New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group

Department of Computer Science

University of York, UK

September 2014

A thesis submitted for the degree of Doctor of Philosophy

Abstract

Exhaustive testing is not feasible in most cases and a test strategy is often used to select a small subset of the inputs for testing the software. Selection of adequate test strategy is crucial for better test performance because the chances of finding failures increases if the selected subset of data effectively represents the whole input domain.

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

The first technique, Dirt Spot Sweeping Random (DSSR) strategy is developed on the assumption that unique failures reside in contiguous block and strips. It starts by testing the code at random. When a failure is identified, the DSSR strategy selects the neighbouring input 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 new failures in SUT.

The second technique, Automated Discovery of Failure Domain (ADFD) is developed with the capability to find failure and the failure-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 technique, Automated Discovery of Failure Domain+ (ADFD+) is an upgraded version of ADFD technique with respect to algorithm and graphical representation of failure domains. To find the effectiveness of ADFD+, it was compared with Randoop using error seeded programs. The ADFD+ correctly pointed out all the seeded failure domains while Randoop identified individual failures but was unable to discover the failure domains.

Contents

List of Figures

List of Tables

Dedication

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

Acknowledgements

The duration at the University of York for my PhD has been the most joyful and rewarding experience 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 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 highly indebted to Prof. Dr. Richard Paige for his generous help, cooperation and guidance throughout my research.

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 to my 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 thankful to my dear wife Dr. Munazza Asbat for her company, help and cooperation throughout my stay at York.

I obtained Departmental Overseas Research Scholarship which 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.

Glossary

Defect:	It is a generic term which refers to either a fault or a failure, where fault pertains to the cause and failure to the effect.
Fault:	It is a flaw in the system which may result in lack of capability or failure of the system.
Failure:	It is a malfunction of a system within the specified requirements or inappropriate performance.
Error:	It is a mistake or omission on the part of humans resulting in faulty software.
Isolation:	It is to identify the basic cause of failure in the SUT.
Detection:	It checks the difference between the observed behaviour of a program and the expected behaviour.
Test case:	It is an artefact which delineates the input, action and expected output corresponding to that input.
Test specification:	It contains requirement which should be satisfied by test cases.
Test execution:	It is the process of running test cases.
Test Plan:	It is a document which defines the goal, scope, method, resources and time schedule of testing.
Test suite:	It is a set of one or more test cases.
Test strategy:	It is a method which defines the procedure of testing distinctive characteristics of a program.
Validation:	It is a process to assess the software in order to assure that it satisfies the customer requirements.
Verification:	It is a process of checking the software to verify that it works correctly.
Correctness:	It is the ability of a software to perform as expected by its specification.
Test oracle:	It is a source containing expected results for comparison with the actual results of the SUT.
Robustness:	It is the ability of a software to handle situations not defined by its specification.
Efficiency:	It is the number of defects discovered by a technique per unit time.

Effectiveness: It is the number of defects discovered in a SUT by a testing technique.

Declaration

I declare that the contents of this thesis derive from my own original research between January 2010 and September 2014, the period during which I was registered for the degree of Doctor of Philosophy at University of York.

Contributions from this thesis have been published in the following papers:

- Mian A Ahmad and Manuel Oriol. Dirt Spot Sweeping Random strategy. *Lecture Notes on Software Engineering (LNSE)*, 2(4) 294-299, 2014.

  Based on research described in Chapter 4 in the thesis.

- Mian A Ahmad and Manuel Oriol. Automated Discovery of Failure Domain. *Lecture Notes on Software Engineering (LNSE)*, 1(3) 289-294, 2013.

  Based on research described in Chapter 5 in the thesis.

- Mian A Ahmad and Manuel Oriol. ADFD+: An Automatic Testing Technique for Finding and Presenting Failure domains. *Lecture Notes on Software Engineering (LNSE)*, 2(4) 331-336, 2014.

  Based on research described in Chapter 6 in the thesis.

Chapter 1

Introduction

Software consists of a set of clearly defined instructions for computer hardware to perform a particular task upon execution. 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 software expect us to believe that software are reliable, robust, safe and secure. Like every other man-made items software are also prone to errors. Maurice Wilkes [6], 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 large economic losses [7]. According to the National Institute of Standards and Technology, US companies alone bear $59.5 billion loss every year due to software failures [8]. Software testing is the most recognized and widely used technique to verify correctness and ensure quality of the software [9]. Therefore, software companies leave no stone unturned to ensure the reliability and accuracy of the software before its practical application. According to Myers et al. some software companies spend up to 50% of the elapsed time and more than 50% of the total development and maintenance cost on software testing [10]. There is therefore a strong motivation for improving the existing or developing new efficient test strategies, and this desire for cost-effectiveness is the context of this thesis.

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, the usual practice is to use a test strategy for the selection of test data

set from a large/infinite domain. Careful selection of the test data set, as a subset of the whole input domain, is a crucial factor in any testing technique because it represents the whole domain for evaluating the structural and/or functional properties [11, 12]. 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 set so that all paths of the SUT are executed at least once [13]. The approach resulted in higher standard of test quality.

Test data set can be generated manually and automatically. However, generating test data set manually is a time-consuming and laborious exercise [14], therefore, automated test data set generation is mostly preferred. Test data generators can be of different types i.e. Path-wise (Section 2.7.1), Goal-oriented (Section 2.7.2), Intelligent (Section 2.7.3), search-based (Section 2.7.4) and Random (Section 2.7.5).

From the importance of relevant test data set in the cost effectiveness of any testing technique, it would appear that the testers should always use the most efficient test strategy during software testing, however, the computation required to generate test data set that satisfy complex constraints also require due consideration because the cost of implementing that very strategy may be greater than the overall savings achieved. We show that random test data set generation approach is simple, widely applicable, easy to implement, faster in computation, free from bias, costs minimum overhead [15] and its effectiveness can be further increased by slightly altering its algorithm.

1.1.1 Software Testing

Software testing is a Verification and Validation (V&V) technique used to ensure that the software adheres to the desired specifications. According to Edsger Dijkstra, program testing can be used to show the presence of bugs but never to show the absence of bugs [16]. 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.

1.1.2 Random Testing

It 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. According to Godefroid et al. [17], “Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs”. The three main phases of random testing including test data generation,

execution and evaluation are shown in Figure 1.1.

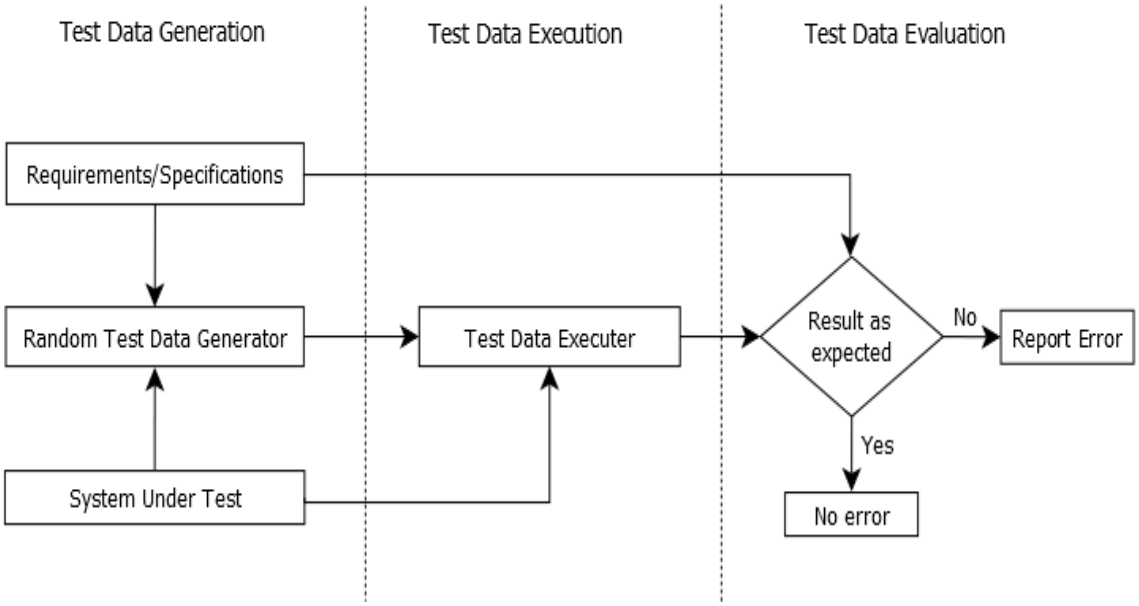


Figure 1.1: Three main phases of random testing

This thesis is a contribution to the literature on the subject, with the aim to bring about improvement in software testing by devising new, improved and effective automated software testing techniques based on random strategy.

1.2 The Problems

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [10, 18]. This research study focuses on the following problems in automated Random Testing (RT):

1. Limitation of RT to discover contiguous failures.
2. Inability of RT to identify failure-domains.
3. Incompetence of RT to present results in graphical form.

1.2.1 Limitation of RT to discover contiguous failures

Chan et al. [1] observed that failure inducing inputs are contiguous and form certain geometrical patterns in the whole input domain. They divided them into point, block and strip patterns on the basis of their shape (Section 2.10.2). The failure-finding ability of random testing technique decreases when the failures lie in contiguous block and strip patterns across the input domain. Attempts are needed to overcome this limitation of RT by developing suitable random strategy.

1.2.2 Inability of RT to Identify Failure-domains

Although failures reside in contiguous locations and form certain failure domains across the input domain [1], the existing random strategies of software testing tries to discover failures individually from the domains and lack the capability to discover the failure-domains. Endeavours are required to be undertaken for developing appropriate random strategy with the potential to identify the failures as well as failure-domains.

1.2.3 Incompetence of RT to Present Results in Graphical Form

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. No random strategy seems to provide graphical representation of the failures and failure-domains. Efforts are therefore required to get the test results of random testing in user-friendly graphical form.

1.3 Research Goals

Goals of the research study are: to understand the nature of failures, how to leverage failure domain for finding more bugs, to develop new improved automated random strategies.

1.4 Contributions

The main contributions of the thesis research are as follows:

1.4.1 Dirt Spot Sweeping Random Strategy

The failure-finding ability of the random testing technique decreases when the failures lie in contiguous locations across the input domain. Dirt Spot Sweeping Random (DSSR) strategy was developed as a new automated technique to overcome the problem. It is based on the assumption that unique failures reside in contiguous blocks and strips. 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 indicate higher failure-finding ability of DSSR strategy as compared with Random and Random+ strategies.

1.4.2 Automated Discovery of Failure Domain

The existing random strategies of software testing discover the failures in the SUT but lack the capability of presenting the failure domains. In the present study, a fully automated testing strategy named, “Automated Discovery of Failure Domain (ADFD)” is developed with the ability to find failures as well as failure domains in a given SUT and provides visualisation of the identified pass and fail domains in a graphical form. The strategy implemented in YETI is described and practically illustrated by executing several programs of one and two dimensions in Chapter 5. The experimental results provide evidence that the newly developed ADFD strategy performs identification of failures as well as failure-domains and provides the results in graphical form.

1.4.3 Automated Discovery of Failure Domain+

Automated Discovery of Failure Domain+ (ADFD+) is an upgraded version of ADFD technique with respect to algorithm and graphical representation of failure-domains. The new algorithm searches for the failure-domain around the failure in a given radius as against

ADFD which limits the search between lower and upper bounds. In addition, ADFD output is improved to provide labelled graphs which make ADFD+ output easily understandable and user friendly. To find the effectiveness of ADFD+, it was compared with Daikon by using error seeded programs. The ADFD+ correctly pointed out all the seeded failure domains while Daikon identified individual failures but was unable to discover the failure domains.

1.5 Structure of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 provides literature review on 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 with particular focus on literature relevant to random testing. Various versions of random testing and the most commonly used automated testing tools based on random algorithms are reviewed.

Chapter 3 presents the York Extensible Testing Infrastructure (YETI), used as a tool in our experiments. YETI has been thoroughly reviewed including an overview, design, core infrastructure, strategy, language-specific binding, construction of test cases, command line options, execution, test oracle, report generation and graphical user interface.

Chapter 4 describes the Dirt Spot Sweeping Random (DSSR) strategy. The proposed new testing technique is implemented in YETI. Experimental evidence is presented in support of the effectiveness of DSSR strategy in finding failures and failure domains as compared with random and random+ strategies.

Chapter 5 presents Automated Discovery of Failure Domain (ADFD) strategy. The proposed new testing technique, implemented in YETI, finds failures and failure 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 Automated Discovery of Failure Domain+ (ADFD+) strategy. It is an upgraded version of ADFD technique with respect to algorithm and graphical representation of failure domains. To find the effectiveness of ADFD+, it was compared with Daikon

using error seeded programs and the results are reported.

Chapter 7 provides conclusions of the study including contributions and the lessons learned.

Chapter 8 highlights the opportunities for future work, challenges that may be faced and possible approaches to overcome the challenges.

Appendix A includes ADFD logic implementation and Java programs with point, block and strip failure domains.

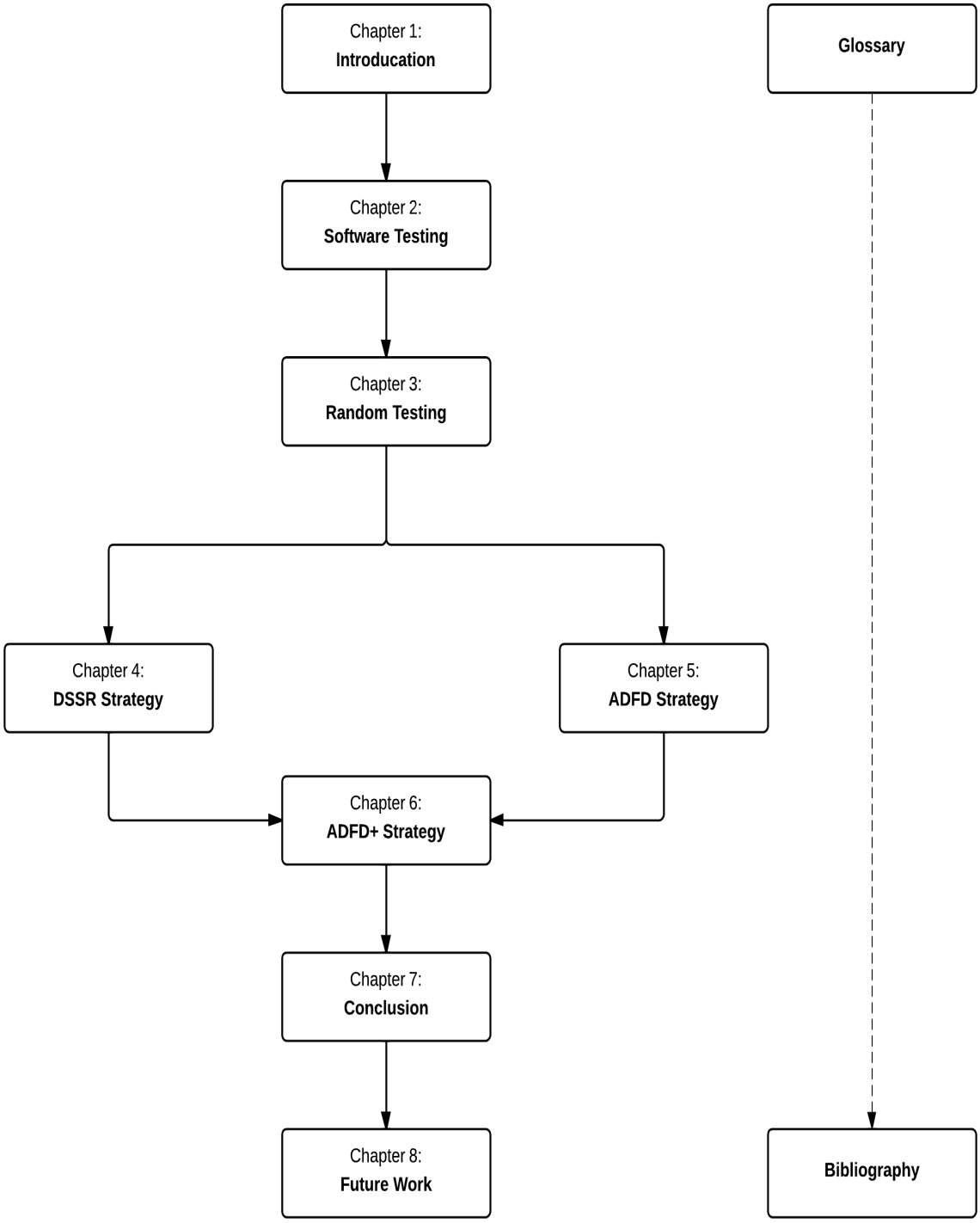


Figure 1.2: Structure of thesis outline

Chapter 2

Literature Review

The famous quote of Paul Ehrlich, “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. In spite of best efforts, some errors may remain in the software after it is finalised. Errors cannot be tolerated in software because a single error may lead to a large upset in the system. The destruction of Mariner 1 rocket (1962) costing $18.5 million, 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 (1996) costing $500 million and many more were caused by minor errors in the software [19]. According to the National Institute of Standards and Technology, US companies alone bear $59.5 billion loss every year due to software faults while one-third of it can be eliminated by improving the testing infrastructure [8]. A software has to satisfy rigorous stages of testing to achieve high quality. The more complex the software the higher the requirements for software testing because of the consequent larger damage involved if a fault remains in the software.

2.1 Software Testing

According to the ANSI/IEEE standard glossary of software engineering [20], testing is defined as, “the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results”. The testing process, being an integral part of Software Development Life Cycle (SDLC), starts from requirement phase and continues throughout the life of the software according to a predefined test plan. Test plan is a document which defines the goal, scope, method, resources and time schedule of testing [21]. 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.

In traditional testing, when a fault is found by the testers, the software is returned to the developers for rectification and the updated version is given back to the testers for retesting. It is important to note that a successful test is the one that fails a software or identifies a fault in the software [10] while fault denotes error made by programmers during software development [20]. The faulty code on execution can lead to software failures. A software that passes all tests without giving any error is not guaranteed to be error free. However, the testing process increases confidence of users and reliability of the tested product [16].

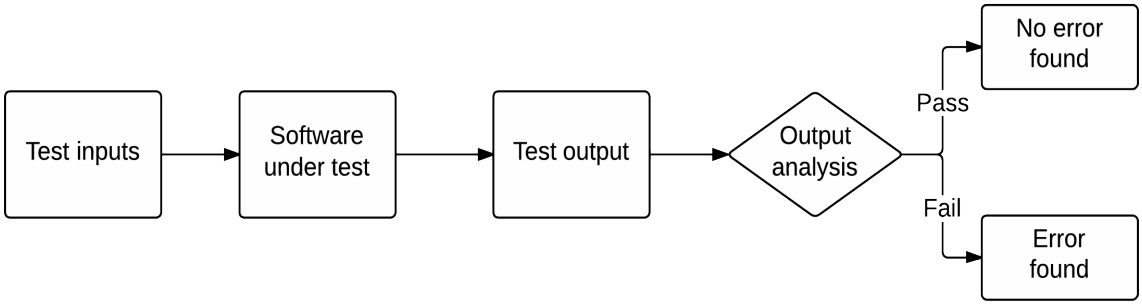


Figure 2.1: A simplified version of software testing process

The process of software testing in its simplest form is shown in Figure 2.1. In the process, test input data selected from the input domain is used to form test cases. The test cases are executed against the SUT and the output obtained is declared as pass or fail on the criteria defined in the test oracle. The input domain, test case and test oracle are briefly described below.

2.1.1 Input Domain

The input domain comprises all possible inputs for a software, including all global variables, method arguments and externally assigned variables. 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: $D = D1 \times D2 \times ... \times Dn$.

2.1.2 Test Case

Test case is an artifact which delineates the input, action and expected output corresponding to the input [22]. The test case is declared pass if the output obtained after executing the test case comply with the expected output meaning thereby that the functionality is working correctly. Alternatively the test case is declared fail if the output obtained after executing the test case does not comply with the expected output meaning thereby that the functionality is working incorrectly. A test suite comprising a series of test cases is required commonly executed to establish the desired level of quality.

2.1.3 Test Oracle

Test oracle is defined as, "a source containing expected results for comparison with the actual result of the SUT" [22]. For a program `P`, an oracle is a function which verifies that the output from `P` is the same as the output from a correct version of `P` [11]. Test oracle sets the acceptable behaviour for test execution [23]. Software testing techniques depend on the availability of test oracle [24]. 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 expertise. Some common issues related to the design of test oracle are as follows:

1. It is assumed that the test results are observable and comparable with the oracle [25].
2. Ideally, test oracle would satisfy desirable properties of program specifications [23].
3. As pointed out by Weyuker, "truly general test oracles are often unobtainable" [25].

Post-conditions are commonly used test oracle in automated software testing. Post-conditions are conditions which must be true after a method is successfully executed. In such oracle a fault is signalled when a post-condition is violated [26]. Some common artefacts used as oracles are as follows:

1. Specification and documentation of software.
2. Products similar to the SUT but different in algorithm.
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 with another for consistency.
6. Generation of model for verification of SUT behaviour.
7. Manual analysis by human experts to verify the test results.

2.2 Software testing view points

Software testing from various view points is presented in Figure 2.2 and each one is described in the following sections.

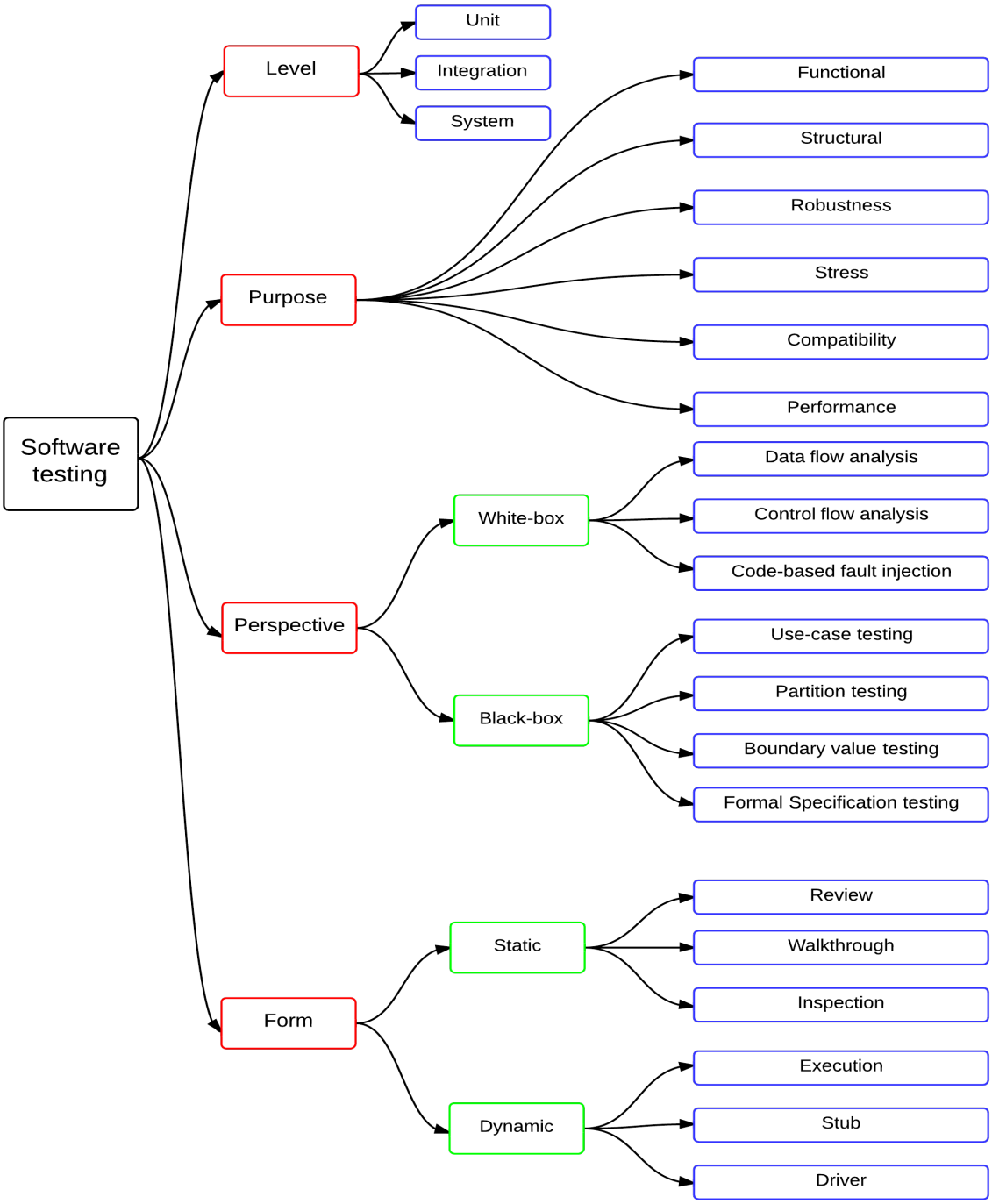


Figure 2.2: Software testing from various view points

2.3 Software Testing Levels

The three main levels of software testing reported in the literature includes Unit testing, Integration testing and System testing [27]. Unit testing deals with evaluation of code piece-by-piece and each piece is considered as independent unit. Integration testing is performed to make sure that the integration in the components formed by the combination of units are working properly. System testing ensures that the system formed by the combination of components proceeds properly to give the required output.

2.4 Software Testing Purpose

The purpose of software testing is to achieve high quality by identifying and eliminating faults in the given SUT. Maximum number of faults can be identified if software is tested exhaustively. However, exhaustive testing is not always possible because of limited resources and infinite number of input values that a 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 aspects of software are working correctly. Structural testing involves analyses of code structure for generating test cases in order to evaluate paths of execution and identification of unreachable or dead code. Robustness testing includes observation of the software behaviour when it receives input outside the expected 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 [28]. Compatibility testing is performed to check and find any fault in the interaction of software with the underlying application or system software.

