New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group

Department of Computer Science

University of York, UK

Nov. 2013

A thesis submitted for the degree of Doctor of Philosophy

Abstract

plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle, plenty of waffle.

Contents

List of Figures

List of Tables

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.

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. Dr. Mushtaq 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, and Dr. Ilyas 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.

Chapter 1

Introduction

Software is a very important and essential component of computer system without which no task can be accomplished. Some softwares are developed for use in simple day to day operations while others are for 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 [48]. Humans are prone to errors and programmers are no exceptions. Maurice Wilkes [131], a British computer pioneer, stated that:

> "As soon as we started programming, we found to our surprise that it was not 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 [63]. 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 [48, 91, 108, 133]. According

to Myers et al. some software companies spend up to 50% cost of the total cost of software development and maintenance on testing [91].

Software testing in the IEEE standard glossary of software engineering terminology [4], 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. 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 [41]. 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 reliability and confidence of users in the tested product.

Random testing is a process (Figure 1.2) 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. For more details read Chapter 3

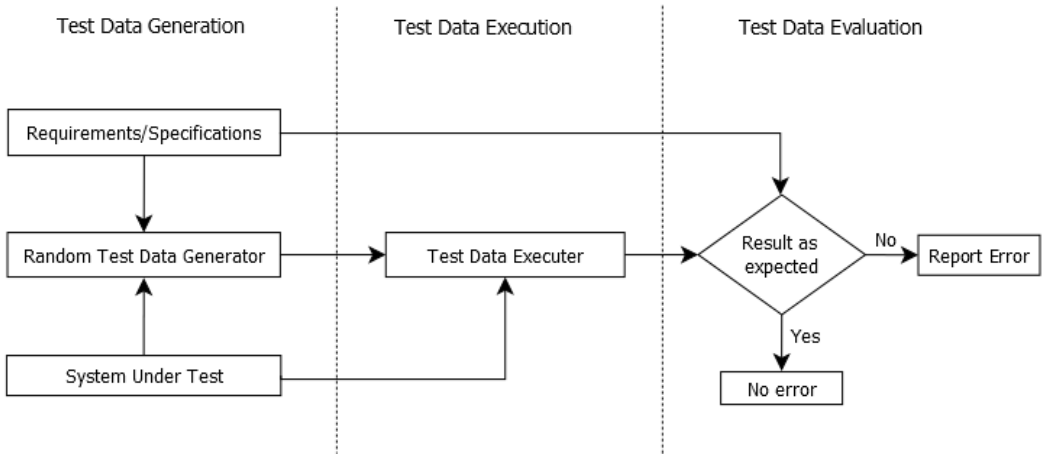


Figure 1.1: Three main phases of random testing

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.

1.1 The Problems

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. This lead to the problem of selecting a test data set, from a large/infinite domain. 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 [62, 83]. 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 [89]. 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.

Test data set can be generated manually and automatically. However, generating test data set manually is a time-consuming and laborious exercise [72]; 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 [130]. 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 [30]. According to Godefroid et al., “Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs” [55].

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [129]. Myers et al.mentioned it as, “Probably the poorest methodology of all is random-input testing...” [91]. However, Ciupa et al. reported that the above stated statement is based on intuition and lacks any experimental evidence [31]. The criticism motivated the researchers to look into various aspects of random testing for evaluation and possible improvement. Adaptive random testing (ART) [17], Restricted Random Testing (RRT) [13], Feedback Directed Random Testing (FDRT) [106], Mirror Adaptive Random Testing (MART) [18] and Quasi Random Testing (QRT) [22] are a few of the enhanced random testing techniques reported in the literature.

Random testing is also considered weak in providing high code coverage [36, 94].

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} available options. If *x* is an integer variable of 32 bit value [55].

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.

1.2 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 fault-finding test data.

1.3 Contributions

The main contributions of the thesis research are stated below:

1.3.1 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 4 indicated higher fault-finding ability of DSSR strategy as compared with Random (R) and Random+ (R+) strategies.

1.3.2 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 5. The experimental results proved that ADFD strategy automatically performed identification of faults and fault domains along with graphical representation in the form of chart.

1.3.3 Invariant Guided Random+ Strategy

Another random test strategy named, "Invariant guided Random+ Strategy (IGRS)" was developed in the current research study. IGRS 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 IGRS is implemented in YETI and generates values in compliance with the added assertions. Experimental results presented in Chapter 6 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.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides an overview of the software testing process its common techniques along with their purpose.

Chapter 3 is a thorough review of the relevant literature, describing random testing, various versions of random testing and the most commonly used automated testing tools based on random algorithms.

Chapter 4 describes newly proposed more efficient random testing technique known as Dirt Spot Sweeping Random (DSSR) strategy, which is based on sweeping of fault clusters in the input domain. The experimental study confirms that DSSR strategy is significantly better than R and R+ strategies. Finally the benefits and drawbacks of the DSSR strategy are discussed.

Chapter 5 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. It is shown that the presence of fault domains across the input domain, which have not been so far identified, can in fact be identified and graphically represented using ADFD strategy.

Chapter 6 presents the newly developed Invariant Guided Random+ Strategy (IGRS) developed with the focus on quick identification of faults and increase in code coverage with the help of assertions. It uses Daikon, a tool to generate likely invariants, to identify and incorporate invariants in the SUT code which serves as contracts to filter any vague test cases.

Chapter 7 is the concluding chapter, summarising the achievements of this research and discussing the future work.

Appendix A XXXXXXXX.

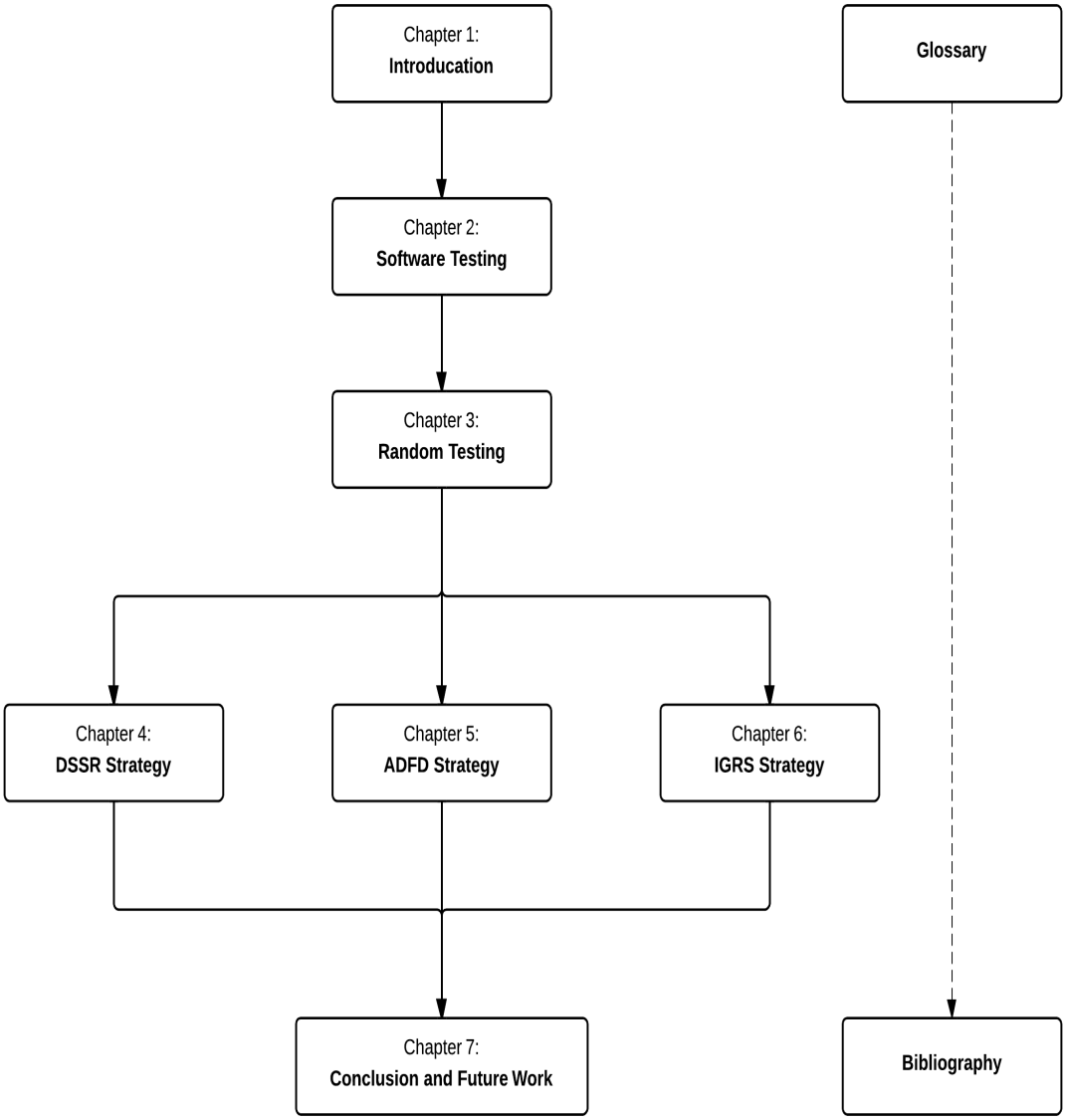


Figure 1.2: Structure of thesis outline

Chapter 2

Literature Review: Software Testing

The very famous quote of Paul, “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. Therefore, in spite of the best efforts, some errors may remain in the software after its completion. Errors cannot be tolerated in software because a single error may cause a large upset in the system. According to the National Institute of Standard and Technology [116], software errors cost an estimated $59.5 billion loss to US economy annually. The destruction of 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 and many others were caused by minor errors in the software [120]. To achieve high quality, the software has to satisfy rigorous stages of testing. The more complex the software, the higher the requirements for software testing and the larger the damage caused when a bug remains in the software.

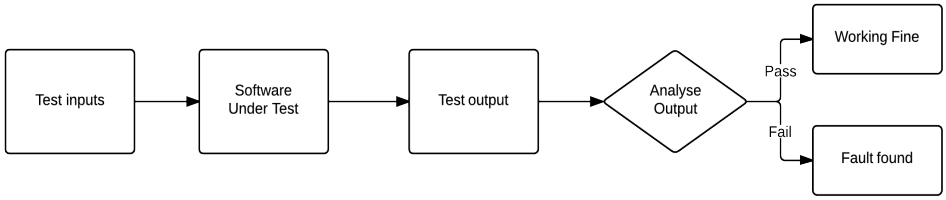


Figure 2.1: The process of software testing

In the IEEE standard glossary of software engineering terminology [4], 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 [90], where fault denotes the error made by programmers during software development [4].

The testing process, being an integral part of Software Development Life Cycle (SDLC), is started 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 [41]. 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 reliability and confidence of users in the tested product.

Table 2.1: Parts of Software Testing [1, 26, 53, 113, 121]

Levels	Purpose	Perspective	Execution
1. Unit 2. Integration 3. System	1. Functionality 2. Structural 3. Robustness 4. Stress 5. Compatibility 6. Performance	1. White Box a. Data Flow Analysis b. Control Flow Analysis c. Code-based fault injection testing 2. Black Box a. Use-case testing b. Partition testing c. Boundary Value testing d. Formal Specification testing	1. Static 2. Dynamic

2.1 Definitions

Before going into further details of software testing and its life cycle, few of the related terms are defined in this section. They have been added to make the thesis self contained.

2.1.1 Test Plan

A software test plan is a well defined document that defines the goal, scope, method, resources and time schedule of the testing [52]. Furthermore, it also describe the testable deliverable and the risk assessment of testable deliverables. Test plan guides the testers, *when, why, who* and *how* to perform a particular activity in the testing process.

2.1.2 Input Domain

Input domain for a given SUT is contained by the set of all possible inputs to that SUT. This includes all the global variables, formal parameters to the functions and the variables that are externally (keyboard input by user) assigned. Consider a program P with a corresponding input vector $P = \{x1, x2, ..., xn\}$, and let the domain of each input be $\{D1, D2, ..., Dn\}$, such that $x1 \in D1, x2 \in D2$ and so forth. The domain D of a function can then be expressed as the cross product of the domains of each input: $D = D1 \times D2 \times ... \times Dn$.

2.1.3 Test Case

A test case is an artifact that delineates the input, action and expected output corresponding to that input [3]. On executing the test case on the SUT if the output received comply to the expected output then the test case is pass, which means the functionality is working fine. However, in the case of any difference the result is fail, which means identification of a fault. Generally, execution of many test cases, collected in a test suite, are required to declare a software sufficiently scrutinized to be relasesed.

