New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group

Department of Computer Science

University of York, UK

March 2014

A thesis submitted for the degree of Doctor of Philosophy

Abstract

Software testing is the process of evaluating the quality of a software or its component.The thesis presents new techniques for improving the effectiveness of automated random testing, evaluates the efficiency of these techniques and proposes directions for future work.

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

Contents

List of Figures

List of Tables

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

Acknowledgements

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

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

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

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

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

Chapter 1

Introduction

Software is an important and essential component of computer system without which no task can be accomplished. Some software are developed for use in simple day to day operations while others are for highly complex processes in specialised fields including education, business, finance, health, science and technology etc. The ever increasing dependency on software expect us to believe that software are reliable, robust, safe and secure. However, like every other man-made items software are also prone to errors. Maurice Wilkes [7], a British computer pioneer stated that,

> "As soon as we started programming, we found to our surprise that it was not as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."

The margin of error in mission-critical and safety-critical systems is so small that a minor fault can lead to huge economic losses [8]. According to the National Institute of Standards and Technology, US companies alone bear $59.5 billion loss every year due to software failures [9]. Therefore, software companies leave no stone unturned to ensure the reliability and accuracy of the software before its practical application. Software testing is the most recognized and widely used technique to verify the correctness and ensure quality of the software [10]. According to Myers et al. some software companies spend up to 50% of the total development and maintenance cost on software testing [11].

1.0.1 Software Testing

It is a technique used during Verification and Validation (V & V) process to ensure that the software adheres to the desired specifications. According to Dijkstra, program testing can be used to show the presence of bugs, but never to show the absence of bugs [12]. 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.

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 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 [13, 14]. Miller and Maloney were the first who comprehensively described a systematic approach of test data set selection known as path coverage. They proposed that testers should select the test data so that all paths of the SUT are executed at least once [15]. The approach resulted in higher standards of test quality and a large number of test strategies were subsequently developed such as boundary value analysis and equivalence class.

Test data set can be generated manually and automatically. However, generating test data set manually is a time-consuming and laborious exercise [16], therefore, automated test data set generation is always preferred. Data generators can be of different types i.e. Path-wise (Section 2.6.1), Goal-oriented (Section 2.6.2), Intelligent (Section 2.6.3) and Random (Section 2.6.5). Random generator produces test data set randomly from the whole domain. Unlike other approaches random technique is simple, widely applicable, easy to implement, faster in computation, free from bias and costs minimum overhead [17].

1.0.2 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. [18] “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 i.e. test data generation, execution and evaluation are shown in Figure 1.1.

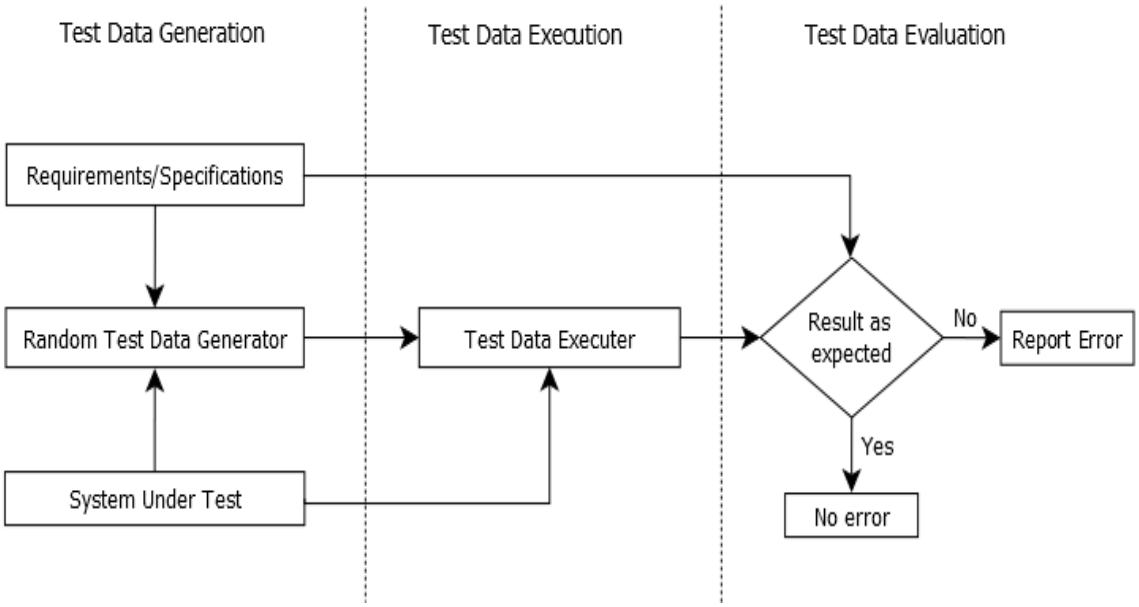


Figure 1.1: Three main phases of random testing

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

1.1 The Problems

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [19, 11]. This research study focuses on the following problems in automated Random Testing (RT):

1. Limitation of RT in the discovery of contagious failures.
2. Inability of RT to identify failure-domains.
3. Incompetence of RT to represent results in graphical form.

1.1.1 Limitation of RT in the Discovery of Contagious Failures

Chan et al. [1] observed that failure inducing inputs are contagious 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 2.9.2). The failure-finding ability of random testing technique decreases when the failures lie in contiguous block and strip patterns across the input domain.

1.1.2 Inability of RT to Identify Failure-domains

The existing random strategies of software testing tries to discover failure individually and lack the capability to discover the failure domain.

1.1.3 Incompetence of RT to Represent Results in a 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 a user-friendly graphical form.

1.2 Research Goals

Goals of the research study are discovering how to leverage failure domain for finding more bugs and understanding the nature of failures and to develop new improved automated random test strategies to achieve the desired goals.

1.3 Contributions

The main contributions of the thesis research are as follows:

1.3.1 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. To overcome the problem, a new automated technique: Dirt Spot Sweeping Random (DSSR) strategy was developed. It is based on the assumption that unique failures reside in contiguous blocks and 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 (R) and Random+ (R+) strategies.

1.3.2 Automated Discovery of Failure Domain

The existing random strategies of software testing discover the faults in the SUT but lack the capability of presenting the fault domains. In the present study, a fully automated testing strategy named, “Automated Discovery of Failure Domain (ADFD)” is developed with the ability to find the failures as well as the 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 prove that ADFD strategy automatically performs identification of failures and failure-domains and provides the results in graphical form.

1.3.3 Automated Discovery of Failure Domain+

Another random test strategy named, “Automated Discovery of Failure Domain+ Strategy” (ADFD+) is developed in the current research study.

1.4 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 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 faults 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 faults and fault domains in a specified limit and plots them on a chart. Experimental evidence is presented in support of ADFD strategy applied to several one and two dimensional programs.

Chapter 6 presents the Automated Discovery of Failure Domain+ Strategy (ADFD+), a newly proposed testing technique that automatically generates invariants of SUT using Daikon, filter and annotate the invariants in the source code to better support testing process. The IGRS technique like DSSR and ADFD is also implemented in YETI. Experimental study is presented in which the effectiveness of IGRS in finding faults is compared with the random, random+ and DSSR strategies.

Chapter 7 concludes with a summary of the contributions and the lessons learned.

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

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

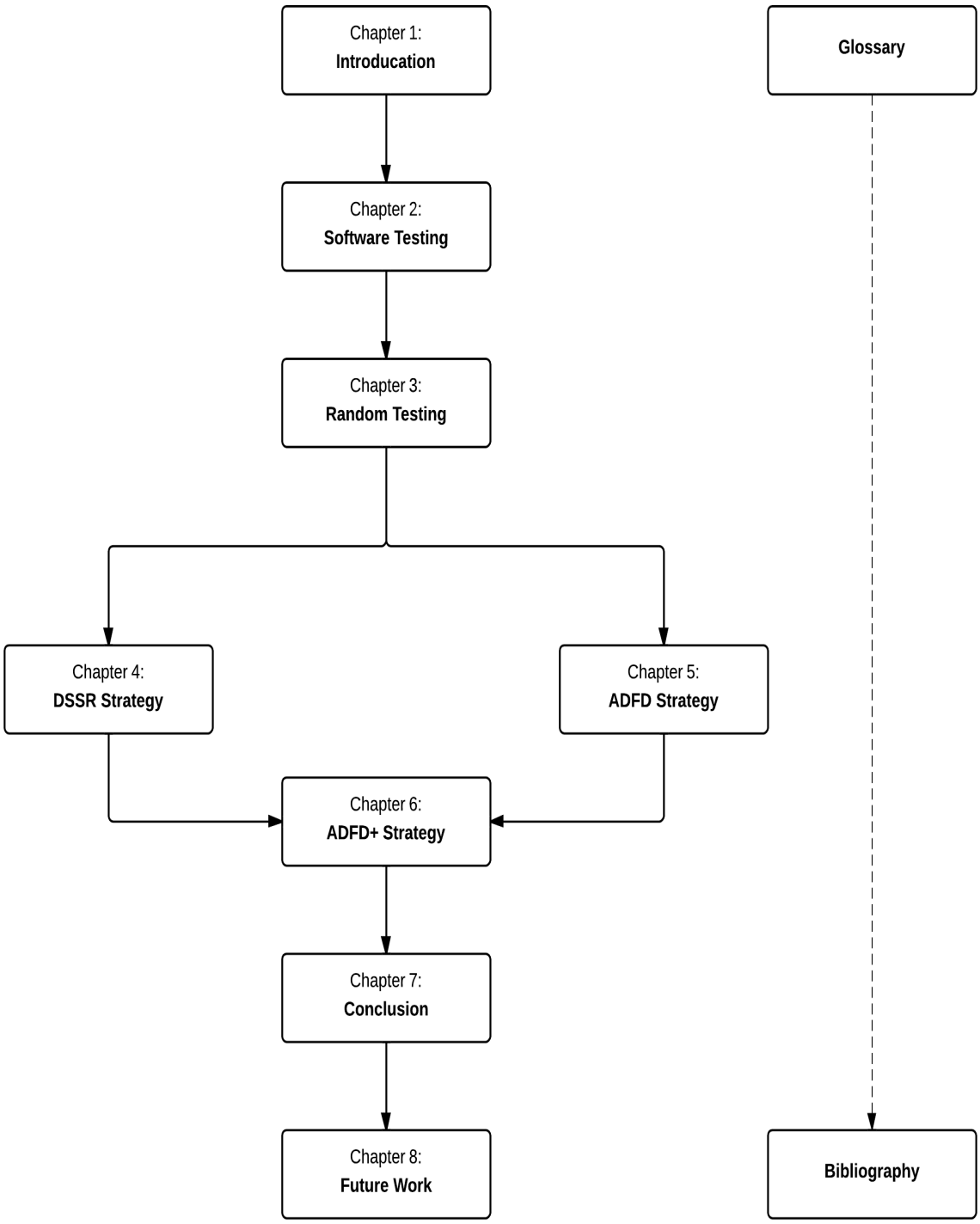


Figure 1.2: Structure of thesis outline

Chapter 2

Literature Review

The famous quote of Paul, “to err is human, but to really foul things up you need a computer”, is quite relevant to the software programmers. Programmers being humans are prone to errors. Therefore, in spite of best efforts, some errors may remain in the software after it is finalised. Errors cannot be tolerated in software because a single error may cause a large upset in the system. The destruction of Mariner 1 rocket (1962) due to unit conversion error 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 were caused by minor errors in the software [20]. According to the National Institute of Standards and Technology, US companies alone bear $59.5 billion loss every year due to software failures and one-third of that can be eliminated by an improved testing infrastructure [9]. To achieve high quality, a software has to satisfy rigorous stages of testing. The more complex the software, the higher the requirements for software testing and the larger the damage caused when a bug remains in the software.

2.1 Software Testing

According to the IEEE standard glossary of software engineering [21], 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 process of

software testing in its simplest form is shown in Figure 2.1.

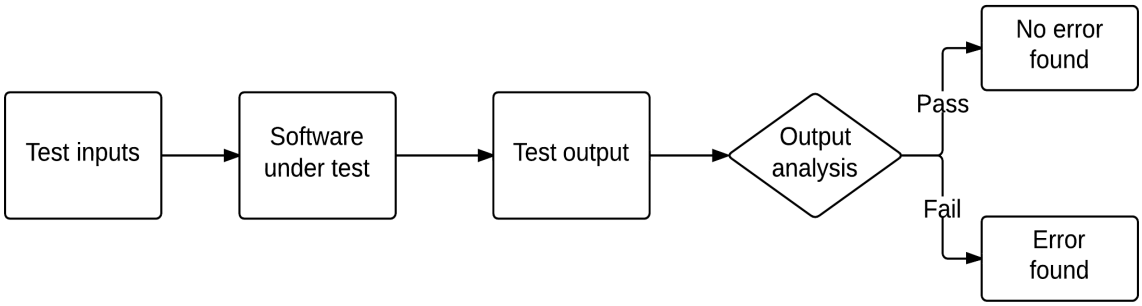


Figure 2.1: A simplified version of software testing process

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 [22]. 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 testers find a fault in the software, it is returned to the developers for rectification and consequently 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 fault in the software [11]. Fault denotes error made by programmers during software development [21]. The faults on execution can lead to software failures. A software that passes all the tests without giving a single error is not guaranteed to contain no error. The testing process, however, increases reliability and confidence of users in the tested product. Software testing can be divided in to various parts as shown in Figure 2.2.

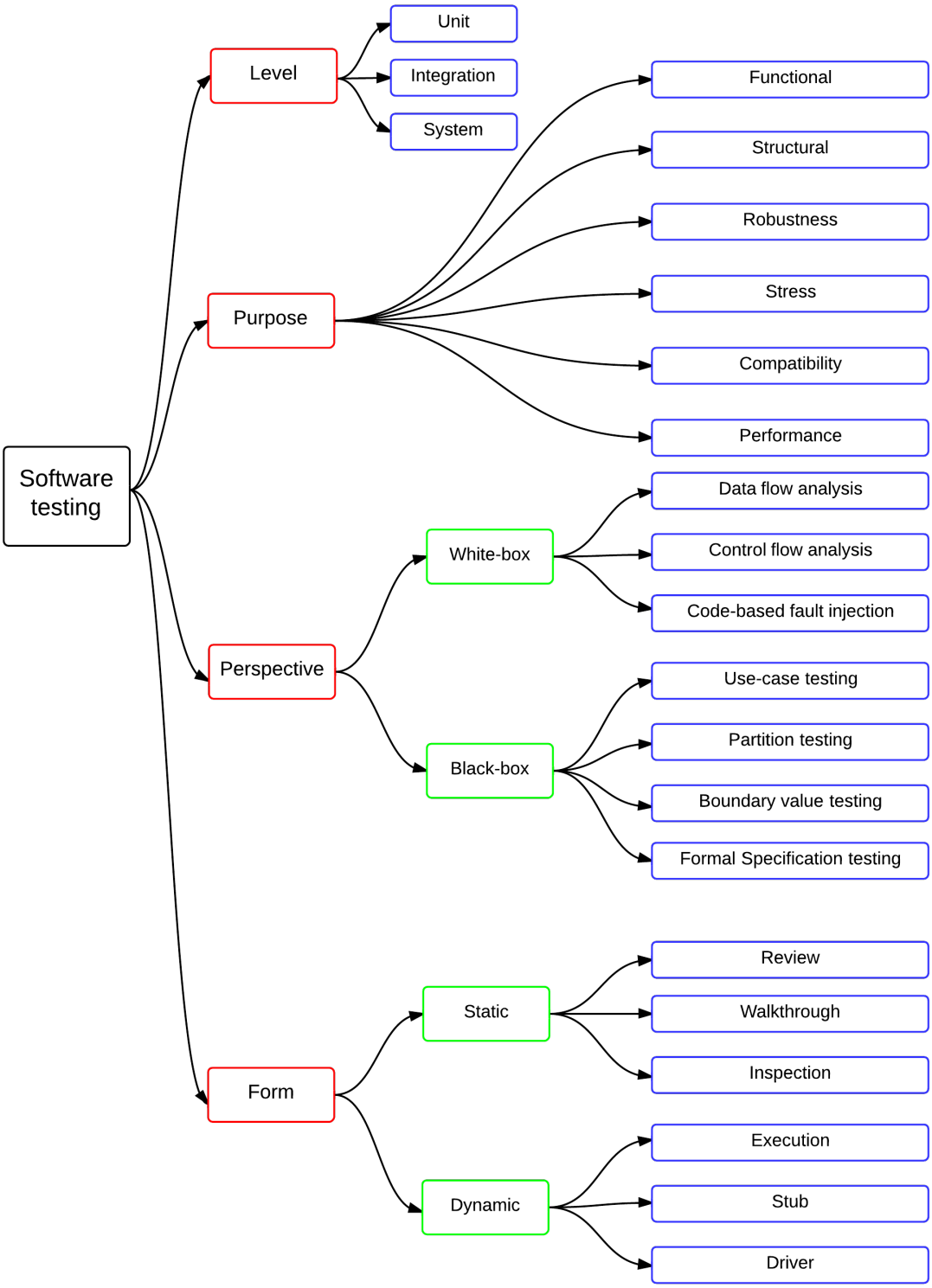


Figure 2.2: Testing from various views

2.2 Software Testing Levels

Unit testing, Integration testing and System testing are the three main levels of software testing reported in the literature [23]. Unit testing deals with evaluation of code piece-by-piece and each piece is considered as independent unit. Units are combined together to form components. Integration testing is performed to make sure that integration of units in a component are working properly. System testing ensures that the system formed by the combination of components proceeds properly to give the required output.

2.3 Software Testing Purpose

The purpose of software testing is identification of faults in the given SUT for necessary correction in order to achieve high quality. Maximum number of faults can be identified if software is tested exhaustively. 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 aspect are working correctly. Structural testing analyses the code structure for generating test cases in order to evaluate paths of execution and identification of unreachable or dead code. In robustness testing the software behaviour is observed when software receives input outside the expected input range. Stress and performance testing aims at testing the response of software under high load and checking its ability to process different nature of tasks [24]. Compatibility testing is performed to see the interaction of software with the underlying operating system.

2.4 Software Testing Perspective

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

2.4.1 White-box Testing

In white-box or structural testing, the testers must know about the complete structure of the software so that they may make necessary modifications, if so required. Test cases are derived from the code structure and test passes if the results are correct and the proper code is followed during test execution [25]. Some commonly used white-box testing techniques are as follows:

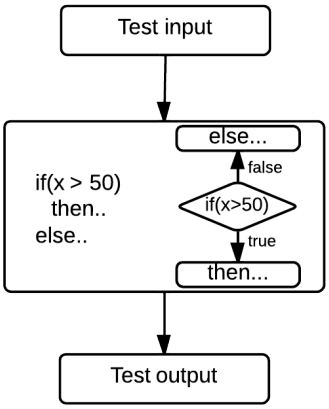


Figure 2.3: White-box testing

2.4.1.1 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 [26]. In this technique a Control Flow Graph (CFG), graphically 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.

The process of program execution can be looked into as data-flow from input to output where data may transform into several intermediary steps before reaching the final state. The process is prone to several errors e.g. references made to non existing variables, values assigned to undeclared variables or change of variables in undesired manner. Ordered use of data is crucial to ensure that the aforementioned errors do not occur [27].

2.4.1.2 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. In this technique a CFG, similar to the one required in DFA, is drawn to determine the traversable paths by a program during the execution. Test cases are generated and executed to verify conformance with CFG on the basis of control. Taking the example of following a specific path between two or more available choices at a particular state, efforts are made to

ensure that the set of selected test cases execute all the possible control choices at least once. Two of the most common measurement criteria defined by Vilkomir et al. are Branch coverage and Condition coverage [28].

2.4.1.3 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 instructions. [29]. 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.

