New Strategies for
Automated Random Testing

Mian Asbat Ahmad

PhD

University of York

Computer Science

September 2014

Abstract

The ever increasing reliance on software-intensive systems is driving research to discover software faults more effectively and more efficiently. Despite intensive research, very few approaches have studied and used knowledge about fault domains to improve the testing or the feedback given to developers. The present thesis addresses this shortcoming: it leverages fault co-localization in a new random testing strategy called Dirt Spot Sweeping Random (DSSR), and it presents two new strategies: Automated Discovery of Failure Domain (ADFD) and Automated Discovery of Failure Domain$^+$ (ADFD$^+$). These improve the feedback given to developers by deducing more information about the failure domain (i.e. point, block, strip) in an automated way. The DSSR strategy adds the value causing the failure and its neighbouring values to the list of interesting values for exploring the underlying failure domain. The comparative evaluation showed significantly better performance of DSSR over Random and Random$^+$ strategies. The ADFD strategy finds failures and failure domains and presents the pass and fail domains in graphical form. The results obtained by evaluating error-seeded numerical programs indicated highly effective performance of the ADFD strategy. The ADFD$^+$ strategy is an extended version of ADFD strategy with respect to algorithm and graphical presentation of failure domains. In comparison with Randoop, ADFD$^+$ strategy successfully detected all failures and failure domains while Randoop identified individual failures but could not detect failure domains. The ADFD and ADFD$^+$ techniques were enhanced by integration of the automatic invariant detector Daikon, and the precision of identifying failure domains was determined through extensive experimental evaluation of real world Java projects contained in a database, namely Qualitas Corpus. The analyses of results, cross-checked by manual testing indicated that the ADFD and ADFD$^+$ techniques are highly effective in providing assistance but are not an alternative to manual testing.

Contents

List of Figures

List of Tables

Dedication

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

Acknowledgements

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

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

I thank my assessor Prof. Dr. John Clark for his constructive feedback on various reports and presentations. I am highly indebted to Prof. Dr. Richard Paige for his generous help, cooperation and guidance throughout my research.

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

I obtained a Departmental Overseas Research Scholarship which is awarded on academic merit and research potential to overseas students for higher studies. I am truly grateful to the Department of Computer Science, University of York for extending all needed resources.

Declaration

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

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

- M. Ahmad and M. Oriol. Dirt Spot Sweeping Random strategy. *Poster presentation in fourth York Doctoral Symposium (YDS 2011)*, York, UK. October 2011.

- M. Ahmad and M. Oriol. Dirt Spot Sweeping Random strategy. In *Proceedings of the International Conference on Software and Information Engineering (ICSIE)*, Singapore. Lecture Notes on Software Engineering, Volume 2(4), pp. 294-299, 2014.

  Based on research described in Chapter 4 of the thesis.

- M. Ahmad and M. Oriol. Automated Discovery of Failure Domain. In *Proceedings of the International Conference on Software and Computer Applications (ICSCA)*, Paris, France. Lecture Notes on Software Engineering, Volume 1(3), pp. 289-294, 2014.

  Based on research described in Chapter 5 of the thesis.

- M. Ahmad and M. Oriol. ADFD$^+$: An Automatic Testing Technique for Finding and Presenting Failure domains. In *Proceedings of the International Conference on Software and Information Engineering (ICSIE)*, Singapore. Lecture Notes on Software Engineering, Volume 2(4), pp. 331-336, 2014.

  Based on research described in Chapter 6 of the thesis.

- M. Ahmad and M. Oriol. Evaluation of ADFD and ADFD$^+$ techniques. In *Proceedings of Seventh York Doctoral Symposium (YDS 2014)*, York, UK. October 2014.

  Based on research described in Chapter 7 of the thesis.

Chapter 1

Introduction

Software is a set of clearly defined instructions for computer hardware to perform a particular task. Some software are developed for use in simple day to day operations while others are for highly complex processes in specialised fields like education, business, finance, health, science and technology, etc. The ever-increasing dependency on software means that software products must be reliable, robust, safe and secure. However, like other man-made items, software products 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 large 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, and improvements in software testing infrastructure might save one-third of this cost [9]. Software testing is the most recognized and widely used technique to verify the correctness and ensure quality of the software [10]. Therefore, software companies make considerable efforts to ensure the reliability and accuracy of the software prior to its deployment. According to Myers et al. some software companies spend up to 50% of the elapsed time and more than 50% of the total development and maintenance cost on software testing [11]. The success of a software testing technique mainly depends on the number of faults discovered in the Software Under Test (SUT). An efficient testing process discovers the maximum number of faults in the minimum possible time. There is therefore a strong motivation for improving existing strategies and developing new efficient test strategies. This research study is a contribution to the literature on the subject with the

aim to improve software testing by devising new and improved automated software testing techniques based on random strategies.

Ideally, exhaustive testing, where software is tested against all possible inputs, may look more attractive, but it is commonly not feasible because of the large size of the input domain, limited resources and strict time constraints. The usual practice is therefore to use a test strategy for the selection of test data set from a large/infinite domain. Careful selection of the test data set, as a subset of the whole input domain, is a crucial factor in any testing technique because it represents the whole domain for evaluating the structural and/or functional properties of the SUT [12, 13]. Miller and Maloney were the first who comprehensively described a systematic approach of test data set selection known as path coverage. They proposed that testers should select the test data set so that all paths of the SUT are executed at least once [14]. The approach resulted in the higher standard of test quality.

Test data set can either be generated manually or automatically. Generating test data set manually is a time-consuming and laborious exercise [15] as compared to the more preferable automated test data set generation. Test data generators can be of different types i.e. Path-wise (Section 2.7.1), Goal-oriented (Section 2.7.2), Intelligent (Section 2.7.3), Search-based (Section 2.7.4) and Random (Section 2.7.5).

Based on the critical importance of relevant test data set, the testers should always use the most efficient test strategy during software testing. However, using such a test strategy involves higher cost to generate test data sets that satisfy complex constraints. The optimum approach maintains a balance between the resources required and the generation of relevant test data set to make the software testing cost effective. Random test data set generation is simple, widely applicable, easy to implement, faster in computation, free from bias and incurs minimum overhead [16].

1.1 Software Testing

Software testing is a Verification and Validation (V&V) technique used to ensure that the software adheres to the desired specifications. According to Edsger Dijkstra, "software testing can be used to show the presence of bugs but never to show the absence of bugs" [17]. It means that the 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. Software testing is discussed in more detail in Section 2.1.

1.2 Random Testing

Random testing is a testing technique in which the test data set is randomly generated in accordance with the requirements, specifications or other test adequacy criteria. The given SUT is executed against the test data set, 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 include data generation, execution and evaluation as shown in Figure 1.1. Random testing is discussed in more detail in Section 2.8.
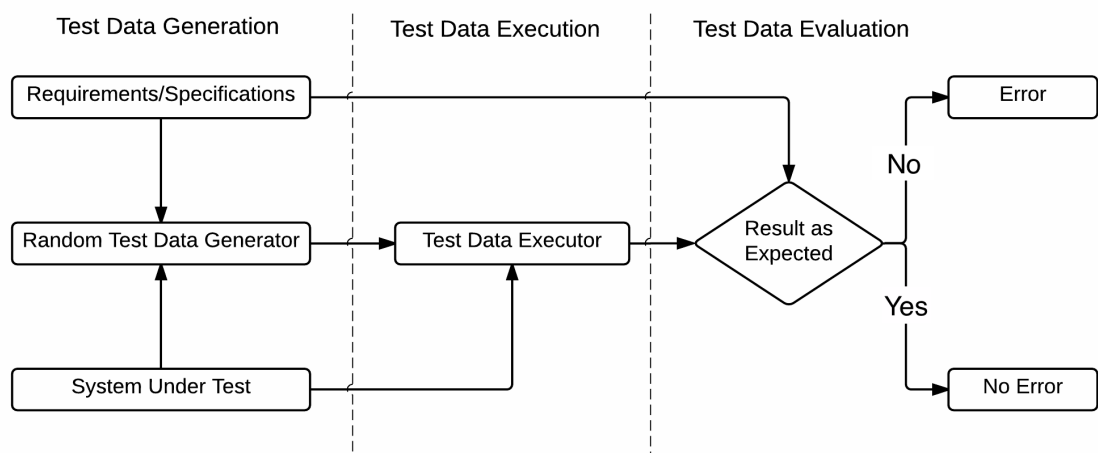
Figure 1.1: Three main phases of random testing

Test Data Generation: It consists of generation/selection of test data for use as input to the SUT. In this phase, the tester is free to generate the test data set arbitrarily from the input domain without any test adequacy criteria. Nevertheless, the two most commonly used methods are uniform distribution and operational profile. With a uniform distribution, all inputs are equally probable for selection. With an operational profile, the test data are selected to reflect the input distribution occurring in the operational environment.

Test Data Execution: It refers to application of generated test data to the SUT. The process consists of three steps: supplying test data as input to the software, executing the software and logging the output obtained. These steps can be achieved manually or automatically by using a script or tool.

Test Data Evaluation: It consists of analysis of the test logs to know whether the test fails or passes. The decision of fail/pass is made by comparing the obtained results with the correct results contained in the test oracle. The data evaluation can be performed either manually or automatically.

1.3 The Problems

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

1. Limitation of RT to discover contiguous unique failures.

2. Inability of RT to identify failure domains.

3. Lack of techniques to present RT results in graphical form.

1.3.1 Limitation of RT to Discover Contiguous Unique Failures

Chan et al. [1] observed that failure-inducing inputs are contiguous and form certain geometrical patterns in the whole input domain. They divided them into point, block and strip domains on the basis of their shape (see Section 2.9 for details). The failure-finding ability of random testing is quite low in detecting the contiguous block and strip failure domains within the input domain. Attempts are needed to overcome this limitation of RT.

1.3.2 Inability of RT to Identify Failure Domains

A majority of failures reside in contiguous locations and form certain failure domains across the input domain [1]. The existing random strategies of software testing try to discover failures individually from the domains and lack the capability to discover the failure domains. There is a need for appropriate random strategies with the potential to identify the failures, as well as failure domains.

1.3.3 Lack of Techniques to Present RT Results in Graphical Form

Random testing is no exception when it comes to the complexity of understanding and evaluating test results. Modern testing techniques simplify results by truncating the lengthy log files and displaying only the fault-revealing test cases in the form of unit tests. No random strategy seems to be used in conjunction with a graphical representation of the failures and failure domains. Efforts are therefore required to get the test results of random testing in user-friendly graphical form in addition to the textual form.

1.4 Research Goals

Research goals of the current study are: to understand the nature of failures, to leverage failure domains for finding more bugs, and to develop new improved automated random strategies.

1.5 Contributions

The main contributions of the thesis research are as follows:

Dirt Spot Sweeping Random Strategy: The failure-finding ability of random testing techniques decreases when the unique failures lie in contiguous locations across the input domain. Dirt Spot Sweeping Random (DSSR) strategy was developed as a new automated technique to overcome the problem. It is based on the assumption that unique failures reside in contiguous blocks and strips. When a failure is identified, the DSSR strategy selects neighbouring values for the subsequent tests. The selected values sweep around the failure, leading to the discovery of new failures in the vicinity. Results presented in Chapter 4 indicate that the failure-finding ability of DSSR is greater than that of Random and Random$^+$ strategies.

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

Automated Discovery of Failure Domain$^+$: The Automated Discovery of Failure Domain$^+$ (ADFD$^+$) is developed as an upgraded version of ADFD with respect to the algorithm and graphical representation of failure domains. The new algorithm searches for the failure domain around the failure in a given radius as against ADFD, which limits the search between lower and upper bounds in particular directions. In addition,

ADFD output is improved to provide labelled graphs that make the output easily understandable and user-friendly. The performance of ADFD$^+$ is compared with that of an automated testing tool Randoop on similar programs. The experimental results are presented in Chapter 6.

Evaluation of ADFD and ADFD$^+$ using Qualitas Corpus: Extensive experimental analysis was carried out on Java projects contained in the Qualitas Corpus [20] to find the effectiveness of the two automated techniques (ADFD and ADFD$^+$) in comparison with manual technique. The impact of nature, location, size, type and complexity of failure domains on the testing techniques was also studied. The results obtained are presented in Chapter 7.

1.6 Thesis Structure

The rest of the thesis is organized as follows:

Chapter 2 provides a literature review on software testing. 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. Different 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. Aspects of YETI are thoroughly reviewed including design, core infrastructure, strategy, language-specific binding, construction of test cases, command line options, execution, test oracle, report generation and graphical user interface.

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

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

Chapter 6 presents the Automated Discovery of Failure Domain$^+$ (ADFD$^+$) strategy. It is an upgraded version of the ADFD technique with respect to the algorithm and graphical representation of failure domains. To find the effectiveness of ADFD$^+$, we compared it with the automated random testing tool Randoop using error seeded programs. The experimental results are presented.

Chapter 7 presents evaluation of the precision of identifying failure domains by ADFD and ADFD$^+$. For the purpose of comparative analysis, Daikon has been integrated in the two techniques and extensive experimental analyses are performed of real world Java projects contained in Qualitas Corpus. The results obtained are analysed and cross-checked with the results of manual testing. The impact of nature, location, size, type and complexity of failure domains on the testing techniques are considered.

Chapter 8 presents conclusions of the study including contributions and the lessons learned.

Chapter 9 highlights the opportunities for future work, challenges that may be faced and possible approaches to overcome the challenges.
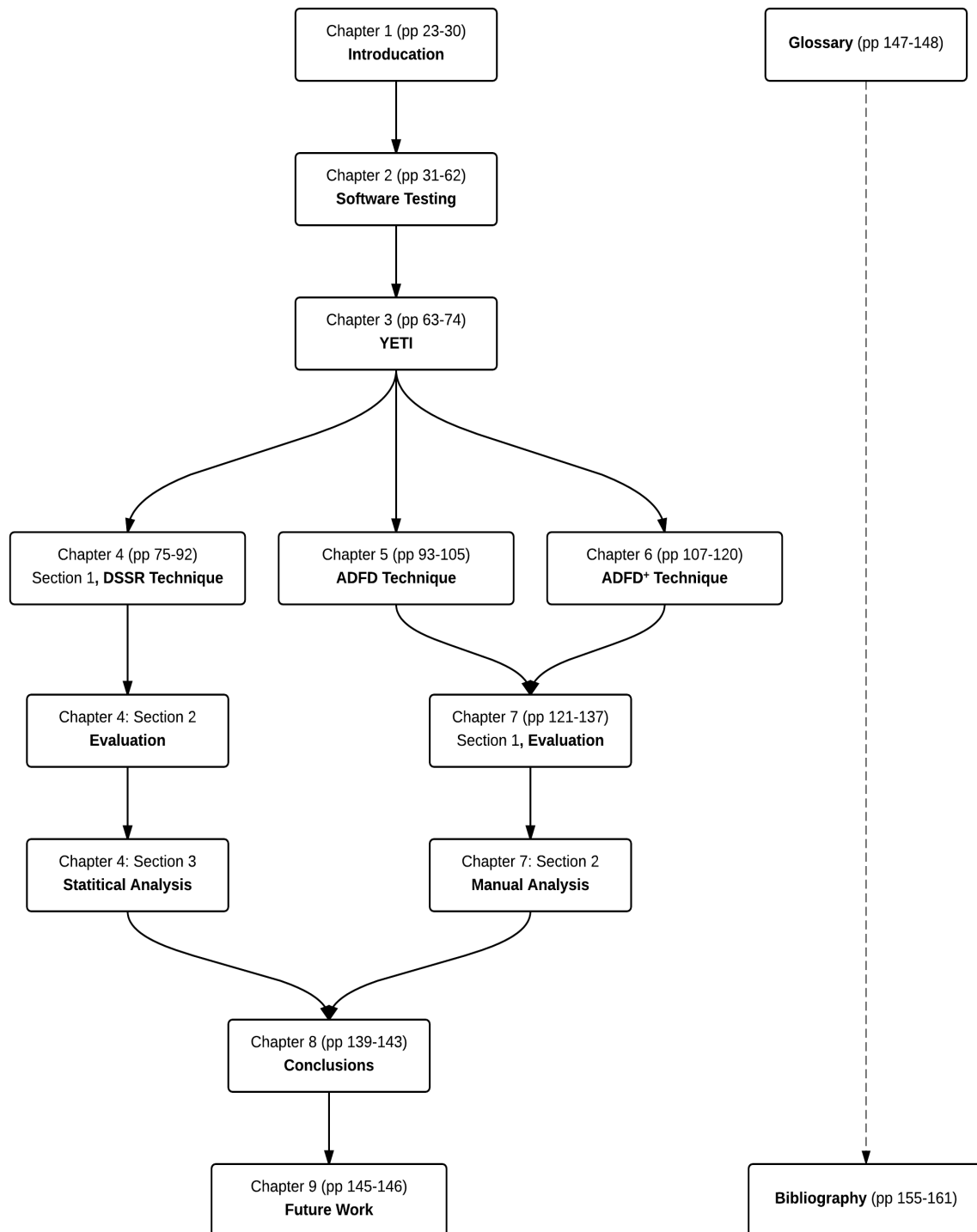
Figure 1.2: Structure of the thesis outline

Chapter 2

Literature Review

The famous quote of Paul Ehrlich, "to err is human, but to really foul things up you need a computer", is quite relevant to the software programmers. Programmers being humans are prone to errors. In spite of best efforts, some errors may remain in the software after it is finalised. In some systems a single error may lead to a catastrophe. The destruction of the Mariner 1 rocket (1962) costing \$18.5 million, the Hartford Coliseum Collapse (1978) costing \$70 million, the Wall Street crash (1987) costing \$500 billion, the failure of long division by the Pentium^{TM} chip (1993) costing \$475 million, the Ariane 5 Rocket disaster (1996) costing \$500 million, and many more were caused by minor errors in the software [21]. According to the National Institute of Standards and Technology, US companies alone bear \$59.5 billion loss every year due to software faults, one-third of which can be eliminated by improving the testing infrastructure [9]. Software has to undergo rigorous stages of testing to achieve high quality. The more complex the software, the higher the requirements for software testing because of the consequent larger damage involved if a fault remains in the software.

2.1   Software Testing

According to the ANSI/IEEE standard glossary of software engineering [22], testing is defined as, "the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results". The testing process is an integral part of the Software Development Life Cycle, which starts from requirement phase and continues throughout the life of the software. The Test Plan document defines the goal, scope, method, resources and time schedule of testing [23]. It includes the testable deliverables and the associated risk assessment. The test plan explains `who, when, why` and `how` to perform a specific activity in the testing process.

In traditional testing, when a fault is found by the testers, the software is returned to the developers for rectification and the updated version is given back to the testers for retesting [24]. It is important to note that a successful test is the one that fails a software or identifies a fault in the software [11], where fault denotes an error made by programmers during software development [22]. The faulty code on execution can lead to software failures. Software that passes all tests is not guaranteed to be free from errors. However, the testing process increases confidence of users and reliability of the tested product [17].
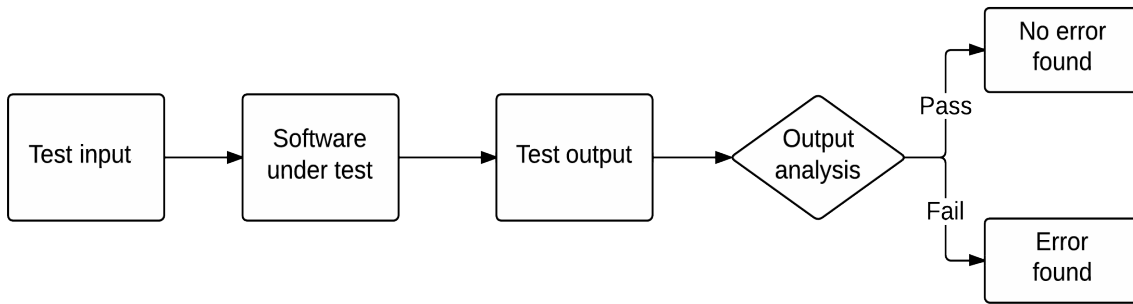
Figure 2.1: A simplified version of software testing process

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

2.1.1 Input Domain

The input domain comprises all possible inputs for a software, including all global variables, method arguments and externally assigned variables. For a given program P with input vector $P = \{x1,\ x2,\ ...,\ xn\}$, having $\{D1,\ D2,\ ...,\ Dn\}$ as the domain of each input so that $x1 \in D1$, $x2 \in D2$ and so on, the domain D of a function is the cross product of the domains of each input: $D = D1 \times D2 \times ... \times Dn$.

2.1.2 Test Case

Test case is an artifact which delineates the input, action and expected output corresponding to the input [25]. The test case is declared a pass if the output obtained after executing the test case agrees with the expected output. Alternatively the test case is declared a fail if the output obtained after executing the test case does not agree with the expected output. A test suite comprising a series of test cases is commonly executed to establish the desired level of quality.

2.1.3 Test Oracle

A test oracle is defined as, "a source containing expected results for comparison with the actual result of the SUT" [25]. 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 [12]. A test oracle sets the acceptable behaviour for test execution [26]. Software testing techniques depend on the availability of a test oracle [27]. Designing a test oracle for ordinary software may be simple and straightforward. However, for relatively complex software, designing an oracle is quite cumbersome and requires special expertise. Some common issues related to the design of test oracles are as follows:

1. It is assumed that the test results are observable and comparable with the oracle [28].

2. Ideally, a test oracle would satisfy desirable properties of program specifications [26].

3. "Truly general test oracles are often unobtainable" [28].