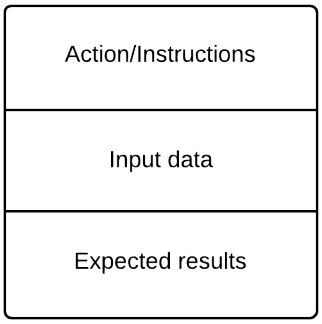


Figure 2.2: Test case

2.2 Software Testing Levels

The three main levels of software testing defined in the literature include unit testing, integration testing and system testing [26]. 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.

2.3 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 [37]. Finally, compatibility testing is performed to see the interaction of software with underlying operating system.

2.4 Software Testing Perspective

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

2.4.1 White-box testing

In white-box or structural testing, the testers 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 [102]. Some of the most common White-box testing techniques are briefly defined:

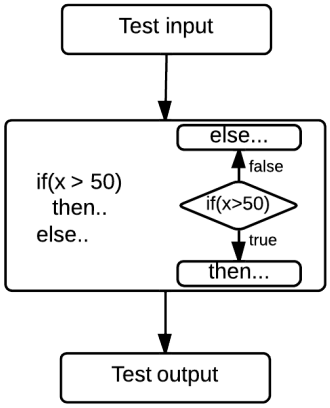


Figure 2.3: White-box testing

2.4.1.1 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 [35]. 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.

Generally, 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 objective of the technique to ensure that none of the aforementioned errors occur [51].

2.4.1.2 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 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 available 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 [124].

2.4.1.3 Code-based fault injection testing

A technique in which additional code is added to the code of the SUT at one or more locations to analyse its behaviour in response to the anomaly [126]. The process of code addition is called instrumentation which is usually performed before compilation and execution of software. The added code can be use for multiple reasons i.e. injection of fault to find the error handling behaviour of software, to determine the effectiveness of test procedure to check whether it discover the injected faults or to measure the code coverage achieved by the testing process.

2.4.2 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 [8]. Some of the most common black-box testing techniques are briefly defined:

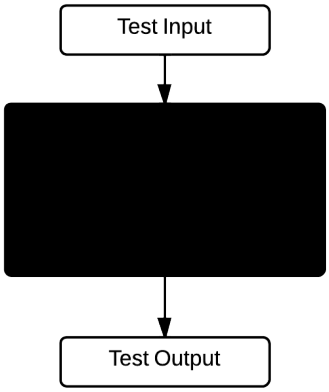


Figure 2.4: Black-box testing

2.4.2.1 Use-case based testing

A verification and validation technique that utilizes use-cases of the system to generate test cases. Use-case define functional requirements at a particular situation

or condition of the system from actor's (user or external system) perspective. It consists of a sequence of actions to represent a particular behaviour of the system. A use case format include a brief description of the event, flow of events, preconditions, postconditions, extension points, context diagram and activity diagram. All the details required for test case is included in the use case, therefore, use case can be easily transformed into test case.

The main benefits of use case testing is cheap generation of test cases, avoidance of test duplication, improved test coverage, easier regression testing and early identification of missing requirements.

2.4.2.2 Partition testing

A testing technique in which the input domain of a given SUT is divided into sub-domain according to some rule and then tests are conducted in each sub-domain. The division in to sub-domain can be according to the requirements or specifications, structure of the code or according to the process by which the software was developed [58].

While the performance of partition testing is directly dependant on the quality of sub-domain [128], it is often however difficult to divide the domain into equal partitions. Therefore, another version of partition testing called Proportional sampling strategy [12] is devised, in which the number of test cases selected from each partition is directly proportional to the size of the partition. Experiments performed by Ntafos [92] confirms the better performance of proportional partition testing over partition testing.

2.4.2.3 Boundary value analysis

Boundary Value Analysis (BVA) is a testing technique which is based on the rationale that errors tends to occur near the extremities of the input variables. Therefore in BVA the data set consists of values which are selected from the borders. According to IEEE standards [110], boundary value is a value that corresponds to minimum or maximum input, internal or external value specified for a component or system.

The BVA technique is also used in conjunction with partition testing where test

values are selected at the borders of each sub-domain instead of the whole input domain. The empirical analysis performed by Reid et al. [112] argue that BVA performs better in finding faults than partition testing. They also stated that like partition testing the performance of BVA is also dependant on the correct identification of partition and selection of boundary values.

The following code illustrate the ability of BVA to find a bug. On passing interesting value `MAX_INT` as argument to the $test$ method. The method code increment it by 1 making it a negative value and thus an error is generated when the system try to build an array of negative size.

```
public void test (int arg) {
arg = arg + 1;
int [] intArray = new intArray[arg];
...
}
```

2.4.2.4 Formal specification testing

Formal specification is defined as "a mathematical based technique, which offers rigorous and effective way to model, design and analyse computer systems" [38, 61]. The mathematical structure allows formal specifications to be manipulated mechanically so that information contained within the specification can be isolated, transformed, assembled, and repackaged to be used as test cases. Furthermore, it also guarantee that the test frames are logical consequences of the specification [42]. Formal specification testing is effective because it is independent from the code of the SUT. Which means that no change is required in the test cases as long as the specifications are unchanged [53]. It uses the existing specification model to verify the test results and thus avoid the oracle problem [9].

2.4.3 Test Oracle

Test oracles set the acceptable behaviour for test executions [6]. It is the test oracle which determines whether the software went successful or unsuccessful through a test. Test oracle is defined as "A source to determine expected results to compare with the actual result of the software under test" [3]. According to Howden [62], an

oracle is a function which, given a program, P, can determine, for each input, x, if the output from P is the same as the output from a correct version of P.

All software-testing techniques depend on the availability of test oracle [53]. Designing test oracle for ordinary software may be simple and 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 [6].
3. A specific oracle to satisfy all conditions is seldom available as rightly pointed out by Weyuker, stating that truly general test oracles are often unobtainable [127].

A number of artifacts can be directly used as oracle or can be used to generate oracle. This include the use of:

1. specification and documentation to generate test oracle.
2. other products that are similar to the SUT but with a different algorithm to solve the same problem.
3. heuristic algorithms which provides exact or approximate results for a set of test cases.
4. statistical characteristics to generate test oracle [82].
5. comparing the results of one test execution to another for consistency.
6. models to generate test oracle for the verification of SUT behaviour. [114].
7. manual analysis by human experts to verify the test results[68].

2.5 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 er-

rors 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 [46].

2.5.1 Manual Software 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 [32]. Manual testing may be effective in some cases but it is generally laborious, time consuming and error-prone [123]. Additionally, it requires that the testers must have appropriate skills, experience and sufficient knowledge of the SUT for evaluation from different perspectives.

2.5.2 Automated Software 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 [75]. 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 demand for increase in software functionality, quick development and decrease in cost of production without compromising quality leave no other choice but to automate the software testing. Automated software testing can be surprisingly effective and highly beneficial for any organisation. Its initial set-up cost may be higher in particular cases, however, a quick return on investment (QRI) outperforms it and brings the key benefits of cost reduction, productivity, availability, reliability and performance to any organisation. Automated testing is particularly effective when the nature of job is repetitive and performed on routine basis like unit testing and regression testing, where the tests are re-executed after each modification [64]. The use of automated software test-

ing made it possible to test large volumes of code, which would have been impossible otherwise [111].

2.6 Test Data Generation

Test data generation in software testing is the process of identifying test input data which satisfies given test selection criterion. A test data generator tool is used for the purpose which assist testers in the generation of test data while the test selection criterion define the properties of test cases to be generated based on the test plan and perspective taken [72]. Various artefacts of the SUT can be considered to generate test data like requirements, model or code. The choice of artefacts selected limits the kind of test selection criteria that can be applied in guiding the test case generation.

A typical test data generator consists of three parts: Program Analyser, Strategy Handler and Generator [45]. Program analyser performs initial assessment of software prior to testing and may perform changes to it, if required. For example it performs code instrumentation or construction of CFG to measure the code coverage during testing. A strategy handler define the test case selection criteria. This may include the formalisation of test coverage criterion, the selection of paths, normalisation of constraints, etc. It may also get input from program analyser or user before or during execution. The Generator taking inputs from the Program analyser and Strategy handler generates test cases according to the set selection criteria. Test data generators based on their approaches are classified into Path-wise, Goal Oriented, Intelligent and Random Test Data Generators. Each one is briefly defined in the following.

2.6.1 Path-wise Test Data Generators

A test data generation technique in which the test data is generated to target path coverage, statement coverage, branch coverage, etc. in a given SUT. The approach generally consists of three main parts: Control Flow Graph (CFG) construction, path selection and test data generation.

In path-oriented approach, the program path(s) to the selected statement are iden-

tified and then input data is generated for exercising that path. The path to be tested can be generated automatically, or it can be provided by the user. In path testing, data is generated for a boolean expression which contain two branches with a true and false node. A reference to the sibling node means, the other node, corresponding to the current executed node. For example the sibling node of True branch is False branch.

Each path belongs to a certain sub domain, which consists of those inputs which are necessary to traverse that path. For generating test cases the execution of the software with test data to find errors in the flow of the control through the software. The test data belong to an input space which is partitioned into a set of sub domains which belong to a certain path in the software. The boundary of these domains is obtained by the predicates in the path condition where a border segment is a section of the boundary created by a single path condition. Two types of boundary test points are necessary; on and off test points. The on test points are on the border within an adjacent domain. If the software generates correct results for all these points then it can be considered that the border locations are correct.

2.6.2 Goal-oriented Test Data Generators

A test data generation technique in which the test data is generated to exercise a specific program point rather than a program path [27]. In this approach the tester can select any path among a set of available paths as long as they reach the target. Many goal-oriented test data generation approaches execute the SUT to generate test data, enabling it to utilize run-time information for computing more accurate test data [47].

2.6.2.1 Chaining Approach

The chaining approach uses data dependency analysis to guide the test data generation process. The data dependency analysis automatically identifies statements that affect the execution of the selected statement. The chaining approach uses these statements to form a sequence of statements that is to be executed prior to the execution of the selected statement [47].

It analysed the program according to its edges and nodes. For each test coverage criterion, a different initial event sequence is defined and the goal nodes are determined accordingly. For example, consider the branch (p, q), where p is the beginning node of the branch and q is the last node in the branch. The initial event sequence E for the branch (p, q) is defined as $E =< (s, \phi), (p, \phi), (q, \phi) >$ provided that s is the starting node of the program and ϕ is the set of variables referred to as a constraint. After this step, the Branch Classification process identifies critical, semi critical and non critical nodes for each branch. Then, during the execution of the program, this classification leads the search to decide which branch to take to reach the goal node or to cover the requested branch.

2.6.2.2 Assertion-Oriented Approach

In this approach the goal is to identify program input on which an assertion(s) is violated. More formally, we can define the problem of assertion-oriented test data generation as follows: For a given assertion A, find program input x on which assertion A is false, i.e., when the program is executed on input x and the execution reaches assertion A, assertion A evaluates to false.

The problem of finding a program input on which an assertion is violated is undecidable. Furthermore, it is not always possible to generate test cases that violate assertions. However, the experiments have shown that the assertion-oriented test data generation may frequently detect errors in the program related to assertion violation. The major advantage of this approach is that each test data generated uncovers an error in the program because the assertion is violated. There are three reasons as to why an assertion is violated: a faulty program, a faulty assertion, or a faulty precondition (input assertion).

An assertion specifies a constraint that applies to some state of a computation. When the assertion evaluates to false during program execution, there exists an incorrect state in the program. Assertions proved to be very effective in testing and debugging cycle. For example, during black- box and white-box testing assertions are evaluated for each program execution. Information about assertion violations is used to localize and fix bugs [73].

2.6.3 Intelligent Test Data Generators

Intelligent Test Data Generators performs sophisticated analysis, such as fuzzy logic, neural networks, or genetic algorithms, on the SUT to guide the search for the test data. The approach involves complex analysis to anticipate the different situations that may arise at any point. While the approach produce test data, which satisfy the SUT requirements the cost involved in doing so consumes more time and resources. Intelligent Test Data Generation is a technique used to overcome the problems associated with traditional data generation techniques that include generation of meaningless data, duplicated data and failing to generate complex test data etc. The approach increases users confidence in the generated test data, testing process and consequently the SUT [111].