2.4.2 Black-box Testing

In black-box or functional testing, the testers do not need to know about the internal code structure of the SUT. Test cases are derived from the software specifications and test passes if the result is according to expected output irrespective of the internal code followed during test execution [30]. Some commonly used black-box testing techniques are as follows.

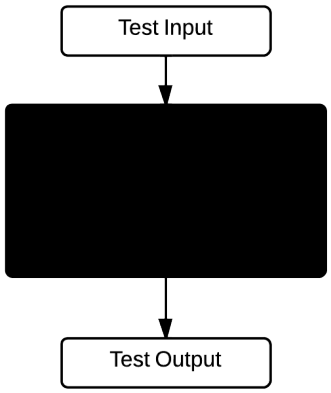


Figure 2.4: Black-box testing

2.4.2.1 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 and flow of events, pre-conditions, post-conditions, extension points, context and activity diagrams. The use-case contains all the information required for test case, therefore, it can be easily transformed into a test case [31]. Use-case testing is

beneficial in terms of cheap generation of test cases, avoidance of test duplication, increased test coverage, easier regression testing and early identification of missing requirements.

2.4.2.2 Partition Testing

It is a testing technique in which the input domain of a given SUT is divided into equal sub-domains for testing each sub-domain individually. The division is based on software specifications, structure of the code and the process involved in software development [32]. The performance of partition testing is directly proportional to the quality of sub-domain [33]. However, division of input domain into equal partitions is often difficult. To overcome the problem, a new version of partition testing, called proportional sampling strategy [1] is devised. In this version, the sub-domains vary in size and the number of test cases selected from each partition is directly proportional to the size of the partition. Experiments performed by Ntafos [34] has provided evidence for better performance of proportional partition testing.

2.4.2.3 Boundary Value Analysis

Boundary Value Analysis (BVA) is a testing technique based on the assumption that errors 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 [35], boundary values contain minimum, maximum, internal and external values specified for a system.

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. [36] have provided evidence in support of better performance of BVA compared to partition testing. However, they have indicated that better performance of BVA is based on accurate identification of partition and selection of boundary values. The following code illustrates the ability of BVA to find a bug.

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

2.4.2.4 Formal Specification Testing

It is a testing technique based on mathematical model which provides the opportunity to handle the specifications mechanically. This feature facilitates the isolation, transformation, assembly and repackaging of the information available in the specifications for use as test cases [37].

The formal specification testing is more productive because of the creation of test cases independent from the code of the SUT [38]. The extra effort of generating test oracle is avoided because of using the available specification model for verifying the test results [39].

2.4.3 Test Case

A test case is an artifact which delineates the input, action and expected output corresponding to that input [40]. After executing the test case, if the output obtained comply with the expected output, the test case is declared pass which means that the functionality is working correctly, otherwise the test case is declared fail, which represents identification of fault. A series of test cases, also known as test suite, are usually required to be executed for establishing the desired level of quality.

2.4.4 Test Oracle

Test oracle is defined as, “a source containing expected results for comparison with the actual result of the SUT” [40]. 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 [13]. Test oracle sets the acceptable behaviour for test execution [41].

All software testing techniques depend on the availability of test oracle [38]. Designing test oracle for ordinary software may be simple and straightforward. However, for relatively complex software, designing of oracle is quite cumbersome and requires special ways to overcome the oracle problem. Some of the common oracle problems are as follows:

1. It is assumed that the test results are observable and comparable with the oracle [42].
2. Ideally, test oracle would satisfy desirable properties of program specifications [41].
3. A test oracle to satisfy all conditions is seldom available as rightly pointed out by Weyuker, “truly general test oracles are often unobtainable” [42].

Postconditions of a method are the most commonly used test oracle in automated software testing. Postconditions are conditions which must be true after a method is executed. In such oracle a fault is signalled when a postcondition is violated [43]. Some common artefacts used as oracles are as follows.

1. Specification and documentation to generate test oracle.
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 to another for consistency.
6. Models to generate test oracle for verification of SUT behaviour.
7. Manual analysis by human experts to verify the test results.

2.5 Forms of Software Testing

There are two forms of software testing: static and dynamic.
In static testing, test cases are analysed statically for checking errors without test execution. In addition to software code, high quality software are supplied with documentation including requirements, design, user manual, technical notes and

marketing information. Reviews, walkthroughs and inspections are most commonly used techniques for static testing.
In dynamic testing the software code is executed and input is converted into output. Results are analysed against expected outputs to find any error in the software. Unit testing, integration testing, system testing, and acceptance testing are most commonly used dynamic testing methods [44].

2.5.1 Manual Software Testing

Manual testing is the technique in which the tester writes the code manually to create test cases and test oracles for finding faults in software [45]. It may be effective at smaller scale but is generally laborious, time consuming and error-prone [46]. Additionally, testers must have appropriate skills, experience and knowledge of the SUT for evaluation from different perspectives.

2.5.2 Automated Software Testing

Automated testing is the technique in which a testing tool is used to perform the testing process automatically for finding faults in software [47]. There are tools for automating a part of testing process e.g. generation of test cases, execution of test cases, evaluation of results. Other tools are available for automating the whole testing process. 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 is performed on routine basis like unit testing and regression testing where the tests are re-executed after each modification [48]. The use of automated software testing makes it possible to test large volumes of code, which may be impossible otherwise [49].

2.6 Test Data Generation

Test data generation in software testing is the process of identifying test input data which satisfies the given test selection criterion. A test data generator is used to assist testers in the generation of test data while the test selection criterion defines the properties of test cases to be generated based on the test plan and perspective taken [16]. 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 [50].
Program analyser performs initial assessment of software prior to testing and may alter the code if so required. For example, it performs code instrumentation or construction of CFG to measure the code coverage during testing.
A Strategy handler defines the test case selection criteria. This may include the formalisation of test coverage criterion, the selection of paths, normalisation of constraints, etc. It may also get input from program analyser or user before or during execution.
The generator takes inputs from the program analyser & strategy handler and generates test cases according to the set selection criteria.
Test data generators, based on their approaches, are classified into path-wise, goal-oriented, intelligent, random and search-based as shown in Figure 2.5. Each type is briefly described in the following section.

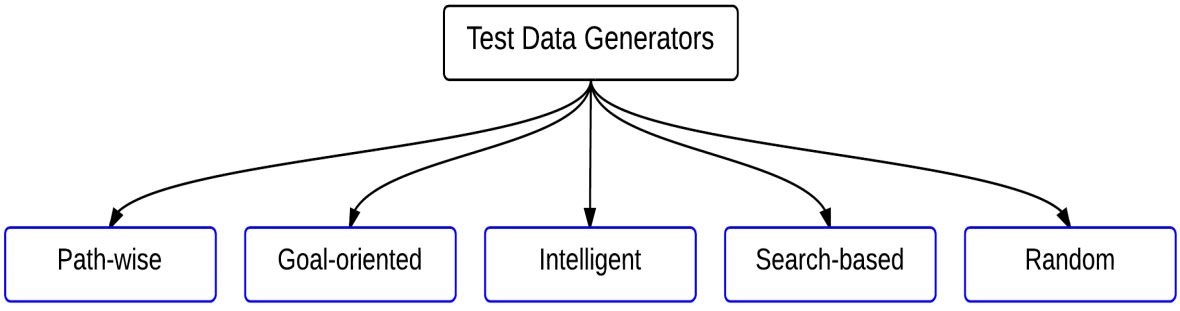


Figure 2.5: Types of test data generators

2.6.1 Path-wise Test Data Generators

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

In path-wise test data generation, the program path to the selected statement is identified and the input data are generated automatically or provided by the user for evaluating the path. The data generated in path testing expresses boolean behaviour i.e. true or false for a particular node in a path. A complete path contains multiple sub-domains, each sub-domain consists of test inputs required to traverse the path. The boundary of the sub-domains are obtained by the predicates in the path condition. The test data traversing a certain path in the software are selected from an input space split into a set of sub-sections.

2.6.2 Goal-oriented Test Data Generators

It is a technique in which the test data is generated to target a specific program point rather than a program path [51]. 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 [52]. Among various methods used in goal-oriented test data generation the following two commonly adopted approaches are briefly described.

2.6.2.1 Chaining Approach

The chaining approach uses data dependent analysis to guide the test data generation. In the process all the related statements affected by execution of the statement under test are selected automatically. The dependant statements are executed before the selected statement to generate the required necessary data for the execution of the statement under test [52]. 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.

2.6.2.2 Assertion-oriented Approach

In this approach assertions are added to the program code with the goal to identify program input on which an assertion is violated, indicating a fault in the SUT. An assertion is a constraint that applies to some 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 code, in assertion and in pre or post-condition.

2.6.3 Intelligent Test Data Generators

Intelligent test data generation is a technique used to overcome the problems associated with traditional data generation techniques like generation of meaningless data, duplicated data and failing to generate complex test data. The approach increases users confidence in the generated test data and the testing process [49]. It performs sophisticated analysis, such as fuzzy logic, neural networks and genetic algorithms on the SUT to assist in finding the appropriate test data. It involves complex analysis to anticipate different situations that may arise at any point. The approach produces test data which satisfy the SUT requirements. However, this approach consumes more time and resources.

2.6.3.1 Genetic Algorithm

Genetic algorithm is a heuristic that mimics the evolution of natural species in searching for the optimal solution of a problem. The solution sought by the genetic algorithm is the test data that cause execution of a given statement, branch, path and condition in the SUT. The genetic algorithm is guided by control dependencies in the program to search for test data which satisfy test requirements. The genetic algorithm constructs new test data from previously generated test data. The algorithm evaluates the existing test data, and guide the direction of search by using the program control-dependence graph [53].

The benefit of the genetic approach is quick generation of test data with focus and direction. New test cases are generated by applying simple operations on existing test cases that are judged to have good potential of satisfying the test requirements. The success of this approach, however, depends heavily on the way in which the existing test data is measured [53].

2.6.4 Search-based Test Data Generation

It is a technique which uses meta-heuristic algorithms to generate test data. In Search-based test data generation technique each input vector `x` can be associated with a measure $cost(x)$ that represents how far the input value `x` is from satisfying the set goal. Input value closer to the set goal have low cost value and the other have high cost value.

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

Consider the above program and 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` can be seen to come closer to satisfying the predi-

cate than a value of `x ==5`. We might evaluate a cost function probe (immediately before the indicated statement) 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`. We can see how finding data to satisfy the branch predicate is essentially a search over the input domain of `x` to find a value such that `cost(x) == 0`.

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

2.6.5 Random Test Data Generators

Random test data generator is the simplest technique for generation of test data. It has the advantage of being used to generate input data for any type of program. However, random test data generation is based solely on probability and cannot accomplish high coverage as its chances of finding semantically small faults are quite low [18]. 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 easy to see that the probability of execution of the first statement is significantly lower than that of the second statement. As the structure gets complex so does the probability of its execution. Thus, such semantically small faults are hard to

find by using random test data generation.

2.7 Random Testing

Random testing was first mentioned in the literature by Hanford in 1970. He reported syntax machine, a tool that randomly generated data for testing PL/I compilers [54]. Later in 1983, Bird and Munoz described a technique to produce randomly generated and self checking test cases [55].

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 [56]. As stated by Richard [57], in random testing, input domain is first identified, then test data are randomly taken from it by means of random generator. The program under test is executed on the test data and the results obtained are compared with the program specifications. The test fails if the results are not according to the specifications and vice versa. Fail results of the test cases reflects failure in the SUT.

Figure 2.6: Random Testing

Generating test data by random generator is quite economical and requires less intellectual and computational efforts [58]. Moreover, no human intervention is involved in data generation which ensures an unbiased testing process. However, generating test cases with out using any background information makes random testing susceptible to criticism. Random testing is criticized for generating many of the test cases that falls at the same state of software. It is also stated that, random testing generates test inputs that violates requirements of the given SUT making it less effective [59, 60]. Myers mentioned random testing as one of the least effective testing technique [11]. However, Ciupa et al. stated [17], that Myers statement was not based on any experimental evidence. Later experiments performed by several researchers [45, 57, 61, 62] confirmed that random testing is as effective as any other testing technique. It is reported [62] that random testing can

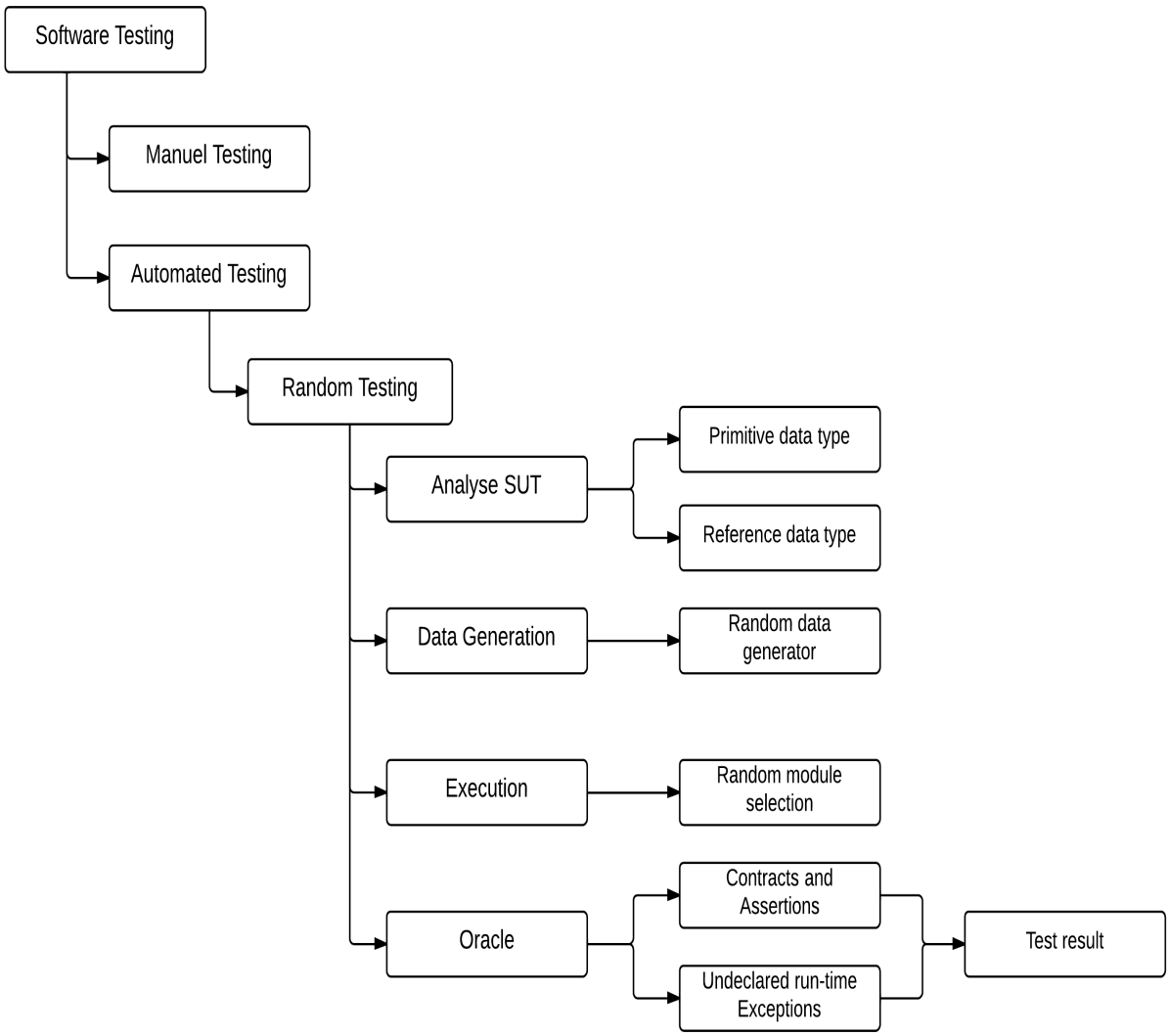


Figure 2.7: Hierarchy and working of Random testing

also discover subtle faults in a given SUT when subjected to large number of test cases. It is pointed out that the simplicity and cost effectiveness of random testing makes it more feasible to run large number of test cases as opposed to systematic testing techniques which require considerable time and resources for test case generation and execution. The empirical comparison proves that random testing and partition testing are equally effective [32]. A comparative study conducted by Ntafos [34] concluded that random testing is more effective as compared to proportional partition testing. A prominent work to mention is that of Miller et al. [63], who generated and used random ASCII character streams to test Unix utilities for abnormal termination or non-terminating behaviour. Subsequently the same tech-

nique was extended to discover errors in software running on X Windows, Windows NT and Mac OS X [64, 65]. Other famous studies using random testing includes low-level system calls [66], and file systems used in missions at NASA [67].

2.8 Input Domain

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

2.8.1 Genuine and Failure-domain

Chan et al. observed that failure inducing inputs are contagious and form certain geometrical shapes in the whole input domain which were divided into point, block and strip failure-domains as described below.

1. **Point:** In the point failure-domain, inputs 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:** In the block failure-domain, inputs 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:** In the strip failure-domain, inputs 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.

In Figure 2.8 the three square boxes indicate the whole input domains. The white space in each box shows legitimate and faultless values while the black colour in

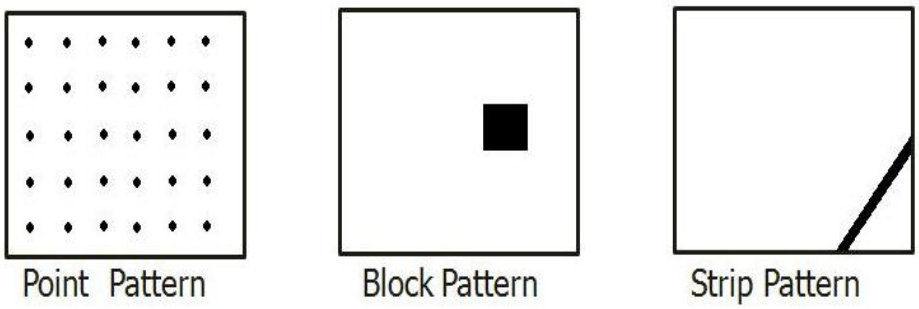


Figure 2.8: Domains of failure causing inputs [1]

the form of points, block and strip, inside the respective boxes indicates the failures in the form of point, block and strip failure-domains.

2.9 Versions of Random testing

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

2.9.1 Random+ Testing

The random+ testing [47] 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 [68] are the values on the start and end of a particular type. For instance, such values for `int` could be `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 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 faults for the SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the pre-defined list of special values. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [45].

2.9.2 Adaptive Random Testing

Adaptive random testing (ART), proposed by Chen et al. [69] is based on the previous work of Chan et al. [1] regarding the existence of failure domains across the input domain (Section 2.8.1).

Chen et al. [69] argued that ordinary random testing might generate test inputs lurking too close or too far from the input inducing failure and thus fails to discover the fault. To generate more fault-targeted test inputs, they proposed Adaptive Random Testing (ART) as a modified version of random testing where test values are selected at random as usual but are evenly spread across the input domain by using two sets. The executed set comprises the test cases and the candidate set includes the test cases to be executed by the system. Initially both the sets are empty. The first test case is selected at random from the candidate set and stored in executed set after execution. The second test case is then selected from the candidate set which is far away from the last executed test case. In this way the whole input domain is tested with greater chances of generating test input from the existing fault patterns.

In the experiments conducted by Chen et al. [69], the number of test cases required to detect first fault (F-measure) was used as a performance matrix instead of the traditional matrices (P-measure) and (E-measure). Experimental results us-

ing ART showed up to 50% increase in performance compared to random testing. The authors pointed out that the issues of increase overhead, spreading test cases across the input domain for complex objects and efficient ways of selecting candidate test cases still exist. Chen et al. continued their work on ART to address some of these issues and proposed its upgraded versions [69, 70].

2.9.3 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) [2] is an improvement on ART by using mirror-partitioning technique to reduce the overhead by decreasing the extra computation involved in ART.