2.5 Software Testing Perspective

Based on the perspective taken, software testing is divided into white-box and black-box testing as described below.

2.5.1 White-box Testing

White-box testing also known as structural testing is a technique that takes into consideration the structure of the software. The testers should know about the complete structure of the software in order to make necessary modifications, if so required. Test cases are derived from the code structure and test passes if the results are correct and the proper code

is followed during test execution [29]. A simplified version of white-box testing is shown in Figure 2.3. Some commonly used white-box testing techniques are as follows:

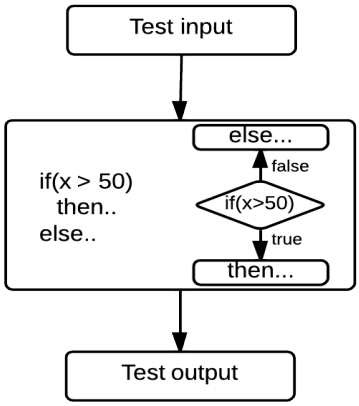


Figure 2.3: White-box testing

2.5.1.1 Data Flow Analysis

Data Flow Analysis (DFA) is a technique which focuses on the input values by observing the behaviour of respective variables during execution of the SUT [30]. In this technique a Control Flow Graph (CFG) representing all possible states of a program is drawn to determine the paths that are traversed by the program during test execution. Test cases are generated and executed to verify conformance with CFG on the basis of data flow.

The data flow analysis observes the program execution as data flow from input to output. The data may transform into several intermediary steps before reaching the final state. The transformation process is prone to several errors e.g. references made to non existing variables, values assigned to undeclared variables and change of variables in undesired manner. Ordered use of data is crucial to ensure that the aforementioned errors do not occur [31].

2.5.1.2 Control Flow Analysis

Control Flow Analysis (CFA) is a technique which takes into consideration the control structure of a given SUT. Control structure is the order in which the statements, instructions and function-calls are executed. Like DFA, this technique also involves drawing of CFG 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 flow. For example,

the CFA ensures that the selected test cases execute all possible control choices at least once when two or more control choices are available to reach a particular state in the given SUT.

2.5.1.3 Code-based Fault Injection Testing

It is a testing technique in which new instructions are added to the code of the SUT at one or more locations to analyse the software behaviour in response to the given instructions [32]. The process of code addition (instrumentation) is performed before compilation and execution of software. Code is added to find error handling behaviour of software, examine the capability of test procedure and measure the code coverage achieved by the testing process.

2.5.2 Black-box Testing

Black-box testing also known as functional testing is a technique that takes into consideration the function of the software. The testers may not know about the structure of the software. Test cases are derived from the software specifications and test passes if the result is according to expected output irrespective of the internal code followed during test execution [33]. A simplified version of black-box testing is shown in Figure 2.4.

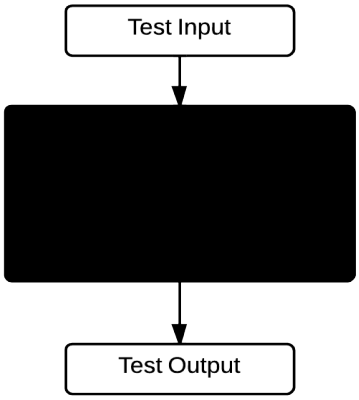


Figure 2.4: Black-box testing

Some commonly used black-box testing techniques are as follows.

2.5.2.1 Use-case Based Testing

It is a testing 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, flow of events, pre-conditions, post-conditions, extension points, context and activity diagrams. The use-case contains all the relevant information required for test case, therefore, it can be easily transformed into a test case [34]. Use-case testing is effective in terms of cheap generation of test cases, avoidance of test duplication, increased test coverage, easier regression testing and early identification of missing requirements.

2.5.2.2 Partition Testing

It is a testing technique in which the input domain of a given SUT is divided into equal sub-domains for testing each sub-domain individually. The division is based on software specifications, code structure and the process involved in software development [35]. The performance of partition testing is directly proportional to the quality of sub-domain [36]. Division of input domain into equal partitions is often difficult. To overcome the problem, a new version of partition testing called proportional partition testing is devised [1]. 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. Ntafos [37] have provided experimental evidence in support of better performance of proportional partition testing than partition testing.

2.5.2.3 Boundary Value Analysis

Boundary Value Analysis (BVA) is a testing technique based on the assumption that errors 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 [38], boundary values contain minimum, maximum, internal and external values specified for a system. The following code illustrates the ability of BVA to find a failure.

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

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 failure is generated when the SUT tries to set the array size to a negative value.

BVA and partition testing may be used in combination by choosing test values from the whole input domain and also from the borders of each sub-domain. Reid et al. [39] have provided evidence in support of better performance of BVA in combination with partition testing as compared to each individually. They have attributed better performance to accurate identification of partition and selection of boundary values.

2.5.2.4 Formal Specification Testing

It is a testing technique based on mathematical model which provides the opportunity to handle the specifications mathematically. This feature facilitates isolation, transformation, assembly and repackaging of the information available in the specifications for use as test cases [40].

The formal specification testing is more productive because of the creation of test cases independent from the code of the SUT [24]. The effort of generating test oracle is avoided by using available specification model for verifying the test results [41].

2.6 Software Testing Types

There are two types of software testing: static and dynamic.

Static testing involves analysis of test cases statically for checking errors without executing the test cases. Static analysis is applicable to evaluate the quality of software code and supporting documents including requirements, design, user manual, technical notes and marketing information. Reviews, walkthroughs and inspections are most commonly used techniques for static testing.

Dynamic testing involves execution of test cases against SUT. The results obtained are analysed against expected output to find any error in the software. Unit testing, integration testing, system testing and acceptance testing are most commonly used methods for dynamic testing [42].

2.6.1 Manual Software Testing

Manual testing is the technique in which the tester writes the code by hand to create test cases and test oracles for finding faults in software [43]. Its advantage is that the test cases

are usually generated to target those parts of the SUT which are assumed to be more error-prone. It is highly useful to capture special cases for which automated testing might not guess. Manual unit testing is particularly popular because of the availability of xUnit family of tools, e.g. jUnit for Java, sUnit for Smalltalk and pyUnit for Python. According to a recent survey, 79% of the Microsoft developers write unit tests for software testing [44]. The tools automate the execution process of the hand written test cases providing significant practical benefits.

The limitations of manual testing are that the testers should have appropriate skills, experience and knowledge of the SUT for its evaluation from different perspectives.Manual testing may be effective at smaller scale of software testing but at a larger scale it is generally laborious, time consuming and error-prone [45]. Manual testing usually produce low coverage because sheer numbers of test cases are required for high coverage.

2.6.2 Automated Software Testing

Automated software testing refers to the technique in which an automated tool is used to perform the testing process automatically [4]. There are some tools available for automating a part of testing process like generation of test cases or execution of test cases or evaluation of results while other tools are available for automating the whole testing process.

A fully automated testing system is capable to test software without any user intervention. This is usually achieved by utilizing the contracts (preconditions, postconditions and invariants) embedded in the program code. Preconditions are used to filter out invalid inputs and postconditions are used as oracle [4]. Eiffel [46] and Spec# barnett2005spec languages have built-in contracts whereas Java can use add-ons like JML [47], iContract [48] or OCL [49] to enable it. Automated software testing may involve higher initial cost but brings the key benefits of lower cost of production, higher productivity, maximum availability, greater reliability, better performance and ultimately proves highly beneficial for any organisation. Automated testing is particularly effective when the nature of job is repetitive and it is performed on routine basis like unit testing and regression testing where the tests are re-executed after each modification [50]. The use of automated software testing makes it possible to test large volumes of code, which may be impossible otherwise [51].

2.7 Test Data Generation

Test data generation in software testing is the process of identifying and selecting input data which satisfies the given criterion. A test data generator is used to assist testers in

the generation of data while the selection criterion defines the properties of test cases to be generated based on the test plan and perspective taken [14]. 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 [52]. 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. Strategy handler defines the test case selection criteria. This may include the formalisation of test coverage criterion, the selection of paths and normalisation of constraints. It may also get input from program analyser or user before or during test execution.

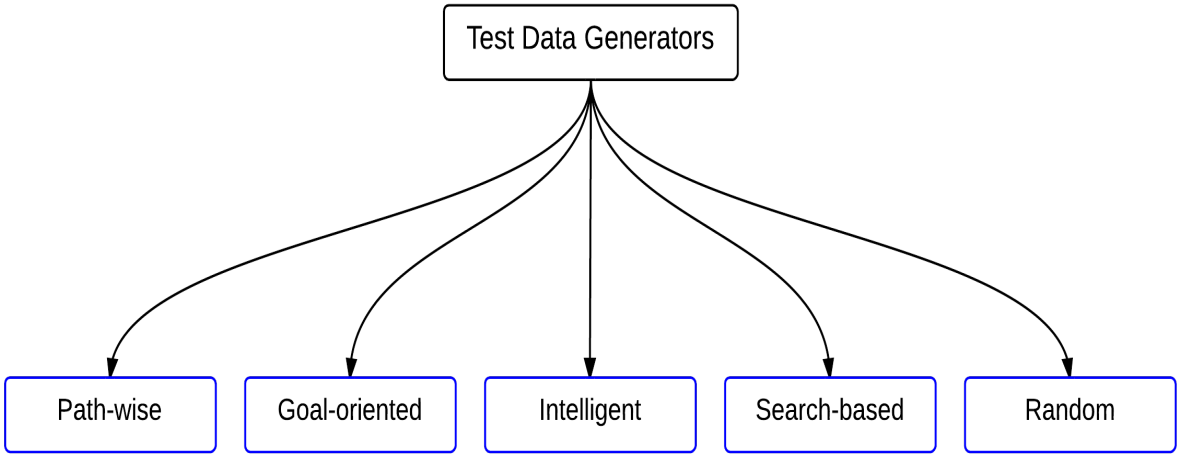


Figure 2.5: Types of test data generator

Generator generates test cases according to the selection criteria identified by the strategy handler. Test data generators are classified into path-wise, goal-oriented, intelligent, search-based and random on the basis of approach followed as shown in Figure 2.5. Each type is briefly described in the following section.

2.7.1 Path-wise Test Data Generator

Path-wise test data generator selects a set of test data from the input domain in order to target path, branch and statement coverage in a given SUT. The process is typically accomplished in three stages: CFG construction, path selection and test data generation.

In the process a particular path is selected either manually or by automatic means. The generator identifies and generates relevant test data required for execution of the intermediary statements along the selected path. The data generated in path testing expresses boolean true/false behaviour for each node in the path. A complete path contains multiple sub-domains, each consisting of test inputs required to traverse the path. The boundary of sub-domains are obtained by the predicates in the path condition.

2.7.2 Goal-oriented Test Data Generator

Goal-oriented test data generator generates data to target a specific program point [53]. The tester can select any path among a set of existing paths as long as it reaches the specified program point. This technique utilizes runtime information for computing accurate test data [54]. Among various methods used in goal-oriented test data generation the following two commonly adopted approaches are briefly described.

2.7.2.1 Chaining Approach

The chaining approach uses data dependent analysis to guide the test data generation. In the process all the related statements affected by execution of the statement under test are selected automatically. The dependent statements are executed before the selected statement to generate the required necessary data for the execution of the statement under test [54]. 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 constraint. The Branch classification process identifies critical, semi-critical and non-critical nodes for each branch. During execution, the classification guides the search process to select specific branches to reach the goal node.

2.7.2.2 Assertion-oriented Approach

The assertion-oriented approach adds assertions 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 is a constraint applicable to a state of computation which can be 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 due to fault in program code, in pre or post-condition or a fault in the assertion itself.

2.7.3 Intelligent Test Data Generator

Intelligent test data generator is used to overcome the problems like generation of meaningless data, duplicated data and complex data associated with traditional data generators. The approach increases user confidence in the generated test data and the testing process [51]. It helps in finding the appropriate test data by performing sophisticated analysis to anticipate different situations that may arise at any point in the SUT such as genetic algorithm. The approach produces test data which satisfy the SUT requirements but consumes more time and resources.

2.7.3.1 Genetic Algorithm

The genetic algorithm is a heuristic that mimics the evolution of natural species for searching optimal solution of a problem. It is guided by control dependencies in the program to search for data which satisfy test requirements. The algorithm evaluates the existing test data, and guides the direction of search by using the program control-dependence graph [55]. The approach emphasises on the execution of a given statement, branch, path and condition in the given SUT. The benefit of the genetic approach is quick generation of test cases with proper focus and direction. The new test cases are generated by applying simple operations on the appropriate existing test cases having good potential of satisfying the test requirements. The success of the approach depends heavily on the way in which the existing test data are measured [55].

2.7.4 Search-based Test Data Generator

Search-based test data generator uses meta-heuristic algorithms to generate test data. In this technique each input vector `x` is associated with a measure `cost(x)` which represents the difference between the input value `x` and the set goal. Input value closer to the set goal has low cost value as against the input value away from the set goal.
Let us consider the following program.

```
void test(int x,int y) {
    if (x >= 20) {
        y = z;
    }
    else {
        y = 2 * z;
    }
}
```

Suppose we want the true branch of the if/else statement to be executed. An input value of `x == 25` clearly satisfies the predicate, and a value of `x == 15` is closer to satisfy the predicate than a value of `x == 5`. We evaluate a cost function probe of the form `cost(x) = max {0, 20 - x}`. Thus `x == 25` has cost `0`, `x == 15` has cost `5` and `x = 5` has cost `15`. It is apparent that 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`. The data which satisfy each of the predicates at different points may be selected to follow a particular path in the code. This leads to a cost function which combines costs at each of the relevant branching point. 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.

2.7.5 Random Test Data Generator

Random test data generator is the simplest approach for generation of test data. Its advantage is the adaptability 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 [17]. If a fault is only revealed by a small percentage of the program input it is said to be a semantically small fault. As an example of a semantically small fault, consider the following code:

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

```
}
```

It is clear that the probability of execution of the first statement is significantly lower than that of the second statement. As the structure gets more and more complex, the probability of execution decreases accordingly. Thus semantically small faults are hardly detectable by using random test data generator.

2.8 Random Testing

Random testing is mentioned in the literature for the first time by Hanford in 1970 who reported syntax machine, a tool that randomly generated data for testing PL/I compilers [56]. Later in 1983, Bird and Munoz described a technique to produce randomly generated self checking test cases [57]. 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 [58]. As stated by Richard [59], the test data are randomly selected from the identified input domain 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 case fails if the results are not according to the specifications reflecting a fault in the given SUT and vice versa. Working mechanism of random testing is shown in Figure 2.6.

Generating test data by random generator is economical and requires less intellectual and computational efforts [60]. 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, which makes it less effective [61, 62]. Myers mentioned random testing as one of the least effective testing technique [10]. However, Ciupa et al. [15] stated that Myers statement is not based on any experimental evidence. Later experiments performed by several researchers [43, 59, 63] confirmed that random testing is as effective as any other testing technique. It is reported [64] 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 shows that random testing and partition testing are equally effective [35]. Ntafos [37] conducted a comparative study and concluded that random testing is more effective as compared to

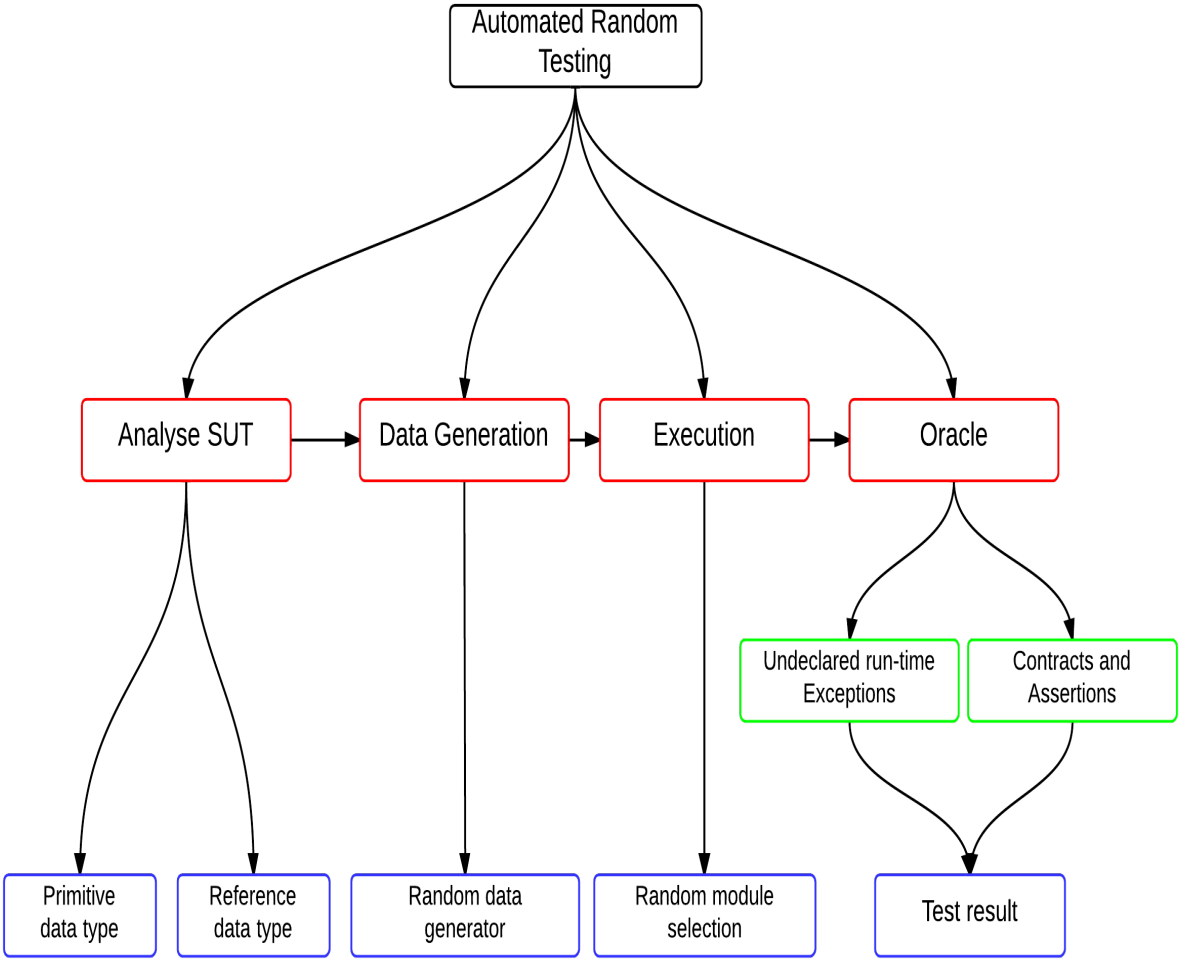


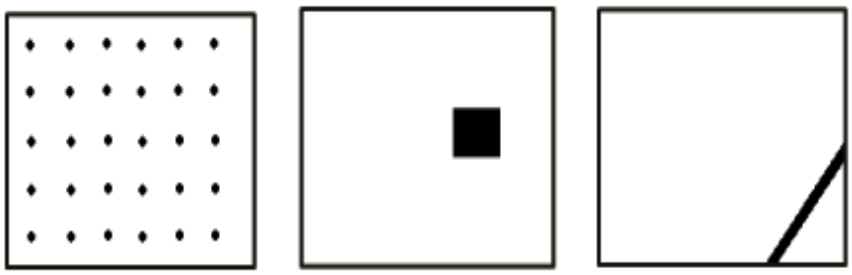
Figure 2.6: Working mechanism of random testing

proportional partition testing. Miller et al. [65] generated random ASCII character streams and used the Unix utilities for abnormal terminating and non-terminating behaviours. Subsequently, the technique was extended to discover errors in software running on X Windows, Windows NT and Mac OS X [66, 67]. Other famous studies using random testing includes low-level system calls [68] and file systems used in missions at NASA [69]. The use of random testing at academic and commercial level reported in the literature cited above provides evidence to the fact that it is a well accepted and widely adapted technique for software testing.

2.9 Pass and Failure-domain

The sequence of test data across the input domain for which the software behaves correctly is called pass-domain and for which the software behave incorrectly is called failure-domain. Chan et al. [1] observed that input inducing failures are contiguous and form certain geometrical shapes in the whole input domain. They divided these into point, block and strip failure-domains as described below.

1. Point domain: In the point failure-domain, input inducing failures are scattered across the input domain in the form of stand-alone points. Example of point failure-domain is the failure caused by the statement: $total = num1/num2$; where $num1$, $num2$ and $total$ are variables of type integer.
2. Block domain: In the block failure-domain, input inducing failures lie in close vicinity to form a block in the input domain. Example of block failure-domain is failure caused by the statement: $if((num{>}10)\&\&(num{<}20))$. Here 11 to 19 are a block of failures.
3. Strip domain: In the strip failure-domain, input inducing failures form a strip across the input domain. Example of strip failure-domain is failure caused by the statement: $num1 + num2 = 20$. Here multiple values of $num1$ and $num2$ can lead to the fault value 20.



(a) Point domain (b) Block domain (c) Strip domain

Figure 2.7: Failure domains across input domain [1]

The Figure 2.7 shows failure domains across the input domains. The squares in the figure indicate the whole input domain. The white space in each square shows legitimate and faultless values while the black colour in the form of points, block and strip indicate failures in the form of point, block and strip failure-domains.

2.10 Versions of Random testing

Researchers have tried various approaches to develop new versions of random testing for better performance. The prominent improved versions of random testing are shown in Figure 2.8.

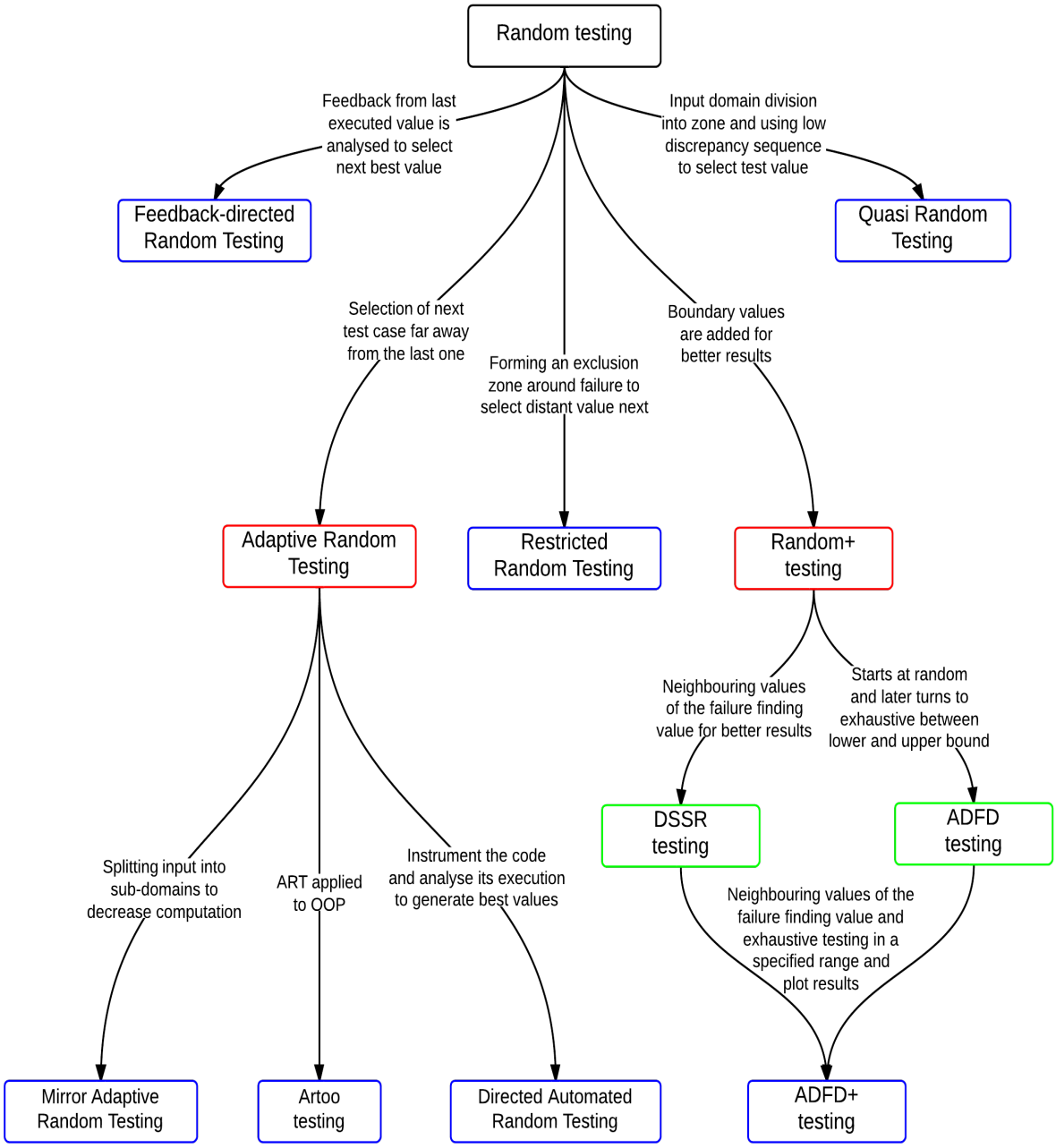


Figure 2.8: Various versions of random testing

2.10.1 Random+ Testing

The random+ testing [4] is an extension of the random testing. 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 [70] are the values at the start and end of a particular data type. For instance, such values for `int` could be -3, -2, -1, 0, 1, 2, 3, `Integer.MAX_VALUE`, `Integer.MAX_VALUE-1`, `Integer.MAX_VALUE-2`, `Integer.MIN_VALUE`, `Integer.MIN_VALUE+1`, `Integer.MIN_VALUE+2`. These special values can add a significant improvement to a testing method.
Let us consider the following piece of code:

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

In the given 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 tries to build an array of negative size. Similarly, the tester may add some other special values that are considered effective for finding faults in 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. Interesting values included in the list are given higher priority than random values because of their relevance and better chances of finding faults in the given SUT. As reported in the literature, interesting values have high impact on the results, particularly for detecting problems in specifications [43].