Postconditions are commonly used test oracles in automated software testing. Postconditions are conditions that must be true after a method is successfully executed. In such an oracle a fault is signalled when a postcondition is violated [29]. Some other common artefacts used as oracles are as follows:

1. Specification and documentation of software.

2. Products similar to the SUT but with different algorithms on implementation.

3. Heuristic algorithms to provide exact results for a set of test cases.

4. Comparison of the result of one test with another for consistency.

5. Generation of a model for verification of SUT behaviour.

6. Manual analysis by human experts to verify the test results.

2.2 Software Testing from Various Viewpoints

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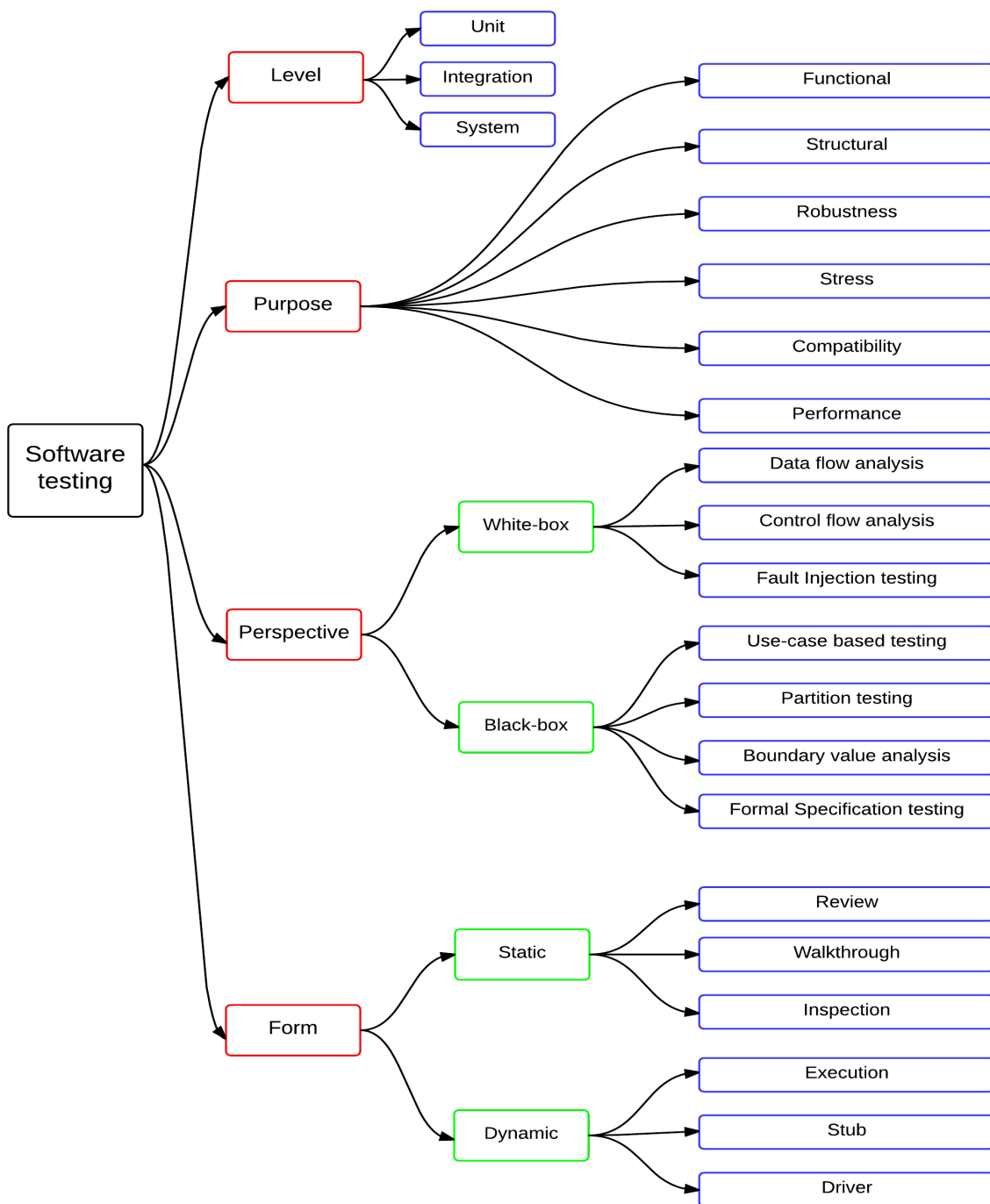
Figure 2.2: Software testing from various viewpoints

2.3 Software Testing Levels

The three main levels of software testing reported in the literature include Unit testing, Integration testing and System testing [30]. Unit testing deals with the evaluation of code piece-by-piece and each piece is considered as an independent unit. Integration testing is performed to make sure that the integration in the components formed by the combination of units is working properly. System testing ensures that the system formed by the combination of components proceeds properly to give the required output.

2.4 Software Testing Purpose

The purpose of software testing is to achieve high quality by identifying and eliminating faults in the given SUT. All faults can be identified if software is tested exhaustively. However, exhaustive testing is not always possible because of limited resources and the infinite number of input values that the software can take. Therefore, the purpose of testing is usually directed to achieve confidence in the system involved from a specific point of view. For example, functionality testing is performed to check that functional aspects of software are correct. Structural testing involves analyses of code structure for generating test cases in order to evaluate paths of execution and identification of unreachable or dead code. Robustness testing includes observation of the software behaviour when it receives input outside the expected range. Stress and performance testing aims at testing the response of software under high load [31]. Compatibility testing is performed to check and find any fault in the interaction of software with the underlying application or system software.

2.5 Software Testing Perspective

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

2.5.1 White-box Testing

White-box testing also known as structural testing is a technique that takes into consideration the structure of the software. Test cases are derived from the code structure and a test passes if the results are correct and the proper code is followed during test execution [32].

A simplified version of white-box testing is shown in Figure 2.3. Some commonly used white-box testing techniques are described below:
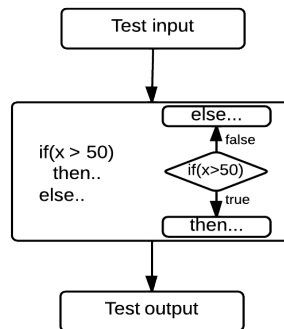
Figure 2.3: White-box testing

2.5.1.1 Data Flow Analysis

Data Flow Analysis (DFA) is a technique that focuses on the behaviour of variables during execution of the SUT [33]. In this technique a Control Flow Graph (CFG) representing all possible states of a program is drawn to determine the paths that are traversed by the program during test execution. Test cases are generated and executed to verify conformance with CFG on the basis of data flow.

The data flow analysis observes the program execution as data flow from input to output. The data may transform into several intermediary steps before reaching a final state. The transformation process is prone to several errors e.g. references made to nonexisting variables, values assigned to undeclared variables and change of variables in undesired manner. Ordered use of data is crucial to ensure that the aforementioned errors do not occur [34].

2.5.1.2 Control Flow Analysis

Control Flow Analysis (CFA) is a technique that takes into consideration the control structure of a given SUT [35]. Control structure is the order in which the statements, instructions and function-calls are executed. Like DFA, this technique also involves creating a CFG to determine the traversable paths by a program during the execution. Test cases are generated until all identified elements of control flow identified from the CFG have been executed. For example, it may be a requirement that the selected test cases execute all possible control choices at least once when two or more control choices are available to reach a particular state in the given SUT.

2.5.1.3 Fault Injection Testing

Fault injection testing is a validation technique to investigate the error handling behaviour of software, examine the capability of test procedure, and measure the code coverage achieved by the testing process [36]. The fault injection is usually software-based or hardware-based as stated below:

Software-based fault injection: In this technique faulty code is injected into the SUT at one or more locations to analyse the software behaviour in response to the faults [37]. For example, changing the value of a variable or return type of a method. The process of code addition (instrumentation) is performed before compilation and execution of software.

Hardware-based fault injection: In this technique faults are injected by disturbing physical environment of the system to analyse the system behaviour in response to the changed condition [38]. For example, injecting different voltage sags, introducing electromagnetic interference, and changes in temperature.

2.5.2 Black-box Testing

Black-box testing also known as functional testing is a technique that takes into consideration the function of the software. The testers may not know about the structure of the software. Test cases are derived from the software specifications and a test passes if the result agrees with the expected output irrespective of the internal code followed during test execution [39]. A simplified version of black-box testing is shown in Figure 2.4. Some commonly used black-box testing techniques are as follows.
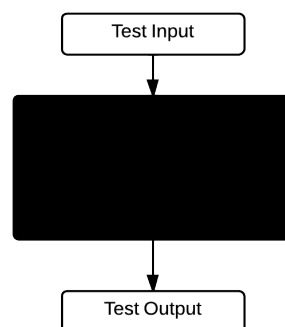
Figure 2.4: Black-box testing

2.5.2.1 Use-case Based Testing

This is a testing technique which utilizes use-cases of the system to generate test cases. A use-case defines the functional requirement at a particular point in the system from the actor's perspective. It consists of a sequence of actions to represent a particular behaviour of the system. A use-case format includes brief description, flow of events, preconditions, postconditions, extension points, context and activity diagrams. The use-case contains all the relevant information required for the test case. Therefore, it can be easily transformed into a test case [40]. Use-case testing is effective in terms of cheap generation of test cases, avoidance of test duplication, increased test coverage, easier regression testing and early identification of missing requirements.

2.5.2.2 Partition Testing

This is a testing technique in which the input domain of a given SUT is divided into sub-domains which are tested individually. The division is based on software specifications, code structure and the process involved in software development [41]. The performance of partition testing is directly proportional to the quality of the sub-domains [42]. Division of the input domain into equal partitions is often difficult. To overcome the problem, a new version of partition testing called proportional partition testing is devised [1]. In this version, the sub-domains vary in size, and the number of test cases selected from each partition is directly proportional to the size of the partition. Ntafos [43] have provided experimental evidence in support of better performance of proportional partition testing than partition testing.

2.5.2.3 Boundary Value Analysis

Boundary Value Analysis (BVA) is a testing technique based on the assumption that errors often reside along the boundaries of the input variables [44]. Thus, border values are taken as the test data set in BVA. According to IEEE standards [45], boundary values contain minimum, maximum, internal and external values specified for the system. The following code illustrates the ability of BVA to find a failure.

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

On passing the interesting value `Integer.MAX_VALUE` as argument to the `test` method, the code in the method increments it by 1 making it a negative value and thus a failure is generated when the SUT tries to set the array size to a negative value.

BVA and partition testing may be used in combination by choosing test values from the whole input domain and also from the borders of each sub-domain. Reid [46] has provided evidence in support of better performance of BVA in combination with partition testing as compared to each individually. They have attributed better performance to accurate identification of partitions and selection of boundary values.

2.5.2.4   Formal Specification Testing

This is a testing technique based on the mathematical model that provides the opportunity to handle the specifications mathematically. The mathematical notations express the formal specifications with precisely defined vocabulary, syntax and semantics. This feature facilitates isolation, transformation, assembly and repackaging of the information available in the specifications for use as test cases [47].

The formal specification testing is more productive because of the creation of test cases independent from the code of the SUT [27]. The effort of generating test oracle is avoided by using available specification model for verifying the test results [48]. The technique is highly effective in identifying errors, incompleteness and inconsistencies in software requirements [49].

2.6   Software Testing Types

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

Static testing: This involves the analysis of the SUT statically for checking errors without executing the test cases. Static analysis is applicable to evaluate the quality of software code and supporting documents including requirements, design, user manual, technical notes and marketing information. Reviews, walkthroughs and inspections are the most commonly used techniques for static testing [50].

Dynamic testing: This involves execution of test cases against the SUT. The results obtained are analysed against expected outputs to find errors in the software. Unit testing, integration testing, system testing and acceptance testing are the most commonly used methods for dynamic testing [50].

2.6.1 Manual Software Testing

In manual testing the testers themselves write code for test input generation and oracles [51]. Its advantage is that the test cases are usually generated to target those parts of the SUT, that are assumed to be more error-prone. It is highly useful to capture special cases which automated testing might miss. Manual unit testing is particularly popular because of the availability of the xUnit family of tools, e.g. jUnit for Java, sUnit for Smalltalk and pyUnit for Python. The tools automate the execution process of the hand written test cases providing significant practical benefits [4]. According to a survey, 79% of the Microsoft developers write unit tests for software testing [52].

The limitations of manual testing are that the testers should have appropriate skills, experience and knowledge of the SUT for its evaluation from different perspectives. Manual testing may be effective at smaller scale but at a larger scale it is generally laborious, time-consuming and error-prone [53]. Manual testing usually produces low coverage because large numbers of test cases are required for high coverage.

2.6.2 Automated Software Testing

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

A fully automated testing system is capable of testing software without any user intervention. This is usually achieved by utilizing the contracts (preconditions, postconditions and invariants) embedded in the program code. Preconditions are used to filter out invalid inputs and postconditions are used as the oracle [4]. Eiffel [54] and Spec# [55] languages have built-in contracts whereas Java can use add-ons like JML [56], iContract [57] or OCL [58]. 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 the job is repetitive, and it is performed on a routine basis, like unit testing and regression testing where the tests are re-executed after each modification [59]. The use of automated software testing makes it possible to test large volumes of code, which may be impossible otherwise [60].

2.7 Test Data Generation

Test data generation in software testing is the process of identifying and selecting input data that satisfies the given criterion. A test data generator is used to assist testers in the generation of data while the selection criterion defines the properties of test cases to be generated based on the test plan, and perspective taken [15]. 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 [61]. A program analyser performs the initial assessment of software prior to testing and may alter the code if so required. For example, it performs code instrumentation or construction of the CFG to measure the code coverage during testing. The strategy handler defines the test case selection criteria. This may include the formalisation of test coverage criterion, the selection of paths and normalisation of constraints. It may also get input from the program analyser or user before or during test execution.
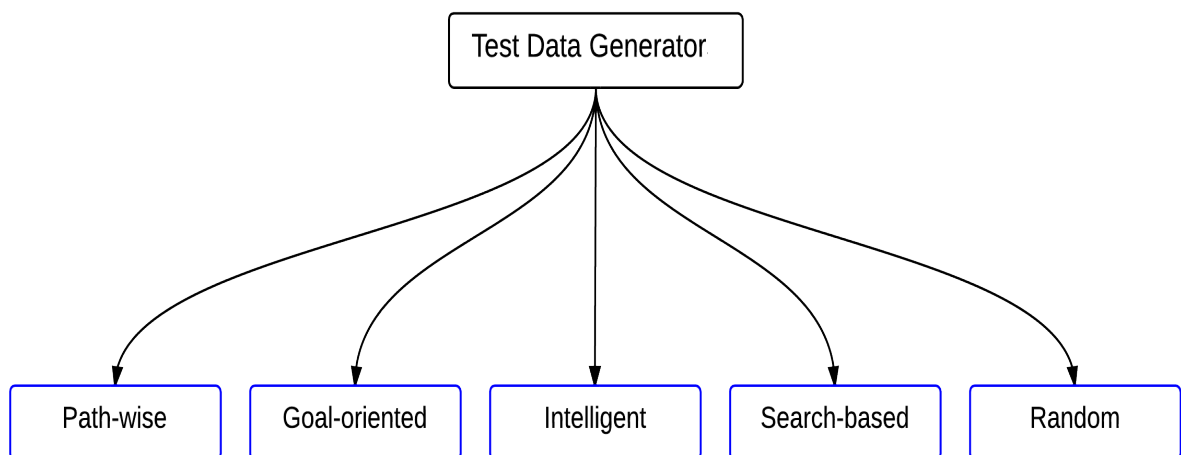
Figure 2.5: Types of test data generator

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

2.7.1 Path-wise Test Data Generator

A path-wise test data generator selects a set of test data from the input domain in order to target path, branch and statement coverage in a given SUT. The process is typically accomplished in three stages: CFG construction, path selection and test data generation. In the process, a particular path is selected either manually or by automatic means. The generator identifies and generates relevant test data required for execution of the intermediary statements along the selected path. A complete path contains multiple sub-domains, each consisting of test inputs required to traverse the path. The boundary of sub-domains is obtained by the predicates in the path condition.

2.7.2 Goal-oriented Test Data Generator

A goal-oriented test data generator generates data to target a specific program point [62]. 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 [63]. Among various methods used in goal-oriented test data generation the following two commonly adopted approaches are briefly described.

2.7.2.1 Chaining Approach

The chaining approach uses the data-dependent analysis to guide the test data generation. In the process, all the related statements affected by execution of the statement under test are selected automatically. The dependent statements are executed before the selected statement to generate the required data for the execution of the statement under test [63]. 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 $E = < (s, \varnothing), (p, \varnothing), (q, \varnothing) >$, provided that s is the starting node of the program and $\varnothing$ is the set of variables referred to as constraint. The Branch classification process identifies critical, semi-critical and non-critical nodes for each branch. During execution, the classification guides the search process to select specific branches to reach the goal node.

2.7.2.2 Assertion-oriented Approach

The assertion-oriented approach adds assertions to the program code with the goal to identify program input on which an assertion is violated indicating a fault in the SUT [64]. An assertion is a constraint applicable to a state of computation which can be either true or false. For example, consider a given assertion A, now find program input x on which assertion A is false, i.e. when the program is executed on input x and the execution reaches assertion A, it is evaluated as false indicating a fault in the SUT. It is not always possible to generate test cases that violate assertions. However, experiments have shown that assertion-oriented test data generation may frequently detect errors in the program related to assertion violation. The major advantage of this approach is that each generated test data uncovers an error in the program with violation of an assertion. An assertion is violated due to fault in program code, in pre or postcondition or a fault in the assertion itself.

2.7.3 Intelligent Test Data Generator

Intelligent test data generator is used to overcome the problems like generation of meaningless data, duplicated data and complex data associated with traditional data generators. The approach increases user confidence in the generated test data and the testing process [60]. It helps in finding the appropriate test data by performing sophisticated analysis to anticipate different situations that may arise at any point in the SUT such as genetic algorithm. The approach produces test data that satisfy the SUT requirements but consumes more time and resources.

2.7.3.1 Genetic Algorithm

Genetic algorithm is a heuristic that mimics the evolution of natural species for searching optimal solution of the problem. It is guided by control dependencies in the program to search for data that satisfy test requirements. The algorithm evaluates the existing test data and guides the direction of search by using the program control-dependence graph [65]. The approach emphasises on the execution of a given statement, branch, path and condition in the given SUT. The benefit of the genetic approach is quick generation of test cases with proper focus and direction. The new test cases are generated by applying simple operations on the appropriate existing test cases having good potential of satisfying the test requirements. The success of the approach depends heavily on the way in which the existing test data are measured [65].

2.7.4 Search-based Test Data Generator

Search-based test data generation is mentioned in the literature for the first time by Webb Miller and David Spooner in 1976 [66]. It uses meta-heuristic algorithms to generate test data. The technique relies on the problem-specific fitness function for the optimization process. Fitness function guides the search process to find an appropriate solution from a large or infinite search space in a given time [67]. Hill-climbing, simulated annealing, and genetic algorithms are the most common optimization algorithms used in search-based test data generation. Search-based test data generation technique can reduce time and effort by automatically generating relevant test cases. It has been successfully applied, especially in structural testing [67, 68].

In search-based test data generation technique, each input vector `arg1` is associated with a measure `cost(arg1)` which represents the difference between the input value `arg1` and a set goal. Input value closer to the goal has low-cost value as against the input value away from the goal. Let us consider the following program.

```
public void test(int arg1, int arg2) {
    if (arg1 >= 20) {
        arg2 = arg3;
    }
    else {
        arg2 = 2 * arg3;
    }
}
```

Suppose in the if/else branch statement, we want the true condition to be executed. An input value of `arg1 == 25` clearly satisfies the predicate, and a value of `arg1 == 15` is closer to satisfy the predicate than a value of `arg1 == 5`. We evaluate a cost function probe of the form `cost(arg1) = max {0, 20 - arg1}`. Thus `arg1 == 25` has cost 0, `arg1 == 15` has cost 5 and `arg1 = 5` has cost 15. It is apparent that finding data to satisfy the branch predicate is essentially a search over the input domain of `arg1` to find a value such that `cost(arg1) == 0`. The data that satisfy each of the predicates at different points may be selected to follow a particular path in the code. This leads to a cost function which combines cost at each of the relevant branching point. The cost function plays the role of oracle for each targeted test requirement. Consequently, the cost function must change as per requirement. Frequent re-instrumentation of program is needed to find test data that fully satisfy common coverage criteria.

2.7.5 Random Test Data Generator

Random test data generator is used as the simplest approach for generation of test data. In this technique test data set is generated randomly from the input domain for testing software. The test data set is in accordance with the software requirements, software specifications or any other test adequacy criteria. Its advantage is the adaptability to generate input data for any type of program. However, random test data generation is based solely on probability and cannot accomplish high coverage as its chances of finding semantically small faults are quite low [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:

```java
public void test(int arg1, int arg2) {
    if (arg1 == arg2)
        System.out.println(arg1 + " = " + arg2);
    else
        System.out.println(arg1 + " != " + arg2);
}
```

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

2.8 Random Testing

Random testing is mentioned in the literature for the first time by Hanford in 1970 who reported syntax machine, a tool that randomly generated data for testing PL/I compilers [69]. Later in 1983, Bird and Munoz described a technique to produce randomly generated self-checking test cases [70]. Random testing is a dynamic black-box testing technique in which the software is tested with non-correlated unpredictable test data from the specified input domain [71]. As stated by Richard [72], the test data are randomly selected from the identified input domain by means of a random generator. The program under test is executed on the test data and the results obtained are compared with the program specifications. The test case fails if the results are not according to the specifications reflecting a fault in the given SUT and vice versa. Generating test data by random generator is economical and requires less intellectual and computational efforts [73]. Moreover, no human intervention is involved in data generation that ensures an unbiased testing process. The working mechanism of random testing is shown in Figure 2.6.
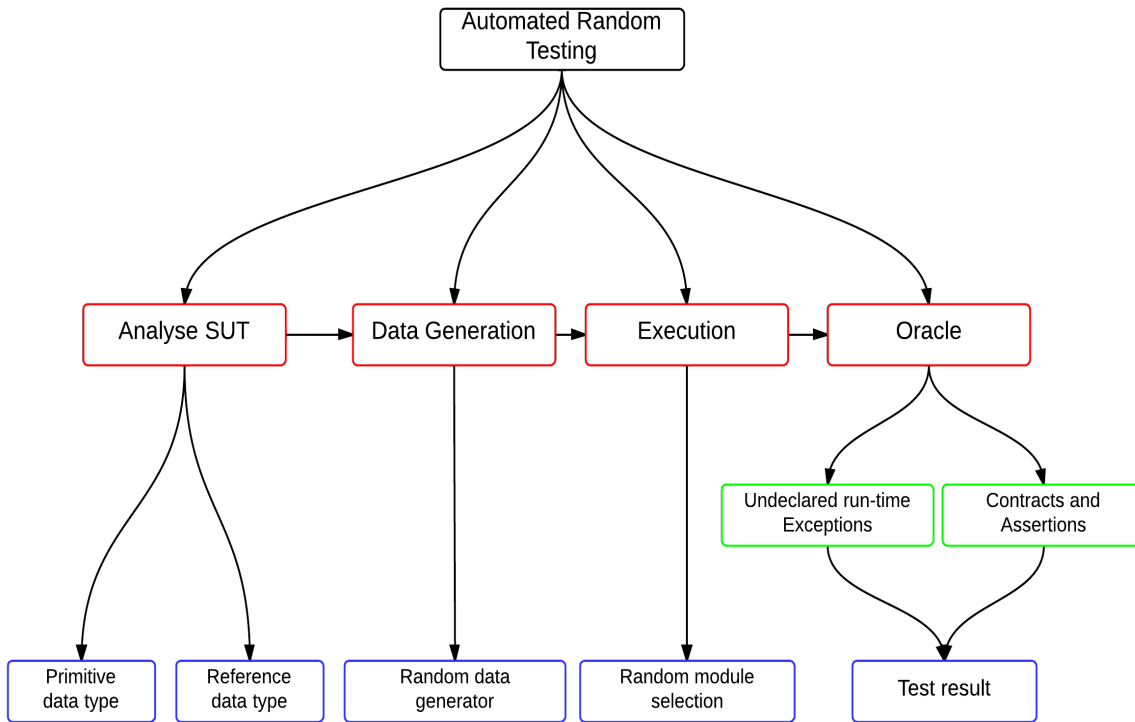
Figure 2.6: Working mechanism of random testing

The generation of test data without 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 violate preconditions of the given SUT, which makes it less effective [74, 75]. Myers mentioned random testing as one of the least effective testing technique [11]. However, Ciupa et al. [16] stated that Myers statement is not based on any experimental evidence. Later experiments performed by several researchers [51, 72, 76] confirmed that random testing is as effective as any other testing technique. It is reported [77] that random testing can also discover subtle faults in a given SUT when subjected to a large number of test cases. It is pointed out that the simplicity and cost effectiveness of random testing makes it more feasible to run large number of test cases as opposed to systematic testing techniques which require considerable time and resources for test case generation and execution. The empirical comparison shows that random testing and partition testing are equally effective [41]. Ntafos [43] conducted a comparative study and concluded that random testing is more effective than proportional partition testing. Miller et al. [78] generated random ASCII character streams and used the Unix utilities for abnormal terminating and non-terminating behaviours. Subsequently, the technique was extended to discover errors in the software running on X Windows, Windows NT and Mac OS X [79, 80]. Other famous

studies using random testing includes low-level system calls [81] and file systems used in missions at NASA [82]. The literature surveyed so far demonstrates that random testing is a well-established technique in both academia and the commercial domain.