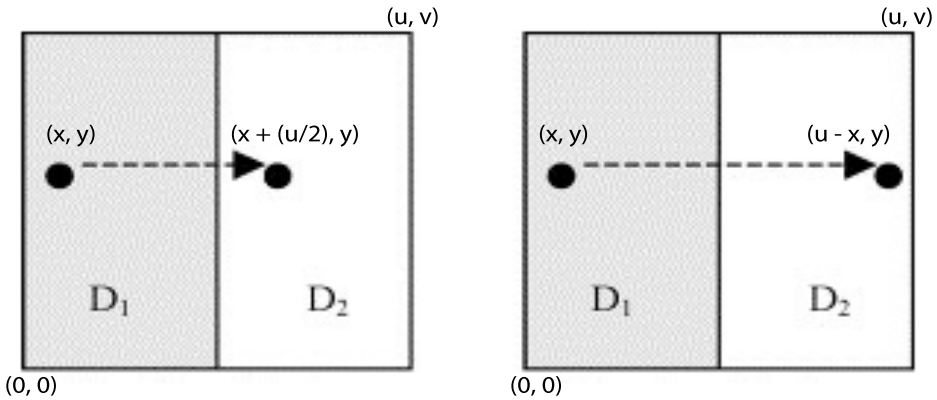


Figure 2.9: Mirror Adaptive Random Testing [2]

In this technique, the input domain of the program under test is divided into n disjoint sub-domains of equal size and shape. One of the sub-domains is called source sub-domain while all others are termed as mirror sub-domains. ART is then applied only to the source sub-domain while test cases are selected from all other sub-domains by using mirror function. In MART `(0, 0), (u, v)` are used to represent the whole input domain where `(0, 0)` is the leftmost and `(u, v)` is the rightmost top corner of the two dimensional rectangle. On splitting it into two sub-domains we get `(0, 0), (u/2, v)` as source sub-domain and `(u/2, 0), (u, v)` as mirror sub-domain. Suppose we get `x` and `y` test cases by applying ART to source sub-domain, now we can linearly translate these test cases to achieve the mirrored effect, i.e. `(x` (u/2), y)+ as shown in the Figure 2.9.

Comparative study of MART with ART provide evidence of equally good results of the two strategies with MART having the added advantage of lower overhead by using only one quarter of the calculation as compared with ART.

2.9.4 Restricted Random Testing

Restricted Random Testing [56] is another approach to overcome the problem of extra overhead in ART. Restricted Random Testing (RRT) achieves this by creating a circular exclusion zone around the executed test case. A candidate is randomly selected from the input domain for the next test case. Before execution the candidate is checked and discarded if it lies inside the exclusion zone. This process repeats until a candidate laying outside the exclusion zone is selected. This ensures that the test case to be executed is well apart from the last executed test case. The radius of exclusion zone is constant in each test case and the area of input domain decreases progressively with successive execution of test cases.

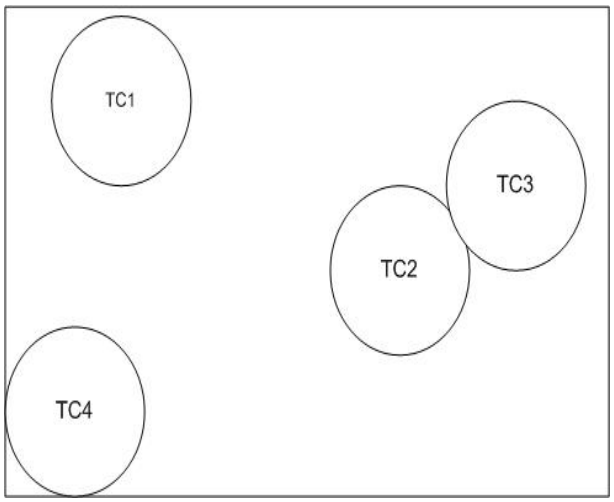


Figure 2.10: Input domain with exclusion zone around the selected test case

The above authors compared RRT with ART and RT to find the comparative performance and reported that the performance of RRT increases with the increase in the size of the exclusion zone and reaches the maximum level when the exclusion zone is raised to largest possible size. They further found that RRT is up to 55% more effective than random testing in terms of F-measure.

2.9.5 Directed Automated Random Testing

Godefroid et al. [18] proposed Directed Automated Random Testing (DART). The following main features of DART are reported in the literature:

1. **Automated Interface Extraction:** DART automatically identifies external interfaces of a given SUT. These interfaces include external variables and methods and the user-specified main method responsible for program execution.
2. **Automatic Test Driver:** DART automatically generate test drivers for running the test cases. All the test cases are randomly generated according to the underlying environment.
3. **Dynamic Analysis of execution:** DART instrument the given SUT at the start of the process in order to track its behaviour dynamically at run time. The results obtained are analysed in real time to systematically direct the test case execution along alternative path for maximum code coverage.

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

2.9.6 Quasi Random Testing

Quasi-random testing (QRT) [71] is a testing technique which takes advantage of failure region contiguity for distributing test cases evenly and thus decreases computation. To achieve even spreading of test cases, QRT uses a class with a formula that forms an s-dimensional cube in s-dimensional input domain and generates a set of numbers with small discrepancy and low dispersion. The set of numbers is then used to generate random test cases that are permuted to make them less clustered and more evenly distributed. An empirical study was conducted to compare the effectiveness of QRT with ART and RT. The results showed that in 9 out

of 12 programs QRT found a fault quicker than ART and RT while there was no significant improvement in the remaining three programs.

2.9.7 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) is a technique that generates unit test suite at random for object-oriented programs [5]. As the name implies FDRT uses the feedback received from the execution of first batch of randomly selected unit test suite to generate next batch of directed unit test suite. In this way redundant and wrong unit tests are eliminated incrementally from the test suite with the help of filtration and application of contracts. For example unit test that produce IllegalArgumentException on execution is discarded, because, selected argument used in this test is not according to the type of argument the method required.

2.9.8 The Artoo Testing

The Adaptive random testing for object oriented (Artoo) strategy is based on object distance. Ciupa et al. [72] defined the parameters that can be used to calculate distance between the objects. Two objects have more distance between them if they have more dissimilar properties. The parameters to specify the distance between the objects are dynamic types and values are assigned to the primitive and reference fields. Strings are treated in terms of directly usable values and Levenshtein formula [73] is used as a distance criterion between the two strings.

In the Artoo strategy, two sets are taken i.e. candidate-set containing the objects ready to be run by the system and the used-set, which is empty. First object is selected randomly from the candidate-set which is moved to used-set after execution. The second object selected from the candidate-set for execution is the one with the largest distance from the last executed object present in the used-set. The process continues till the bug is found or the objects in the candidate-set are finished [72].

The Artoo strategy, implemented in AutoTest [58], was evaluated in comparison with Directed Random (D-RAN) strategy by selecting classes from EiffelBase library [74]. The experimental results indicated that some bugs found by the Artoo were not identified by the D-RAN strategy. Moreover the Artoo found first bug with

small number of test cases than the D-RAN strategy. However, computation required to select test case in the Artoo strategy was more than the D-RAN strategy and took more time and cost to generate a test case.

.

2.10 Tools for Automated Random Testing

A number of open-source and commercial automatic random testing tools reported in the literature are briefly described in the following section.

2.10.1 JCrasher

JCrasher is an automatic robustness testing tool developed by Csallner and Smaragadakis [3]. JCrasher tries to crash the Java program with random input and exceptions thrown during the process are recorded. The exceptions are then compared with the list of acceptable standards, defined in advance as heuristics. The undefined runtime exceptions are considered as errors. Since users interact with programs through its public methods with different kinds of inputs, therefore, JCrasher is designed to test only the public methods of the SUT with random inputs.

The working of JCrasher is illustrated by testing a `.java` program as shown in the Figure 2.11. The source file is first compiled using `javac` and the byte code obtained is passed as input to JCrasher which uses Java reflection library [75] 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 the exceptions produced during test case executions are collected and compared with robustness heuristic for any violation and reported as errors.

JCrasher is a pioneering tool with the capability to perform fully automatic testing, including test case generation, execution, filtration and report generation. JCrasher has the novelty to generate test cases as JUnit files which can also be easily read and used for regression testing. Another important feature of JCrasher is to execute each new test on a “clean slate” ensuring that the changes made by the previous tests do not affect the new test.

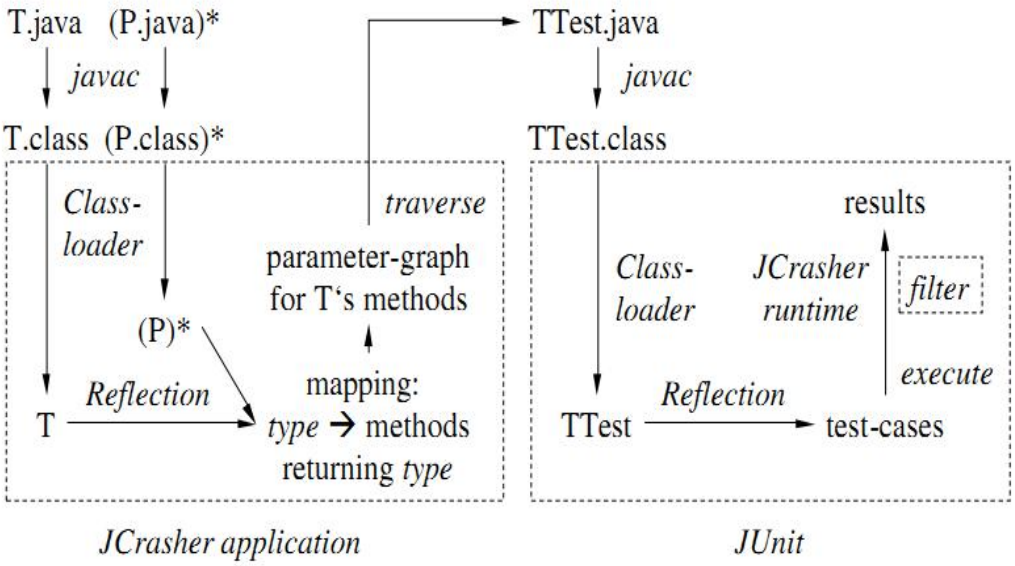


Figure 2.11: Illustration of robustness testing of Java program with JCrasher [3]

2.10.2 Jartege

Jartege (Java random test generator) is an automated testing tool [76] that randomly generates unit tests for Java classes with contracts specified in JML. The contracts include, methods pre and post-conditions and class invariants. Initially Jartege uses the contracts to eliminate irrelevant test cases and later on the same contracts serve as test oracle to differentiate between errors and false positives. Jartege uses simple random testing to test classes and generate test cases. In addition, it parametrise its random aspect in order to prioritise testing a specific part of the class or to get interesting sequences of calls. The parameters include the following:

- Operational profile of the classes i.e. the likely use of the class under test by other classes.
- Weight of the class and method under test. Higher weight prioritizes the class or method over lower weight during test process.
- Probability of creating new objects during test process. Low probability means creation of fewer objects and more re-usability for different operations while high probability means numerous new objects with less re-usability.

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

2.10.3 Eclat

Eclat [4] is an automated testing tool which generates and classifies unit tests for Java classes. The process is accomplished in three stages. In the first stage, it selects a small subset of test inputs that are likely to reveal faults in the given SUT.

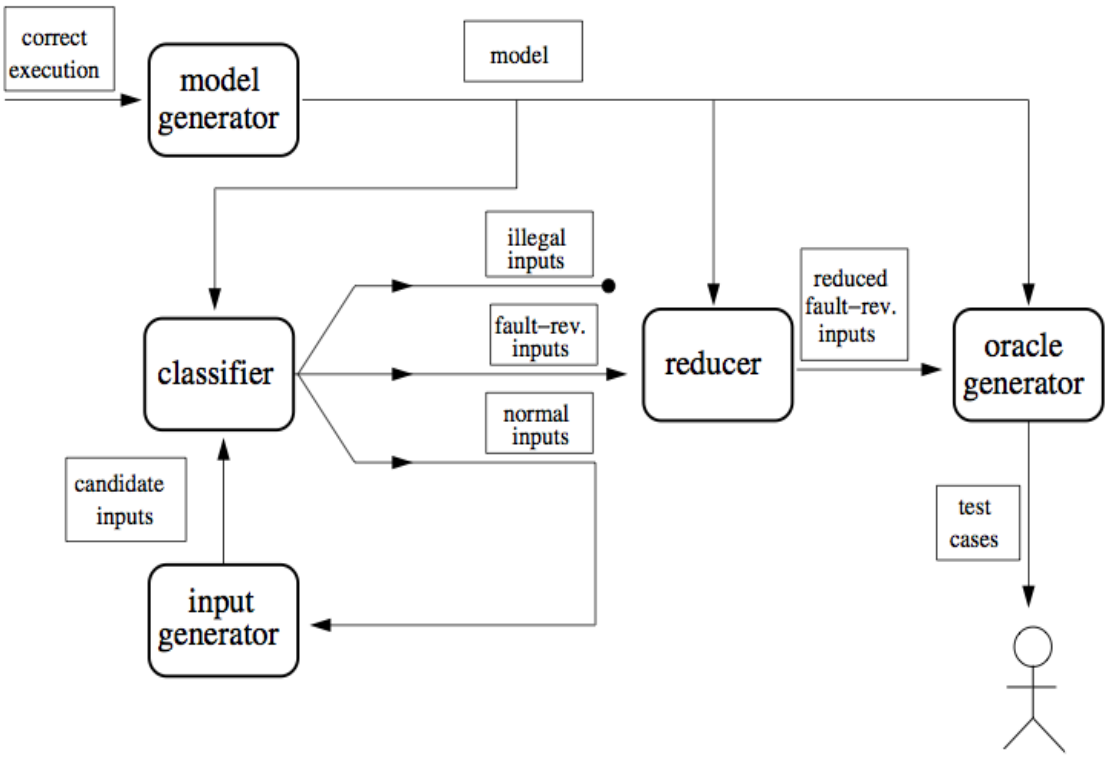


Figure 2.12: Main component of Eclat contributing to generate test input [4]

The tool takes a software and a set of test cases for which the software runs properly. It creates an operational model, based on the correct software operations,

and apply the test data. If the operational pattern of execution of the test data differs from the model, the following three outcomes may be possible: (a) a fault in the given SUT (b) model violation despite normal operation (c) illegal input which the program is unable to handle. In the second stage, reducer function is used to discard any redundant input, leaving only a single input per operational pattern. In the third stage, the acquired test inputs are converted into test cases and oracles are created to determine the success or failure of the test.

Csallner and Smaragdakis 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. Existence of errors in the pool leads to the creation of wrong operational model which adversely affects the testing process [5].

2.10.4 Randoop

Random tester for object oriented programs (Randoop) is the tool used for implementing FDRT technique [5]. Randoop is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes as input a set of classes, contracts, filters and time limit and gives out 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 a public method from the class under test and arguments for these methods are selected from the predefined pool in case of primitive types and a sequence or null value in case of reference type. Randoop maintains two sets called `ErrorSeqs` and `NonErrorSeqs` to record the feedback. It extends `ErrorSeqs` set in case of contract or filter violation and `NonErrorSeqs` set when no violation is recorded in the feedback. The use of this dynamic feedback evaluation at runtime brings an object to an interesting state. On test completion, `ErrorSeqs` and `NonErrorSeqs` are produced as JUnit/NUnit test suite. In terms of coverage and number of faults discovered, Randoop implementing FDRT was compared with JCrasher and JavaPathFinder and 14 libraries of both Java and .Net were evaluated [77]. The results showed that Randoop achieved more coverage than JCrasher in branch coverage and faults detection.

2.10.5 QuickCheck

QuickCheck [78] is a lightweight random testing tool used for testing of Haskell programs [79]. Haskell is a functional programming language where programs are evaluated by using expressions rather than statements as in imperative programming. In Haskell most of the functions are pure except the IO functions, thus main focus of the tool is on testing pure functions. QuickCheck 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 inputs is the function to be tested and properties of the program defined by tester (Haskell functions). The tool uses built-in random generator to generate test data, however, it also provides the facility to use custom build data generator. On executing the function with test data, the tester-defined-properties must hold for the function to be correct. Any violation of the defined properties suggest error in the function.

2.10.6 AutoTest

The AutoTest, based on formal automated testing is used to test Eiffel language programs [5]. The Eiffel language uses the concept of contracts which is effectively utilized by AutoTest. For example, the auto generated inputs are filtered using preconditions and non complying test inputs are discarded. The contracts are also used as test oracle to determine if the tests are pass or fail. Beside automated testing the AutoTest also allows the tester to manually write the test cases to target specific section of the code. The AutoTest takes one or more methods/classes as inputs and automatically generate test input data according to the requirements of the methods or classes.

According to Figure 2.13, the architecture of AutoTest can be split into the following main parts:

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

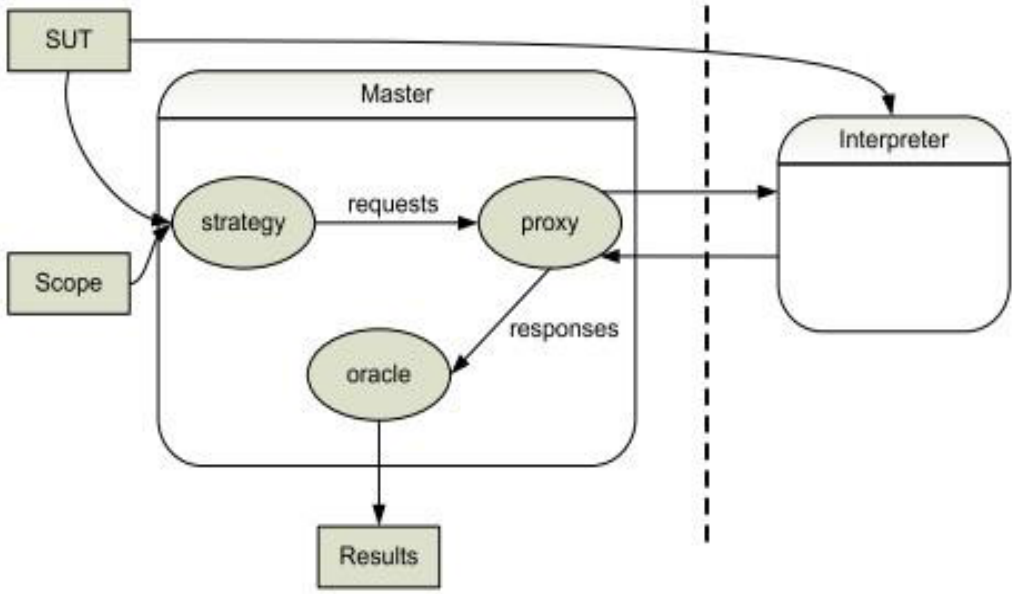


Figure 2.13: Architecture of Autotest [5]

2. **Proxy:** It handles inter-process communication. It receives execution requests from the strategy and forward them to the interpreter. It also sends the execution results the oracle part.
3. **Interpreter:** It execute 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 evaluate the results to see if the contracts are satisfied. The outcome of the tests are formatted in HTML and stored on disk.

2.10.7 TestEra

TestEra [6] is a novel framework for auto generation and evaluation of test inputs for a Java program. It takes methods specifications, numerical value and the method to test as input. It uses pre-conditions of a method to generate all non isomorphic valid test inputs to the specified limit. The test inputs are executed on the method and the results are compared against the 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 [80] to express constraints on test in-

puts and Alloy Analyser [81] to solve these constraints and generate test inputs. Alloy Analyzer performs the following three functions: (a) it translates Alloy predicates into propositional formulas, i.e. constraints where all variables are boolean (b) it evaluates the propositional formulas to find the outcome (c) it translates each outcome from propositional domain into the relational domain.