2.10.2 Adaptive Random Testing

Adaptive random testing (ART) proposed by Chen et al. [71] is based on the previous work of Chan et al. [1] regarding the existence of failure domains across the input domain (Section 2.9). Chen et al. [71] 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 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. The technique uses the candidate set as well as the executed set both of which are initially and as soon as the testing begins ART fills the candidate set with

randomly selected test cases from the input domain. The first test case selected at random from the candidate set is executed and stored in the executed set. The second test case is the one selected from the candidate set which is located far away from the previously executed test case. The process continues till test completion and provide greater chances of finding failures from failure domains.

Chen et al. [71] used ART in their experiments with the number of test cases required to detect first fault (F-measure) as a performance matrix instead of the traditional matrices (P-measure) and (E-measure). The results showed up to 50% increase in performance compared to random testing. However, the authors pointed out indicated their concern regarding the issues of spreading test cases across the input domain for complex objects, efficient ways of selecting candidate test cases and higher overhead.

2.10.3 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) is an improvement on ART by using mirror-partitioning technique to reduce the overhead and decrease the extra computation involved in ART [72].

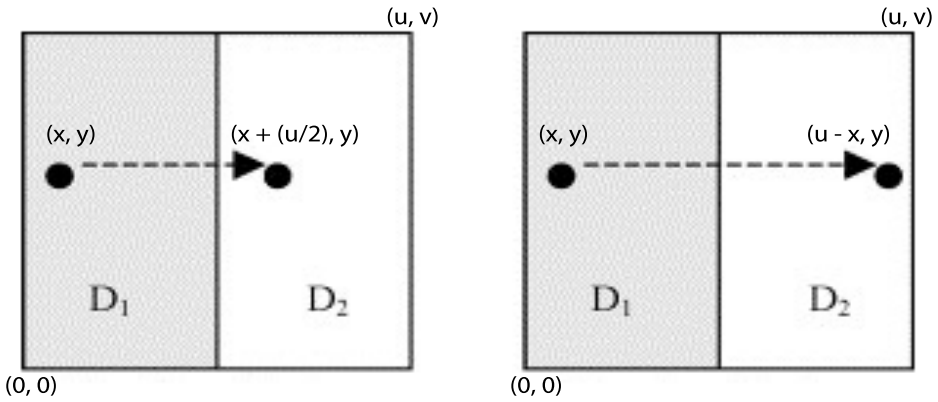


Figure 2.9: Mirror functions for mapping of test cases

In this technique, the input domain of the program under test is divided into n number of 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 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, so we can linearly translate these test cases to achieve the mirror effect, i.e. $(x+(u/2),y)$ as shown in Figure 2.9. The experimental results by comparing MART with ART provides evidence of equally good performance of the two techniques with the added advantage of lower overhead in MART by using only one quarter of the calculation as compared with ART [72].

2.10.4 Restricted Random Testing

Restricted Random Testing (RRT) is another approach [58] to overcome the problem of extra overhead in ART. The RRT achieves this by creating a circular exclusion zone around the executed test case. A candidate is randomly selected from the input domain as a next test case. Before execution the candidate is checked and discarded if it lies inside the exclusion zone. This process repeats until a candidate present outside the exclusion zone is selected. It 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.

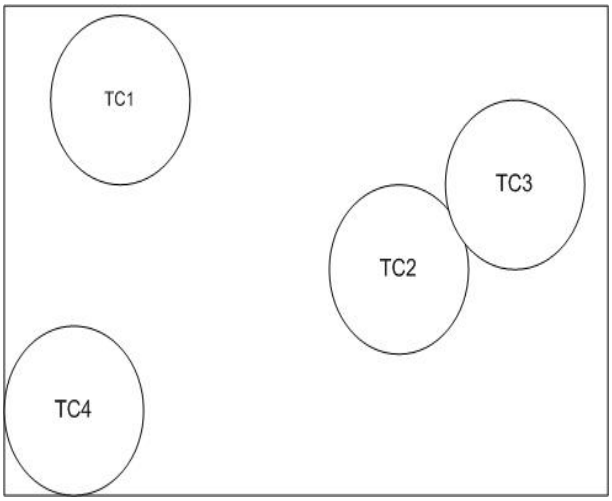


Figure 2.10: Input domain showing exclusion zones around selected test cases

The authors compared MART with ART to find the comparative performance and reported that the performance of both techniques is similar with the added advantage of lower overhead in case of MART which uses only one quarter of the calculations as compared with

ART. They further found that RRT is up to 55% more effective than RT in terms of F-measure.

2.10.5 Directed Automated Random Testing

Godefroid et al. [17] proposed Directed Automated Random Testing (DART). In DART process, the given SUT is instrumented to track the dynamic behaviour of the SUT at run time. It also identifies external interfaces of a given SUT. These interfaces include external variables, external methods and the user-specified main method responsible for program execution. After that it automatically generates test drivers for running the randomly generated test cases. Finally 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 uses 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), public methods or interfaces are used to mimic the scenario. DART attempts to cover different paths of the program code to trigger errors. Its oracle consists of checking for crashes, failed assertions and non-termination.

2.10.6 Quasi Random Testing

Quasi-random testing (QRT) is a testing technique [73] 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 per-mutated 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.

2.10.7 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) is a technique that generates unit test suite at random for object-oriented programs [74]. 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 a unit test that produces $IllegalArgumentException$ on execution is discarded because the arguments used in the unit test are not according to the required type.

Pacheco et al. performed a case study in which a team of testers applied FDRT to a critical component of .NET architecture. Results showed that the faults discovered by FDRT in 15 hours of manual processing and 150 hours of CPU processing are more than a test engineer finds in one year by manual and other automated techniques. As a result FDRT has been added to the tool list used at Microsoft for the betterment of software [**?**].

2.10.8 The Artoo Testing

The Adaptive random testing for object oriented (Artoo) testing is based on object distance. Ciupa et al. [75] 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, values of primitive fields and values of reference fields. Strings are treated in terms of directly usable values and Levenshtein formula [76] is used as a distance criterion between the two strings.

In the Artoo testing, two sets are taken, the candidate-set containing the objects ready to be run by the system and the used-set which is initially empty. The 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 in the used-set. The process continues till the fault is found or the objects in the candidate-set are finished [75].

The Artoo testing, implemented in AutoTest [60], was evaluated in comparison with Directed Random (D-RAN) testing [17] by selecting classes from EiffelBase library [77]. The experimental results indicated that some faults found by the Artoo were not identified by the D-RAN strategy. Moreover the Artoo found first fault with small number of test cases than the D-RAN testing. However, more computation was required to select a test case in the Artoo strategy and the process required more time and cost to generate test cases as compared to D-RAN testing.

2.11 Automatic Random Testing Tools

A number of automatic random testing tools used in research and reported in the literature are briefly described in the following section.

2.11.1 JCrasher

Java Crasher (JCrasher) is an automatic robustness testing tool developed by Csallner and Smaragadakis [2]. JCrasher tests the Java program with random input. The exceptions thrown during the testing process are recorded and compared with the list of acceptable standards defined as heuristics. The undefined runtime exceptions are considered as failures. JCrasher randomly tests only the public methods of SUT based on the fact that users interact with programs through public methods.

The working mechanism of JCrasher is illustrated by testing a $.java$ program as shown in Figure 2.11. The source file is first compiled using $javac$ to get the byte code. The byte code obtained is passed as input to JCrasher, which uses Java reflection library [78] 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 a file $TTest.java$. The file is compiled and executed by JUnit. All exceptions produced during test case executions are collected and compared with robustness heuristic and resulted violations are reported as errors.

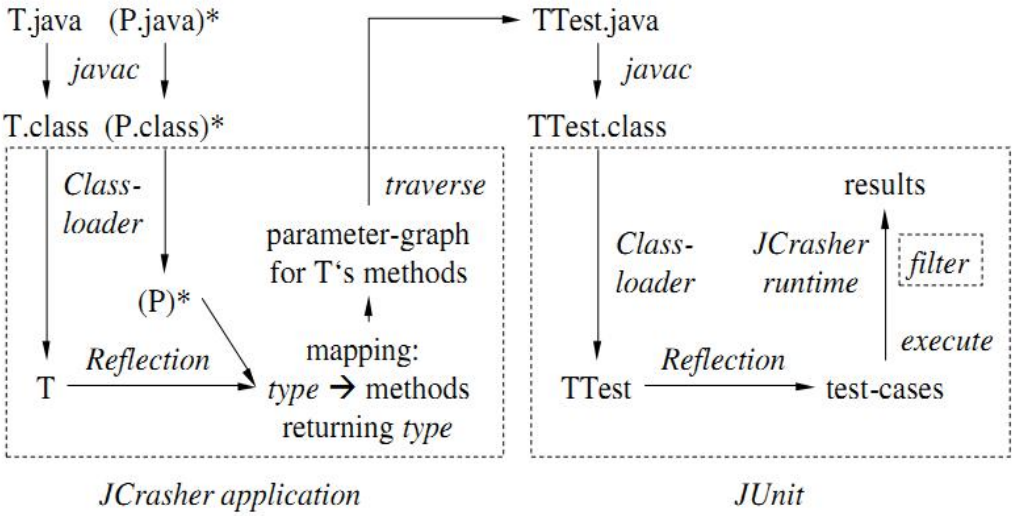


Figure 2.11: How a class T can be checked for robustness with JCrasher. First, the JCrasher application generates a range of test cases for T and writes them to TTest.java. Second, the test cases can be executed with JUnit, and third, the JCrasher runtime filters exceptions according to the robustness heuristic [2]

JCrasher is a pioneering tool with the capability to perform fully automatic testing, including test case generation, execution, filtration and report generation. Its novel feature is the

generation of test cases as JUnit files which can be easily read and used for regression testing. Another important feature of JCrasher is independent execution of each new test on clean-slate. This ensures that the changes made by the previous tests do not affect the new test.

2.11.2 Jartege

Java random test generator (Jartege) is an automated testing tool [79] that randomly generates unit tests for Java classes with contracts specified in JML. The contracts include class invariants and the pre and post-conditions of methods. Initially Jartege uses the contracts to eliminate irrelevant test cases and later on the same contracts serve as test oracle to differentiate between faults and false positives. Jartege uses simple random testing to test classes and generate test cases. Additionally, testing of a specific part of the class can be prioritized by changing the parameters to get interesting sequence of calls if so desired by the tester. 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 faults either in the methods or in the specifications. The Jartege checks for faults in program code as well as in specifications and the JUnit tests produced by Jartege can be used later for regression testing. Its limitation is the prior requirement of JML specifications of the program.

2.11.3 Eclat

Eclat [3] is an automated testing tool which generates and classifies unit tests for Java classes. The tool takes a software and a set of test cases for which the software runs properly. Based on the correct software operations an operational model is created to test the selected data. If the operational pattern 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.

The testing process is accomplished by Eclat in three stages as shown in Figure 2.12 . In the first stage, a small subset of test inputs is selected, which may likely reveal faults in the given SUT. 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.

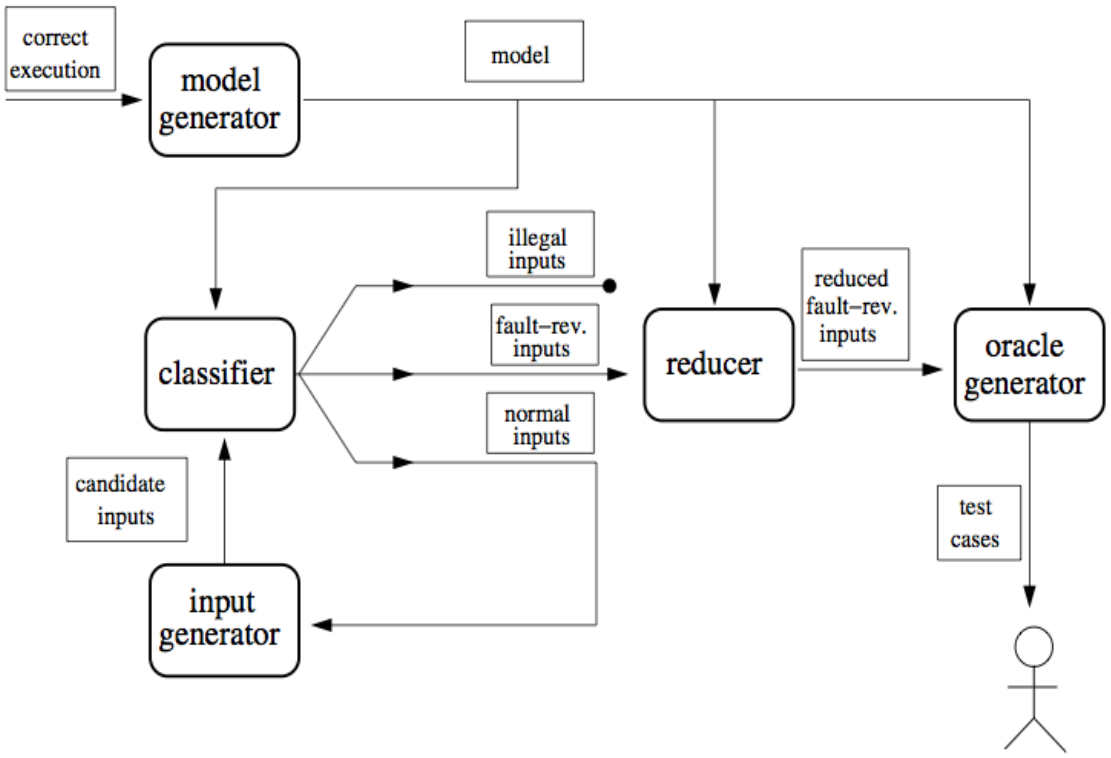


Figure 2.12: The input selection technique. Implicit in the diagram is the program under test. Rectangles with rounded corners represent steps in the technique, and rectangles with square corners represent artifacts [3]

Csallner and Smaragdakis [2] compared Eclat with JCrasher by executing nine programs on both tools. They 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. Any error in the pool may lead to the creation of wrong operational model which will adversely affects the testing process.

2.11.4 Randoop

Random tester for object oriented programs (Randoop) is the tool used for implementing FDRT technique [74]. Randoop is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes a set of classes, contracts, filters and time limit as input and gives a suite of JUnit for Java and NUnit for .Net program as output. 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. Arguments for these methods are selected from the predefined pool in case of primitive type and as sequence of null values 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 or NUnit test suite. The following command runs Randoop to test OneDimPoint-FailDomain for 100 seconds in CLI mode. Values is a text file containing interesting values which is maintained manually by the tester.

```
$ java randoop.main.Main gentests    \
 --testclass=OneDimPointFailDomain    \
 --testclass=Values --timelimit=100
```

In terms of coverage and number of faults discovered, Randoop using FDRT technique was compared with JCrasher and JavaPathFinder and 14 libraries of both Java and .Net were evaluated [80]. The results showed that Randoop achieved more branch coverage and better fault detection than JCrasher.

2.11.5 QuickCheck

QuickCheck [81] is a lightweight random testing tool used for testing of Haskell programs [82]. Haskell is a functional programming language where programs are evaluated by using expressions rather than statements. Most of the functions in Haskell are pure except the IO functions, therefore QuickCheck mainly focuses on testing pure functions. QuickCheck 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.

QuickCheck takes as inputs the function to be tested and properties of the program (Haskell functions). The tool uses built-in random generator to generate test data, but it is also capable to use custom built data generator. The tester-defined-properties must hold while

executing the function on the generated test data. Any violation of the defined properties will indicate error in the function.

2.11.6 AutoTest

The AutoTest is used to test Eiffel language programs [83]. The Eiffel language uses the concept of contracts which is effectively utilized by AutoTest. For example, the auto generated inputs are filtered using pre-conditions and non-complying test inputs are discarded. The post-conditions are used as test oracle to determine whether the test passes or fails. Besides automated testing the AutoTest also allows the tester to manually write the test cases to target specific section of the code. The AutoTest takes one or more methods or classes as inputs and automatically generates test input data according to the requirements of the methods or classes. As shown in Figure 2.13, the architecture of AutoTest

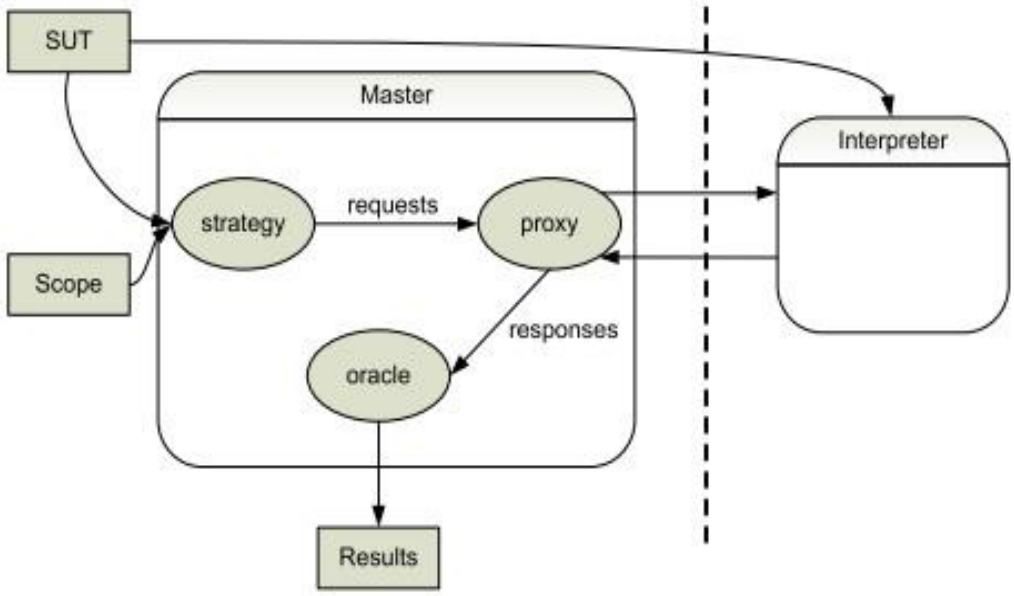


Figure 2.13: AutoTest architecture [4]

can be split into the following four parts:

1. **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 random data to evaluate the methods/classes under test.
2. **Proxy:** It handles inter-process communication. The proxy receives execution requests from the strategy and forward these to the interpreter. It also sends the execution results to the oracle part.

3. **Interpreter:** It executes operations on the SUT. The most common operations 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 evaluates the results to see if the contracts are satisfied. The outcome of the tests are formatted in HTML and stored on disk.

2.11.7 TestEra

TestEra [5] is a novel framework for auto generation and evaluation of test inputs for a Java program. It takes specifications, numerical value and the method to be tested 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 and the results are compared against the post-conditions of the method serving as oracle. Any test case that fails to satisfy post-condition is considered as a fault. TestEra uses the Alloy modelling language [84] to express con-

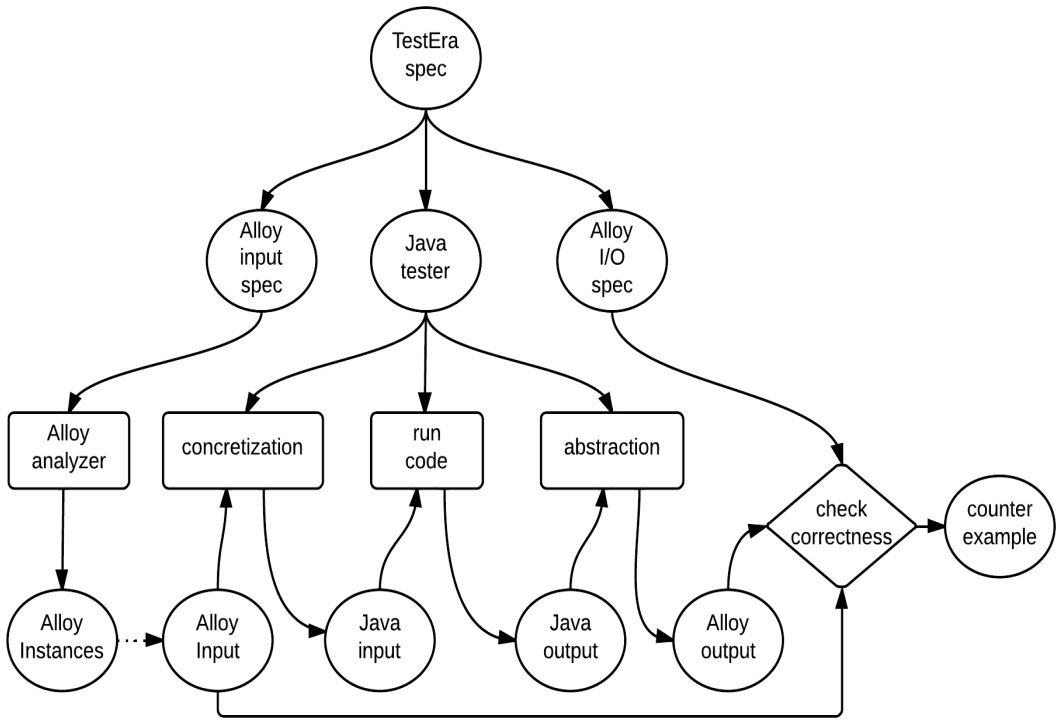


Figure 2.14: TestEra framework [5]

straints on test inputs and uses Alloy Analyser [85] to solve these constraints and generate

test inputs. Alloy Analyzer performs the following three functions: (a) it translates Alloy predicates into propositional formulas(b) it evaluates the propositional formulas to find the outcome (c) it translates each outcome from propositional domain into the relational domain.

TestEra uses program specifications to guide the auto generation of test inputs as against Jartege (Section 2.11.2) and AutoTest (Section 2.11.6) which use program specifications for filtrating the irrelevant random generated test data. However, all the three tools use program specifications in a similar way for test oracle.

2.11.8 Korat

Korat [86] like TestEra [5], both developed by the same team, is a framework for automated testing of Java programs based on the formal specifications [87]. Korat uses Java Modelling Language (JML) for specifications. It uses bounded-exhaustive testing in which the code is tested against all possible inputs within the specified bounds [88]. Korat takes imperative predicates and finitization value as inputs. It systematically explores the input space of the predicates 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 the fields accessed during executions. These inputs are taken as test cases.

Korat uses `repOK()` and `checkRep()` methods. The `repOK()` is used to check the class invariants for validating test inputs while `checkRep()` is used to verify the post-conditions for validating the test case. Korat does not require existing set of operations to create input values. Therefore it has the advantage of generating input values that may be difficult or impossible with a given set of operations. The disadvantage of the approach is the requirement of significant manual efforts [61].

2.11.9 YETI

York Extensible Testing Infrastructure (YETI) is an open-source automated random testing tool. YETI, coded in Java, 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, capability to test programs using multiple strategies, high speed tests execution, real time logging, GUI support and auto generation of test report at the end of test session. Detailed information about YETI is presented in Chapter 3.

2.12 Summary

The software testing is summarized graphically with the help of two dimensional venn diagram as shown in Figure 2.15. The positive x-axis represent black-box while negative x-axis represent white-box testing. Similarly on positive y axis we have dynamic testing and on negative y axis we have static testing. If a testing technique is black-box and dynamic then it will fall in 0 to 90 degree and if it is black-box and static it will fall in 270 to 360 degree. On the other hand if the test is white-box and dynamic then it will fall in 90 to 180 degree and if the test is white-box and static then it will fall in 180 to 270 degrees.

Figure 2.15: Types of software testing

The chapter gives an overview of software testing, including definition, common types, need, purpose and uses. It differentiates manual and automated software testing and describes various ways of software test data generation, being the crucial part of any testing system. The later part describes random testing and the various ways of improving

the performance of random testing. Finally, information is presented on how the automated testing tools implement random technique for software testing. Main features of automatic testing tools used in random testing are summarized in Figure 2.16.

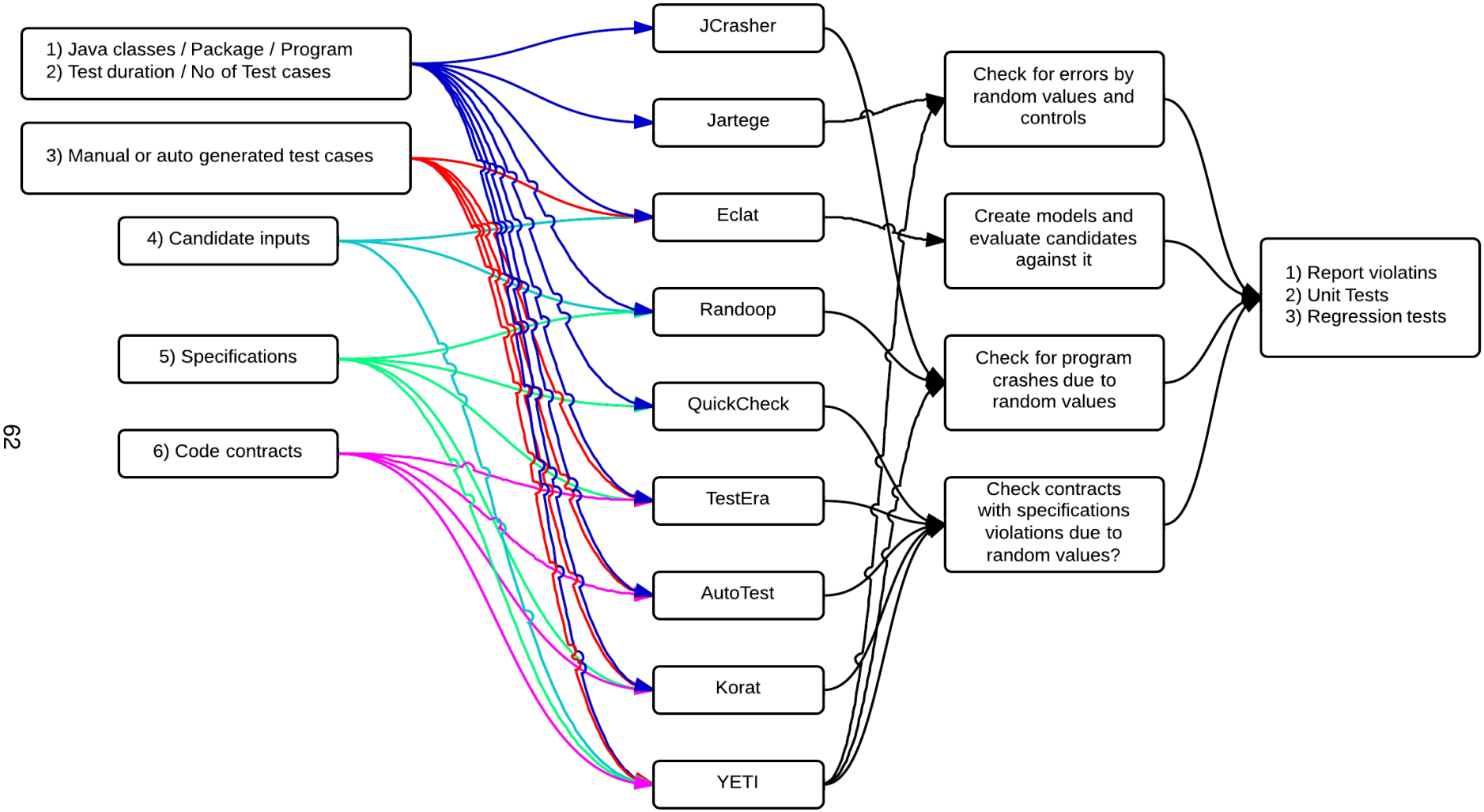


Figure 2.16: Main features of automatic testing tools using random testing

Chapter 3

York Extensible Testing Infrastructure

3.1 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 [89]. YETI takes program byte code as input and execute it with random generated but syntactically-correct inputs to find a failure. 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 [90]. The latest version of YETI can be downloaded from `www.yetitest.org`. Figure 3.1 briefly presents the working process of YETI.

