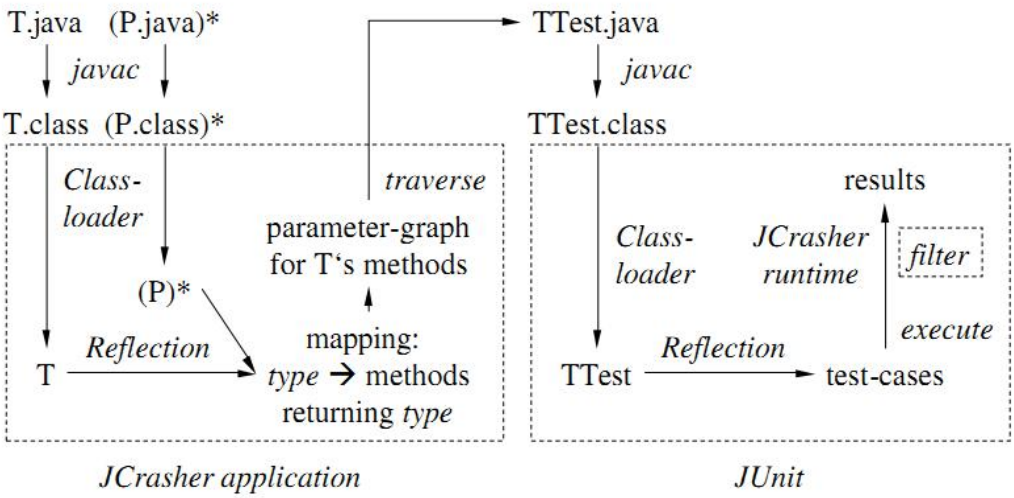



Figure 2.5: Process of robustness testing of Java program with JCrasher [79]

Figure 2.5 illustrate the working of JCrasher by testing a java program namely T.java. The source file is first compiled using javac and the obtained byte code is passed into JCrasher. The JCrasher using Java reflection [12] analyse all the methods declared by class T and by their transitive parameter types P to generate the most appropriate test data set. The test data set is

written to a file TTest.java that is compiled and executed by JUnit. All the exceptions produced during test case executions are collected and compared with robustness heuristic to check for violation. Any violated test case is reported as error.

JCrasher is a pioneer to perform fully automatic testing from test case generation, execution, filtration to reporting the results. One of its novelty is that it generates test case as JUnit files that can be easily read and can be used for regression testing. Another important feature of JCrasher is to execute each new test on a "clean slate" ensuring that the changes made by the previous tests do not affect the new test.

### 2.4.2 Jartege

Jartege (Jawa random test generator) [69] is an automated testing tool that randomly generates unit tests for Java classes with contracts specified in Java Modelling Language (JML). The contracts include methods pre- and post-conditions and class invariants. Initially Jartege uses the contracts to eliminate irrelevant test cases and later on the same contracts serve as test oracle to differentiate between errors and false positives. Jartege uses simple random testing to test classes and generate test cases, however, it facilitate to parametrise its random aspect in order to prioritise testing specific part of the class or to get interesting sequences of calls. These includes:

- To define the operational profile of the classes i.e. the likely use of the class under test by other classes.
- To define the weight of each class and method under test and give test priority to the one's with highest weight and skip those with null weight.
- To control the creation of newly created objects with creation probability functions. Low probability means creation of fewer objects and more re-usability for different operations while high probability means numerous new objects with less re-usability.

The Jartege technique evaluate class by entry pre-condition and internal pre-condition. Entry pre-conditions are the contracts which must be met by the generated test data to test the method while internal pre-conditions are the ones which are inside the methods and their violation are considered error either in method or in the specification. The benefits of Jartege is that it checks for errors in both program and specifications and the Junit tests produced by Jartege can be used later as regression tests. Its minor short coming is that the SUT JML specifications must exist or may be written manually in order to be tested by Jartege.

### 2.4.3 Eclat

Eclat [78] testing tool automatically generates and classify unit tests for Java classes. The process can be divided into three main components. In the first component, it selects small subset of test inputs, likely to reveal faults in the given SUT, from a large set of test inputs.

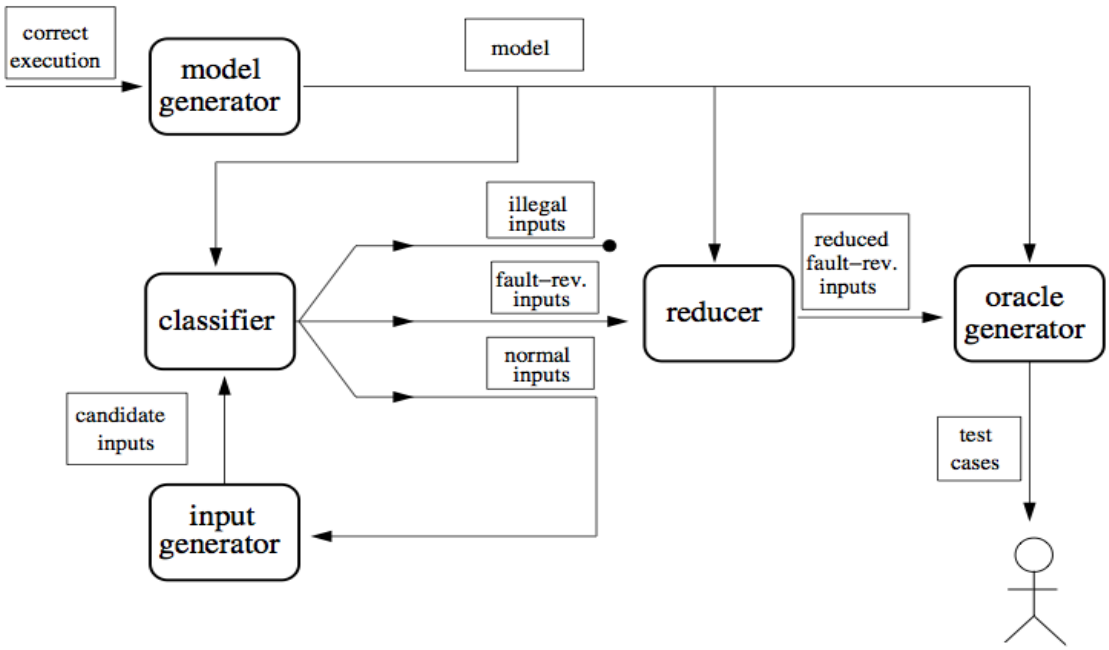



Figure 2.6: Main component of Eclat contributing to generate test input [78]

The tool takes a software and a set of test cases for which the software runs properly. It then creates an operational model based on the correct software operations and apply the test data to it. As a result any inputs whose operational pattern of execution differs from the operational model are (1) likely to reveal fault in the given SUT, (2) Likely to produce normal operations despite violating the model, (3) illegal input that the program is not required to handle. In the second component, reducer function is used to discard any redundant input, leaving only one input per operational pattern.The third and final component facilitate automated testing by converting the acquired test inputs into test cases and creation of oracle to determine whether the test succeeds or fail.

To measure the effectiveness, the researchers tested 9 programs on both Eclat and JCrasher [79]. The experimental results revealed that Eclat outperformed JCrasher. On average, Eclat selected 5.0 inputs per run, and 30% of those revealed a fault. While JCrasher selected 1.13 inputs per

run, and 0.92% of those revealed a fault. The short coming of Eclat is its dependence on the initial pool of correct test cases which is usually written manually and existence of any errors in it can propagate and affect the whole testing process.

### 2.4.4 QuickCheck

QuickCheck [31] is a lightweight random testing tool that can be used for testing of Haskell programs [51]. Haskell is a functional programming language where programs are evaluated using expressions rather than statements as in the case of imperative programming. Since in Haskell most of the functions are pure other than IO functions, therefore it focuses on testing pure functions. These are the functions which depends upon its input parameters and make changes to them only.
QuickCheck takes as input the testers defined properties of the program called Haskell functions and the program to be tested. The tool uses built-in random generator to generate test data or the tester may provide custom define test data generator. Any generated test inputs that satisfies Haskell functions are declared as valid tests input. To cope with oracle problem, the authors have designed a simple domain-specific language of testable specifications embedded in Haskell. The tester use it to define expected properties of the functions under test. QuickCheck then checks and declare a fault in the function where a test case violate these properties.

### 2.4.5 Autotost

Based on Formal Automated testing AutoTest is a tool used for testing of Eiffel programs [27]. The Eiffel language use the concept of contracts (pre-conditions, postconditions and class invariants). Input can be a single class, method or a set of classes which is then processed by AutoTest to generate test cases. It generates both primitive and object type test cases. All the generated test cases are kept in a pool and then randomly a test case is selected from it for execution. A user can set the features of the AutoTest options include: Number of test cases to generate, whether to monitor pre or post condition, order of testing and the initial values of the primitives variables.

### 2.4.6 TestEra

TestEra [53] is a novel framework for testing Java applications. All the tests are produced and executed in an automated fashion. Tests are conducted on the basis of the method specifications [13]. TestEra takes methods specifications, integer value as a limit to the generated test

cases and the method under test. It uses pre-conditions of a method from specifications to automatically generate test cases up to the specified limit. These test cases are then executed on the method and the result is compared against the postconditions (oracle) of that method. Any test case that fails to satisfy postcondition is considered as a fault. The complete error log is displayed in the Graphical User Inteface (GUI).

### 2.4.7 Korat

Korat [8] is a novel framework for automated testing of Java programs based on their formal specifications [14]. As the test start, it uses methods pre-condition to generate all non-isomorphic test cases up to a given size. It then executes each of the test case and compare the obtained results to the methods post-condition, which serves as an oracle to evaluate the correctness of each test case. Korat uses a black-box testing approach where it uses methods pre-conditions as predicates. After the Java predicates and finitization (that bounds the predicates input space) are defined, Korat systematically explore the predicates input space and generate all non isomorphic inputs for which the predicates return true. The core part of Korat monitors execution of the predicates on candidate inputs to filter the inputs based on the fields accessed during executions.

## 2.5 YETI

York Extensible Testing Infrastructure (YETI) is an automated tool for testing Java, JML and .NET assemblies [74]. YETI execute the program under test with random generated but type-correct inputs and declare a fault if the response is an unexpected exception or a contract violation. YETI has been designed with an emphasis on extensibility. Its three main parts: the core infrastructure, strategies and language bindings are loosely coupled to easily accommodate new languages and strategies. To keep the process fully automated YETI uses two approaches for oracle (pass/fail judgement). If available, YETI uses code contracts as oracle if not it uses undeclared runtime exceptions of the underlying language as oracle. The test cases revealing errors are reproduced at the end of each test session for unit and regression testing. Other prominent features of YETI include its Graphical User Interface (GUI) for user friendliness and ability to distribute large testing tasks in cloud for parallel execution [75]. The following sections briefly describe internal working and execution of YETI tool.

