New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group

Department of Computer Science

University of York, UK

March 2014

A thesis submitted for the degree of Doctor of Philosophy

Abstract

The thesis presents new techniques for improving the effectiveness of automated random testing, evaluates the efficiency of these techniques and proposes directions for future work.

Software testing, the process of evaluating the correctness and quality of a software or its component, is a well used approach for detecting software failures. Testing involves generation and execution of test inputs and evaluation of results for correctness either manually or by automatic means. Automated software testing save time and human effort involved in manual testing. There are two major challenges in software testing: the generation of appropriate test inputs and evaluation of the test results. This thesis both points.

To address the issue of selecting appropriate input data a new technique named Dirt Spot Sweeping Random (DSSR) strategy is developed on the assumption that unique failures reside in contiguous block strips. It starts by testing the code at random. When a failure is identified, the DSSR strategy selects the neighbouring input values except duplicate values for the subsequent tests. The selected values sweep around the identified failure, leading to the discovery of new failures in the vicinity. This results in quick and efficient identification of faults in Software Under Test (SUT).

To address the issue of evaluating test results a new technique named as, Automated Discovery of Failure Domain (ADFD) is developed with the capability to find faults as well as the fault domains in a given SUT and provides visualization of the identified pass and fail domains within a specified range in the form of a chart. The new technique is highly effective in testing and debugging and provides an easy to understand test report in the visualized form.

The third new technique proposed in this thesis is, Invariant Guided

Random+ Strategy (IGRS) which is an extended form of a random strategy with boundary values (Random+) strategy guided by software invariants. In this technique, Invariants from the given SUT are automatically collected by Daikon, filtered through DynComp and annotated in the source code as assertions. (The experiments are in progress, the results obtained will be compared with the DSSR, Random and Random+ strategies and the findings will be included in the thesis and abbreviated in the abstract as soon as possible.)

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 spent on my PhD degree at the University of York has been the most joyful and rewarding in my academic career. The institution provided me with everything I needed to thrive: challenging research problems, excellent company, and supportive environment. I am deeply grateful to all those people who shared this experience with me.

Several people have contributed to the completion of my PhD dissertation. 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. Dr. John Clark for his constructive feedback on various reports and presentations. I am also thankful and highly indebted to Prof. Dr. Richard Paige for his generous help, cooperation and guidance during my research at the University of York.

Thanks to my father Prof. Dr. Mushtaq A. Mian who provided a conducive environment, valuable guidance and crucial support at all levels of my educational career and my very beloved mother whose love, affection and prayers have been my most precious assets. I am also 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 Asbat for her company, help and cooperation throughout my stay at York.

I received Departmental Overseas Research Scholarship. The scholarship is awarded to overseas students for higher studies on academic merit and research potential. I am truly grateful to the Department of Computer Science, University of York for financial support which enabled me to complete my PhD program.

Chapter 1

Introduction

Computer processors execute instructions composing programs designed and created by programmers.. The set of all such machine readable instructions of a system is called software. This includes human-understandable instructions (source code) as well as machine-understandable instructions (binary code). Software is often written in high level languages that are close to natural language and are generally portable to multiple architectures. These languages require compiler or interpreter to transform them into an architecture specific, machine language before execution.

Software is an important and essential component of computer system without which no task can be accomplished. Some software are developed for use in simple day to day operations while others are for highly complex processes in specialised fields including education, business, finance, health, science and technology etc. The ever increasing dependency on softwares expect us to believe that softwares are reliable, robust, safe and secure. However, like every other man-made items softwares are also prone to errors. Maurice Wilkes [11], 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 [12]. According to the National Institute of Standards and Technology (NIST), US companies alone bear $59.5 billion loss every year due to software failures [13]. Therefore, software companies leave no stone unturned to ensure the reliability and accuracy of the software before its practical application. The process of evaluating the correctness and quality of the software or its component is called software testing.

Software testing is one of the techniques used during Verification and Validation (V & V) process to ensure that the software adheres to the desired specifications. According to Dijkstra, program testing can be used to show the presence of bugs, but never to show the absence of bugs [15]. It means that, a Software Under Test (SUT) that passes all the tests without giving any 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 in which generation of test data is created at 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. The three main phases of random testing i.e. test data generation, execution and evaluation are shown in Figure 1.1.

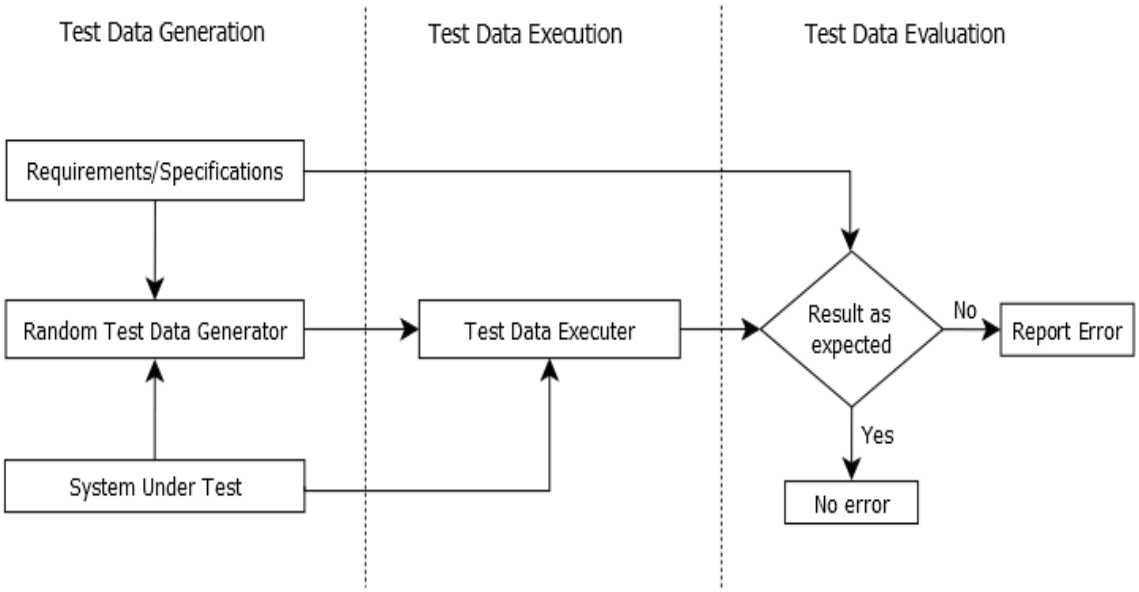


Figure 1.1: Three main phases of random testing

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 leads 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 [16, 17]. 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 should select the test data so that all paths of the SUT are executed at least once [18]. The implementation of the strategy resulted in higher standards of test quality and a large number of test strategies were subsequently developed such as 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 [19]; 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 [20]. 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 [21]. According to Godefroid et al. [22] “Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs”.

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [23]. Myers et al. [24] mentioned, “probably the poorest methodology of all is random-input testing”. However, Ciupa et al. [25] reported that the above stated statement of Myers et al. is based on intuition and lacks any experimental evidence. The criticism motivated the researchers to look into various aspects of random testing for evaluation and possible improvement. Adaptive

random testing (ART) [4], Restricted Random Testing (RRT) [26], Feedback Directed Random Testing (FDRT) [27], Mirror Adaptive Random Testing (MART) [1] and Quasi Random Testing (QRT) [28] are a few of the enhanced random testing techniques reported in the literature.

Random testing is also considered weak in providing high code coverage [29, 30]. 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 [22].

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. The 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 presenting the fault domains. In the current research study, a fully automated testing strategy named, "Automated Discovery of Failure Domain (ADFD)" is developed with the ability to find the faults as well as the fault domains in a given SUT and provides visualisation 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 Chapter 5. The experimental results prove that ADFD strategy automatically performs 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) is 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, filtered by using DynComp and annotated in to source code as assertions. The IGRS is implemented in YETI and generates values in compliance with the added assertions. Experimental results presented in Chapter 6 indicate improved features of IGRS in terms of higher code coverage and identification of

subtle errors that R, R+ and DSSR strategies are either unable to accomplish or require larger duration.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 presents an overall view of software testing. Software testing is introduced with particular reference to its level, purpose, perspective and execution. Various types of software testing followed by major stages of testing, including test data generation, execution, oracle and report production are reviewed. Finally, a summary of the chapter is presented.

Chapter 3 is the follow up of Chapter 2 with particular focus on literature relevant to random testing. It includes various versions of random testing and the most commonly used automated testing tools based on random algorithms. The York Extensible Testing Infrastructure (YETI), used as a tool in our experiments, has been duly focused.

Chapter 4 describes Dirt Spot Sweeping Random (DSSR) strategy. The proposed new testing technique is implemented in YETI tool. Experimental evidence are presented in support of the effectiveness of DSSR strategy in finding faults as compared with random and random+ strategies. In majority of the classes DSSR strategy indicates higher fault-finding ability than random and random+ strategies.

Chapter 5 presents Automated Discovery of Failure Domain (ADFD) strategy. The proposed new testing technique, implemented in YETI, finds faults and fault domains in a specified limit and plots them on a chart. Experimental evidence is presented in support of ADFD strategy applied to several one and two dimensional programs.

Chapter 6 presents the Invariant Guided Random+ Strategy (IGRS), a newly proposed testing technique that automatically generates invariants of SUT using Daikon,

filter and annotate the invariants in the source code to better support testing process. The IGRS technique like DSSR and ADFD is also implemented in YETI. Experimental study is presented in which IGRS effectiveness of finding faults is compared with the random, random+ and DSSR strategies. For the majority of the classes IGRS indicated higher fault-finding ability than the rival strategies.

Chapter 7 provides conclusion of the thesis

Chapter 8 Gives proposals for future research in the relevant field.

Appendix A ADFD logic implementation and java programs with point, block and strip fault domain.