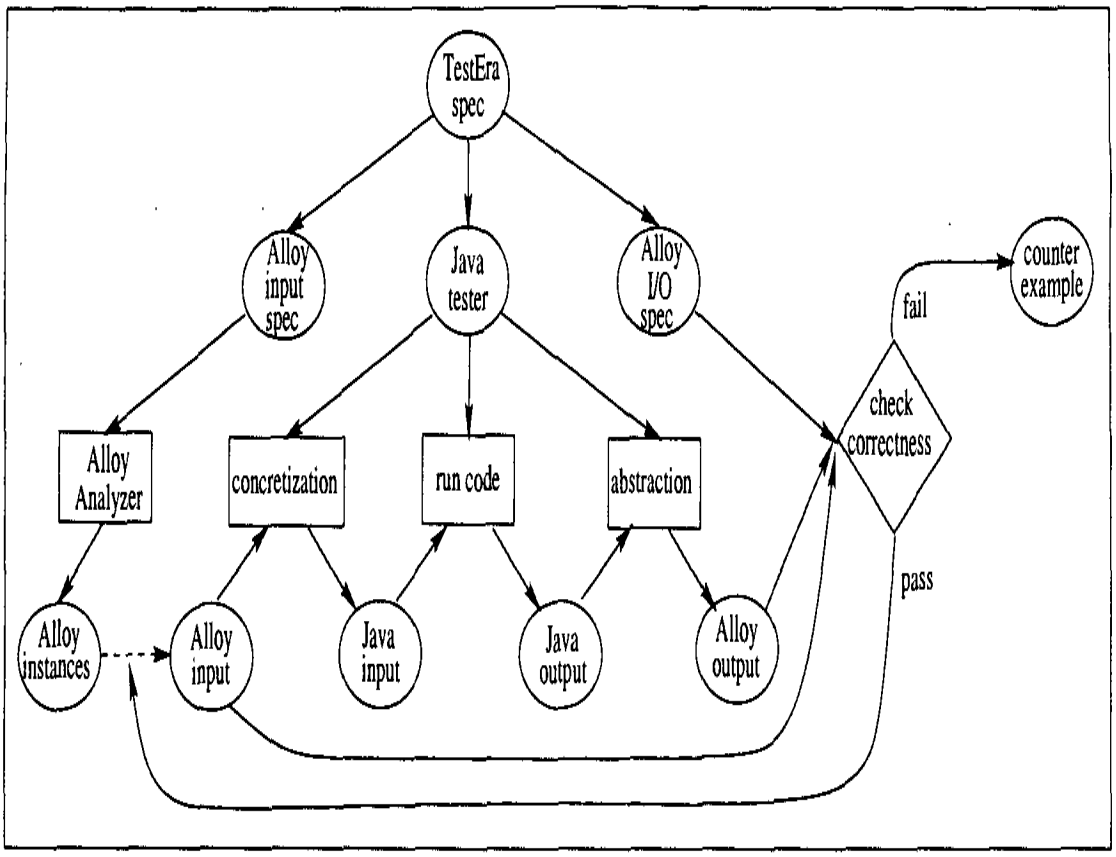


Figure 2.14: Architecture of TestEra [6]

TestEra uses program specifications to guide the auto generation of test inputs. However, in Jartege (Section 2.10.2) and AutoTest (Section 2.10.6) specifications are used to filter the irrelevant random generated test data. All the tools uses specifications in a similar way for oracle.

2.10.8 Korat

Korat [82] is a novel framework for automated testing of Java programs based on the formal specifications [83]. Korat and TestEra [6] were developed by the same

team and perform specification based testing. The difference however is that Korat uses Java Modelling Language (JML) while TestEra uses Alloy Modelling Language (AML) for specifications. Moreover, Korat uses bounded-exhaustive testing in which the code is tested against all possible inputs within the given small bound [84].

Korat generate structurally complex inputs by solving imperative predicates. An imperative predicate is a piece of code that takes a structure as input and evaluates it to a boolean value. Korat takes imperative predicates and finitization value as inputs. It systematically explores the predicates input space and generates all non-isomorphic inputs for which the predicates return true. The core part of Korat monitors execution of the predicates on candidate inputs to filter out the fields accessed during executions. These inputs are taken as test cases. Korat depends or developers written `repOK()` and `checkRep()` methods, where `repOK()` is used to check the class invariants and `checkRep()` is used to verify the post-conditions to validate the correctness of the test case.

The key benefit of Korat and TestEra, representation level approaches, is that no existing set of operations are required to create input values. Therefore, they can create input values that may be difficult or impossible to create using a given set of operations. The disadvantage of the approaches however is the requirement of significant amount of manual efforts [59].

2.11 Summary

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

Chapter 3

York Extensible Testing Infrastructure

3.1 YETI Overview

York Extensible Testing Infrastructure (YETI), an automated random testing tool developed in Java, is capable of testing programs written in Java, JML and .NET languages [85]. YETI takes program byte code as input and execute it with random generated but syntactically-correct inputs to find a fault. It runs at a high level of performance with 10^6 calls per minute on Java code. One of its prominent feature is Graphical User Interface (GUI), which make YETI user friendly and provides option to change testing process in real time. It can also distribute large testing tasks in cloud for parallel execution [90]. The latest version of YETI can be downloaded from `https://code.google.com/p/yeti-test/downloads/list`. Figure 3.1 briefly presents the working process of YETI.

3.1.1 YETI Design

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

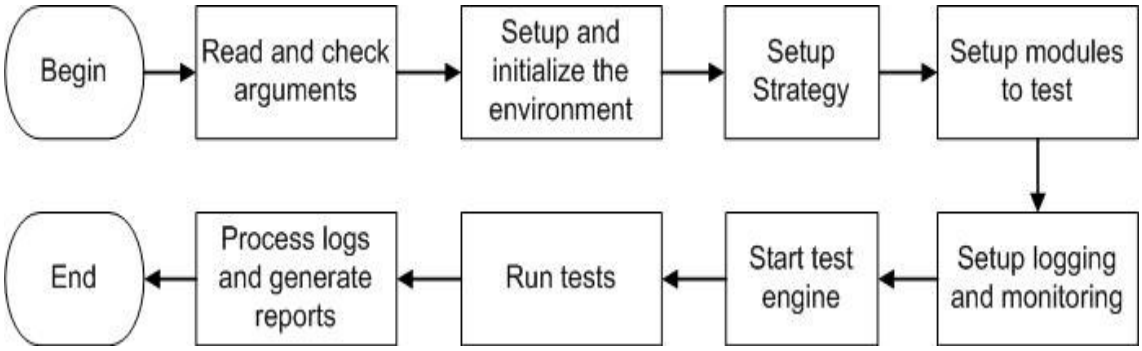


Figure 3.1: Working process of YETI

part is briefly described below.

3.1.1.1 Core Infrastructure

The core infrastructure is responsible for test data generation, test process management and test report generation. The core infrastructure is split into four packages: yeti, yeti.environments, yeti.monitoring, yeti.strategies. The package yeti uses classes from yeti.monitoring and yeti.strategies packages and calls classes in the yeti.environment package as shown in the Figure 3.2.

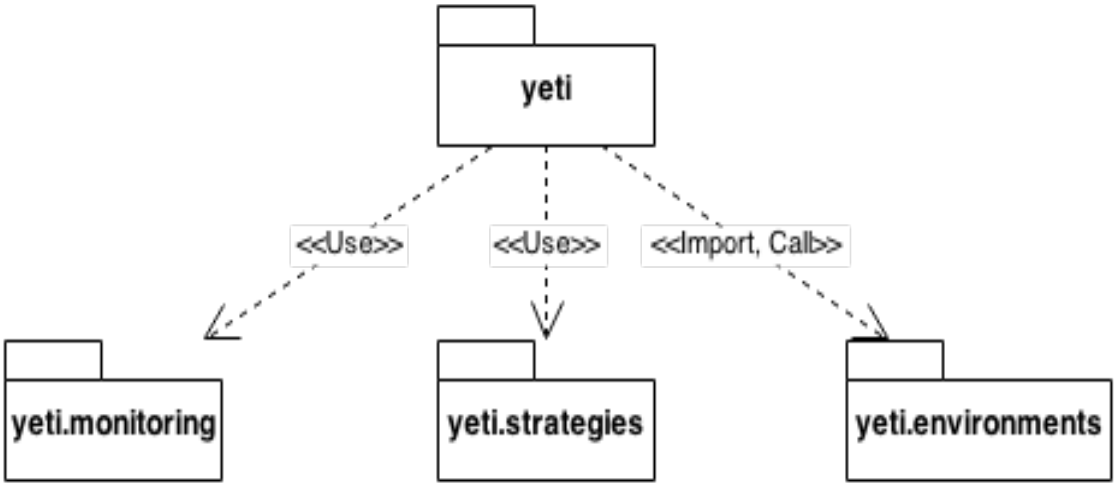


Figure 3.2: Main packages of YETI with dependencies

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.

3.1.1.2 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 fault value. The strategy is described in detail in Chapter 4.
5. **ADFDStrategy:** It extends random+ strategy by adding the feature of graphical representation of faults and their domains. The strategy is described in detail in Chapter 5.
6. **YetiRandomDecreasingStrategy:** 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.

3.1.1.3 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.

3.1.2 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.

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

3.1.4 YETI 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 3.3.

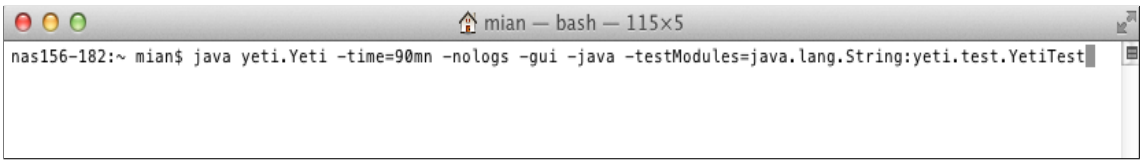


Figure 3.3: Command to launch YETI from CLI

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 3.4 has been created for use in the present study.

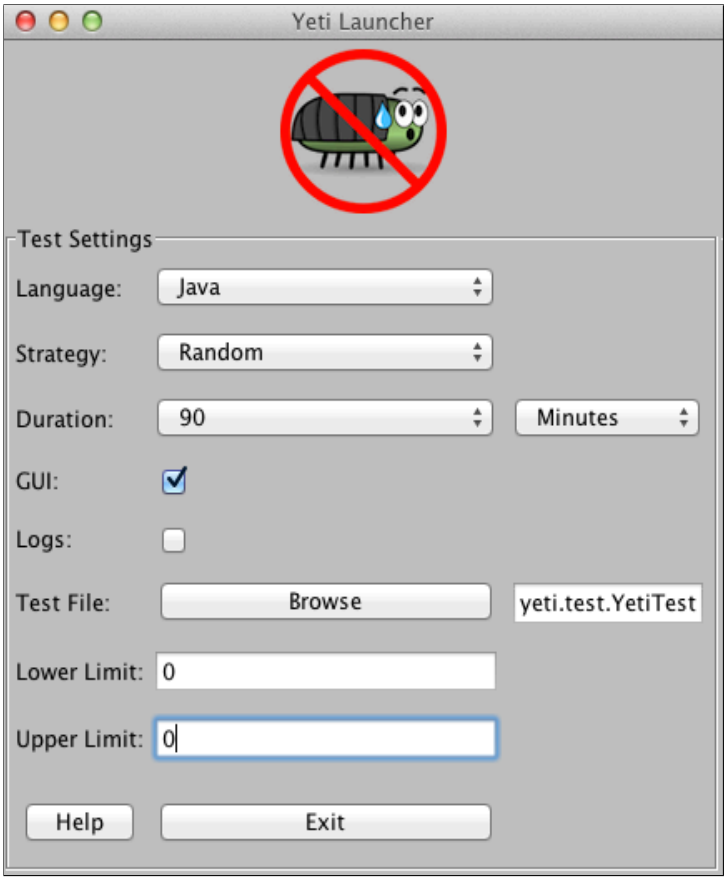


Figure 3.4: GUI launcher of YETI

3.1.5 YETI 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.

3.1.6 YETI 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 3.5: 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 3.6. 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 3.6: A sample of YETI bug report

3.1.7 YETI 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 3.7 present the YETI GUI comprising of thirteen labelled components.

1. Menu bar:

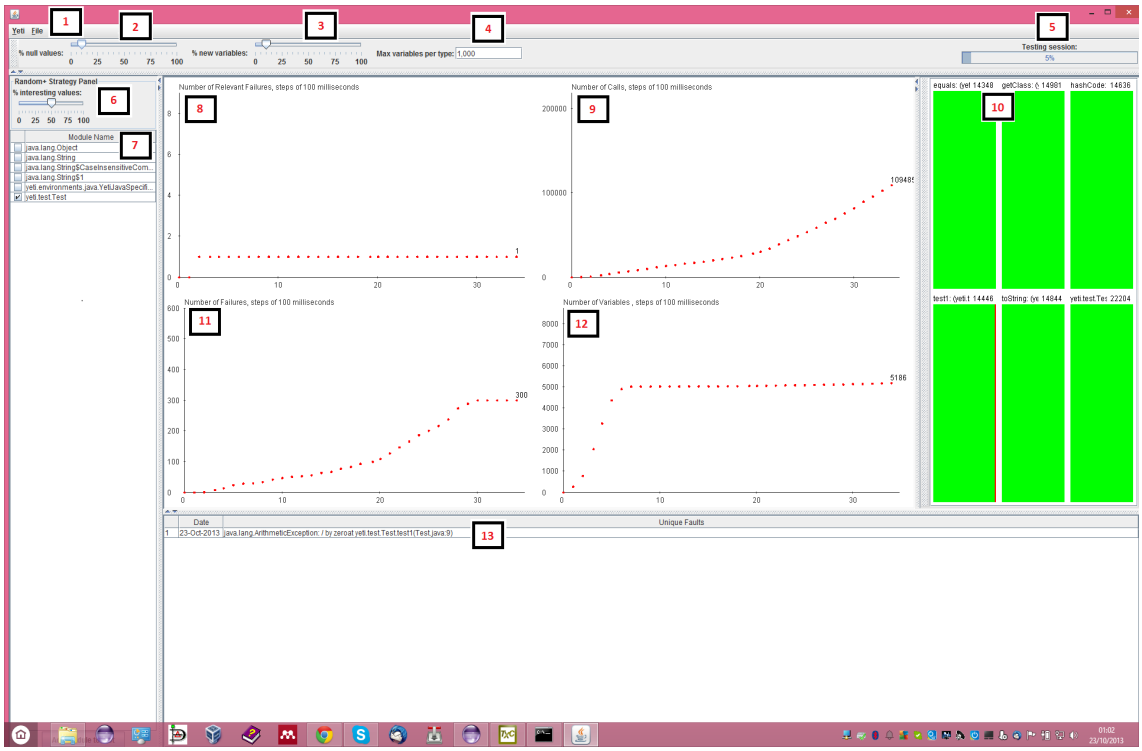


Figure 3.7: GUI of YETI

(a) Yeti menu:

(b) File menu:

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

3.1.8 Summary

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

Table 3.1: YETI command line options

Options	Purpose
-java	To test Java programs
-jml	To test JML programs
-dotnet	To test .NET programs
-ea	To check code assertions
-nTests	To specify number of tests
-time	To specify test time
-testModules	To specify one or more modules to test
-rawlogs	To print real time logs during test
-nologs	To omit real time logs and print end result
-yetiPath	To specify path to the test modules
-gui	To show test session in GUI
-DSSR	To specify Dirt Spot Sweeping Random strategy
-ADFD	To specify Automated Discovery Failure Domain strategy
-random	To specify Random test strategy
-randomPlus	To specify Random plus test strategy
-nullProbability	To specify probability of inserting null values
-randomPlusPeriodic	To specify Random plus periodic test strategy
-newInstanceProability	To specify probability of inserting new objects

Table 3.2: Main features of automatic testing tools used in random testing

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

Chapter 4

Dirt Spot Sweeping Random Strategy

4.1 Introduction

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

Chan et al. [1] discovered that there are patterns of failure-causing inputs across the input domain. They divided these into point, block and strip patterns on the basis of their occurrence across the input domain. Chen et al. [69] 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 input is selected evenly across the whole input domain. This was mainly attributed to the better distribution of input which increased the chance of selecting inputs from failure patterns. Similarly Restricted Random Testing [56], Feedback Directed Random Test Generation [5], Mirror Adaptive Ran-

dom Testing [2] and Quasi Random Testing [71] 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[1] Random (DSSR) strategy which starts as a random+ 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. As in previous studies [86] we approximate faults with unique failures. Since this strategy is an extension of random testing strategy, it has the full potential to find all unique failures in the program, but additionally we expect it to be faster at finding unique failures, for classes in which failure domains are contiguous, as compared with random (R) and random+ (R+) strategies.

We implemented the DSSR strategy in the random testing tool YETI[2]. To evaluate our approach, we tested 30 times each one of the 60 classes of 32 different projects from the Qualitas Corpus[3] with each of the three strategies R, R+ and DSSR. We observed that for 53% of the classes all three strategies perform equally, for remaining 47% classes, DSSR strategy performs 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 random+ strategy with a dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, faults

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

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

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

and unique failures reside more frequently in contiguous block and strip pattern. 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 Random and the Random+ strategy.

4.2.1 Random Strategy (R)

The random strategy is a black-box testing technique in which the SUT is executed using randomly selected test data. Test results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. In the absence of contracts and assertions the exceptions defined by the programming language are used as test oracles. Because of its black-box testing nature, this strategy is particularly effective in testing software where the developers want to keep the source code secret [87]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [88, 45]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [58]. YETI [89, 90], AutoTest [17, 47], QuickCheck [78], Randoop [91] and Jartege [76] are some of the most common automated testing tools based on random strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [11] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [17, 62, 92, 57, 93] have proved experimentally that random testing is simple to implement, cost effective, efficient and free from human bias as compared to its rival techniques.

Programs tested at random typically fail a large number of times (there are a large number of calls), therefore, it is necessary to cluster failures that likely represent the same fault. The traditional way of doing it is to compare the full stack traces and error types and use this as an equivalence class [17, 94] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

4.2.2 Random+ Strategy (R+)

The random+ strategy [47] is an extension of the random strategy. It uses some special pre-defined values which can be simple boundary values or values that

have high tendency of finding faults in the SUT. Boundary values [68] 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 faults for the SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the pre-defined list of special values. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [45].

4.2.3 Dirt Spot Sweeping (DSS)

Chan et al. [1] found that there are patterns of failure-causing inputs across the input domain. Figure 4.1 shows these patterns for two dimensional input domain. They divided these patterns into three types called point, block and strip patterns. 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 fault patterns if test cases far away from each other are selected. Other researchers [56, 2, 71], also tried to generate test cases further away from one another targeting these patterns and achieved better performance.

The increase in performance indicates that faults more often occur contiguously across the input domain. When test value reveals a fault 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 fault from the same region.