### 2.5.1 Construction of Test Cases

YETI construct test cases at random by creating objects of the class under test and randomly calling its methods with inputs according to its parameter's-space. Strategy section contains seven different strategies and inputs to the tested methods is defined by one of the selected strategy. To completely automate the data generation YETI split input values into two types i.e. primitive data types and user defined classes. For Java primitive data types, which includes short, byte, char, int, float, double, long etc., YETI uses its own built-in random value generation library. However, in the case of user defined classes where objects data type is a user defined class YETI calls its constructor to generate object of that class at run time. It may be possible that the constructor require another object and in this case YETI will recursively calls the constructor of that object. This process is continued until the an object with blank constructor, constructor with only primitive types (type 1) or the set level of recursion is reached.

### 2.5.2 Command-line Options

While YETI GUI launcher has been developed during this research study, to take maximum benefit of the available options one still need to launch YETI from CLI mode. These command-line options are case insensitive and can be provided as input to the tool in CLI mode. For example, to save processing power command line option -nologs can be used to bypass real-time logging. The following table describes few of the most common command-line options available in YETI.

### 2.5.3 YETI Execution

YETI being developed in Java is highly portable and can easily run on any operating system with Java Virtual Machine (JVM). YETI can be executed from both command line and GUI. To build and execute YETI, it is necessary to specify the project and all the .jar library files particularly javassist.jar in the CLASSPATH or JVM would not be able to find and execute it. There are several options available as discussed in section xxx??? to accommodate specific needs but the typical command to invoke YETI is given in figure **??**. In this particular command YETI tests java.lang.String and yeti.test.YetiTest modules, for details of other options please see section xxx????.

Alternately, runnable jar file by the name YetiLauncher is also available to launch YETI from GUI. However, till the writing of this thesis, the GUI version of YETI only supports the basic options of YETI. Figure xxx??? shows the equivalent of above command in GUI mode.

Table 2.2: YETI command line options

| Levels | Purpose |
|---|---|
| -java | Test programs coded in Java |
| -jml | Test programs coded in JML |
| -dotnet | Test programs coded in .NET |
| -ea | To check code assertions |
| -nTests | Specify number of tests after which the test stops |
| -time | Specify time in seconds or minutes after which the test stops |
| -testModules | Specify one or more modules to test |
| -rawlogs | Prints real time logs during test |
| -nologs | Omit real time logs and print end result only |
| -yetiPath | Specify path to the test modules |
| -gui | Show test session in GUI |
| -DSSR | Specify Dirt Spot Sweeping Random strategy for this session |
| -ADFD | Specify Automated Discovery of Failure Domain strategy for this session |
| -random | Specify random test strategy for this session |
| -randomPlus | Specify random plus test strategy for this session |
| -randomPlusPeriodic | Specify random plus periodic test strategy for this session |
| -nullProbability | Specify probability of inserting null as input value |
| -newInstanceProability | Specify probability of inserting new object as input value |

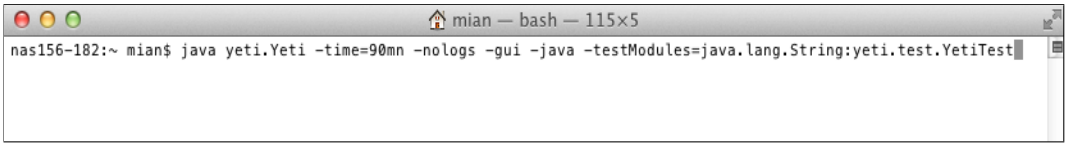



Figure 2.7: Command to launch YETI from CLI

As a result of both the above commands YETI launch its own GUI window and start testing the assigned programs.

### 2.5.4 YETI Report

### 2.5.5 Summary of automated testing tools

## 2.6 Conclusion

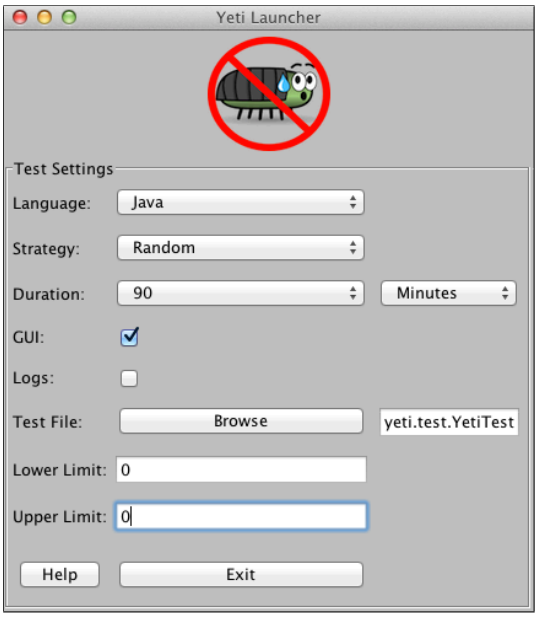



Figure 2.8: Command to launch YETI from GUI

| Tool | Lang | Input | Strategy | Output | Benefit |
|---|---|---|---|---|---|
| QuickCheck | H | Specification & Functions | Specification hold to random TC? | Pass/Fail | Easy to Use, Program Doc |
| Jcrasher | J, JML | Program | Method Type to predict input, Randomly find values of crash | TC | Automated TC, Use of Heuristic Rules |
| Parasoft Jtest | J | Package | Static Analysis of Code & RT | Exceptions & TC | Eclipse plug-in, GUI & Quick |
| Jartage | J | Classes | Random strategy with controls like class weight | TC, RT | Quick, Automated |
| Randoop | J, .N | Specification, Code & Time | Generate then Execute Methods & give Feedback for next generation | Faulty TC, RT | Quick, Easy to use |
| Eclet | J | Classes, Pass TC & candidate inputs | Create model from TC, classify each candidate as Pass/Fail | Faulty TC | Produce output as text, JML |
| AgitarOne | J | Package, Time & Manual TC | Analyze code with auto and provided data in given time | TC, RT | Eclipse plug-in, GUI & Easy to use |
| AutoTest | J | Classes, Time & Manuel tests | Heuristic Rules to evaluate Contracts | voilations, RT | GUI in HTML, Easy to use |
| Korat | J | Specification & Manual TC | Check contracts with specifications | Contracts violations | Give faulty TC |
| TestEra | J | Specifications, Integer & Manuel TC | Check contracts with specifications | Contracts voilations | GUI, give faulty example |
| YETI | J, .N, JML | Code, Time | Random Plus, Pure Random | Traces of found faults | GUI, give faulty example, quick |

Figure 2.9: Summary of automated testing tools

# Chapter 3

# Dirt Spot Sweeping Random Strategy

## 3.1 Introduction

The success of a software testing technique is mainly based on the number of faults it discovers in the Software Under Test (SUT). An efficient testing process discovers the maximum number of faults in a minimum possible time. Exhaustive testing, where software is tested against all possible inputs, is mostly not feasible because of the large size of the input domain, limited resources and strict time constraints. Therefore, strategies in automated software testing tools are developed with the aim to select more fault-finding test input from input domain for a given SUT. Producing such targeted test input is difficult because each system has its own requirements and functionality.

Chan et al. [9] discovered that there are patterns of failure-causing inputs across the input domain. They divided the patterns into point, block and strip patterns on the basis of their occurrence across the input domain. Chen et al. [15] found that the performance of random testing can be increased by slightly altering the technique of test case selection. In adaptive random testing, they found that the performance of random testing increases by up to 50% when test input is selected evenly across the whole input domain. This was mainly attributed to the better distribution of input which increased the chance of selecting inputs from failure patterns. Similarly Restricted Random Testing [10], Feedback directed Random Test Generation [80], Mirror Adaptive Random Testing [16] and Quasi Random Testing [20] stress the need for test case selection covering the whole input domain to get better results.

In this paper we take the assumption that for a significant number of classes failure domains are contiguous or are very close by. From this assumption, we devised the Dirt Spot Sweeping[1]

[1]The name refers to the cleaning robots strategy which insists on places where dirt has been found in large amount.

Random (DSSR) strategy which starts as a random+ strategy — a random strategy focusing more on boundary values. When a new failure is found, it increases the chances of finding more faults using neighbouring values. As in previous studies [72] we approximate faults with unique failures. Since this strategy is an extension of random testing strategy, it has the full potential to find all unique failures in the program, but additionally we expect it to be faster at finding unique failures, for classes in which failure domains are contiguous, as compared with random (R) and random+ (R+) strategies.

We implemented the DSSR strategy in the random testing tool YETI[1]. To evaluate our approach, we tested 30 times each one of the 60 classes of 32 different projects from the Qualitas Corpus[2] with each of the three strategies R, R+ and DSSR. We observed that for 53% of the classes all three strategies find the same unique failures, for remaining 47% DSSR strategy perform up to 33% better than random strategy and up to 17% better than random+ strategy. We also validated the approach by comparing the significance of these results using t-tests and found out that for 7 classes DSSR was significantly better than both R+ and R, for 8 classes DSSR performed similarly to R+ and significantly better than R, while in 2 cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Numerically, the DSSR strategy found 43 more unique failures than R and 12 more unique failures than R+ strategy.

The rest of this paper is organised as follows:
Section 3.2 describes the DSSR strategy. Section 3.3 presents implementation of the DSSR strategy. Section 3.4 explains the experimental setup. Section 3.5 shows results of the experiments. Section 3.6 discusses the results. Section 3.7 presents related work and Section 3.8, concludes the study.

## 3.2 Dirt Spot Sweeping Random Strategy

The new software testing technique named, Dirt Spot Sweeping Random (DSSR) strategy combines the random+ strategy with a dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, faults and unique failures reside in contiguous block and strip pattern. If this is true, DSS increase the performance of the test strategy. Before presenting the details of the DSSR strategy, it is pertinent to review briefly the Random and the Random+ strategy.

[1] http://www.yetitest.org
[2] http://www.qualitascorpus.com

### 3.2.1 Random Strategy (R)

The random strategy is a black-box testing technique in which the SUT is executed using randomly selected test data. Test results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. In the absence of contracts and assertions the exceptions defined by the programming language are used as test oracles. Because of its black-box testing nature, this strategy is particularly effective in testing softwares where the developers want to keep the source code secret [19]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [25; 29]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [28]. YETI [75**?** ], AutoTest [27; 55], QuickCheck [31], Randoop [81], JArtege [69] are some of the most common automated testing tools based on random strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [**?** ] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [27; 37; 38; 48; 67] have proved experimentally that random testing is simple to implement, cost effective, efficient and free from human bias as compared to its rival techniques.

Programs tested at random typically fail a large number of times (there are a large number of calls), therefore, it is necessary to cluster failures that likely represent the same fault. The traditional way of doing it is to compare the full stack traces and error types and use this as an equivalence class [27; 73] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

### 3.2.2 Random Plus Strategy (R+)

The random+ strategy [55] is an extension of the random strategy. It uses some special predefined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [6] are the values on the start and end of a particular type. For instance, such values for `int` could be `MAX_INT`, `MAX_INT-1`, `MAX_INT-2`; `MIN_INT`, `MIN_INT+1`, `MIN_INT+2`. Similarly, the tester might also add some other special values that he considers effective in finding faults for the SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the predefined list of special values. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [29].