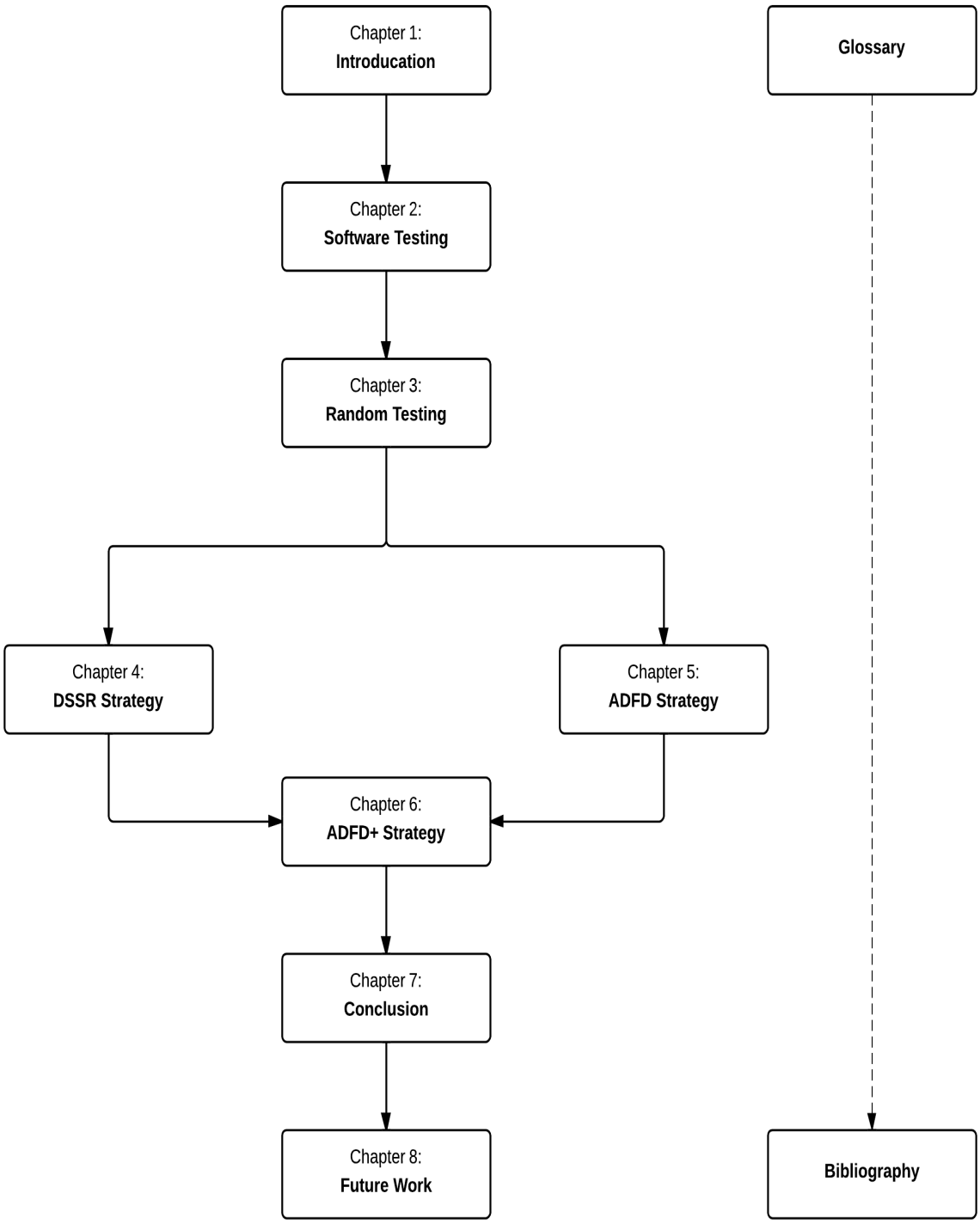


Figure 1.2: Structure of thesis outline

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 it is finalised. Errors cannot be tolerated in software because a single error may cause a large upset in the system. 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 [32]. To achieve high quality, a 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.

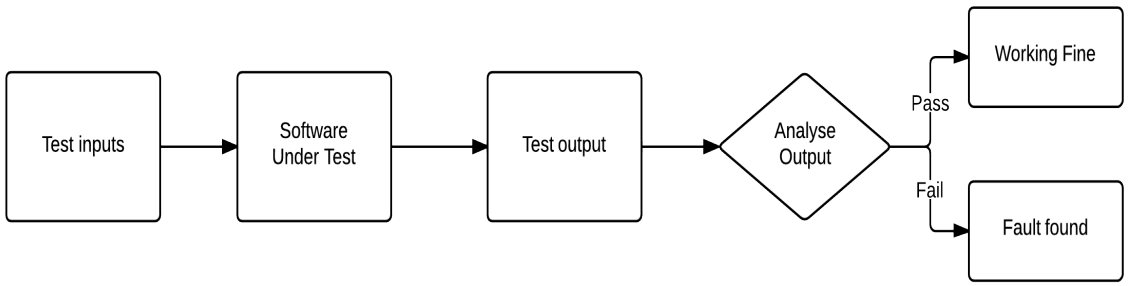


Figure 2.1: The process of software testing

In the IEEE standard glossary of software engineering terminology [14], 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 the specified requirements and results". The process of software testing in its simplest form is shown in Figure 2.1. A successful test is the one that fails a software or identify a fault in the sonftware [33], where fault denotes the error made by programmers during software development [14].

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 a 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" [15]. In other words, a SUT that passes all the tests without giving a single error is not guaranteed to contain no error. The testing process, however, increases reliability and confidence of users in the tested product.

Table 2.1: Parts of Software Testing

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

2.1 Definitions

This section presents important definitions:

2.1.1 Test Plan

Test plan is a document which defines the goal, scope, method, resources and time schedule of testing [34]. In addition, it includes the testable deliverables and the associated risk assessment. The test plan explains, *who, when, why* and *how* to perform a specific activity in the testing process.

2.1.2 Input Domain

The input domain comprises of all possible inputs for a software, including all the global variables, method arguments and the externally assigned variables, like keyboard inputs etc. For a given program P with input vector $P = \{x1, x2, ..., xn\}$, having $\{D1, D2, ..., Dn\}$ as the domain of each input so that $x1 \in D1, x2 \in D2$ and so on. The domain D of a function is 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 which delineates the input, action and expected output corresponding to that input [35]. After executing the test case, if the output obtained comply with the expected output, the test case is pass and the functionality is working correctly, otherwise the test case is fail, which represents identification of fault. Generally, a series of test cases, also known as test suite, are required to be executed for establishing the desired level of quality.

2.2 Software Testing Levels

Unit testing, integration testing and system testing are the three main levels of software testing reported in the literature [7]. Unit testing deals with evaluation of code piece-by-piece and each piece is considered as independent unit. Units are combined together to form components. Integration testing is performed to make sure that integration of units in a component is working properly. Finally, system testing ensures that the system formed by the combination of components proceeds properly to give the required 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 that functional aspect are 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 checking its ability to process different nature of tasks [36]. Finally, compatibility testing is performed to see the interaction of software with the underlying operating system.

2.4 Software Testing Perspective

Software testing can be divided into white-box and black-box testing based on the perspective taken.

2.4.1 White-box testing

In white-box or structural testing, the testers must know about the complete structure of the software so that they may make necessary modifications, if so required. Test cases are derived from the code structure and test passes only if the results are correct and the proper code is followed during test execution [37]. Some commonly used white-box testing techniques are as follows:

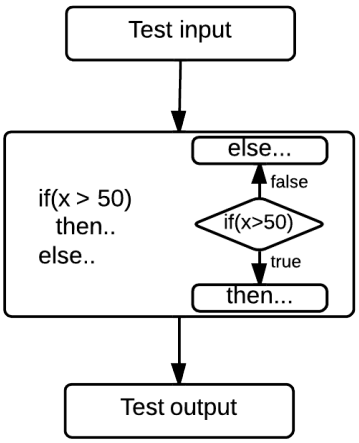


Figure 2.2: White-box testing

2.4.1.1 Data Flow Analysis

Data Flow Analysis (DFA) is a testing technique which focuses on the input values by observing the behaviour of respective variables during the execution of the SUT [38]. In this technique a Control Flow Graph (CFG), graphically representing all possible states of a program, drawn to determine the paths that are traversed by the program during its execution. Test cases are generated and executed to verify its conformance with CFG.

Generally, program execution includes data input, its processing with the defined algorithm and output of results. The process can be looked into as data-flow from input to output where data may transform into several intermediary steps before reaching the final state. The process is prone to several errors e.g. references made to non existing variables, values assigned to undeclared variables or change of variables in undesired manner. Ordered use of data is crucial to ensure that the aforementioned errors do not occur [39].

2.4.1.2 Control Flow Analysis

Control Flow Analysis (CFA) 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 DFA, is drawn to determine the traversable paths by a program during the execution. Test cases are generated and executed

to verify conformance with CFG on the basis of control. Taking the example of following a specific path between two or more available choices at a particular state: efforts are made to ensure that, at least once, the set of selected test cases execute all the possible control choices. The effectiveness of the testing technique depends on measurement of control. Two of the most common measurement criteria defined by Vilkomir et al. are Branch coverage and Condition coverage [40].

2.4.1.3 Code-based fault injection testing

It is a testing technique in which additional instructions are added to the code of the SUT at one or more locations to analyse the software behaviour in response to the anomaly [41]. The process of code addition is called instrumentation which is performed before compilation and execution of software. Code is added for several reasons i.e. to find error handling behaviour of software by injecting faults, to examine the capability of test procedure with respect to the discovery of injected faults and 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 software specifications and test passes if the result is according to expected output [42]. Some commonly used black-box testing techniques are stated below:

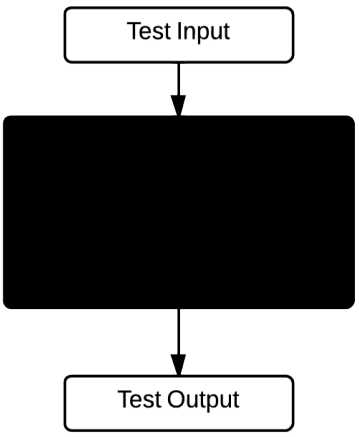


Figure 2.3: Black-box testing

2.4.2.1 Use-case based testing

It is a verification and validation technique which utilizes use-cases of the system to generate test cases. Use-case defines functional requirement at a particular point in the system from actor's perspective. It consists of a sequence of actions to represent a particular behaviour of the system. A use-case format includes brief description and flow of events, preconditions, postconditions, extension points, context and activity diagrams. The use-case contains all the information required for test case, therefore, it can be easily transformed into a test case. Use-case testing is beneficial in terms of cheap generation of test cases, avoidance of test duplication, increased test coverage, easier regression testing and early identification of missing requirements.

2.4.2.2 Partition Testing

It is a testing technique in which the input domain of a given SUT is divided into sub-domains for testing each sub-domain individually. The division is based on software specifications, structure of the code and the process involved in software development [43]. The performance of partition testing is directly dependant on the quality of sub-domain [44]. However, division of input domain into equal partitions is often difficult. To overcome the problem, a new version of partition testing, called Proportional sampling strategy [5] is devised. In this version, the sub-domains vary in size and the number of test cases selected from each partition is directly proportional to the size of the partition. Experiments performed by Ntafos [45] has provided experimental evidence for the better performance of Proportional partition testing.