2.9 Pass and Fail domains

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

1. Point domain: In the point failure domain, input inducing failures are scattered across the input domain in the form of stand-alone points. Example of point failure domain is the failure caused by the statement: $total = num1/num2$. Here a single value of `num2 = 0` can cause failure, when `num1`, `num2` and `total` are variables of type int.

2. Block domain: In the block failure domain, input inducing failures lie in close vicinity to form a block in the input domain. Example of block failure domain is failure caused by the statement: $if((num > 10) \&\& (num < 20))$. Here the values of `num` from 11 to 19 form a block failure domain.

3. Strip domain: In the strip failure domain, input inducing failures form a strip across the input domain. Example of strip failure domain is failure caused by the statement: $int[\,]\ myIntArray = new\ int[num1]$. Here multiple values of `num1` can lead to failure, i.e. when the value of `num1` is negative.

(a) Point domain   (b) Block domain   (c) Strip domain
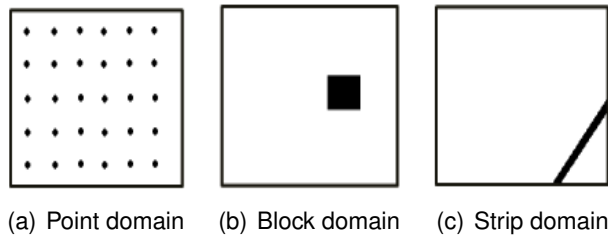
Figure 2.7: Failure domains across input domain [1]

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

2.10   Versions of Random testing

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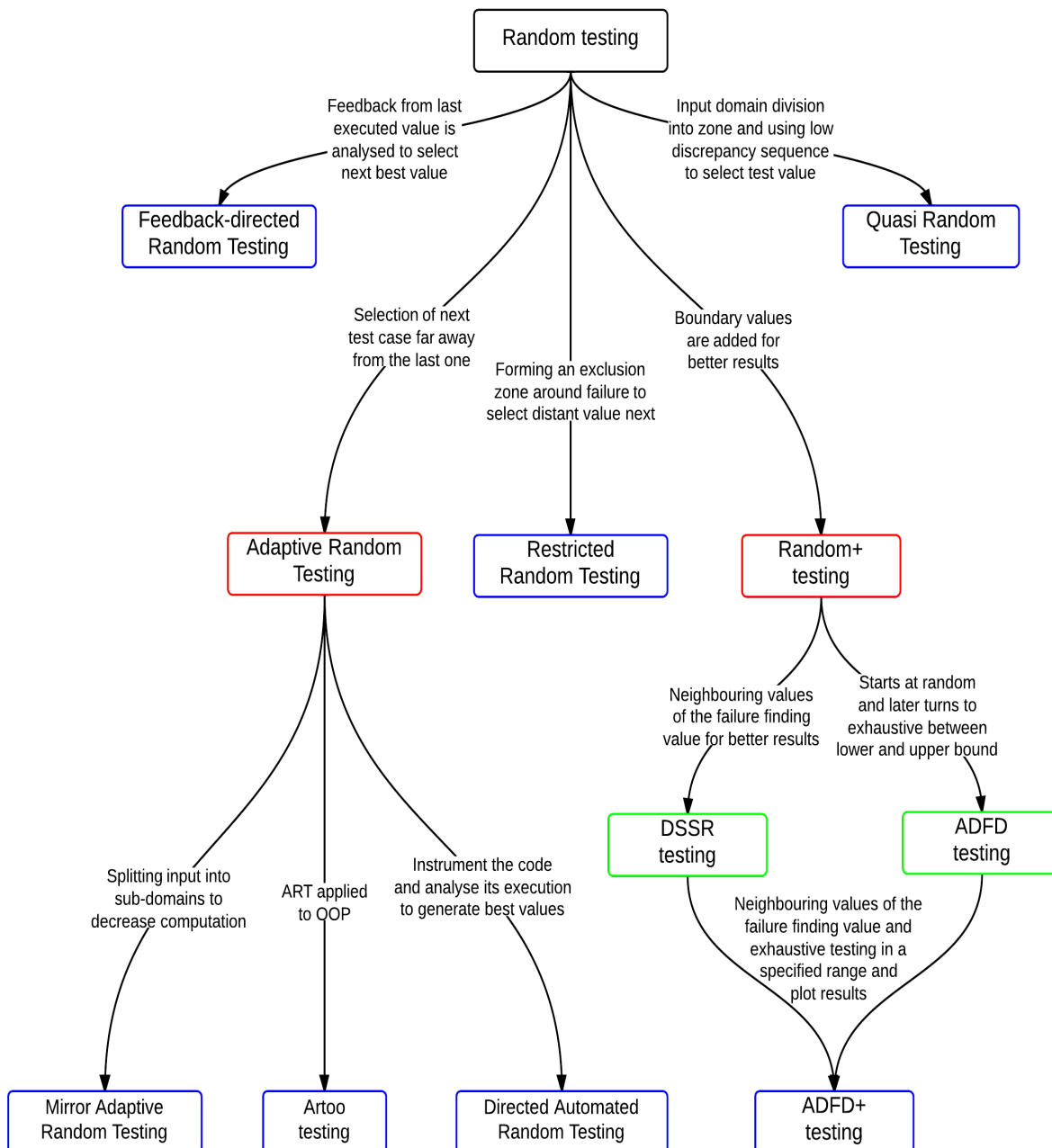
Figure 2.8: Various versions of random testing

2.10.1 Random$^+$ Testing

The random$^+$ testing strategy [16, 51] is an extension of the random testing. It uses some special predefined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [83] are the values at the start and end of a particular data type. For instance, such values for `int` could be -3, -2, -1, 0, 1, 2, 3, `Integer.MAX_VALUE`, `Integer.MAX_VALUE-1`, `Integer.MAX_VALUE-2`, `Integer.MIN_VALUE`, `Integer.MIN_VALUE+1`, `Integer.MIN_VALUE+2`. These special values can add a significant improvement to a testing method.

Let us consider the following piece of code:

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

In the given code, on passing the interesting value `MAX_INT` as argument, the code increments by 1 making it a negative value and thus the error is generated when the system tries to build an array of negative size. Similarly, the tester may add some other special values that are considered effective for finding faults in the SUT. For example, if a program under test has a loop from -50 to 50 then, the tester can add -55 to -45, -5 to 5 and 45 to 55 to the pre-defined list of special values. This static list of interesting values is manually updated before the start of the test. Interesting values included in the list are given higher priority than random values because of their relevance and better chances of finding faults in the given SUT. As reported in the literature, interesting values have high impact on the results, particularly for detecting problems in specifications [51].

2.10.2 Adaptive Random Testing

Adaptive random testing (ART) proposed by Chen et al. [24] is based on the previous work of Chan et al. [1] regarding the existence of failure domains across the input domain (Section 2.9). Chen et al. [24] argued that ordinary random testing might generate test inputs lurking too close or too far from the input inducing failure and thus fail to discover the fault. To generate more fault-targeted test inputs, they proposed ART as a modified version of random testing where test values are selected at random as usual but are evenly spread across the input domain. The technique uses the candidate set as well as the executed set both of which are initially empty and as soon as the testing begins ART fills the candidate

set with randomly selected test cases from the input domain. The first test case selected at random from the candidate set is executed and stored in the executed set. The second test case is the one selected from the candidate set which is located far away from the previously executed test case. The process continues till test completion and provides greater chances of finding failures from failure domains as indicated in [24].

In their experiment with ART Chen et al. [24] use the number of test cases required to find the first fault (F-measure) as a performance metric instead of the traditional probability of detecting at least one failure (P-measure) and expected number of failures detected (E-measure). The results showed up to 50% increase in performance compared to random testing. However, the authors raised concerns regarding the spreading of test cases across the input domain, the efficiency of selecting test cases, and the high overhead cost of generating tests.

2.10.3   Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) is an improvement on ART by using mirror-partitioning technique to reduce the overhead and decrease the extra computation involved in ART [84].
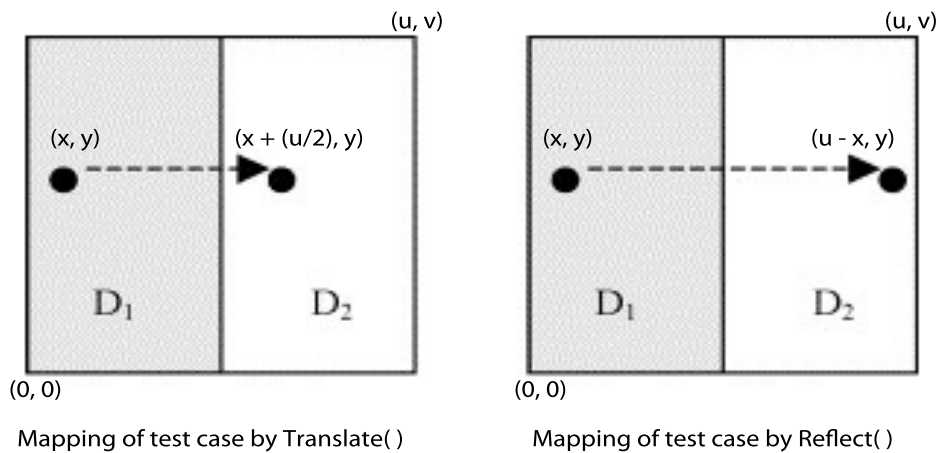
Figure 2.9: Mirror functions for mapping of test cases

In this technique, the input domain of the program under test is divided into n number of disjoint sub-domains of equal size and shape. One of the sub-domains is called the source sub-domain while all others are termed as mirror sub-domains. ART is then applied only to the source sub-domain while test cases are selected from all sub-domains by using a 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 rectan-

gle. On splitting it into two sub-domains we get $(0,0),(u/2,v)$ as source sub-domain and $(u/2,0),(u,v)$ as mirror sub-domain. Suppose we get x and y test cases by applying ART to source sub-domain, so we can linearly translate these test cases to achieve the mirror effect, i.e. $(x+(u/2),y)$ as shown in Figure 2.9. The experimental results by comparing MART with ART provides evidence of equally good performance of the two techniques with the added advantage of lower overhead in MART by using only one quarter of the calculation as compared with ART [84].

2.10.4   Restricted Random Testing

Restricted Random Testing (RRT) is another approach [85] to overcome the problem of extra overhead in ART. The RRT achieves this by creating a circular exclusion zone around the executed test case. A candidate is randomly selected from the input domain as a next test case. Before execution the candidate is checked and discarded if it lies inside the exclusion zone. This process repeats until a candidate present outside the exclusion zone is selected. It ensures that the test case to be executed is well apart from the last executed test case. The radius of exclusion zone is constant in each test case, and the area of the input domain decreases progressively with successive execution of test cases.
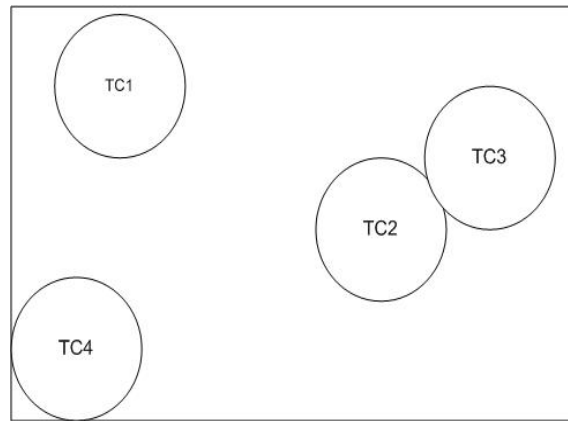
Figure 2.10: Input domain showing exclusion zones around selected test cases

The authors compared RRT with ART to find the comparative performance and reported that the performance of both techniques is similar with the added advantage of lower overhead in case of RRT, which uses only one-quarter of the calculations as compared with ART. They further found that RRT is up to 55% more effective than RT in terms of F-measure.

2.10.5 Directed Automated Random Testing

Godefroid et al. [18] proposed Directed Automated Random Testing (DART). In DART process, the given SUT is instrumented to track the dynamic behaviour of the SUT at run time. It also identifies external interfaces of a given SUT. These interfaces include external variables, external methods and the user-specified main method responsible for program execution. After that it automatically generates test drivers for running the randomly generated test cases. Finally, the results obtained are analysed in real time to systematically direct the test case execution along alternative path for maximum code coverage. The DART algorithm is implemented in the tool that is completely automatic and accepts only the test program as input. After the external interfaces are extracted it uses the preconditions and postconditions of the program under test to validate the test inputs. For languages that do not support contracts inside the code (like C), public methods or interfaces are used to mimic the scenario. DART attempts to cover different paths of the program code to trigger errors. Its oracle consists of checking for crashes, failed assertions and non-termination.

2.10.6 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) is a technique that generates a unit tests at random for object-oriented programs [86]. As the name implies, FDRT uses the feedback received from the execution of first batch of randomly selected unit tests to generate next batch of directed unit tests. In this way, redundant and wrong unit tests are eliminated incrementally from the test suite with the help of filtering and application of contracts. For example a unit test that produces *IllegalArgumentException* on execution is discarded because the arguments used in the unit test are not according to the required type. Pacheco et al. performed a case study in which a team of testers applied FDRT to a critical component of .NET architecture. Results showed that the faults discovered by FDRT in 15 hours of manual processing and 150 hours of CPU processing are more than a test engineer finds in one year by manual and other automated techniques. As a result, FDRT has been added to the tool list used at Microsoft for the betterment of the software [87].

2.10.7 The ARTOO Testing

The Adaptive Random Testing for Object-Oriented (ARTOO) programs technique is based on object distance. Ciupa et al. [88] defined the parameters that can be used to calculate the distance between the objects. Two objects have more distance between them if they have more dissimilar properties. The parameters specifying the distance between the objects are: dynamic types, values of primitive fields and values of reference fields. Strings

are treated in terms of directly usable values, and the Levenshtein formula [89] is used as a distance criterion between the two strings.

In the ARTOO testing, two sets are taken, the candidate-set containing the objects ready to be run by the system and the used-set, which is initially empty. The first object is selected randomly from the candidate-set that is moved to used-set after execution. The second object selected from the candidate-set for execution is the one with the largest distance from the last executed object in the used-set. The process continues till the fault is found or the objects in the candidate-set are finished [88].

The ARTOO technique, implemented in AutoTest [73], was evaluated in comparison with DART [18] by selecting classes from EiffelBase library [90]. The experimental results indicated that some faults found by the ARTOO technique were not identified by the DART technique. Moreover, the ARTOO technique found first faults with a small number of test cases than the DART technique. However, more computation was required to select a test case in the ARTOO technique and the process required more time and cost to generate test cases as compared to DART.

2.11 Automatic Random Testing Tools

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

2.11.1 JCrasher

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

The working mechanism of JCrasher is illustrated by testing a $.java$ program as shown in Figure 2.11. The source file is first compiled using $javac$ to get the byte code. The byte code obtained is passed as input to JCrasher, which uses Java reflection library [91] to analyse all the methods declared by class T. The JCrasher uses method's transitive parameter types P to generate the most appropriate test data set which is written to a file $TTest.java$. The file is compiled and executed by JUnit. All exceptions produced during test case executions are collected and compared with robustness heuristic and resulted violations are reported as errors.
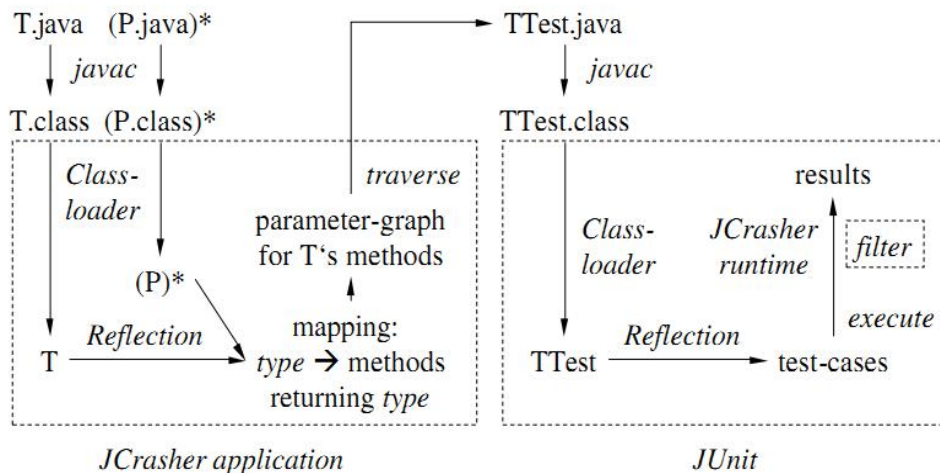
Figure 2.11: How a class T can be checked for robustness with JCrasher. First, the JCrasher application generates a range of test cases for T and writes them to TTest.java. Second, the test cases can be executed with JUnit, and third, the JCrasher runtime filters exceptions according to the robustness heuristic [2]

32

JCrasher is a pioneering tool with the capability to perform fully automatic testing, including test case generation, execution, filtration and report generation. Its novel feature is the generation of test cases as JUnit files that can be easily read and used for regression testing. Another important feature of JCrasher is independent execution of each new test on a clean-slate. This ensures that the changes made by the previous tests do not affect the new test.

2.11.2 Jartege

The Java random test generator (Jartege) is a testing tool [92] that randomly generates unit tests for Java classes with contracts specified in JML. The contracts include class invariants and the precondition and postconditions of methods. Initially, Jartege uses the contracts to eliminate irrelevant (pre-condition violating) test cases and later on the same contracts serve as test oracles (post-conditions). Jartege uses uniform random testing to test classes and generate test cases. Additionally, testing of a specific part of the class can be prioritized by changing the parameters to get interesting sequences of calls if so desired by the tester. The parameters include the following:

- Operational profile of the classes i.e. the likely use of the class under test by other classes.

- Weight of the class and method under test. Higher weight prioritizes the class or method over lower weight during the test process.

- Probability of creating new objects during the 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 preconditions and internal preconditions. Entry preconditions are the contracts to be met by the generated test data for testing the method while internal preconditions are the contracts which are inside the methods and their violations are considered as faults either in the methods or in the specifications. Jartege checks for faults in program code as well as in specifications and the JUnit tests produced by Jartege can be used later for regression testing. Its limitation is the prior requirement of Java Modelling Language (JML) specifications of the program.

2.11.3 Eclat

Eclat [3] is an automated testing tool which generates and classifies unit tests for Java classes. The tool takes a piece of software and a set of test cases for which the software runs properly. Based on the correct software operations an operational model is created to test the selected data. If the operational pattern of the test data differs from the model, the following three outcomes may be possible: (a) a fault in the given SUT (b) model violation despite normal operation (c) illegal input that the program is unable to handle.

The testing process is accomplished by Eclat in three stages as shown in Figure 2.12. In the first stage, a small subset of test inputs is selected, which may likely reveal faults in the given SUT. In the second stage, reducer function is used to discard any redundant input, leaving only a single input per operational pattern. In the third stage, the acquired test inputs are converted into test cases, and oracles are created to determine the success or failure of the test.
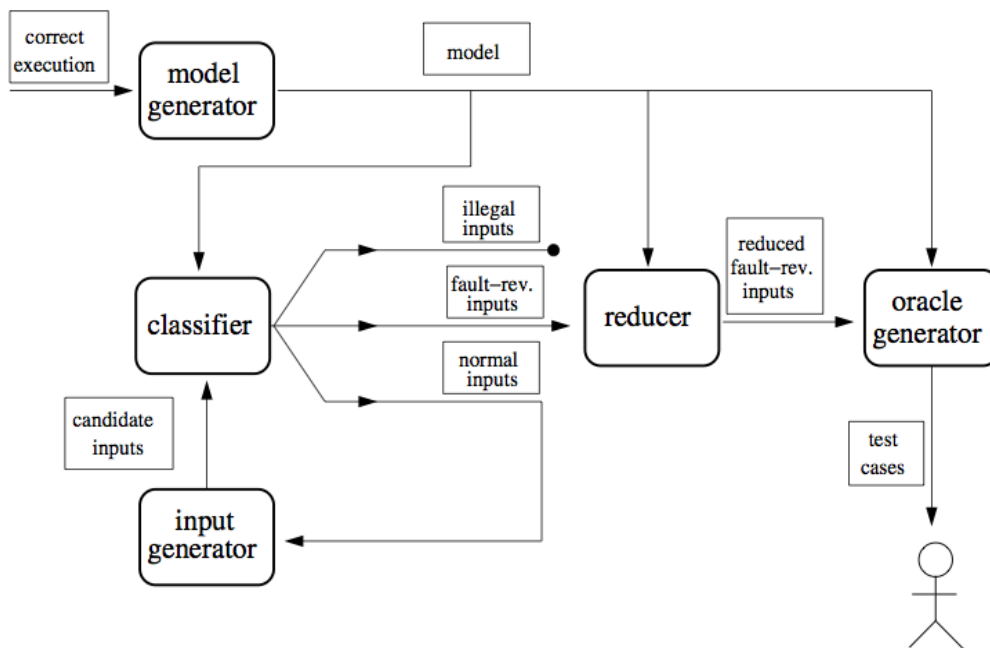
Figure 2.12: The input selection technique. Implicit in the diagram is the program under test. Rectangles with rounded corners represent steps in the technique, and rectangles with square corners represent artifacts [3]

To evaluate, they compared Eclat with JCrasher [2] 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 its dependence

on the initial pool of correct test cases. Any error in the pool may lead to the creation of wrong operational model which will adversely affect the testing process.

2.11.4 Randoop

Random tester for object-oriented programs (Randoop) is a tool used for implementing the FDRT technique [86]. Randoop is a fully automatic tool, capable of testing Java classes and .NET binaries. It takes a set of classes, contracts, filters and time limit as input and gives a suite of JUnit for Java and NUnit for .Net program as output. Each unit test in a test suite is a sequence of method calls (hereafter referred as sequence). Randoop builds the sequence incrementally by randomly selecting a public method from the class under test. Arguments for these methods are selected from a predefined pool in case of primitive type and as a sequence of null values in case of reference type. Randoop maintains two sets called *ErrorSeqs* and *NonErrorSeqs* to record the feedback. It extends *ErrorSeqs* set in case of contract or filter violation and *NonErrorSeqs* set when no violation is recorded in the feedback. The use of this dynamic feedback evaluation at runtime brings an object to an interesting state. On test completion, *ErrorSeqs* and *NonErrorSeqs* are produced as JUnit or NUnit test suite. The following command runs Randoop to test OneDimPointFailDomain for 100 seconds in CLI mode. Values is a text file containing interesting values which is maintained manually by the tester.

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

In terms of coverage and number of faults discovered, Randoop using the FDRT technique was compared with JCrasher and JavaPathFinder on 14 libraries of both Java and .Net programs [93]. The results showed that Randoop achieved more branch coverage and better fault detection than JCrasher.

2.11.5 QuickCheck

QuickCheck [94] is a lightweight random testing tool used for testing of Haskell programs [95]. Haskell is a functional programming language where programs are evaluated by using expressions rather than statements. Most of the functions in Haskell are pure except the IO functions, therefore QuickCheck mainly focuses on testing pure functions. QuickCheck is designed to have a simple domain-specific language of testable specifications embedded in Haskell. This language is used to define expected properties of the functions under test.