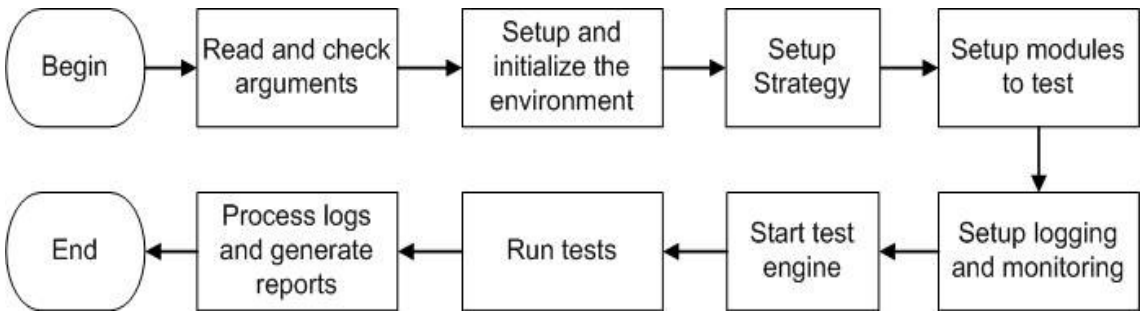


Figure 3.1: Working process of YETI

3.2 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.

3.2.1 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 3.3.

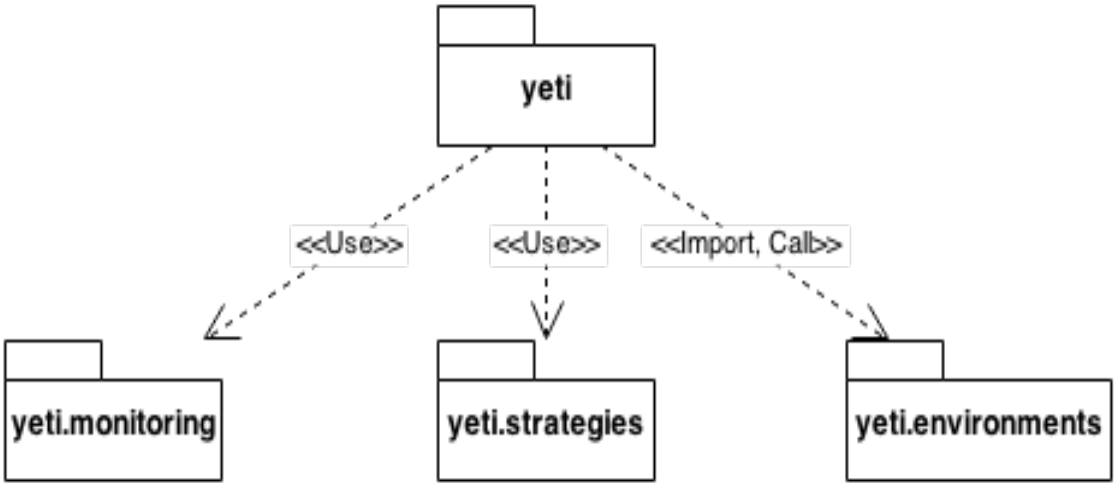


Figure 3.2: Main packages of YETI with dependencies

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

1. **Yeti:** 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:** It prints debugging and testing logs.
3. **YetiLogProcessor:** It is an interface for processing testing logs.

4. **YetiEngine:** It binds YetiStrategy and YetiTestManager together, which carry out the actual testing process.
5. **YetiTestManager:** 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:** It is an abstract class for test initialization.

3.2.2 Strategy

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

1. **YetiStrategy:** It is an abstract class which provides interface for every strategy in YETI.
2. **YetiRandomStrategy:** 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:** 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:** It extends random+ strategy by adding the values surrounding the failure value. The strategy is described in detail in Chapter 4.
5. **ADFDStrategy:** It extends random+ strategy by adding the feature of graphical representation of failures and their domains. The strategy is described in detail in Chapter 5.
6. **YetiRandomDecreasingStrategy:** It extends the random+ strategy by setting the probability value to starts at 100% and ends at 0% when the test finishes.
7. **YetiRandomPeriodicStrategy:** It extends random+ strategy by setting the probability in such a way that it decreases and increases randomly during test session.

3.2.3 Language-specific Binding

The language-specific binding facilitates modelling of programming languages. It is extended to provide support for a new language in YETI. The language-specific binding includes the following classes:

1. **YetiVariable:** It is a sub-class of YetiCard, which represents a variable in YETI.
2. **YetiType:** It represents type of data in YETI, e.g. integer, float, double, long, boolean and char.
3. **YetiRoutine:** It represents constructor, method and function in YETI. It has a specific name, a return type and a It is a super type of routines which represents functions, methods and constructors. A routine is given a name, return type and list of arguments.
4. **YetiModule:** It represents a module in YETI and stores one or more routines of the module.
5. **YetiName:** It represents a unique name assigned to each instance of YetiRoutine.
6. **YetiCard:** It represents a wildcard or a variable in YETI, having a specific type and name.
7. **YetiIdentifier:** It represents an identifier for an instance of a YetiCard.

3.2.4 Construction of Test Cases

YETI construct test cases by creating objects of the classes under test and randomly calls methods with random inputs according to the parameter's-space. YETI splits input values into two types i.e. primitive data types and user defined classes. For primitive data types as methods parameters, YETI in random strategy calls $Math.random()$ method to generate arithmetic values are converted to the required type using Java cast operation. In the case of user-defined classes as a parameter YETI calls constructor or method to generate object of the class at run time. The constructor may possibly require another object, then YETI recursively calls the constructor or method of that object. This process is continued till an object with empty constructor or a constructor with only primitive types or the set level of recursion is reached.

3.2.5 Call sequence of YETI

3.2.6 Command-line Options

YETI is provided with several command line options which a tester can enable or disable according to the test requirement. These options are case insensitive and can be provided in any order as input to YETI from command line interface. As an example, a tester can use command line option $-nologs$ to bypass real time logging and save processing power

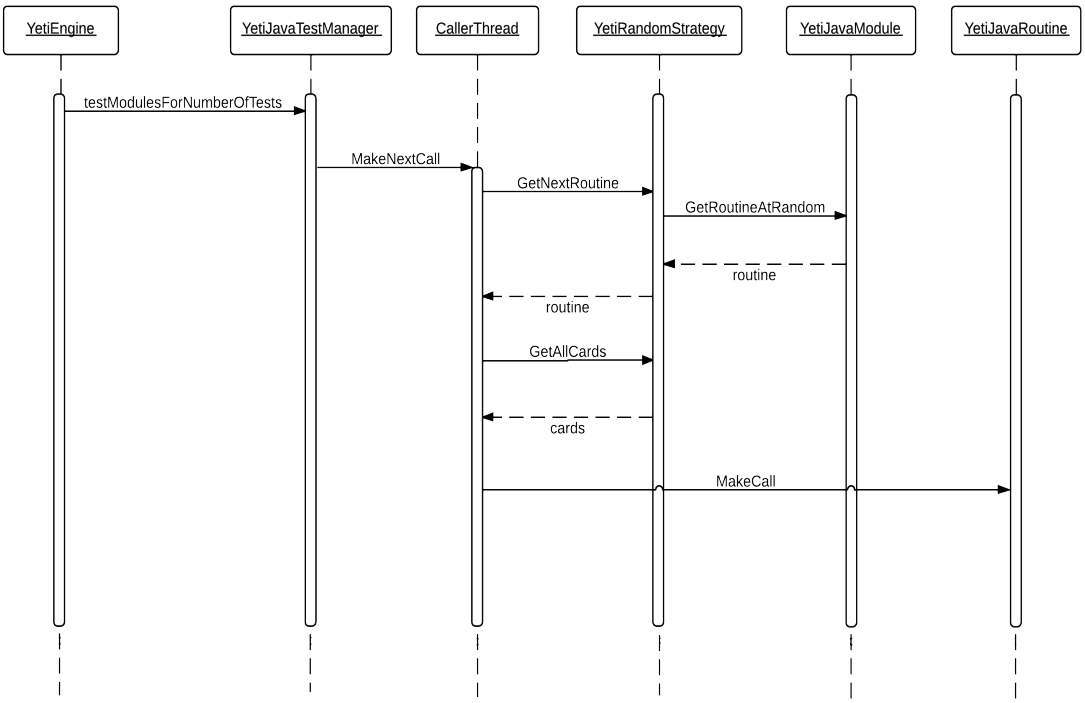


Figure 3.3: Call sequence of YETI with Java binding

by reducing overhead. Table 3.1 includes some of the common command line options available in YETI.

Table 3.1: YETI command line options

Options	Purpose
-java, -Java	To test Java programs
-jml , -JML	To test JML programs
-dotnet, -DOTNET	To test .NET programs
-ea	To check code assertions
-nTests	To specify number of tests
-time	To specify test time
-initClass	To use a user defined class to initialize the system
-msCalltimeout	To set a time out for a method call
-testModules	To specify one or more modules to test
-rawlogs	To print real time test logs
-nologs	To omit real time logs and print end result only
-yetiPath	To specify path to the test modules
-gui	To show test session in GUI
-help, -h	To print the help about using YETI
-DSSR	To specify Dirt Spot Sweeping Random strategy
-ADFD	To specify Automated Discovery of Failure Domain strategy
-ADFDPlus	To specify Automated Discovery of Failure Domain plus strategy
-noInstanceCap	To remove cap on the maximum number of instances of specific type
-branchCoverage	To measure the branch coverage
-tracesOutputFile	To specify the file to store output traces
-tracesInputFile	To specify the file to input traces
-random	To specify Random test strategy
-printNumberOfCallsPerMethod	To print the number of calls per method
-randomPlus	To specify Random plus test strategy
-probabilityToUseNullValue	To specify probability of inserting null values
-randomPlusPeriodic	To specify Random plus periodic test strategy
-newInstanceInjectionProability	To specify probability of inserting new objects

3.2.7 Execution

YETI, developed in Java, is highly portable and can easily run on any operating system with Java Virtual Machine (JVM) installed. It can be executed from both CLI and GUI. To execute YETI, it is necessary to specify the $project$ and the relevant jar library files, particularly $javassist.jar$ in the $CLASSPATH$. The typical command to execute YETI from CLI is given in Figure 3.4.

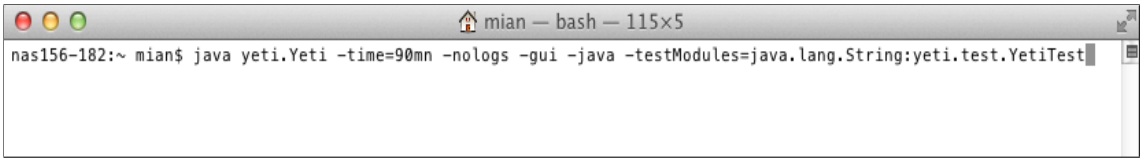


Figure 3.4: Command to launch YETI from CLI

In this command YETI tests java.lang.String and yeti.test.YetiTest modules for 90 minutes using the default random strategy. Other CLI options are already indicated in Table 3.1. To execute YETI from GUI, $YetiLauncher$ presented in Figure 3.5 has been created for use in the present study.

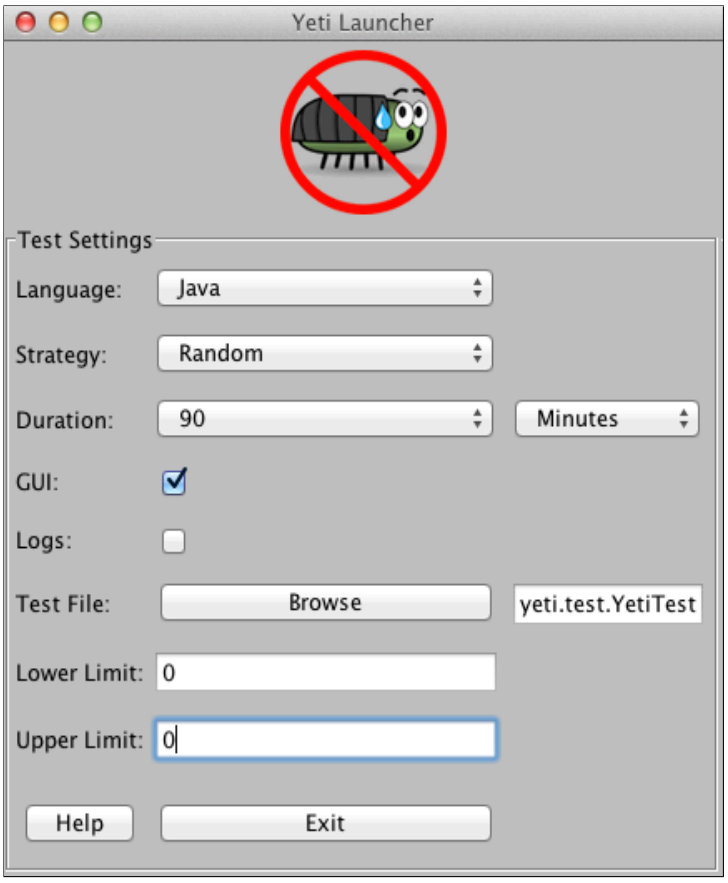


Figure 3.5: GUI launcher of YETI

3.2.8 Test Oracle

YETI uses one of the two approaches for oracle. In the presence of program specifications, it checks for inconsistencies between the code and the specifications. In the absence of specifications it checks for assertion violation.If specifications or assertions are absent, YETI performs robustness testing which considers any undeclared runtime exceptions as

failures.

3.2.9 Report

YETI gives a complete test report at the end of each test session. The report contains 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 assigned values 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 3.6: Successful method calls of YETI

YETI separates the found bugs from successful executions to simplify the test report. This helps debuggers to easily track the origin of the problem rectification. When a bug is identified during testing, YETI saves the details and present it in the bug report as shown in Figure 3.7. The information includes all identifiers of the parameters the method call had along with 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 3.7: A sample of YETI bug report

3.2.10 Graphical User Interface

YETI supports a GUI that allows testers to monitor the test session and modify the characteristics in real time during test execution. It is useful to have the option of modifying the test parameters at run time and observing the test behaviour in response. Figure 3.8 present the YETI GUI comprising of thirteen labelled components.

1. **Menu bar:** It contains two menu items i.e. Yeti and File.
   (a) **Yeti menu:** It provides information about YETI contributors and the option to quit the GUI.
   (b) **File menu:** It provides the option to rerun the previously executed scripts.
2. **Slider of % null values:** It displays the set probability of null values in percentage used as null instance for each variable. The default value of probability is 1.
3. **Slider of % new variables:** It displays the set probability of creating new instances at each call. The default value of probability is 1.
4. **Text-box of Max variables per type:** It displays the number of variables created for a given type. The default value is 1000.
5. **Progress bar of testing session:** It displays the test progress in percentage.
6. **Slider of strategy:** It displays the set random strategy for the test session. Each strategy has its own control to change its various parameters.
7. **Module Name:** It 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.
8. **Graph window 1:** It displays the total number of unique failures over time in the module under test.
9. **Graph window 2:** It displays the total number of calls over time to the module under test.
10. **Routine's progress:** It displays test progress of each routine in the module represented by four colours. Mostly green and red colour appears indicating successful and unsuccessful calls respectively. Occasionally black and yellow colours appear indicating no calls and incomplete calls respectively.
11. **Graph window 3:** It displays the total number of failures over time in the module under test.
12. **Graph window 4:** It displays the total number of variables over time generated by YETI in the test session.

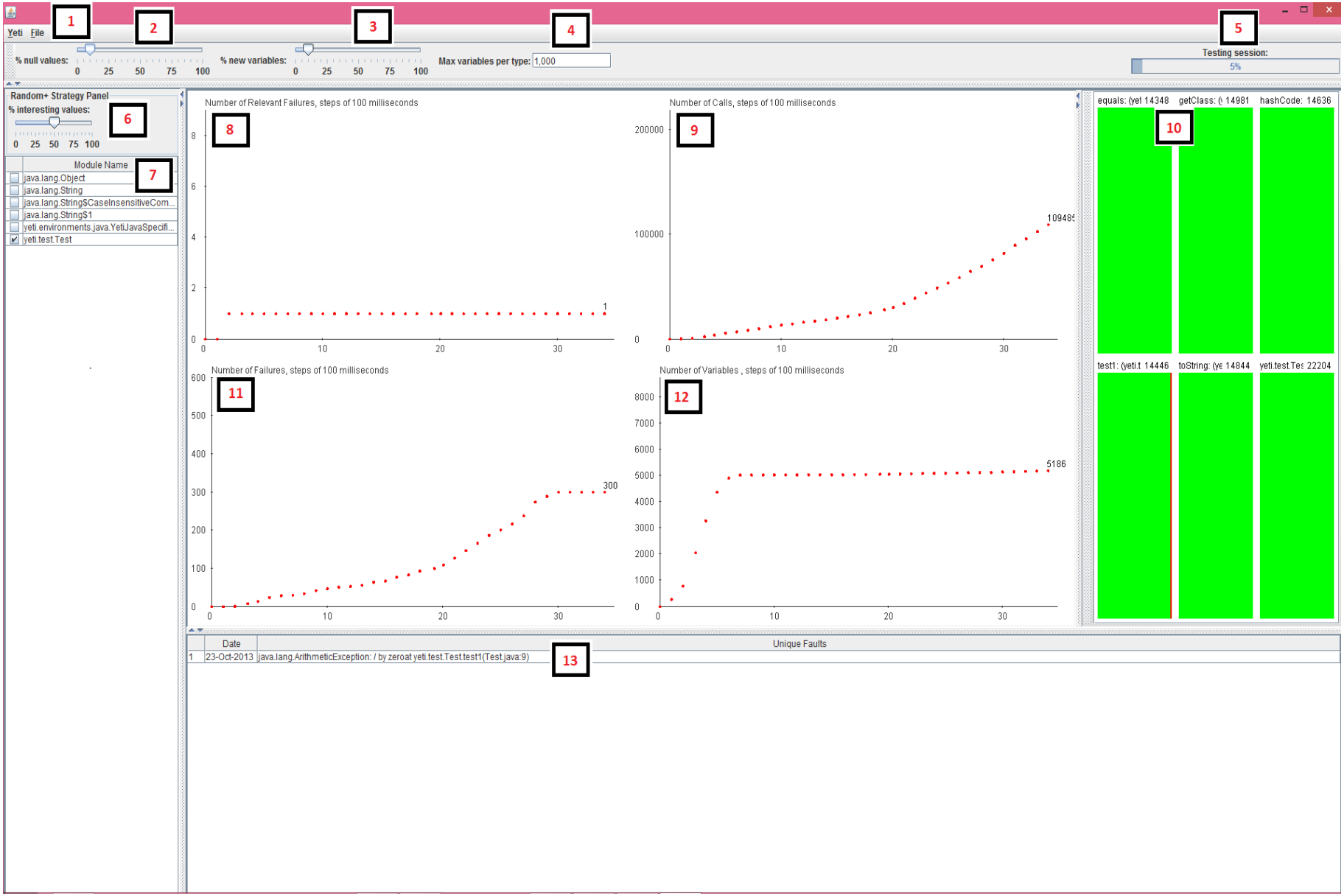


Figure 3.8: GUI front-end of YETI

13. **Report section:** It displays the number of unique failure by date and time, location and type detected in the module under test.

3.3 Summary

The chapter explains in detail the tool YETI which is being used in this study. YETI has been thoroughly reviewed including an overview, design, core infrastructure, strategy, language-specific binding, construction of test cases, command line options, execution, test oracle, report generation and graphical user interface.

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 input domain, limited resources and strict time constraints. Therefore, strategies in automated software testing tools are developed with the aim to select more failure-finding test inputs from the input domain. Producing such targeted test input is difficult because each system has its own requirements and functionality.

Chan et al. [1] discovered that there are domains of failure-causing inputs across the input domain. They divided these into point, block and strip domains on the basis of their occurrence across the input domain. Chen et al. [71] 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 increases up to 50% when test inputs are 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 domains. Similarly Restricted Random Testing [58], Feedback-Directed Random Testing [74], Mirror Adaptive Random Testing [72] and Quasi Random Testing [73] stress the need for test case selection covering the whole input domain to get better results.

In this chapter we assume that failure domains are contiguous for a significant number of classes. Based on this assumption, we devised the Dirt Spot Sweeping Random (DSSR) strategy which starts as a random+ (R+) strategy focusing more on boundary values. When a new failure is found, it increases the chances of finding more failures by DSSR strategy using the neighbouring values. Since this strategy is an extension of random (R) strategy,

it has the full potential to find all failures in the program, but additionally we expect it to be faster at finding failures, for classes in which failure domains are contiguous, as compared with R and R+ strategies.

We implemented the DSSR strategy in York Extensible Testing Infrastructure (YETI) http://www.yetitest.org. To evaluate our approach, we tested 30 times each one of the 60 classes of 32 different projects from the Qualitas Corpus http://www.qualitascorpus.com with each of the three strategies R, R+ and DSSR. We observed that for 53% of the classes all three strategies perform equally, for remaining 47% classes, DSSR strategy performed up to 33% better than R and up to 17% better than R+ strategy.

We also validated the approach by comparing the significance of these results using t test 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 classes, DSSR, R+ and R showed no significant difference statistically. Numerically however, 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 R+ strategy with a dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, failures reside more frequently in contiguous blocks and strip domain. If this is true, the Dirt Spot Sweeping (DSS) feature of the strategy will increase the performance of the test. Before presenting the details of the DSSR strategy, it is pertinent to review briefly the R and the R+ strategy.

4.2.1 Random Strategy

The random strategy is a black-box 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 nature, this strategy is particularly effective in testing software where the developers want to keep the source code secret [91]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [43, 92]. It is mainly for this reason that various researchers have recommended

R strategy for automated testing tools [60]. YETI [90], AutoTest [4, 15], QuickCheck [81], Randoop [93] and Jartege [79] are some of the most common automated testing tools based on R strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [10] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [15, 59, 64, 94, 95] 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 [15, 96] called a unique failure. This way of grouping failures is also used for R+ and DSSR.

4.2.2 Random+ Strategy

The random+ strategy [4] is an extension of the R strategy. It uses some special predefined values which can be boundary values or values that have high tendency of finding failures in the SUT. Boundary values [70] 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 tries to build an array of negative size.

Similarly, the tester might also add some other special values that he considers effective in finding failures 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 high priority than selection of random values because of more relevance and high chances of finding failures for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [43].

4.2.3 Dirt Spot Sweeping

Chan et al. [1] found that there are domains of failure-causing inputs across the input domain. Section 2.9 shows these domains for two-dimensional input domain. They divided these domains into three types called point, block and strip domains. The black area inside the box in the form of points, block and strip shows 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 failure domains if test cases far away from each other are selected. Other researchers [72, 58, 73], also tried to generate test cases further away from one another targeting these domains and achieved better performance. The increase in performance indicates that failures more often occur contiguously across the input domain. When test value reveals a failure in a program then DSS may not look farthest away for the selection of next test value but picks the closest test values for the next several tests to find another failure from the same region.

Dirt spot sweeping is the feature 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 domain, then adding it's neighbouring values will explore other failures present in the block or strip. In DSSR strategy the list of interesting values is dynamic and changes during the test execution of each program. While in R+ strategy, the list of interesting values remain static and manually changed before the start of each test.

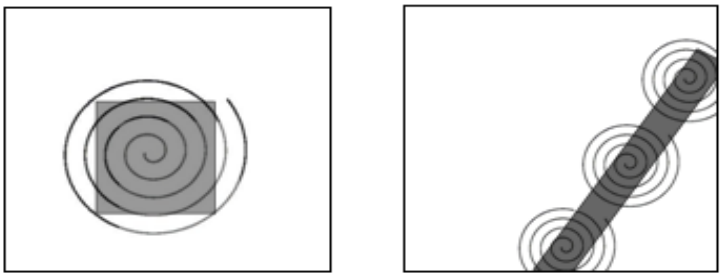


Figure 4.1: Exploration of failures by DSS in block and strip domain

Figure 4.1 shows how DSS explores the failures residing in the block and strip domains of a program. The coverage of block and strip domain 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 domain then the added values may not be effective because point domain is only an arbitrary failure point in the whole input domain.