2.6.3.1 Genetic Algorithm

A genetic algorithm is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. In the test-data generation application, the solution sought by the genetic algorithm is test data that causes execution of a given statement, branch, path, or definition-use pair in the program under test. The genetic algorithm is guided by the control dependencies in the program, to search for test data to satisfy test requirements. The genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. The algorithm evaluates the candidate test data, and hence guide the direction of the search, using the programs control-dependence graph [107].

The main benefit of the approach is that it can generate test data quickly, but with focus and direction. New test cases are generated by applying simple operations on existing test cases that are judged to have good potential to satisfy the test requirements. The success of this approach, however, depends heavily on the way in which the existing test data is measured [107].

2.6.4 Random Test Data Generators

Random test data generation is probably the simplest method for generation of test data. The advantage of this is that it can be used to generate input for any type of

program. Thus to generate test data we can randomly generate a bit stream and let it the represent the data type needed. However, random test data generation does not generate quality test data as it does not perform well in terms of coverage. Since the data generated is based solely on probability it cannot accomplish high coverage as the chances of it finding semantically small faults is quite low.[3]

If a fault is only revealed by a small percentage of the program input it is said to be a semantically small fault. For example of a semantically small fault consider the following code:

```
void test(char x,char y) {
    if (x==y)
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}
```

It is easy to see that the probability of execution of the first statement is significantly lesser than that of the second statement. As the structures in it grow complex so does the probability of its execution. Thus, such semantically small faults are hard to find using random test data generation. However, Random Test Data Generation is usually used as a benchmark as it has the lowest acceptable rate of generating test data.

2.6.5 Search-Based Test Data Generation

The field of searchbased software engineering reformulates software engineering problems as optimization problems and uses metaheuristic algorithms to solve them. Metaheuristic algorithms combine various heuristic methods in order to find solutions to computationally hard problems where no problem specific heuristic exists. In SBDT each input vector x can be associated with a measure $cost(x)$ that represents how far away the input vector x is from satisfying the required goal. Input test values that come close to satisfying the desired goal have low cost values, those that do not come close to satisfying the goals have high cost values.

Consider a program with an initial branch statement: $if(x >= 20)y = z; elsey = 2 * z;$ and suppose we want the true branch to be executed. An input value of $x == 25$ clearly satisfies the predicate, and a value of $x == 15$ can be seen to come

closer to satisfying the predicate than a value of $x == 5$. We might evaluate a cost function probe (immediately before the indicated statement) of the form $cost(x) = max0, 20 - x$. Thus $x == 25$ has cost 0, $x == 15$ has cost 5 and $x = 5$ has cost 15. We can see how finding data to satisfy the branch predicate is essentially a search over the input domain of x to find a value such that $cost(x) == 0$. In a similar fashion, finding data to follow a particular path through the code can be couched as one of satisfying each of a number of predicates at points in program execution (giving rise to a cost function that combines in some way the costs apparent at each of the relevant branching points). The technique is blind to the particular interpretation of predicates and so the same approach can be used to search for test data that satisfies the pre-condition of an operation but which gives rise to a final state and output that falsifies the post-condition. Tracey referred to this as specification falsification [121]. In addition, healthiness pre-conditions for statements can be targeted and broken and so exception generating test data may also be sought [122].

The approach requires the measurement of state at appropriate points in a programs execution. Furthermore, the cost function plays the role of oracle for each targeted test requirement. As a consequence, the cost function must change as the requirement (e.g. to follow another branch or path) is changed. Thus, frequent re-instrumentation of a program is required to find test data to fully satisfy common coverage criteria. The can be seen in the most extensive early framework for search based test data generation.

2.7 Summary

The chapter gives an overview of software testing process, starting from defining what software testing is, why it is necessary, its common types and the purpose for which they are used. It then differentiate between manual and automated software testing and finally various ways of software test data generation, being the most critical and crucial part of any testing system are studied.

Chapter 3

Literature Review: Random Testing

Work on random test generation dates back to a 1970 paper by Hanford. In it he reports on the "syntax machine", a program that generated random but syntactically-correct programs for testing PL/I compilers [60]. The next published work on random testing was a 1983 paper by Bird and Munoz [10]. They describe a technique to generate randomly-generated "self-checking" test cases: test cases that in addition to generating test inputs generated checks for expected outputs using knowledge of the semantics of the software under test. For example, when generating random programs for testing a compiler, the generator kept track of the expected values of arithmetic computations and inserted the appropriate checks into the test case. They also applied their technique to checking sorting and merging procedures and graphic display software [103].

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 [13]. The input domain is a set of all possible inputs to the software under test. According to Richard [59], 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.

Generating test data by random generator is quite economical and requires less intellectual and computational efforts [31]. Moreover, no human intervention is involved in data generation that ensures an unbiased testing process.

Figure 3.1: Random Testing

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 [103, 115]. Myers mentioned random testing as one of the least effective testing technique [90].

It is stated by Ciupa et al. [30], that Myers statement was not based on any experimental evidence. Later experiments performed by several researchers [59, 32, 76, 43] confirmed that random testing is as effective as any other testing technique. It was found experimentally [43] 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 [58]. A comparative study conducted by Ntafos [92] concluded that random testing is more effective as compared to proportional partition testing.

One of the best-known works in the field is Miller et al.s fuzz testing paper, where they generate random ASCII character streams and use them as input to test Unix utilities for abnormal termination or non-terminating behaviour [88]. In subsequent work, they extended fuzz testing to generate sequences of keyboard and mouse events, and found errors in applications running in X Windows, Windows NT and Mac OS X [49, 87]. Today, fuzz testing is used routinely in industry. It is frequently used as a tool for finding security vulnerabilities and is applied to test formats and protocols that attackers might use to gain unauthorized access to computers over a network. Other work that applies random test generation to operating systems code includes Kropps use of random test generation to test low-level system calls [74], and Groce et al.s use of random test generation at NASA to test a file system used in space missions [56]. In all these studies, a random test generator invariable

found many errors in the software under test, and many errors were critical in nature. For example, Groce et al. found dozens of errors in the file system under test, many of which could have jeopardized the success of a mission.

3.1 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:

3.1.1 Adaptive Random Testing

Adaptive random testing (ART), proposed by Chen et al. [17] is based on the previous work of Chan et al. [12] 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.

In figure 3.2 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

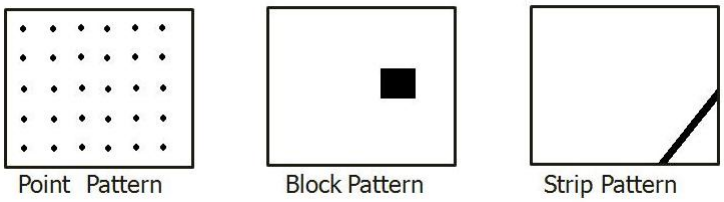


Figure 3.2: Patterns of failure causing inputs

of points, block and strip inside the respective box indicates the point, block and strip fault patterns.

Chen et al. [17] 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. [17] 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 [19] and [22].

3.1.2 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) [18] is an improvement on ART by using mirror-partitioning technique to decrease the extra computation involved in ART

and reduce the overhead.

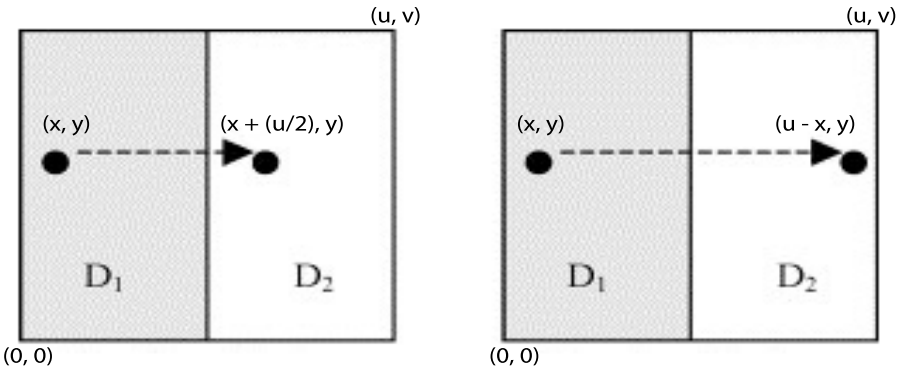


Figure 3.3: Mirror Adaptive Random Testing [18]

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 3.3. 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.

3.1.3 Restricted Random Testing

Restricted Random Testing [14] 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.

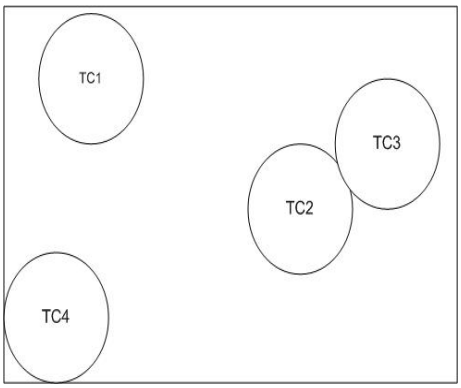


Figure 3.4: Input domain with exclusion zone around the selected test case

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).

3.1.4 Directed Automated Random Testing

Godefroid et al. [55] 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

3.1.5 Quasi Random Testing

Quasi-random testing (QRT) [22] 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.

3.1.6 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) [106] 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.

3.1.7 The ARTOO Testing

The Adaptive Random Testing for Object Oriented (ARTOO) strategy is based on object distance. Ciupa et al. [29], defined the parameters that can be used to calculate distance between the objects. 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 and values assigned to the primitive and reference fields. Strings are treated in terms of directly usable values and Levenshtein formula [78] is used as a distance criteria between the two strings.

In the ARTOO strategy, two sets are taken i.e. candidate-set containing the objects ready to be run by the system and the used-set, which is empty. First object is selected randomly from the candidate-set which is moved to used-set after execution. The second object selected from the candidate-set for execution is the one with the largest distance from the last executed object present in the used-set. The process continues till the bug is found or the objects in the candidate-set are finished [29].

The ARTOO strategy, implemented in AutoTest tool [31], was evaluated in comparison with Directed Random (D-RAN) strategy by selecting classes from EiffelBase library [86]. The experimental results indicated that some bugs found by the ARTOO were not identified by the D-RAN strategy. Moreover the ARTOO found first bug with small number of test cases than the D-RAN strategy. However, computation to select test case in the ARTOO strategy was more than the D-RAN strategy thus it took more time to generate a test case.

.

3.1.8 Design by Contract

Modern software development as it is well known has adopted the paradigm of Object Oriented (OO) Programming. The primary and most important reason for this evolution is the desire for better quality software and a more efficient way to bridge the gap between requirements and code. Industrial use of OO confirmed the superiority of this approach over procedural programming languages. However, so far no final approach has been generally accepted over a methodology on how to construct OO software in order to achieve one of the most important factors of quality, which is reliability or else robustness. As the work of Dr. Bertrand Meyer

has shown [85, 84] the decision on how to make software more reliable is crucial if the developers want to use the benefits of OO. These benefits include: 1 Reuse of software components. This implies using a software component at multiple environments apart from the environment in which its developers originally deployed it. 2 The goal of having reliability plays an important role in characterizing the quality of a software module. 3 The abstract types that OO introduces. A reliable way to construct abstract types emerges. To achieve the previous goals the software development world has used two approaches, Defensive programming and Design by Contract. As the next paragraphs show the latter one has been showing more advantages than the first one. That is why most researches on Automated Testing use design by contract as a requirement that the SUT must conform to, in order for their tool to find as many bugs as possible [75]. The idea of Defensive Programming, which is inherited from the software development world prior to OO, is in general advising to include as many checks as possible even if they are redundant [85]. This concept is intuitively stating that having extra checking can never do harm especially if someone wants to protect the software from inexperienced users. This last concept unfortunately is not correct. This kind of approach puts more code inside the software and this contributes to greater complexity of the software, and complexity as Dr. Meyer has very correctly stated is the single worst obstacle to software quality in general, and to reliability in particular [85]. The reason is that this extra code is a source of things that might go wrong and so we must also check it, and so on to infinitum. The need for a more systematic approach is evident based on the concept that software elements are implementations of well-understood specifications by the developers and that is exactly what Design by Contract does. Design by Contract concept adopts the idea that any operation a routine of a software performs, should bind the caller of that routine to the routine itself. This binding provides specific obligations that each party has, along with benefits. Essentially the obligations of one part describe the benefits of the other part. These ideas apply to software routines (i.e. methods, functions of classes) via assertions that implement the binding, which is like a contract between two persons. These assertions are classified into pre-conditions, post-conditions and class invariants. Figure 2.1 which is a figure of the Applying Design by Contract article [85], illustrates the use of these assertions. As anyone can understand from Figure ?? pre and post-conditions denote the idea that if the caller promises to invoke the routine with the pre-conditions holding, the routine guarantees the caller that it will return the system in a final state in which the post-conditions hold.