2.4.2.3 Boundary Value Analysis

Boundary Value Analysis (BVA) is a testing technique based on the assumption that errors may often reside along the boundaries of the input variables. Thus border values are taken as the test data set in BVA. According to IEEE standards [46], boundary value is a value that corresponds to minimum or maximum input, internal or external value specified for a system.

The BVA technique and partition testing are also used simultaneously by choosing test values at the borders of each sub-domain as well as the whole input domain. Reid et al. [47] have provided evidence in support of better performance of BVA compared to partition testing. However, they have indicated that better performance of BVA is based on accurate identification of partition and selection of boundary values.

The following code illustrates the ability of BVA to find a bug. On passing interesting value `MAX_INT` as argument to the *test* method, the code in the method 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

It is a testing technique based on mathematical model which provides the opportunity to handle the specifications mechanically. This feature facilitates the isolation, transformation, assembly and repackaging of the information available in the specifications for use as test cases [50].

The formal specification testing is more productive because of the creation of test cases independent from the code of the SUT [8]. The extra effort of generating test oracle is avoided because of using the available specification model for verifying the test results [51].

2.4.3 Test Oracle

Test oracle is defined as, “a source containing expected results for comparison with the actual result of the SUT” [35]. According to Howden [16], an oracle is a function which verifies if the output from program P is the same as the output from a correct version of P. Test oracles set the acceptable behaviour for test executions [52].

All software-testing techniques depend on the availability of test oracle [8]. 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 associated with the orcale problem are as follows:

1. It is assumed that the test results are observable and comparable with the oracle.
2. Ideally, test oracle would satisfy desirable properties of program specifications [52].
3. A specific oracle to satisfy all conditions is seldom available as rightly pointed out by Weyuker, “truly general test oracles are often unobtainable” [53].

Some of the most common artifacts used as oracles are stated below.

1. Specification and documentation to generate test oracle.
2. Products similar to the SUT but different in algorithm to solve the similar problem.
3. Heuristic algorithms to provide exact results for a set of test cases.
4. Statistical characteristics to generate test oracle.
5. Comparison of the result of one test to another for consistency.
6. Models to generate test oracle for verification of SUT behaviour.
7. Manual analysis by human experts to verify the test results.

2.5 Software Test Execution

Software test execution can be either static or dynamic. In static testing test cases are analysed statically for checking errors without test execution. Besides code, high quality softwares are supplied with documentation including requirements, design, user manual, technical and 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 [57].

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 [58]. Manual testing may be effective in some cases but it is generally laborious, time consuming and error-prone [59]. Additionally, it requires that the testers must have appropriate skills, experience and 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 a testing tool is used to perform the testing process automatically [60]. There are tools which can automate part of a testing process e.g. generation of test cases or execution of test cases or evaluation of results. Other tools are available which can automate the whole testing process. The increase in functionality, productivity and lower cost of production without compromising quality are the desirable features in favour of automating the process of software testing. Automated software testing can be very effective and highly beneficial for any organisation. Its initial cost may be higher, however, a quick return on investment outperforms it and brings the key benefits of cost reduction, higher productivity, availability, reliability and performance. Automated testing is particularly effective when the nature of job is repetitive and is performed on routine basis like unit testing and regression testing, where the tests are re-executed after each modification [61]. The use of automated software testing made it possible to test large volumes of code, which would have been impossible otherwise [62].

2.6 Test Data Generation

Test data generation in software testing is the process of identifying test input data which satisfies the given test selection criterion. A test data generator tool is used to 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 [19]. Various artefacts of the SUT can be considered to generate test data like requirements, model, code etc. 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 [63]. Program analyser performs initial assessment of software prior to testing and may alter the code if so 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. Each type is briefly described in the following section.

2.6.1 Path-wise Test Data Generator

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

In path-wise test data generation, the program path to the selected statement is identified and the input data are generated for evaluating the path which can be either generated automatically or provided by the user. The data generated in path testing expresses boolean behaviour i.e. true or false for a particular node in a path.

A complete path contains multiple sub-domains, each sub-domain consists of test inputs required to traverse the path. The boundary of the sub-domains are ob-

tained by the predicates in the path condition. The test data traversing a certain path in the software are selected from an input space split into a set of subsections.

2.6.2 Goal-oriented Test Data Generators

It is a technique in which the test data is generated to target a specific program point rather than a program path [64]. The tester can select any path among a set of existing paths as long as it reaches to the specified program point. This technique utilizes runtime information for computing accurate test data [65]. Among various methods used in goal-oriented test data generation the following two commonly adopted approaches are briefly described.

2.6.2.1 Chaining Approach

The chaining approach uses data dependent analysis to guide the test data generation. In the process all the related statement are selected automatically by the technique that are affected by the execution of the selected statement under test. The dependant statements are executed before the selected statement to generate the required necessary data for the execution of the statement under test [65].

The chaining approach analyses the program according to the edges and nodes. For each test coverage criterion different initial event sequence and goal nodes are determined. For example, consider the branch (p, q), where p is the starting 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. The the Branch Classification process identifies critical, semi-critical and non-critical nodes for each branch. 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 specified branch.

2.6.2.2 Assertion-oriented Approach

In this approach assertions are added to the program code with the goal to identify program input on which an assertion is violated, indicating a fault in the SUT. An assertion specifies a constraint that applies to some state of a computation which evaluates to either true or false. For example, consider a given assertion A, now 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. It is evaluated as false indicating a fault in the SUT.

It is not always possible to generate test cases that violate assertions. However, experiments have shown that 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 generated test data uncovers an error in the program with violation of an assertion. An assertion is violated because of three reasons: a faulty, a faulty assertion and a faulty precondition.

2.6.3 Intelligent Test Data Generators

Intelligent test data generation is a technique used to overcome the problems associated with traditional data generation techniques like generation of meaningless data, duplicated data and failing to generate complex test data. The approach increases users confidence in the generated test data and the testing process [62]. It performs sophisticated analysis, such as fuzzy logic, neural networks and genetic algorithms on the SUT to assist in finding the appropriate test data. It involves complex analysis to anticipate different situations that may arise at any point. The approach produces test data which satisfy the SUT requirements, however, it consumes more time and resources.

2.6.3.1 Genetic Algorithm

Genetic algorithm is a heuristic that mimics the evolution of natural species in searching for the optimal solution of a problem. The solution sought by the genetic algorithm is the test data that causes execution of a given statement, branch, path and condition in the SUT. The genetic algorithm is guided by control dependencies in the program to search for test data which satisfy test requirements. The

genetic algorithm constructs new test data from previously generated test data. The algorithm evaluates the existing test data, and guide the direction of search by using the programs control-dependence graph [67].

The benefit of the genetic approach is quick generation of test data with focus and direction. New test cases are generated by applying simple operations on existing test cases that are judged to have good potential of satisfying the test requirements. The success of this approach, however, depends heavily on the way in which the existing test data is measured [67].

2.6.4 Random Test Data Generators

Random test data generator is the simplest technique for generation of test data. It has the advantage of being used to generate input data for any type of program. However, random test data generation is based solely on probability and cannot accomplish high coverage as its chances of finding semantically small faults are quite low [3find that reference].

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 lower than that of the second statement. As the structure gets complex so does the probability of its execution. Thus, such semantically small faults are hard to find by using random test data generation.

2.6.5 Search-based Test Data Generation

It is a technique that uses meta-heuristic algorithms to guide generation of test data. In Search-based test data generation technique 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 set goal. Input test values closer to the set goal have low cost values and the other 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 $cost(x) == 0$.

Similarly, finding data to follow a particular path in the code can be considered as the one which satisfy each of the number of predicate at different points. This leads to a cost function which combines the costs at each of the relevant branching points. The approach requires the measurement of state at appropriate points in a programs execution. Moreover, the cost function plays the role of oracle for each targeted test requirement. Consequently, the cost function must change as per requirement. Frequent re-instrumentation of program is required to find test data that fully satisfy common coverage criteria.

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

Random testing was first mentioned in the literature by Hanford in 1970. He reported syntax machine, a tool that randomly generated data for testing PL/I compilers [69]. Later in 1983, Bird and Munoz described a technique to produce randomly generated and self checking test cases [70].

Random testing is a dynamic black-box testing technique in which the software is tested with non-correlated unpredictable test data from the specified input domain [26]. As stated by Richard [71], in random testing, input domain is first identified, then test data are randomly taken from it by means of random generator. The program under test is executed on the test data 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. Fail results of the test cases reflects failure in the SUT.

Figure 3.1: Random Testing

Generating test data by random generator is quite economical and requires less intellectual and computational efforts [25]. Moreover, no human intervention is involved in data generation which ensures an unbiased testing process. However,

generating test cases with out using any background information makes random testing susceptible to criticism. Random testing is criticized for generating many of the test cases that falls at the same state of software. It is also stated that, random testing generates test inputs that violates requirements of the given SUT making it less effective [72, 73]. Myers mentioned random testing as one of the least effective testing technique [33]. However, Ciupa et al. stated [21], that Myers statement was not based on any experimental evidence. Later experiments performed by several researchers [58, 71, 74, 75] confirmed that random testing is as effective as any other testing technique. It is reported [75] that random testing can also discover subtle faults in a given SUT when subjected to large number of test cases. It is 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 [43]. A comparative study conducted by Ntafos [45] concluded that random testing is more effective as compared to proportional partition testing. A prominent work to mention is that of Miller et al. [76], who generated and used random ASCII character streams to test Unix utilities for abnormal termination or non-terminating behaviour. Subsequently the same technique was extended to discover errors in softwares running on X Windows, Windows NT and Mac OS X [77, 78]. Other famous studies using random testing includes low-level system calls [79], and file systems used in missions at NASA [80].

3.1 Various versions of random testing

Researchers have tried various approaches to bring about improved versions of random testing with better performance. The prominent versions of random testing are as follows:

3.1.1 Adaptive Random Testing

Adaptive random testing (ART), proposed by Chen et al. [4] is based on the previous work of Chan et al. [5] regarding the existence of failure patterns across the

input domain. Chan et al. observed that failure inducing inputs formed certain geometrical patterns in the whole input domain which were divided 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 the 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 the statement: $num1 + num2 = 20$. Here multiple values of $num1$ and $num2$ can lead to the fault value 20.