4.2.4 Working of DSSR 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 [44]. 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 dynamic list of interesting values.

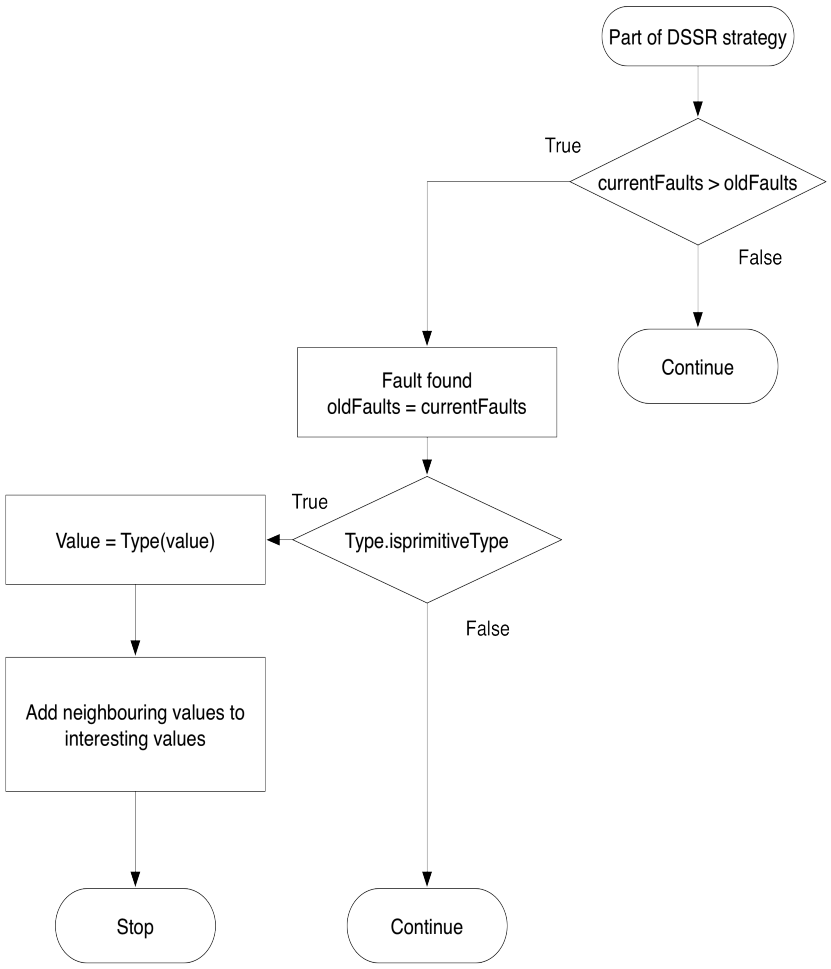


Figure 4.2: Working mechanism of DSSR Strategy

The flowchart presented in Figure 4.2 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 provides 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 having a high tendency of finding failures in the SUT are added to the list of interesting values by R+ strategy prior to the start of test session where as in DSSR strategy the failure-finding and its surrounding values are added at runtime when a failure is found.

Table 4.1 presents the values 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 a primitive type, string or user defined objects. All values are converted to their respective types before adding them to the list of interesting values.

Table 4.1: Data types and Corresponding values to be added

Data Type	Values to be added
X is int, double, float, long, byte, short & char	X, X+1, X+2, X+3, X-1, X-2, X-3
X is String	X X + " " " " + X X.toUpperCase() X.toLowerCase() X.trim() X.substring(2) X.substring(1, X.length()-1)
X is object of user defined class	Call its constructor recursively until empty or primitive values