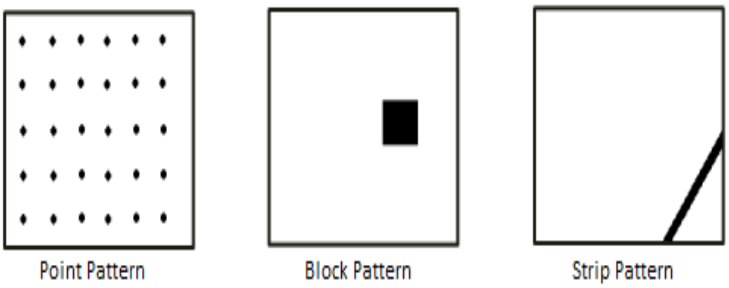


Figure 4.1: Failure patterns across input domain [1]

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

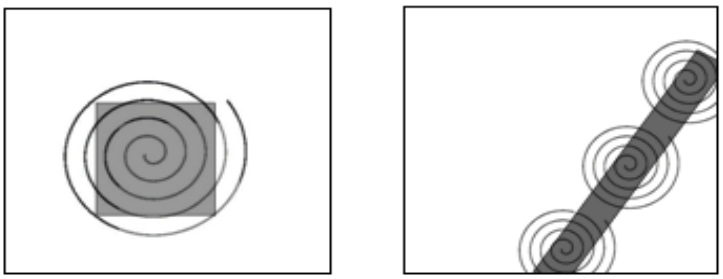


Figure 4.2: Exploration of failures by DSS in block and strip pattern

Figure 4.2 shows how DSS explores the failures residing in the block and strip patterns of a program. The coverage of block and strip pattern is shown in spiral form because first failure leads to second, second to third and so on till the end. In case the failure is positioned on the point pattern then the added values may

not be effective because point pattern is only an arbitrary failure point in the whole input domain.

4.2.4 Structure of the Dirt Spot Sweeping Random Strategy

The DSSR strategy continuously tracks the number of failures during the execution of the test. This tracking is done in a very effective way with zero or minimum overhead [95]. The test execution is started by R+ strategy and continues till a failure is found in the SUT after which the program copies the values leading to the failure as well as the surrounding values to the variable list of interesting values.

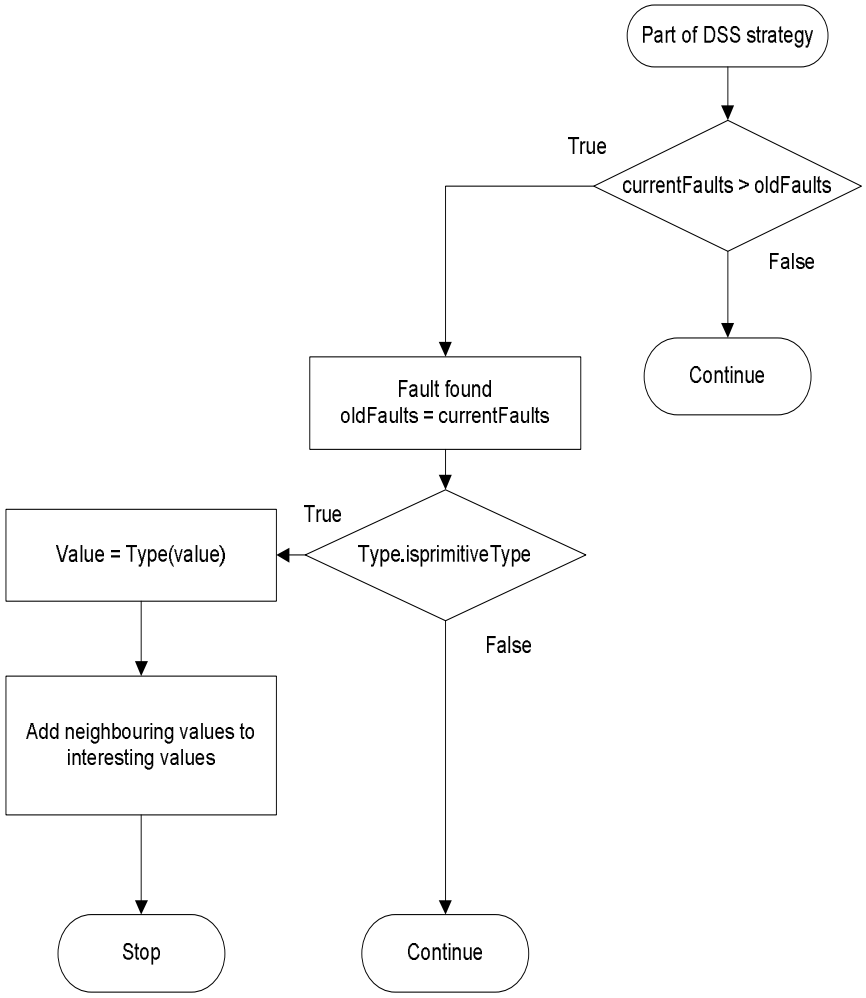


Figure 4.3: Working mechanism of DSSR Strategy

The flowchart presented in Figure 4.3 depicts that, when the failure finding value is

of primitive type, the DSSR strategy identifies its type and add values only of that particular type to the list of interesting values. The resultant list of interesting values 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 faults in the SUT are added to the list of interesting values by random+ strategy prior to the start of test session where as in DSSR strategy the fault-finding and its surrounding values are added at runtime when a failure is found.

Table 4.1 presents the values added to the list of interesting values when a failure is found. In the table the test value is represented by X where X can be $int, double, float, long, byte, short, char$ and $String$. All values are converted to their respective types before adding them to the list of interesting values.

Table 4.1: Neighbouring values for primitive types and String

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

4.2.5 Explanation of DSSR Strategy by Example

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

```
/**
* 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 random and random+ 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.

4.3 Implementation of the 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 Cloud [90]. A number of hitherto faults have successfully been found by YETI in various production software [96, 94].

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 random strategy if no particular strategy is defined during test initialisation. It also enables the user to control the probability of using null values

and the percentage of newly created objects for each test session. YETI provides an interactive Graphical User Interface (GUI) in which users can see 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 4.4.

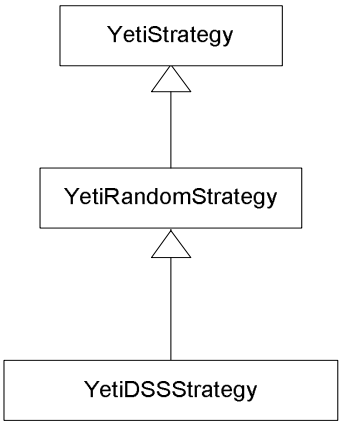


Figure 4.4: Class Hierarchy of DSSR in YETI

4.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [47]. General factors such as system software and hardware, YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability have been kept constant in the experiments.

4.4.1 Research Questions

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

1. Is there an absolute best 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 on software engineering. These projects have been collected in an organised form containing the source and binary forms. 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.[4] The approach similar to that used in previous studies when the contracts and assertions in the code under test are absent [94] was followed in the study. The undeclared exceptions were treated as unique failures.

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

4.4.3 Performance Measurement Criteria

Various measures including the E-measure (expected number of failures detected), P-measure (probability of detecting at least one failure) and F-measure (number of test cases used to find the first fault) have been reported in the literature for finding the effectiveness of random test strategy. The E-measure and P-measure

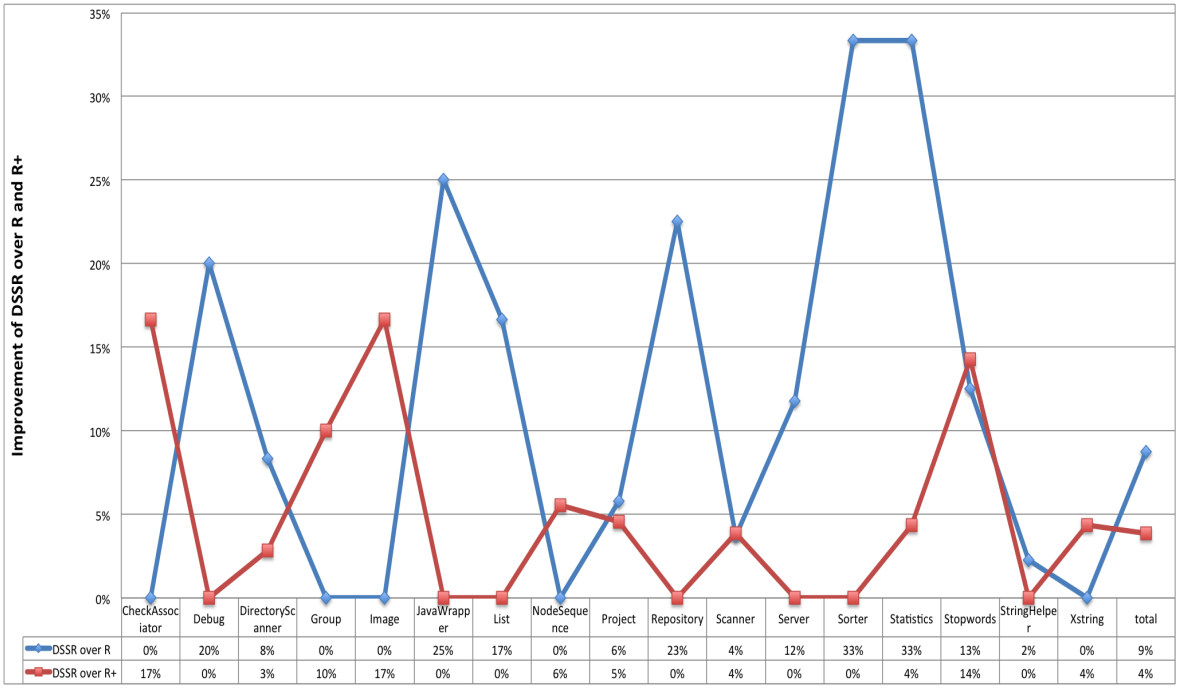
[4] The total number of tests is thus $60 \times 30 \times 3 \times 10^5 = 540 \times 10^6\ tests$.

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

have been criticised [69] 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 fault quickly than the other but on completion of test session that very strategy found lower number of total faults than the rival strategy. The preference given to a strategy by F-measure because it finds the first fault quickly without giving due consideration to the total number of faults is not fair [100].

The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered faults in one go and F-measure is not the favourable choice. In our experiments, performance of the strategy was measured by the maximum number of faults detected in SUT by a particular number of test calls [17, 91, 101]. This measurement was effective because it considers the performance of the strategy when all other factors are kept constant.



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

Figure 4.5: Performance of DSSR in comparison with R and R+ strategies.

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

4.5 Results

Results of the experiments including class name, Line of Code (LOC), mean value, maximum and minimum number of unique failures and relative standard deviation for each of the 60 classes tested using R, R+ and DSSR strategy are presented in Table 4.3. Each strategy found an equal number of faults in 31 classes while in the remaining 29 classes the three strategies performed differently from one another. The total of mean values of unique failures 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 faults found by DSSR (1032) is also higher than for R (973) and R+ (1009) which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

4.5.1 Is there an absolute best amongst 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 4.5 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.

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

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

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

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 Can we pick the best default strategy amongst R, R+ & DSSR?

Analysis of the experimental data revealed that DSSR strategy had an edge over R and R+. This is because of the additional feature of spot sweeping in DSSR strategy. In spite of 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 faults, 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

Table 4.4: Results of T-test applied on the experimental data of 29 classes

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

which might be due 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 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 faults on the results: The DSSR strategy performed better when the number of faults was higher in the code. The reason might be that in case of more faults, the fault 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 fault 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 unique 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 [3, 4, 78]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [57]. Some of the well known automated tools based on random strategy includes JCrasher [3], Eclat [4], AutoTest [17, 58], Jartege [76] and YETI [90, 94].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [2, 71, 87, 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 fault patterns 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 random strategy in which interesting values, beside pure random values, are added to the list of test inputs [47]. These inter-

esting values include border values which have high tendency of finding faults in the given SUT [68]. Results obtained with R+ strategy showed significant improvement over random strategy [47]. DSSR strategy is an extension of R+ strategy which starts testing as R+ until a fault is found and then switches to spot sweeping.

A common practice to evaluate performance of an extended strategy is to compare the results obtained by applying the new and existing strategy to identical programs [32, 92, 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 [94, 97, 108].

4.8 Conclusions

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases. We developed 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 random and random+ strategies with a 5% overhead. We found out that for 7 (12%) classes DSSR was significantly better than both R+ and R, for 8 (13%) classes DSSR performed 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.

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 [30]. 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 faults. Debuggers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality 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 [45, 65]. It involves minimum or no overhead in input selection and lacks human bias [57, 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 re-

fined 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 [69]. Similarly Restricted Random Testing (RRT) [102], Mirror Adaptive Random Testing (MART) [2], Adaptive random testing for object oriented program (Artoo) [58], Directed Adaptive Random Testing (DART) [18], Lattice-based Adaptive Random Testing (LART) [112] and Feedback-directed Random Testing (FDRT) [5, 91] 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 fault domain and suggested that the fault finding ability of testing could be improved by taking into consideration these failure domains. In Figure 5.1 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.

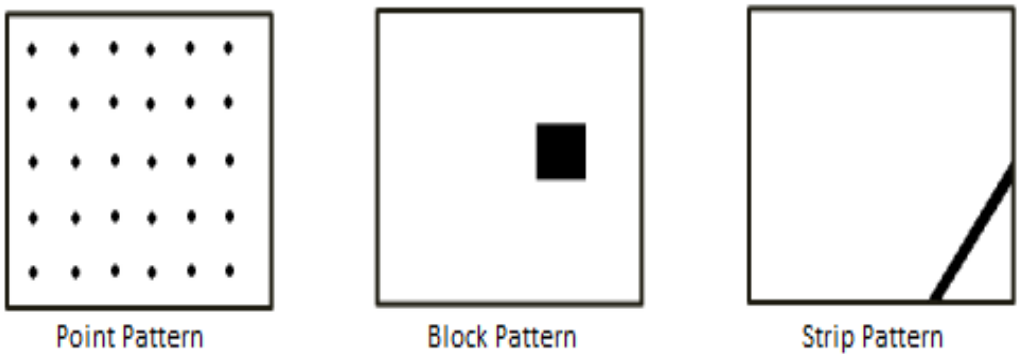


Figure 5.1: Failure domains across input domain [1]

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

- Implementation of the new ADFD strategy in York Extensible Testing Infras-

tructure (YETI).

- Evaluation to assess ADFD strategy by testing classes with different fault domains.
- Reduction in test duration by identification of all fault domains instead of a single instance of fault.
- Improvement in test efficiency by helping debugger to consider all fault 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 faults as well as the fault domains. The output produced at the end of test session is a chart showing the passing value or range of values in green and failing value or range of values in red. The complete work flow of ADFD strategy is given in Figure 5.2.

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

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

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 fault domain in the specified range. Default range for minimum and maximum is taken as Integer.MIN_INT and

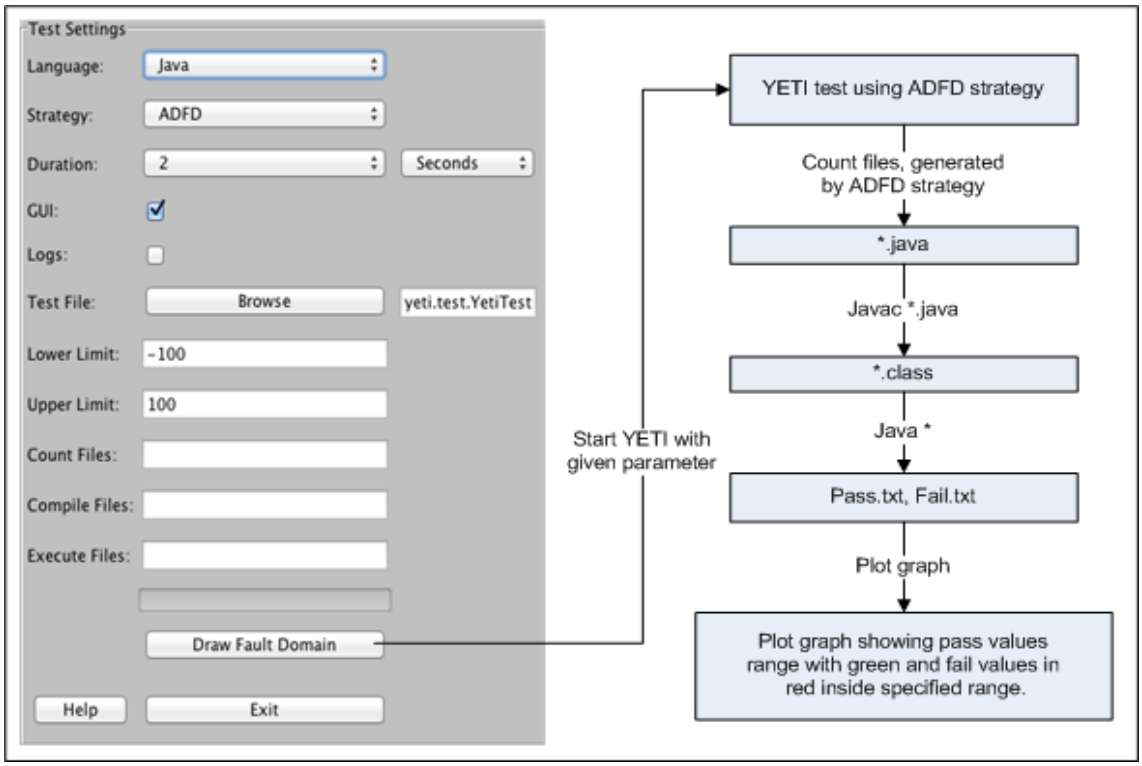


Figure 5.2: Work flow of ADFD strategy

Integer.MAX_INT respectively.

5.2.2 Automated Finding of Fault:

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

5.2.3 Automated Generation of Modules:

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

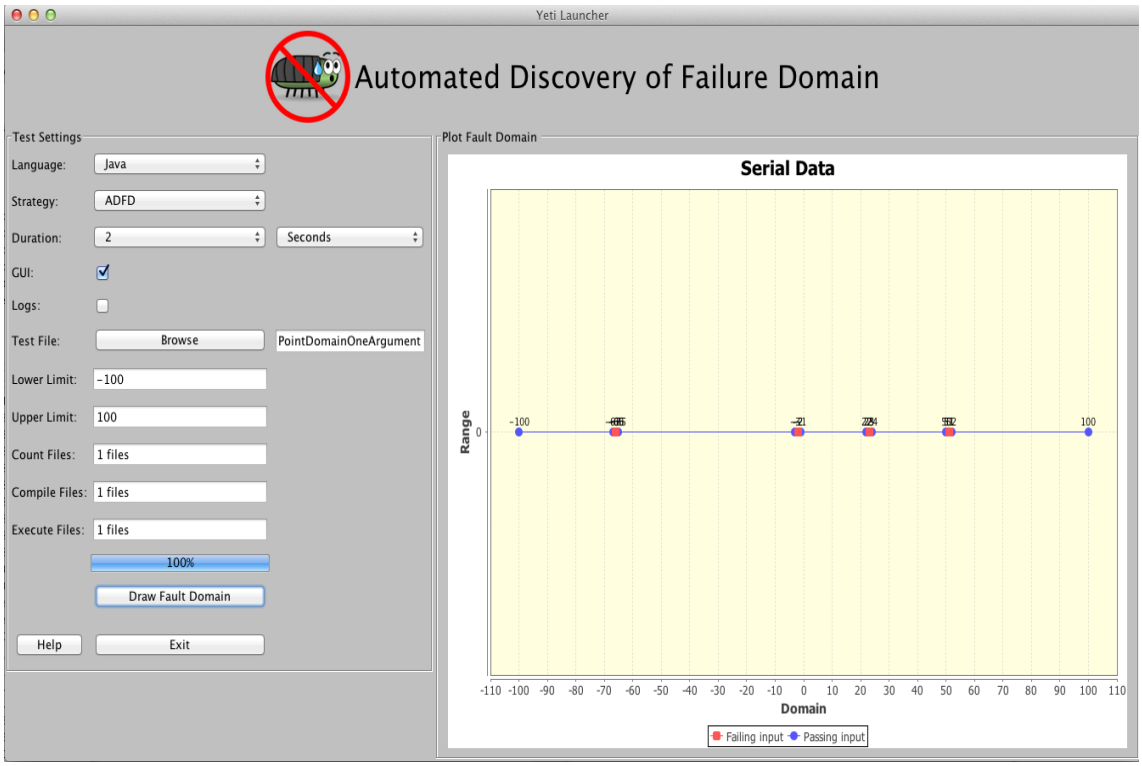


Figure 5.3: Front-end of ADFD strategy

argument one program is generated, for module with two arguments two programs and so on. To track fault 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 to Discover Domains:

The java modules generated in previous step are compiled using $javac*$ command to get their binary $.class$ files. The $java*$ command is applied to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument 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 Showing Domains:

The values from the pass and fail files are used to plot (x, y) chart using JFreeChart. JFreeChart is a free open-source java library that helps developers to display complex charts and graphs [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 faults 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();
    }
}
```

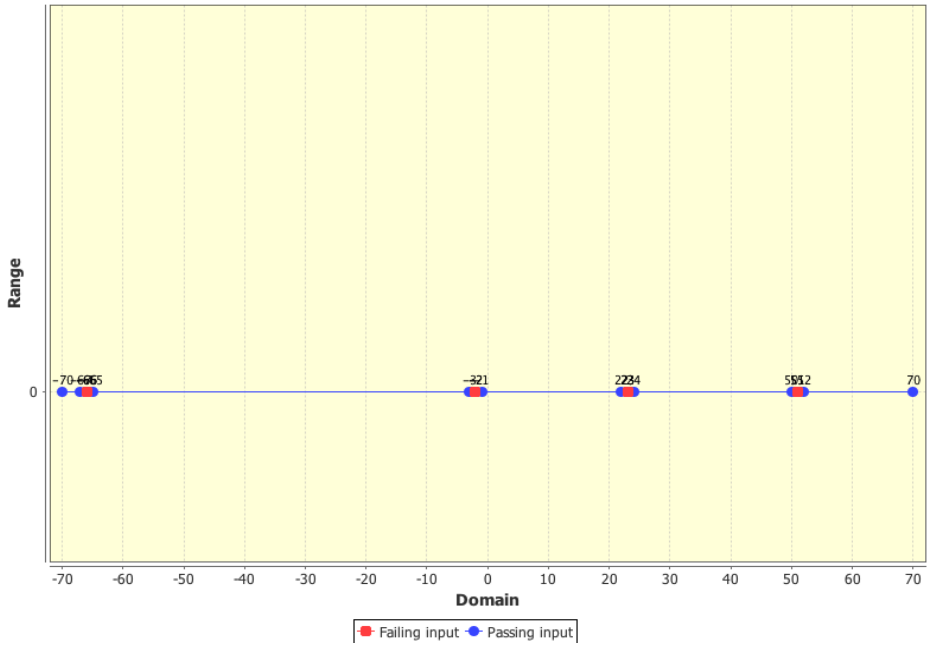


Figure 5.4: ADFD strategy plotting pass and fault domain of the given class

As soon as any one of the above four faults are discovered the ADFD strategy 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 5.4 that ADFD strategy not only finds all the faults 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.