### 3.2.3 Dirt Spot Sweeping (DSS)

Chan et al. [9] found that there are patterns of failure-causing inputs across the input domain. Figure 3.1 shows these patterns for two dimensional input domain. They divided these patterns into three types called points, block and strip patterns. The black area (points, block and strip) inside the box show the input which causes the system to fail while white area inside the box represent the genuine input. Boundary of the box (black solid line) surrounds the complete input domain and represents the boundary values. They argue that a strategy has more chances of hitting these fault patterns if test cases far away from each other are selected. Other researchers [10; 16; 20], also tried to generate test cases further away from one another targeting these patterns and achieved better performance. Such increase in performance indicate that faults more often occur contiguous across the input domain. In Dirt Spot Sweeping we propose that if a value reveals fault from the block or strip pattern then for the selection of the next test value, DSS may not look farthest away from the known value and rather pick the closest test value to find another fault from the same region.

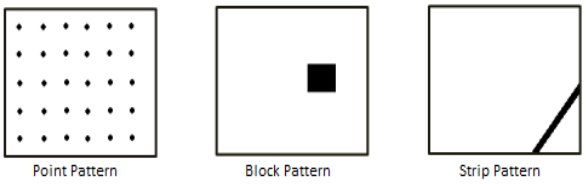



Figure 3.1: Failure patterns across input domain [15]

Dirt spot sweeping is the part of DSSR strategy that comes into action when a failure is found in the system. On finding a failure, it immediately adds the value causing the failure and its neighbouring values to the existing list of interesting values. For example, in a program when the `int` type value of 50 causes a failure in the system then spot sweeping will add values from 47 to 53 to the list of interesting values. If the failure lies in the block or strip pattern, then adding it's neighbouring values will explore other failures present in the block or strip. As against random plus where the list of interesting values remain static, in DSSR strategy the list of interesting values is dynamic and changes during the test execution of each program.

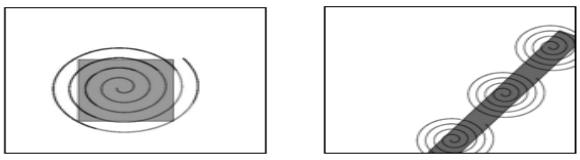


Figure 3.2: DSSR covering block and strip pattern

Figure 3.2 shows how DSS explores the failures residing in the block and strip patterns of a program. The coverage of block and strip pattern is shown in spiral form because first failure leads to second, second to third and so on till the end. In case the failure is positioned on the point pattern then the added values may not be effective because point pattern is only an arbitrary failure point in the whole input domain.

### 3.2.4 Structure of the Dirt Spot Sweeping Random Strategy

The DSSR strategy continuously tracks the number of failures during the execution of the test. This tracking is done in a very effective way with zero or minimum overhead to keep the overhead up to bare minimum [57]. The test execution is started by R+ strategy and continues till a failure is found in the SUT after which the program copies the values leading to the failure as well as the surrounding values to the variable list of interesting values.

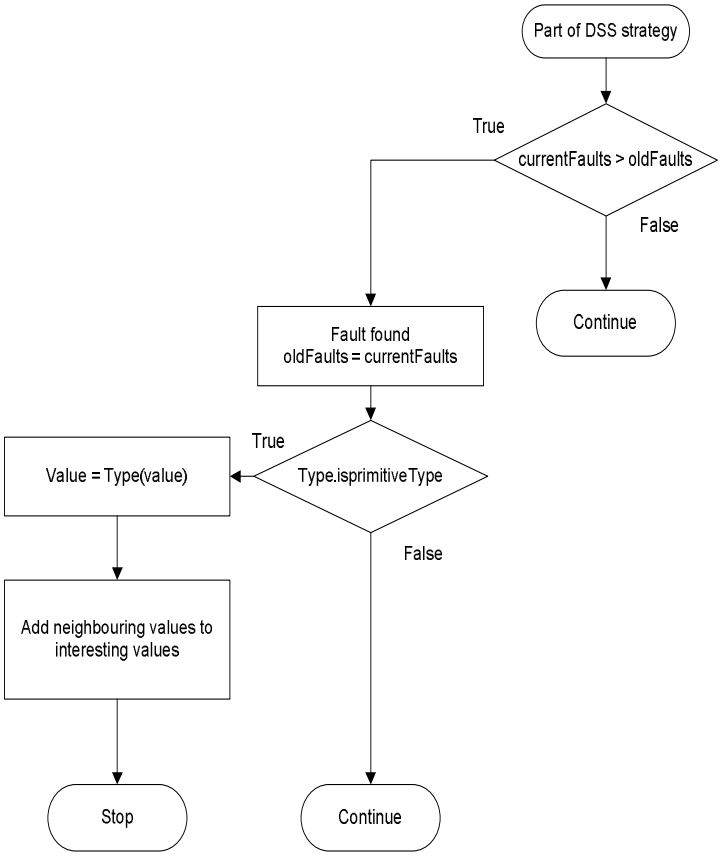



Figure 3.3: Working mechanism of DSSR Strategy

The flowchart presented in Figure 3.3 depicts that, when the failure finding value is of primitive type, the DSSR strategy identifies its type and add values only of that particular type to the list of interesting values. The resultant list of interesting values provide relevant test data for the remaining test session and the generated test cases are more targeted towards finding new failures around the existing failures in the given SUT.

Boundary and other special values that have a high tendency of finding faults in the SUT are added to the list of interesting values by random+ strategy prior to the start of test session where as in DSSR strategy the fault-finding and its surrounding values are added at runtime when a failure is found.

Table 3.1 presents the values are added to the list of interesting values when a failure is found. In the table the test value is represented by X where X can be int, double, float, long, byte, short, char and String. All values are converted to their respective types before adding them to the list of interesting values.

Table 3.1: Neighbouring values for primitive types and String

| Type | Values to be added |
|---|---|
| X is int, double, float, long, byte, short & char | X, X+1, X+2, X-1, X-2 |
| X is String | X<br>X + " "<br>" " + X<br>X.toUpperCase()<br>X.toLowerCase()<br>X.trim()<br>X.substring(2)<br>X.substring(1, X.length()-1) |

### 3.2.5 Explanation of DSSR strategy on a concrete example

The DSSR strategy is explained through a simple program seeded with three faults. The first fault is a division by zero exception denoted by 1 while the second and third faults are failing assertion denoted by 2 and 3 in the given program below followed by description of how the strategy perform execution.

```
/**
* Calculate square of given number
* and verify results.
```

```
* The code contain 3 faults.
* @author (Mian and Manuel)
*/
public class Math1 {
 public void calc (int num1) {
  // Square num1 and store result.
  int result1 = num1 * num1;
  int result2 = result1 / num1; // 1
  assert Math.sqrt(result1) == num1; // 2
  assert result1 >= num1; // 3
 }
}
```

In the above code, one primitive variable of type `int` is used, therefore, the input domain for DSSR strategy is from `-2,147,483,648 to 2,147,483,647`. The strategy further select values (`0, Integer.MIN_VALUE` & `Integer.MAX_VALUE`) as interesting values which are prioritised for selection as inputs. As the test starts, three faults are quickly discovered by DSSR strategy in the following order.

**Fault 1:** The strategy select value `0` for variable `num1` in the first test case because `0` is available in the list of interesting values and therefore its priority is higher than other values. This will cause Java to generate division by zero exception (1).

**Fault 2:** After discovering the first fault, the strategy adds it and its surrounding values to the list of interesting values i.e. `0, 1, 2, 3 and -1, -2, -3` in this case. In the second test case the strategy may pick `-3` as a test value which may lead to the second fault where assertion (2) fails because the square root of `9` is `3` instead of the input value -3.

**Fault 3:** After a few tests the strategy may select `Integer.MAX_VALUE` for variable `num1` from the list of interesting values leading to discovery of the 3rd fault because int variable `result1` will not be able to store the square of `Integer.MAX_VALUE`. Instead of the actual square value Java assigns a negative value (Java language rule) to variable result1 that will lead to the violation of the next assertion (3).

The above process explains that including the border, fault-finding and surrounding values to the list of interesting values in DSSR strategy lead to the available faults quickly and in fewer tests as compared to random and random+ strategy. R and R+ takes more number of tests and time to discover the second and third faults because in these strategies the search for new unique failures starts again randomly in spite of the fact that the remaining faults are very close to the first one.

## 3.3 Implementation of the DSSR strategy

Implementation of the DSSR strategy is made in the YETI open-source automated random testing tool. YETI, coded in Java language, is capable of testing systems developed in procedural, functional and object-oriented languages. Its language-agnostic meta model enables it to test programs written in multiple languages including Java, C#, JML and .Net. The core features of YETI include easy extensibility for future growth, high speed ( up to one million calls per minute on java code), real time logging, real time GUI support, capability to test programs with multiple strategies and auto generation of test report at the end of test session. For large-scale testing there is a cloud-enabled version of YETI, capable of executing parallel test sessions in Cloud [75]. A number of hitherto faults have successfully been found by YETI in various production softwares [71; 73].

YETI can be divided into three decoupled main parts: the core infrastructure, language-specific bindings and strategies. The core infrastructure contains representation for routines, a group of types and a pool of specific type objects. The language specific bindings contain the code to make the call and process the results. The strategies define the procedure of selecting the modules (classes), the routines (methods) and generation of values for instances involved in the routines. By default, YETI uses the random strategy if no particular strategy is defined during test initialisation. It also enables the user to control the probability of using null values and the percentage of newly created objects for each test session. YETI provides an interactive Graphical User Interface (GUI) in which users can see the progress of the current test in real time. In addition to GUI, YETI also provides extensive logs of the test session for more in-depth analysis.

The DSSR strategy is an extension of YetiRandomPlusStrategy, an extended form of the YetiRandomStrategy. The class hierarchy is shown in Figure 3.4.

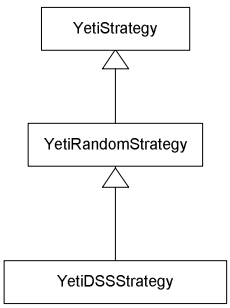



Figure 3.4: Class Hierarchy of DSSR in YETI

## 3.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [55]. General factors such as system software and hardware, YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability have been kept constant in the experiments.

### 3.4.1 Research questions

For evaluating the DSSR strategy, the following research questions have been addressed in this study:

1. Is there an absolute best among R, R+ and DSSR strategies?
2. Are there classes for which any of the three strategies provide better results?
3. Can we pick the best default strategy between R, R+ and DSSR?

### 3.4.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [89]. The Qualitas Corpus is a curated collection of open source java projects built with the aim of helping empirical research on software engineering. These projects have been collected in an organised form containing the source and binary forms. Version 20101126, which contains 106 open source java projects is used in the current evaluation. In our experiments we selected 60 random classes from 32 random projects. All the selected classes produced at least one fault and did not time out with maximum testing session of 10 minutes. Every class is tested thirty times by each strategy (R, R+, DSSR). Name, version and size of the projects to which the classes belong are given in table 3.2 while test details of the classes is presented in table 3.3. Line of Code (LOC) tested per class and its total is shown in column 3 of table 3.3.

Every class is evaluated through $10^5$ calls in each test session.[1] Because of the absence of the contracts and assertions in the code under test, Similar approach as used in previous studies [73] is followed using undeclared exceptions to compute unique failures.

All tests are performed with a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon processor with 6 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java™SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

[1]The total number of tests is thus $60 \times 30 \times 3 \times 10^5 = 540 \times 10^6\ tests$.