QuickCheck takes as inputs the function to be tested and properties of the program (Haskell

functions). The tool uses a built-in random generator to generate test data, but it is also capable to use a custom built data generator. The tester-defined properties must hold while executing the function on the generated test data. Any violation of the defined properties will indicate an error in the function.

2.11.6 AutoTest

AutoTest is used to test Eiffel language programs [96]. The Eiffel language uses the concept of contracts that is effectively utilized by AutoTest. For example, the auto generated inputs are filtered using preconditions and non-complying test inputs are discarded. The postconditions are used as test oracles to determine whether the test passes or fails. Besides automated testing, the AutoTest also allows the tester to write the test cases manually to target a specific section of the code. The AutoTest takes one or more methods or classes as inputs and automatically generates test input data according to the requirements of the methods or classes. As shown in Figure 2.13, the architecture of AutoTest can be split into the following four parts:
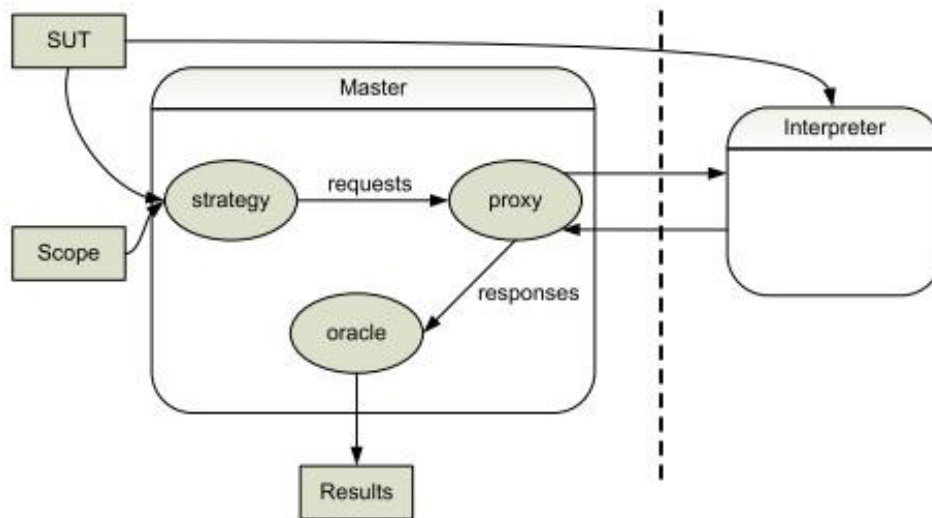
Figure 2.13: AutoTest architecture [4]

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

2. **Proxy:** This handles inter-process communication. The proxy receives execution requests from the strategy and forwards these to the interpreter. It also sends the execution results to the oracle part.

3. **Interpreter:** It executes operations on the SUT. The most common operations in-

clude: create object, invoke routine and assign result. The interpreter is kept separate to increase robustness.

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

2.11.7 TestEra

TestEra [5] is a novel framework of auto-generation and evaluation of test inputs for a Java program. Input of the tool includes specifications, numerical value and method to be tested. It uses preconditions of the method to generate all non-isomorphic valid test inputs within the specified limit. The test inputs are executed, and the results obtained are compared against postconditions of the method serving as oracles. A test case that fails to satisfy postconditions is considered as a fault. TestEra uses the Alloy modelling language [97] to
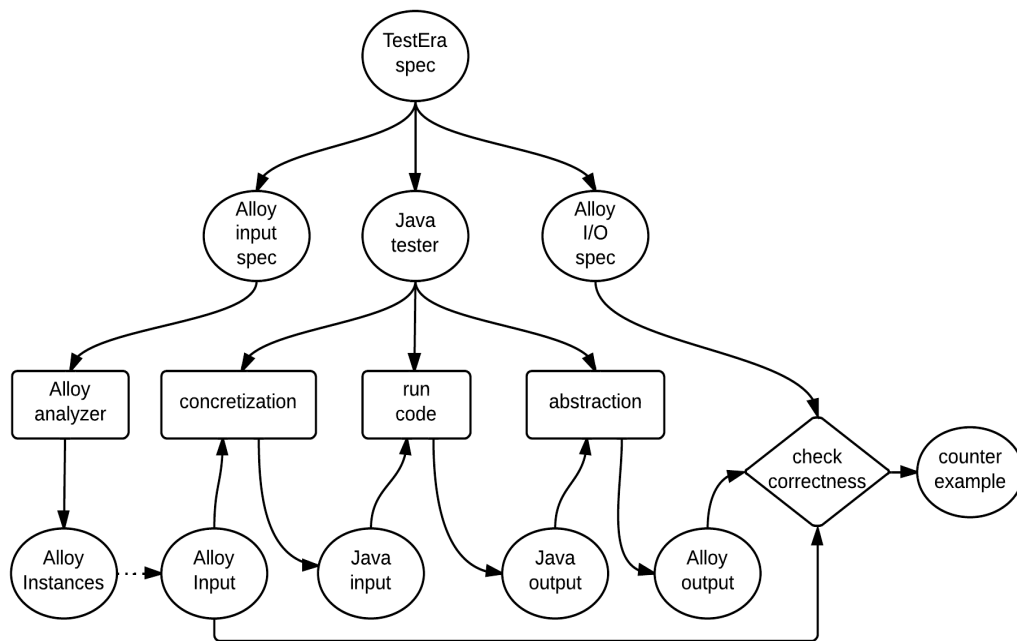
Figure 2.14: TestEra framework [5]

express constraints of test inputs from program specifications. The constraints are solved by the Alloy Analyser [98] which performs the following three functions: (a) it translates Alloy predicates into propositional formulas (b) it evaluates the propositional formulas to

37

find the outcome (c) it translates each outcome from propositional domain into the relational domain.

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

2.11.8 Korat

Korat [99] is a framework for automated testing of Java programs based on formal specifications [100]. Korat uses Java Modelling Language (JML) for specifications. It uses bounded-exhaustive testing in which the code is tested against all possible inputs within the specified bounds [101]. Input to the tool includes imperative predicates and finitization value. Korat systematically explores the input space of the predicates and generates all non-isomorphic inputs for which the predicates return true. The core part of Korat monitors execution of the predicates on candidate inputs to filter out the fields accessed during executions. These inputs are taken as test cases.

Korat uses `repOK()` and `checkRep()` methods. The `repOK()` is used to check the class invariants for validating test inputs while `checkRep()` is used to verify the postconditions for validating the test case. Korat does not require an existing set of operations to create input values. Therefore, it has the advantage of generating input values that may be difficult or impossible with a given set of operations. However, it requires significant manual effort [74].

2.11.9 YETI

York Extensible Testing Infrastructure (YETI) is an open-source automated random testing tool. YETI, coded in Java, is capable of testing systems developed in procedural, functional and object-oriented languages. Its language agnostic meta-model enables it to test programs written in multiple languages including Java, C#, JML and .NET. The core features of YETI include easy extensibility for future growth, capability to test programs using multiple strategies, high-speed tests execution, real-time logging, GUI support and auto-generation of test report at the end of test session. Detailed information about YETI is presented in Chapter 3.
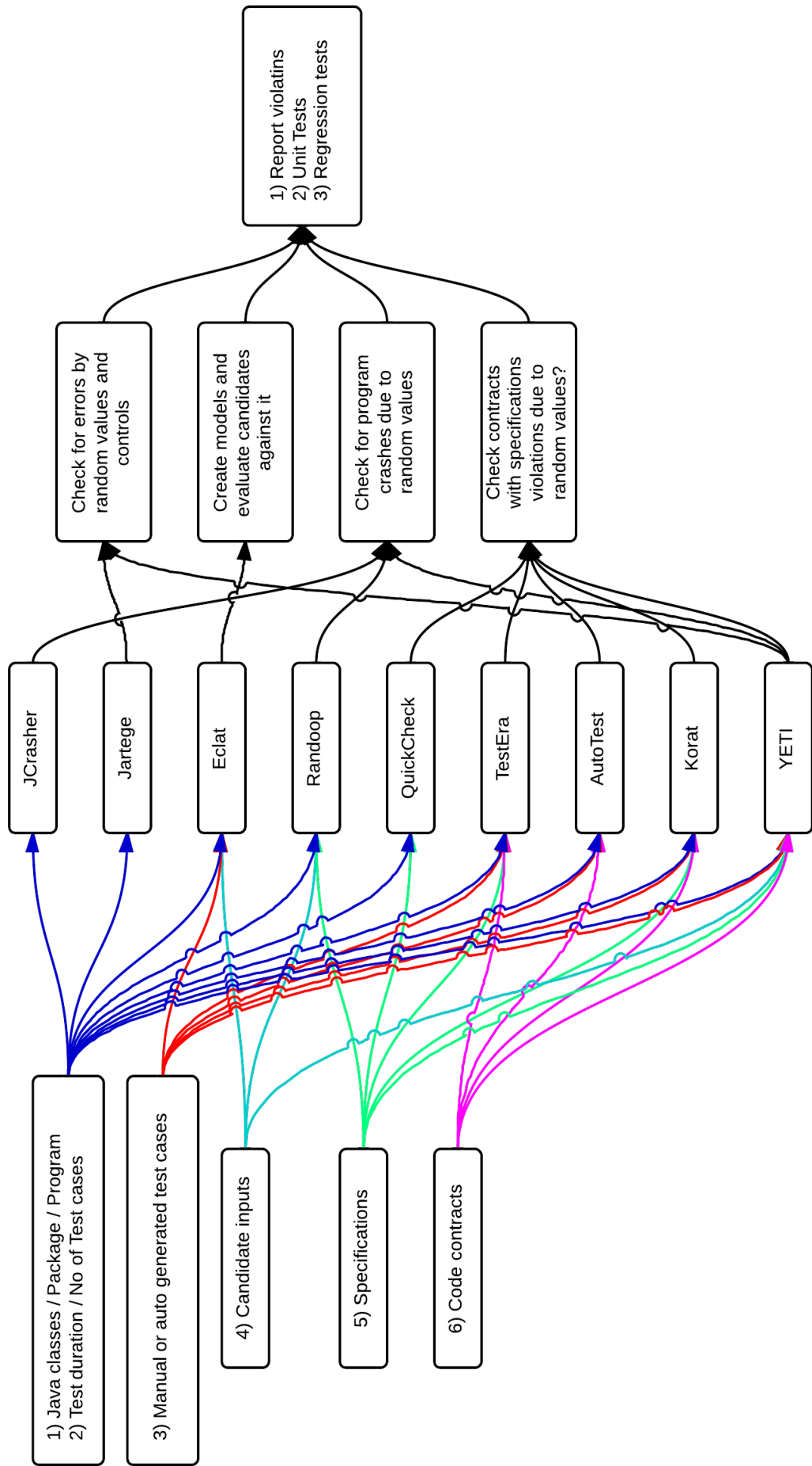
Figure 2.15: Main features of automatic testing tools using random testing

2.12 Summary

The software testing is summarized graphically in Figure 2.16.
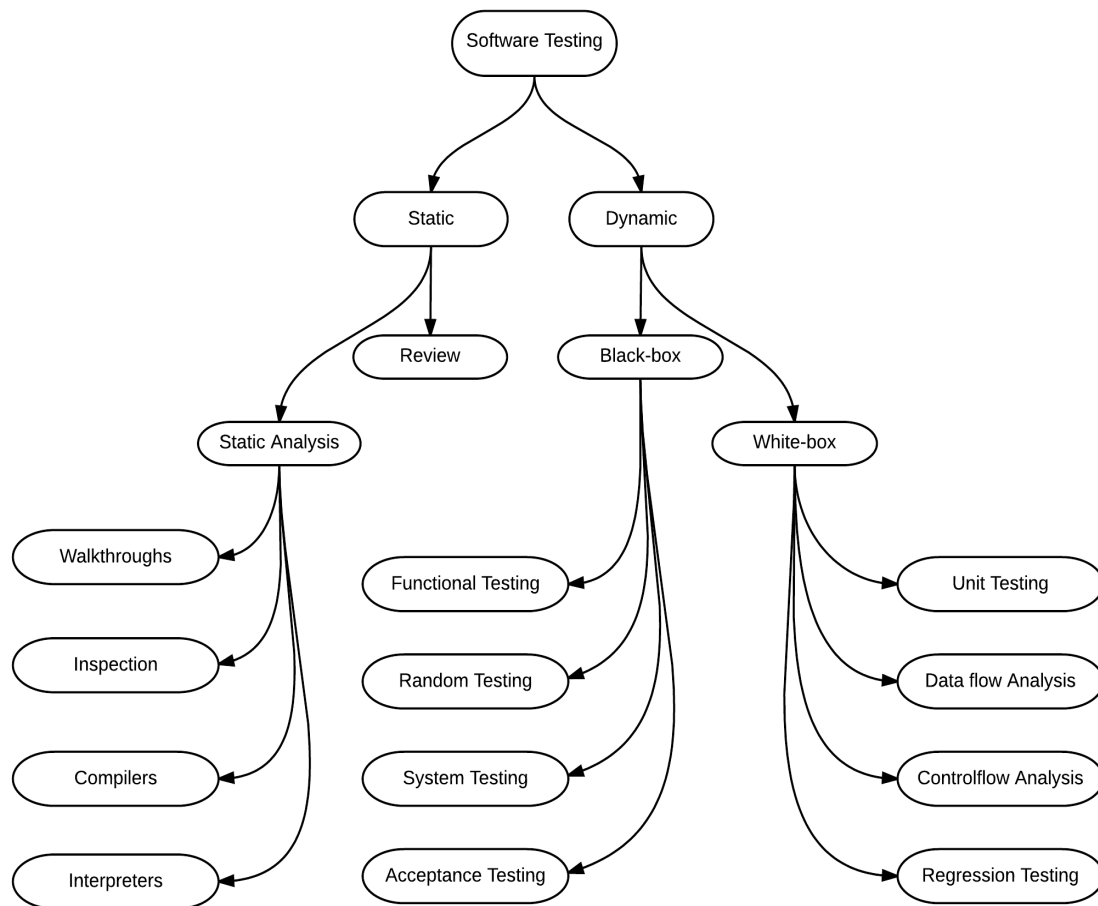
Figure 2.16: Types of software testing

The chapter gives an overview of software testing, including the definition, common types, need, purpose and uses. It differentiates manual and automated software testing and describes various ways of software test data generation. The later part describes random testing and the various ways of improving the performance of random testing. Finally, information is presented on how the automated testing tools implement random techniques for software testing. The main features of automatic testing tools used in random testing are summarized in Figure 2.15.

2.13 The Proposed Area of Research

The wide-ranging review of software testing and randomised software testing in particular has revealed significant scope for improving both the effectiveness and efficiency of randomised test data generation. In addition, there is significant scope to provide failure related information to developers that will allow them to fix identified problems with greater confidence and accuracy. In particular, identifying and communicating the nature of failure domains to developers should improve the overall quality of the testing process.

In this theses I aim to marry randomised testing with new approaches for discovering and presenting failure domains. My work is carried out using the YETI tool discussed fully in the next Chapter.

Chapter 3

York Extensible Testing Infrastructure

3.1 Overview

The York Extensible Testing Infrastructure (YETI) is an automated random testing tool developed by Manuel Oriol [102]. It is capable of testing programs written in Java, JML and .NET languages [103]. YETI takes the program byte-code as input and executes it with randomly generated syntactically-correct inputs to find a failure. It runs at a high level of performance with 10^6 calls on simple and efficient methods (i.e. methods from String etc) per minute on Java code. One of its prominent features is the Graphical User Interface (GUI), which makes YETI user-friendly and provides an option to change the testing parameters in real time. It can also distribute large testing tasks in the cloud for parallel execution [104]. The main motivation for developing YETI is provision of a testing workbench to testers and developers for research. YETI by design is easily extendable to facilitate inclusion of new languages and testing strategies. Several researchers [103, 104, 105, 106] have contributed various features and strategies to the YETI project. The current study extends YETI with three more test strategies i.e. DSSR [107], ADFD [108] and ADFD$^+$ [109] for software testing and with a graphical front-end to enable its execution from any GUI which supports Java. The latest version of YETI can be downloaded freely from www.yetitest.org. Figure 3.1 briefly presents the working process of YETI.
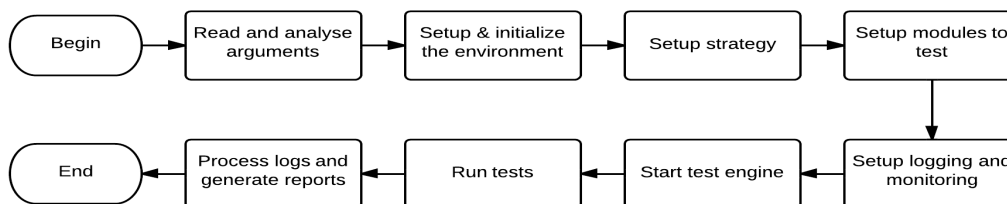
Figure 3.1: Working process of YETI

42

3.2 Design

YETI is a lightweight platform with around 10000 lines of code. It has been designed with the provision of extensibility for future growth. YETI enforces strong decoupling between test strategies and the actual language constructs, which adds new binding, without any modification in the available test strategies. YETI can be divided into three main parts on the basis of functionality: the core infrastructure, the strategy, and the language-specific binding. Each part is briefly described below.

3.2.1 Core Infrastructure of YETI

The core infrastructure of YETI provides extendibility through specialization. The abstract classes included in this section can be extended to create new strategies and language bindings. It is responsible for test data generation, test process management and test report production. The core infrastructure of YETI is split into four packages, i.e. `yeti`, `yeti.environments`, `yeti.monitoring` and `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.
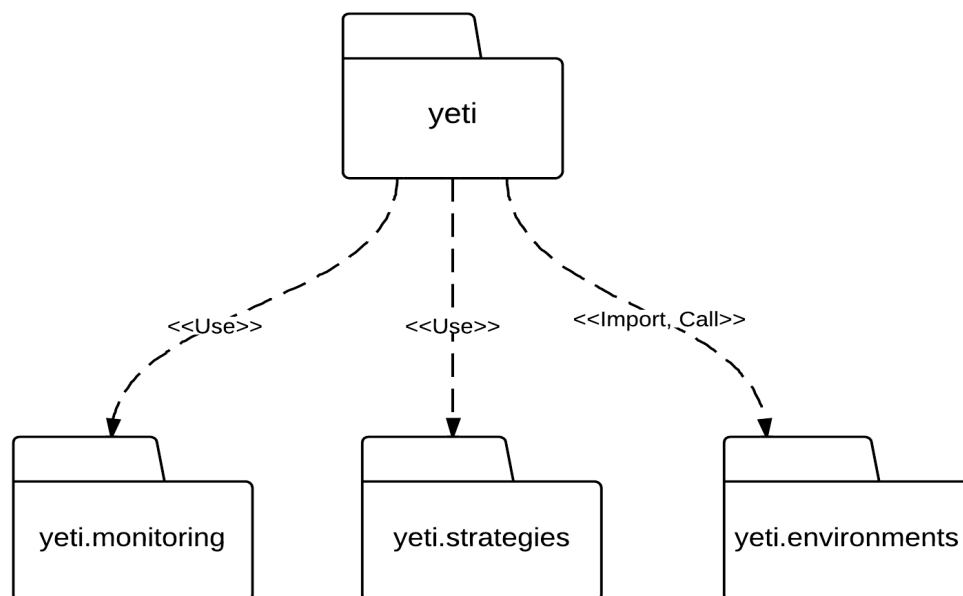
Figure 3.2: Main packages of YETI with dependencies

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

1. **Yeti** is the entry point to the tool 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** prints debugging and testing logs.

3. **YetiLogProcessor** is an interface for processing testing logs.

4. **YetiEngine** binds `YetiStrategy` and `YetiTestManager` together and carries-out the actual testing process.

5. **YetiTestManager** makes the actual calls based on the `YetiEngine` configuration, and activates the `YetiStrategy` to generate test data and select the routines.

6. **YetiProgrammingLanguageProperties** is a place-holder for all language-related instances.

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

3.2.2 Strategy

The strategy defines a specific way to generate test inputs. The strategy section consist of seven essential strategies stated below:

1. **YetiStrategy** is an abstract class which provides an interface for every strategy in YETI.

2. **YetiRandomStrategy** implements the random strategy and generates random values for testing. In this strategy the user can change the null values probability and the percentage of creating new objects for the test session.

3. **YetiRandomPlusStrategy** extends the random strategy by adding values to the list of interesting values. It allows the user to adjust the percentage of using interesting values in the test session.

4. **YetiDSSRStrategy** extends YetiRandomPlusStrategy by adding values surrounding the failure finding value. The strategy is described in detail in Chapter 4.

5. **YetiADFDStrategy** extends YetiRandomPlusStrategy by adding the feature of graphical representation of failures and their domains within the specified lower and upper bounds. The strategy is described in detail in Chapter 5.

6. **YetiADFDPlusStrategy** extends ADFD strategy by adding the feature of graphical representation of failures and failure domains in a given radius in simplified form. The strategy is described in detail in Chapter 6.

7. **YetiRandomDecreasingStrategy** extends the YetiRandomPlusStrategy in which all three probability values (null values, new objects, interesting values) are 100% at the beginning and decrease to 0 at the end of the test.

8. **YetiRandomPeriodicStrategy** extends the YetiRandomPlusStrategy in which all three probability values (null values, new objects, interesting values) decrease and increase at random within the given range.

3.2.3   Language-specific Binding

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

1. **YetiVariable** is a sub-class of `YetiCard` representing a variable in YETI.

2. **YetiType** represents the type of data in YETI, e.g. integer, float, double, long, boolean and char.

3. **YetiRoutine** represents constructor, method and function in YETI.

4. **YetiModule** represents a module in YETI and stores one or more routines of the module.

5. **YetiName** represents a unique name assigned to each instance of `YetiRoutine`.

6. **YetiCard** represents a wild-card or a variable in YETI.

7. **YetiIdentifier** represents an identifier for an instance of a `YetiCard`.

3.2.4   Construction of Test Cases

YETI constructs 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, which are converted to the required type using Java cast operation. In the case of user-defined classes as methods parameters YETI calls constructor or method to generate object of the class at run time. In the case, when the constructor requires another object, YETI recursively calls the constructor or method of that object. This process is continued until an object with an empty constructor or a constructor with only primitive types or the set level of recursion is reached.