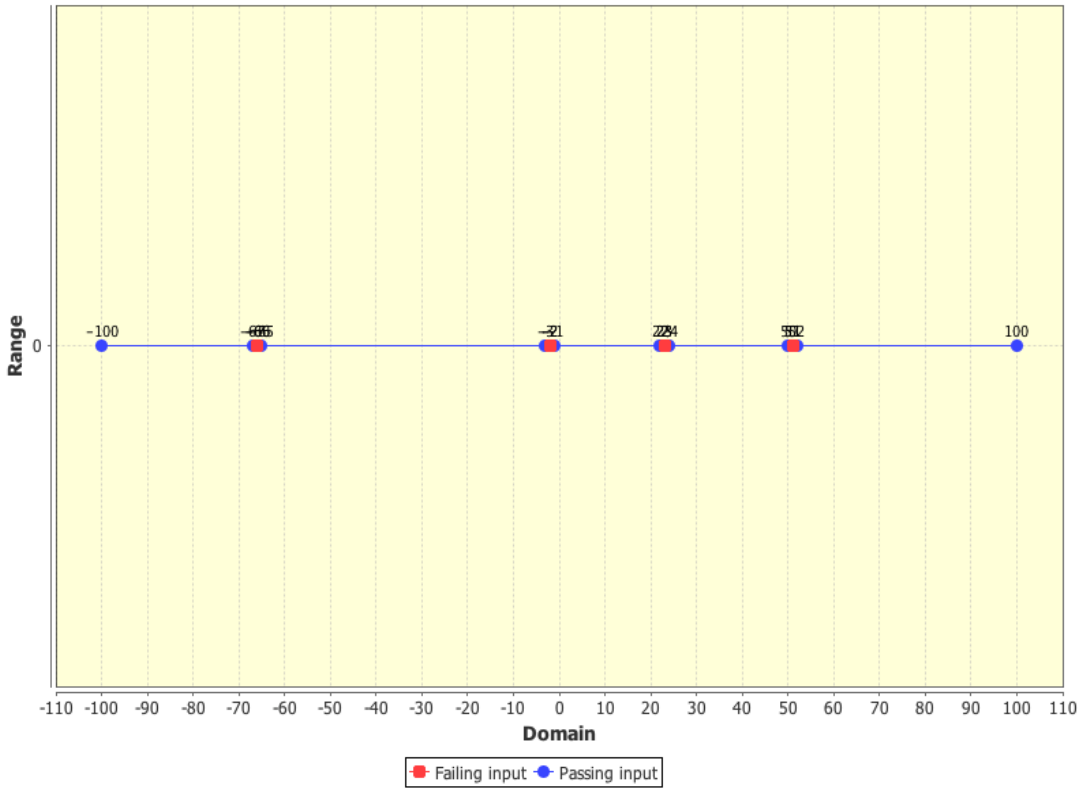
All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35].

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.

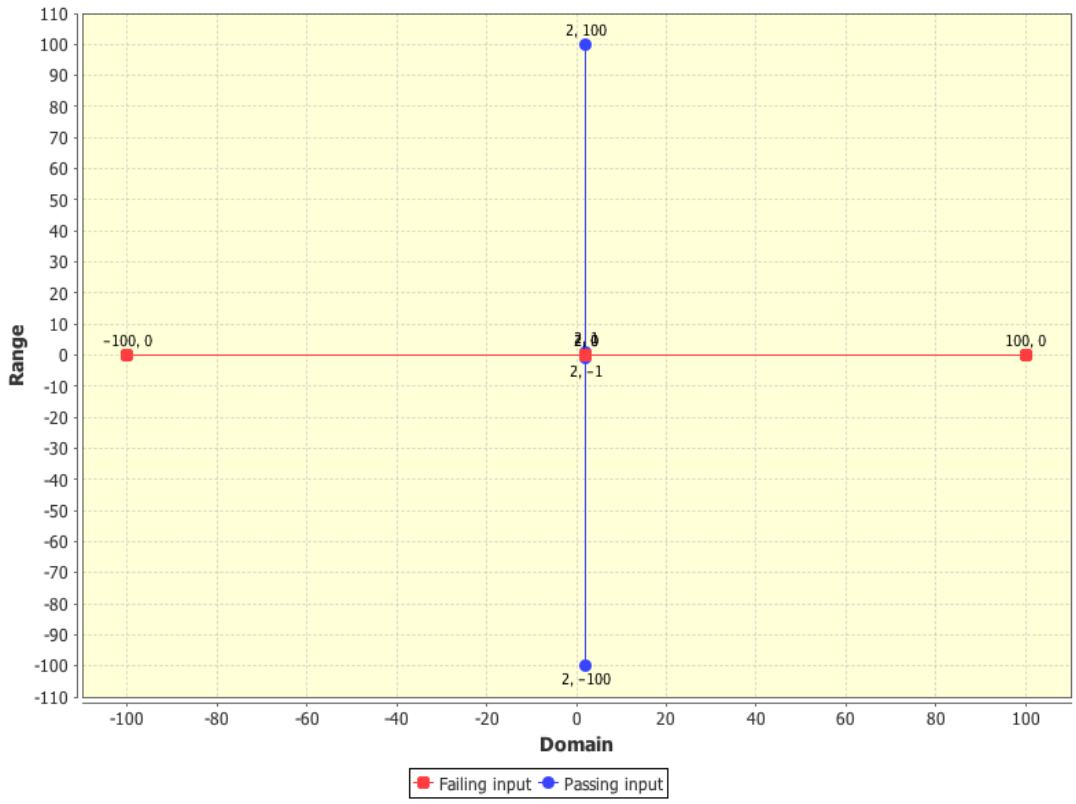
Table 5.1: Experimental results of programs tested with ADFD strategy

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 6.5(a) presents range of pass and fail values for point fault domain in one-dimension whereas Figure 5.5(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.



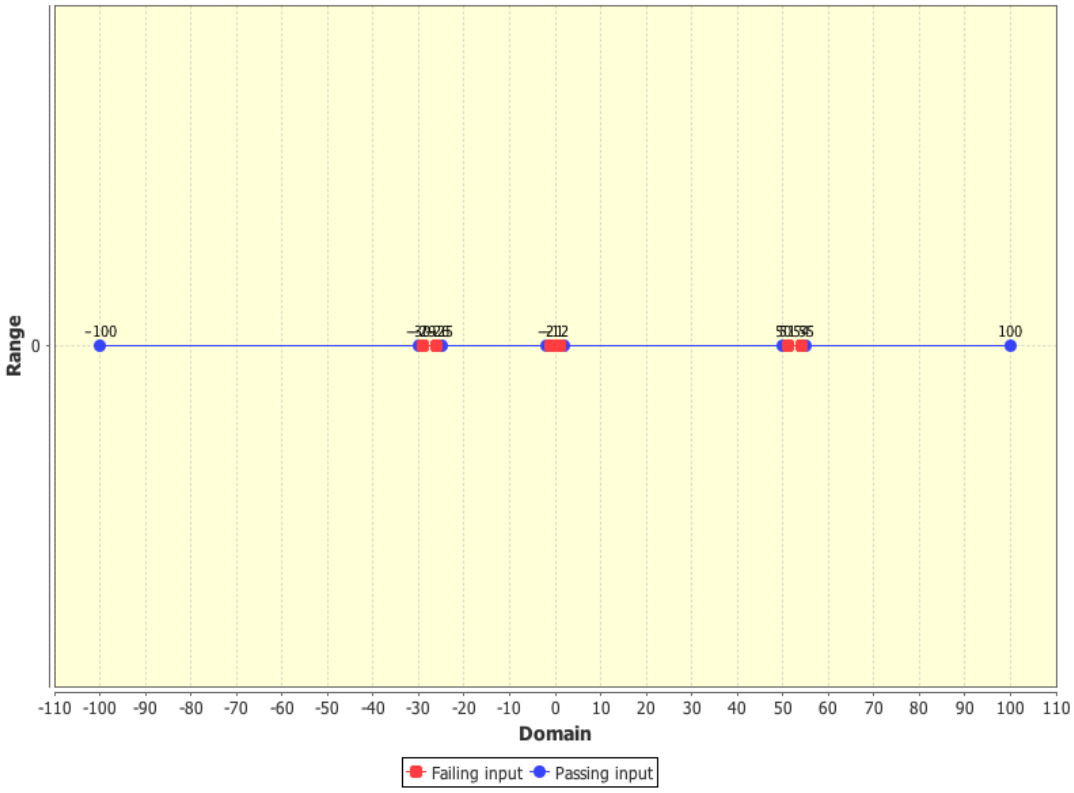
(a) One dimension module



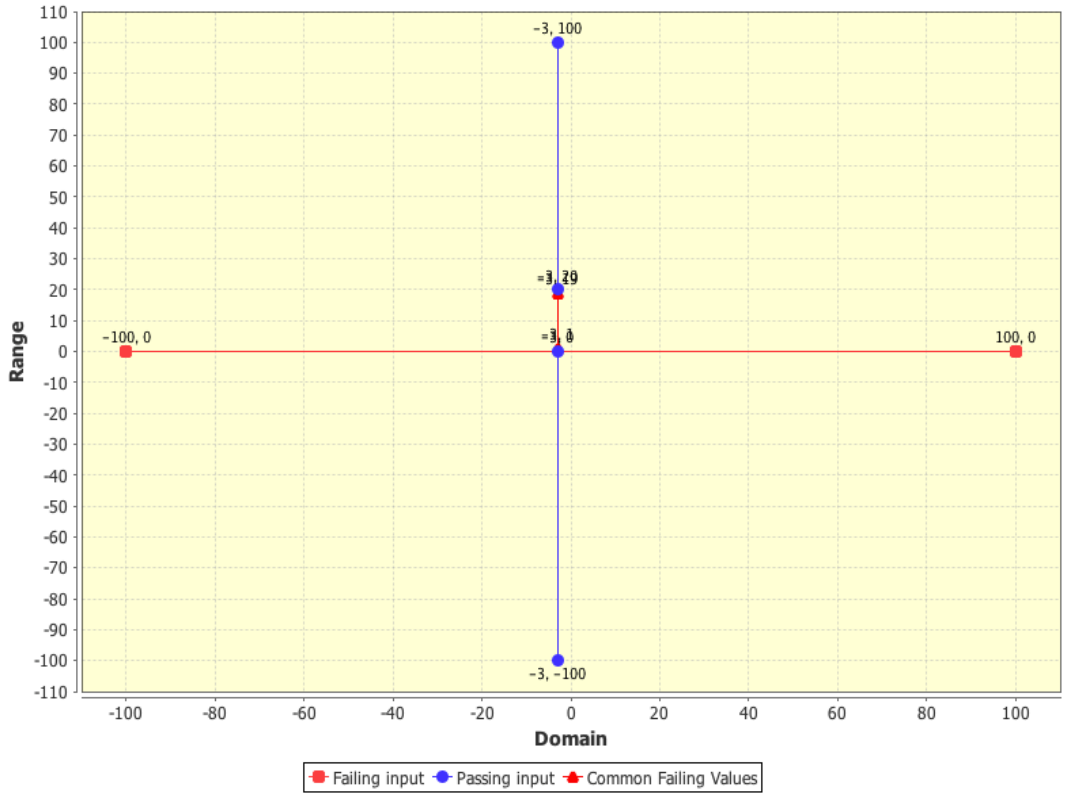
(b) Two dimension module

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

Block Fault Domain: Two separate Java programs Pro4 and Pro5 given in Appendix A.1 (4, 5) were tested with ADFD strategy in YETI to get the findings for block fault domain in one and two-dimensional program. Figure 6.5(b) presents range of pass and fail values for block fault domain in one-dimension whereas Figure 5.6(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.



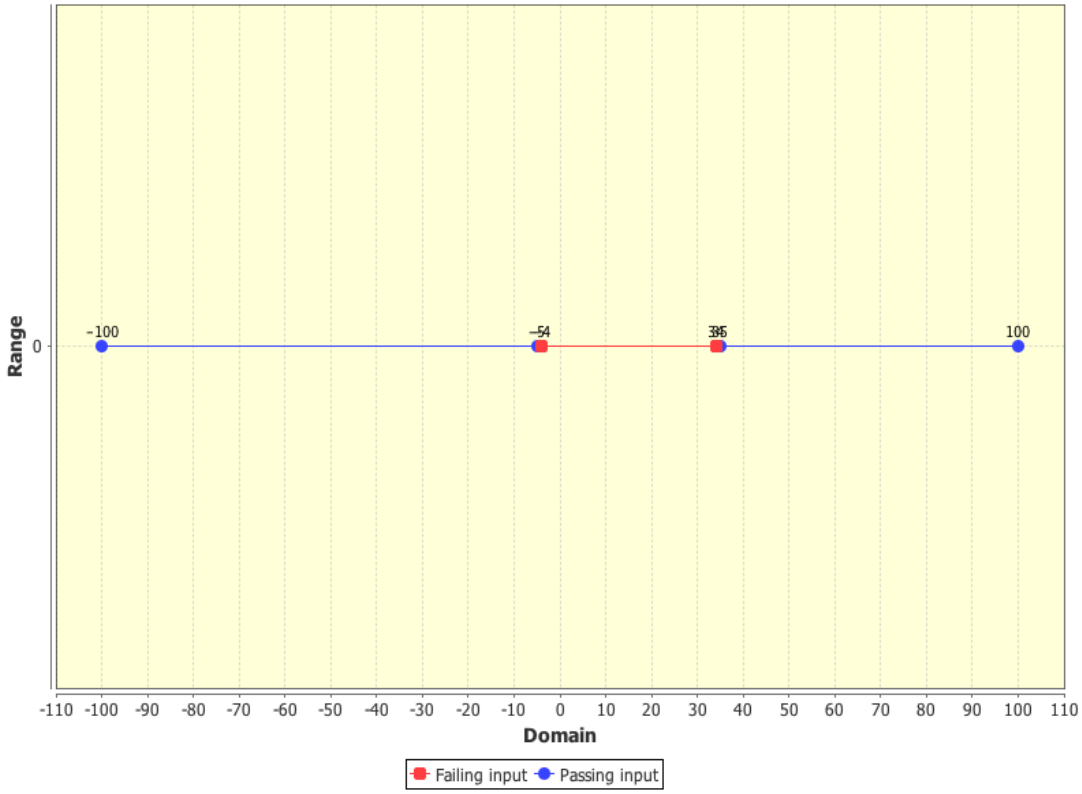
(a) One dimension module



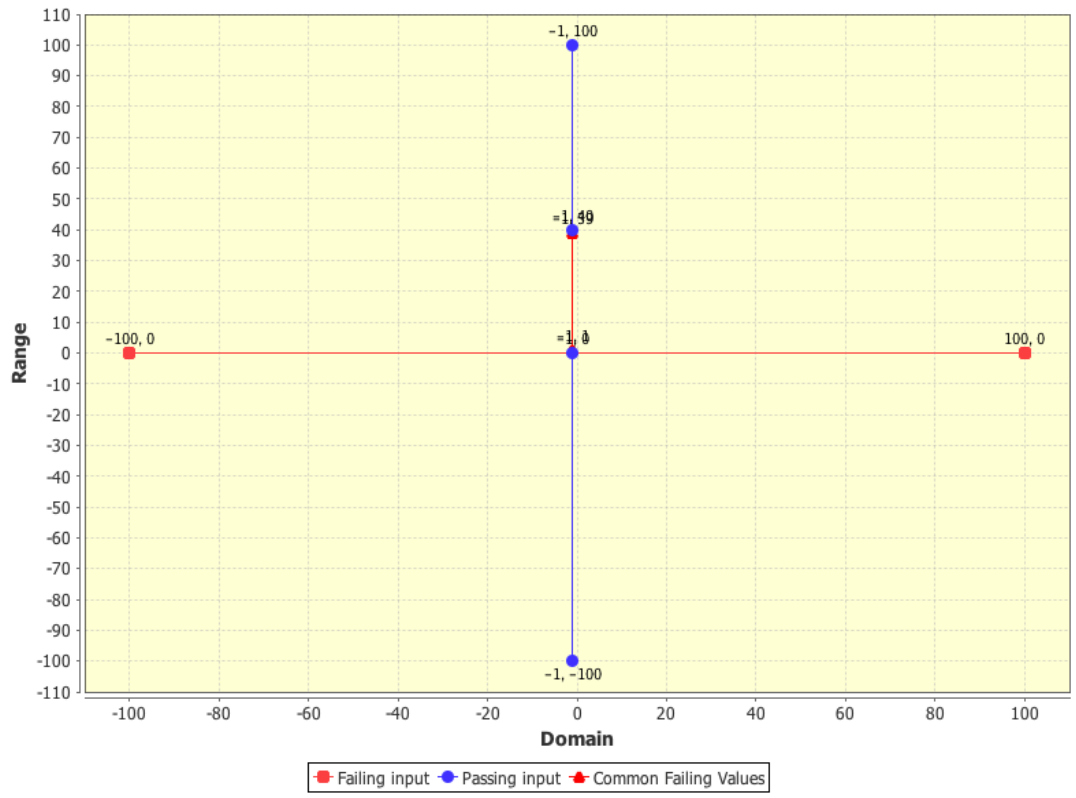
(b) Two dimension module

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

Strip Fault Domain: Two separate Java programs Pro6 and Pro7 given in Appendix A.1 (6, 7) were tested with ADFD strategy in YETI to get the findings for strip fault domain in one and two-dimension program. Figure 6.5(c) presents range of pass and fail values for strip fault domain in one-dimension whereas Figure 5.7(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 5.7: 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 identify faults, fault 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 fault, saving the program to a permanent media, compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automatically without any human intervention.