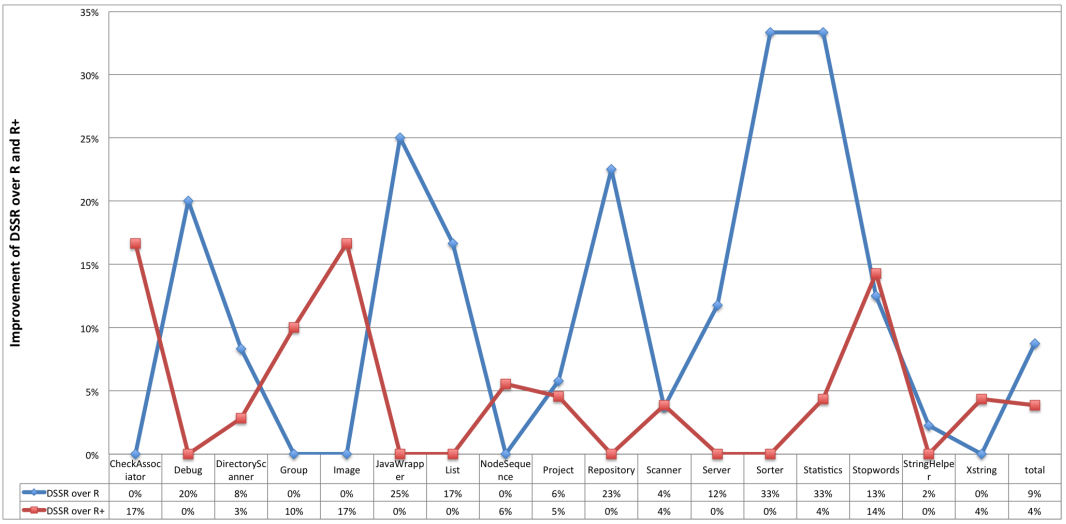
Table 3.2: Name and versions of 32 Projects randomly selected from the Qualitas Corpus for the experiments

| S. No | Project Name | Version | Size (MB) |
|---|---|---|---|
| 1 | apache-ant | 1.8.1 | 59 |
| 2 | antlr | 3.2 | 13 |
| 3 | aoi | 2.8.1 | 35 |
| 4 | argouml | 0.30.2 | 112 |
| 5 | artofillusion | 281 | 5.4 |
| 6 | aspectj | 1.6.9 | 109.6 |
| 7 | axion | 1.0-M2 | 13.3 |
| 8 | azureus | 1 | 99.3 |
| 9 | castor | 1.3.1 | 63.2 |
| 10 | cayenne | 3.0.1 | 4.1 |
| 11 | cobertura | 1.9.4.1 | 26.5 |
| 12 | colt | 1.2.0 | 40 |
| 13 | emma | 2.0.5312 | 7.4 |
| 14 | freecs | 1.3.20100406 | 11.4 |
| 15 | hibernate | 3.6.0 | 733 |
| 16 | hsqldb | 2.0.0 | 53.9 |
| 17 | itext | 5.0.3 | 16.2 |
| 18 | jasml | 0.10 | 7.5 |
| 19 | jmoney | 0.4.4 | 5.3 |
| 20 | jruby | 1.5.2 | 140.7 |
| 21 | jsXe | 04_beta | 19.9 |
| 22 | quartz | 1.8.3 | 20.4 |
| 23 | sandmark | 3.4 | 18.8 |
| 24 | squirrel-sql | 3.1.2 | 61.5 |
| 25 | tapestry | 5.1.0.5 | 69.2 |
| 26 | tomcat | 7.0.2 | 24.1 |
| 27 | trove | 2.1.0 | 18.2 |
| 28 | velocity | 1.6.4 | 27.1 |
| 29 | weka | 3.7.2 | 107 |
| 30 | xalan | 2.7.1 | 85.4 |
| 31 | xerces | 2.10.0 | 43.4 |
| 32 | xmojo | 5.0.0 | 15 |

### 3.4.3 Performance measurement criteria

Various measures including the E-measure (expected number of failures detected), P-measure (probability of detecting at least one failure) and F-measure (number of test cases used to find the first fault) have been used by researchers to find the effectiveness of the random test strategy. The E-measure and P-measure have been heavily criticised [15] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [18; 23]. In our initial experiments the F-measure is used to evaluate the efficiency. However it was realised that this is not the right choice. In some experiments a strategy found the first fault quickly than the other but on completion of test session that very strategy found lower number of total faults than the rival strategy. The preference given to a strategy by F-measure because it finds the first fault quickly without giving due consideration to the total number of faults is not fair [60].

The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered faults in one go therefore, F-measure is not the favourable choice. In our experiments, performance of the strategy is measured by the maximum number of faults detected in SUT by a particular number of test calls [27; 30; 80]. This measurement is effective because it considers the performance of the strategy when all other factors are kept constant.




| | CheckAssociator | Debug | DirectoryScanner | Group | Image | JavaWrapper | List | NodeSequence | Project | Repository | Scanner | Server | Sorter | Statistics | Stopwords | StringHelper | Xstring | total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| DSSR over R | 0% | 20% | 8% | 0% | 0% | 25% | 17% | 0% | 6% | 23% | 4% | 12% | 33% | 33% | 13% | 2% | 0% | 9% |
| DSSR over R+ | 17% | 0% | 3% | 10% | 17% | 0% | 0% | 6% | 5% | 0% | 4% | 0% | 0% | 4% | 14% | 0% | 4% | 4% |

Figure 3.5: Improvement of DSSR strategy over Random and Random+ strategy.

Table 3.3: Experiments result presenting Serial Number (S.No), Class Name, Line of Code (LOC), mean, maximum and minimum number of faults and relative standard deviation for each Random (R), Random+ (R+) and Dirt Spot Sweeping Random (DSSR) strategies.

| S. No | Class Name | LOC | R | | | | R+ | | | | DSSR | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Mean | Max | Min | R-STD | Mean | Max | Min | R-STD | Mean | Max | Min | R-STD |
| 1 | ActionTranslator | 709 | 96 | 96 | 96 | 0 | 96 | 96 | 96 | 0 | 96 | 96 | 96 | 0 |
| 2 | AjTypeImpl | 1180 | 80 | 83 | 79 | 0.02 | 80 | 83 | 79 | 0.02 | 80 | 83 | 79 | 0.01 |
| **3** | **Apriori** | **292** | **3** | **4** | **3** | **0.10** | **3** | **4** | **3** | **0.13** | **3** | **4** | **3** | **0.14** |
| 4 | BitSet | 575 | 9 | 9 | 9 | 0 | 9 | 9 | 9 | 0 | 9 | 9 | 9 | 0 |
| 5 | CatalogManager | 538 | 7 | 7 | 7 | 0 | 7 | 7 | 7 | 0 | 7 | 7 | 7 | 0 |
| **6** | **CheckAssociator** | **351** | **7** | **8** | **2** | **0.16** | **6** | **9** | **2** | **0.18** | **7** | **9** | **6** | **0.73** |
| **7** | **Debug** | **836** | **4** | **6** | **4** | **0.13** | **5** | **6** | **4** | **0.12** | **5** | **8** | **4** | **0.19** |
| **8** | **DirectoryScanner** | **1714** | **33** | **39** | **20** | **0.10** | **35** | **38** | **31** | **0.05** | **36** | **39** | **32** | **0.04** |
| 9 | DiskIO | 220 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 |
| 10 | DOMParser | 92 | 7 | 7 | 3 | 0.19 | 7 | 7 | 3 | 0.11 | 7 | 7 | 7 | 0 |
| 11 | Entities | 328 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 |
| 12 | EntryDecoder | 675 | 8 | 9 | 7 | 0.10 | 8 | 9 | 7 | 0.10 | 8 | 9 | 7 | 0.08 |
| 13 | EntryComparator | 163 | 13 | 13 | 13 | 0 | 13 | 13 | 13 | 0 | 13 | 13 | 13 | 0 |
| 14 | Entry | 37 | 6 | 6 | 6 | 0 | 6 | 6 | 6 | 0 | 6 | 6 | 6 | 0 |
| 15 | Facade | 3301 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 |
| 16 | FileUtil | 83 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 |
| 17 | Font | 184 | 12 | 12 | 11 | 0.03 | 12 | 12 | 11 | 0.03 | 12 | 12 | 11 | 0.02 |
| 18 | FPGrowth | 435 | 5 | 5 | 5 | 0 | 5 | 5 | 5 | 0 | 5 | 5 | 5 | 0 |
| 19 | Generator | 218 | 17 | 17 | 17 | 0 | 17 | 17 | 17 | 0 | 17 | 17 | 17 | 0 |
| **20** | **Group** | **88** | **11** | **11** | **10** | **0.02** | **10** | **4** | **11** | **0.15** | **11** | **11** | **11** | **0** |
| 21 | HttpAuth | 221 | 2 | 2 | 2 | 0 | 2 | 2 | 2 | 0 | 2 | 2 | 2 | 0 |
| **22** | **Image** | **2146** | **13** | **17** | **7** | **0.15** | **12** | **14** | **4** | **0.15** | **14** | **16** | **11** | **0.07** |
| 23 | InstrumentTask | 71 | 2 | 2 | 1 | 0.13 | 2 | 2 | 1 | 0.09 | 2 | 2 | 2 | 0 |
| 24 | IntStack | 313 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 |
| 25 | ItemSet | 234 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 |
| 26 | Itextpdf | 245 | 8 | 8 | 8 | 0 | 8 | 8 | 8 | 0 | 8 | 8 | 8 | 0 |
| **27** | **JavaWrapper** | **513** | **3** | **2** | **2** | **0.23** | **4** | **4** | **3** | **0.06** | **4** | **4** | **3** | **0.05** |
| 28 | JmxUtilities | 645 | 8 | 8 | 6 | 0.07 | 8 | 8 | 7 | 0.04 | 8 | 8 | 7 | 0.04 |
| **29** | **List** | **1718** | **5** | **6** | **4** | **0.11** | **6** | **6** | **4** | **0.10** | **6** | **6** | **5** | **0.09** |
| 30 | NameEntry | 172 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 |
| **31** | **NodeSequence** | **68** | **38** | **46** | **30** | **0.10** | **36** | **45** | **30** | **0.12** | **38** | **45** | **30** | **0.08** |
| 32 | NodeSet | 208 | 28 | 29 | 26 | 0.03 | 28 | 29 | 26 | 0.04 | 28 | 29 | 26 | 0.03 |
| 33 | PersistentBag | 571 | 68 | 68 | 68 | 0 | 68 | 68 | 68 | 0 | 68 | 68 | 68 | 0 |
| 34 | PersistentList | 602 | 65 | 65 | 65 | 0 | 65 | 65 | 65 | 0 | 65 | 65 | 65 | 0 |
| 35 | PersistentSet | 162 | 36 | 36 | 36 | 0 | 36 | 36 | 36 | 0 | 36 | 36 | 36 | 0 |
| **36** | **Project** | **470** | **65** | **71** | **60** | **0.04** | **66** | **78** | **62** | **0.04** | **69** | **78** | **64** | **0.05** |
| **37** | **Repository** | **63** | **31** | **31** | **31** | **0** | **40** | **40** | **40** | **0** | **40** | **40** | **40** | **0** |
| 38 | Routine | 1069 | 7 | 7 | 7 | 0 | 7 | 7 | 7 | 0 | 7 | 7 | 7 | 0 |
| 39 | RubyBigDecimal | 1564 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 | 4 | 4 | 4 | 0 |
| 40 | Scanner | 94 | 3 | 5 | 2 | 0.20 | 3 | 5 | 2 | 0.27 | 3 | 5 | 2 | 0.25 |
| **41** | **Scene** | **1603** | **26** | **27** | **26** | **0.02** | **26** | **27** | **26** | **0.02** | **27** | **27** | **26** | **0.01** |
| 42 | SelectionManager | 431 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 |
| **43** | **Server** | **279** | **15** | **21** | **11** | **0.20** | **17** | **21** | **12** | **0.16** | **17** | **21** | **12** | **0.14** |
| **44** | **Sorter** | **47** | **2** | **2** | **1** | **0.09** | **3** | **3** | **2** | **0.06** | **3** | **3** | **3** | **0** |
| 45 | Sorting | 762 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 |
| **46** | **Statistics** | **491** | **16** | **17** | **12** | **0.08** | **23** | **25** | **22** | **0.03** | **24** | **25** | **22** | **0.04** |
| 47 | Status | 32 | 53 | 53 | 53 | 0 | 53 | 53 | 53 | 0 | 53 | 53 | 53 | 0 |
| **48** | **Stopwords** | **332** | **7** | **8** | **7** | **0.03** | **7** | **8** | **6** | **0.08** | **8** | **8** | **7** | **0.06** |
| **49** | **StringHelper** | **178** | **43** | **45** | **40** | **0.02** | **44** | **46** | **42** | **0.02** | **44** | **45** | **42** | **0.02** |
| 50 | StringUtils | 119 | 19 | 19 | 19 | 0 | 19 | 19 | 19 | 0 | 19 | 19 | 19 | 0 |
| 51 | TouchCollector | 222 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 | 3 | 3 | 3 | 0 |
| 52 | Trie | 460 | 21 | 22 | 21 | 0.02 | 21 | 22 | 21 | 0.01 | 21 | 22 | 21 | 0.01 |
| 53 | URI | 3970 | 5 | 5 | 5 | 0 | 5 | 5 | 5 | 0 | 5 | 5 | 5 | 0 |
| 54 | WebMacro | 311 | 5 | 5 | 5 | 0 | 5 | 6 | 5 | 0.14 | 5 | 7 | 5 | 0.28 |
| 55 | XMLAttributesImpl | 277 | 8 | 8 | 8 | 0 | 8 | 8 | 8 | 0 | 8 | 8 | 8 | 0 |
| 56 | XMLChar | 1031 | 13 | 13 | 13 | 0 | 13 | 13 | 13 | 0 | 13 | 13 | 13 | 0 |
| 57 | XMLEntityManger | 763 | 17 | 18 | 17 | 0.01 | 17 | 17 | 16 | 0.01 | 17 | 17 | 17 | 0 |
| 58 | XMLEntityScanner | 445 | 12 | 12 | 12 | 0 | 12 | 12 | 12 | 0 | 12 | 12 | 12 | 0 |
| 59 | XObject | 318 | 19 | 19 | 19 | 0 | 19 | 19 | 19 | 0 | 19 | 19 | 19 | 0 |
| **60** | **XString** | **546** | **23** | **24** | **21** | **0.04** | **23** | **24** | **23** | **0.02** | **24** | **24** | **23** | **0.02** |
| | **Total** | 35,785 | 1040 | 1075 | 973 | 2.42 | 1061 | 1106 | 1009 | 2.35 | 1075 | 1118 | 1032 | 1.82 |