Thus, the invoker knows nothing about how the routine reaches the final state but can depend on the results. This implies another thing, which is that the routine is responsible only for the cases where the pre- conditions hold. This means rejecting the whole concept of defensive programming because if something is an assertion in the pre-condition the routine does not have to handle it in its body/-code and vice versa. Overall, it is forbidden for the same assertion to exist in both parts. How strong or weak a pre-condition should be is a decision of the developer. So far the notions of pre and post-conditions are assertions that developers can use even with procedural programming languages at each of their routines. The notion that enhances more the efficiency of the contracts is the notion of the class invariant. Through class invariants a developer can describe a general condition that any instantiation of that class must hold at any time [85, 84]. By this the developers classify the requirements that they have in the same abstract way in which they develop the software; making the bridging of requirements to code more traceable, thus more possible to achieve all the requirements, the basic goal of software development. In addition, to the previous benefits design by contract can facilitate documentation. By providing contract information documentation describes a module of code certainly better than just presenting the interface and the result it yields. Furthermore, monitoring these assertions can be very helpful while debugging. It is up to the developers to decide what the software must do when one of the parties breaks a contract. It can stop the execution, prompt the event, ask the user to decide etc. It is a decision based on the specifications. Basically as the article of Meyer B. [85] describes, three things can mainly occur 1. An alternative algorithm starts due to this exceptional behaviour 2. System stops its executions and returns to a prior consistent state 3. A rare but possible false alarm has happened due to operating system or hardware signals and after correcting actions execution continues The general approach is to consider and to monitor any violation because most of the times this violation describes a bug. Either in the logic of the requirements (pre-conditions, class invariants) or in the algorithm of the implementation (post-conditions). Sometimes research papers mention the contracts of the code as partial specifications [40] because they provide more information than just requirements documents but at the same time they are not a full specification model which describes the whole algorithm of a software module as formal methods do. Design by Contract principle helps developers use appropriately polymorphism and dynamic binding. With contracts, designers can have inheritance and reassure that the class that inherits another class will respect the

original contract and if the designer wants he/she can add assertions based on the functionality of the child class. The rules that the designers must follow, to maximally use the contracts, is to allow the child classes, when it is desired, to: i) have weaker pre-conditions or ii) stronger post-conditions. In this way contracts provide an immediate guideline to the designer.

3.1.9 Daikon

3.2 Tools for Automated Random Testing

A number of open-source and commercial random testing tools that automatically generate test cases, reported in the literature, are briefly described in the following section.

3.2.1 JCrasher

JCrasher is an automatic robustness testing tool developed by Csallner and Smaragadakis [105]. JCrasher tries to crash the Java program with random input and exceptions thrown during the process are recorded. The exceptions are then compared with the list of acceptable standards, defined in advance as heuristics. The undefined runtime exceptions are considered as errors. Since users interact with programs through its public methods with different kind of inputs, therefore, JCrasher is designed to test only the public methods of the SUT with random inputs.

Figure 3.5 illustrates the working of JCrasher by testing a *T.java* program. The source file is first compiled using *javac* and the byte code obtained is passed as input to JCrasher. The JCrasher, with the help of Java reflection [15], analyse all the methods declared by class *T* using methods transitive parameter types *P* to generate the most appropriate test data set which is written to a file *TTest.java*. The file is compiled and executed by JUnit. All the exceptions produced during test case executions are collected and compared with robustness heuristic for any violation which is reported as error.

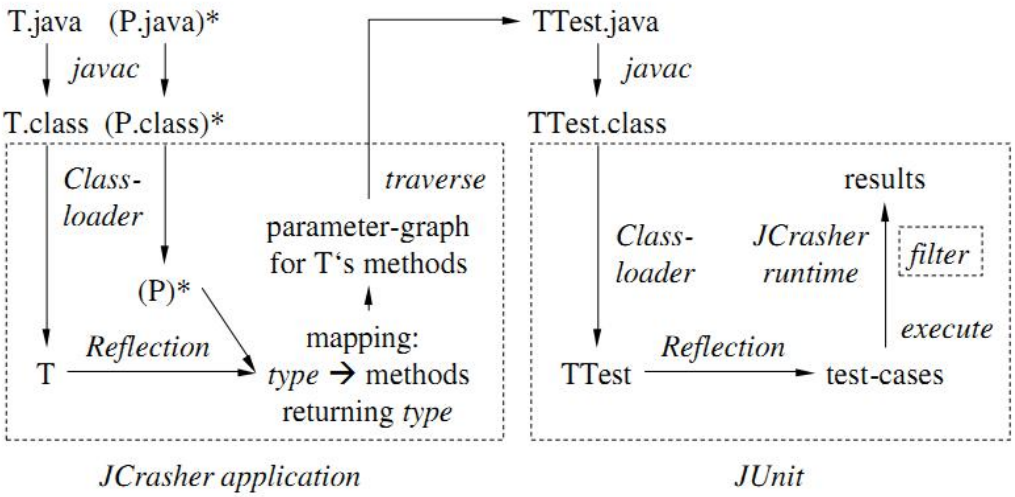


Figure 3.5: Illustration of robustness testing of Java program with JCrasher [105]

JCrasher is a pioneering tool with the capability to perform fully automatic testing, including test case generation, execution, filtration to report generation. JCrasher has the novelty to generate test cases as JUnit files which can also be easily read and 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.

3.2.2 Jartege

Jartege (Java random test generator) [95] 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. In addition, it parametrise its random aspect in order to prioritise testing specific part of the class or to get interesting sequences of calls. The parameters include the following:

- Operational profile of the classes i.e. the likely use of the class under test by other classes.
- Weight of the class and the method under test. Higher weight prioritizes the

class or method over lower weight during test process.

- Probability of creating new objects during test process. 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 evaluates a class by entry pre-condition and internal pre-conditions. Entry pre-conditions are the contracts to be met by the generated test data for testing the method while internal pre-conditions are the contracts which are inside the methods and their violation are considered as error either in the method or in the specification. The Jartege checks for errors in program code as well as in specifications and the Junit tests produced by Jartege can be used later as regression tests. Its limitation is the requirement of prior existence of the program JML specifications.

3.2.3 Eclat

Eclat [104] is an automated testing tool which generates and classifies unit tests for Java classes. The process is accomplished in three main components. In the first component, it selects a small subset of test inputs, likely to reveal faults in the given SUT, from a large set.

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. If the operational pattern of execution of the test data differs from the model, the following three outcomes may be possible: (a) it sight a fault in the given SUT, (b) it might produce normal operations despite model violation, (c) it might be an illegal input that the program cannot handle. (note for author extend 2nd and 3rd component) In the second component of the process, reducer function is used to discard any redundant input, leaving only a single input per operational pattern. In third component the acquired test inputs are converted into test cases, by creation of oracle, to determine the success or failure of the test. Eclat was compared with JCrasher by executing nine programs on both tools individually [105]. Based on the experimental results it was revealed that Eclat perform better than JCrasher. On the average, Eclat selected 5.0 inputs per run out of which 30% revealed faults, while JCrasher selected 1.13 inputs per run out of which 0.92% of those revealed faults. The limitation of Eclat is dependence on the

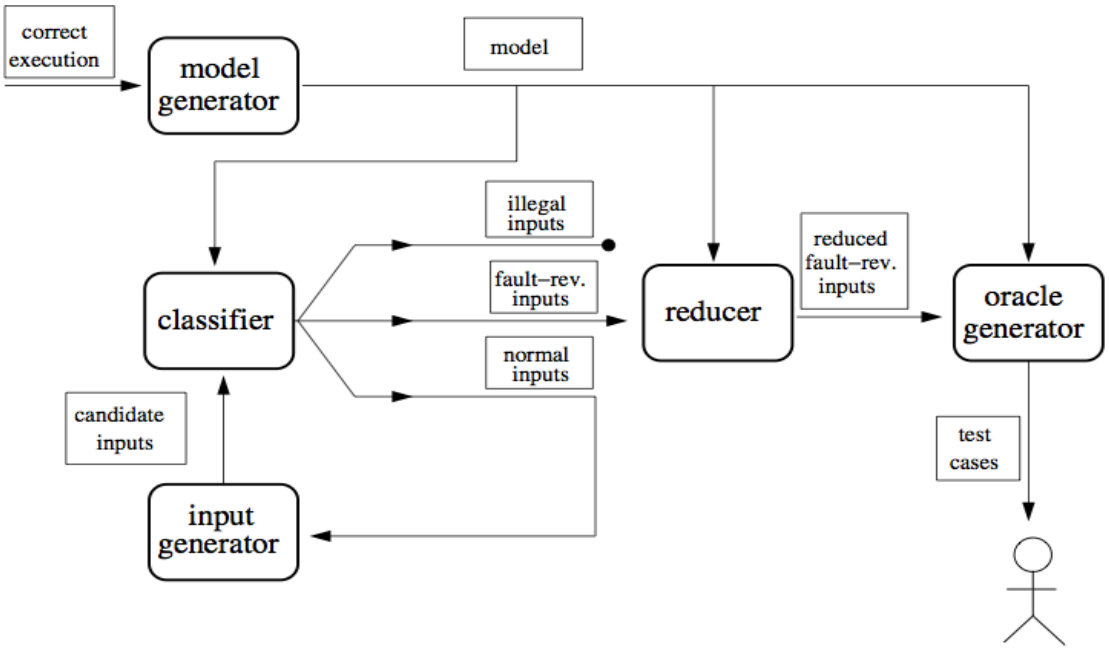


Figure 3.6: Main component of Eclat contributing to generate test input [104]

initial pool of correct test cases and existence of any errors in the pool may lead to the creation of wrong operational model which will adversely affect the testing process.

3.2.4 Randoop Tool

Random tester for Object Oriented Programs (RANDOOP) is the tool used for implementing FDRT technique [105]. RANDOOP is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes as input a set of classes, contracts, filters and time limit while its output is a suite of JUnit and NUnit for Java and .Net program 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 [125]. In their 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.

3.2.5 QuickCheck Tool

QuickCheck [34] is a lightweight random testing tool used for testing of Haskell programs [65]. Haskell is a functional programming language where programs are evaluated by using expressions rather than statements as in imperative programming. In Haskell most of the functions are pure except the IO functions, thus main focus of the tool is on testing pure functions. These are the functions which depend on its input parameters and make changes to them only. QuickCheck tool is designed to have a simple domain-specific language of testable specifications embedded in Haskell. This language is used to define expected properties of the functions under test - for example, reversing a list with single element must result in the same list.
The QuickCheck takes function to be tested and properties of the program defined by tester (Haskell functions) as input. The tool uses built-in random generator to generate effective test data, however, to get adequate coverage in the case of custom data types, the testers can also develop their own generator. On executing the function with test data, the tester defined properties must hold for the function to be correct. Any violation of the defined properties suggest error in the function.

3.2.6 Autotost Tool

The Autotest tool, based on Formal Automated testing is used to test Eiffel language programs [30]. The Eiffel language uses the concept of contracts which is effectively utilized by Autotest - for example the auto generated input is filtered using pre-conditions and unwanted test input is discarded. The contracts are also

used as test oracle to determine if the test is pass or fail. Beside automated testing the Autotest also allow the tester to manually write the test cases to target specific behaviour or section of the code. The Autotest scope can be a single method/-class or cluster of methods/classes as inputs, it then automatically generate test input data according to the requirement of the methods or classes.

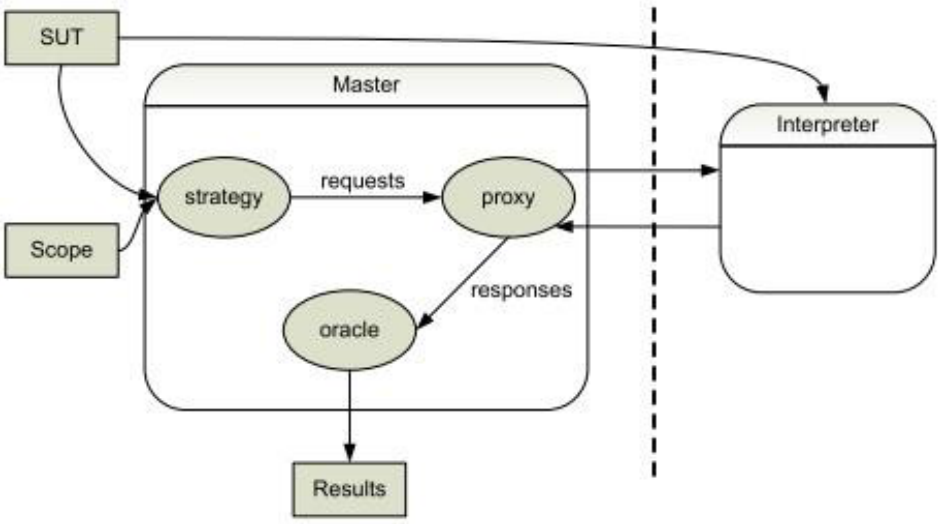


Figure 3.7: Architecture of Autotest

According to Figure 3.7 [75], the architecture of Autotest tool can be split into the following main parts:

1. **Testing Strategy:** is a pluggable component where testers can fit any strategy according to their testing requirements. The strategy contains the directions for testing - for example what instructions should be executed on the SUT. Using the information the strategy synthesize test cases and forward it to the proxy. The default strategy creates test cases that uses random input to exercise the classes under test.