3.2.5 Call sequence of YETI

The sequence diagram given in Figure 3.3 depicts the interactions of processes and their order when a Java program in byte-code is tested by YETI for n number of times using the default random strategy. The steps involved are as follows:
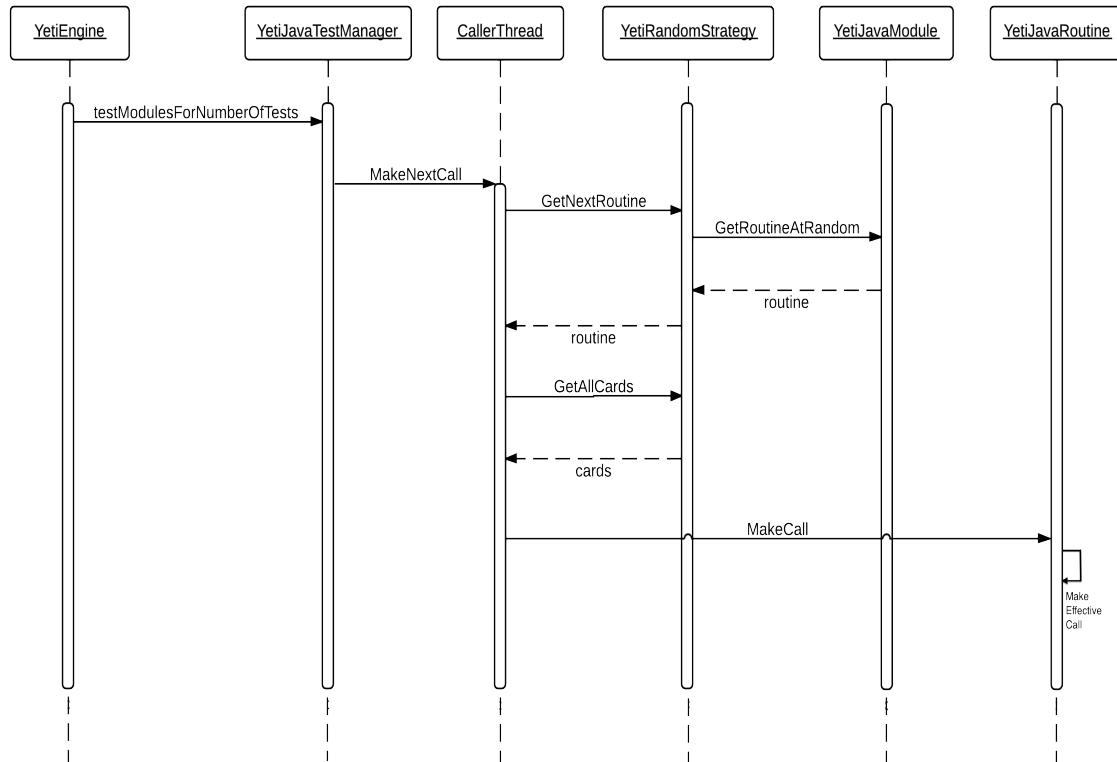
Figure 3.3: Call sequence of YETI with Java binding

1. When the test starts, the test engine (`YetiEngine`) instructs the test manager (`YetiJavaTestManager`) to initiate the testing of given class (`YetiJavaModule`) for n number of times (`testModuleForNumberofTimes`) using the test strategy (`YetiRandomStrategy`).

2. The test manager creates (`makeNextCall`) a thread (`CallerThread`) which handles the testing of a given program. Threading is introduced for two reasons: (1) To enable the test manager to block/terminate a thread that is stuck because of an infinite loop inside the method or taking too long because of recursive calls. (2) To increase the speed of the testing process when multiple classes are under test.

3. The thread on instantiation, requests the test strategy (`YetiRandomStrategy`) to fetch a routine (`constructor/method`) from the given SUT (`getNextRoutine`).

4. The test strategy random strategy selects (`getRoutineAtRandom`) a routine from the class (`YetiJavaModule`).

5. The thread requests the test strategy to generate the arguments (`cards`) for the selected routine (`getAllCards`).

6. The test strategy (`YetiRandomStrategy`) generates the required arguments and sends them to the requesting thread.

7. The thread tests the selected routine (`YetiJavaMethod`) of the module with the generated arguments (`makeCall`).

8. The routine (`YetiJavaRoutine`) is mapped to an instance of the class method (`YetiJavaMethod`) with the help of class inheritance and dynamic binding. The (`YetiJavaMethod`) executes the method under test with the supplied arguments using Java Reflection API (`makeEffectiveCall`).

9. The output obtained from (`makeEffectiveCall`) is evaluated against the Java oracle that resides in the (`makeCall`) method of (`YetiJavaMethod`).

3.2.6   Command-line Options

YETI is originally developed as a command line program, which can be initiated from the Command Line Interface (CLI) as shown in Figure 3.4. During this study a `yeti.jar` package is created which allows the setting of the main parameters and the initiation of YETI from the GUI by double clicking the icon as shown in Figure 3.5. 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 the 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 overheads. Table 3.1 includes some of the common command line options available in YETI.

Table 3.1: YETI command line options

Options	Purpose
-java, -Java	To test Java programs
-jml , -JML	To test JML programs
-dotnet, -DOTNET	To test .NET programs
-ea	To check code assertions
-nTests	To specify number of tests
-time	To specify test time
-initClass	To use user-defined class for initialization
-msCalltimeout	To set a time out for a method call
-testModules	To specify one or more modules to test
-rawlogs	To print real-time test logs
-nologs	To omit real time logs
-yetiPath	To specify the path to the test modules
-gui	To show the test session in the GUI
-help, -h	To print the help about using YETI
-DSSR	To specify Dirt Spot Sweeping Random strategy
-ADFD	To specify ADFD strategy
-ADFDPlus	To specify $ADFD^{+}$ strategy
-noInstanceCap	To remove the cap on no. of specific type instances
-branchCoverage	To measure the branch coverage
-tracesOutputFile	To specify the file to store output traces
-tracesInputFile	To specify the file for reading input traces
-random	To specify the Random test strategy
-printNumberOfCallsPerMethod	To print the number of calls per method
-randomPlus	To specify the Random plus test strategy
-probabilityToUseNullValue	To specify the probability of inserting null values
-randomPlusPeriodic	To specify the Random plus periodic test strategy
-newInstanceInjectionProability	To specify the probability of inserting new objects

3.2.7 Execution

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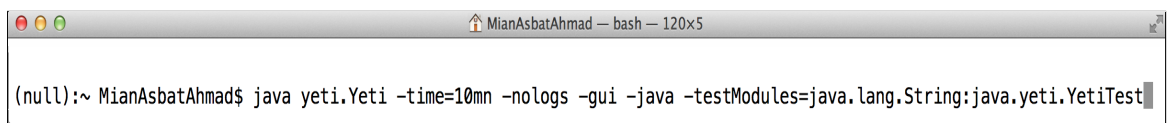
Figure 3.4: Command to launch YETI from CLI

In this command YETI tests `java.lang.String` and `yeti.test.YetiTest` modules for `10` minutes using the default random strategy. Other CLI options are already indicated in Table 3.1. To execute YETI from the GUI, `YetiLauncher` presented in Figure 3.5 has been created for use in the present study.
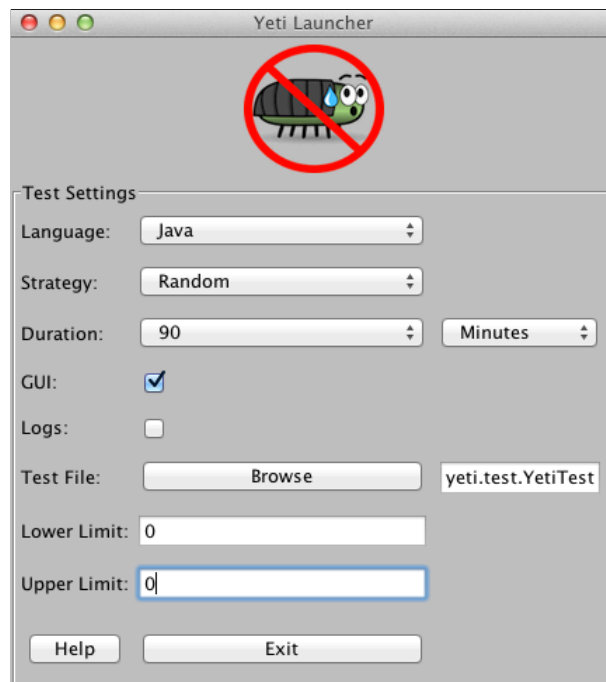
Figure 3.5: GUI launcher of YETI

3.2.8 Test Oracle

Oracles in YETI are language-dependent. 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 violations, which are considered as failures. If specifications or assertions are absent, YETI performs robustness testing, which considers any undeclared runtime exceptions as failures.

3.2.9 Report

YETI gives a complete test report at the end of each test session. The report contains all the successful calls with the name of the routines and the unique identifiers for the parameters in each execution. These identifiers are recorded with the assigned values to help in debugging the identified fault as shown in Figure 3.6.

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

Figure 3.6: Successful method calls of YETI

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

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

Figure 3.7: Sample of YETI bug report

3.2.10 Graphical User Interface

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

1. **Menu bar:** contains two menu items i.e. Yeti and File.

    (a) **Yeti menu:** provides details of YETI contributors and option to quit the GUI.

    (b) **File menu:** provides an option to rerun the previously executed scripts.

2. **Slider of % null values:** displays the set probability of choosing a null value expressed as percentage for each variable. The default value of the probability is 10%.

3. **Slider of % new variables:** displays the set probability of creating new instances at each call. The default value of the probability is 10%.

4. **Text-box of Max variables per type:** displays the number of variables created for a given type. The default value is 1000.

5. **Progress bar of testing session:** displays the test progress as a percentage.

6. **Slider of strategy:** displays the set random strategy for the test session. Each strategy has its control to change its various parameters.

7. **Module Name:** shows the list of modules under test.

8. **Graph window 1:** displays the total number of unique failures over time in the module under test.

9. **Graph window 2:** displays the total number of calls over time to the module under test.

10. **Routine's progress:** 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:** displays the total number of failures over time in the module under test.

12. **Graph window 4:** displays the total number of variables over time generated by YETI in the test session.

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

Figure 3.8: GUI front-end of YETI

3.3 Summary

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

Chapter 4

Dirt Spot Sweeping Random Strategy

There is a strong evidence that Adaptive Random Testing (ART) detects the first failure in a given SUT more efficiently than Random Testing (RT) [24]. This is mainly attributed to the better distribution of test data set which focus on the failure domains within the input domain. On the assumption of the existence of different types of failure domains (see Section 2.9 for details), we developed a new test strategy called Dirt Spot Sweeping Random (DSSR) strategy. It has two main advantages. Firstly, it is based on the Random (R) strategy which makes it highly cost effective. Secondly, it discovers more failures than R and Random$^+$ (R$^+$) strategies in applications under identical conditions, which makes it highly efficient.

In this chapter, we present the DSSR strategy and describe its various parts. We then explain the working mechanism of the new strategy and illustrate its function with the help of an example. In the later part, experimental evidence is presented in support of the effectiveness of the DSSR strategy in finding failures and failure domains as compared with R and R$^+$ strategies.

4.1 Dirt Spot Sweeping Random Strategy

The new DSSR strategy extends the R$^+$ strategy with the feature of dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can produce a high impact on test results. Second, unique failures reside more frequently in contiguous blocks and strip domains. The feature of dirt spot sweeping used in the DSSR strategy increases the performance of finding faults. Before presenting the details of the DSSR strategy, a brief review of the R and R$^+$ strategy is stated below.

4.1.1 Random Strategy

The Random strategy is a test data generation technique in which random data set is generated from the input domain of the software. The randomly generated data set is in accordance with the requirements, specifications or any other test adequacy criteria. The SUT is executed on the test data and the results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. Because of its black-box nature, this strategy is particularly effective in testing software where the developers want to keep the source code secret [110]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational effort [51, 111]. It is mainly for this reason that various researchers have recommended the R strategy for automated testing tools [73]. YETI [104, 112], AutoTest [4, 16], QuickCheck [94], Randoop [86, 113] and Jartege [92] are some of the most common automated testing tools based on the R strategy. See Section 1.2, 2.7.5 and 2.8 for more details on random strategy, generator and testing.

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 [16, 112] called a unique failure. This way of grouping failures is also used for the R^+ and DSSR techniques.

4.1.2 Random$^+$ Strategy

The random$^+$ strategy [16, 51] is an extension of the R strategy. It uses some special pre-defined values, which can be boundary values or values that have a high tendency of finding failures in the SUT. Boundary values [83] 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. See Section 2.10.1 for more details on R$^+$ strategy and testing.

4.1.3 Dirt Spot Sweeping

Chan et al. [1] found that there are domains of failure-causing inputs across the input domain. Section 2.9 shows the failure domains for a two-dimensional input program. They divided these domains into three types called point, block and strip domains. It is argued that a strategy has more chances of hitting the failure domains if test cases far away from each other are selected. Other researchers [71, 84, 114] tried to generate test cases far

away from each other targeting failure domains and achieved better performance, which indicates that failures more often occur contiguously across the input domain. However, if the contiguous failures are unique then the above mentioned strategies will not be able to identify them. Therefore, we proposed that when a test value reveals a failure in a program then the 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 unique failure from the same region.

Dirt spot sweeping is the feature of DSSR strategy that comes into action when a failure is found in the system. On finding a failure, it immediately adds the value causing the failure and its neighbouring values to the existing list of interesting values. For example, in a program when the `int` type value of `50` causes a failure in the system than spot sweeping will add values from `47` to `53` to the list of interesting values. If the failure lies in a block or strip domain, then adding its neighbouring values to the prioritized list will explore other failures present in the block or strip. In the DSSR strategy, the list of interesting values is dynamic and changes during the test execution of each program. While in the R^+ strategy, the list of interesting values remains static and are manually changed before the start of each test, if so required.
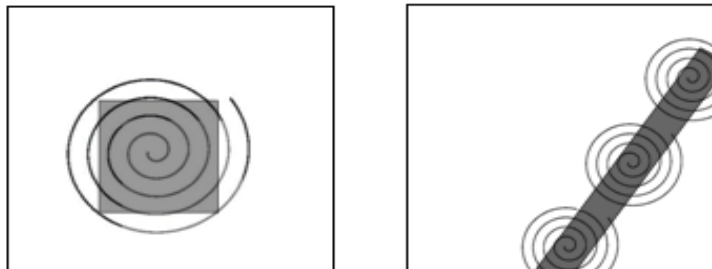
Figure 4.1: Exploration of failures by DSS in block and strip domain

Figure 4.1 shows how DSS explores the failures residing in the block and strip domains of the program. The coverage of block and strip domain is shown in spiral form because first failure leads to second, second to third and so on till the end. In case the failure is positioned on the point domain, then the added values may not be effective because a point domain is only an arbitrary failure point in the whole input domain.

4.1.4 Working of DSSR Strategy

The DSSR strategy continuously tracks the number of failures during the execution of the test. This tracking is done in a very effective way with zero or minimum overhead [115]. The test execution is started by the R$^+$ strategy and continues until a failure is found in the SUT after which the program copies the values leading to the failure as well as the surrounding values to the dynamic list of interesting values.
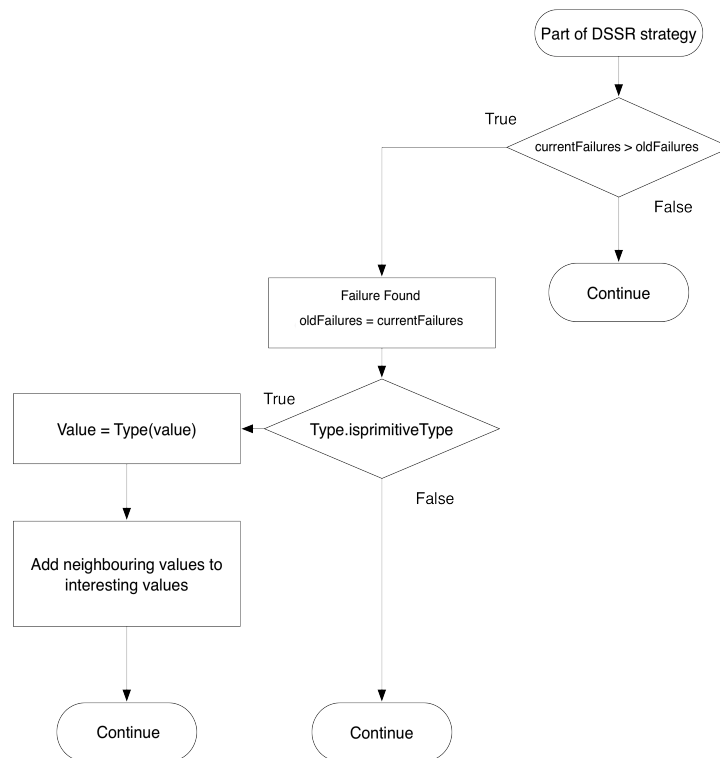
Figure 4.2: Working mechanism of DSSR strategy

The flowchart presented in Figure 4.2 depicts that, when the failure finding value is of a primitive type, the DSSR strategy identifies its type and add values only of that particular type to the list of interesting values. The resultant list of interesting values provides relevant test data for the remaining test session, and the generated test cases are more targeted towards finding new failures around the existing failures in the given SUT.

Boundary and other special values having a high tendency of finding failures in the SUT are added to the list of interesting values by the R$^+$ strategy prior to the start of test session whereas in the DSSR strategy the failure-finding and its surrounding values are added at runtime when a failure is found.

1.	oldFailures ← 0
2.	n ← 5 // This is the radius value to search around the failure set by the

tester.

3.	**if** currentFailures > oldFailures
4.	oldFailures ← currentFailures
5.	**if** type.isPrimitiveType == true
6.	value ← value(type)
7.	value ← value + 3
8.	**for** n ≠ 0
9.	list ← value - n // list is the list of interesting values
10.	n = n - 1
11.	**end for**
12.	**end if**
13.	**end if**

Algorithm of failure spot sweeping due to primitive type in DSSR strategy

When a failure is detected, the strategy analyses the methods argument list to find the values and types which caused a failure. The strategy adds the neighbouring values of that particular type to the list of interesting values. Table 4.1 presents the values added to the list of interesting values when a failure is found. The specific choices of extra interesting values are motivated by the literature [44, 46]. They also correspond to faults intuitive values for exploration about the initial value. In the table the test value is represented by X where X can be a primitive type, string or user-defined objects. All values are converted to their respective types before adding them to the list of interesting values.

Table 4.1: Data types and corresponding values to be added

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

4.1.5 Explanation of DSSR Strategy by Example

The DSSR strategy is explained through a simple program seeded with at least three faults. The first and second fault is called by failing assertions, denoted by (1) and (2) while the third fault is a division by zero error denoted by (3). The program uses only one primitive variable of type `int`. Therefore, the input domain for DSSR strategy is from $-2,147,483,648$ to $2,147,483,647$. The DSSR strategy further selects values (`0`, `Integer.MIN_VALUE` & `Integer.MAX_VALUE`) as interesting values which are prioritised for selection as inputs.

```java
/**
 * Calculate square of given number
 * and verify result.
 * 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;
  assert result1 >= num1; -------------------------(1)
  assert Math.sqrt(result1) == num1; ---------------(2)
  assert num1 == result1 / num1; -------------------(3)
  }
}
```

As the test starts, three failures are quickly discovered by the DSSR strategy in the following order.

Failure 1: The strategy selects 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 the last statement denoted by (3), which will 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 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 another 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 positive value `1` (Java language rule) to variable `result1` that will lead to the failure where assertion (1) fails.
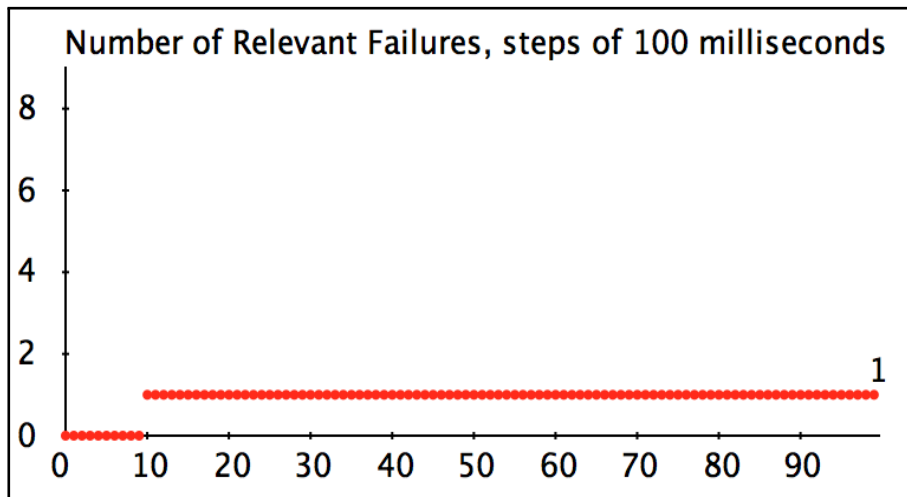
Figure 4.3: Test result of random strategy for the example code
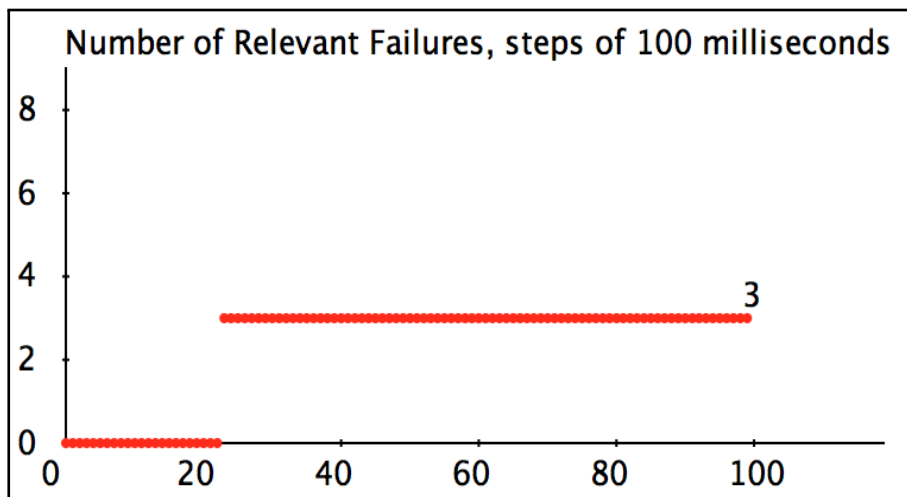
Figure 4.4: Test result of DSSR strategy for the example code

Figures 4.3 and 4.4 present the graphs generated by YETI for the Random and DSSR strategies respectively. The difference is clearly visible i.e. the DSSR strategy immediately detects the second and third failures after the first failure. The Random strategy is not practically capable of detecting these.

The above process explains that including the border, failure-finding and surrounding values to the list of interesting values in the DSSR strategy leads to the available failures quickly and in fewer tests as compared to R and R$^+$ strategies. R and R$^+$ take more 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.

4.2 Implementation of DSSR Strategy

The DSSR strategy is implemented in the YETI open-source automated random testing tool. YETI, coded in Java language, is capable of testing systems developed in procedural, functional and object-oriented languages. YETI can be divided into three decoupled main parts: the core infrastructure, language-specific bindings and strategies. See Chapter 3 for more details on YETI. The strategies part define the procedure of selecting the modules (classes), the routines (methods) and generation of values for instances involved in the routines. It contains all the test strategies for generation of test data. On top of the hierarchy in strategies part is an abstract class *YetiStrategy*, which is extended to form the R strategy (*YetiRandomStrategy*). The R strategy is extended to form the R$^+$ strategy (*YetiRandomPlusStrategy*). Finally, the DSSR strategy (*YetiDSSRStrategy*) is an extension of the R$^+$ strategy. The class hierarchy is shown in Figure 4.5.
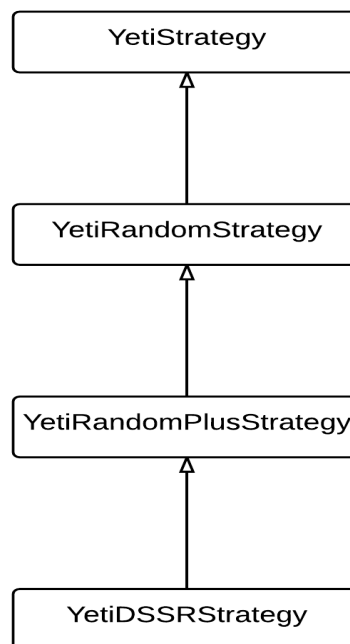
Figure 4.5: Class Hierarchy of DSSR strategy in YETI

4.3 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of the R and R^+ strategies [16, 51]. General factors such as system software and hardware, YETI specific factors like percentage of null values, percentage of newly created objects and the interesting value injection probability have been kept constant in the experiments.