4.2.5 Explanation of DSSR Strategy by Example

The DSSR strategy is explained through a simple program seeded with three fault. 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:

```
/**
* 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 failures are quickly discovered by DSSR strategy in the following order.

Failure 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 violation of assertion (1) and Java to generate division by zero exception.

Failure 2: After discovering the first failure, 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 failure where assertion (2) fails because the square root of `9` is `3` instead of the input value -3.

Failure 3: After a few tests the strategy may select `Integer.MAX_VALUE` for variable `num1` from the list of interesting values leading to the discovery of 3rd failure 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, failure-finding and surrounding values to the list of interesting values in DSSR strategy leads to the available failures quickly and in fewer tests as compared to R and R+ strategy. R and R+ takes more number of tests and time to discover the second and third failures because in these strategies the search for new unique failures starts again randomly in spite of the fact that the remaining failures lie in close proximity to the first one.

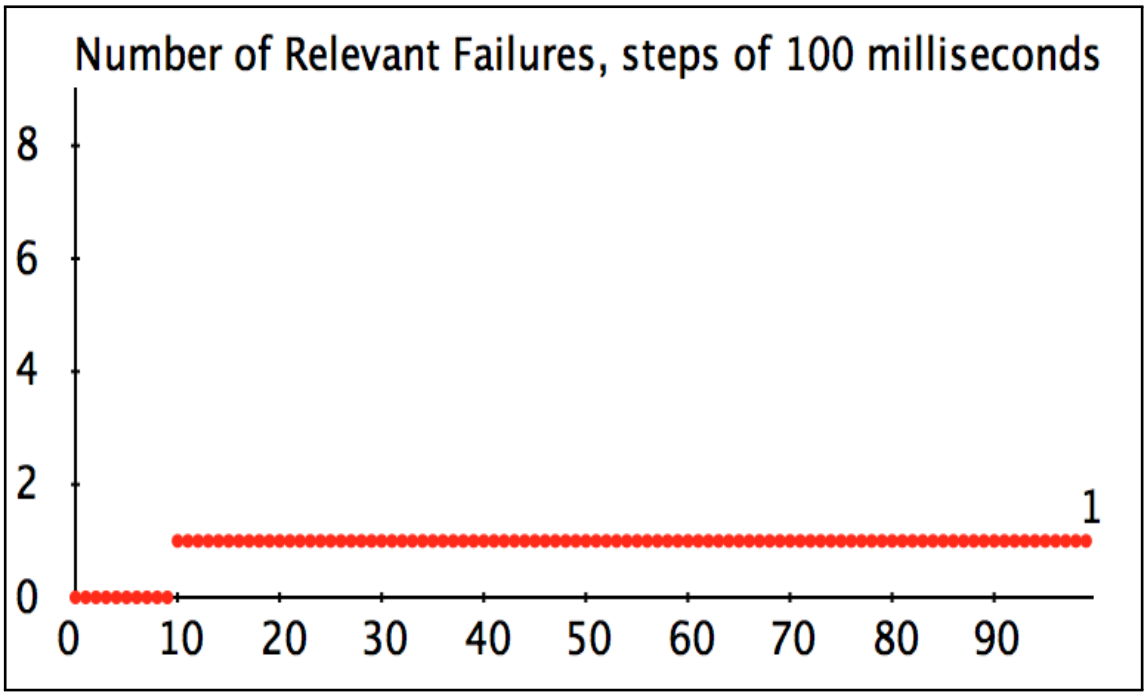


Figure 4.3: Test result of random strategy for the example code

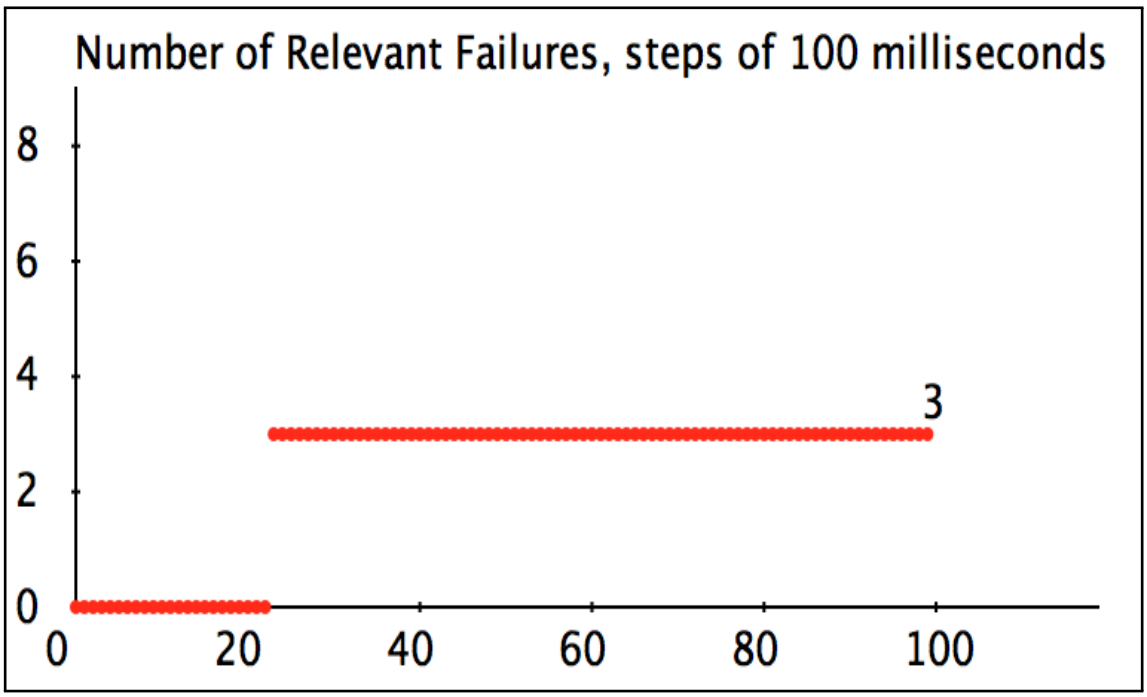


Figure 4.4: Test result of DSSR strategy for the example code

4.3 Implementation of DSSR Strategy

The DSSR strategy is implemented in 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 the cloud [90]. A number of hitherto failures have successfully been found by YETI in various production software [96].

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 R 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 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. The class hierarchy is shown in Figure 4.5.

4.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of R and R+ strategy [4]. 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:

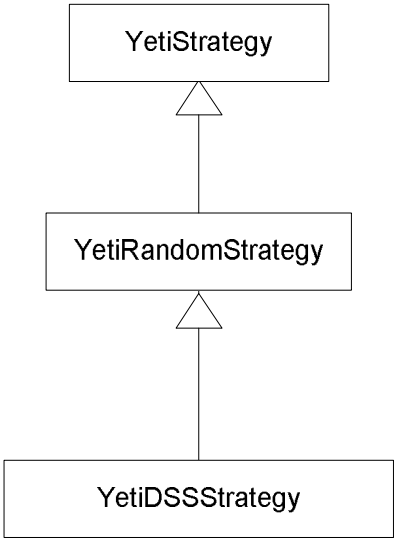


Figure 4.5: Class Hierarchy of DSSR in YETI

1. Is there an absolute best amongst 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 amongst R, R+ and DSSR?

4.4.2 Experiments

We performed extensive testing of programs from the Qualitas Corpus [97]. The Qualitas Corpus is a curated collection of open source Java projects built with the aim of helping empirical research in the field of software engineering. These projects have been collected in an organised form containing the source and binary forms. The Qualitas Corpus [version 20101126] containing 106 open source Java projects was used in the current evaluation. In our experiments we randomly selected 60 classes from 32 projects taken at random. All the selected classes produced at least one failure and did not time out with maximum testing session of 10 minutes. Every class was 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 are presented in Table 4.3. Line of Code (LOC) tested per class and the total LOC's tested are shown in column 3 of Table 4.3.

Every class is evaluated through 10^5 calls in each test session. The approach similar to that used in previous studies when the contracts and assertions in the code under test are absent was followed in the study [96]. The undeclared exceptions were treated as failures.

Table 4.2: Specifications of projects randomly selected from Qualitas Corpus

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 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 failure) have been reported in the literature for finding the effectiveness of R strategy. The E-measure and P-measure have been criticised [71] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [98, 99]. In our initial experiments, the F-measure was used to evaluate the efficiency of test strategy. However it was later realised that this was not the right choice. In some experiments a strategy found the first failure quickly than the other but on completion of test session that very strategy found lower number of total failures than the rival strategy. The preference given to a strategy by F-measure because it finds the first failure quickly without giving due consideration to the total number of failures is not fair [100].

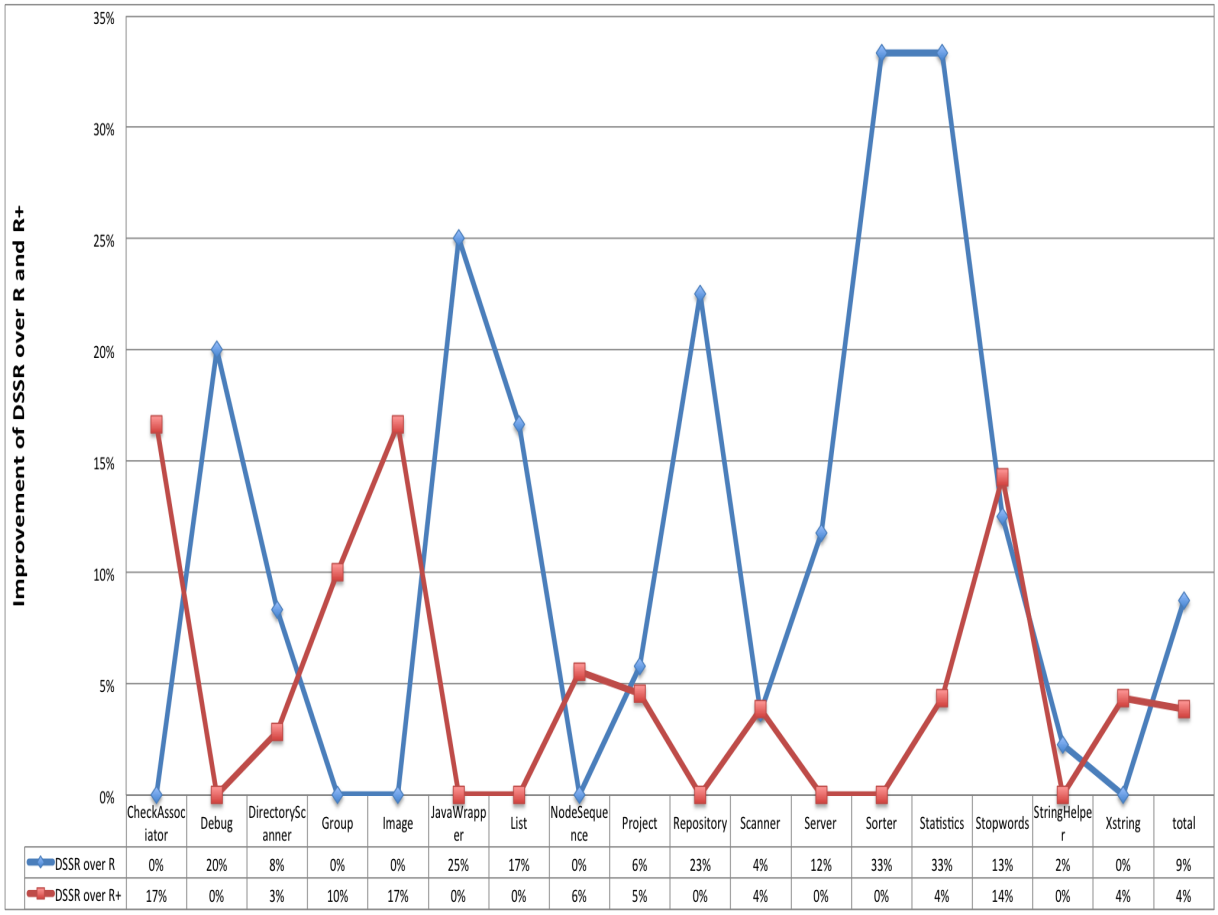
The literature review revealed that the F-measure is used where testing stops after identification of the first failure and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered failures in one go and F-measure is not the favourable choice. In our experiments, performance of the strategy was measured by the maximum number of failures detected in SUT by a particular number of test calls [15, 93, 101]. This measurement was effective because it considers the performance of the strategy when all other factors are kept constant.

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 failures 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 was higher in DSSR (1075) as compared to R (1040) and R+ (1061) strategies. DSSR found higher number of maximum unique failures (1118) than R (1075), and R+ (1106). DSSR found 43 and 12 more unique failures compared to R and R+ strategies respectively. The minimum number of unique failures found by DSSR (1032) is also higher

Table 4.3: Comparative performance of R, R+ and 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



	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 4.6: Performance of DSSR in comparison with R and R+ strategies.

than for R (973) and R+ (1009) which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

4.5.1 Absolute best in R, R+ & DSSR strategies

Based on our findings DSSR is at least as good as R and R+ in almost all cases, it is significantly better than both R and R+ in 12% of the classes. Figure 4.6 presents the performance of DSSR in comparison with R and R+ strategies in 17 classes showing significant difference. 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 improvement of DSSR over R and R+ strategy is calculated by applying the formula (1) and (2) respectively.

$$\frac{Average failures_{(DSSR)} - Average failures_{(R)}}{Average failures_{(R)}} * 100 \quad (4.1)$$

$$\frac{Average failures_{(DSSR)} - Average failures_{(R+)}}{Average failures_{(R+)}} * 100 \quad (4.2)$$

The DSSR strategy performed up to 33% better than R and up to 17% better than R+ strategy. In some cases DSSR performed equally well with R and R+ but in no case DSSR performed lower than R and R+ strategies. Based on the results it can be stated that on overall basis DSSR strategy performed better than R and R+ strategies.

4.5.2 Classes for which any of the three strategies perform better

T test applied to data given in Table 4.4 indicated significantly better performance of DSSR in 7 classes from both R and R+ strategies, in 8 classes from R strategy and in 2 classes from R+ strategy. In no class R and R+ strategies performed significantly better than DSSR strategy. Expressed in percentage, 72% of the classes showed statistically no significant difference whereas in 28% of the classes, the DSSR strategy performed significantly better than either R or R+. The better performance of DSSR may be attributed to the additional feature of spot sweeping over and above the desirable characteristics present in R+ strategy.

4.5.3 The best default strategy in R, R+ & DSSR

Analysis of the experimental data revealed that DSSR strategy had an edge over R and R+. This is due to the additional feature of spot sweeping in DSSR strategy. In spite of the better performance of DSSR as compared to R and R+ strategies, the present study does not provide ample evidence to pick it as the best default strategy. This is primarily due to the overhead induced by DSSR strategy, discussed in Section 4.6. Further study might provide some conclusive findings.

4.6 Discussion

In this section we discuss various factors affecting the results of DSSR, R and R+ strategies including time taken, test duration, number of tests, number of failures, identification of first failure, level of coverage and threats to validity.

Time taken by the strategies to execute equal number of test cases: The DSSR strategy took slightly more time (up to 5%) than both R and R+ strategies which might be due

Table 4.4: Results of t test applied on experimental

S. No	Class Name	t test Results			Interpretation
		DSSR - R	DSSR - R+	R - R+	
1	AjTypeImpl	1	1	1	Difference not significant
2	Apriori	**0.03**	0.49	0.16	Difference not significant
3	CheckAssociator	**0.04**	**0.05**	0.44	DSSR $>$ R & R+
4	Debug	**0.03**	0.14	0.56	Difference not significant
5	DirectoryScanner	**0.04**	**0.01**	0.43	DSSR $>$ R & R+
6	DomParser	**0.05**	0.23	0.13	Difference not significant
7	EntityDecoder	**0.04**	0.28	0.3	Difference not significant
8	Font	0.18	0.18	1	Difference not significant
9	Group	0.33	**0.03**	**0.04**	DSSR = R $>$R+
10	Image	**0.03**	**0.01**	0.61	DSSR $>$ R & R+
11	InstrumentTask	0.16	0.33	0.57	Difference not significant
12	JavaWrapper	**0.001**	0.57	0.004	DSSR = R+ $>$R
13	JmxUtilities	0.13	0.71	0.08	Difference not significant
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	Difference not significant
17	Project	**0.001**	0.57	**0.004**	DSSR $>$ R & R+
18	Repository	**0**	1	**0**	DSSR = R+ $>$R
19	Scanner	1	**0.03**	**0.01**	DSSR $>$ R & R+
20	Scene	**0**	**0**	1	DSSR $>$ R & R+
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 $>$ R & R+
27	WebMacro	0.33	1	0.16	Difference not significant
28	XMLEntityManager	0.33	0.33	0.16	Difference not significant
29	XString	0.14	**0.03**	0.86	Difference not significant

to the feature of maintaining sets of interesting values during the execution.

Effect of test duration and number of tests on the results: If testing is continued for a long duration and sufficiently large number of tests are executed, in that case all three strategies might find the same number of unique failures. However for the same number of test cases, DSSR performed significantly better than R and R+ strategies. Further experiments are desirable to determine the comparative performance of the three strategies with respect to test duration and number of tests.

Effect of number of failures on the results: The DSSR strategy performed better when the number of failures was higher in the code. The reason might be that in case of more failures, the failure domains are more connected thus DSSR strategy might work better.

Effect of identification of first failure on the results: During the experiments, It was noticed that quick identification of first failure was highly desirable in achieving better results from DSSR strategy. This was due to the feature of DSS which added the failure finding and surrounding values to the list of interesting values. However, when identification of first failure was delayed, no values were added to the list of interesting values and the DSSR performed equivalent to R+ strategy. This indicated that better ways of populating failure-inducing values were 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 failure right away.

Level of coverage: Random strategies typically achieve low level of coverage [90] and DSSR might be no exception. However it might be interesting to compare DSSR with R and R+ with respect to the achieved coverage.

Threats to validity: As usual with empirical studies, the present work might also suffer from a non-representative selection of classes. However selection in the study was made through random process and objective criteria to make it more representative.

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 [2, 3, 81]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [59]. Some of the well known automated tools based on R strategy includes JCrasher [2], Eclat [3], AutoTest [15, 60], Jartege [79] and YETI [90, 96].

In pursuit of better test results and lower overhead, many variations of R strategy have been proposed [72, 73, 91, 102, 103]. 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. ART through dynamic partitioning and MART are the two strategies developed to improve the performance of ART by reducing the overhead. This was achieved mainly by the even spread of test input to increase the chance of exploring the failure domains present in the input domain.

A more recent research study [104] 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 R+ strategy is an extension of the R strategy in which interesting values, beside pure random values, are added to the list of test inputs [4]. These interesting values include border values which have high tendency of finding failures in the given SUT [70]. Results obtained with R+ strategy showed significant improvement over R strategy [4]. DSSR strategy is an extension of R+ strategy which starts testing as R+ until a failure is found and then switches to dirt spot sweeping.

A common practice to evaluate performance of an extended strategy is to compare the results obtained by applying the new and existing strategy to identical programs [35, 94, 105]. Arcuri et al. [106], 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 [107] 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 [96, 97, 108].

4.8 Summary

The main goal of the present study was to develop a new random strategy which could find more failures in a lower number of test cases. We developed the "DSSR strategy" as an extension of R+, based on the assumption that in a significant number of classes, failure domains are contiguous. The DSS feature of DSSR strategy adds neighbouring values of the failure finding value to the list of interesting values. The strategy was implemented in the random testing tool YETI to test 60 classes from Qualitas Corpus, 30 times each with each of the three strategies i.e. R, R+ and DSSR. The newly developed DSSR strategy uncovered more unique failures than both R and R+ 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 significantly better than R, while in 2 (3%) classes DSSR performed significantly better than R+. In all other cases, performance of DSSR, R and R+ showed no significant difference. On overall basis, DSSR produced encouraging results.

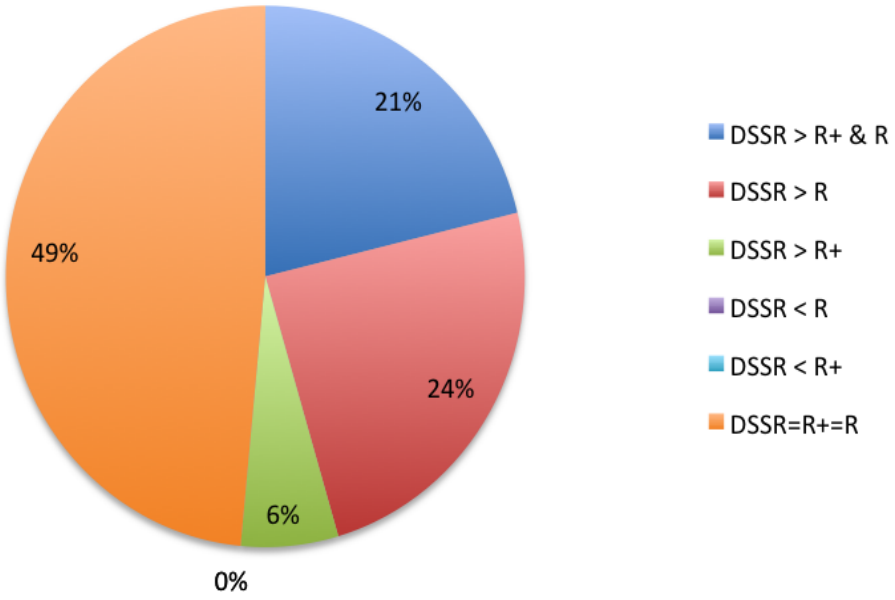


Figure 4.7: Results of DSSR strategy in comparison to Random and Random+

Chapter 5

Automated Discovery of Failure Domain

5.1 Introduction

Testing is a fundamental requirement for assessing the quality of any software. Manual testing is labour-intensive and error-prone; therefore automated testing is often used which significantly reduces the cost of software development and its maintenance [33]. Most of the modern black-box testing techniques execute the SUT with specific input and results obtained are compared against the test oracle. A report is generated at the end of each test session depicting any discovered faults and the input values which triggers the failures. Developers 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 already set in test plan is achieved.

The fact that exhaustive testing for any non-trivial program is not possible, compels the testers to come up with some strategy of input selection from the whole input domain. Random is one of the possible strategies widely used in automated tools. It is intuitively simple and easy to implement [43, 67]. It involves minimum or no overhead in input selection and lacks human bias [59, 109]. Random testing has several benefits but there are some limitations as well, including low code coverage [110] and discovery of lower number of faults [111]. To overcome these limitations and retain the benefits intact many researchers have successfully refined random testing. Adaptive Random Testing (ART) is one of the most significant refined version of of random testing. Experiments performed using ART showed up to 50% better results compared to the traditional random testing [71]. Similarly Restricted Random Testing (RRT) [102], Mirror Adaptive Random Testing (MART) [72], Adaptive random testing for object oriented program (Artoo) [60], Directed Adaptive

Random Testing (DART) [17], Lattice-based Adaptive Random Testing (LART) [112] and Feedback-directed Random Testing (FDRT) [74, 93] are some of the improved versions of random testing.

All the above-mentioned versions of random testing are based on the observation of Chan et al. [1] that failure causing inputs across the whole input domain form certain kinds of domains. They classified these domains into point, block & strip failure domain and suggested that the failure finding ability of testing could be improved by taking into consideration these failure domains. In Section 2.9 the square boxes represent the whole input domain. The black points, block and strip inside the boxes represent the faulty values while the white area inside the boxes represent legitimate values for a specific system.

It is interesting that where many random strategies are based on the principle of contiguous failure domains inside the input domain, no specific strategy has been developed to evaluate these failure 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 chapter include:

- Implementation of the new ADFD strategy in York Extensible Testing Infrastructure (YETI).
- Evaluation to assess ADFD strategy by testing classes with different failure domains.
- Reduction in test duration by identification of all failure domains instead of a single instance of failure.
- Improvement in test efficiency by helping debugger to consider all failure occurrences during 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 failures as well as the failure 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 work flow of ADFD strategy is given in Figure 5.1.

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 failure
3. Automated generation of modules
4. Automated compilation and execution of modules to discover domains
5. Automated generation of graph showing domains

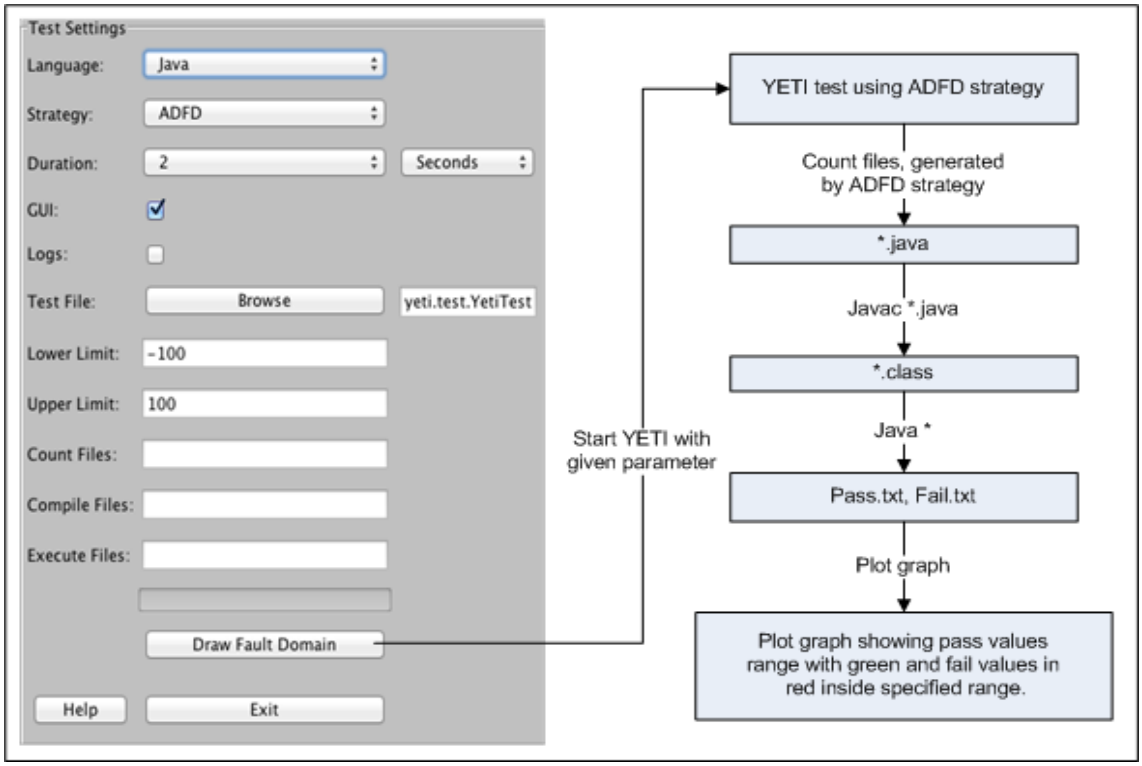


Figure 5.1: Work flow of ADFD strategy

5.2.1 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 program language, strategy, duration, enable or disable YETI GUI, logs and program in byte code. In addition, it also takes minimum and maximum values to search for failure domain in the specified range. Default range for minimum and maximum is taken as Integer.MIN_INT and Integer.MAX_INT respectively.

5.2.2 Automated Finding of Failure:

ADFD, being extended form of R+ strategy, relies on R+ to find the first failure. Random+ is an improvement on random strategy with preference for the boundary values to provide

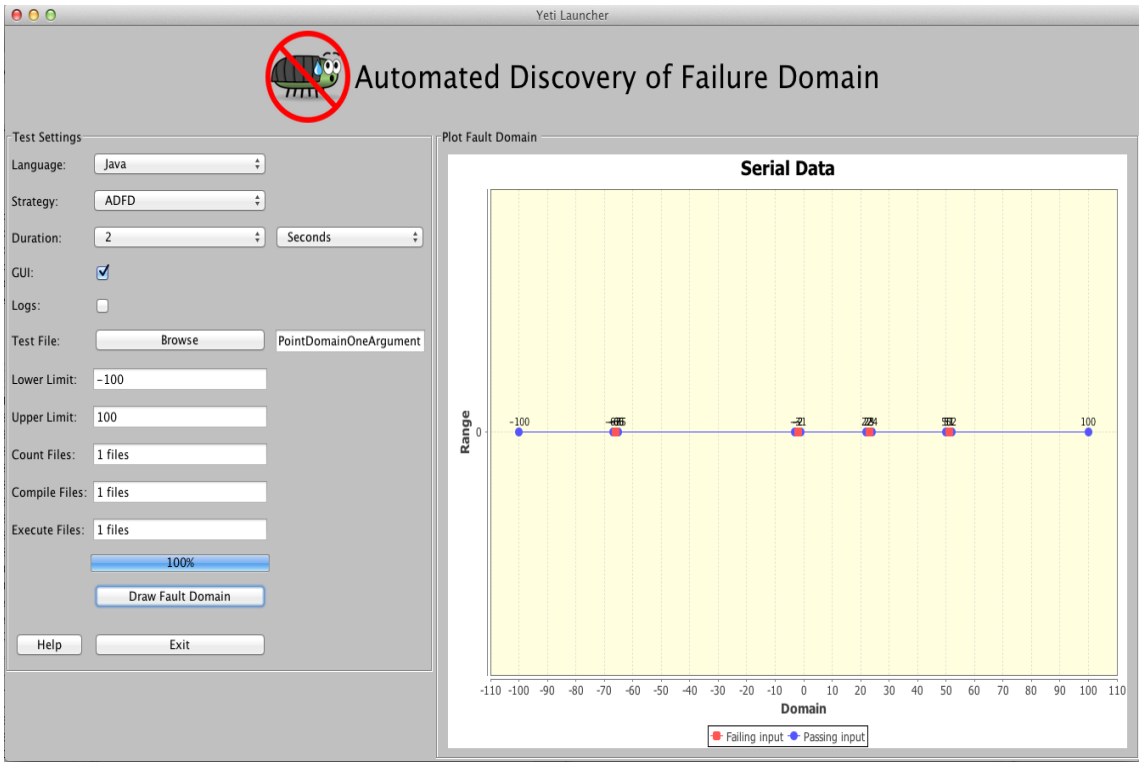


Figure 5.2: Front-end of ADFD strategy

better failure finding ability.

5.2.3 Automated Generation of Modules:

After a failure is found in the SUT, ADFD strategy generates complete new Java program to search for failure 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 module with two arguments two programs and so on. To track failure domain, the program keeps only one argument as variable and the remaining arguments as constant in the program generated at run time.

5.2.4 Automated Compilation and Execution of Modules:

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 receives all the values ranging, 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.

5.2.5 Automated Generation of Graph:

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 [113]. Lines and circles with blue colour represent pass values while lines and squares with red colour represents fail values. Resultant graph clearly depicts both the pass and fail domain across the specified input domain. The graph shows red points when the program fails for only one value, blocks when the program fails for multiple values and strips when the 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 3 for more details on YETI.

5.3.1 York Extensible Testing Infrastructure

YETI is a testing tool developed in Java that tests 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 failures found during the test session.

5.3.2 ADFD strategy in YETI

ADFD strategy is implemented in the strategies section of YETI. This section contains various strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default strategy for testing YETI 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.

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

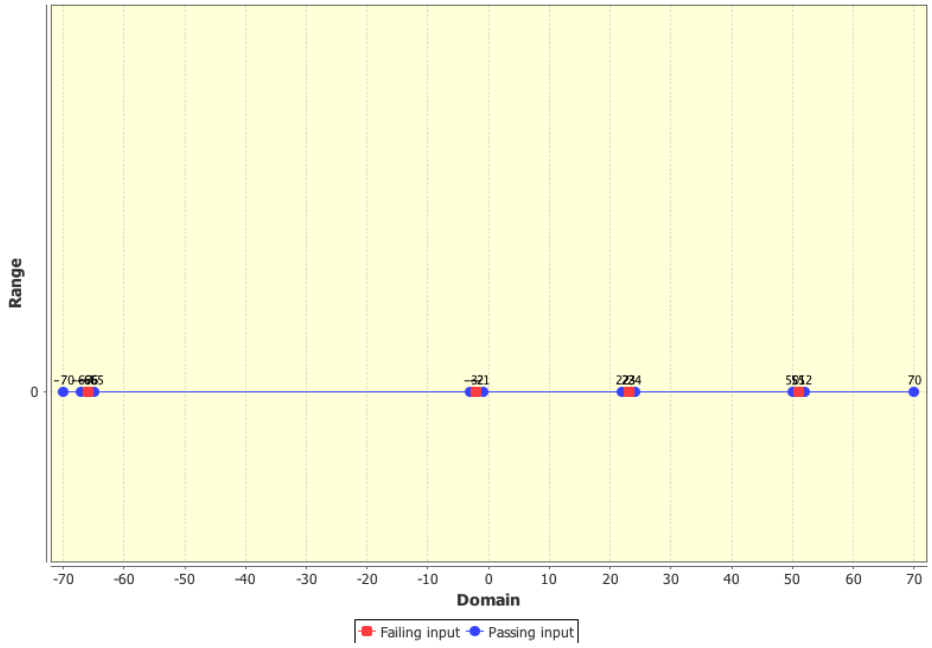


Figure 5.3: ADFD strategy plotting pass and fail domain of the given class

As soon as any one of the above four failures are discovered the ADFD strategy generates 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 5.3 that ADFD strategy not only finds all the failures but also plots the pass and fail domains.

5.4 Experimental Results

This section includes the experimental set-up and results obtained by 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 point, block and strip domains. Each selected program contained various combinations of one or more fault domains.

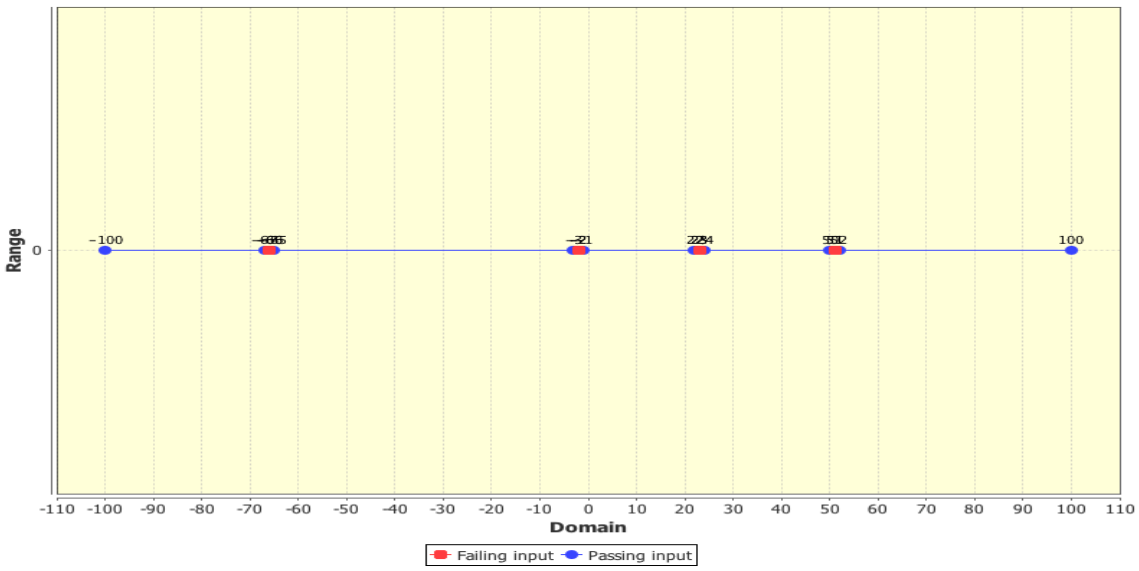
All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35].

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

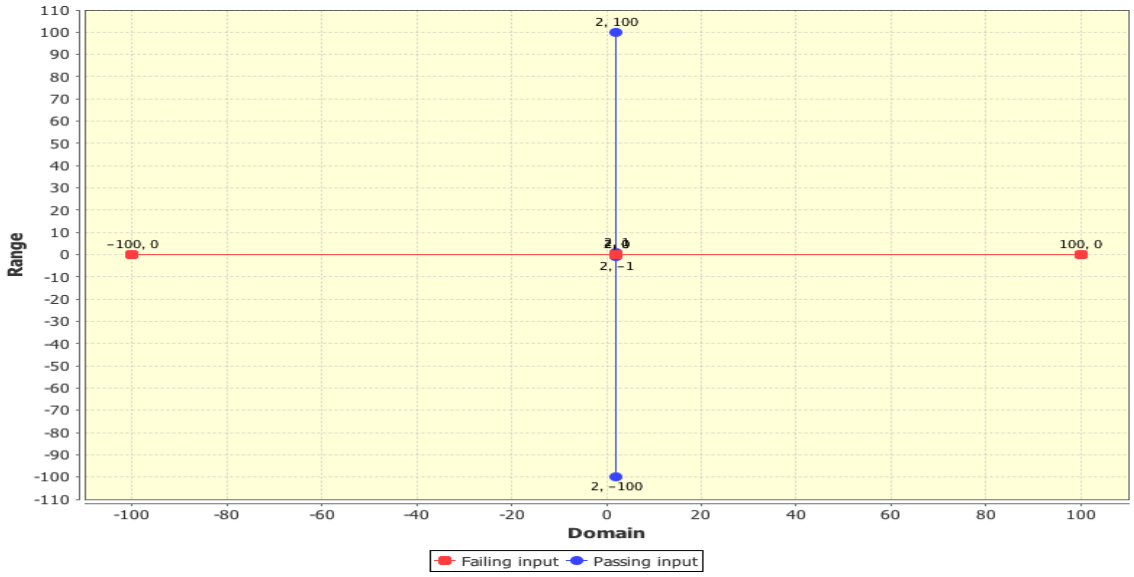
Table 5.1: Experimental results of programs tested with ADFD strategy

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 5.4(a) presents range of pass and fail values for point fault domain in one-dimension whereas Figure 5.4(b) presents 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 is given in Table 5.1.



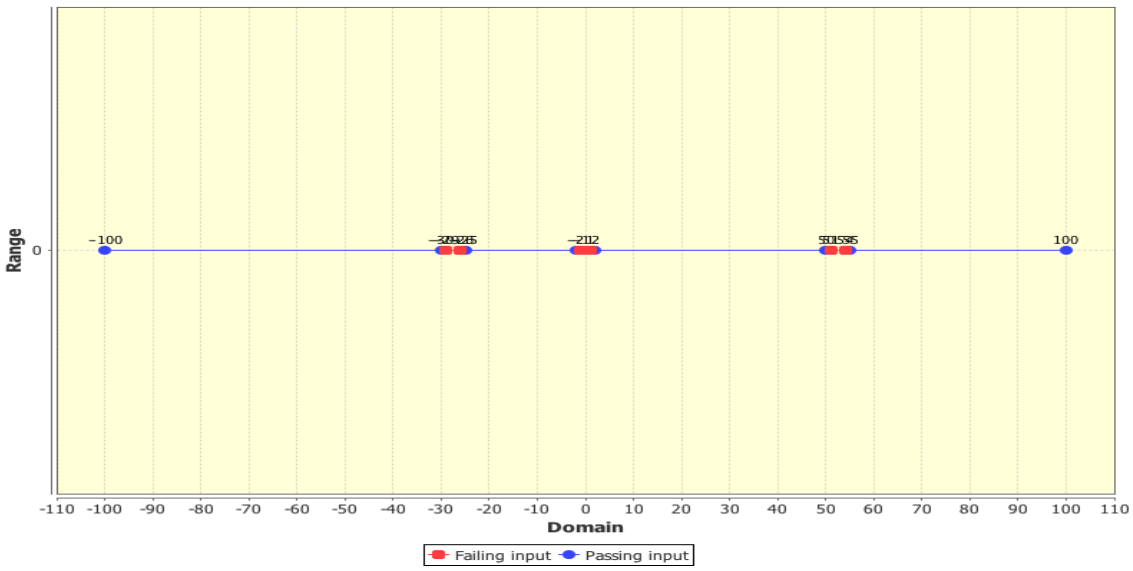
(a) One dimension module



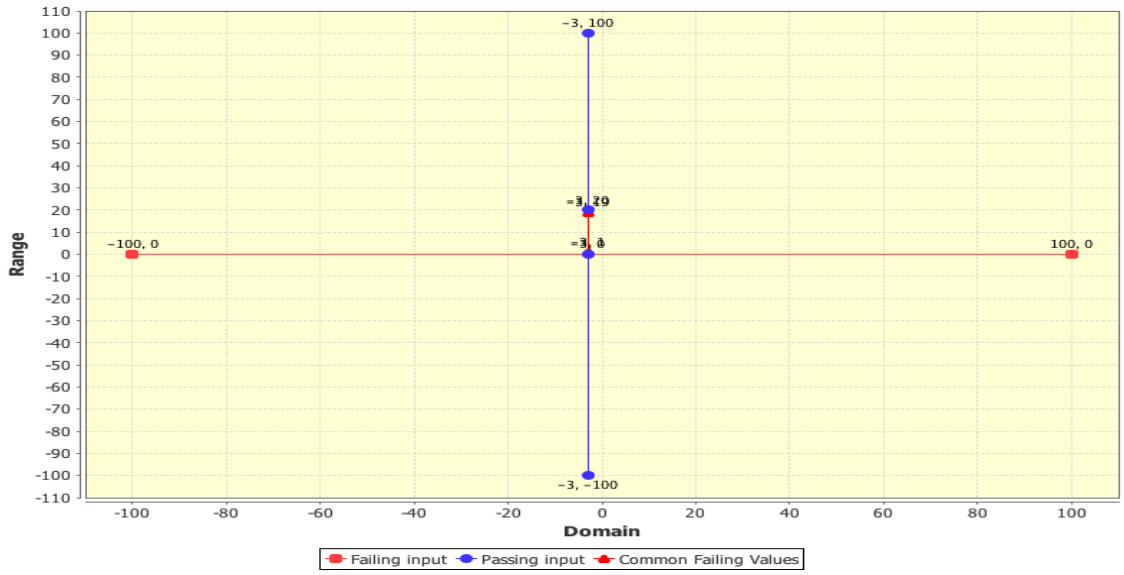
(b) Two dimension module

Figure 5.4: 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-dimensional program. Figure 5.5(a) presents range of pass and fail values for block fault domain in one-dimension whereas Figure 5.5(b) presents 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 is given in Table 5.1.



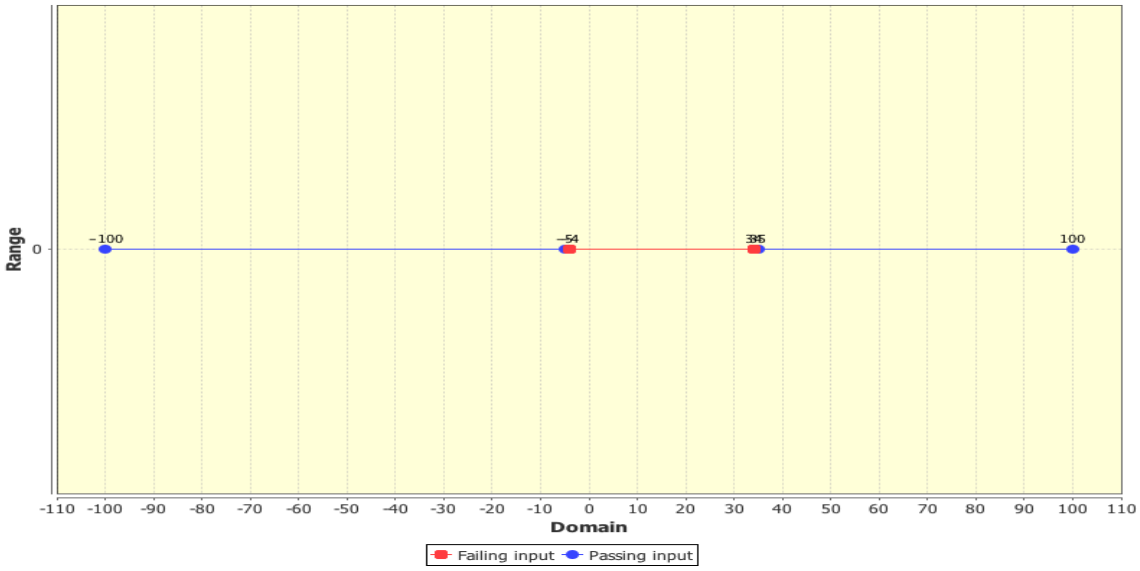
(a) One dimension module



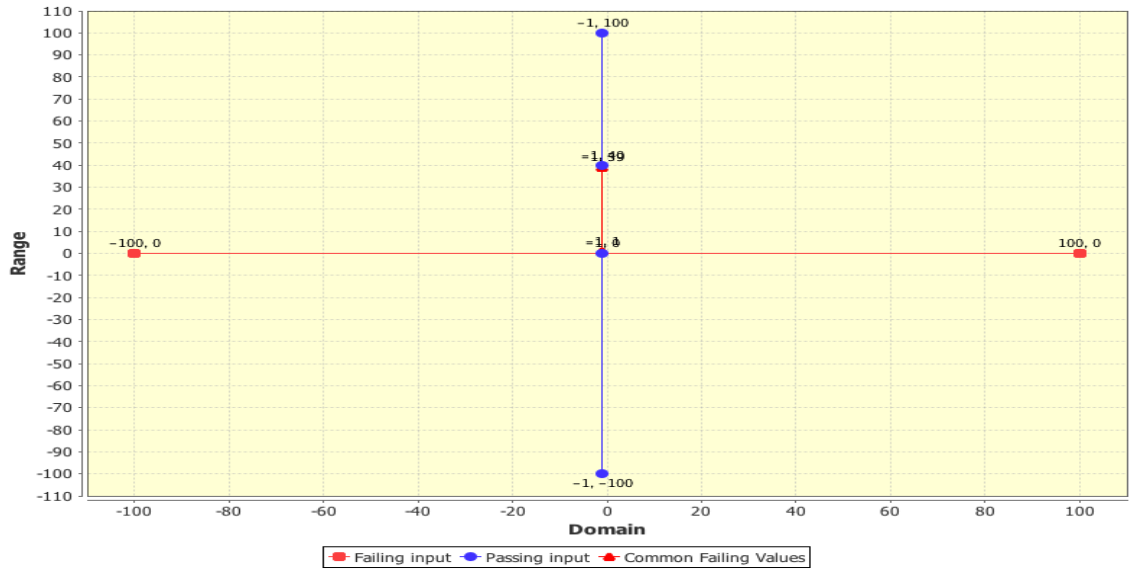
(b) Two dimension module

Figure 5.5: 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 5.6(a) presents range of pass and fail values for strip fault domain in one-dimension whereas Figure 5.6(b) presents 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 is given in Table 5.1.



(a) One dimension module



(b) Two dimension module

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

5.5 Discussion

ADFD, with a simple graphical user interface, is a fully automated testing strategy which identifies failures, failure domains and visually present the pass and fail domains in the form of chart. Since all the default settings are set to optimum level, 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 failure, 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 automatically without any human intervention.

As evident from the results, ADFD strategy effectively identified failures and failure domains in a program. Identification of failure domain is simple for one and two-dimensional numerical program but as the dimension increases the process gets more and more complicated. Moreover, 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 are defined.

ADFD strategy initiates testing with random+ strategy to find the failure and later switches to brute-force strategy to apply all the values between upper and lower bounds for finding pass and failure domains. It was found that failures at boundary of the input domain usually passes unnoticed through random test strategy but not through ADFD strategy because it scans all the values between lower and upper bounds.

The overhead in terms of execution time associated with ADFD strategy is dependent mainly on the lower and upper bounds. If the lower and upper bounds are set to maximum range (i.e. minimum for int is Integer.MIN_INT and maximum Integer.MAX_INT) then the test duration is also maximum. It is rightly so because for identification of failure 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 domains unless they are of considerable size. Test duration is also influenced by identification of the first failure 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 program between the testers and debuggers as it identify all the failures in one go. Second, it identifies locations of all failure 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 plotting of the objects 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 [75], 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. In the current study, ADFD strategy has only considered scalar data of one and two-dimensions. 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 clear view on the chart. To solve this problem, zoom feature was added to the strategy to magnify 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 [10, 110]. 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 [1].

Identification, classification of pass and fail domains and visualisation of domains have not received due attention of the researchers. Podgurski et al. [114] proposed a semi-automated procedure to classify faults and plot them by using a Hierarchical Multi Dimension Scaling (HMDS) algorithm. A tool named Xslice [115] 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 [116]. A serious limitation of the above mentioned tools is that they are not fully automated and require human intervention during execution. Moreover these tools are based on the already existing test cases where as ADFD strategy generates test cases, discovers faults, identifies pass and fault domains and visualises them in graphical form in a fully automated manner.

5.8 Summary

Experimental results obtained by ADFD strategy to error-seeded numerical program provide evidence that the strategy is highly effective in identifying the faults and plotting pass

and fail domains of a given SUT. ADFD strategy can find boundary failures quickly as against the traditional random testing, which is either, unable or takes comparatively longer time to discover the failures.

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 with a single execution. It improves debugging efficiency as the debuggers keep all the instances of a fault under consideration during debugging. The strategy has the potential to be used at large scale. However future studies are required to use it with programs of more than two dimension and different non-scaler argument types.

Chapter 6

Automated Discovery of Failure Domain Plus

6.1 Introduction

Software testing is most widely used for verification and validation process. Efforts have been continuously made by researchers to make the testing process more and more effective and efficient. Testing is efficient when maximum number of test cases are executed in minimum possible time and it is effective when it finds maximum number of faults in minimum number of test cases. During up-gradation and development of testing techniques, focus is always on increasing the efficiency by introducing partial or complete automation of the testing process and the effectiveness by improving the algorithm.

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [117, 118]. According to Chan et al. [1] the clusters are arranged in the form of point, block and strip failure domains. In the point domain the failure revealing inputs stand-alone and are evenly spread through out the input domain. In block domain the failure revealing inputs are contiguously clustered in one area. In strip domain the failure revealing inputs are clustered in one long elongated strip.

To target failures and evaluate the failure domains we developed earlier ADFD technique [119]. The ADFD+, an improved version of ADFD, is a fully automatic technique which finds failures and failure domains within a specified radius and presents the results on a graphical chart. The efficiency and effectiveness of ADFD+ technique is evaluated by comparing its performance with that of a mature testing tool Random tester for object oriented programs (Randoop) [74]. The results generated by ADFD+ and Randoop for the error-seeded programs shows better performance of ADFD+ with respect to time and number of test cases to find failure domains. Additionally ADFD+ presents the results graphically showing iden-

tified point block and strip domains visually as against Randoop which lacks graphical user interface.

6.2 Automated Discovery of Failure Domain+

It is an improved version of ADFD, a technique developed earlier by Ahmad and Oriol [119]. The technique automatically finds failures, failure domains and present the results in graphical form. In this technique, the test execution is initiated by random+ and continues till the first failure is found in the SUT. The technique then copies the values leading to the failure and the surrounding values to the dynamic list of interesting values. The resultant list provides relevant test data for the remaining test session and the generated test cases are effectively targeted towards finding new failures around the existing failures in the given SUT.
The improvements made in ADFD+ over ADFD technique are stated as follows.

- ADFD+ generates a single Java file dynamically at run time to plot the failure domains as compared to one Java file per failure in ADFD. This saves sufficient time and makes the execution process quicker.
- ADFD+ uses (x, y) vector-series to represent failure domains as opposed to the (x, y) line-series in ADFD. The vector-series allows more flexibility and clarity to represent failure and failure domains.
- ADFD+ takes a single value for the radius within which the strategy searches for a failure domain whereas ADFD takes two values as lower and upper bounds representing x and y-axis respectively. This results in consumption of lower number of test cases for detecting failure domain.
- In ADFD+, the algorithm of dynamically generating Java file at run-time has been made simplified and efficient as compared to ADFD resulting in reduced overhead.
- In ADFD+, the point, block and strip failure domains generated in the output graph present a clear view of pass and fail domains with individually labelled points of failures as against a less clear view of pass and fail domains and lack of individually labelled points in ADFD.

6.2.1 Workflow of ADFD+

ADFD+ is a fully automatic technique requiring the user to select radius value and feed the program under test followed by clicking the $DrawFaultDomain$ button for test execution. As soon as the button is clicked, YETI comes in to play with ADFD+ strategy to search

for failures in the program under test. On finding a failure, the strategy creates a Java file which contains calls to the program on the failing and surrounding values within the specified radius. The Java file is executed after compilation and the results obtained are analysed to separate pass and fail values which are accordingly stored in the text files. At the end of test, all the values are plotted on the graph with pass values in blue and fail values in red colour as shown in Figure 6.2.

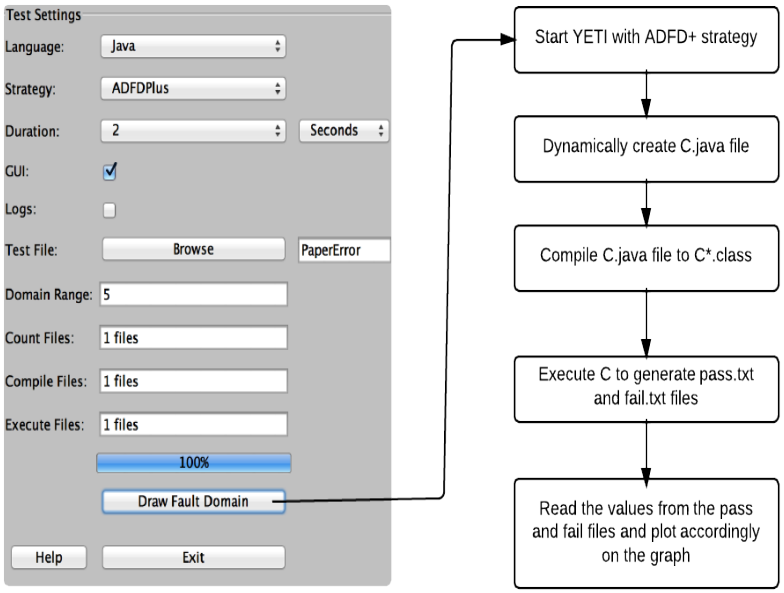


Figure 6.1: Workflow of ADFD+

6.2.2 Implementation of ADFD+

The ADFD+ technique is implemented in YETI which is available in open-source at http://code.google.com/p/yeti-test/. A brief overview of YETI is given with the focus on parts relevant to implementation of ADFD+ strategy.
YETI is a testing tool developed in Java for automatic testing of programs using random strategies. 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, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both strategies and languages sections have pluggable architecture for easily incorporating new strategies and languages making YETI a favourable choice for implementing ADFD+ strategy. YETI is also capable of generating test cases to reproduce the failures found during the test session. The strategies section in YETI contains different strategies including random, random+, DSSR and ADFD for selection according to specific needs. ADFD+ strategy is

implemented in this section by extending the $YetiADFDStrategy$.

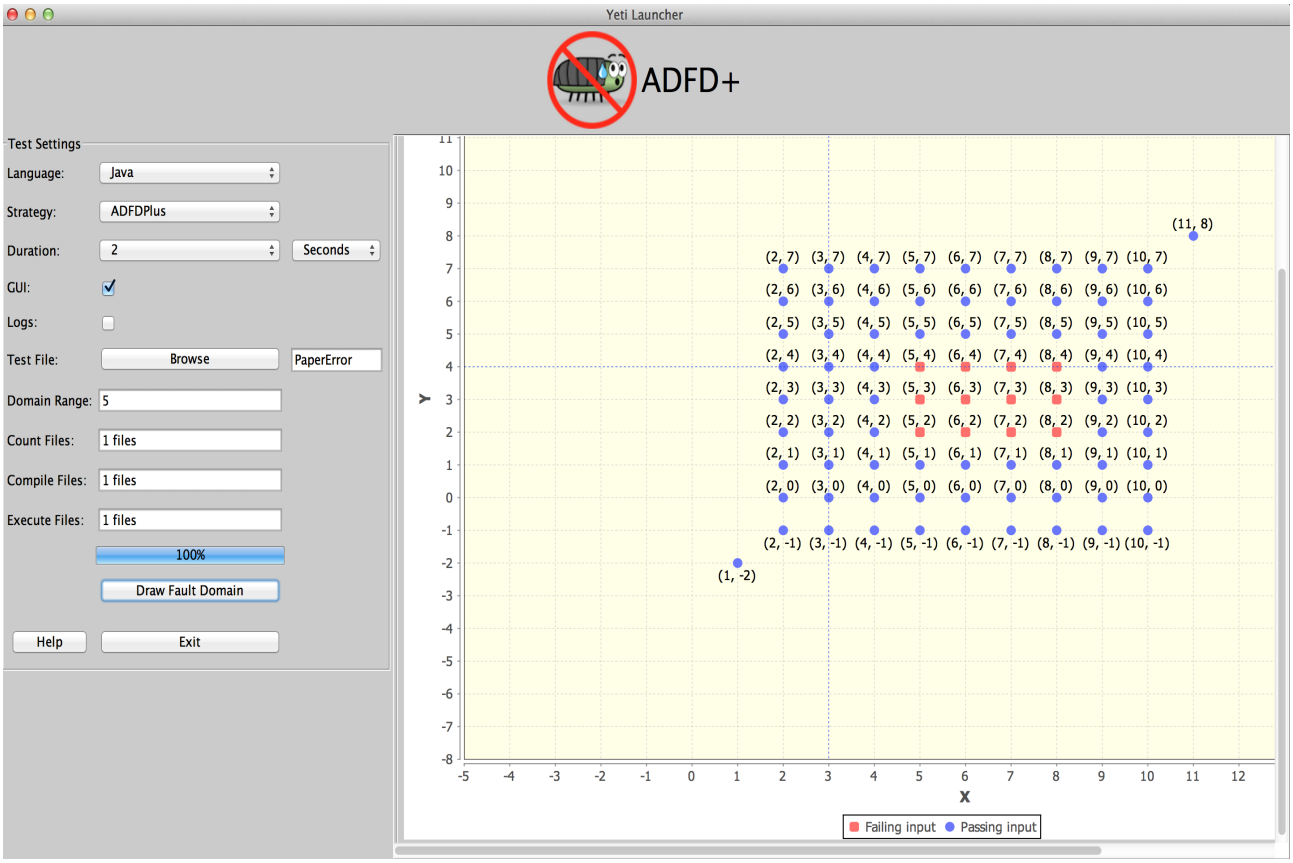


Figure 6.2: The output of ADFD+ for the above code.

6.2.3 Example to illustrate working of ADFD+

Suppose we have the following error-seeded class under test. It is evident from the program code that an $ArithmeticException$ (divison by zero) failure is generated when the value of variable x ranges between 5 to 8 and the value of variable y between 2 to 4.