2. **Proxy:** handles inter-process communication. It receives execution requests from the strategy and forward them to the interpreter. The execution results are sent to the oracle.

3. **Interpreter:** execute instructions on the SUT. The most common instructions include: create object, invoke routine and assign result. The interpreter process is kept separate to increase robustness.

4. **Oracle:** is based on contract-based testing. It evaluate the results to see if they satisfy the contracts or not. The outcome of the tests are formatted in

HTML and stored on disk.

3.2.7 TestEra Tool

TestEra [71] is a novel framework for auto generation and evaluation of test inputs for a Java program. It takes methods specifications, integer value and the method under test as input. It uses pre-conditions of a method to generate all non isomorphic valid test input to the specified limit. The test inputs are executed on the method and the results are compared against the postconditions of the method serving as oracle. Any test case that fails to satisfy postcondition is considered as a fault.

TestEra uses the Alloy modelling language [67] to express constraints on test inputs and Alloy Analyser tool [66] to solve these constraints and generate test inputs. Alloy Analyzer performs the following three functions: (a) it translates Alloy predicates into propositional formulas, i.e. constraints where all variables are boolean; (b) it evaluates the propositional formulas to find its outcome; (c) it translates each outcome from propositional domain into the relational domain.

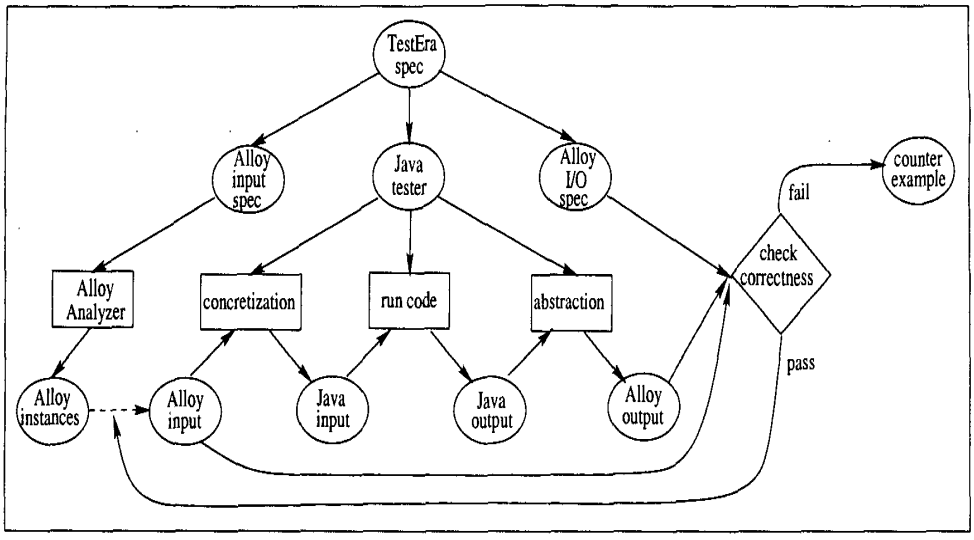


Figure 3.8: Architecture of TestEra

TestEra and Korat are similar tools because they both uses program specifications to guide the auto generation of test inputs. However they are different from Jartege and AutoTest which uses specifications to filter and truncate the unnecessary ran-

dom generated inputs. While the tools use program specifications differently for test input generation, they all uses it in a similar way for oracle.

3.2.8 Korat Tool

Korat [11] is a novel framework for automated testing of Java programs based on their formal specifications [16]. Korat and TestEra [71] were developed by the same team and both performs specification based testing. The difference however is that Korat uses Java Modelling Language (JML) while TestEra uses Alloy Modelling Language for specifications. Korat uses bounded-exhaustive testing [70] in which the code is tested against all possible inputs within the given small bound.

Korat generate structurally complex inputs by solving imperative predicates. An imperative predicate is a piece of code that takes a structure as input and evaluates to a boolean value. Korat takes imparative predicate and additionally a finitization value, that bounds the size of the structures that are inputs to the predicate, as input. It systematically explore the predicates input space and generate all non isomorphic inputs (inherently distinct inputs in the same input domain) 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. These inputs are taken as test cases. Korat depends or developers written *repOK()* and *checkRep()* methods, where *repOK()* is used to check the class invariants and *checkRep()* is used to verify the post-conditions to validate the correctness of the test case.

The key benefit of Korat and TestEra, representation level approaches, is that no existing set of operations are required to create input values and therefore they can achieve to create input values that may be difficult or impossible using a given set of operations. However, The only disadvantage to this approach is the requirement of significant amount of manual efforts [103].

3.3 YETI Overview

YETI stands for York Extensible Testing Infrastructure. It is an automated random testing tool developed in Java that is capable of testing programs written in Java, JML and .NET languages [100]. YETI takes program byte code as input and execute it with random generated but type-correct inputs to find a fault. It runs at a high level of performance with 10^6 calls per minute on Java code. One of its prominent feature is Graphical User Interface (GUI) for user friendliness along with the option to change testing process in real time. It can also distribute large testing tasks in cloud for parallel execution [101]. The latest version of YETI can be downloaded from https://code.google.com/p/yeti-test/downloads/list. The following sections briefly describe internal working and execution of YETI tool.

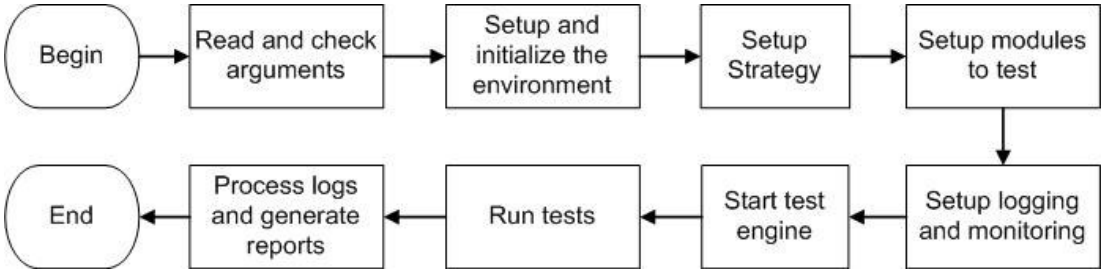


Figure 3.9: YETI test cycle

3.3.1 YETI Design

YETI has been designed with an emphasis on extensibility to facilitate future growth. YETI enforces strong decoupling between testing strategies and the actual language constructs. Therefore, adding a new binding, need no modification what so ever to any existing test strategies. On the basis of functionality YETI is divided into three main sections: the core infrastructure, the strategy and the language-specific bindings. Each of the section is defined below.

3.3.1.1 YETI Core Infrastructure

The YETI core infrastructure is responsible for test data generation, test process management and generation of test report. It is further split into four packages: yeti, yeti.environments, yeti.monitoring, yeti.strategies. The $yeti$ package uses

classes from $yeti.monitoring$ and $yeti.strategies$ packages and calls classes in the $yeti.environment$ package as shown in the figure 3.10.

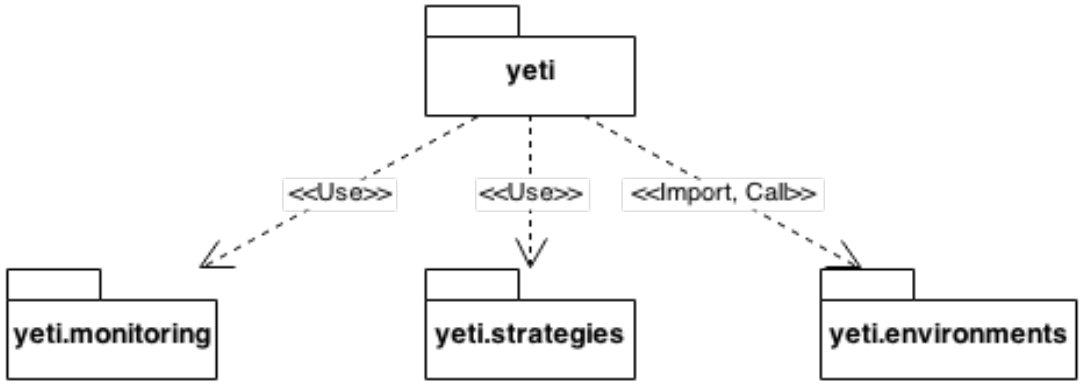


Figure 3.10: Main packages of YETI with dependencies

The most essential classes included in the YETI core infrastructure are:

1. **Yeti:** class is the entry point to the system and contains the main method. It is responsible for launching YETI. It parses the arguments, setups the environment, initializes the testing and delivers the reports of the test results.
2. **YetiLog:** It is the class that facilitate printing debug and testing logs of the current test session.
3. **YetiLogProcessor:** It is an interface for processing test logs.
4. **YetiEngine:** class binds YetiStrategy and YetiTestManager together (Figure 10). YetiEngine, YetiStrategy and YetiTestManager are responsible for the actual testing process. YetiTestManager makes the actual calls based on the YetiEngines configuration and delegates to the YetiStrategey to generate the test data and select the routines (constructors and methods) to test.
5. **YetiTestManager:** class carries out testing using test data provided by YetiStrategy.
6. **YetiProgrammingLanguageProperties:** It is a place holder for all language related instances during test.
7. **YetiInitializer:** It is an abstract class for test initialization.

3.3.1.2 YETI Strategy

Strategy section contains seven different strategies and inputs to the tested methods is defined by one of the selected strategy. The strategies sub package of Yeti package consist of the following classes.

1. **YETI Strategy:** It is an abstract class which provides an interface for the strategies in YETI.
2. **YETI Random Strategy:** In this mode of testing codes the testing tool incurs random values for testing, where two probabilities can be used by the user namely null value probabilities and the percentage of the new objects that were created during the process in order to use when a testing is performed by any user.
3. **YETI Random Plus Strategy:** Involves the usage of Interesting values that are in random ranging from -10 to +10 in case of java binding and it could be controlled by the user where random plus is considered to be the extension of Pure random testing and it is considered to be the most significant strategy to identify errors.
4. **DSSR Strategy:** It is a class which implements the Dirt Spot Sweeping Random strategy. It is an extended class of YetiRandomPlusStrategy.
5. **ADFD Strategy:** It is a class which implements the Automated Discovery of Failure Domain strategy. It is also an extended class of YetiRandomPlusStrategy.
6. **Random Decreasing Strategy:** It is an extended random plus in which the probability values start from 100% and keeps decreasing till it reaches 0% when it reaches the end of testing process.
7. **Random Periodic Strategy:** It is a testing mode in which both the probabilities decrease and increase randomly while the testing is carried out.

3.3.1.3 Language-specific Bindings

The language-specific binding includes classes that are responsible for modelling a programming language.They are language independent and can be extended to add support for a new language.

1. **YetiVariable:** represents a regular variable in YETI and it is a subclass of YetiCard.
2. **YetiType:** represents a type of data (Integer, String, Boolean etc).
3. **YetiRoutine:**represents a super type for routines, which can be functions, methods or constructors. A routine has a name, a return type and a list of its arguments types.
4. **YetiModule:** represents a module under test. It stores a list of the modules routines to test.
5. **YetiName:** represents a unique name assigned to each YetiRoutine instance in YETI.
6. **YetiCard:** represents a card, which is a YETI specific concept that can map either onto a wildcard (a late bound value) or to a YetiVariable. A card has a type and an identifier.
7. **YetiIdentifier:**represents an identifier for a YetiCard instance.