4.3.1 Research Questions

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

1. Is there empirical evidence in support of declaring the most efficient strategy among the 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?

4.3.2 Experiments

We performed extensive testing of programs from the Qualitas Corpus [20]. The Qualitas Corpus is a curated collection of open source Java projects built with the aim of helping empirical research in the field of software engineering. These projects have been collected in an organised form containing the source and binary forms. The Qualitas Corpus [version 20101126] containing 106 open source Java projects was used in the current evaluation. In our experiments, we randomly selected 60 classes from 32 projects taken at random. All the selected classes produced at least one failure and did not time out with maximum testing session of 10 minutes. Every class was tested thirty times by each strategy (R, R^+, DSSR). Name, version and size of the projects to which the classes belong are given in Table 4.2 while test details of the classes are presented in Table 4.3. Lines 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. An approach similar to that used in previous studies when the contracts and assertions in the code under test are absent was followed in the study [2, 112, 116]. The undeclared exceptions were treated as 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

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

of the Java^{TM}SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

4.3.3 Performance Measurement Criteria

Various measures including the E-measure, P-measure and F-measure (see Section 2.10.2 for more details on measurements criteria) have been reported in the literature for finding the effectiveness of the R strategy. The E-measure and P-measure have been criticised [24] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [117, 118]. In our initial experiments, the F-measure was used to evaluate the efficiency of the test strategy. However, it was later realised that this was not the right choice. In some experiments, a strategy found the first failure more quickly than the others but on completion of test session that very strategy found a lower number of total failures than the rival strategy. The preference given to a strategy by F-measure because it finds the first failure quickly without giving due consideration to the total number of failures is not fair [119].

The literature review revealed that the F-measure is used where testing stops after identification of the first failure and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered failures in one go and therefore the F-measure is not a favourable choice. In our experiments, performance of the strategy was measured by the maximum number of failures detected in the SUT by a particular number of test calls [16, 113, 120]. This measurement was effective because it considers the performance of the strategy when all other factors are kept constant.

4.4 Results

Results of the experiments including class name, Lines of Code (LOC), mean value, maximum and minimum number of unique failures, and relative standard deviation for each of the 60 classes tested using the R, R⁺ and DSSR strategies are presented in Table 4.3. Each strategy found an equal number of failures in 31 classes while in the remaining 29 classes the three strategies performed differently from one another. The total of mean values of unique failures was higher in DSSR (1075) as compared to R (1040) and R⁺ (1061) strategies. DSSR found the higher number of maximum unique failures (1118) than R (1075) and R⁺ (1106). DSSR found 43 and 12 more unique failures compared to R and R⁺ strategies respectively. The minimum number of unique failures found by DSSR (1032)

Table 4.3: Comparative performance of R, R^+ and DSSR strategies

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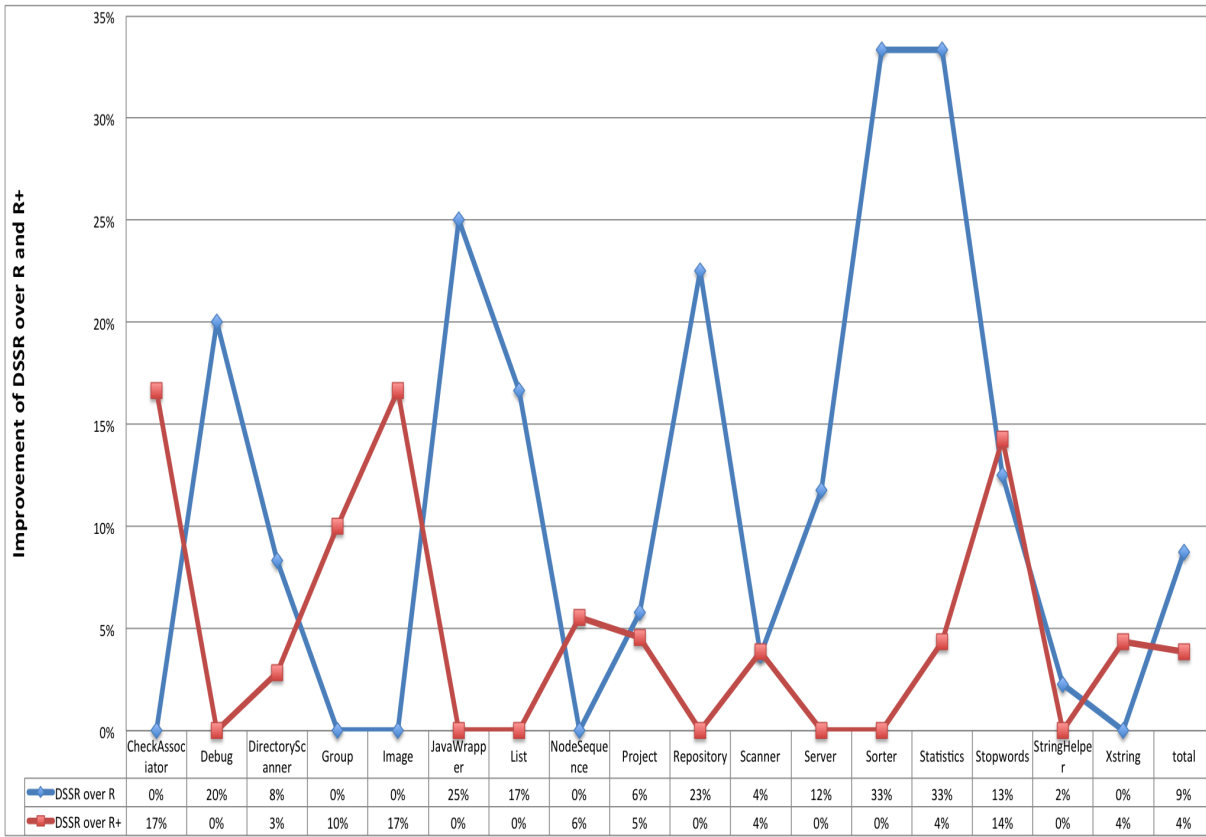
Figure 4.6: Performance of DSSR in comparison with R and R^+ strategies.

is also higher than for R (973) and R^+ (1009) which indicates the higher efficiency of the DSSR strategy over R and R^+ strategies.

4.4.1 Absolute Best in R, R^+ and DSSR Strategies

Based on our findings DSSR is at least as good as R and R^+ in almost all cases. It is significantly better than both R and R^+ in 12% (5/29) of the classes. Figure 4.6 presents the performance of DSSR in comparison with R and R^+ strategies in 16 classes showing significant difference. The blue line with a diamond symbol shows performance of DSSR over R and the red line with square symbols depicts the improvement of DSSR over the R^+ strategy.

The improvement of DSSR over R and R^+ strategy is calculated by applying the formula (4.1) and (4.2) respectively. This is in accordance with the previous work of Chan et al. [121].

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

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

The DSSR strategy performed up to 33% better than R and up to 17% better than the 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 the overall basis DSSR strategy performed better than R and R^+ strategies.

4.4.2 Classes For Which any of the Three Strategies Performs Better

The t-test is used to identify which of the strategies performed significant better because it compares the actual difference between two means in relation to the variation in the data (expressed as the standard deviation of the difference between the means). Result of the t-test analysis, given in Table 4.4, indicated significantly better performance of DSSR in 5 classes from both R and R^+ strategies, in 8 classes from R strategy and in 3 classes from R^+ strategy. In no class R and R^+ strategies performed significantly better than DSSR strategy. 73% (44/60) of the classes showed statistically no significant difference whereas in 27% (16/60) 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 the R^+ strategy.

4.4.3 The Best Default Strategy in R, R^+ and DSSR

Analysis of the experimental data revealed that the DSSR strategy had an edge over R and R^+. This is due to the additional feature of spot sweeping in the DSSR strategy. In spite of the better performance of DSSR as compared to R and R^+ strategies, the present study does not provide ample evidence to pick it as the best default strategy. This is primarily due to the overhead induced by the DSSR strategy, discussed in Section 4.5. Further study might provide some conclusive findings.

4.5 Discussion

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

Table 4.4: Results of t-test applied on experimental data

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**	**0.01**	0.65	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	**0.01**	DSSR = R$^+$ > R
26	Trie	0.1	0.33	0.47	Difference not significant
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

Time taken by the strategies to execute equal number of test cases: The DSSR strategy took slightly more time (up to 5%) than both R and R^+ strategies which might be due to the feature of maintaining sets of interesting values during the execution.

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

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

Effect of identification of first failure on the results: During the experiments, It was noticed that quick identification of first failure was highly desirable in achieving better results from DSSR strategy. This was due to the feature of DSS which added the failure finding and surrounding values to the list of interesting values. However, when identification of first failure was delayed, no values were added to the list of interesting values and the DSSR performed equivalently 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)    {
    abort();        /*  error  */
 }
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion [18]. These literals can be obtained from the constant pool in the class files of the SUT. In the example above, the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the failure right away.

Level of coverage: Random strategies typically achieve a low level of coverage [122], 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 a random process and objective criteria to make it more representative.

4.6 Related Work

Random testing is a popular technique with a simple algorithm but proven to find subtle faults in complex programs and Java libraries [2, 3, 94]. It is simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [72]. Some of the well known automated tools based on R strategy includes JCrasher [2], Eclat [3], AutoTest [16, 73], Jartege [92] and YETI [104, 112].

In the pursuit of better test results and lower overhead, many variations of R strategy have been proposed [84, 85, 114, 110, 123]. ART, Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs randomly but evenly spread across the input domain. ART through dynamic partitioning and MART are the two strategies developed to improve the performance of ART by reducing the overhead. This was achieved mainly by the even spread of test input to increase the chance of exploring the failure domains present in the input domain. A more recent research study [124] 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 compared to 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-heuristic 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 regenerates the data very cheaply in a dynamic fashion without any prior test data and with comparatively lower overhead.

The R^+ strategy is an extension of the R strategy in which interesting values, beside pure random values, are added to the list of test inputs [16, 51]. These interesting values include border values, which have a high tendency of finding failures in the given SUT [83]. Results obtained with the R^+ strategy showed significant improvement over the R strategy [4]. The DSSR strategy is an extension of R^+ that starts testing as R^+ until a failure is found and then switches to dirt spot sweeping.

A common practice to evaluate the performance of an extended strategy is to compare the results obtained by applying a new and existing strategy to identical programs [41, 125, 126]. Arcuri et al. [127], stress the use of random testing as a baseline for comparison with other test strategies. We followed the procedure and evaluated the DSSR strategy against R and R^+ strategies under identical conditions.

In our experiments, we selected projects from the Qualitas Corpus [20] which is a collection of open source Java programs maintained for independent empirical research. The projects in Qualitas Corpus are carefully selected and span the whole set of Java applications [112, 128, 129].

4.7 Summary

The main goal of the present study was to develop a new random strategy which could find more failures in a lower number of test cases. We developed the "DSSR strategy" as an extension of R^+, based on the assumption that in a significant number of classes, failure domains are contiguous. The DSS feature of DSSR strategy adds neighbouring values of the failure finding value to the list of interesting values. The strategy was implemented in the random testing tool YETI to test 60 classes from Qualitas Corpus, 30 times each with each of the three strategies i.e. R, R^+ and DSSR. The newly developed DSSR strategy uncovered more unique failures than both R and R^+ strategies with a 5% overhead. We found out that for 5/29 (8%) classes DSSR was significantly better than both R and R^+, for 8/29 (13%) classes DSSR performed significantly better than R, while in 3/29 (5%) classes DSSR performed significantly better than R^+. In all other cases, performance of DSSR, R and R^+ showed no significant difference. Overall, DSSR produced encouraging results.
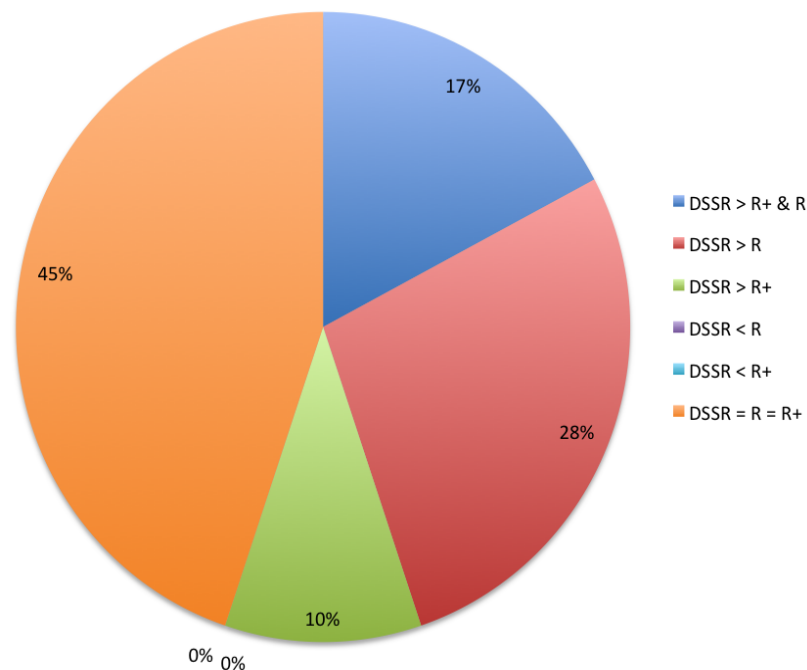
Figure 4.7: Results of DSSR strategy in comparison with R and R^+

Chapter 5

Automated Discovery of Failure Domain

5.1 Introduction

Most of the modern black-box testing techniques execute the Software Under Test (SUT) with specific input and results obtained are compared against the test oracle. A report is generated at the end of each test session depicting any discovered faults and the input values which triggers the failures. Developers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality already set in the test plan is achieved or the provided resources are exhausted [130].

The Adaptive Random Testing (ART) [24], Restricted Random Testing (RRT) [85], Mirror Adaptive Random Testing (MART) [84], Adaptive random testing for object oriented software (ARTOO) [73], Directed Automated Random Testing (DART) [18], Lattice-based Adaptive Random Testing (LART) [131] and Feedback-directed Random Testing (FDRT) [86, 113] are a few of the improved versions of random testing based on the existence of contiguous failure domains within the input domain (see Section 2.9 for more details on failure domains). All these techniques try to detect a single instance of failure ignoring the underlying failure domain. To the best of our knowledge, no specific strategy has been developed to evaluate the failure domains. This chapter describes a new test strategy called Automated Discovery of Failure Domain (ADFD), which not only finds the failure and failure domains but also presents the pass and fail domains graphically. Such graphic representations may provide insight into the nature of failures in the SUT and enable developers to make more effective diagnoses of causes, leading to more appropriate fixes. Some important aspects of ADFD strategy presented in the chapter include:

- Description and implementation of the new ADFD strategy in YETI.

- Evaluation to assess ADFD strategy by testing classes with different failure domains.

- Reduction in test duration by identification of all failure domains instead of a single instance of failure.

- Improvement in test efficiency by helping debugger to consider all failure occurrences during debugging.

5.2 Automated Discovery of Failure Domain

The Automated Discovery of Failure Domain (ADFD) strategy is proposed as the improvement on R^+ strategy with capability of finding failure domains as well as the failures. The output produced at the end of test session is a chart showing the passing value or range of values in blue and failing value or range of values in red. The complete work-flow of ADFD strategy is given in Figure 5.1.

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

1. GUI front-end for providing input

2. Automated finding of failure

3. Automated generation of modules

4. Automated compilation and execution of modules to discover domains

5. Automated generation of graph showing domains

5.2.1 GUI Front-end for Providing Input

The 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, options to enable or disable YETI GUI, logs and program in byte code. In addition, it also takes minimum and maximum values to search for failures in the specified range. The default range for minimum and maximum is taken as Integer.MIN_VALUE and Integer.MAX_VALUE respectively. The GUI front-end of ADFD technique is given in Figure 5.2.

73
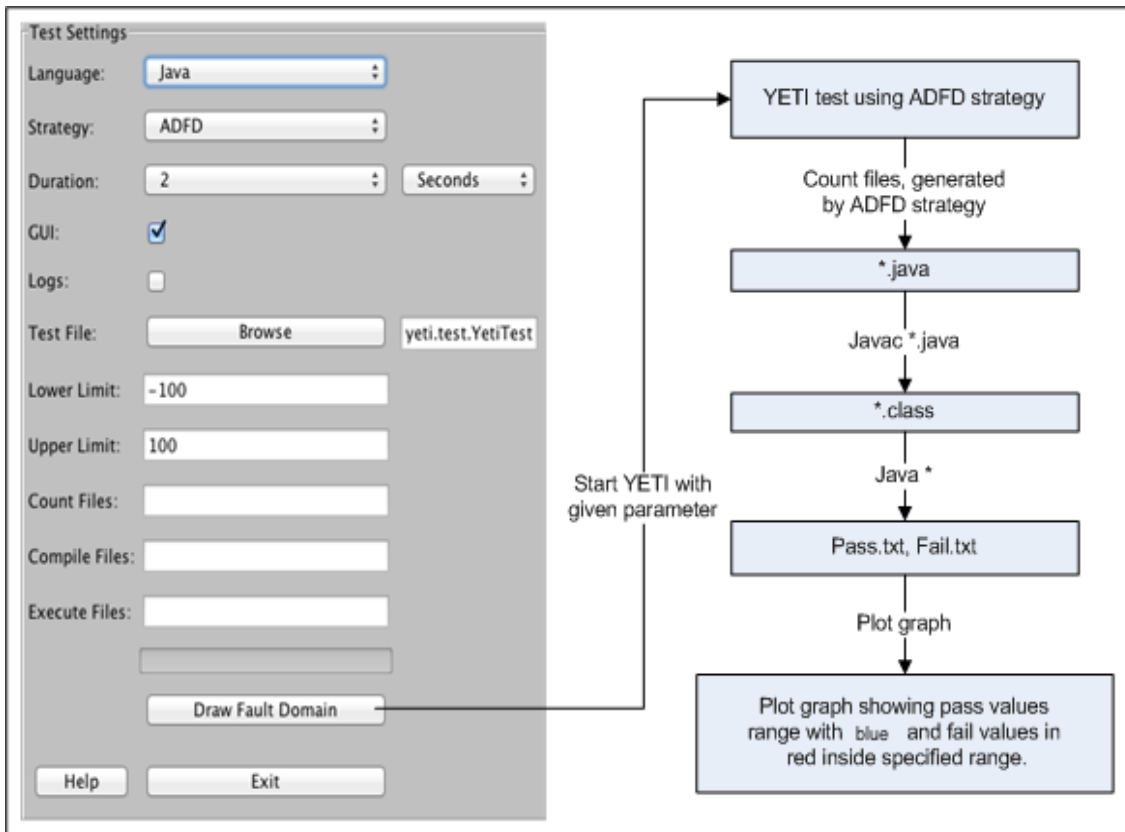
Figure 5.1: Work-flow of ADFD strategy

5.2.2 Automated Finding of Failure

ADFD, being extended form of R^+ strategy, relies on the R^+ strategy to find the first failure. The Random$^+$ strategy is an improvement on random strategy with preference for the boundary values and other special pre-defined values to provide better failure finding ability.

5.2.3 Automated Generation of Modules

After a failure is found in the SUT, ADFD strategy generates a complete new Java program to search for failure domains in the given SUT. These programs with ".java" extensions are generated through dynamic compiler API included in Java 6 under javax.tools package. The number of programs generated can be one or more, depending on the number of arguments in the test module i.e. for a module with one argument one program is generated, for a module with two arguments two programs, and so on. To track the failure domain, the program keeps only one argument as variable and the remaining arguments as constant in the program generated at run time.
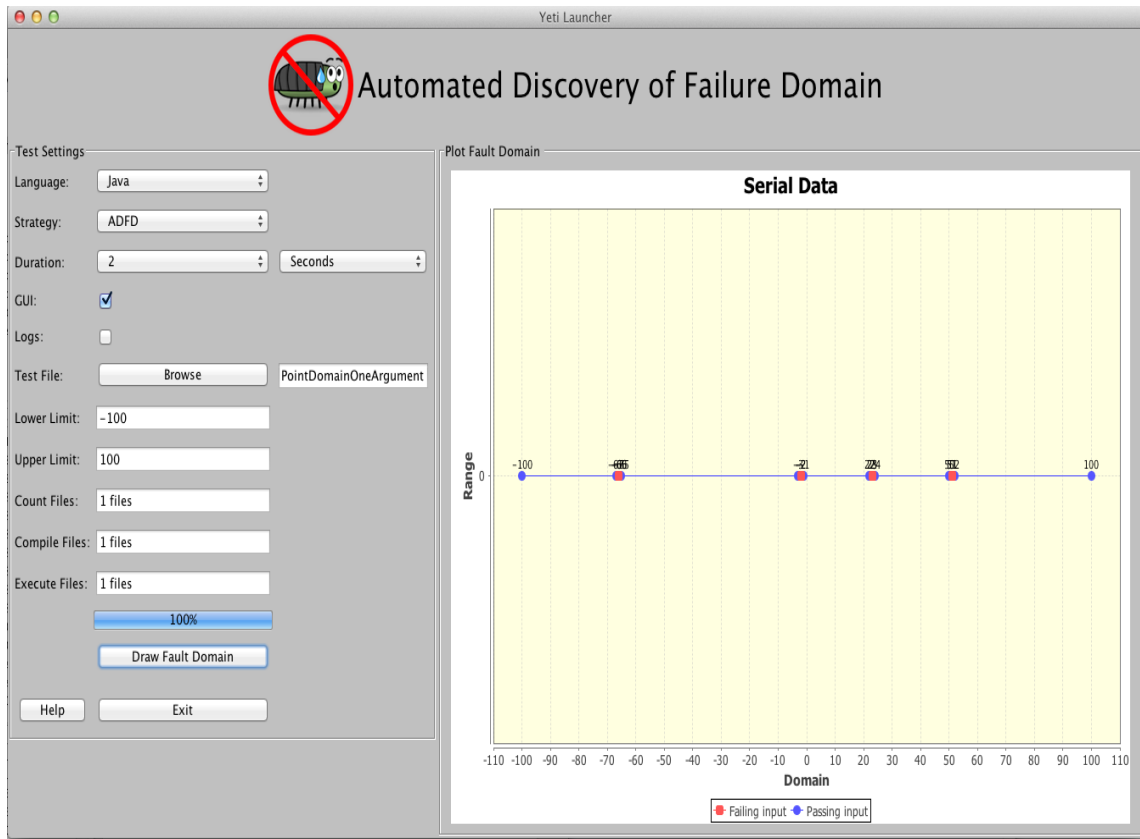
Figure 5.2: Front-end of ADFD strategy

5.2.4 Automated Compilation and Execution of Modules

The java modules generated in the 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 at the beginning of the test. After execution is completed we get two text files of $Pass.txt$ and $Fail.txt$. The pass file contains all the values for which the modules behave correctly while the fail file contains all the values for which the modules fail.