```
public class Error {
  public static void Error (int x, int y){
    int z;
    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
          {
               z = 50/0;
          }
   }
}
```

At the beginning of the test, ADFD+ strategy evaluates the given class with the help of YETI and finds the first failure at x = 6 and y = 3. Once a failure is identified ADFD+ uses the surrounding values around it to find a failure domain. The radius of surrounding values is limited to the value set by the user in the $DomainRange$ variable. When the value of $DomainRange$ is set to 5, ADFD+ evaluates a total of 83 values of x and y around the found failure. All evaluated (x, y) values are plotted on a two-dimensional graph with red filled circles indicating fail values and blue filled circles indicating pass values. Figure 6.2 shows that the failure domain forms a block pattern and the boundaries of the failure are $(5, 2), (5, 3), (5, 4), (6, 2), (6, 4), (7, 2), (7, 4), (8, 2), (8, 3), (8, 4)$.

6.3 Evaluation

For evaluating the efficiency and effectiveness, we compared ADFD+ with Randoop, following the common practice of comparison of the new tool with a mature random testing tool [3, 79, 120]. Testing of several error-seeded one and two dimensional numerical programs was carried out as per program code [119]. The programs were divided in to set A and B containing one and two-dimensional programs respectively. Each program was injected with at least one failure domain of point, block or strip nature. The failure causing values are given in Table 6.2. Every program was tested independently for 30 times by both ADFD+ and Randoop. Time taken and number of tests executed to find all failure domains were used as criteria for efficiency and effectiveness respectively. The external parameters were kept constant in each test. Due to the absence of contracts and assertions in the code under test, undeclared exceptions were taken as failures in accordance with the previous studies [96, 119].

6.3.1 Research questions

The following research questions have been addressed in the study for evaluating ADFD+ technique with respect to efficiency, effectiveness and presentation of failure domains:

1. How efficient is ADFD+ as compared to Randoop?
2. How effective is ADFD+ as compared to Randoop?
3. How failure domains are presented by ADFD+ as compared to Randoop?

6.3.2 Randoop

Random tester for object oriented programs (Randoop) is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes as input a set of classes, time limit or

Table 6.1: Table depicting values of x and y arguments forming point, block and strip failure domain in Figure 6.7(a), 6.7(b), 6.7(c) and Figure 6.8(a), 6.8(b), 6.8(c) respectively

Dim	Point failure	Block failure	Strip failure
One	x = -66 x = -2 x= 51 x= 23	x = -1, 0, 1 x =-26 – -29 x = 51 – 54	x = -4 – 34
Two	x=2, y=10 x=4, y=10 x=7, y=10 x=9, y=10	x = 5, y = 2 x = 6, y = 2 x = 7, y = 2 x = 8, y = 2 x = 5, y = 3 x = 6, y = 3 x = 7, y = 3 x = 8, y = 3 x = 5, y = 4 x = 6, y = 4 x = 7, y = 4 x = 8, y = 4	x = 7, y = 0 x = 8, y = 0 x = 8, y = 1 x = 9, y = 1 x = 9, y = 2 x = 10, y = 2 x = 10, y = 3 x = 11, y = 3 x = 11, y = 4 x = 12, y = 4 x = 12, y = 5 x = 13, y = 6 x = 14, y = 6 x = 14, y = 7

number of tests and optionally a set of configuration files to assist testing. Randoop checks for assertion violations, access violations and un-expected program termination in a given class. Its output is a suite of JUnit for Java and NUnit for .Net program. 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 public methods from the class under test. Arguments for these methods are selected from the pre-defined pool in case of primitive types and as sequence of null values in case of reference type. Randoop uses feedback mechanism to filter out duplicate test cases.

6.3.3 Experimental setup

All experiments were conducted with a 64-bit Mac OS X Mountain lion version 10.8.5 running on 2.7 GHz Intel Core i7 with 16 GB (1600 MHz DDR3) of RAM. YETI runs on top of the Java™SE Runtime Environment [version 1.6.0_35]. The ADFD+ Jar file is available at https://code.google.com/p/yeti-test/downloads/list/ and Randoop at https://randoop.googlecode.com/files/randoop.1.3.3.zip.

The following two commands were used to run the ADFD+ and Randoop respectively. Both tools were executed with default settings, however, Randoop was provided with a seed value as well.

```
$ java -jar adfd_yeti.jar -------------(1)

$ java randoop.main.Main gentests \
--testclass=OneDimPointFailDomain \
--testclass=Values --timelimit=100 ----(2)
```

6.4 Experimental results

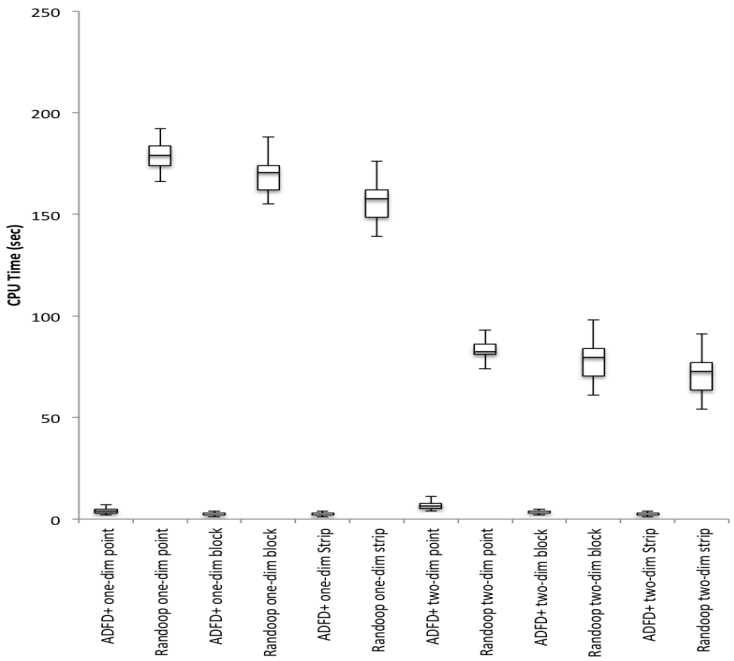


Figure 6.3: Time taken to find failures

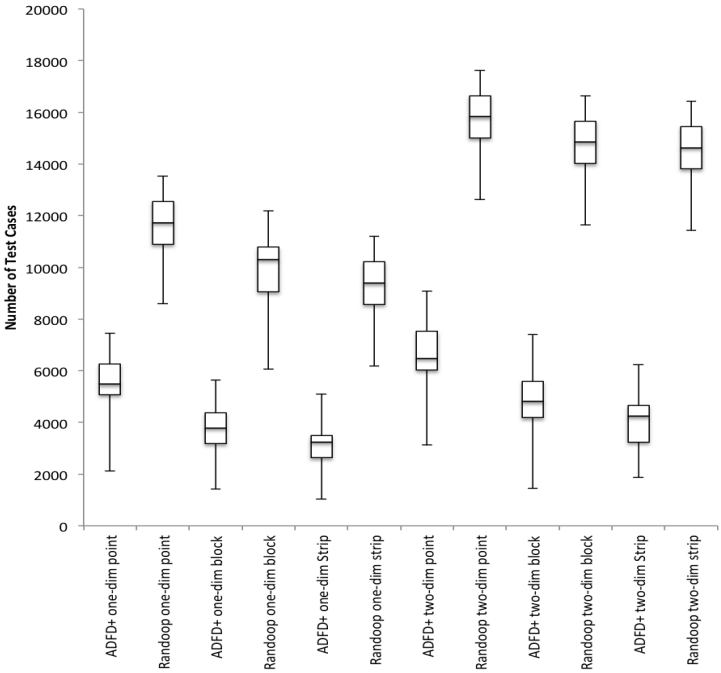


Figure 6.4: Number of test cases taken to find failures

6.4.1 Efficiency

Figure 6.5 shows the comparative efficiency of ADFD+ and Randoop. The $x - axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y - axis$ represents average time taken by the tools to detect the failure domains. As shown in the figure ADFD+ showed extra ordinary efficiency by taking two orders of magnitude less time to discover failure domains as compared to Randoop.

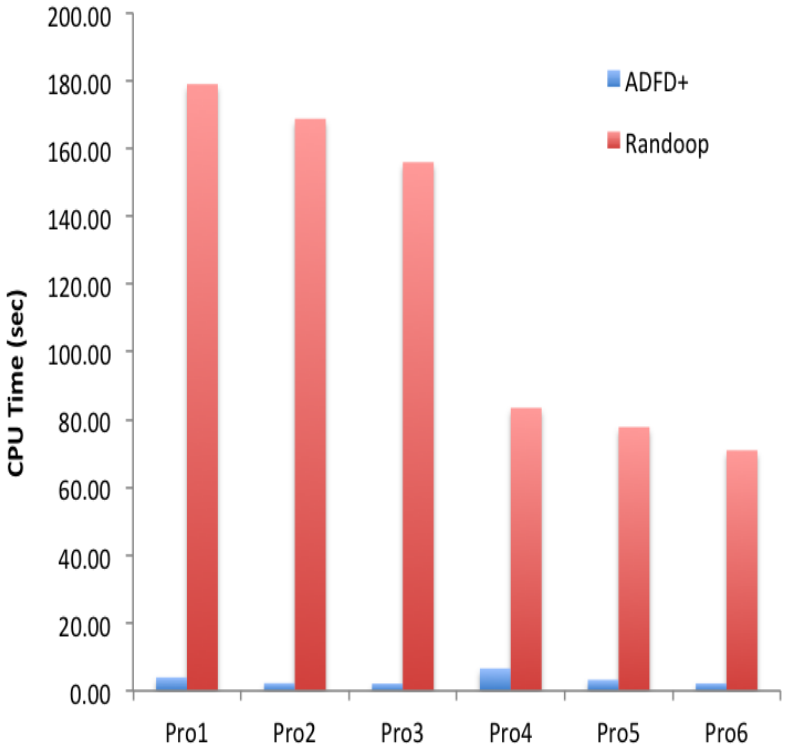


Figure 6.5: Time taken to find failure domains

This may be partially attributed to the very fast processing of YETI, integrated with ADFD+. YETI is capable of executing 10^6 test calls per minute on Java code. To counter the contribution of YETI and assess the performance of ADFD+ by itself, the effectiveness of ADFD+ was compared with Randoop in terms of the number of test cases required to identify the failure domains without giving any consideration to the time consumed for completing the test session. The results are presented in the following section.

6.4.2 Effectiveness

Figure 6.6 shows the comparative effectiveness of ADFD+ and Randoop. The $x - axis$ represents one and two-dimensional programs with point, block and strip failure domains

while the $y - axis$ represents average number of test cases used by the tools to detect the failure domains. The figure shows higher effectiveness in case of ADFD+, amounting to 100% or more. The higher effectiveness of ADFD+ may be attributed to its working mechanism in comparison with Randoop for identifying failures. ADFD+ dynamically changes its algorithm to exhaustive testing in a specified radius around the failure as against Randoop which uses the same random algorithm for searching failures.

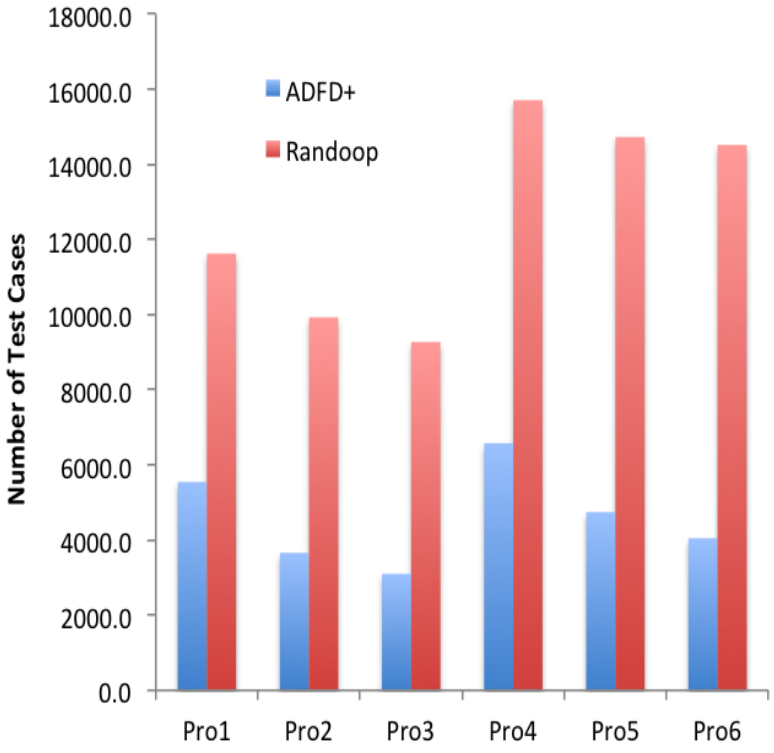


Figure 6.6: Test cases taken to find failure domains

6.4.3 Failure Domains

The comparative results of the two tools with respect to presentation of the identified failure domains reveal better performance of ADFD+ by providing the benefit of presenting the failure domains in graphical form as shown in Figure 6.7 and 6.8. The user can also enable or disable the option of showing the failing values on the graph. In comparison Randoop lacks the ability of graphical presentation and the option of showing the failure domains separately and provides the results scattered across the textual files.

6.5 Discussion

The results indicated that ADFD+ is a promising technique for finding failure and failure domain efficiently and effectively. It has the added advantage of showing the results in graphical form. The pictorial representation of failure domains facilitates the debuggers to easily identify the underlying failure domain and its boundaries for troubleshooting.

In the initial set of experiments Randoop was executed for several minutes with default settings. The results indicated no identification of failures after several executions. On analysis of the generated unit tests and Randoop's manual, it was found that the pool of values stored in Randoop database for int primitive type contains only 5 values including -1, 0, 1, 10 and 100. To enable Randoop to select different values, we supplied a configuration file with the option to generate random values between -500 and 500 for the test cases as all the seeded errors were in this range.

As revealed in the results ADFD+ outperformed Randoop by taking two orders of magnitude less time to discover the failure domains. This was partially attributed to the very fast processing of YETI integrated with ADFD+. To counter the effect of YETI the comparative performance of ADFD+ and Randoop was determined in terms of the number of test cases required to identify the failure domains giving no consideration to the time taken for completing the test session. As shown in the results ADFD+ identified all failure domains in 50% or less number of test cases.

The ADFD+ was found quite efficient and effective in case of block and strip domains but not so in case of point domains where the failures lied away from each other as shown in the following code. This limitation of ADFD+ may be due to the search in vain for new failures in the neighbourhood of failures found requiring the additional test cases resulting in increased overhead.