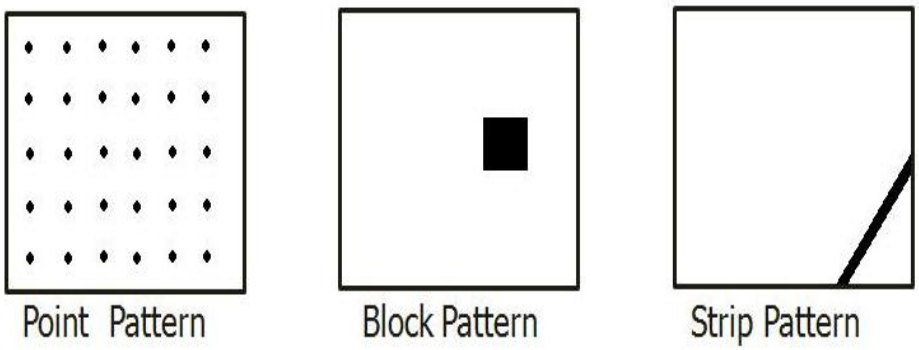


Figure 3.2: Patterns of failure causing inputs [5]

In Figure 3.2 the three square boxes indicate the whole input domains. The white space in each box shows legitimate and faultless values while the black colour in the form of points, block and strip, inside the respective boxes indicates the faults in the form of point, block and strip patterns.

Chen et al. [4] 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 fault-targeted test inputs, they proposed Adaptive Random Testing (ART) as a modified version of 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 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. [4], the number of test cases required to detect first fault (F-measure) was used as a performance matrix instead of the traditional matrices (P-measure) and (E-measure). Experimental results using ART showed up to 50% increase in performance compared to random testing. The authors pointed out 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 versions [28, 81].

3.1.2 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) [1] is an improvement on ART by using mirror-partitioning technique to reduce the overhead by decreasing the extra computation involved in ART.

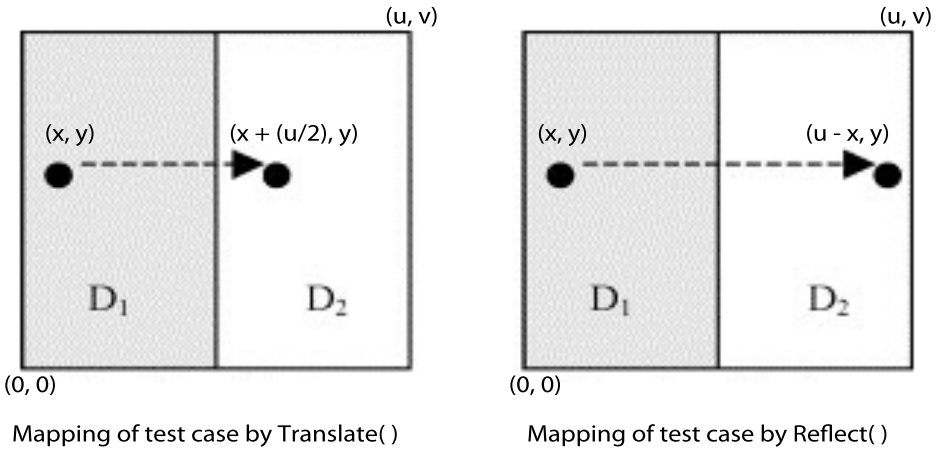


Figure 3.3: Mirror Adaptive Random Testing [1]

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 others are termed as mirror sub-domains. ART is then applied only to the source sub-domain while test cases are selected from all other sub-domains 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. 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 lower overhead by using only one quarter of the calculation as compared with ART.

3.1.3 Restricted Random Testing

Restricted Random Testing [82] 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 in each test case and the area of input domain decreases progressively with successive execution of test cases.

The above authors compared RRT with ART and RT to find the comparative performance and reported 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 random testing in terms of F-measure.

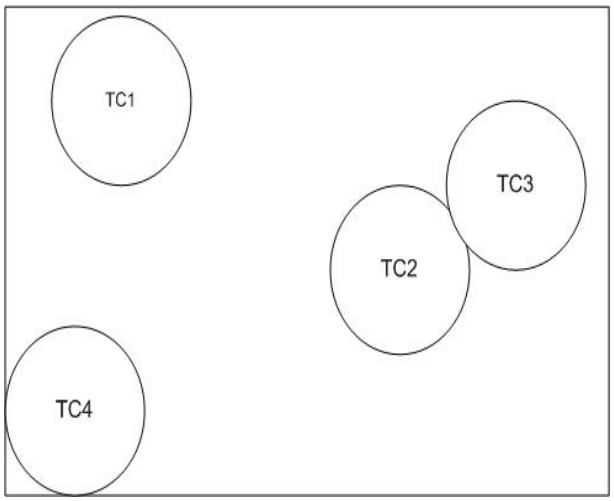


Figure 3.4: Input domain with exclusion zone around the selected test case

3.1.4 Directed Automated Random Testing

Godefroid et al. [22] proposed Directed Automated Random Testing (DART). The following main features of DART are reported in the literature:

1. **Automated Interface Extraction:** DART automatically identifies external interfaces of a given SUT. These interfaces include external variables and methods and the user-specified main method responsible for program execution.

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

3. **Dynamic Analysis of execution:** DART instrument 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) [28] is a testing technique which takes advantage of failure region contiguity for distributing test cases evenly and thus decreases 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 results 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) is a technique that generates unit test suite at random for object-oriented programs [27]. 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 directed unit test suite. In this way redundant and wrong 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 is 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. [83] 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 are assigned to the primitive and reference fields. Strings are treated in terms of directly usable values and Levenshtein formula [84] is used as a distance criterion 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 [83].

The ARTOO strategy, implemented in AutoTest tool [25], was evaluated in comparison with Directed Random (D-RAN) strategy by selecting classes from EiffelBase library [85]. 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 required to select test case in the ARTOO strategy was more than the D-RAN strategy and took more time and cost to generate a test case.

.

3.2 Tools for Automated Random Testing

A number of open-source and commercial automatic random testing tools 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 [2]. 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 kinds of inputs, therefore, JCrasher is designed to test only the public methods of the SUT with random inputs.

The working of JCrasher is illustrated by testing a $T.java$ program as shown in the Figure 3.5. The source file is first compiled using $javac$ and the byte code obtained is passed as input to JCrasher which uses Java reflection library [89] to analyse all the methods declared by class T. The JCrasher uses methods transitive parameter types P to generate the most appropriate test data set which is written to

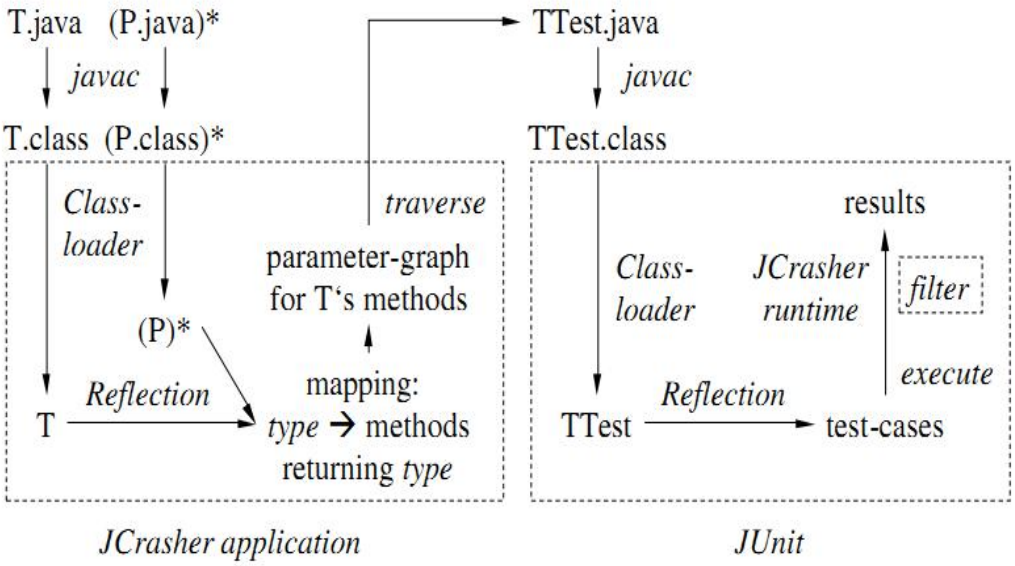


Figure 3.5: Illustration of robustness testing of Java program with JCrasher [2]

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 and reported as errors.

JCrasher is a pioneering tool with the capability to perform fully automatic testing, including test case generation, execution, filtration and 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) is an automated testing tool [90] 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 priori-

tise testing a 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 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-conditions 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 violations are considered as errors either in the methods or in the specifications. 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 [3] is an automated testing tool which generates and classifies unit tests for Java classes. The process is accomplished in three stages. In the first stage, it selects a small subset of test inputs that are likely to reveal faults in the given SUT.

The tool takes a software and a set of test cases for which the software runs properly. It 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) a fault in the given SUT (b) model violation despite normal operation (c) illegal input which the program is unable to handle. In the second stage, reducer function is used to discard any redundant input, leaving only a single input per operational pattern. In the third stage, the acquired test inputs are converted into test cases and oracles are created to determine the success or failure of the test.

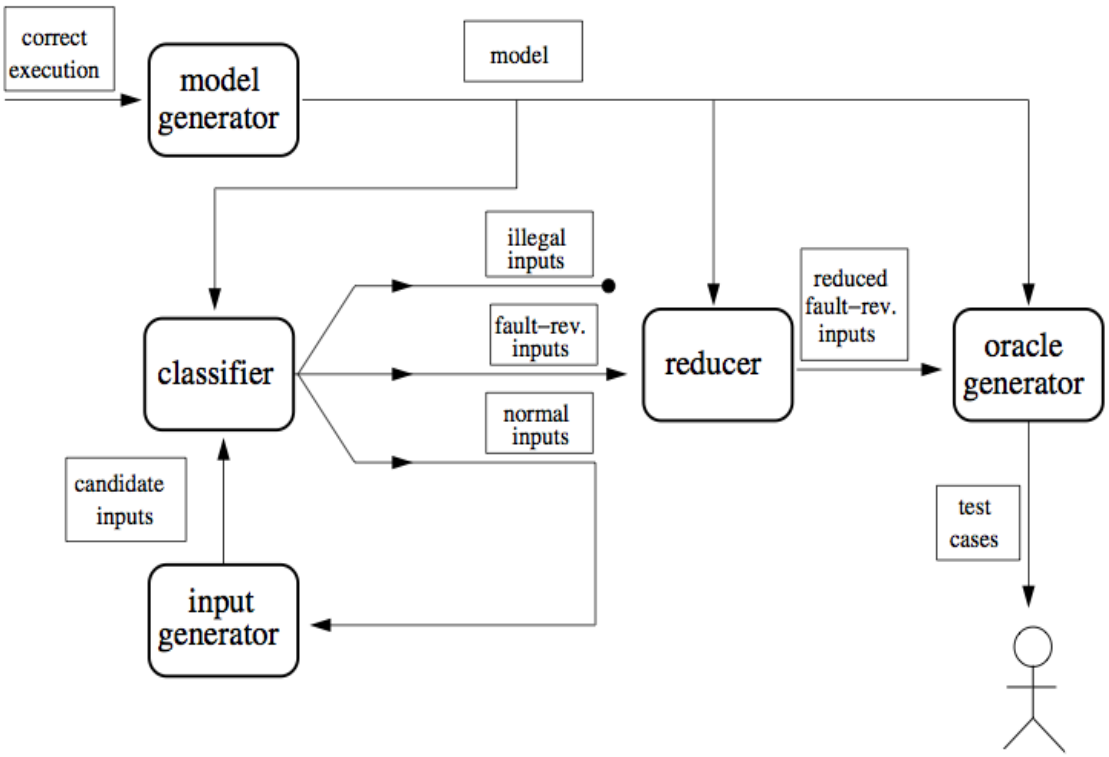


Figure 3.6: Main component of Eclat contributing to generate test input [3]

—— compared Eclat with JCrasher by executing nine programs on both tools. —— reported that Eclat performed 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% revealed faults. The limitation of Eclat is dependence on initial pool of correct test cases. Existence of errors in the pool leads to the creation of wrong operational model which adversely affects the testing process [2].