## 3.5 Results

Results of the experiments including class name, Line of Code (LOC), mean value, maximum and minimum number of unique failures and relative standard deviation for each of the 60 classes tested using R, R+ and DSSR strategy are presented in Table 3.3. Each strategy found an equal number of faults in 31 classes while in the remaining 29 classes the three strategies performed differently from one another. The total of mean values of unique failures in DSSR (1075) is higher than for R (1040) or R+ (1061) strategies. DSSR also finds a higher number of maximum unique failures (1118) than both R (1075), and R+ (1106). DSSR strategy finds 43 and 12 more unique faults compared to R and R+ respectively. The minimum number of unique faults found by DSSR (1032) is also higher than for R (973) and R+ (1009) which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

### 3.5.1 Is there an absolute best among R, R+ and DSSR strategies?

Based on our findings DSSR is at least as good as R and R+ in almost all cases, it is also significantly better than both R and R+ in 12% of the classes. Figure 3.5 presents the average improvements of DSSR strategy over R and R+ strategy over the 17 classes for which there is a significant difference between DSSR and R or R+. The blue line with diamond symbol shows performance of DSSR over R and the red line with square symbols depicts the improvement of DSSR over R+ strategy. The classes where blue line with diamond symbols show the improvement of DSSR over R and red line with square symbols show the improvement of DSSR over R+.

The improvement of DSSR over R and R+ strategy is calculated by applying the formula (1) and (2) respectively.

$$\frac{Average faults_{(DSSR)} - Average faults_{(R)}}{Average faults_{(R)}} * 100 \tag{3.1}$$

$$\frac{Average faults_{(DSSR)} - Average faults_{(R+)}}{Average faults_{(R+)}} * 100 \tag{3.2}$$

The findings show that DSSR strategy perform up to 33% better than R and up to 17% better than R+ strategy. In some cases DSSR perform equally well with R and R+ but in no case DSSR performed lower than R and R+. Based on the results it can be stated that DSSR strategy is a better choice than R and R+ strategy.

Table 3.4: T-test results of the classes

| S. No | Class Name | T-test Results | | | Interpretation |
|---|---|---|---|---|---|
| | | DSSR, R | DSSR, R+ | R, R+ | |
| 1 | AjTypeImpl | 1 | 1 | 1 | |
| 2 | Apriori | **0.03** | 0.49 | 0.16 | |
| 3 | CheckAssociator | **0.04** | **0.05** | 0.44 | DSSR better |
| 4 | Debug | **0.03** | 0.14 | 0.56 | |
| 5 | DirectoryScanner | **0.04** | **0.01** | 0.43 | DSSR better |
| 6 | DomParser | **0.05** | 0.23 | 0.13 | |
| 7 | EntityDecoder | **0.04** | 0.28 | 0.3 | |
| 8 | Font | 0.18 | 0.18 | 1 | |
| 9 | Group | 0.33 | **0.03** | **0.04** | DSSR = R >R+ |
| 10 | Image | **0.03** | **0.01** | 0.61 | DSSR better |
| 11 | InstrumentTask | 0.16 | 0.33 | 0.57 | |
| 12 | JavaWrapper | **0.001** | 0.57 | 0.004 | DSSR = R+ >R |
| 13 | JmxUtilities | 0.13 | 0.71 | 0.08 | |
| 14 | List | **0.01** | 0.25 | **0** | DSSR = R+ >R |
| 15 | NodeSequence | 0.97 | **0.04** | **0.06** | DSSR = R >R+ |
| 16 | NodeSet | **0.03** | 0.42 | 0.26 | |
| 17 | Project | **0.001** | 0.57 | **0.004** | DSSR better |
| 18 | Repository | **0** | 1 | **0** | DSSR = R+ >R |
| 19 | Scanner | 1 | **0.03** | **0.01** | DSSR better |
| 20 | Scene | **0** | **0** | 1 | DSSR better |
| 21 | Server | **0.03** | 0.88 | **0.03** | DSSR = R+ >R |
| 22 | Sorter | **0** | 0.33 | **0** | DSSR = R+ >R |
| 23 | Statistics | **0** | 0.43 | **0** | DSSR = R+ >R |
| 24 | Stopwords | **0** | 0.23 | **0** | DSSR = R+ >R |
| 25 | StringHelper | **0.03** | 0.44 | 0.44 | DSSR = R+ >R |
| 26 | Trie | 0.1 | 0.33 | 0.47 | DSSR better |
| 27 | WebMacro | 0.33 | 1 | 0.16 | |
| 28 | XMLEntityManager | 0.33 | 0.33 | 0.16 | |
| 29 | XString | 0.14 | **0.03** | 0.86 | |

### 3.5.2 Are there classes for which any of the three strategies provide better results?

T-tests applied to the data given in Table 3.4 show that DSSR is significantly better in 7 classes from R and R+ strategy, in 8 classes DSSR performed similarly to R+ but significantly higher than R, and in 2 classes DSSR performed similarly to R but significantly higher than R+. There is no case R and R+ strategy performed significantly better than DSSR strategy. Expressed in percentage: 72% of the classes do not show significantly different behaviours whereas in 28% of hte classes, the DSSR strategy performs significantly better than at least one of R and R+. It is interesting to note that in no single case R and R+ strategies performed better than DSSR strategy. We attribute this to DSSR possessing the qualities of R and R+ whereas containing the spot sweeping feature.

### 3.5.3 Can we pick the best default strategy between R, R+ and DSSR?

Analysis of the experimental data reveal that DSSR strategy has an edge over R and R+. This is because of the additional feature of Spot Sweeping in DSSR strategy.

In spite of the better performance of DSSR strategy compared to R and R+ strategies the present study does not provide ample evidence to pick it as the best default strategy because of the overhead induced by this strategy (see next section). Further study might give conclusive evidence.

## 3.6 Discussion

In this section we discuss various factors such as the time taken, effect of test duration, number of tests, number of faults in the different strategies and the effect of finding first fault in the DSSR strategy. **Time taken to execute an equal number of test cases:** The DSSR strategy takes slightly more time (up to 5%) than both pure random and random plus which may be due to maintaining sets of interesting values during the execution. We do not believe that the overhead can be reduced.

**Effect of test duration and number of tests on the results:** All three techniques have the same potential for finding failures. If testing is continued for a long duration then all three strategies will find the same number of unique failures and the results will converge. We suspect however that some of the unique failures will take an extremely long time to be found by using random or random+ only. Further experiments should confirm this point.

**Effect of number of faults on results:** We found that the DSSR strategy performs better when the number of faults is higher in the code. The reason seems to be that when there are

more faults, their domains are more connected and DSSR strategy works better. Further studies might use historical data to pick the best strategy.