As evident from the results, ADFD strategy effectively identified faults and fault domains in a program. Identification of fault 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 fault and later switches to brute-force strategy to apply all the values between upper and lower bounds for finding pass and fault domains. It was found that faults 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 fault domain the program is executed for every input available in the specified range. Similarly increasing the range also shrinks the produced graph making it difficult to identify clearly point, block and strip domains unless they are of considerable size. Test duration is also influenced by identification of the first fault and the complexity of module under test.

ADFD strategy can help the debuggers in two ways. First, it reduces the ‘to’ and ‘from’ movement of the program between the testers and debuggers as it identify

all the faults in one go. Second, it identifies locations of all fault domains across the input domain in a user-friendly way helping debugger to fix the fault keeping in view its all occurrences.

5.6 Threats to Validity

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

Another issue is the 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 [72], 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 [11, 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 Conclusions

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 faults quickly as against the traditional random testing, which is either, unable or takes comparatively longer time to discover the faults.

The use of ADFD strategy is highly effective in testing and debugging. It provides an easy to understand test report visualising pass and fail domains. It reduces the number of switches of SUT between testers and debuggers because all the faults are identified 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

Analysis of Failure-Domain by ADFD+ and Daikon

6.1 Introduction

Testing is an essential and most widely used method for verification and validation process. Efforts have been continuously made by researchers to make it more and more effective and efficient. Testing is effective when it finds maximum number of faults in minimum number of test cases and efficient when it executes maximum number of test cases in minimum possible time. Upgrading existing techniques and developing new test strategies focus on increasing test effectiveness while automating one or more components or complete system aims at increasing efficiency.

Boundary Value Analysis (BVA) is one of the technique used of increasing test effectiveness. In BVA test cases with boundary values are added to the test suite with the assumption that errors reside along the boundaries [35]. Daikon [117] is an automatic tool used to improve the efficiency. It saves testers time by automatically generating likely program invariants.

However, the two approaches can adversely affect the testing process if wrong boundaries or invariants are taken into consideration. It is therefore motivating to accurately identify the boundaries of the input domain in BVA and measure the degree of correctness of auto-generated invariants by Daikon in the case of point, block and strip failure domain. To analyse the failure domains the ADFD+ tech-

nique was developed and experiments were conducted by testing several error-seeded one and two-dimensional numerical programs with ADFD+ and Daikon. The results obtained were analysed and reported.

The main contributions of the study are:

- **ADFD+:** It is an extension of Automated Discovery of Failure Domain (ADFD) strategy developed by Ahmad and Oriol [118]. The new technique improves the search algorithm of ADFD and makes the report more intuitive (Section 6.3).
- **Implementation of ADFD+:** It is implemented and integrated in the York Extensible Testing Infrastructure (Section 6.3.2).
- **Evaluation:** The results generated by ADFD+ and Daikon about failure domains in the error-seeded programs are evaluated (Section 6.5). The results show that although Daikon generate invariant to identify the failure yet it is not able to identify the boundary of failure domain as accurately as ADFD+.
- **Future work:** ADFD+ can be extended to find and plot failure domains in multi-dimensional non-numerical programs (Section 6.10).

6.2 Preliminaries

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [119, 120, 121]. According to Chan et al. [1] the clusters are arranged in the form of point, block and strip failure domain. In the point domain the failure revealing inputs are stand-alone, and spread through out the input domain. In block domain the failure revealing inputs are clustered in one or more contiguous areas. In strip domain the failure revealing inputs are clustered in one long elongated area. Figure 6.1 shows the failure domains in two-dimensional input domain.

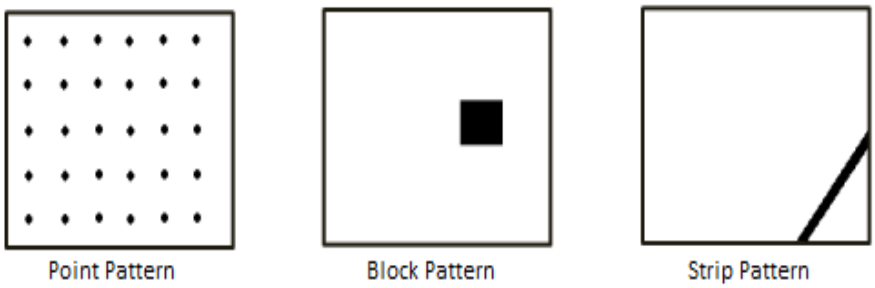


Figure 6.1: Failure domains across input domain [1]

6.3 Automated Discovery of Failure Domain+

ADFD+ is an improved and extended form of ADFD strategy developed previously by Ahmad and Oriol [118]. It is an automated framework which finds the failures and their domains within a specified range and present them on a graphical chart.

The main improvements of ADFD+ over ADFD strategy 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 a failure and its domain.
- ADFD+ takes a single value as range with in which the strategy search for a failure domain whereas ADFD takes two values for lower and upper bound representing x and y-axis respectively.
- In ADFD+, the algorithm of dynamically generating Java file, created at runtime after a failure is detected, is made more simplified and efficient.
- In ADFD+, the failure domain is focused in the graph, which gives a clear view of, pass and fail points. The points are also labelled for clarification as shown in Figure 6.2.

6.3.1 Workflow of ADFD+

ADFD+ is a completely automatic process and all the user has to do is to specify the program to test and click the $DrawFaultDomain$ button. The default value for range is set to 5, which means that ADFD+ will search 83 values around the failure. On clicking the button YETI is executed with ADFD+ strategy to search for a failure in two-dimension program. On finding a failure the ADFD+ strategy creates a Java file which contains calls to the program on the failing value and its surrounding values within the specified range. The Java file is compiled and executed and the result is analysed to check for pass and fail values. Pass and fail values are stored in pass and fail text files respectively. 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.

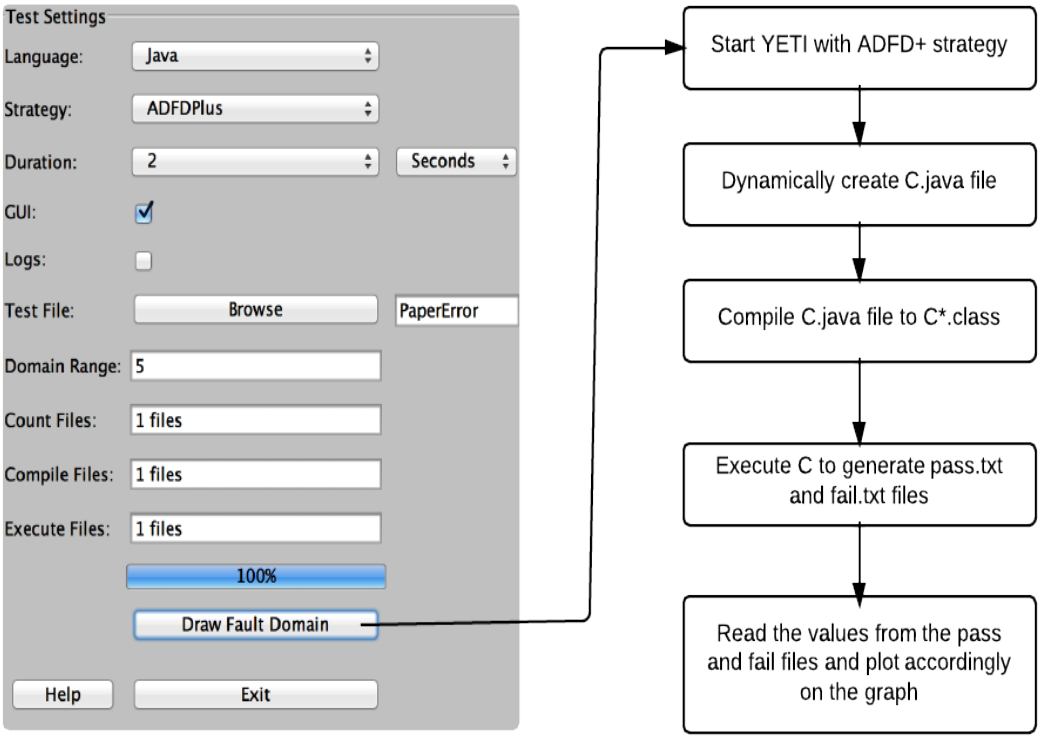


Figure 6.2: Workflow of ADFD+

6.3.2 Implementation of ADFD+

The ADFD+ technique is implemented in YETI. The tool YETI 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 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, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both strategies and languages sections have 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. The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. In strategies package, on top of the hierarchy, is an abstract class $YetiStrategy$, which is extended by $YetiRandomPlusStrategy$ and is further extended to get ADFD+ strategy.

6.3.3 Example to Illustrate Working of ADFD+

Suppose we have the following error-seeded class under test. From the program code, it can be easily noticed that an $ArithmeticException$ (DivisonByZero) 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;
```

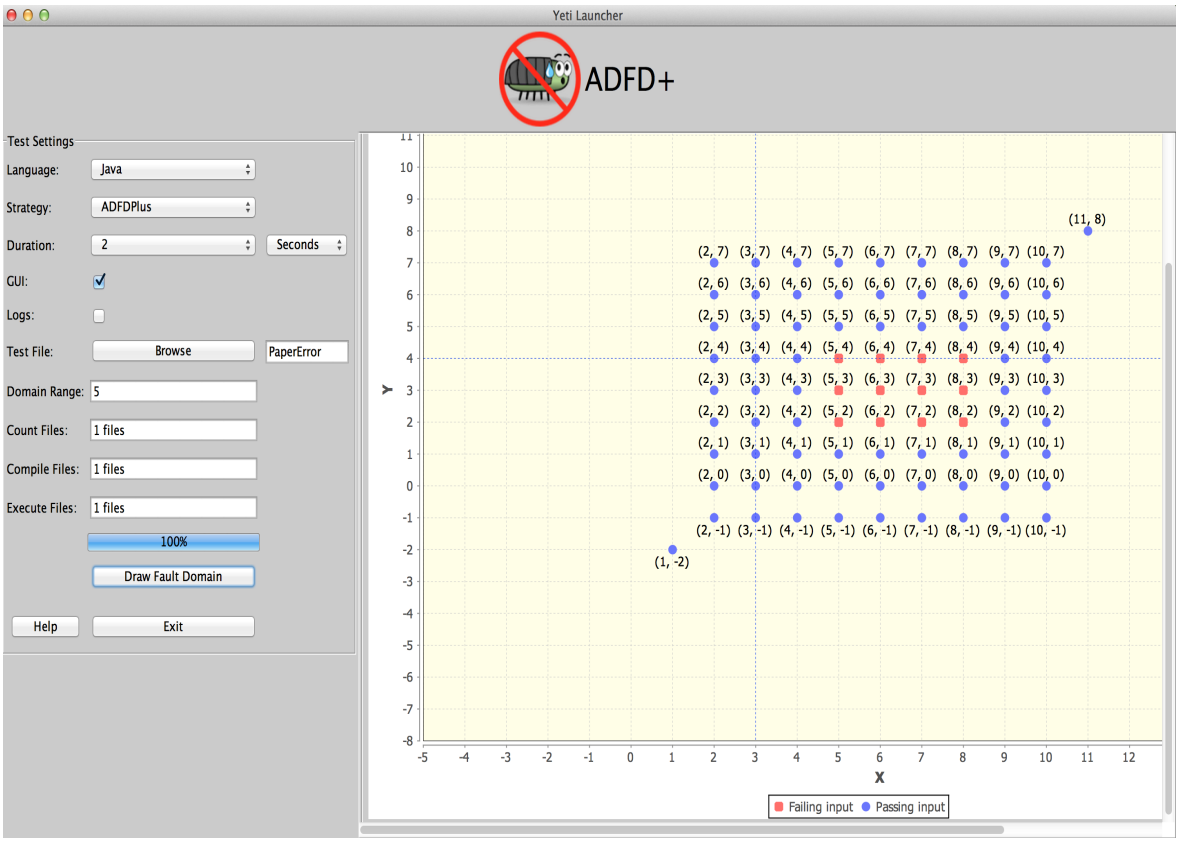


Figure 6.3: The output of ADFD+ for the above code.