3.2.4 Randoop Tool

Random tester for Object Oriented Programs (RANDOOP) is the tool used for implementing FDRT technique [2]. 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 and gives output either as a suite of JUnit or 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 builds 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 an interesting state. On test completion, ErrorSeqs and NonErrorSeqs are produced as JUnit/NUnit test suite. In terms of coverage and number of faults discovered, RANDOOP implementing FDRT was compared with JCrasher and JavaPathFinder and 14 libraries of both Java and .Net were evaluated [91]. The results showed that RANDOOP achieved more coverage than JCrasher in branch coverage and faults detection. It can achieve on par coverage with systematic approaches like JavaPathFinder.

3.2.5 QuickCheck Tool

QuickCheck [92] is a lightweight random testing tool used for testing of Haskell programs [93]. 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. 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 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 Autotest Tool

The Autotest tool, based on formal automated testing is used to test Eiffel language programs [21]. 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 allows the tester to manually write the test cases to target specific section of the code. The Autotest may have a single method/class or suite of methods/classes as inputs, it then automatically generate test input data according to the requirement of the methods or classes.

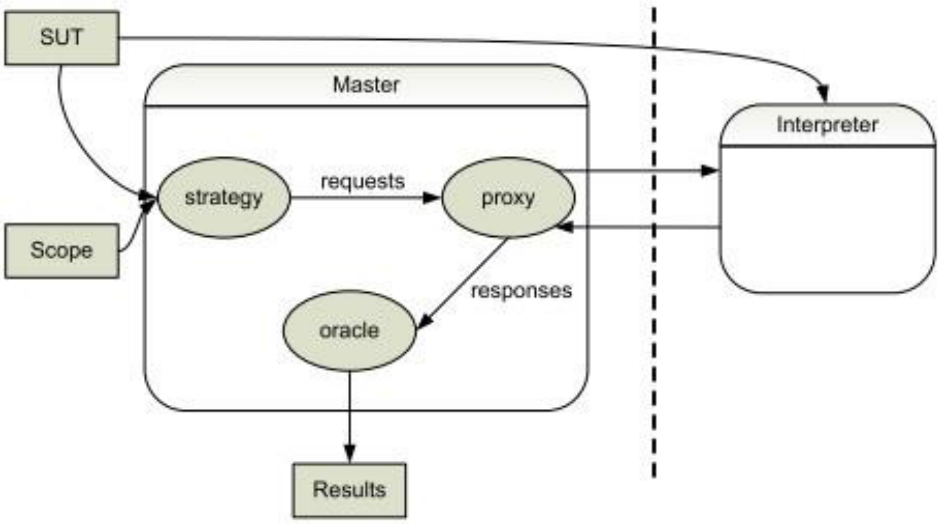


Figure 3.7: Architecture of Autotest [60]

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

1. **Testing Strategy:** It is a pluggable component where testers can fit any strategy according to the testing requirement. The strategy contains the directions for testing.The default strategy creates test cases that uses random input to exercise the methods/classes under test.

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

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

4. **Oracle:** It is based on contract-based testing. It evaluate the results to see if the contracts are satisfied. The outcome of the tests are formatted in HTML and stored on disk.

3.2.7 TestEra Tool

TestEra [94] 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 inputs 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 [95] to express constraints on test inputs and Alloy Analyser tool [96] 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 the outcome (c) it translates each outcome from propositional domain into the relational domain.

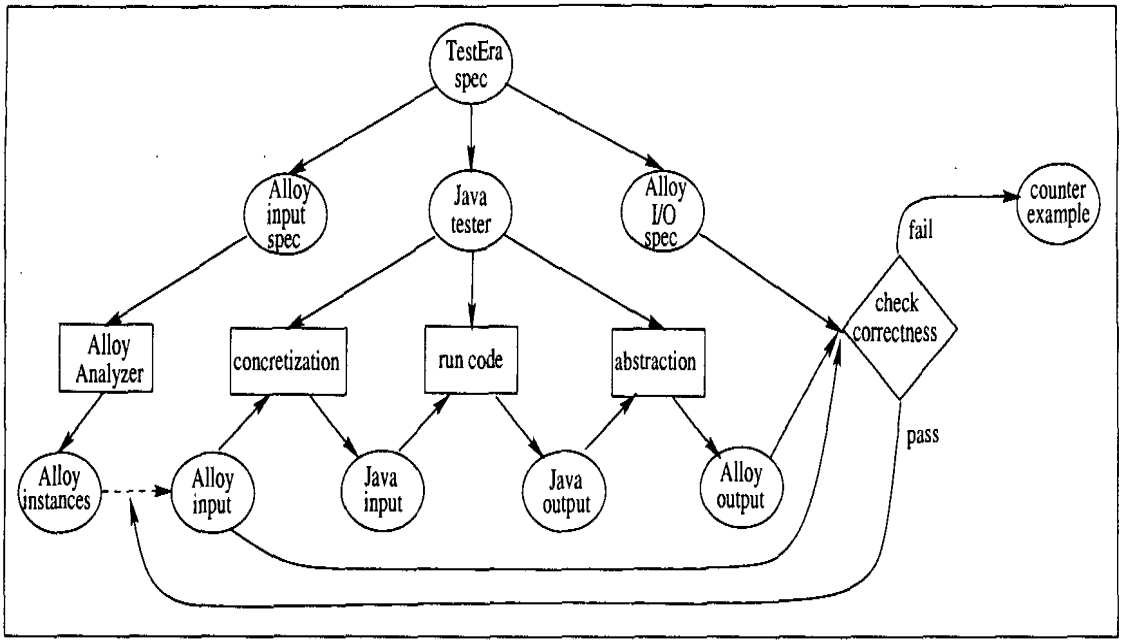


Figure 3.8: Architecture of TestEra [94]

—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 random 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 [97] is a novel framework for automated testing of Java programs based on the formal specifications [98]. Korat and TestEra [94] were developed by the same team and perform specification based testing. The difference however is that Korat uses Java Modelling Language (JML) while TestEra uses Alloy Modelling Language for specifications. Moreover, Korat uses bounded-exhaustive testing in which the code is tested against all possible inputs within the given small bound [99].

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 it to a boolean value. Korat takes imperative predicates and finitization value as inputs. It systematically explores the predicates input space and generates all non-isomorphic inputs for which the predicates return true. The core part of Korat monitors execution of the predicates on candidate inputs to filter out these fields accessed the particular fields 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 [72].

3.3 YETI Overview

York Extensible Testing Infrastructure (YETI), an automated random testing tool developed in Java, 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 syntactically-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), which make YETI user friendly and provides 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`. Figure 3.9 briefly presents the working process of YETI.

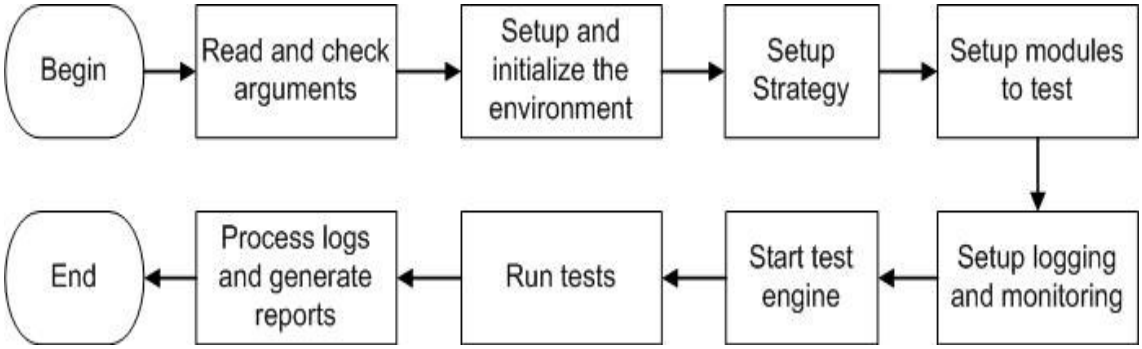


Figure 3.9: Working process of YETI

3.3.1 YETI Design

YETI has been designed with the provision of extensibility for future growth. YETI enforces strong decoupling between test strategies and the actual language constructs, which adds new binding, without any modification in the available test strategies. YETI can be divided into three main parts on the basis of functionality: the core infrastructure, the strategy and the language-specific binding. Each part is briefly described below.

3.3.1.1 Core Infrastructure

The core infrastructure is responsible for test data generation, test process management and test report generation. The core infrastructure is split into four packages: yeti, yeti.environments, yeti.monitoring, yeti.strategies. The package yeti uses classes from yeti.monitoring and yeti.strategies packages and calls classes in the yeti.environment package as shown in the Figure 3.10.