**Dependence of DSSR strategy to find the first unique failure early enough:** During the experiments we noticed that if a unique failure is not found quickly enough, there is no value added to the list of interesting values and then the test becomes equivalent to random+ testing. This means that better ways of populating failure-inducing values are needed for sufficient leverage to DSSR strategy. As an example, the following piece of code would be unlikely to fail under the current setting:

```
public void test(float value){
 if(value == 34.4445)   10/0;
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained from the constant pool in the class files of the SUT.

In the example above the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the unique failure right away.

**DSSR strategy and coverage:** Random strategies typically achieve high level of coverage [75]. It might also be interesting to compare R, R+ and DSSR with respect to the achieved coverage or even to use a DSSR variant that adds a new interesting value and its neighbours when a new branch is reached.

**Threats to validity:** As usual with such empirical studies, the present work might suffer from a non-representative selection of classes. The selection in the current study is however made through random process and objective criteria, therefore, it seems likely that it would be representative.

The parameters of the study might also have prompted incorrect results. But this is unlikely due to previous results on random testing [73].

## 3.7 Related Work

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [31; 35; 78]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [48]. Some of the well known automated tools based on random strategy includes Jartege [69], Eclat [78], JCrasher [35], AutoTest [27; 28] and YETI [73; 75].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [10; 16; 19; 20; 21]. Adaptive random testing (ART), Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror ART and ART through dynamic partitioning increased the performance by reducing the overhead of ART. The main reason behind better performance of the strategies is that even spread of test input increases the chance of exploring the fault patterns present in the input domain.

A more recent research study [98] stresses on the effectiveness of data regeneration in close vicinity of the existing test data. Their findings showed up to two orders of magnitude more efficient test data generation than the existing techniques. Two major limitations of their study are the requirement of existing test cases to regenerate new test cases, and increased overhead due to "meta heuristics search" based on hill climbing algorithm to regenerate new data. In DSSR no pre-existing test cases are required because it utilises the border values from R+ and regenerate the data very cheaply in a dynamic fashion different for each class under test without any prior test data and with comparatively lower overhead.

The random+ (R+) strategy is an extension of the random strategy in which interesting values, beside pure random values, are added to the list of test inputs [55]. These interesting values includes border values which have high tendency of finding faults in the given SUT [6]. Results obtained with R+ strategy show significant improvement over random strategy [55]. DSSR strategy is an extension of R+ strategy which starts testing as R+ until a fault is found then it switches to spot sweeping.

A common practice to evaluate performance of an extended strategy is to compare the results obtained by applying the new and existing strategy to identical programs [38; 46; 47]. Arcuri et al. [4], stress on the use of random testing as a baseline for comparison with other test strategies. We followed the procedure and evaluated DSSR strategy against R and R+ strategies under identical conditions.

In our experiments we selected projects from the Qualitas Corpus [88] which is a collection of open source java programs maintained for independent empirical research. The projects in Qualitas Corpus are carefully selected that spans across the whole set of java applications [73; 87; 89].

## 3.8 Conclusions

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases. We developed a new strategy named. "DSSR strategy" as an extension of R+, based on the assumption that in a significant number of classes,

failure domains are contiguous or located closely. The DSS strategy, a strategy which adds neighbouring values of the failure finding value to a list of interesting values, was implemented in the random testing tool YETI to test 60 classes, 30 times each, from Qualitas Corpus with each of the 3 strategies R, R+ and DSSR. The newly developed DSSR strategy uncovers more unique failures than both random and random+ strategies with a 5% overhead. We found out that for 7 (12%) classes DSSR was significantly better than both R+ and R, for 8 (13%) classes DSSR performed similarly to R+ and significantly better than R, while in 2 (3%) cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Overall, DSSR yields encouraging results and advocates to develop the technique further for settings in which it is significantly better than both R and R+ strategies.

# Chapter 4

# Automated Disovery of Failure Domain

## 4.1 Introduction

Testing is fundamental requirement to assess the quality of any software. Manual testing is labour-intensive and error-prone; therefore emphasis is to use automated testing that significantly reduces the cost of software development process and its maintenance [7]. Most of the modern black-box testing techniques execute the System Under Test (SUT) with specific input and compare the obtained results against the test oracle. A report is generated at the end of each test session containing any discovered faults and the input values which triggers the faults. Debuggers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality, set in test plan, is achieved.

The fact that exhaustive testing for any non-trivial program is impossible, compels the testers to come up with some strategy of input selection from the whole input domain. Pure random is one of the possible strategies widely used in automated tools. It is intuitively simple and easy to implement [29], [41]. It involves minimum or no overhead in input selection and lacks human bias [48], [59]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [68] and discovery of lower number of faults [22]. To overcome these limitations while keeping its benefits intact many researchers successfully refined pure random testing. Adaptive Random Testing (ART) is the most significant refinements of random testing. Experiments performed using ART showed up to 50% better results compared to the traditional/pure random testing [15]. Similarly Restricted Random Testing (RRT) [10], Mirror Adaptive Random Testing (MART) [18], Adaptive Random Test-

ing for Object Oriented Programs (ARTOO) [29], Directed Adaptive Random Testing (DART) [45], Lattice-based Adaptive Random Testing (LART) [61] and Feedback-directed Random Testing (FRT) [81] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

All the above-mentioned variations in random testing are based on the observation of Chan et. al., [9] that failure causing inputs across the whole input domain form certain kinds of domains. They classified these domains into point, block and strip fault domain. In Figure 4.1 the square box represents the whole input domain. The black point, block and strip area inside the box represent the faulty values while white area inside the box represent legitimate values for a specific system. They further suggested that the fault finding ability of testing could be improved by taking into consideration these failure domains.

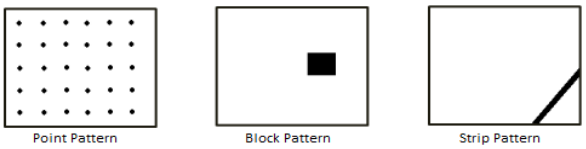



Figure 4.1: Failure domains across input domain [9]

It is interesting that where many random strategies are based on the principle of contiguous fault domains inside the input domain, no specific strategy is developed to evaluate these fault domains. This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD), which not only finds the pass and fail input values but also finds their domains. The idea of identification of pass and fail domain is attractive as it provides an insight of the domains in the given SUT. Some important aspects of ADFD strategy presented in the paper include:

- Implementation of the new ADFD strategy in York Extensible Testing Infrastructure (YETI) tool.
- Evaluation to assess ADFD strategy by testing classes with different fault domains.
- Decrease in overall test duration by identification of all the fault domains instead of a single instance of fault.
- Increase in test efficiency by helping debugger to keep in view all the fault occurrences when debugging.

The rest of this paper is organized as follows:
Section 4.2 describes the ADFD strategy. Section 4.3 presents implementation of the ADFD

strategy. Section 4.4 explains the experimental results. Section 4.5 discusses the results. Section 4.6 presents the threats to validity. Section 4.7 presents related work and Section 4.8, concludes the study.

## 4.2 Automated Discovery of Failure Domain

Automated Discovery of Failure Domain (ADFD) strategy is proposed as improvement on R+ strategy with capability of finding faults as well as the fault domains. The output produced at the end of test session is a chart showing the passing value or range of values in green and failing value or range of values in red. The complete workflow of ADFD strategy is given in Figure 4.2.

The process is divided into five major steps given below and each step is briefly explained in the following paras.

1. GUI front-end for providing input
2. Automated finding of fault
3. Automated generation of modules
4. Automated compilation and execution of modules to discover domains
5. Automated generation of graph showing domains

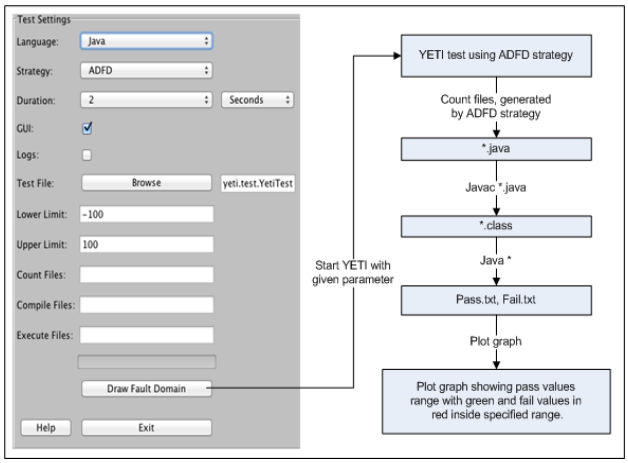



Figure 4.2: Work flow of ADFD strategy

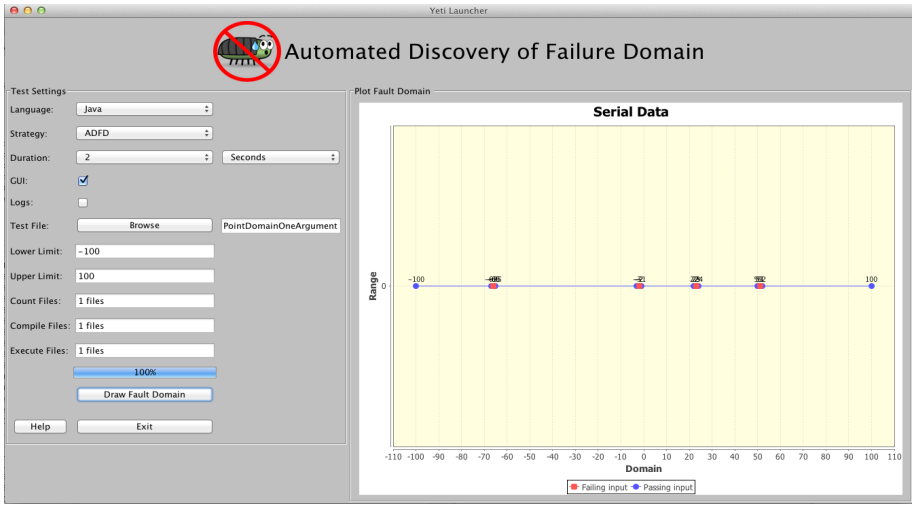



Figure 4.3: Front-end of ADFD strategy

**GUI front-end for providing input:**

ADFD strategy is provided with an easy to use GUI front-end to get input from the user. It takes YETI specific input including language of the program, strategy, duration, enable or disable YETI GUI, logs and a program to test in the form of java byte code. In addition it also takes minimum and maximum values to search for fault domain in the specified range. Default range for minimum and maximum is Integer.MIN_INT and Integer.MAX_INT respectively.

**Automated finding of fault:**

To find the failure domain for a specific fault, the first requirement is to identify that fault in the system. ADFD strategy extends R+ strategy and rely on R+ strategy to find the first fault. Random+ (R+) is an improvement over random strategy with preference to the boundary values to provide better fault finding ability. ADFD strategy is implemented in YETI tool which is famous for its simplicity, high speed and proven ability of finding potentially hazardous faults in many systems [71], [73]. YETI is quick and can call up to one million instructions in one second on Java code. It is also capable of testing VB.Net, C, JML and CoFoJa beside Java.

**Automated generation of modules:**

After a fault is found in the SUT, ADFD strategy generate complete new Java program to search for fault domains in the given SUT. These programs with ".java" extensions are generated through dynamic compiler API included in Java 6 under javax.tools package. The number of programs generated can be one or more, depending on the number of arguments in the test

module i.e. for module with one argument one program is generated, for two argument two programs and so on. To track fault domain the program keeps one or more than one argument constant and only one argument variable in the generated program.

**Automated compilation and execution of modules to discover domains:**

The java modules generated in previous step are compiled using "javac *" command to get their binary ".class" files. The "java *" command is applied to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument receive all the values in range, from minimum to maximum, specified in the beginning of the test. After execution is completed we get two text files of "Pass.txt" and "Fail.txt". Pass file contains all the values for which the modules behave correctly while fail file contains all the values for which the modules fail.

**Automated generation of graph showing domains:**

The values from the pass and fail files are used to plot (x, y) chart using JFreeChart. JFreeChart is a free open-source java library that helps developers to display complex charts and graphs in their applications [44]. Green colour lines with circle represents pass values while red colour line with squares represents the fail values. Resultant graph clearly depicts both the pass and fail domain across the specified input domain. The graph shows red points in case the program fails for only one value, blocks when the program fails for multiple values and strips when a program fails for a long range of values.

## 4.3 Implementation

The ADFD strategy is implemented in a tool called York Extensible Testing Infrastructure (YETI). YETI is available in open-source at `http://code.google.com/p/yeti-test/`. In this section a brief overview of YETI is given with the focus on the parts relevant to the implementation of ADFD strategy. For integration of ADFD strategy in YETI, a program is used as an example to illustrate the working of ADFD strategy. Please refer to [71], [73], [75], [70] for more details on YETI tool.

### 4.3.1 York Extensible Testing Infrastructure

YETI is a testing tool developed in Java that test programs using random strategies in an automated fashion. YETI meta-model is language-agnostic which enables it to test programs written in functional, procedural and object-oriented languages.

YETI consists of three main parts including core infrastructure for extendibility through specialisation, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both the languages and strategies sections have a pluggable architecture to easily incorporate new strategies and languages making YETI a favourable choice to implement ADFD strategy. YETI is also capable of generating test cases to reproduce the faults found during the test session.

### 4.3.2 ADFD strategy in YETI

The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. On top of the hierarchy in strategies, is an abstract class YetiStrategy, which is extended by YetiRandomPlusStrategy and it is further extended to get ADFD strategy.

### 4.3.3 Example

For a concrete example to show how ADFD strategy in YETI proceeds, we suppose YETI tests the following class with ADFD strategy selected for testing. Note that for more clear visibility of the output graph generated by ADFD strategy at the end of test session, we fix the values of lower and upper range by 70 from Integer.MIN_INT and Integer.MAX_INT.