```
public class Error {
  public static void Error (int x, int y){
    int z;
    if (x == 10000)
         {  z = 50/0;   }

    if (y == -2000)
         {  z = 50/0;   }
   }
}
```

The number of test cases to be undertaken in search of failures around the previous failure

found is set in the range value by the user. The time taken by test session is directly proportional to the range value. Higher range value leads to larger graphical output requiring zoom feature which has been incorporated in ADFD+ for use when the need arise.

6.6 Threats to validity

The study faces threats to external and internal validity. The external threats are common to most of the empirical evaluations. It includes the extent to which the programs under test the generation tools and the nature of seeded errors are representative of the true practice. The present findings will serve as foundation for future research studies needed to be undertaken with several types of classes, test generation tools and diversified nature of seeded errors in order to overcome the threats to external validity. The internal threats to validity includes error-seeded and limited number of classes used in the study. These may be avoided by taking real and higher number of classes in future studies.

6.7 Related Work

The increase in complexity of programs poses new challenges to researchers for finding more efficient and effective ways of software testing with user friendly easy to understand test results. Adaptive Random Testing [71], Proportional random testing [1] and feedback directed random testing [74] are some of the prominent upgraded versions of random testing with better performance. Automated random testing is simple to implement and capable of finding hitherto bugs in complex programs [2, 3]. ADFD+ is a promising technique for finding failures and failure domains efficiently and effectively with the added advantage of presenting the output in graphical form showing point, block and strip domains.

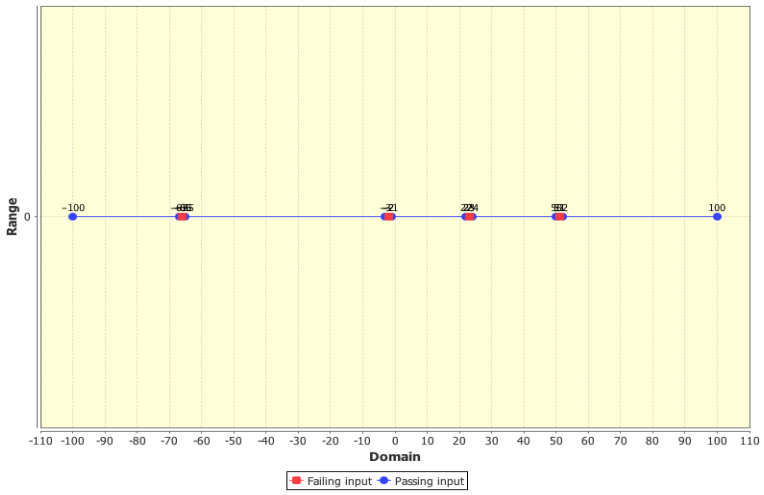
Some previous research studies have reported work on Identification, classification and visualisation of pass and fail domains in the past [114, 115, 116]. This includes Xslice [115] is used to differentiate the execution slices of passing and failing part of a test in a visual form. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [116]. Hierarchical Multi Dimension Scaling (HMDS) describes a semi-automated procedure of classifying and plotting the faults [114]. A serious limitation of the above mentioned tools is that they are not fully automated and require human intervention during execution. Moreover these tools need the requirement of existing test cases to work on where as ADFD+ strategy generates test cases, discovers failures, identifies pass and fail domains and visualises the results in a graphical form operating in fully automated manner.

6.8 Conclusion

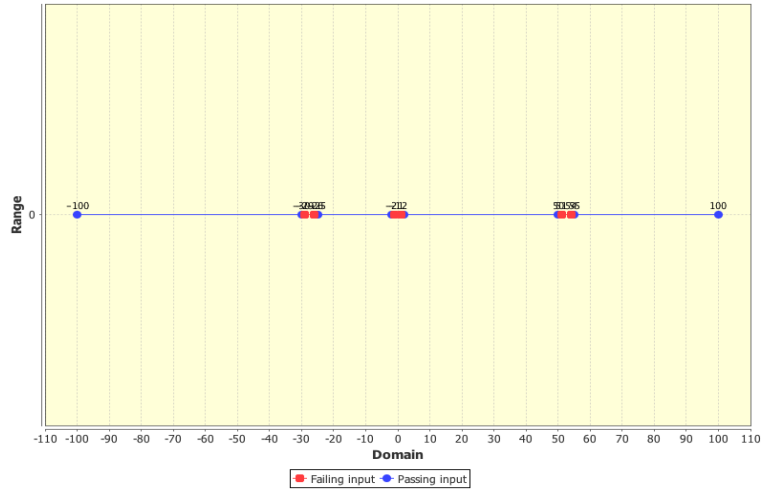
The newly developed ADFD+ technique is distinct from other random testing techniques because it not only identifies failures but also discovers failure domains and provides the result output in easily understandable graphical form. The paper highlights the improved features of ADFD+ in comparison with ADFD technique previously developed by our team [119]. The paper then analyses and compares the experimental results of ADFD+ and Randoop for the point, block and strip failure domains. The ADFD+ demonstrated extra ordinary efficiency by taking less time to the tune of two orders of magnitude to discover the failure domains and it also surpassed Randoop in terms of effectiveness by identifying the failure domains in 50% or less number of test cases. The better performance of ADFD+ may be attributed mainly to its ability to dynamically change algorithm to exhaustive testing in a specified radius around the first identified failure as against Randoop which uses the same random algorithm continuously for searching failures.

6.9 Future Work

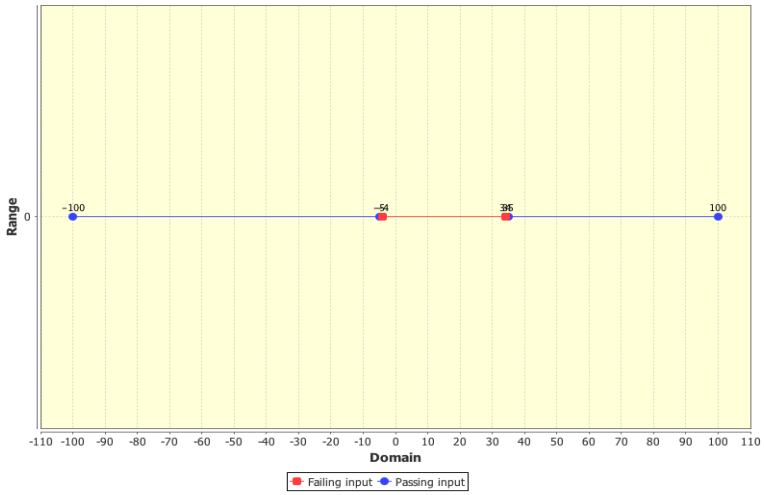
The ADFD+ strategy is capable of testing numerical programs and needs to be extended for testing of non numerical and reference data types to enable it to test all types of data. ADFD+ has the capability of graphical presentation of results for one and two-dimensional numerical programs. It is worthwhile to extend the technique to enable it to present the results of multi-dimensional numerical and non numerical programs in the graphical form.



(a) Point failure domain in one-dimension

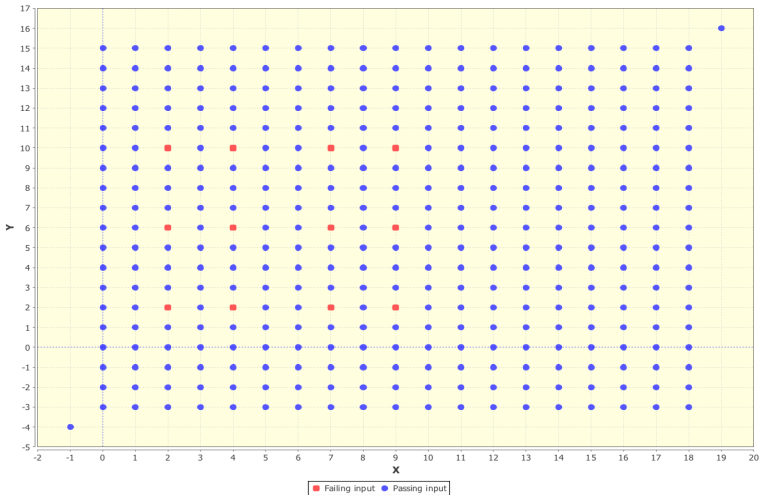


(b) Block failure domain in one-dimension

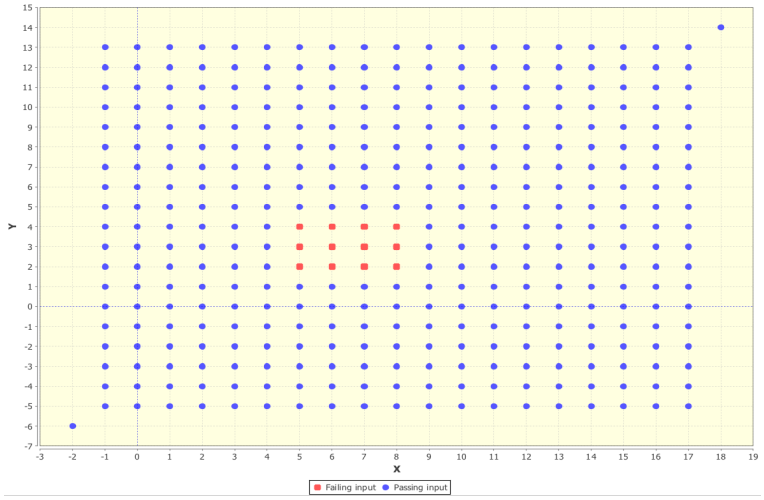


(c) Strip failure domain in one dimension

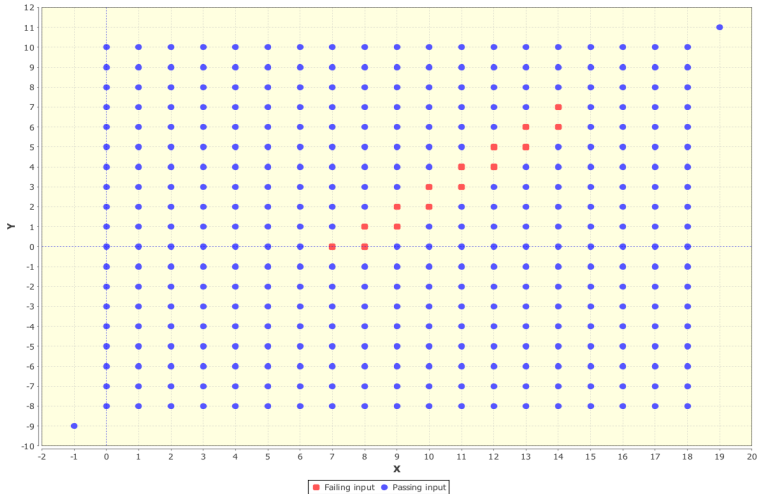
Figure 6.7: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs



(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Figure 6.8: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

Table 6.2: Table depicting results of ADFD and ADFD+

S. No	Project	Class	Method	Dim	Failure	AD
1	ant	LeadPipeInputStream	LeadPipeInputStream(i)	one		
2	antlr	BitSet	BitSet.of(i,j)	two		
3	artofillusion	ToolPallete	ToolPalette(i,j)	two		
4	aspectj	AnnotationValue	whatKindIsThis(i)	one		
		IntMap	idMap(i)	one		
5	cayenne	ExpressionFactory	expressionOfType(i)	one		
6	collections	ArrayStack	ArrayStack(i)	one		
		BinaryHeap	BinaryHeap(i)	one		
		BondedFifoBuffer	BoundedFifoBuffer(i)	one		
		FastArrayList	FastArrayList(i)	one		
		StaticBucketMap	StaticBucketMap(i)	one		
		PriorityBuffer	PriorityBuffer(i)	one		
7	colt	GenericPermuting	permutation(i,j)	two		
		LongArrayList	LongArrayList(i)	one		
		OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	one		
8	drjava	Assert	assertEquals(i,j)	two		
		ByteVector	ByteVector(i)	one		
9	emma	ClassLoaderResolver	getCallerClass(i)	one		
		ElementFactory	newConstantCollection(i)	one		
		IntIntMap	IntIntMap(i)	one		
		ObjectIntMap	ObjectIntMap(i)	one		
		IntObjectMap	IntObjectMap(i)	one		
10	heritrix	ArchiveUtils	padTo(i,j)	two		
		BloomFilter32bit	BloomFilter32bit(i,j)	two		
11	hsqld	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	one		
		ObjectCacheHashMap	ObjectCacheHashMap(i)	one		
12	htmlunit	ObjToIntMap	ObjToIntMap(i)	one		
		Token	typeToName(i)	one		
13	itext	PRTokeniser	isDelimiterWhitespace(i)	one		
		PdfAction	PdfAction(i)	one		
		PdfLiteral	PdfLiteral(i)	one		
14	jung	PhysicalEnvironment	PhysicalEnvironment(i)	one		
15	jedit	IntegerArray	IntegerArray(i)	one		
16	jgraph	AttributeMap	AttributeMap(i)	one		
17	jruby	ByteList	ByteList(i)	one		
		WeakIdentityHashMap	WeakIdentityHashMap(i)	one		

Chapter 7

Conclusions

This research aims at understanding the nature of failures in software, discovering how to leverage failure domain for finding more bugs and developing new improved automated random test strategies to achieve the desired objectives. The existing random test strategies find individual failures and do not focus on failure domain. The knowledge of failure along with failure domain is of great benefit to debuggers for quick and effective removal of failures.

Various aspects of failure-domains with respect to automated random testing were explored. The study focused on three main issues. First, to minimize the number of test cases required to discover a failure-domain. Second, to identify the pass and fail input domains and generate the result in graphical form. Third, to compare the developed strategy with the standard automated tool Daikon for finding the failure domain was the third focus of the study.
It was revealed in the study that the input inducing failures residue in contagious locations forming certain geometrical shapes in the input domain. These shapes can be divided into point, block and strip domains. A set of techniques and tools have been developed for improving the effectiveness of automated random testing in finding failures and failure-domains.

The first technique, Dirt Spot Sweeping Random (DSSR) strategy starts by testing the program at random. When a failure is identified, the strategy selects the neighbouring input 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 a quick and efficient identification of failures in the software under test. The results stated in Chapter 4 showed that DSSR performs significantly better than random and random+ strategies.

The second technique, Automated Discovery of Failure Domain (ADFD) was developed to find failure and failure-domains in a given software and provides visualization of the iden-

tified pass and fail domains within a specified range in the form of a chart. The technique starts with a random+ strategy to find the first failure. When a failure is identified, a new Java program is dynamically created at run-time which, then compiled and executed to search for failure-domains along the projections on various axis. The output of the program shows pass and fail domains in the graphical form. The results stated in Chapter 5 showed that ADFD technique correctly identify the failure domains. The technique is highly effective in testing and debugging by providing an easy to understand test report in the visualized form.

The third technique, Automated Discovery of Failure Domain+ (ADFD+) is an upgraded version of ADFD technique with respect to algorithm and graphical representation of failure domains. The new algorithm searches for the failure-domain around the failure in a given radius as against ADFD which limits the search between lower and upper bounds. The ADFD+ graphical output was further improved by providing labelled graphs to make it easily understandable and user friendly. To find the effectiveness of ADFD+, it was compared with Daikon using error seeded programs. The ADFD+ correctly pointed out all the seeded failure domains while Daikon identified individual failures but was unable to discover the failure domains.

7.1 Lessons Learned

Research in the field of software testing has been proceeding for more than three decades but only a handful of free and open source fully automated testing tools are available for software testing. The current study is a continuation of the research efforts to find improved testing techniques capable to identify failures and failure domains quickly, efficiently and effectively. In this section, the lessons learned during the study are presented in the summarized form which may be of interest to the researchers pursuing future research.

Selection of performance measurement criteria

Among the three measuring techniques used for finding the effectiveness of random testing, the E-measure and P-measure have been criticised [71] whereas the F-measure has been often used by researchers [98, 99]. In our experiments, the F-measure was initially used but its weakness was soon realised as stated in Section 4.4.3. The F-measure is effective in traditional testing and counts the number of test cases used to find the first failure. The system is then handed over to developers for fixing the identified failure. Automated testing tools test the whole system and report all discovered failures in one go, thus the F-measure is not the favourable choice. We addressed the issue by measuring the maximum number of failures detected in a particular number of test calls as the criterion for finding the effectiveness of the test strategy.

Test results in random testing keep on changing

In random testing, due to the random generation of test input, the results keep on changing even if all the test parameters and the program under test remain the same. Therefore the efficiency of one technique in comparison with the other becomes difficult. We addressed the issue by taking five steps. 1) Each experiment was repeated 30 times and the average was taken for comparison 2) In each experiment 10000 test cases were executed to minimize the random effect 3) Sufficiently large number of representative test samples (60) were taken for evaluation 4) Error seeded programs with known locations of faults were used to verify the results 5) The experimental results were statistically analysed to estimate the difference on statistical basis.

Testing of neighbouring values around the failure decreases computation

Developing new versions of random testing with higher fault finding ability usually result in increased computation, higher overhead and lower performance. We addressed the issue by developing new strategies which uses neighbouring values around the failure finding value for the subsequent tests. This approach saves the computation involved in generating suitable test values from the whole input domain.

Random testing coupled with exhaustive testing

Random testing test values at random while exhaustive testing test the whole input domain which is very effective but not usually feasible for a larger domain. The issue was addressed by coupling the random testing with exhaustive testing. In our newly developed strategies the testing starts at random till a failure is identified and then switches to ex-

haustive testing to select the values around the failure finding value in the sub-domain set by the tester. This partially provides the benefit of exhaustive testing in the random testing and results in quick identification of the neighbouring failures which may be difficult to find by using random testing alone.

Easy to understand user friendly test output

Random testing is no exception when it comes to the complexity of understanding and evaluating test results. No random strategy seems to provide graphical representation of the failures and failure-domains. The issue of getting in easy to understand user friendly format has been addressed the present study. The ADFD strategy has been developed with the feature of giving the result output in the visualized graphical form. This feature has been further improved in the ADFD+ strategy which clarify and label individual failures and the failure domains in two-dimensional graph. The identification of failures and failure domains in graphical form helps developers to easily follow the test reports while fixing the faults.

Auto-generation of primitive and user-defined data types

We noticed that auto-generation of user defined data type is more complex as compared to the primitive data type. We addressed the issue by creating objects of the classes under test and randomly calling the methods with random inputs in accordance with the parameter's space. The inputs were divided into primitive types and user defined data types. For primitive data generation `Math.random()` method is used and for generation of user-defined data object of the class is created at run time as stated in Section-sec:constructionOfTestCases. The approach adopted helps in achieving a fully automated testing system.

Chapter 8

Future Work

This chapter presents the scope and potential of future work as an extension of the present research study. The topics suggested include: use of contracts and assertions to discover failure; introducing object distance in DSSR strategy to enhance its testing ability; measuring code coverage of the three newly developed strategies to assess additional aspect of performance; extension of ADFD+ strategy for testing non numerical data; enhancing the plotting ability of ADFD+ strategy to more than two-dimensional charts; introducing additional features in the user interface of ADFD+; extension of ADFD+ to apply it to the real world scenario; research on the prevalence of point block and strip failure domains; improvement in the DSSR strategy to reduce overhead.

Improvement in the DSSR strategy to reduce overhead

The DSSR strategy is an extension of random+ strategy based on the assumption that failure domains are contagious. The dirt spot sweeping feature of the strategy adds the neighbouring value of the failure finding value to the list of interesting values to cover the failure domain. This add 5% overhead to DSSR strategy compared to R+ strategy. In future studies the algorithm may be modified to decrease the overhead and make the strategy more effective.

Use of contracts and assertions to discover failure

The common practice to use undefined run-time exceptions of the programming language as test oracles in the absence of contracts and assertions was followed in the study. It is worthwhile to study the fault-detection ability of an automated strategy in the presence of contracts and assertions. To generate explicit oracles a tool like Daikon may be integrated in the system for achieving the automatic generation of invariants and their annotation in to source code.

Introducing object distance in DSSR strategy to enhance its testing ability

The newly developed DSSR strategy add the neighbouring values for primitive type data and Strings. It has a limitation that no neighbouring values are added when the failure is found by a reference type data. It is suggested for future research work to extend the DSSR strategy by incorporating a suitable technique like Artoo to find the neighbouring objects for including these in the list of interesting values.

Measuring code coverage of the three newly developed strategies to assess additional aspect of performance

In spite of the fact that the strategies developed in the study generate more test cases from the surrounding area where a failure is discovered for better coverage, it is worthwhile to measure the code coverage achieved by the new strategies to ensure the effectiveness. The instrumentation technique may be applied to the software under test to achieve the desired objective.

Extension of ADFD+ strategy for testing non numerical data

The ADFD+ strategy tests numerical programs and can be used for testing the software of numerical data types in the real world context. The strategy may be extended to include testing of non numerical and reference data types to enable the strategy to test all types of data.

Enhancing the plotting ability of ADFD+ strategy to more than two-dimensional charts

The newly developed ADFD+ has the capability of graphical representation of results for one and two-dimensional numerical programs. It is worthwhile to extend the strategy so as to be capable of graphical representation of results for multi-dimensional numerical and non numerical programs.

Introducing additional features in the user interface of ADFD+

The user interface of ADFD+ provides a fully automated mechanism of testing the program, processing the results and visually representing the results in graphical form. The user interface may be extended in future to give choice to the tester for real time interaction, manual addition of test cases, showing thumbnail view of previous graphs and 3D support to present multi-dimensional arguments.

Extension of ADFD+ to apply it to the real world scenario

The newly developed ADFD+ strategy uses error-seeded programs for assessment of accuracy and effectiveness. This may likely expose it to external validity threat. Future studies may be undertaken in the real world scenario by including the feature of testing non numerical and reference data types so that there is no more threat to validity.

Research on the prevalence of point block and strip failure domains

In accordance with the reported literature, the three newly developed strategies used the concept of failure laying in point block and strip failure domains. It is worthwhile to undertake study for determining the prevalence and proportionate distribution of the failure domains in the input domain. This will improve the testing efficiency by giving due focus to the more prevalent types of failure domain.

Appendix A

A.1 Sample code to identify failure domains

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);
        }
    }
}
```

A.2 Error-seeded code to evaluate the performance of ADFD and ADFD+

Program 1 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 2 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 3 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 4 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 5 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 6 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] FT Chan, Tsong Yueh Chen, IK Mak, and Yuen-Tak Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

[2] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[3] Carlos Pacheco and Michael D Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

[4] Andreas Leitner and Ilinca Ciupa. Reconciling manual and automated testing: the autotest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, pages 3–6. Technology, 2007.

[5] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 22–31. IEEE, 2001.

[6] Maurice Wilkes. *Memoirs of a computer pioneer*. Massachusetts Institute of Technology, 1985.

[7] 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.

[8] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

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

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

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

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

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

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

[15] 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*, pages 84–94. ACM, 2007.

[16] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

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

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

[19] Simson Garfinkel. History's worst software bugs. *Wired News, Nov*, 2005.

[20] 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.

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

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

[23] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2:9, 2001.

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

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

[26] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

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

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

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

[30] 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.

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

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

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

[34] Frank Armour and Granville Miller. *Advanced use case modeling: software systems*. Pearson Education, 2000.

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

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

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

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

[39] 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.

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

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

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

[43] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 157–166. IEEE, 2008.

[44] Andreas Leitner, Alexander Pretschner, Stefan Mori, Bertrand Meyer, and Manuel Oriol. On the effectiveness of test extraction without overhead. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 416–425. IEEE, 2009.

[45] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 2000.

[46] ECMA ECMA. 367: Eiffel analysis, design and programming language. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr*, 2005.

[47] GT Leavens, E Poll, C Clifton, Y Cheon, C Ruby, D Cok, and J Kiniry. Jml reference manual (draft), 2005.

[48] Reto Kramer. icontract-the java tm design by contract tm tool. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 295–307. IEEE, 1998.

[49] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Conceptual Modeling–ER98*, pages 449–464. Springer, 1998.

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

[51] 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.

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

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

[54] 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.

[55] 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.

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

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

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

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

[60] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 71–80. IEEE, 2008.

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

[62] 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.

[63] 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.

[64] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

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

[71] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[72] Tsong Yueh Chen, F-C Kuo, Robert G Merkel, and Sebastian P Ng. Mirror adaptive random testing. *Information and Software Technology*, 46(15):1001–1010, 2004.

[73] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. *Reliability, IEEE Transactions on*, 56(3):562–568, 2007.

[74] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[75] 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*, pages 55–63. ACM, 2006.

[76] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

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

[78] 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.

[79] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality*, pages 242–256. Springer, 2005.

[80] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[81] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[82] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[83] Ilinca Ciupa, Andreas Leitner, et al. Automatic testing of object-oriented software. In *In Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science*. Citeseer, 2007.

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

[85] 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.

[86] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133. ACM, 2002.

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

[88] 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.

[89] Manuel Oriol and Sotirios Tassis. Testing .NET code with YETI. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 264–265. IEEE, 2010.

[90] Manuel Oriol and Faheem Ullah. YETI on the cloud. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 434–437. IEEE, 2010.

[91] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[92] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[93] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.

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

[95] Simeon C. Ntafos. On comparisons of random, partition, and proportional partition testing. *Software Engineering, IEEE Transactions on*, 27(10):949–960, 2001.

[96] Manuel 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. IEEE, 2012.

[97] 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*, pages 3–12. Australian Computer Society, Inc., 2010.

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

[99] Tsong Yueh Chen and Yuen Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109–119, 1996.

[100] 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.

[101] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81. IEEE, 2008.

[102] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing. In *Software QualityECSQ 2002*, pages 321–330. Springer, 2006.

[103] Tsong Yueh Chen, Robert G Merkel, G Eddy, and PK Wong. Adaptive random testing through dynamic partitioning. In *QSIC*, pages 79–86, 2004.

[104] Shin Yoo and Mark Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012.

[105] Walter J Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661–674, 1999.

[106] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[107] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[108] Ewan 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. IEEE, 2008.

[109] Richard C Linger. Cleanroom software engineering for zero-defect software. In *Proceedings of the 15th international conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1993.

[110] A Jefferson Offutt and J Huffman Hayes. A semantic model of program faults. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 195–200. ACM, 1996.

[111] 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.

[112] 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.

[113] D Gilbert. The JFreeChart class library version 1.0. 9: Developers guide. *Refinery Limited, Hertfordshire*, 48, 2008.

[114] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda 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. IEEE, 2003.

[115] 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.

[116] 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*, pages 467–477. ACM, 2002.

[117] George B Finelli. NASA software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[118] Christoph Schneckenburger and Johannes Mayer. Towards the determination of typical failure patterns. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90–93. ACM, 2007.

[119] Mian A Ahmad and Manuel Oriol. Automated discovery of failure domain. *Lecture Notes on Software Engineering*, 03(1):289–294, 2013.

[120] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.