5.2.5 Automated Generation of Graph

The values from the pass and fail files are used to plot (x, y) chart using JFreeChart [132]. JFreeChart is a free open-source Java library that helps developers to display complex charts and graphs. Among several types of available chart, Line chart is selected because

it can represent the available data in the most effective way. The Lines and circles with blue colour represent pass values while lines and squares with red colour represents fail values. The 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.

5.2.6   Implementation of ADFD Strategy

The ADFD strategy is implemented as a strategy of YETI (see Chapter 3 for more details on 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 section, is an abstract class YetiStrategy, which is extended by YetiRandomStrategy and it is further extended to get YetiRandomPlusStrategy. YetiADFDStrategy is developed by extending the YetiRandomPlusStrategy.

```
                    ┌─────────────────────┐
                    │    YetiStrategy     │
                    └─────────────────────┘
                               △
                               │
                    ┌─────────────────────┐
                    │ YetiRandomStrategy  │
                    └─────────────────────┘
                               △
                               │
                    ┌─────────────────────┐
                    │YetiRandomPlusStrategy│
                    └─────────────────────┘
                               △
                               │
                    ┌─────────────────────┐
                    │  YetiADFDStrategy   │
                    └─────────────────────┘
```
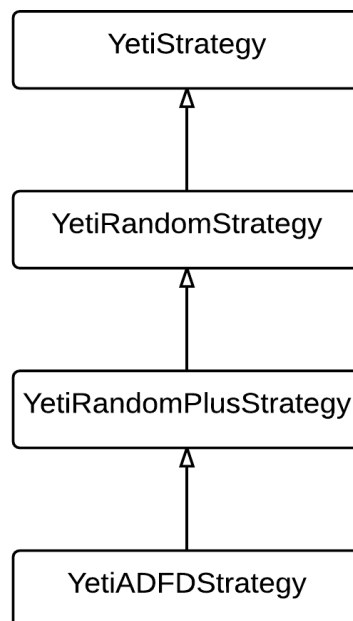
Figure 5.3: Class Hierarchy of ADFD strategy in YETI

5.2.7 Explanation of ADFD Strategy by 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 set the values of lower and upper range by -70 and 70 from Integer.MIN_VALUE and Integer.MAX_VALUE respectively.

```java
/**
 * Point Fault Domain example for one argument program
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{
    public static void pointErrors (int x){
        if (x == -66)
                abort();        /* error */
        if (x == -2)
                abort();        /* error */
        if (x == 51)
                abort();        /* error */
        if (x == 23)
                abort();        /* error */
    }
}
```
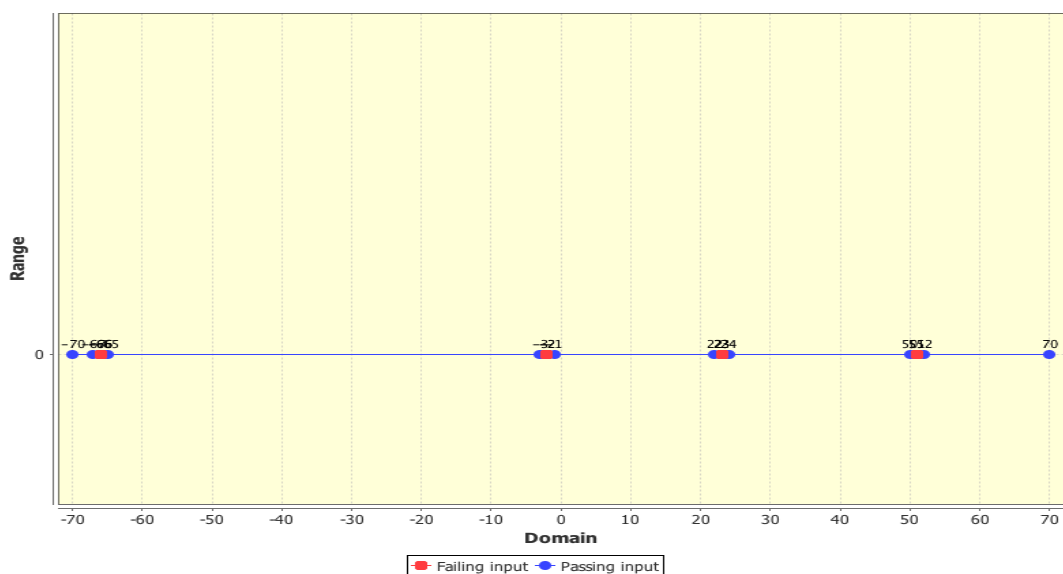

Figure 5.4: ADFD strategy plotting pass and fail domain of a given class

As soon as any one of the above four failures are discovered the ADFD strategy generates a dynamic program given in Appendix 1 (7). This program is automatically compiled to get the binary file and then executed to find the pass and fail domains inside the specified range. The identified domains are plotted on a two-dimensional graph. It is evident from the output presented in Figure 5.4 that ADFD strategy not only finds all the failures but also plots the pass and fail domains.

5.3 Experimental Results

This section includes the experimental set-up and results obtained by using ADFD strategy. Six numerical programs with one or two input parameters were selected. These programs were error-seeded in such a way to get all the three forms of point, block and strip failure domains. Each selected program contained various combinations of one or more failure 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 programs for one and two-dimensional point, block and strip failure domains. The code of the selected program is given in Appendix 1 (1-6). 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 -26, 51 to 54,
		Two	BFDTwoA(-2, i)	(-2, 100) to (-2, 20), (-2, -1) to (-2, -100)	(-2 , 19) to (-2, 0),
			BFDTwoA(i, 0)	Nil	(-100, 0) to (100, 0)
3	Strip	One	SFDOneA(i)	-100 to -5, 35 to 100	-4 to 34
		Two	SFDTwoA(-5, i)	(-5, 100) to (-5, 40), (-5, -1) to (-5, -100)	(-5, 39) to (-5, 0),
			SFDTwoA(i, 0)	Nil	(-100, 0) to (100, 0)

78

Point Failure Domain: Two separate Java programs `Program1` and `Program2`, given in Appendix 1 (1, 2), were tested with the ADFD strategy in YETI to get the findings for point failure domains in one and two-dimensional programs. Figure 5.5(a) presents range of pass and fail values for the point failure domain in one-dimension whereas Figure 5.5(b) presents range of pass and fail values for the point failure domain in a two-dimensional program. The range of pass and fail values for each program in point failure domains is given in Table 5.1.
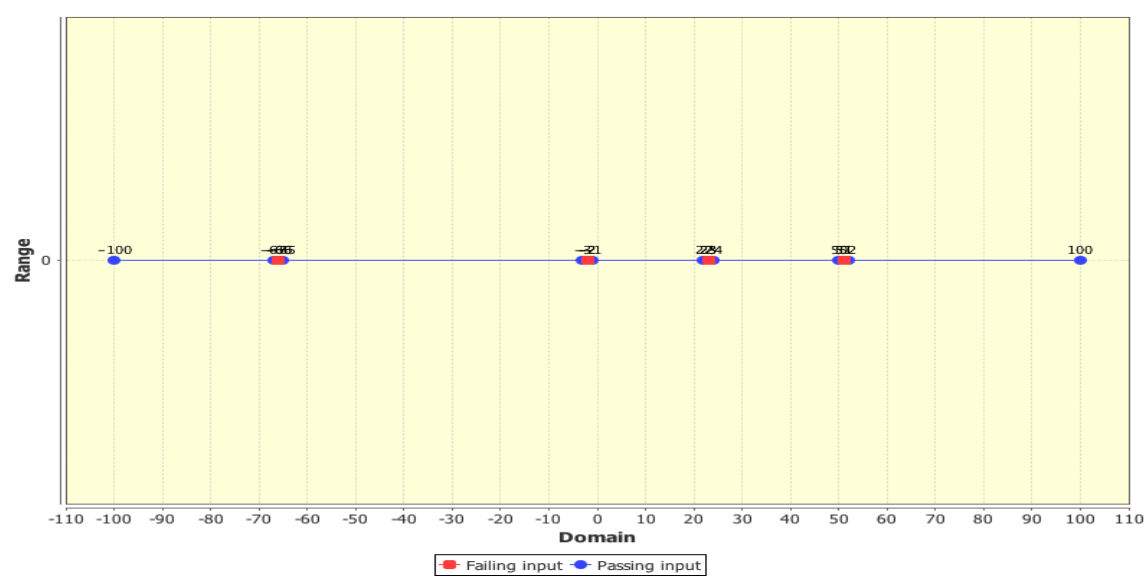
(a) One dimension module
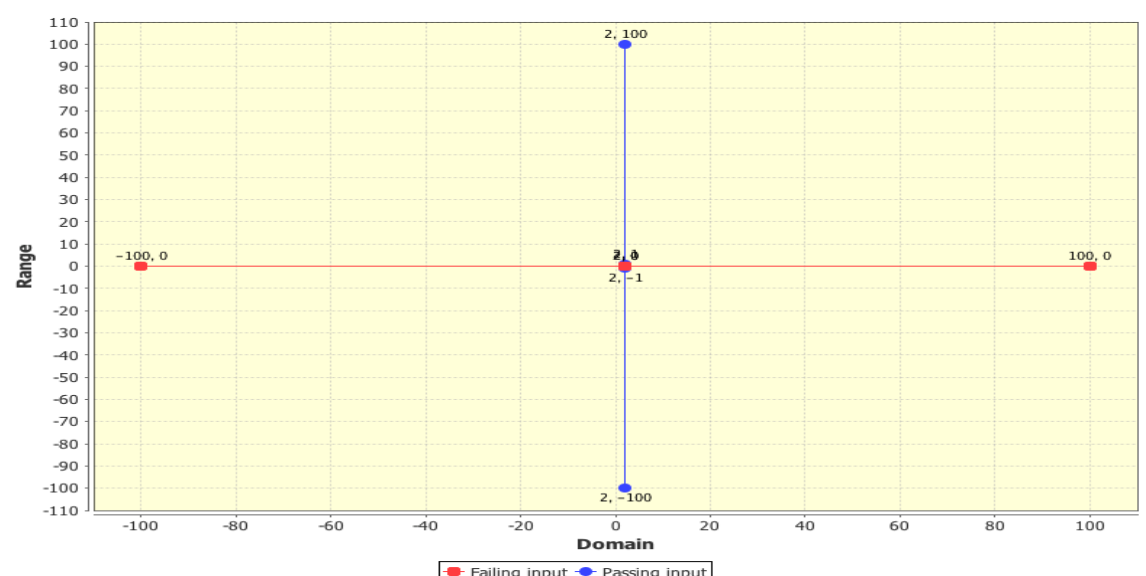
(b) Two dimension module

Figure 5.5: Chart generated by ADFD strategy presenting point failure domains

79

Block Failure Domain: Two separate Java programs `Program3` and `Program4` given in Appendix 1 (3, 4) were tested with the ADFD strategy in YETI to get the findings for block failure domains in one and two-dimension program. Figure 5.6(a) presents a range of pass and fail values for block failure domains in one-dimension whereas Figure 5.6(b) presents a range of pass and fail values for block failure domains in a two-dimensional program. The range of pass and fail values for each program in block failure domains is given in Table 5.1.

(a) One dimension module

(b) Two dimension module
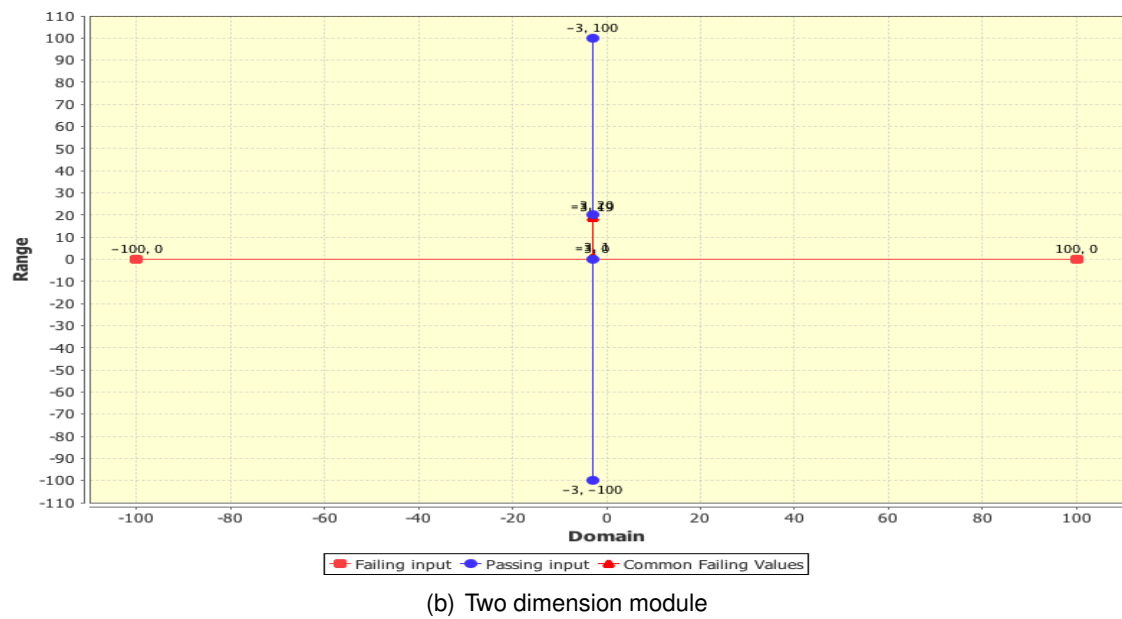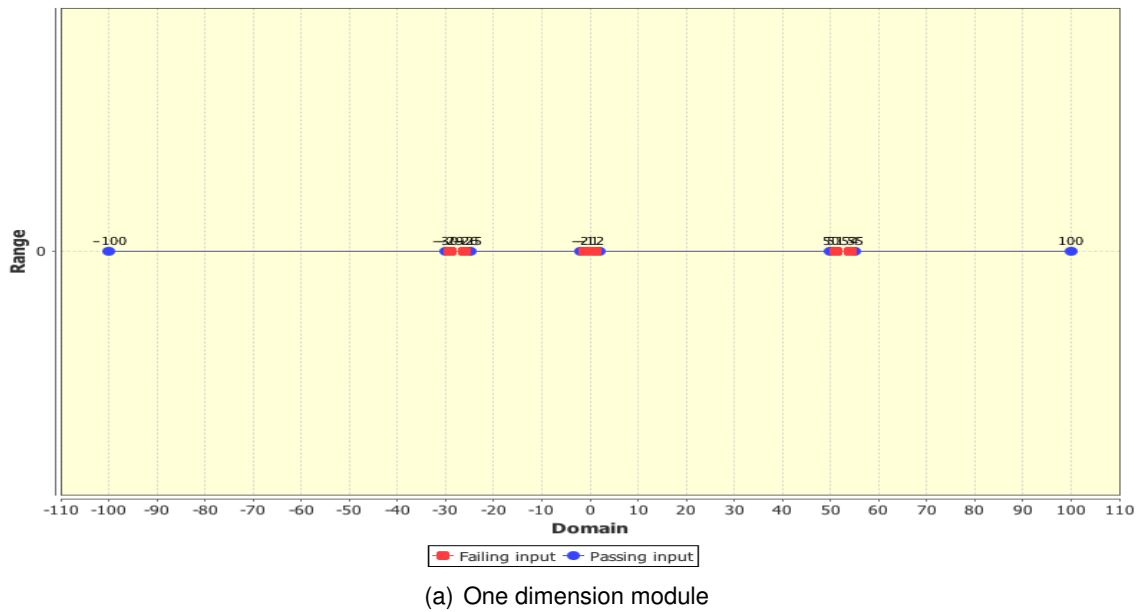
Figure 5.6: Chart generated by ADFD strategy presenting block failure domain

Strip Failure Domain: Two separate Java programs `Program5` and `Program6` given in Appendix 1 (5, 6) were tested with the ADFD strategy in YETI to get the findings for strip failure domains in one and two-dimension programs. Figure 5.7(a) presents range of pass and fail values for strip failure domain in one-dimension whereas Figure 5.7(b) presents a range of pass and fail values for strip failure domain in two-dimension program. The range of pass and fail values for each program in strip failure domain is given in Table 5.1.
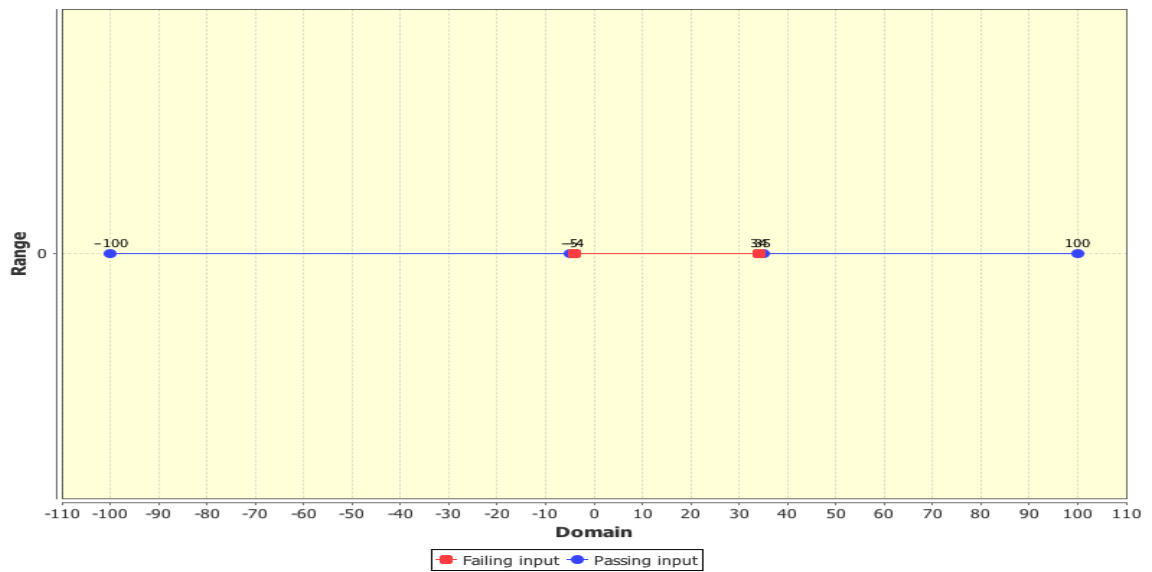
(a) One dimension module
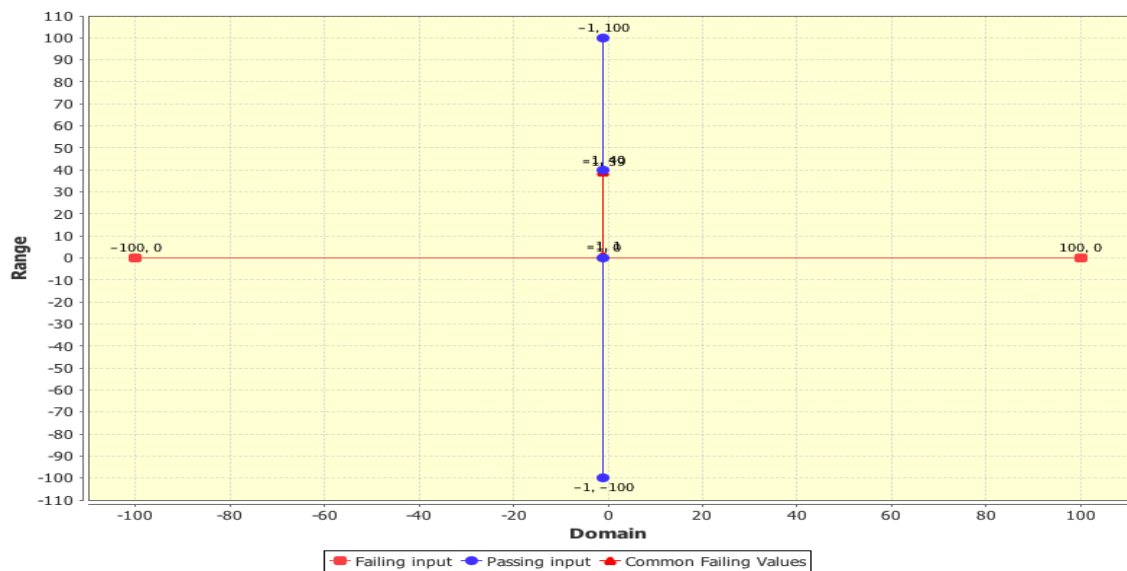
(b) Two dimension module

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

5.4 Discussion

The ADFD, with a simple graphical user interface, is a fully automated testing strategy which identifies failures, failure domains and visually presents the pass and fail domains in the form of a chart. Since all the default settings are set to the optimum level, the user needs only to specify the module to be tested and click the "Draw fault domain" button to start test execution. All the steps including identification of fault, generation of dynamic Java program to find the domain of the identified failure, saving the program to a permanent media, compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automatically without any human intervention.

As evident from the results, the ADFD strategy effectively identified failures and failure domains in selected programs. Identification of the failure domain is simple for one and two-dimensional numerical programs 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.

The ADFD strategy initiates testing with R^+ to find the failure and later switches to an exhaustive strategy to apply all the values between upper and lower bounds for finding pass and failure domains. It was found that failures at boundary of the input domain usually pass unnoticed through random test strategy [46] but not through the 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 the maximum range (i.e. minimum for int is Integer.MIN_VALUE and maximum Integer.MAX_VALUE) then the test duration is also maximum. It is rightly so because for identification of the failure domain the program is executed for every input available in the specified range. Similarly increasing the range also shrinks the produced graph making it difficult to identify clearly point, block and strip domains unless they are of considerable size. Test duration is also influenced by identification of the first failure and the complexity of the module under test.

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

5.5 Threats to Validity

The major external threat to the use of ADFD strategy on a commercial scale is the selection of a small set of error-seeded programs of only primitive types such as integer used in the experiments. However, the present study will serve as the 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 [88], to facilitate easy plotting. Plotting composite objects is beyond the scope of the present study. However, further studies are required to look into the matter in depth.

Another threat to validity includes evaluating programs 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_VALUE to Integer.MAX_VALUE) is difficult to adjust and does not give a clear view on the chart. To solve this problem, a zoom feature was incorporated into the GUI to magnify the areas of interest on the chart.

5.6 Related Work

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, 122]. To overcome the performance issues without compromising on its benefits, various researchers have altered its algorithm as explained in Section 2.10. Most of the alterations are based on the existence of faults and failure domains across the input domain [1].

The ADFD is an automated technique which can also be used with other techniques, e.g. structural coverage approaches, because it is mainly concerned with traversing fault domains and does not depend on how a starting fault within a fault domain is found.

Identification, classification of pass and fail domains and visualisation of domains have not received due attention by researchers. Podgurski et al. [133] proposed a semi-automated procedure to classify faults and plot them by using a Hierarchical Multi Dimension Scaling

(HMDS) algorithm. A tool named Xslice [134] visually differentiates the execution slices of passing and failing parts of the test. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [135]. 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 availability of existing test cases whereas the ADFD strategy generates new test cases, discovers faults, identifies pass and fail domains and visualises them in graphical form in a fully automated manner.

5.7 Summary

The newly developed ADFD technique identifies failure, failure domains and graphically represents the test results using XY line chart. Experimental results obtained by applying the ADFD strategy to error-seeded numerical programs provide evidence that the strategy is highly effective in identifying the failures and plotting pass and fail domains of a given SUT. The ADFD strategy can find boundary failures quickly compared with traditional random testing, which is either unable or takes a longer time to discover the failures.

Experimental results reveal that the use of ADFD strategy is highly effective in identifying and presenting failures and failure domains. 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. 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-dimensions and different non-scalar argument types.