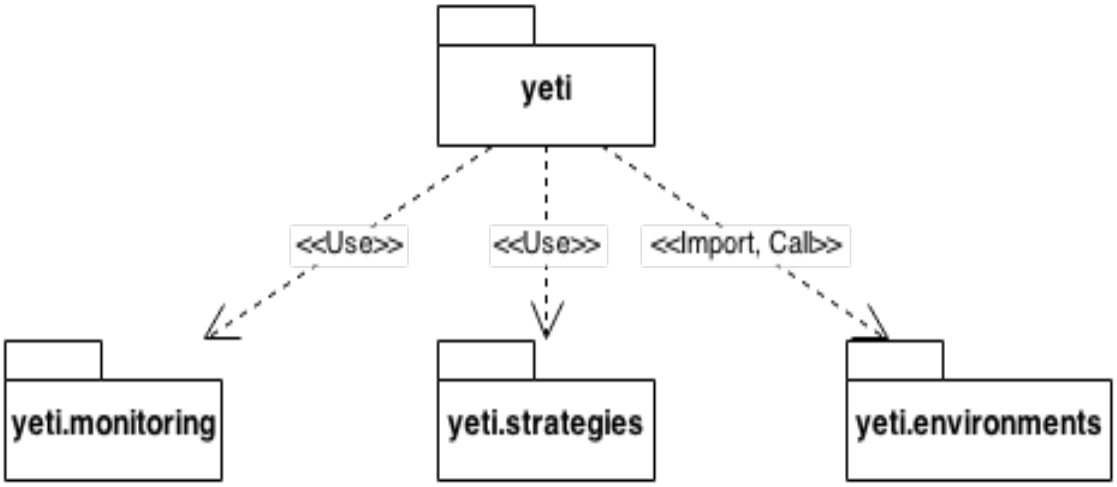


Figure 3.10: Main packages of YETI with dependencies

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

1. **Yeti class:** It is the entry point to YETI and contains the main method. It parses the arguments, sets up the environment, initializes the testing and delivers the reports of the test results.
2. **YetiLog class:** It prints debugging and testing logs.
3. **YetiLogProcessor class:** It is an interface for processing testing logs.
4. **YetiEngine class:** It binds YetiStrategy and YetiTestManager together, which carry out the actual testing process.
5. **YetiTestManager class:** It makes the actual calls based on the YetiEngine configuration, activate the YetiStrategy to generate test data and select the routines.
6. **YetiProgrammingLanguageProperties class:** It is a place holder for all language related instances.

7. **YetiInitializer class:** It is an abstract class for test initialization.

3.3.1.2 Strategy

The strategy defines a specific way to generate test inputs. This part contains six essential strategies stated below.

1. **YetiStrategy class:** It is an abstract class which provides interface for every strategy in YETI.
2. **YetiRandomStrategy class:** It implements the random strategy and generates random values for testing. The strategy gives choice to the user to adjust null values probability and the percentage of creating new objects for the test session.
3. **YetiRandomPlusStrategy class:** It extends the random strategy by adding interesting values to the list of test values. The strategy gives the choice to the user to select the percentage of interesting values used in the test session.
4. **DSSRStrategy class:** It extends random+ strategy by adding the values surrounding the fault value. The strategy is described in detail in Chapter 4.
5. **ADFDStrategy class:** It extends random+ strategy by adding the feature of graphical representation of faults and their domains. The strategy is described in detail in Chapter 5.
6. **YetiRandomDecreasingStrategy class:** It extends the random+ strategy by setting the probability value to starts at 100% and ends at 0% when the test finishes.
7. **YetiRandomPeriodicStrategy class:** It extends random+ strategy by setting the probability in such a way that it decreases and increases randomly during test session.

3.3.1.3 Language-specific Binding

The language-specific binding provides support for modelling a programming language. They are language independent and need to be extended for supporting a

new language.

1. **YetiVariable class:** It is a sub class of YetiCard representing a variable in YETI.
2. **YetiType class:** It denotes type of data including Integer, String, Boolean etc.
3. **YetiRoutine class:** It is a super type of routines which represents functions, methods and constructors. A routine is given a name, a return type and a list of its arguments types.
4. **YetiModule class:** It represents a module under test. It stores a list of the modules routines to test.
5. **YetiName class:** It represents a unique name which is assigned to each instance of YetiRoutine.
6. **YetiCard class:** It is a YETI specific term which map to a wildcard or to a YetiVariable. It has a specific type and an identifier name.
7. **YetiIdentifier class:** It represents an identifier for an instance of a YetiCard.

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.

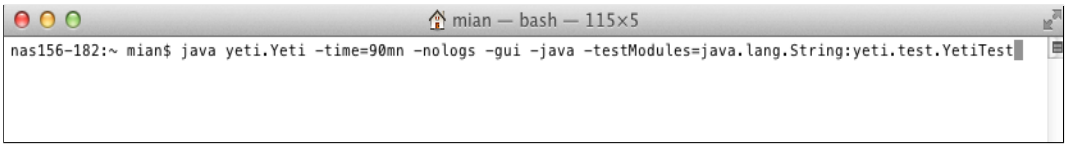


Figure 3.11: Command to launch YETI from CLI

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.

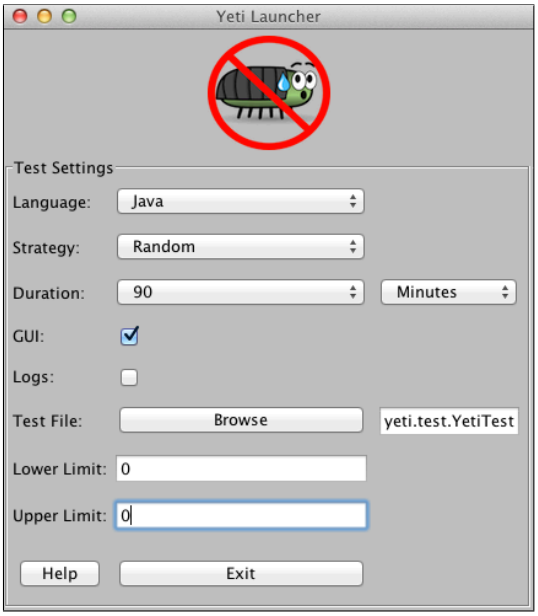


Figure 3.12: GUI launcher of YETI

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).

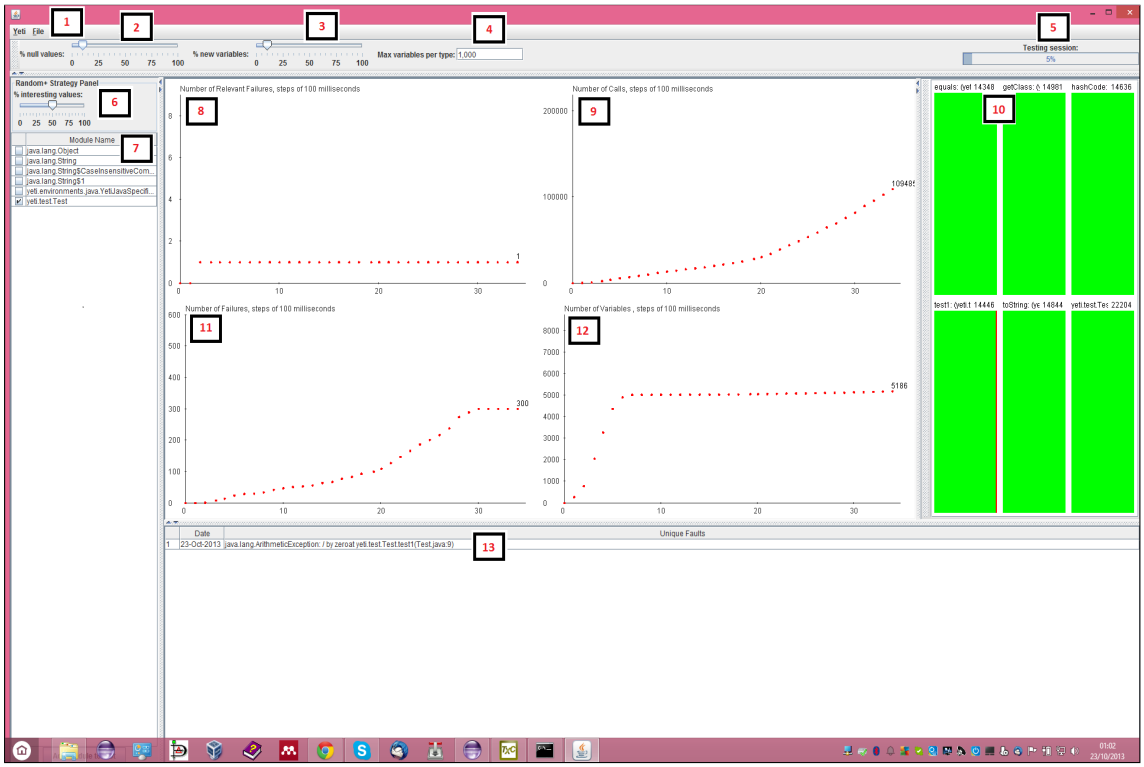


Figure 3.15: GUI of YETI Tool

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. [5] 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. [4] 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 [26], Feedback directed Random Test Generation [27],

Mirror Adaptive Random Testing [1] and Quasi Random Testing [28] 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 [102] 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. It is based on two intuitions. First, boundaries have interesting values and using

[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

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 [103]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [104, 58]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [25]. YETI [105, 101], AutoTest [21, 60], QuickCheck [92], Randoop [27], JArtege [90] 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 [33] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [21, 75, 106, 71, 107] 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 [21, 108] 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 [60] 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 [109] 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 [58].

4.2.3 Dirt Spot Sweeping (DSS)

Chan et al. [5] 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 [26, 1, 28], 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.

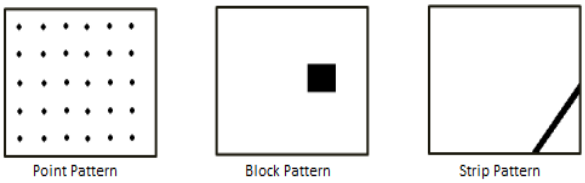


Figure 4.1: Failure patterns across input domain [4]

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.

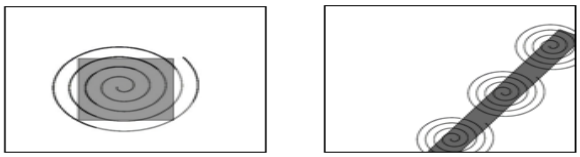


Figure 4.2: DSSR covering block and strip pattern

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 [110]. 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.