```
/**
 * Point Fault Domain example for one argument
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{
    public static void pointErrors (int x){
            if (x == -66)
                abort();

            if (x == -2)
                abort();

            if (x == 51)
                abort();

            if (x == 23)
                abort();
```

```
        }
}
```

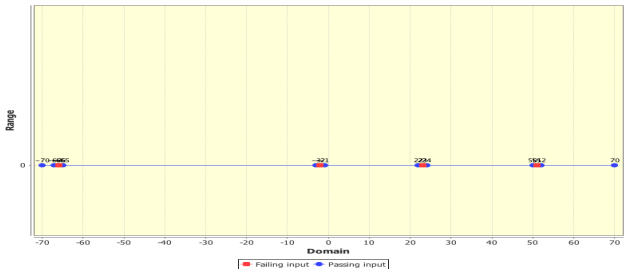



Figure 4.4: ADFD strategy plotting pass and fault domain of the given class

As soon as any one of the above four faults are discovered the ADFD strategy generate a dynamic program given in Appendix **??** (1). This program is automatically compiled to get binary file and then executed to find the pass and fail domains inside the specified range. The identified domains are plotted on two-dimensional graph. It is evident from the output presented in Figure **??** that ADFD strategy not only finds all the faults but also the pass and fail domains.

## 4.4 Experimental Results

This section includes the experimental setup and results obtained after using ADFD strategy. Six numerical programs of one and two-dimension were selected. These programs were error-seeded in such a way to get all the three forms of fault domains including point, block and strip fault domains. Each selected program contained various combinations of one or more fault domains.

All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java$^{TM}$SE Runtime Environment [version 1.6.0_35].

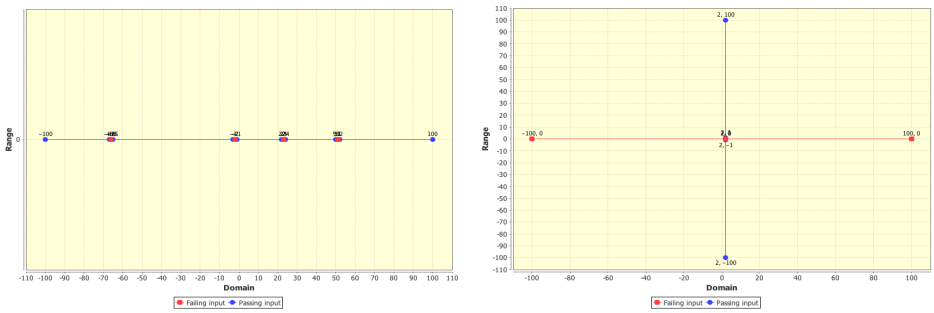
To elucidate the results, six programs were developed so as to have separate program for one and two-dimension point, block and strip fault domains. The code of selected programs is given in Appendix **??** (2-7). The experimental results are presented in table **??** and described under the following three headings.

**Point Fault Domain:** Two separate Java programs Pro2 and Pro3 given in Appendix **??** (2, 3) were tested with ADFD strategy in YETI to get the findings for point fault domain in one and two-dimension program. Figure 4.5(a) present range of pass and fail values for point fault

| S. No | Fault Domain | Module Dimension | Specific Fault | Pass Domain | Fail Domain |
|---|---|---|---|---|---|
| 1 | Point | One | PFDOneA(i) | -100 to -67, -65 to -3, -1 to 50, 2 to 22, 24 to 50, 52 to 100 | -66, -2, 23, 51 |
| | | Two | PFDTwoA(2, i) | (2, 100) to (2, 1), (2, -1) to (2, -100) | (2, 0) |
| | | | PFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |
| 2 | Block | One | BFDOneA(i) | -100 to -30, -25 to -2, 2 to 50, 55 to 100 | -1 to 1, -29 to -24, 51 to 54, |
| | | Two | BFDTwoA(-2, i) | (-2, 100) to (-2, 20), (-2, -1) to (-2, -100) | (-2 , 1) to ( -2, 19), (-2, 0) |
| | | | BFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |
| 3 | Strip | One | SFDOneA(i) | -100 to -5, 35 to 100 | -4, 34 |
| | | Two | SFDTwoA(-5, i) | (-5, 100) to (-5, 40), (-5, 0) to (-5, -100) | (-5, 39) to (-5, 1), (-5, 0) |
| | | | SFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |

Table 4.1: Pass and Fail domain with respect to one and two dimensional program

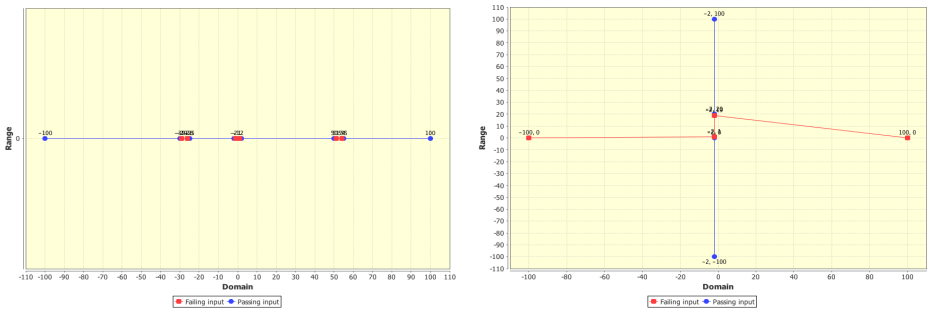
domain in one-dimension whereas Figure 4.5(b) present range of pass and fail values for point fault domain in two-dimension program. The range of pass and fail values for each program in point fault domain are given in (Table 4.1, Serial No. 1).




(a) One dimension module (b) Two dimension module

Figure 4.5: Chart generated by ADFD strategy presenting point fault domain

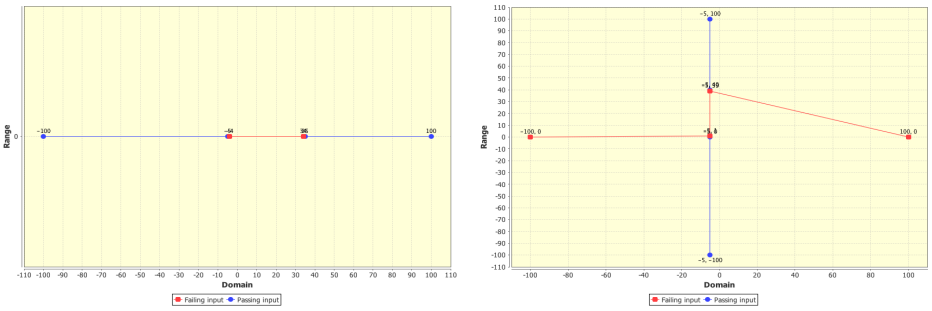
**Block Fault Domain:** Two separate Java programs Pro4 and Pro5 given in Appendix **??** (4, 5) were tested with ADFD strategy in YETI to get the findings for block fault domain in one and two-dimension program. Figure 4.6(a) present range of pass and fail values for block fault domain in one-dimension whereas Figure 4.6(b) present range of pass and fail values for block fault domain in two-dimension program. The range of pass and fail values for each program in block fault domain are given in (Table 4.1, Serial No. 2).




(a) One dimension module (b) Two dimension module

Figure 4.6: Chart generated by ADFD strategy presenting block fault domain

**Strip Fault Domain:** Two separate Java programs Pro6 and Pro7 given in Appendix **??** (6, 7) were tested with ADFD strategy in YETI to get the findings for strip fault domain in one and two-dimension program. Figure 4.7(a) present range of pass and fail values for strip fault domain in one-dimension whereas Figure 4.7(b) present range of pass and fail values for strip fault domain in two-dimension program. The range of pass and fail values for each program in strip fault domain are given in (Table 4.1, Serial No. 3).




(a) One dimension module (b) Two dimension module

Figure 4.7: Chart generated by ADFD strategy presenting Strip fault domain

## 4.5 Discussion

ADFD strategy with a simple graphical user interface is a fully automated process to identify and plot the pass and fault domains on the chart. Since the default settings are all set to optimum the user needs only to specify the module to be tested and click “Draw fault domain” button to start test execution. All the steps including Identification of fault, generation of dynamic java program to find domain of the identified fault, saving the program to a permanent media,

compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automated without any human intervention.

In the experiments (section 4.4), the ADFD strategy effectively identified faults and faults domain in a program. Identification of fault domain is simple for one and two dimension numerical program but the difficulty increases as the program dimension increases beyond two. Similarly no clear boundaries are defined for non-numerical data therefore it is not possible to plot domains for non-numerical data unless some boundary criteria is defined.

ADFD strategy initiate testing with random+ strategy to find the fault and later switch to brute-force strategy to apply all the values between upper and lower bound for finding pass and fault domain. It is found that faults at boundary of the input domain can pass unnoticed through ordinary random test strategy but not from ADFD strategy as it scan all the values between lower and upper range.

The overhead in terms of execution time associated with ADFD strategy is dependent mainly on the lower and upper bound. If the lower and upper bound is set to maximum range (i.e. minimum for int is Integer.MIN_INT and maximum Integer.MAX_INT) then the test duration is maximum. It is rightly so because for identification of fault domain the program is executed for every input available in the specified range. Similarly increasing the range also shrinks the produced graph making it difficult to identify clearly point, block and strip domain unless they are of considerable size. Beside range factor, test duration is also influenced by the identification of the fault and the complexity of module under test.

ADFD strategy can help the debuggers in two ways. First, it reduces the to and from movement of the project between the testers and debuggers as it identity all the faults in one go. Second, it identifies locations of all fault domains across the input domain in a user-friendly way helping debugger to fix the fault keeping in view its all occurrences.

## 4.6 Threats to Validity

The major external threat to the use of ADFD strategy on commercial scale is the selection of small set of error-seeded programs of only primitive types such as integer used in the experiments. However, the present study will serve as foundation for future work to expand it to general-purpose real world production application containing scalar and non-scalar data types.

Another issue is the easy plotting of numerical data in the form of distinctive units, because it is difficult to split the composite objects containing many fields into units for plotting. Some work has been done to quantify composite objects into units on the basis of multiple features[26],to facilitate easy plotting. Plotting composite objects is beyond the scope of the

present study. However, further studies are required to look in to the matter in depth.

Another threat to validity includes evaluating program with complex and more than two input arguments. ADFD strategy has so far only considered scalar data of one and two-dimensions. However, plotting domain of programs with complex non-scalar and more than two dimension argument is much more complicated and needs to be taken up in future studies.

Finally, plotting the range of pass or fail values for a large input domain (Integer.MIN_INT to Integer.MAX_INT) is difficult to adjust and does not give a clearly understandable view on the chart. Therefore zoom feature is added to the strategy to zoom into the areas of interest on the chart.

## 4.7 Related Works

Traditional random testing is quick, easy to implement and free from any bias. In spite of these benefits, the lower fault finding ability of traditional random testing is often criticised [68], [65]. To overcome the performance issues without compromising on its benefits, various researchers have altered its algorithm as explained in section 1. Most of the alterations are based on the existence of faults and fault domains across the input domain [9].

Identification, classification of pass and fail domains and visualisation of domains have not received due attention of the researchers. Podgurski et. al., [83] proposed a semi-automated procedure to classify similar faults and plot them by using a Hierarchical Multi Dimension Scaling (HMDS) algorithm. A tool named Xslice [2] visually differentiates the execution slices of passing and failing part of a test. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [52]. A serious limitation of the above mentioned tools is that they are not fully automated and require human interaction during execution. Moreover these tools are based on the already existing test cases where as ADFD strategy generate test cases, discover faults, identify pass and fault domains and visualise them in a fully automated manner.

## 4.8 Conclusion

Results of the experiments (section 4), based on applying ADFD strategy to error-seeded numerical programs provide, evidence that the strategy is highly effective in identifying the faults and plotting pass and fail domains of a given SUT. It further suggests that the strategy may prove effective for large programs. However, it must be confirmed with programs of more than two-dimension and different non-scalar argument types. ADFD strategy can find boundary faults quickly as against the traditional random testing, which is either, unable or takes

comparatively long time to discover the faults.

The use of ADFD strategy is highly effective in testing and debugging. It provides an easy to understand test report visualising pass and fail domains. It reduces the number of switches of SUT between testers and debuggers because all the faults are identified after a single execution. It improves debugging efficiency as the debuggers keep all the instances of a fault under consideration when debugging the fault.