3.3.2 Construction of Test Cases

YETI construct test cases by creating objects of the classes under test and randomly calling its methods with random inputs according to its parameter's-space. 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, in its simplest random strategy, calls *Math.random()* method to generate an arithmetic value which is converted to the required type using casting rule of Java language. However, if the method under test needs an object of a user-defined class as a parameter then YETI calls its constructor or method to generate object of that class at run time. It may be possible that the constructor require another object and in that case YETI will recursively calls the constructor of that object. This process is continued until an object with blank constructor, constructor with only primitive types or the set level of recursion is reached.

3.3.3 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 in any order. For example, to save processing power and reduce overhead for a test session, command line option -nologs can be use to bypass real-time logging. The following table 3.1 describes few of the most common command-line options available in YETI.

Table 3.1: YETI command line options

Options	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

3.3.4 YETI Execution

YETI being developed in Java is highly portable and can easily run on any operating system with Java Virtual Machine (JVM) installed. 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 associated *.jar library files* particularly *javassist.jar* in the *CLASSPATH* to help JVM in identifying the YETI source. The typical command to invoke YETI is given in figure 3.11.

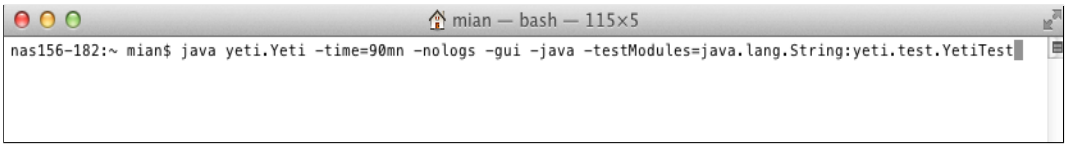


Figure 3.11: Command to launch YETI from CLI

In this particular command YETI tests java.lang.String and yeti.test.YetiTest modules for 90 minutes using the default random strategy. For details of other options please see table 3.1. 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 execution. Figure 3.12 shows the equivalent of above command in GUI mode.

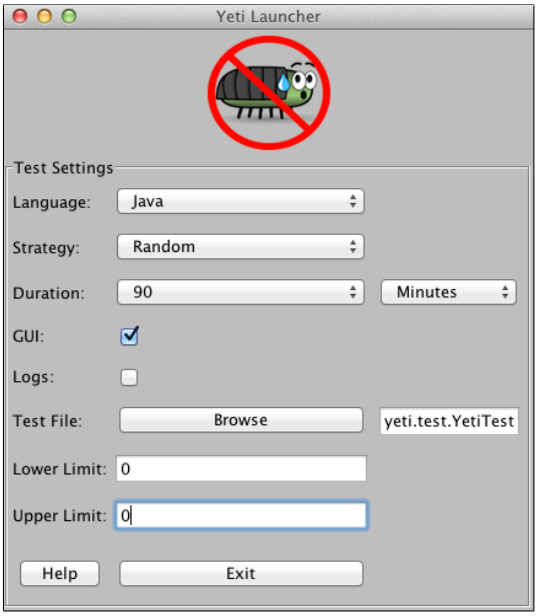


Figure 3.12: Command to launch YETI from GUI

As a result of both the above commands YETI launch its own GUI window and

start testing the assigned programs.

3.3.5 YETI Test Oracle

Oracles in YETI are language dependant. YETI uses two approaches for oracle (pass/fail judgement). In the presence of program specifications, YETI checks for inconsistencies between the code and the specifications. In the absence of specifications YETI checks for assertion violations if assert statements are included by the programmer. However in the absence of both specifications and assertions YETI performs robustness testing that considers any undeclared runtime exceptions as failures.

3.3.6 YETI Report

YETI gives a complete test report after execution of each test. The report contain all the successful calls with the name of the routines and the unique identifiers for the parameters in each execution. These identifiers are recorded with the assign value to help in debugging the identified fault.

```
java.lang.String v286=java.lang.String.valueOf(v285); // time:1248634864647
java.lang.String v301=java.lang.String.valueOf(v101); // time:1248634864697
yeti.test.YetiTest v309=new yeti.test.YetiTest(); // time:1248634864701
char v310='\u1da1'; // time:1248634864702
v309.printChar(v310); // time:1248634864702
double v348=2.1271971229466633d; // time:1248634864728
java.lang.String v349=java.lang.String.valueOf(v348); // time:1248634864729
java.lang.String v388=java.lang.String.valueOf(v310); // time:1248634864986
java.lang.String v400=java.lang.String.valueOf(v122); // time:1248634864991
```

Figure 3.13: YETI successful method calls

YETI separates the bugs from successful executions to simplify the test report. This approach helps debuggers to easily track the origin of the problem and rectify it. When a bug is identified during testing YETI saves that state and present it in the bug report. The information includes all the identifiers of the parameters the method call had at the time of execution. It also report the time at which the exception occurs.

```
java.lang.Double v1136=java.lang.Double.valueOf(v1135); // time:1248634867661
/**BUG FOUND: RUNTIME EXCEPTION**/ // time:1248634867662
/**YETI EXCEPTION - START
java.lang.NumberFormatException: empty String
        at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)
        at java.lang.Double.valueOf(Unknown Source)
YETI EXCEPTION - END**/
/** original locs: 1741 minimal locs: 18**/
}
```

Figure 3.14: YETI bug reports.

3.3.7 YETI Graphical User Interface

YETI supports a GUI that not only allows test engineers to monitor the current test session but also to modify its characteristics in real time during test execution. It is useful to have the facility of modifying or aborting the test parameters at run time and observing the test behaviour in response. The figure 3.3.7 present the YETI GUI comprising of six numbered windows (the windows are labelled for illustration purposes only).

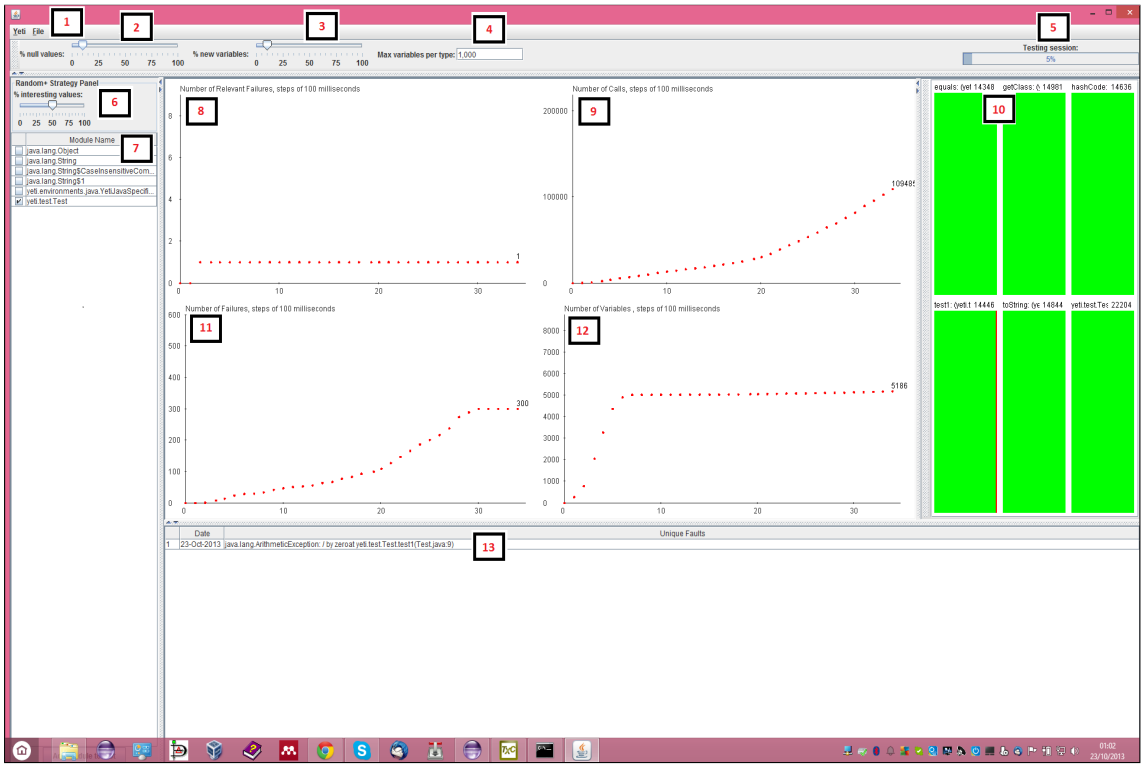


Figure 3.15: GUI of YETI Tool

1. Menu bar:

(a) Yeti menu:

(b) File menu:

2. Standard toolbar:

   (a) Slider: %null values displays probability to use a null instance at each variable. The probability is set before the testing by using the option probabilityToUseNullValue. The default probability is 1.

   (b) Slider % new variables displays probability to create new instances at each call. Same as %null values, it is set before the testing by using the option newInstanceInjectionProbability and the default value is 1.

   (c) Text box Max variables per type displays the cap on the number of instances for any given type. User can modify the sliders and text box during the testing according different test strategies.

   (d) The progress bar Testing session indicates the percentage of the test progress.

3. Module Name shows the list of the modules under the test. The modules with ticks are the modules under test. The module names also show all the class names in the test module.

4. displays the number of unique of bugs detect in the module under test over time.

5. displays the number of calls to the module under test over time.

6. displays the number of failures over time. They are not both related to the module under test.

7. displays the number of object instances created by YETI over time.

8. displays all the routines in the module under test with a rectangle. Each rectangle presents the results of calls of the routine. The rectangle can have in 4 colors. Black indicates no any calls of this routine. Green indicates that has successful calls of this routine. Red indicates that this routine is called unsuccessfully which means that the call to this routine results in an exception. Yellow indicates undecidable calls, for example if a call cannot finish in predefined time and Yeti stops this call, in this case yeti cannot decide this call is successful or unsuccessful. The text next the routines name show

how many calls of this routine and text displays percentage of passed, failed and undescided when the cursor over the rectangle.

9. displays a table which contains the unique faults are detected by Yeti. It records the detail of exceptions.
10. Window No 1 displays the failures of the tested module over time.
11. Window No 2 displays the total number of failures over time. These may be generated from calls not related to the tested module.
12. Window No 3 displays the total number of calls to the tested module over time.
13. Window No 4 displays the total number of variables generated by YETI over time.
14. Window No 5 displays colored rectangles: one for each constructor and method under test. Each rectangle represents the calls to a constructor or a method.
15. The colors in a rectangle have the following meaning:
16. Green indicates successful calls (). A successful call is one that does not raise an exception or if it does, the method or the constructor declares to throw it.
17. Red indicates failed calls (X). A failed call results from raised RuntimeException or one of its subclasses.
18. Yellow indicates undecidable calls (?). A call is undecidable if for some reason it takes too long to complete and needs to be stopped, or if a YetiSecurityException (custom exception in YETI) is thrown.

3.3.8 Summary

In this chapter we define random testing and the various ways of performing random testing. We then showed how the automated testing tools implement random technique for software testing. Finally the chapter explains in detail the YETI tool which is being used in this study. The main features of all the tools are noted in the following table.

Table 3.2: Summary of automated testing tools

Tool	Language	Input	Strategy	Output	Benefits
JCrasher	Java, JML	Program	Method type to predict input, Randomly find values of crash	TC	Automated TC, Use of Heuristic Rules
Jartege	Java	Classes	Random strategy with controls like weight etc.	TC, RT	Quick, automated
Eclat	Java	Classes, pass TC	Create model from TC, execute each candidate on the model	Faulty TC	produce output text, JML
Quickcheck	Haskell	Specifications and Functions	Specification hold to random TC?	Pass/Fail	Easy to use, program documentation
Randoop	Java, .NET	Specifications, code and time	Generate and execute methods & give feedback for next generation	Fault TC, RT	
AgitarOne	Java	Package, time and manual TC	Analyse SUT with auto and provided data in specified time	TC, RT	Eclipse plug-in & easy to use
AutoTest	Java	Classes, time and manual TC	Heuristic rules to evaluate contracts	violations, RT	GUI in HTML, easy to use
TestEra	Java	Specifications, integer & manual TC	Check contracts with specifications	Contracts violations	short report with faulty TC only
Korat	Java	Specifications and manual tests	Check contracts with specifications	Contracts violations	GUI, short report with faulty TC only
YETI	Java, .NET, JML	Code, Time	RandomPlus, Random	Traces of found faults	GUI, give faulty examples, Quick

Chapter 4

Dirt Spot Sweeping Random Strategy

4.1 Introduction

The success of a software testing technique is mainly based on the number of faults it discovers in the 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. [12] 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. [17] 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 [13], Feedback directed Random

Test Generation [106], Mirror Adaptive Random Testing [18] and Quasi Random Testing [22] stress the need for test case selection covering the whole input domain to get better results.