```
    if ((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
            {
                z = 50/0;
            }
    }
}
```

On test execution, the ADFD+ strategy evaluates the 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 range of surrounding values is limited to the value set by the user in the $DomainRange$ variable. When the value of $DomainRange$ is 5, ADFD+ evaluates 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 6.3 shows that the failure domain forms a block pattern and the boundaries of the failure are $(5, 2), (5, 3), (5, 4), (6, 2), (6, 4), (7, 2), (7, 4),$

$(8,2),(8,3),(8,4)$.

6.4 Daikon

Daikon is a tool [117], which uses machine-learning technique to automatically generate likely invariants of the program written in C, C++, Java and Pearl. Daikon takes the program and a few test cases as input. The test cases may be either generated manually or by an automated tool. Daikon executes the test cases on the program under test and observes the values that the program computes. At the end of the test session it reports the properties that were true for the observed executions. A feature of Daikon facilitate to process the generated invariants to mitigate non-interesting and redundant invariants. Another feature allows to inserts the generated invariants in to the source code as assertions. The report generated by Daikon is useful in understanding program logic, generating invariants, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistencies in data structures and checking the validity of data streams.

6.5 Evaluation of Daikon by ADFD+

Because of using error-seeded one and two dimensional numerical programs, we were aware of the failure domain present in each program. The correct identification and presentation of the failure domain by ADFD+ prove the correct working of ADFD+. We then evaluated the same program by Daikon and plot its results. The unit test cases required by Daikon for generating invariants were generated using Randoop [5]. YETI being capable of generating the test cases is not used for this step to keep the second completely independent from first.

6.5.1 Research Questions

The following research questions have been addressed in this study:

1. Is Daikon capable of generating invariants to identify the failure?
2. Is Daikon capable of generating invariants to identify the failure domain?

3. Is Daikon capable of generating invariants to identify the boundaries of the failure domain?

6.5.2 Experimental Setup

To evaluate the performance of Daikon, we carried out testing of several error-seeded one and two-dimensional numerical programs written in Java. The programs were divided in to two sets. Set A and B contains one and two-dimensional programs respectively. Each program was injected with at least one failure domain of point, block or strip nature. Every program was tested independently for 30 times by both ADFD+ and Daikon. The external parameters were kept constant in each test run and the initial test cases required by Daikon were generated by using an automated testing tool Randoop. The code for the programs under test is given in Appendix A.1 while the test details are presented in Table 6.3.

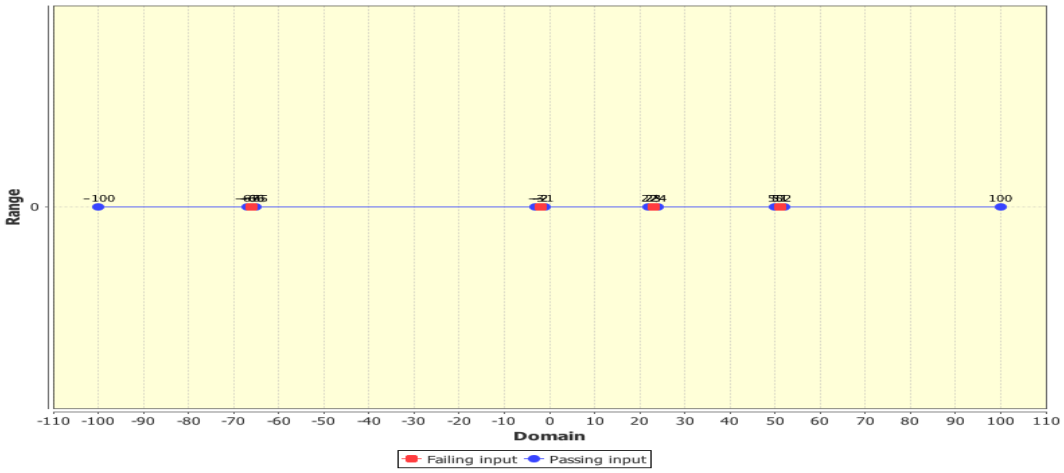
Every class was evaluated through 10^5 calls in each test session of ADFD+. Due to the absence of contracts and assertions in the code under test, undeclared exceptions were taken as failures in accordance with the previous studies [118, 96]. All tests were performed with a 64-bit Mac OS X Lion Version 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^{TM}SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

6.6 Results

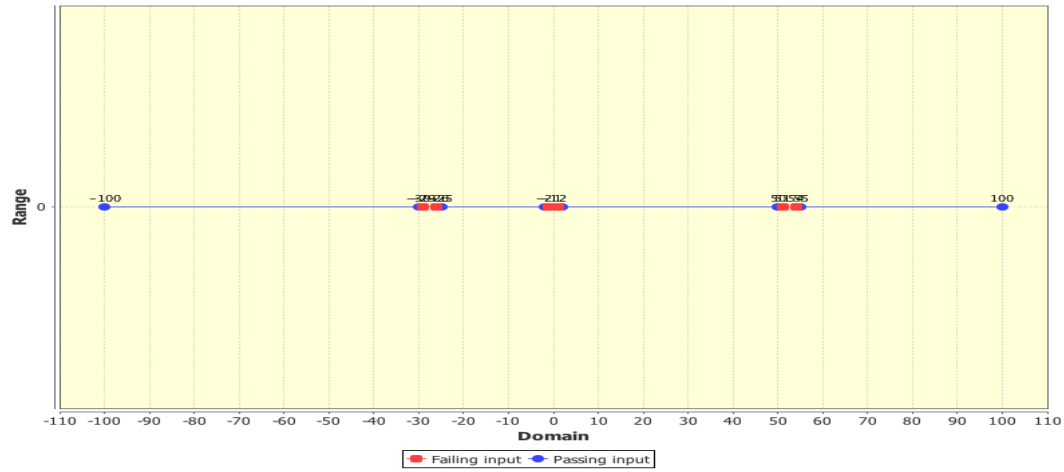
The results are split in to four sub-sections for convenience.

6.6.1 Test of One-dimension Programs by ADFD+

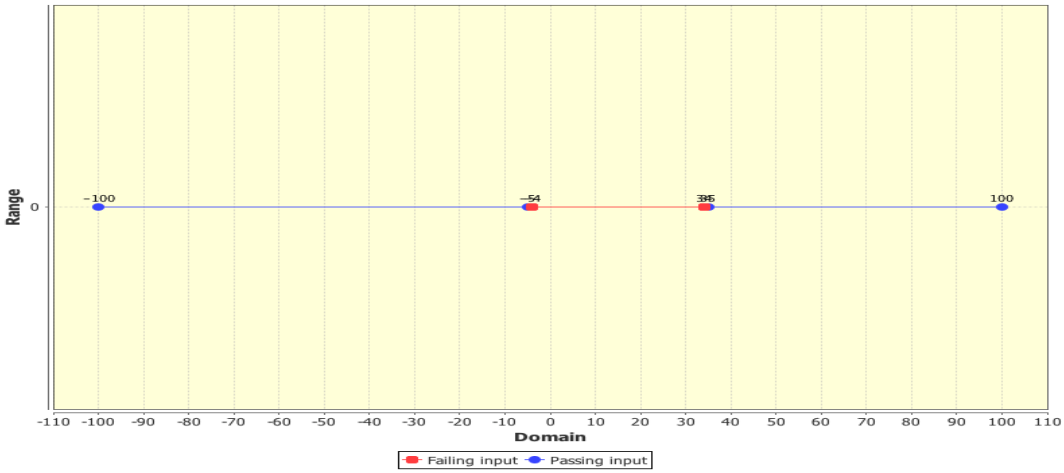
In each of the 30 experiments, The ADFD+ successfully discovered and plotted the failure domains for point, block and strip pattern as shown in the Figure 6.4. The lower and upper bound for each experiment are set to -100 and 100 respectively.



(a) Point failure domain in one-dimension



(b) Block failure domain in one-dimension



(c) Strip failure domain in one dimension

Figure 6.4: Pass and fail values plotted by ADFD+ for one-dimension programs

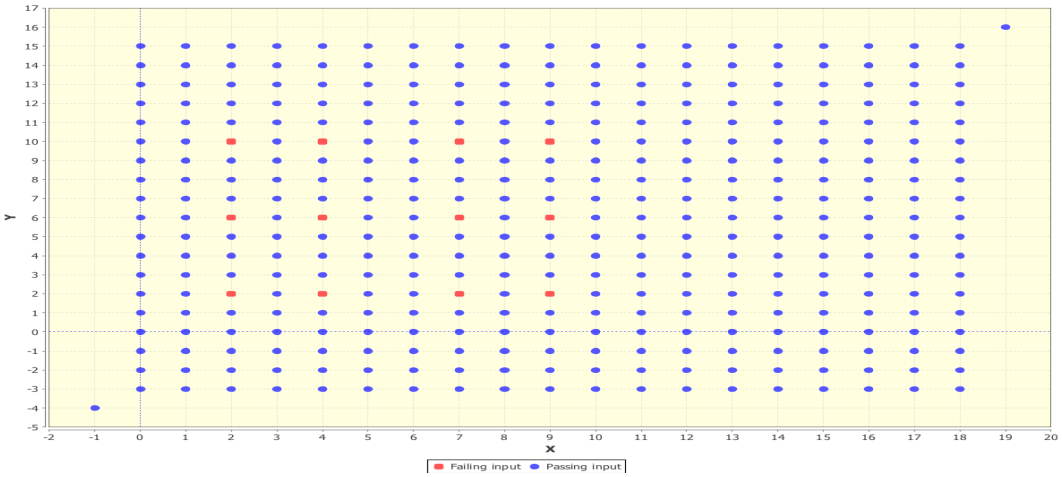
Table 6.1: Table depicting values of x and y arguments responsible for forming point, block and strip failure domain in Figure 6.4

Point failure	Block failure	Strip failure
x = -66 x = -2 x = 51 x = 23	x = -1 x = 0 x = 1	x = -4 — 34

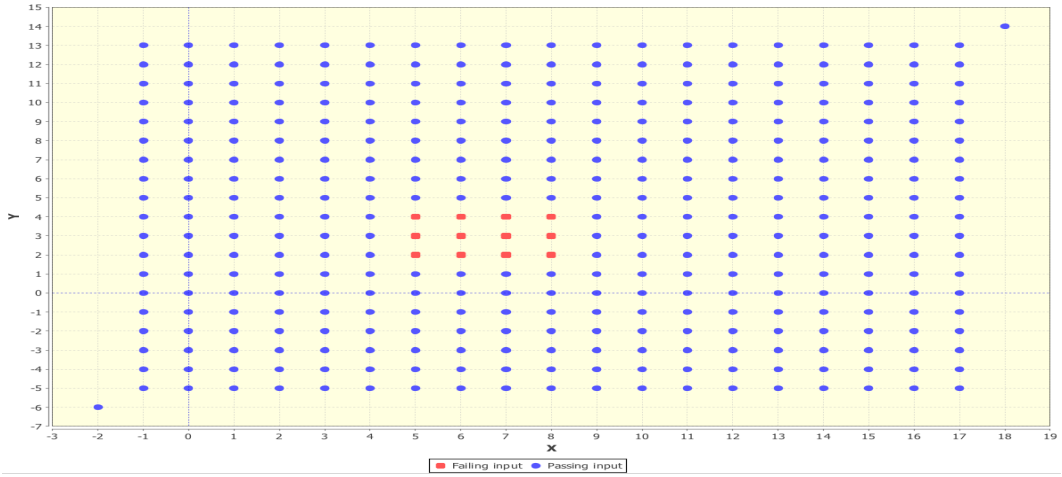
6.6.2 Test of One-dimension Programs by Daikon

6.6.3 Test of Two-dimension Programs by ADFD+

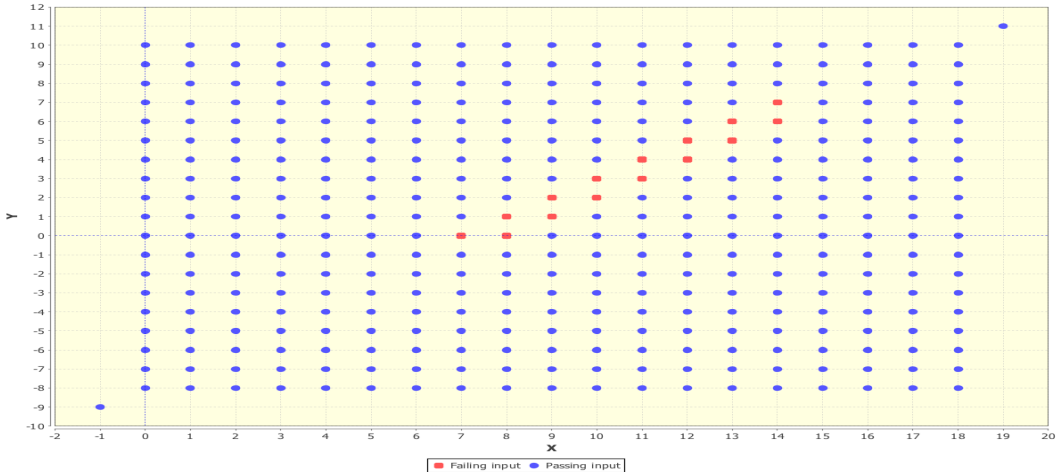
In each of the 30 experiments, The ADFD+ once again successfully discovered and plotted the failure domain for point, block and strip failure domain as shown in the Figure **??**. The range value for each experiment is set to 10. Labels are disabled in the charts given in Figure **??** for clarity purpose. The failure values in each of the point, block and strip failure domain is given in Table 6.1.



(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Figure 6.5: Pass and fail values plotted by ADFD+ for two-dimension programs

Table 6.2: Table depicting values of x and y arguments responsible for forming point, block and strip failure domain in Figure 6.5

Point failure	Block failure	Strip failure
x = 2, y = 10	x = 5, y = 2	x = 7, y = 0
x = 4, y = 10	x = 6, y = 2	x = 8, y = 0
x = 7, y = 10	x = 7, y = 2	x = 8, y = 1
x = 9, y = 10	x = 8, y = 2	x = 9, y = 1
	x = 5, y = 3	x = 9, y = 2
	x = 6, y = 3	x = 10, y = 2
	x = 7, y = 3	x = 10, y = 3
	x = 8, y = 3	x = 11, y = 3
	x = 5, y = 4	x = 11, y = 4
	x = 6, y = 4	x = 12, y = 4
	x = 7, y = 4	x = 12, y = 5
	x = 8, y = 4	x = 13, y = 6
		x = 14, y = 6
		x = 14, y = 7

6.6.4 Test of Two-dimension Programs by Daikon

6.7 Discussion

We have shown that ADFD+ is a promising technique to find a failure and using it as a focal point find the whole failure domain. We have also shown that ADFD+ can graphically draw the failure domain on a chart. The failure values are drawn in red and the pass values are drawn in green. The pictorial representation of failure domain helps in easily identifying the underlying pattern and its boundaries.

As a pilot study, we also ran an empirical study to evaluate several error-seeded programs. While it would be surprising if production programs produced much different results, it would be worthwhile to check.

More importantly, the implementation of ADFD+ for this pilot study has significant limitations in practice, as it requires only one and two dimensional numerical pro-

Table 6.3: Table depicting values of failure points identified by ADFD+ Daikon

Technique	Dimension	Test cases	Point failure	Block failure	Strip failure
ADFD+	One	N/A			
Daikon	One	10			
Daikon	One	20			
ADFD+	Two	N/A			
Daikon	Two	10			
Daikon	Two	20			

grams. Though it is not difficult to extend the approach to test more than two-dimensional programs containing other primitive types, it would however be difficult to plot them on the chart as the number of coordinates increases. The approach can also be extended to test object-oriented programs by implementing objects distance proposed by Ciupa et al. [72]. The details of such an implementation will take some effort.

The ADFD+ range value specifies how many values to test around the failure. The range can be set to any number before the test starts. The value of range is directly proportional to the time taken because the higher the range value the higher number of values to test. Higher range value also leads to a very large graph and the tester has to use the zoom feature of graph to magnify the failure region.

6.8 Threats to Validity

The research study faces threats to external and internal validity. The threats to external validity are the same, which are common to most of the empirical evaluations i.e. to what degree the classes under test and test generation tool (Randoop) are representatives of true practice. The classes under test contains failure patterns in only one and two-dimensional input domain. The threats may be reduced to a greater extent in future experiments by taking several types of classes and different test generation tools.

The threat to internal validity includes annotation of invariants that can bias the

results, which may have been caused by error-seeded classes used in our experiments. Internal threats may be avoided by taking real classes and failures in the experiments. Moreover, testing a higher number of classes will also increase the validity of the results.

6.9 Conclusions

Automated Discovery of Failure Domain+ (ADFD+) is distinctive from other random test strategies in the sense that it is not only limited to identifying a failure in the program. Instead, the failure is exploited to identify and graphically plot its failure domain.

In the first section, we describe ADFD+ in detail which is based on our previous approach ADFD [118]. We then describe the main improvements of ADFD+ over ADFD.

In the second section, we analysed and compared the results of the experiments performed by both ADFD+ and Daikon in the case of programs with point, block and strip failure domain.

We showed that Daikon lakes to accurately identify the failure boundary and therefore cannot generate invariants for such failures. We further explain why Daikon does not work well for boundary failures. The main reason we identified for this behaviour is Daikons dependence on initial set of test cases, which are required by Daikon for generating invariants. With increase in number of test suite or high quality test suite improves the performance of invariants.

6.10 Future Work

The current approach can be extended to a larger set of real world multi-dimensional programs, using real failure instead of error-seeded programs. However, to plot failure domains of complex multi-dimensional nature, more sophisticated graphical tools like Matlab will be required rathar than JFreeChart used in the current study. This may not restrict the formation of new failure domains to point, block and strip failure domain in one and two-dimensional numerical programs.

Chapter 7

Conclusions

The thesis explored various aspects of failure-domains across the input domain with respect to automated random testing. It investigated and established three new techniques that could effectively identify and plot the failure-domains. The thesis mainly focused on: First, to minimize the number of test cases required to discover a failure-domain. Second, to identify the legitimate and failure-domains and represent them in a graphical form. Third, to upgrade the existing strategy and compare its behaviour with Daikon in finding the failure-domain. Section 1.2 and Section 1.3 presented the goals and contributions of the thesis.

Failures reside in contagious locations forming point, block and strip failure-domains. The existing random test strategies tries to find individual failures but do not focus on its domain. Providing the knowledge of failure along with its domain benefits debuggers to remove the failure from its root quickly and effectively. The test reports could be further simplified when the tested values are shown on a graphical form separating pass and fail values and highlighting the failure 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 DSSR (Chapter 4), starts with random strategy to find the first failure. 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. Experimental results show that DSSR performed significantly better than random and random+ strategy. The second technique ADFD (Chapter 5), starts with random+ strategy to find the first failure. When a failure is identified a new Java program is dynamically created at run-time. The program is

compiled and executed to search for failure-domains in the specified range. The output of the tests are collected and represented in the graphical form. It improves the debugging efficiency as debuggers keep all the instances of a failure under consideration during debuggin. Experimental results show that ADFD technique correctly identified the failure domains in the error-seeded programs. The third and final technique ADFD+ ((Chapter 6), is an improvement of ADFD strategy with a better algorithm to find failure and failure-domains. It searches for the failure-domains around the found failure in a given radius instead of the lower and upper bound. The results are collected and represented on a graph in a more simplified manner. The ADFD+ output is also compared with the Daikon's output to analyse their behaviour in response to known failure domains. Experimental results show that ADFD+ correctly pointed out the planted failure-domains whereas Daikon, which rely on the existing of test cases to generate invariants, was not able to generate invariants that correctly point to the failure-domain. The ADFD+ graphical output is made further user friendly providing labelled graphs making it more simple to understand. Testers and/or developers can inspect those generated graphs for failures and failure-domains, instead of inspecting a large number of all generated tests.

7.1 Lessons Learned

Research in the field of software testing has been carried out for more than three years. However, only a handful of fully automated testing tools are available free and open-source. Better tool support is needed to help the research community in order to devise new testing techniques to meet the demand for high software reliability. The research in this thesis has developed new techniques to improve the effectiveness of automated random testing. Our research is motivated to investigate how to improve the effectiveness of random testing for contagious failures across the input domain and graphical representation of the failure-domains. Our research has shed light on this promising direction and pointed out future work along this direction. In this section, we summarize some lessons that we learned from this research and we hope these lessons may be helpful to other researchers in pursuing future research.

7.1.1 Performance measurement criteria may be carefully chosen

Various measures including the E-measure (expected number of failures detected), P-measure (probability of detecting at least one failure) and F-measure (number of test cases used to find the first fault) have been reported in the literature for finding the effectiveness of random test strategy. The E-measure and P-measure have been criticised [69] 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 fault quickly than the other but on completion of test session that very strategy found lower number of total faults than the rival strategy. The preference given to a strategy by F-measure because it finds the first fault quickly without giving due consideration to the total number of faults is not fair [100].

7.1.2 Test results of random testing are fluctuating

In random testing the results of the experiments keep on changing even if all the test parameters and the program under test remain the same. The difference in results is due to the random generation of test input. It is therefore complicated to find the efficiency of one technique in comparison to the other technique. We cope the problem in four different ways. 1) To repeat the experiment for sufficient number of times (30 times in our case) and calculate the average. 2) To execute tests in each experiment for relatively large number of times (10,000 in our case). 3) To take sufficient number of test samples (60 in our case) before reaching any conclusion. 4) To Experiment on error-seeded programs where the precise locations of faults are well know before the test. All the above suggestions minimize the effect of randomness, however, the experiments must be repeated for at least 30 times before deciding the effectiveness of one approach over the other.

7.1.3 More computation decreases performance and increases overhead

Versions of random testing [] improves the fault finding ability of the technique, however, the cost of generating test data is too high and results in unwanted overhead. It is difficult to achieve high quality strategy without any overhead but efforts should be made to keep it to bare minimum. For example, if we test a program with two strategies A and B. Strategy A consumes 100 tests to find a fault while strategy B consumes 500 tests to find the same fault. It looks that strategy A supersedes strategy B however the time to generate the test cases may also be considered which can be equal for both the strategies. In addition to time other resources like hardware etc should also be kept minimum.

7.1.4 Starting with random testing and switching to exhaustive strategy helps

In our techniques ADFD and ADFD+, we started the test using random strategy and later switched to exhaustive strategy to find the failure domain of a program. In the strategies the testing starts with random strategy and when a failure is identified the test strategy changes to exhaustive strategy and search all the values around the failure to the specified range set by the tester. The benefit of the approach is that it quickly identify the neighbouring faults which may be difficult to find using random strategy.

7.1.5 Graphical form helps to easily understand the output

In our technique we represented the test output in an easy to understand graphical form. The lines and circles with blue colour represent pass values while lines and squares with red colour represents fail values. The graph shows red points when the program fails for only one value, block when the program fails for multiple values and strip when the program fails for a long range of values. The technique 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 faults in one go.

Second, it identifies locations of all fault domains across the input domain in a user-friendly way helping debugger to fix the fault keeping in view its all occurrences.

7.1.6 Auto-generation of data is simple for primitive types and complicated for reference data types

We found that comparatively it is less challenging to auto-generate primitive data type values than reference or object or user defined data type values using random testing. To meet the challenge our strategy constructed test cases by creating objects of the classes under test and randomly calls its methods with random inputs according to the parameter's-space. We divided input values into two types i.e. primitive data types and user defined data types. For primitive data types as methods parameters, the random strategy calls `Math.random()` method to generate arithmetic values which are converted to the required type using Java cast operation. For user-defined data type as a parameter the strategy calls the constructor to generate object of the class at run time. The constructor may possibly require another object, then the strategy recursively calls the constructor 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.

7.1.7 Graphical representation of multi-argument method and reference data type is complicated

The difficulty to represent results is directly proportional to the number of arguments used in a method. Which means as the number of arguments in a method increases, the more dimensions we need and the more complicated the process get in to. Though it is not difficult to extend the approach to test more than two-dimensional programs containing other primitive types, it would however be difficult to plot them on the chart. The approach can also be extended to test object-oriented programs by implementing objects distance proposed by Ciupa et al. [72]. The details of such an implementation with graphical representation will be complicated.

7.1.8 Contracts are helpful in finding failures, used as documentation and oracle

Although our research has developed testing techniques that do not require contracts, we found that the effectiveness of automated testing could be further improved if the tools are given extra guidance in the form of pre-conditions, post-conditions and class invariants. Contracts are like assert statements and its violation i.e. when its value turn false indicate an error in the program. It also serves the purpose of documentation and you can tell on the fly what data a software take and the state before and after execution. The post-condition of a program also serves as an oracle.

Chapter 8

Future Work

8.1 Introduction

1. Extend tool to check all Java primitive data types: Current implementation checks only those Java classes which use integer data types, integer arrays, other classes which are checked, or arrays of these classes. This restricts the usability of the developed tool. This can be solved by including checking for other Java primitive data types. 2. Include other Java features: Current implementation restricts classes to have only fields, methods, and constructors. To extend this tool to check all Java programs it is required to check all features offered by Java. 3. Experiment with other static checkers: Although ESC/Java2 is a powerful static checker it makes very strong assumptions about Java which results in warning of many of the correct invariants suggested by Daikon. A static checker which results in less number of false warnings is required. 4. Generate test cases which ensure code coverage: Current random test case generator does not guarantee code coverage which forces manual inspection of invariants generated to check if behaviour of program is conveyed properly. The reliance on manual inspection can be reduced if test cases can guarantee code coverage.

Appendix A

A.1 Sample code to identify failure domains

Program 1 Program generated by ADFD on finding fault in SUT

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

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

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

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

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

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

Program 2 Point domain with One argument

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

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

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

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

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

Program 3 Point domain with two argument

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

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

}
```

Program 4 Block domain with one argument

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

public class BlockDomainOneArgument{

public static void blockErrors (int x){

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

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

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

   }
}
```

Program 5 Block domain with two argument

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

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

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

    }

}
```

Program 6 Strip domain with One argument

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

public class StripDomainOneArgument{

    public static void stripErrors (int x){

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

Program 7 Strip domain with two argument

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

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

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

```
        }

    }
```

References

[1] 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] 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.

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

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

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

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

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

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

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

[17] 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.

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

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

[20] Dave Towey. Software quality assurance.

[21] 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.

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

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

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

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

[26] 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.

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

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

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

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

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

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

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

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

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

[36] 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.

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

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

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

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

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

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

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

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

[45] 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.

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

[47] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a. IEEE, 2007.

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

[49] 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.

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

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

[52] 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.

[53] 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.

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

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

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

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

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] 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.

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

[69] 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.

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

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

[72] 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.

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

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

[75] 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.

[76] 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.

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

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

[79] 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.

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

[81] 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.

[82] 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.

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

[84] 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.

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

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

[87] 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.

[88] 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.

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

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

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

[94] 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.

[95] 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.

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

[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] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

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

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

[120] 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.

[121] Lee J White and Edward I Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.