# Chapter 5

# Invariant Guided Random+ Strategy

## 5.1 Introduction

# Appdx A

and here I put a bit of postamble ...

# Appdx B

and here I put some more postamble ...

# References


[1] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982. vii, 7

[2] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143 –151, oct 1995. 55

[3] NY. American National Standards Institute. New York, Institute of Electrical, and Electronics Engineers. *Software Engineering Standards: ANSI/IEEE Std 729-1983, Glossary of Software Engineering Terminology ...* Inst. of Electrical and Electronics Engineers, 1984. 2, 6

[4] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, 38:258–277, 2012. 43

[5] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2, 2001. 10

[6] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. 29, 43

[7] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. 9, 45

[8] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM. 22

[9] F.T. Chan, T.Y. Chen, I.K. Mak, and Y.T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775 – 782, 1996. vi, 12, 27, 30, 46, 55

[10] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, ECSQ '02, pages 321–330, London, UK, UK, 2002. Springer-Verlag. 3, 10, 27, 30, 43, 45

[11] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Normalized restricted random testing. In *Reliable Software TechnologiesAda-Europe 2003*, pages 368–381. Springer, 2003. 14

[12] Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java Class Libraries, Volume 1: Supplement for the Java 2 Platform, Standard Edition, V 1.2*, volume 1. Addison-Wesley Professional, 1999. 18

[13] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. *SIGSOFT Softw. Eng. Notes*, 24(6):285–302, 1999. 21

[14] Juei Chang and Debra J Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software EngineeringESEC/FSE99*, pages 285–302. Springer, 1999. 22

[15] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008. vi, 3, 12, 13, 27, 30, 37, 45

[16] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, page 4, Washington, DC, USA, 2003. IEEE Computer Society. vi, 3, 13, 27, 30, 43

[17] Tsong Yueh Chen, De Hao Huang, F-C Kuo, Robert G Merkel, and Johannes Mayer. Enhanced lattice-based adaptive random testing. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 422–429. ACM, 2009. 13

[18] Tsong Yueh Chen, Fei-Ching Kuo, and R. Merkel. On the statistical properties of the f-measure. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 146 – 153, sept. 2004. 37, 45

[19] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66, January 2010. 29, 43

[20] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 309–312, New York, NY, USA, 2005. ACM. 3, 13, 15, 27, 30, 43

[21] T.Y. Chen, R. Merkel, P.K. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 79 – 86, sept. 2004. 43

[22] T.Y. Chen and Y.T. Yu. On the relationship between partition and random testing. *Software Engineering, IEEE Transactions on*, 20(12):977 –980, dec 1994. 45

[23] T.Y. Chen and Y.T. Yu. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109 –119, feb 1996. 37

[24] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994. vii, 7

[25] I Ciupa, A Pretschner, M Oriol, A Leitner, and B Meyer. On the number and nature of faults found by random testing. *Software Testing Verification and Reliability*, 9999(9999):1–7, 2009. 29

[26] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 55–63, New York, NY, USA, 2006. ACM. 16, 17, 54

[27] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM. 3, 17, 21, 29, 37, 42

[28] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM. 3, 11, 29, 42

[29] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 157–166, Washington, DC, USA, 2008. IEEE Computer Society. 10, 11, 29, 45, 46

[30] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 72–81, Washington, DC, USA, 2008. IEEE Computer Society. 37

[31] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM. 21, 29, 42

[32] Lori A Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *Software Engineering, IEEE Transactions on*, 15(11):1318–1332, 1989.

[33] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997. 3

[34] Julie Cohen, Daniel Plakosh, and Kristi L Keeler. Robustness testing of software-intensive systems: Explanation and guide. 2005. 8

[35] Christoph Csallner and Yannis Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, September 2004. 42

[36] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972. 7

[37] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press. 11, 29

[38] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, july 1984. 29, 43

[39] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978. 9

[40] National Institute for Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Plannin Report 02-03, May 2002. 1, 2

[41] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association. 45

[42] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

[43] Marie-Claude Gaudel. Software testing based on formal specification. In *Testing Techniques in Software Engineering*, pages 215–242. Springer, 2010. vii, 7, 10

[44] D. Gilbert. *The JFreeChart class library version 1.0.9: Developer's guide*. Refinery Limited, Hertfordshire, 2008. 49

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005. 3, 15, 46

[46] W.J. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661 –674, sep/oct 1999. 43

[47] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, dec 1990. 11, 43

[48] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994. 10, 11, 29, 42, 45

[49] William E Howden. A functional approach to program testing and analysis. *Software Engineering, IEEE Transactions on*, (10):997–1005, 1986. 3

[50] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004. 1

[51] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM. 21

[52] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM. 55

[53] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-Based testing of java programs using SAT. *Automated Software Engineering*, 11:403–434, 2004. 10.1023/B:AUSE.0000038938.10589.b9. 21

[54] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990. 3

[55] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 261a–, Washington, DC, USA, 2007. IEEE Computer Society. 10, 29, 35, 43

[56] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420. ACM, 2007. 11

[57] Andreas Leitner, Alexander Pretschner, Stefan Mori, Bertrand Meyer, and Manuel Oriol. On the effectiveness of test extraction without overhead. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 416–425, Washington, DC, USA, 2009. IEEE Computer Society. 31

[58] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. 10(8):707–710, 1966. 17

[59] Richard C. Linger. Cleanroom software engineering for zero-defect software. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 2–13, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 45

[60] Huai Liu, Fei-Ching Kuo, and Tsong Yueh Chen. Comparison of adaptive random testing and random testing under various testing and debugging scenarios. *Software: Practice and Experience*, 42(8):1055–1074, 2012. 37

[61] Johannes Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 333–336. ACM, 2005. 46

[62] Thomas J McCabe. *Structured testing*, volume 500. IEEE Computer Society Press, 1983. 3

[63] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6(2):58–63, February 1963. 3

[64] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. 11

[65] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. Wiley, 2011. 2, 3, 55

[66] Simeon Ntafos. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998. 11

[67] Simeon C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, 27:949–960, October 2001. 29

[68] A. Jefferson Offutt and J. Huffman Hayes. A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, 21(3):195–200, May 1996. 3, 45, 55

[69] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004. 19, 29, 42

[70] M. Oriol. The york extensible testing infrastructure (yeti). 2010. 49

[71] M. Oriol. York extensible testing infrastructure, 2011. 34, 48, 49

[72] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201–210, 2012. 28

[73] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201 –210, april 2012. 29, 34, 35, 42, 43, 48, 49

[74] Manuel Oriol and Sotirios Tassis. Testing .net code with yeti. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, pages 264–265, Washington, DC, USA, 2010. IEEE Computer Society. 22

[75] Manuel Oriol and Faheem Ullah. Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:434–437, 2010. 22, 29, 34, 42, 49

[76] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002. 8

[77] Carlos Pacheco. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009. 11

[78] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, pages 504–527, 2005. vi, 20, 42

[79] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007. vi, 16, 18, 20

[80] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. 3, 27, 37

[81] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society. 15, 29, 46

[82] Ron Patton. *Software testing*, volume 2. Sams Indianapolis, 2001. 2

[83] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465 – 475, may 2003. 55

[84] CV Ramamoorthy and Sill-bun F Ho. Testing large software with automated software evaluation systems. In *ACM SIGPLAN Notices*, volume 10, pages 382–394. ACM, 1975. 10

[85] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM, 1992. vii, 7

[86] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332. ACM, 2007. 11

[87] E. Tempero. An empirical study of unused design decisions in open source java software. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 33 –40, dec. 2008. 43

[88] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, December 2010. 43

[89] Ewan Tempero, Steve Counsell, and James Noble. An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pages 3–12, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc. 35, 43

[90] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 285–288. IEEE, 1998. vii, 7

[91] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 1999. 10

[92] Sergiy A Vilkomir, Kalpesh Kapoor, and Jonathan P Bowen. Tolerance of control-flow testing criteria. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 182–187. IEEE, 2003.

[93] Willem Visser, Corina S P?s?reanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004. 16

[94] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. 10

[95] Lee J. White. Software testing and verification. *Advances in Computers*, 26(1):335–390, 1987. 3

[96] Wikipedia. Plagiarism — Wikipedia, the free encyclopedia, 20013. [Online; accessed 23-Mar-2013]. 3

[97] Maurice Wilkes. *Memoirs of a Computer Pioneer*. The MIT Press, 1985. 1

[98] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Softw. Test. Verif. Reliab.*, 22(3):171–201, May 2012. 43

[99] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008. 2