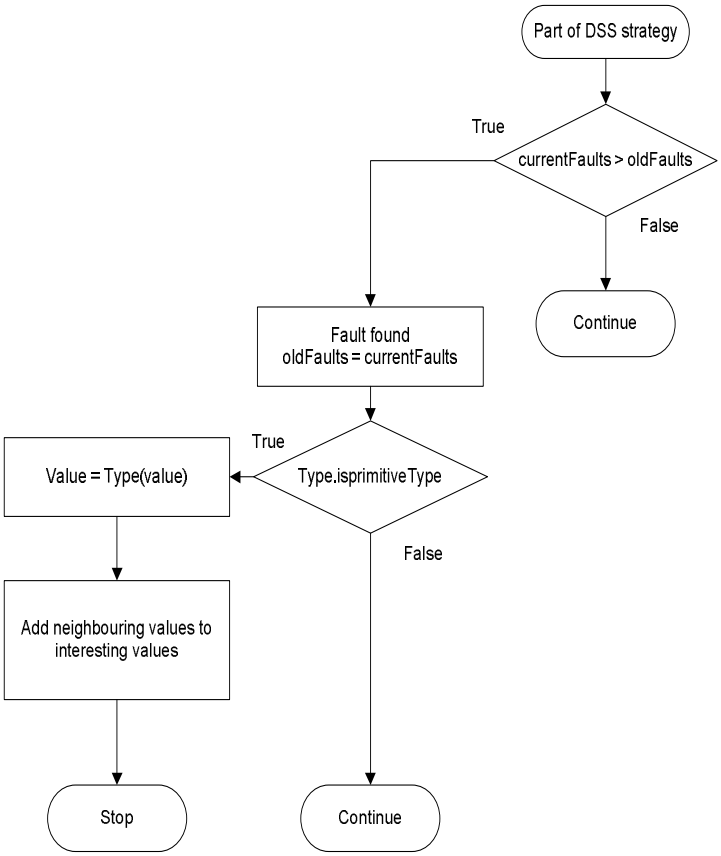


Figure 4.3: Working mechanism of DSSR Strategy

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 [111, 108].

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.

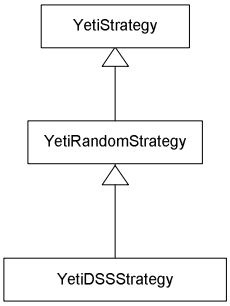


Figure 4.4: Class Hierarchy of DSSR in YETI

4.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [60]. 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 [112]. 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 [108], 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 [4] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [113, 114]. 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 [115].

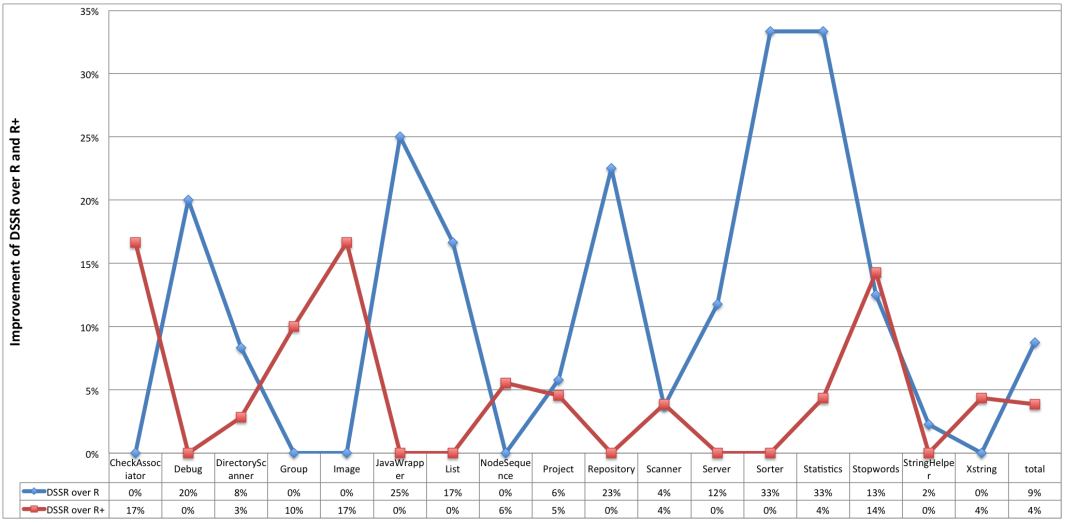
The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print

[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 [21, 116, 27]. 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 [108].

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 [117, 92, 3]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [71]. Some of the well known automated tools based on random strategy includes Jartege [90], Eclat [3], JCrasher [117], AutoTest [21, 25] and YETI [108, 101].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [26, 1, 103, 28, 118]. 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 [119] 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 [60]. These interesting values includes border values which have high tendency of finding faults in the given SUT [109]. Results obtained with R+ strategy show significant improvement over random strategy [60]. 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 pro-

grams [106, 120, 43]. Arcuri et al. [121], 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 [122] 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 [108, 123, 112].

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 [42]. 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 [58, 124]. It involves minimum or no overhead in input selection and lacks human bias [71, 125]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [30] and discovery of lower number of faults [126]. 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 [4]. Similarly Restricted Random Testing (RRT) [26], Mirror Adaptive Random Testing (MART) [113], Adaptive Random Testing for Object Oriented Programs (ARTOO) [58], Directed Adaptive Random Testing (DART) [22], Lattice-based Adaptive Random Testing (LART) [127] and Feedback-directed Random Testing (FRT) [27] 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. [5] 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.

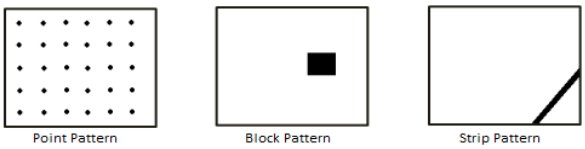


Figure 5.1: Failure domains across input domain [5]