In this chapter 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] 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 [98] 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[2]. To evaluate our approach, we tested 30 times each one of the 60 classes of 32 different projects from the Qualitas Corpus[3] 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.

4.2 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.

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

[2] http://www.yetitest.org

[3] http://www.qualitascorpus.com

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.

4.2.1 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 [21]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [28, 32]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [31]. YETI [96, 101], AutoTest [30, 75], QuickCheck [34], Randoop [106], JArtege [95] 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 [90] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [30, 43, 44, 59, 93] 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 [30, 99] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

4.2.2 Random Plus Strategy (R+)

The random+ strategy [75] is an extension of the random strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [7] 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`. These special values can add a significant improvement to any testing method. For example:

```
public void test (int arg) {
arg = arg + 1;
int [] intArray = new intArray[arg];
...
}
```

In the above piece of code, on passing interesting value `MAX_INT` as argument the code increment it by 1 making it a negative value and thus an error is generated when the system try to build an array of negative size.

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 pre-defined 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 [32].

4.2.3 Dirt Spot Sweeping (DSS)

Chan et al. [12] found that there are patterns of failure-causing inputs across the input domain. Figure 4.1 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 [13, 18, 22], 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 for the next couple of tests to find another fault from the same region.

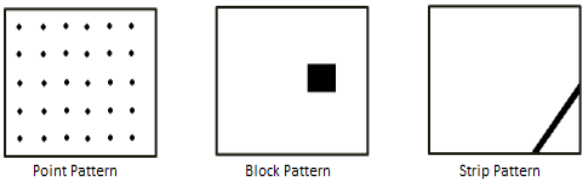


Figure 4.1: Failure patterns across input domain [17]

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.

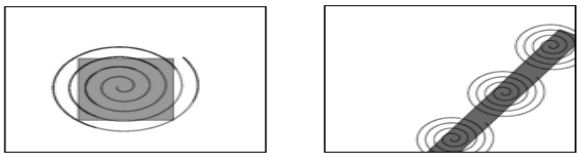


Figure 4.2: DSSR covering block and strip pattern

Figure 4.2 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.

4.2.4 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 [77]. 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.

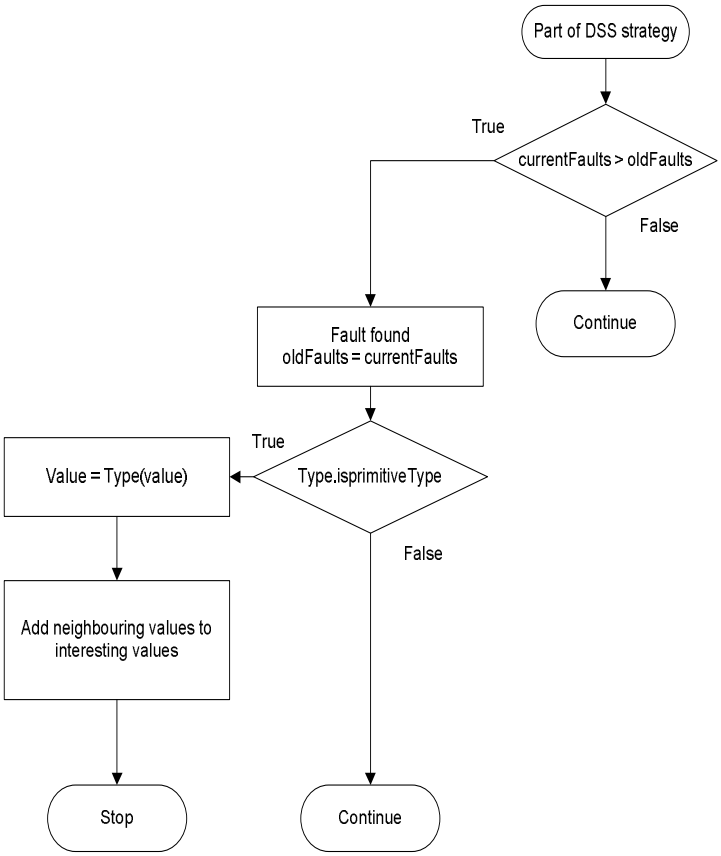


Figure 4.3: Working mechanism of DSSR Strategy

The flowchart presented in Figure 4.3 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 4.1 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 4.1: 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 X + " " " " + X X.toUpperCase() X.toLowerCase() X.trim() X.substring(2) X.substring(1, X.length()-1)

4.2.5 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 be-

cause 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.

4.3 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 [101]. A number of hitherto faults have successfully been found by YETI in various production softwares [97, 99].

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 4.4.

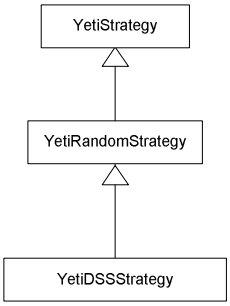


Figure 4.4: Class Hierarchy of DSSR in YETI

4.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [75]. 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.

4.4.1 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?

4.4.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [119]. 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 4.2 while test details of the classes is presented in table 4.3. Line of Code (LOC) tested per class and its total is shown in column 3 of table 4.3.

Every class is evaluated through 10^5 calls in each test session.[4] Because of the absence of the contracts and assertions in the code under test, similar approach as used in previous studies [99], 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.

4.4.3 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 [17] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [20, 25]. 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 [80].

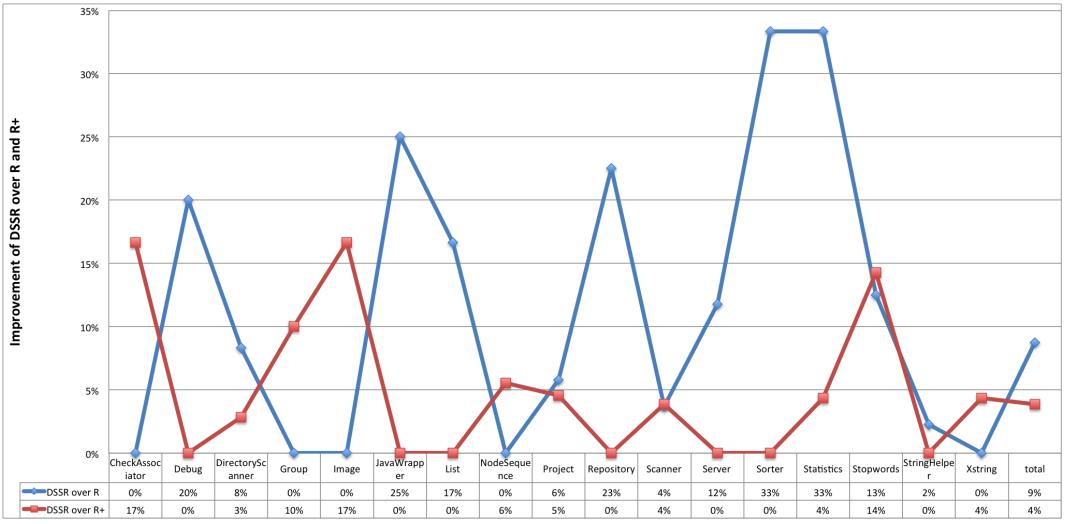
The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print

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

Table 4.2: 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

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 [30, 33, 106]. 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 4.5: Improvement of DSSR strategy over Random and Random+ strategy.

4.5 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 4.3. 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.

Table 4.3: 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

4.5.1 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 4.5 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.

$$\frac{Average faults_{(DSSR)} - Average faults_{(R)}}{Average faults_{(R)}} * 100 \tag{4.1}$$

$$\frac{Average faults_{(DSSR)} - Average faults_{(R+)}}{Average faults_{(R+)}} * 100 \tag{4.2}$$

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.

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

T-tests applied to the data given in Table 4.4 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

Table 4.4: 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	

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.

4.5.3 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.

4.6 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 [101]. 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 [99].

4.7 Related Work

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [39, 34, 104]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [59]. Some of the well known automated tools based on random strategy includes Jartege [95], Eclat [104], JCrasher [39], AutoTest [30, 31] and YETI [99, 101].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [13, 18, 21, 22, 23]. 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 [132] 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 [75]. These interesting values includes border values which have high tendency of finding faults in the given SUT [7]. Results obtained with R+ strategy show significant improvement over random strategy [75]. 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 [44, 57, 58]. Arcuri et al. [5], 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 [118] 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 [99, 117, 119].

4.8 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 5

Automated Discovery of Failure Domain

5.1 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 [8]. Most of the modern black-box testing techniques execute the 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 [32, 50]. It involves minimum or no overhead in input selection and lacks human bias [59, 79]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [94] and discovery of lower number of faults [24]. 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 [17]. Similarly Restricted Random Testing (RRT) [13], Mirror Adaptive Random Testing (MART) [20], Adaptive Random Testing for Object Oriented Programs (ARTOO) [32], Directed Adaptive Random Testing (DART) [55], Lattice-based Adaptive Random Testing (LART) [81] and Feedback-directed Random Testing (FRT) [106] 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. [12] 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 5.1 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.

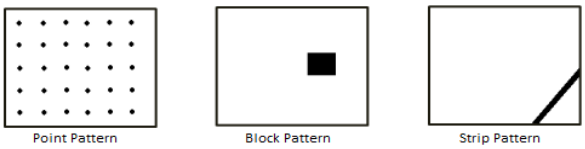


Figure 5.1: Failure domains across input domain [12]

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 chapter 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.

5.2 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 5.2.

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

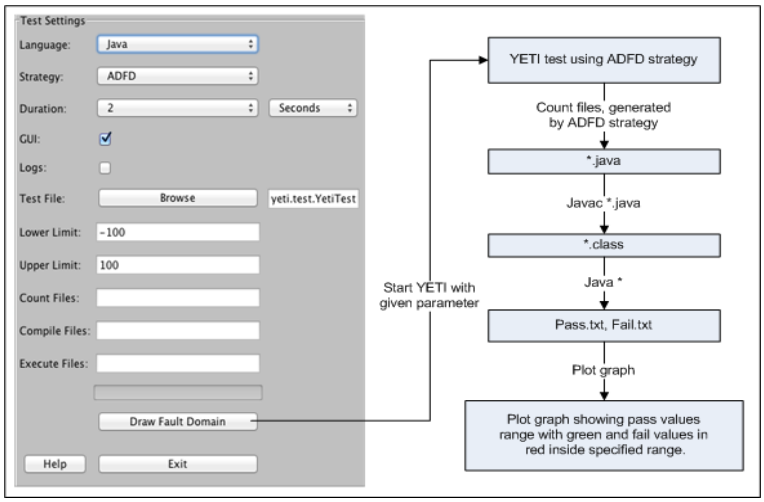


Figure 5.2: Work flow of ADFD strategy

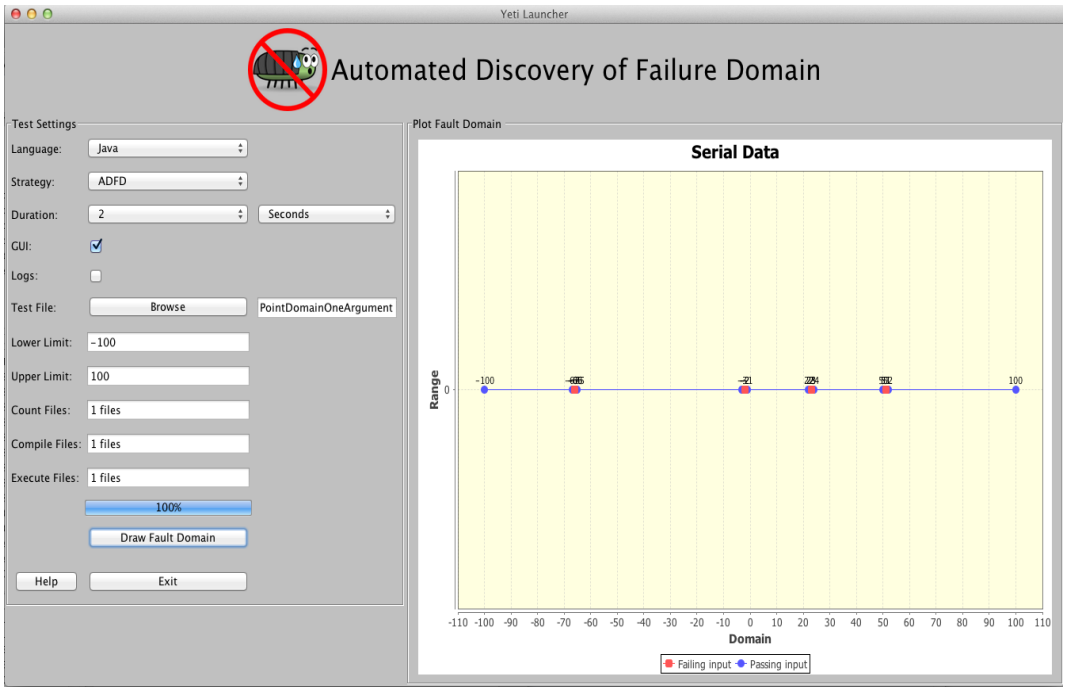


Figure 5.3: Front-end of ADFD strategy

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 [97, 99]. 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 [54]. 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.