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

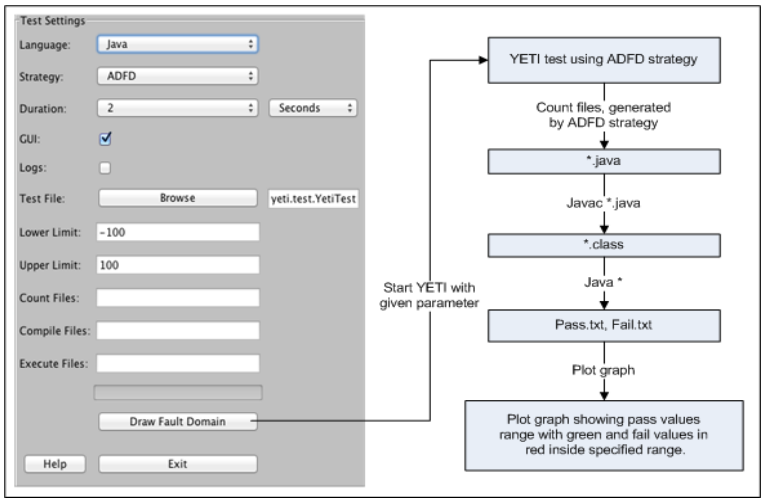


Figure 5.2: Work flow of ADFD strategy

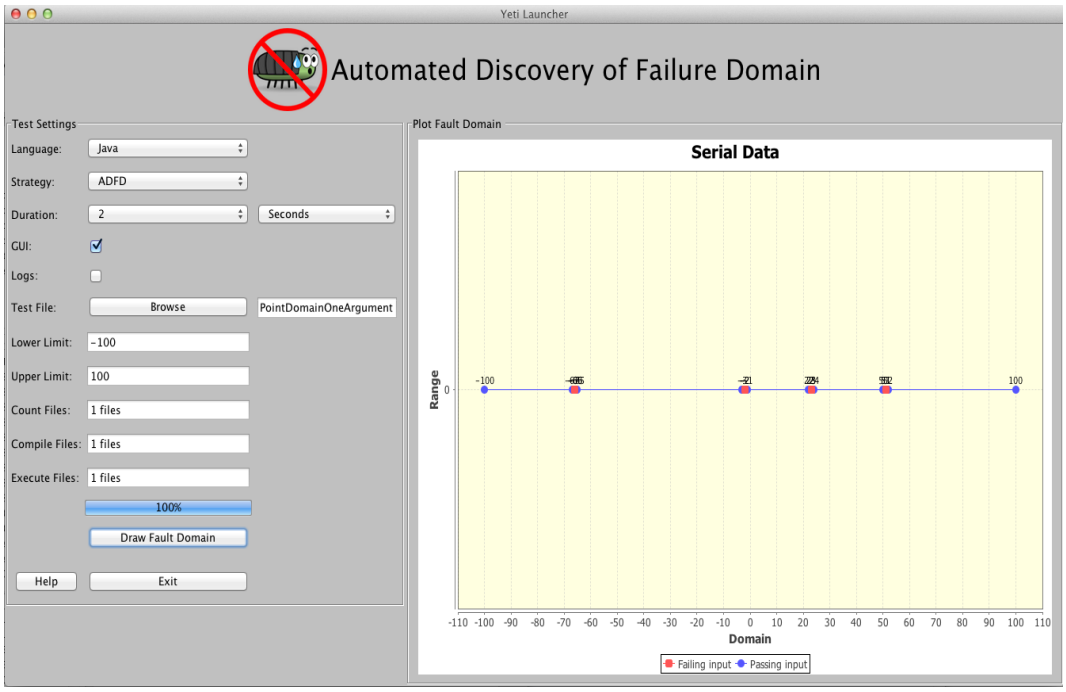


Figure 5.3: Front-end of ADFD strategy

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 [111, 108]. 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 [128]. 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 [111, 108, 101, 105] 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();
    }
}
```

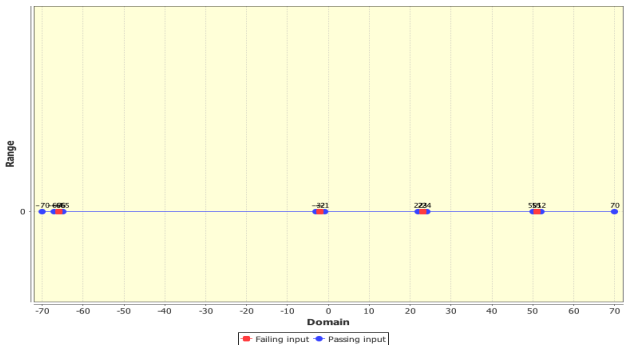


Figure 5.4: ADFD strategy plotting pass and fault domain of the given class

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].

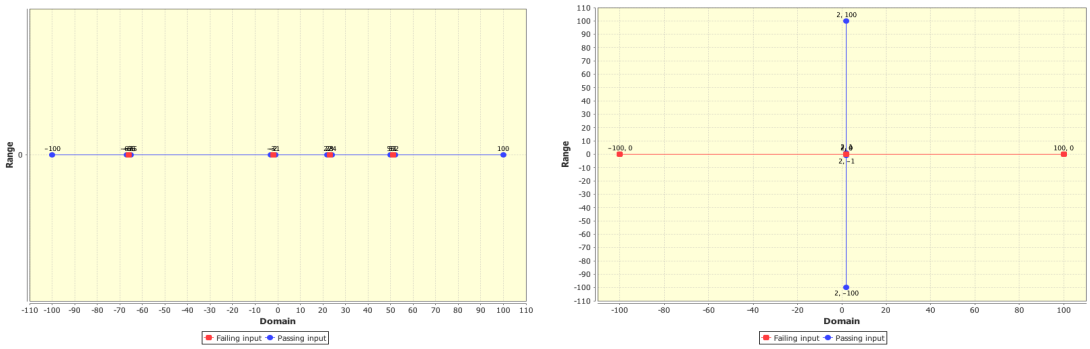
To elucidate the results, six programs were developed so as to have separate program for one and two-dimension point, block and strip fault domains. The code of selected programs is given in Appendix A.1 (2-7). The experimental results are presented in table 5.1 and described under the following three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

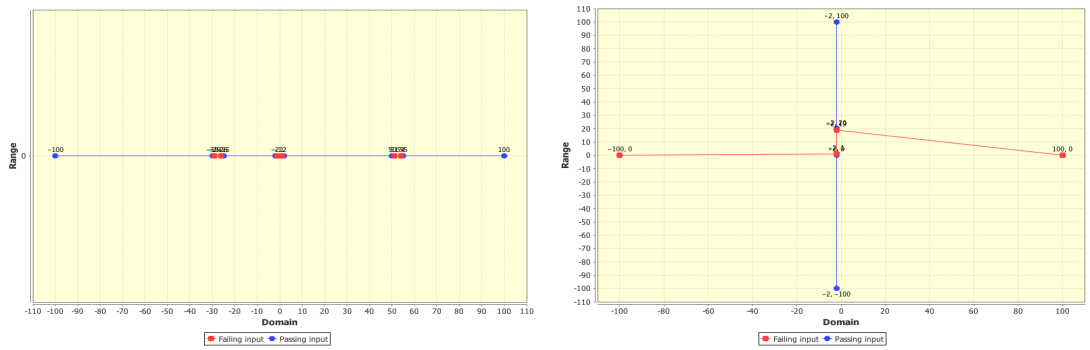
program. The range of pass and fail values for each program in point fault domain are given in (Table 5.1, Serial No. 1).



(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).

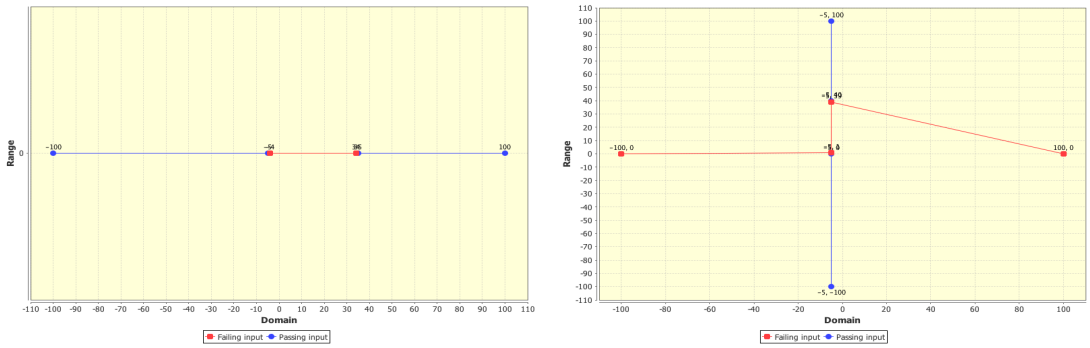


(a) One dimension module (b) Two dimension module

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[83],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 [24, 30]. To overcome the performance issues without compromising on its benefits, various researchers have altered its algorithm as explained in section 5.1. Most of the alterations are based on the existence of faults and fault domains across the input domain [5].

Identification, classification of pass and fail domains and visualisation of domains have not received due attention of the researchers. Podgurski et. al., [129] 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 [130] 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 [131]. 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 5.4), based on applying ADFD strategy to error-seeded numerical programs provide, evidence that the strategy is highly ef-

fective 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 [60] 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 [109] 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 [58].

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

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 [42]. 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 [58], [124]. It involves minimum or no overhead in input selection and lacks human bias [71], [125]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [30] and discovery of lower number of faults [126]. 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 [4]. Similarly Restricted Random Testing (RRT) [26], Mirror Adaptive Random Testing (MART) [113], Adaptive Random Testing for Object Oriented Programs (ARTOO) [58], Directed Adaptive Random Testing (DART) [22], Lattice-based Adaptive Random Testing (LART) [127] and Feedback-directed Random Testing (FRT) [27] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

Chapter 8

Future Work

8.1 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 [42]. 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 [58], [124]. It involves minimum or no overhead in input selection and lacks human bias [71], [125]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [30] and discovery of lower number of faults [126]. 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 [4]. Similarly Restricted Random Testing (RRT) [26], Mirror Adaptive Random Testing (MART) [113], Adaptive Random Testing for Object Oriented Programs (ARTOO) [58], Directed Adaptive Random Testing (DART) [22], Lattice-based Adaptive Random Testing (LART) [127] and Feedback-directed Random Testing (FRT) [27] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

Appendix A

A.1 Sample code to identify failure domains

Program 1 Program generated by ADFD on finding fault in SUT

```
/**
 * Dynamically generated code by ADFD strategy
 * after a fault is found in the SUT.
 * @author (Mian and Manuel)
 */
import java.io.*;
import java.util.*;

public class C0
{
    public static ArrayList<Integer> pass = new ArrayList<Integer>();
    public static ArrayList<Integer> fail = new ArrayList<Integer>();
    public static boolean startedByFailing = false;
    public static boolean isCurrentlyFailing = false;
    public static int start = -80;
    public static int stop = 80;

    public static void main(String []argv){
        checkStartAndStopValue(start);
        for (int i=start+1;i<stop;i++){
            try{
                PointDomainOneArgument.pointErrors(i);
                if (isCurrentlyFailing)
                {
                    fail.add(i-1);
                    fail.add(0);
                    pass.add(i);
                    pass.add(0);
                    isCurrentlyFailing=false;
                }
            }
            catch(Throwable t) {
                if (!isCurrentlyFailing)
                {
```

```
                pass.add(i-1);
                pass.add(0);
                fail.add(i);
                fail.add(0);
                isCurrentlyFailing = true;
            }
        }
    }
    checkStartAndStopValue(stop);
    printRangeFail();
    printRangePass();
}

public static void printRangeFail() {
    try {
        File fw = new File("Fail.txt");
        if (fw.exists() == false) {
            fw.createNewFile();
        }
        PrintWriter pw = new PrintWriter(new FileWriter (fw, true));
        for (Integer i1 : fail) {
            pw.append(i1+"\n");
        }
        pw.close();
    }
    catch(Exception e) {
        System.err.println(" Error : e.getMessage() ");
    }
}
public static void printRangePass() {
    try {
        File fw1 = new File("Pass.txt");
        if (fw1.exists() == false) {
            fw1.createNewFile();
        }
        PrintWriter pw1 = new PrintWriter(new FileWriter (fw1, true));
        for (Integer i2 : pass) {
            pw1.append(i2+"\n");
        }
        pw1.close();
    }
    catch(Exception e) {
        System.err.println(" Error : e.getMessage() ");
    }
}
public static void checkStartAndStopValue(int i) {
    try {
        PointDomainOneArgument.pointErrors(i);
        pass.add(i);
        pass.add(0);
    }
    catch (Throwable t) {
        startedByFailing = true;
        isCurrentlyFailing = true;
        fail.add(i);
```

```
            fail.add(0);
        }
    }
}
```

Program 2 Point domain with One argument

```
/**
 * Point Fault Domain example for one argument
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{

    public static void pointErrors (int x){
        if (x == -66 )
            x = 5/0;

        if (x == -2 )
            x = 5/0;

        if (x == 51 )
            x = 5/0;

        if (x == 23 )
            x = 5/0;
    }
}
```

Program 3 Point domain with two argument

```
/**
 * Point Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{

    public static void pointErrors (int x, int y){
        int z = x/y;
    }

}
```

Program 4 Block domain with one argument

```
/**
 * Block Fault Domain example for one arguments
 * @author (Mian and Manuel)
 */

public class BlockDomainOneArgument{

public static void blockErrors (int x){

    if((x > -2) \&\& (x < 2))
        x = 5/0;
```

```
    if((x > -30) \&\& (x < -25))
        x = 5/0;

    if((x > 50) \&\& (x < 55))
        x = 5/0;

   }
}
```

Program 5 Block domain with two argument

```
/**
 * Block Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class BlockDomainTwoArgument{

    public static void pointErrors (int x, int y){

        if(((x > 0)\&\&(x < 20)) || ((y > 0) \&\& (y < 20))){
        x = 5/0;
        }

    }

}
```

Program 6 Strip domain with One argument

```
/**
 * Strip Fault Domain example for one argument
 * @author (Mian and Manuel)
 */

public class StripDomainOneArgument{

    public static void stripErrors (int x){

        if((x > -5) \&\& (x < 35))
            x = 5/0;
     }
}
```

Program 7 Strip domain with two argument

```
/**
 * Strip Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class StripDomainTwoArgument{

    public static void pointErrors (int x, int y){

        if(((x > 0)\&\&(x < 40)) || ((y > 0) \&\& (y < 40))){
        x = 5/0;
        }
```

```
        }

    }
```

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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

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

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

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

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

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

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

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

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

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

[32] Dave Towey. Software quality assurance.

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

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

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

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

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

[38] 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.

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

[40] 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.

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

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

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

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

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

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

[47] 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.

[48] 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.

[49] 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.

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

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

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

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

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

[55] 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.

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

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

[58] 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.

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

[60] 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.

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

[62] 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.

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

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

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

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

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

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

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

[83] 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.

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

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

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

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

[88] 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.

[89] 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.

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

[91] 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.

[92] 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.

[93] 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.

[94] 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.

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

[96] 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.

[97] 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.

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

[99] 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.

[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] 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.

[103] 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.

[104] 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.

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

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

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

[108] 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.

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

[110] 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.

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

[112] 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.

[113] 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.

[114] 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.

[115] 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.

[116] 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.

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

[118] 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.

[119] 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.

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

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

[122] 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.

[123] 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.

[124] 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.

[125] 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.

[126] 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.

[127] 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.

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

[129] 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.

[130] 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.

[131] 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.