5.3 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 [97, 99, 101, 96] for more details on YETI tool.

5.3.1 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.

5.3.2 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.

5.3.3 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.

```java
/**
 * 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();
    }
}
```

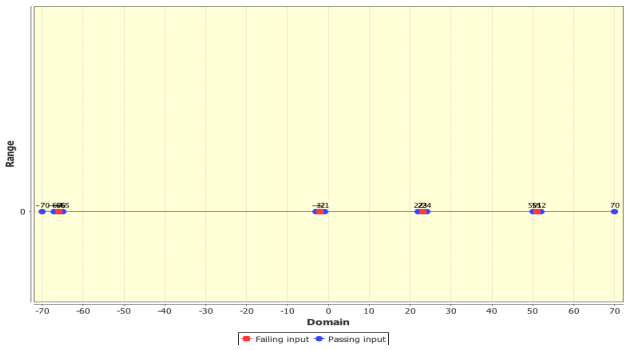


Figure 5.4: ADFD strategy plotting pass and fault domain of the given class

As soon as any one of the above four faults are discovered the ADFD strategy generate a dynamic program given in Appendix A.1 (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 5.4 that ADFD strategy not only finds all the faults but also the pass and fail domains.

5.4 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].

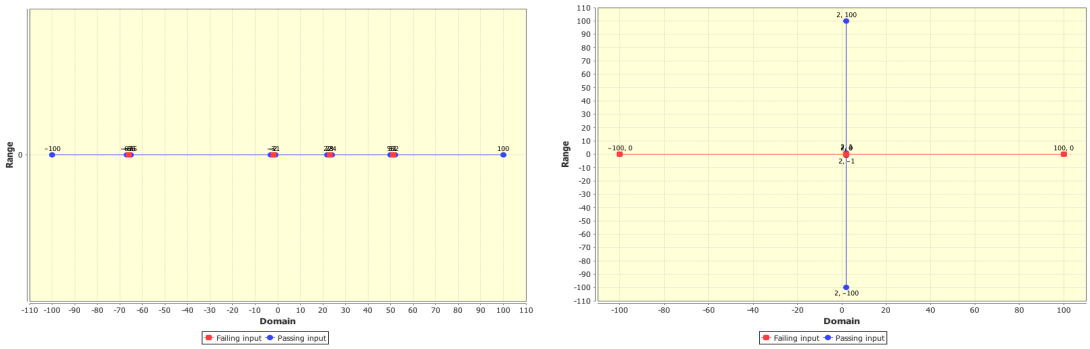
To elucidate the results, six programs were developed so as to have separate program for one and two-dimension point, block and strip fault domains. The code of selected programs is given in Appendix A.1 (2-7). The experimental results are presented in table 5.1 and described under the following three headings. 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 A.1 (2-7). The experimental results are presented in table 5.1 and described under the following three headings.

Table 5.1: Pass and Fail domain with respect to one and two dimensional program

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)

Point Fault Domain: Two separate Java programs Pro2 and Pro3 given in Appendix A.1 (2, 3) were tested with ADFD strategy in YETI to get the findings for point fault domain in one and two-dimension program. Figure 5.5(a) present range of pass and fail values for point fault domain in one-dimension whereas Figure 5.5(b) present range of pass and fail values for point fault domain in two-dimension

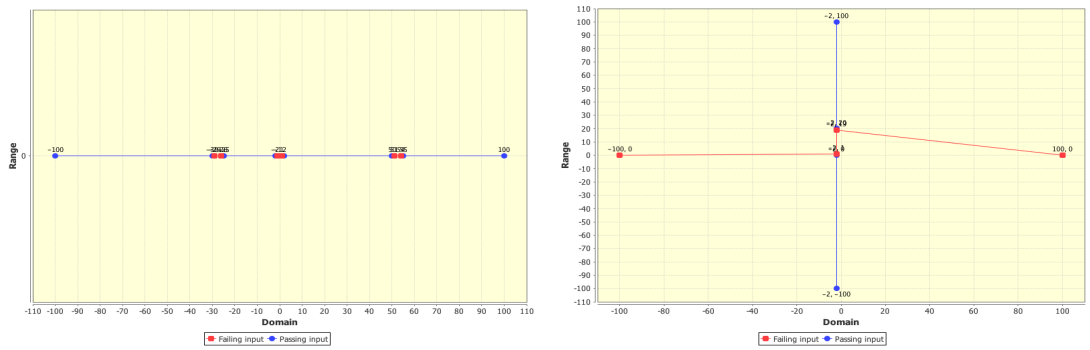
program. The range of pass and fail values for each program in point fault domain are given in (Table 5.1, Serial No. 1).



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

Figure 5.5: Chart generated by ADFD strategy presenting point fault domain

Block Fault Domain: Two separate Java programs Pro4 and Pro5 given in Appendix A.1 (4, 5) were tested with ADFD strategy in YETI to get the findings for block fault domain in one and two-dimension program. Figure 5.6(a) present range of pass and fail values for block fault domain in one-dimension whereas Figure 5.6(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 5.1, Serial No. 2).

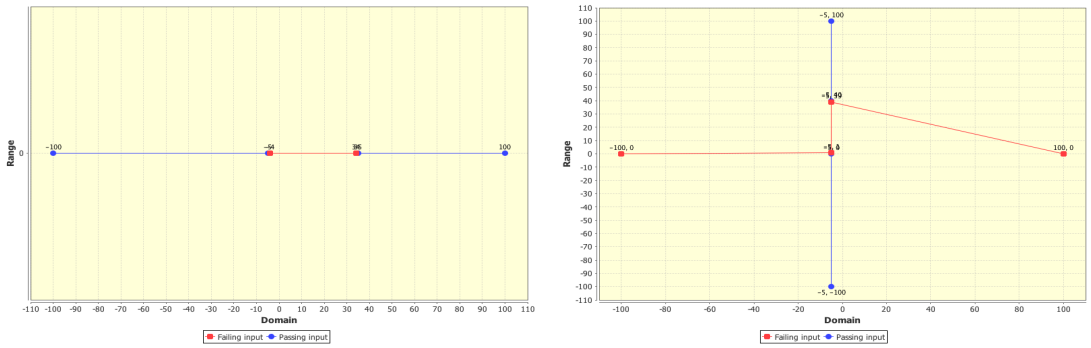


(a) One dimension module (b) Two dimension module

Figure 5.6: Chart generated by ADFD strategy presenting block fault domain

Strip Fault Domain: Two separate Java programs Pro6 and Pro7 given in Appendix A.1 (6, 7) were tested with ADFD strategy in YETI to get the findings for strip fault domain in one and two-dimension program. Figure 5.7(a) present range

of pass and fail values for strip fault domain in one-dimension whereas Figure 5.7(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 5.1, Serial No. 3).



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

Figure 5.7: Chart generated by ADFD strategy presenting Strip fault domain

5.5 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 5.4), 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.

5.6 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[29],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.

5.7 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 [91, 94]. 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 [12].

Identification, classification of pass and fail domains and visualisation of domains have not received due attention of the researchers. Podgurski et. al., [109] 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 [69]. 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.

5.8 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 6

Invariant Guided Random+ Strategy

6.1 Introduction

6.2 Invariant Guided Random+ Strategy

6.2.1 Daikon

6.2.2 Random Plus Strategy (R+)

The random+ strategy [75] is an extension of the random strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [7] 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 pre-defined 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 [32].

6.2.3 Structure of the Invariant Guided Random+ Strategy

6.2.4 Explanation of IGRS strategy on a concrete example

6.3 Implementation of the IGRS strategy

6.4 Evaluation

6.4.1 Research questions

1. A
2. B
3. C

6.4.2 Experiments

6.4.3 Performance measurement criteria

6.5 Results

6.5.1 Answer A

6.5.2 Answer B

6.5.3 Answer C

6.6 Discussion

6.7 Related Work

6.8 Conclusions

Chapter 7

Conclusion and Future Work

7.1 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 [8]. 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 [32], [50]. It involves minimum or no overhead in input selection and lacks human bias [59], [79]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [94] and discovery of lower number of faults [24]. 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 [17]. Similarly Restricted Random Testing (RRT) [13], Mirror Adaptive Random Testing (MART) [20], Adaptive Random Testing for Object Oriented Programs (ARTOO) [32], Directed Adaptive Random Testing (DART) [55], Lattice-based Adaptive Random Testing (LART) [81] and Feedback-directed Random Testing (FRT) [106] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

Appendix A

Sample Title

A.1 Appendix hello

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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.

[2] Hiraral Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[3] Ashfaque Ahmed. *Software testing as a service*. CRC Press, 2010.

[4] 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.

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

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

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

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

[9] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE'07*, pages 85–103. IEEE, 2007.

[10] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] Insang Chung and James M Bieman. Automated test data generation using a relational approach.

[28] 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] 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.

[30] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Pro-*

ceedings of the 2007 international symposium on Software testing and analysis, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] Ralston T Craigen D, Gerhart S. On the use of formal methods in industry – an authoritative assessment of the efficacy, utility, and applicability of formal methods to systems design and engineering by the analysis of real industrial cases. In *Report to the US National Institute of Standards and Technology*, 1993.

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

[40] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194. ACM, 2007.

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

[42] Michael R Donat. Automating formal specification-based testing. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 833–847. Springer, 1997.

[43] 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.

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

[45] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

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

[47] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

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

[49] 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 USENIX Windows System Symposium*, pages 59–68, 2000.

[50] 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.

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

[52] Robert T Futrell, Linda I Shafer, and Donald F Shafer. *Quality software project management*. Prentice Hall PTR, 2001.

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

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

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

[56] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631. IEEE, 2007.

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

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

[59] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[60] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[61] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.

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

[63] 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.

[64] Zhenyu Huang. Automated solutions: Improving the efficiency of software testing, 2003.

[65] 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.

[66] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The alloy constraint analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733. IEEE, 2000.

[67] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, 2001.

[68] Pankaj Jalote. *An integrated approach to software engineering*. Springer, 1997.

[69] 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.

[70] Sarfraz Khurshid and Darko Marinov. Checking java implementation of a naming architecture using testera. *Electronic Notes in Theoretical Computer Science*, 55(3):322–342, 2001.

[71] 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.

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

[73] Bogdan Korel and Ali M Al-Yami. Assertion-oriented automated test data generation. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 71–80. IEEE, 1996.

[74] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239. IEEE, 1998.

[75] 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.

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] Johannes Mayer, Ralph Guderlei, et al. Test oracles using statistical methods. In *SOQUA/TECOS*, pages 179–189, 2004.

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

[84] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[85] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[86] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. Eiffel: object-oriented design for software engineering. In *ESEC'87*, pages 221–229. Springer, 1987.

[87] Barton P Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54. ACM, 2006.

[88] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

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

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

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

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

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

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

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

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

[97] M. Oriol. York extensible testing infrastructure, 2011.

[98] 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.

[99] 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.

[100] 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.

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

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

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

[104] 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.

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

[106] 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.

[107] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

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

[109] 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.

[110] Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[111] 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.

[112] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64–73. IEEE, 1997.

[113] 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.

[114] Harry Robinson. Finite state model-based testing on a shoestring. In *Proceedings of the 1999 International Conference on Software Testing Analysis and Review (STARWEST 1999)*, 1999.

[115] 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.

[116] Tassey. The economic impacts of inadequate infrastructure for software testing. Technical report, National Institute of Science and Technology, 2002.

[117] 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.

[118] 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.

[119] 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.

[120] Dave Towey. Software quality assurance.

[121] 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.

[122] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software-Practice and Experience*, 30(1):61–79, 2000.

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

[124] 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.

[125] 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.

[126] Jeffrey M Voas and Gary McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

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

[128] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703–711, 1991.

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

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

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

[132] 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.

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