Chapter 6

Automated Discovery of Failure Domain$^+$

6.1   Introduction

Software testing is most widely used for verification and validation. Efforts have been continuously made by researchers to make the testing process more effective and efficient. Testing is efficient when maximum number of test cases are executed in minimum possible time and it is effective when it finds the maximum number of faults in minimum number of test cases [136]. During up-grading and development of testing techniques, the focus is always on increasing the efficiency by introducing partial or complete automation of the testing process and the effectiveness by improving the algorithm.

To target failures and evaluate the failure domains we developed earlier ADFD technique [108]. The ADFD$^+$, an improved version of ADFD, is a fully automatic technique which finds failures and failure domains within a specified radius and presents the results on a graphical chart [109]. The efficiency and effectiveness of the ADFD$^+$ technique is evaluated by comparing its performance with that of a mature random testing tool for object-oriented programs (Randoop) [86]. The results generated by ADFD$^+$ and Randoop for the error-seeded programs shows better performance of ADFD$^+$ with respect to time and number of test cases to find failure domains. Additionally ADFD$^+$ presents the results graphically showing identified point, block and strip domains visually as against Randoop, which lacks a graphical user interface.

6.2   Automated Discovery of Failure Domain$^+$

ADFD$^+$ is an improved version of ADFD, a technique developed earlier by Ahmad and Oriol [108]. The technique automatically finds failures, failure domains and present the results in graphical form. In this technique, the test execution is initiated by random$^+$ and

continues till the first failure is found in the SUT. The technique then copies the values leading to the failure and the surrounding values to the dynamic list of interesting values. The resultant list provides relevant test data for the remaining test session and the generated test cases are effectively targeted towards finding new failures around the existing failures in the given SUT.

The improvements made in ADFD$^+$ over ADFD technique are stated as follows.

- ADFD$^+$ generates a single Java file dynamically at run time to plot the failure domains as compared to one Java file per failure in ADFD. This saves time and makes the execution process quicker.

- ADFD$^+$ uses (x, y) vector-series to represent failure domains as opposed to the (x, y) line-series in ADFD. The vector-series allows more flexibility and clarity to represent failure and failure domains.

- ADFD$^+$ takes a single value for the radius within which the strategy searches for a failure domain whereas ADFD takes two values as lower and upper bounds representing x and y-axis respectively. This results in consumption of fewer test cases for detecting failure domain.

- In ADFD$^+$, the algorithm of dynamically generating a Java file at run-time is simpler and more efficient compared to ADFD, resulting in reduced overhead.

- In ADFD$^+$, the point, block and strip failure domains generated in the output graph present a clear view of pass and fail domains with individually labelled points of failures as against a less clear view of the pass and fail domains and lack of individually labelled points in ADFD.

6.2.1 Implementation of ADFD$^+$

The ADFD$^+$ technique is also implemented in the automated random testing tool YETI. As stated earlier YETI consists of three main parts including core infrastructure for extendibility, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both strategies and languages sections have pluggable architecture for easily incorporating new strategies and languages. At the moment, there are seven different random strategies including our previously developed DSSR and ADFD strategies. The ADFD$^+$ strategy is also added to the strategies section of YETI by extending the $YetiADFDStrategy$. Please see Chapter 3 and Chapter 5 for more details about YETI and ADFD respectively.

6.2.2 Workflow of ADFD$^+$

ADFD$^+$ is a fully automatic technique requiring the user to select radius value (Domain Range) and feed the program under test followed by clicking the "Draw Fault Domain" button for test execution. The work-flow of ADFD$^+$ is given in the following pseudocode. The default values are set to optimum, however the user can change it according to the requirements. As soon as the "Draw Fault Domain" button is clicked, YETI comes into play with the ADFD$^+$ strategy to search for failures in the program under test. If a failure is found, the strategy creates a Java file which contains calls to the program on the failing and surrounding values within the specified range. The Java file is executed after compilation and the results obtained are analysed to separate pass and fail values, which are accordingly stored in the text files. The integrated tool Daikon also observes the execution and generates the likely invariants. The generated invariants are later displayed in the GUI. All the values are plotted on a graph with pass values in blue and fail values in red as shown in Figure 6.1. Presenting the identified failure domains by two means (graphical and generated invariants) provides clearer and verifiable results.

01.   Initiate the GUI of ADFD$^+$
02.   Specify YETI specific parameters
03.   Specify the value of range
04.   Specify method for testing
05.   Click the Draw Fault Domain button to start test
06.   YETI executes the method to find failure
07.   On finding failure, a file C.java is dynamically created
08.   The file is compiled using javac C.java to get C.class file
09.   The C.class file is executed on inputs specified in range
10.   Daikon observes execution and generates invariants
11.   The execution also write values to Pass.txt and Fail.txt
12.   The values are passed to JFreeChart
13.   The JFreeChart evaluates the values to draw the graphs
14.   Invariants and Graph are displayed on the GUI.
15.   The process finishes

Workflow of the ADFD$^+$ process from start to finish

6.2.3 Example to Illustrate Working of ADFD$^+$

Suppose we have the following error-seeded class under test. It is evident from the program code that a failure is generated when the value of variable x ranges between 5 to 8 and the value of variable y between 2 to 4.

```java
public class Error {
  public static void errorProgram (int x, int y){
    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
        abort();          /* error */
  }
}
```

At the beginning of the test, the ADFD$^+$ strategy evaluates the given class with the help of YETI and finds the first failure at x = 6 and y = 3. Once a failure is identified ADFD$^+$ uses the surrounding values around it to find a failure domain. The radius of surrounding values is limited to the value set by the user in the *Domain Range* variable. When the value of *Domain Range* is set to 5, ADFD$^+$ evaluates a total of 83 values of x and y around the found failure. All evaluated (x,y) values are plotted on a two-dimensional graph with red filled squares indicating fail values and blue filled circles indicating pass values. Figure 6.1 shows that the failure domain forms a block pattern and the boundaries of the failures are $(5,2),(5,3),(5,4),(6,2),(6,4),(7,2),(7,4),(8,2),(8,3),(8,4)$.
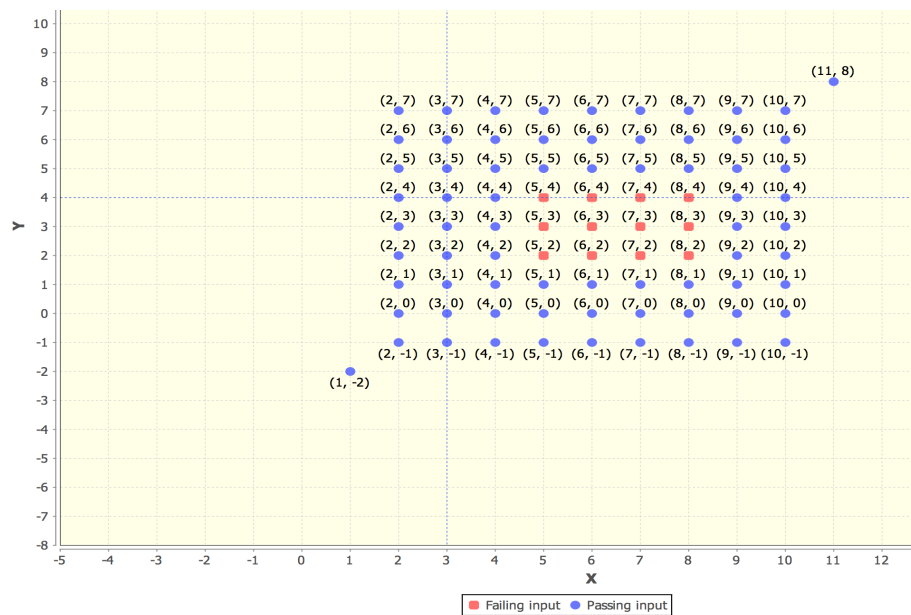
Figure 6.1: The output of ADFD$^+$ for the above code

6.3 Evaluation

For evaluating the efficiency and effectiveness, we compared ADFD$^+$ with Randoop, following the common practice of comparison of the new tool with a mature random testing tool [3, 92, 137]. Testing of several error-seeded one and two dimensional numerical programs was carried out. The programs were divided in to set A and B containing one and two-dimensional programs respectively. Each program was injected with at least one failure domain of point, block or strip nature. The failure causing values are given in Table 6.1. Every program was tested independently for 30 times by both ADFD$^+$ and Randoop. Time taken and the number of tests executed to find all failure domains were used as criteria for efficiency and effectiveness respectively. The external parameters were kept constant in each test. Due to the absence of contracts and assertions in the code under test, undeclared exceptions were taken as failures in accordance with the previous studies [2, 112, 116].

Table 6.1: Table depicting values of x and y arguments forming point, block and strip failure domain in Figure 6.6(a), 6.6(b), 6.6(c) and Figure 6.7(a), 6.7(b), 6.7(c) respectively

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

6.3.1 Research Questions

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

1. How efficient is $ADFD^+$ as compared to Randoop?

2. How effective is $ADFD^+$ as compared to Randoop?

3. How failure domains are presented by $ADFD^+$ as compared to Randoop?

6.3.2 Randoop

The random tester for object-oriented programs (Randoop) is a fully automatic tool, capable of testing Java classes and .NET binaries. It takes as input a set of classes, time limit or number of tests and optionally a set of configuration files to assist testing. Randoop checks for assertion violations, access violations and unexpected program termination in a given class. Its output is a suite of JUnit programs for Java and NUnit programs for .NET program. Each unit test in a test suite is a sequence of method calls (hereafter referred as sequences). Randoop builds the sequence incrementally by randomly selecting public methods from the class under test. Arguments for these methods are selected from the pre-defined pool in case of primitive types and as a sequence of null values in case of reference types. Randoop uses the feedback mechanism to filter out duplicate test cases. For more details about Randoop, please see Section 2.11.4.

6.3.3 Experimental Setup

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

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

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


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

6.4 Experimental Results
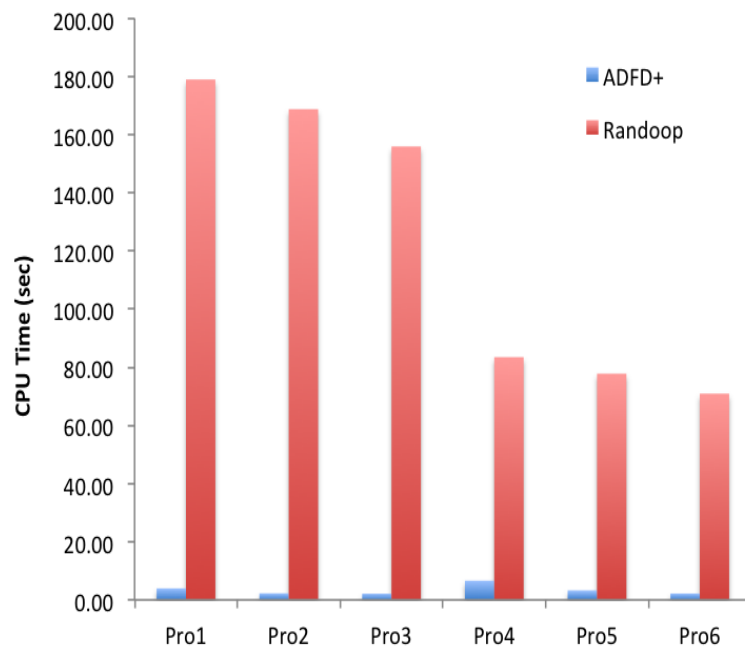
Figure 6.2: Time taken to find failure domains
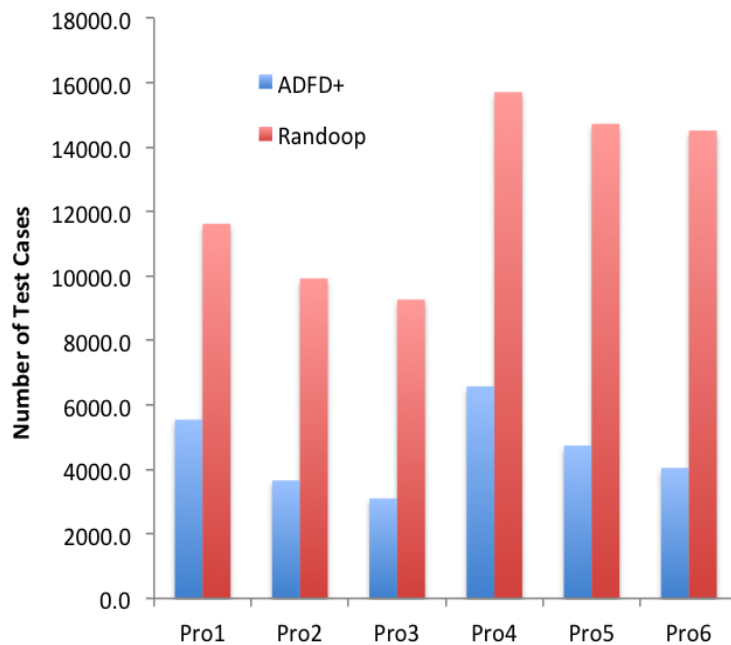
Figure 6.3: Number of test cases taken to find failure domains

6.4.1 Efficiency

Figure 6.4 shows the comparative efficiency of ADFD$^+$ and Randoop. The $x-axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y-axis$ represents the average time taken by the tools to detect the failure domains. As shown in the figure, ADFD$^+$ showed extraordinary efficiency by taking two orders of magnitude less time to discover failure domains as compared to Randoop.
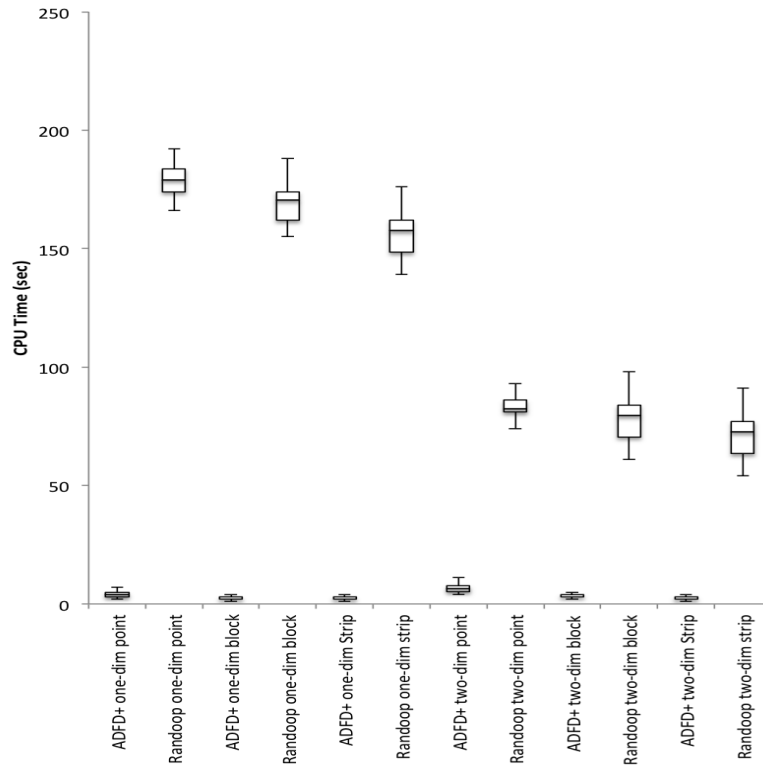
Figure 6.4: Time taken to find failure domains

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

6.4.2 Effectiveness

Figure 6.5 shows the comparative effectiveness of ADFD$^+$ and Randoop. The $x-axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y-axis$ represents average number of test cases used by the tools to detect the failure domains. The figure shows higher effectiveness in the case of ADFD$^+$, amounting to 100% or more. The higher effectiveness of ADFD$^+$ may be attributed to its working mechanism in comparison with Randoop for identifying failures. ADFD$^+$ dynamically changes its algorithm to exhaustive testing in a specified radius around the failure as against Randoop, which uses the same random algorithm for searching failures.
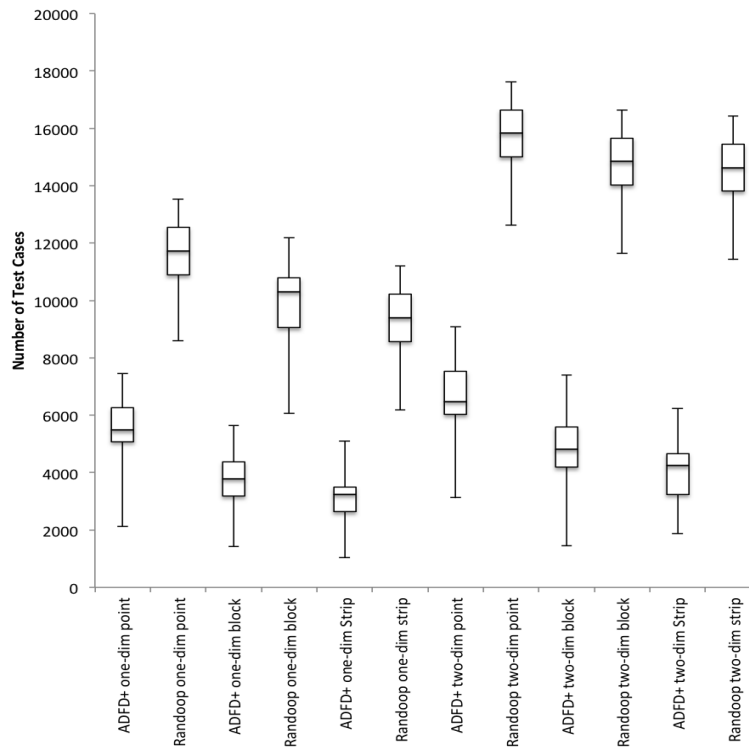
Figure 6.5: Test cases taken to find failure domains

6.4.3 Presentation of Failure Domains

The comparative results of the two tools with respect to presentation of the identified failure domains reveal better performance of ADFD$^+$ by providing the benefit of presenting the failure domains in graphical form as shown in Figure 6.6 and 6.7. The user can also enable or disable the option of showing the failing values on the graph. In comparison, Randoop lacks the ability of graphical presentation and the option of showing the failure domains separately. It provides the results scattered across textual files.

94
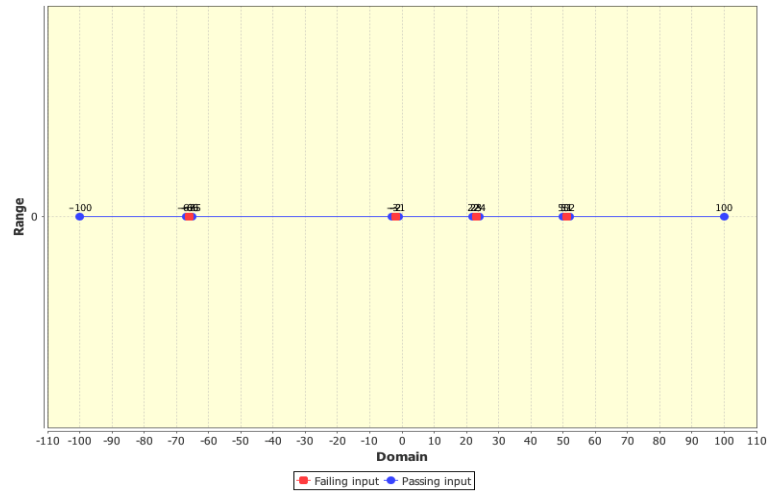
(a) Point failure domain in one-dimension
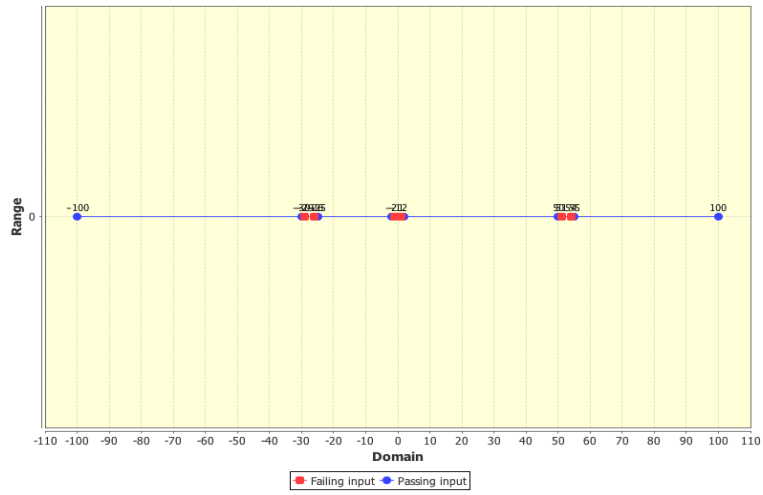
(b) Block failure domain in one-dimension
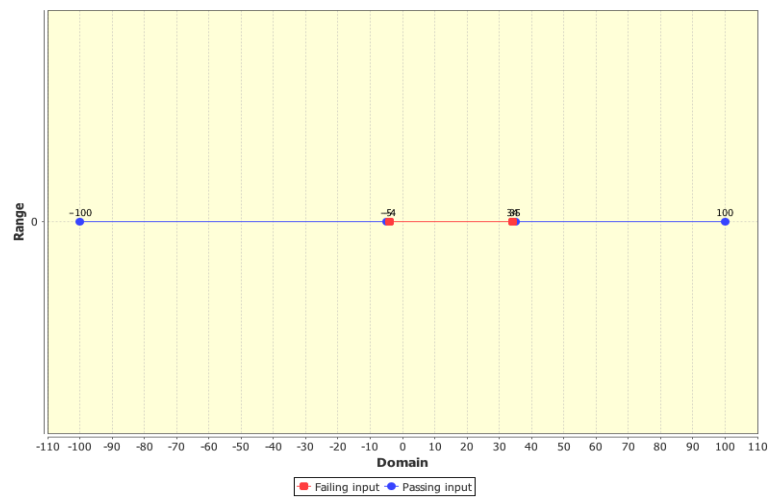
(c) Strip failure domain in one dimension

Figure 6.6: Pass and fail values plotted by ADFD$^+$ in three different cases of one-dimension programs
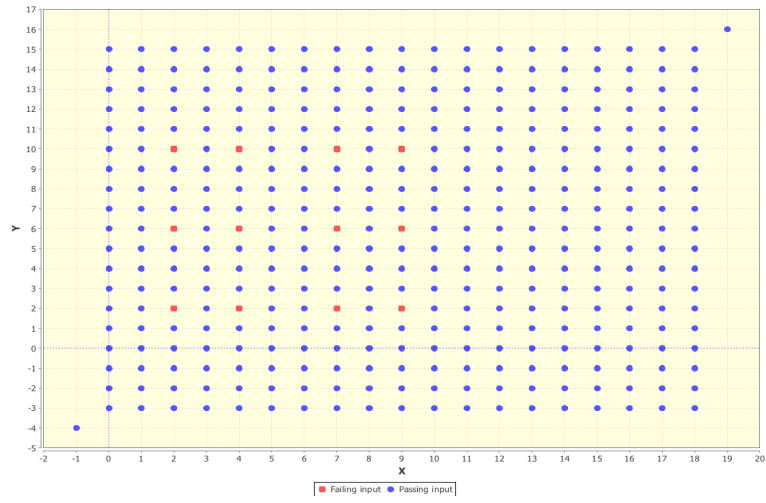
(a) Point failure domain in two-dimension

(b) Block failure domain in two-dimension

(c) Strip failure domain in two-dimension

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

96
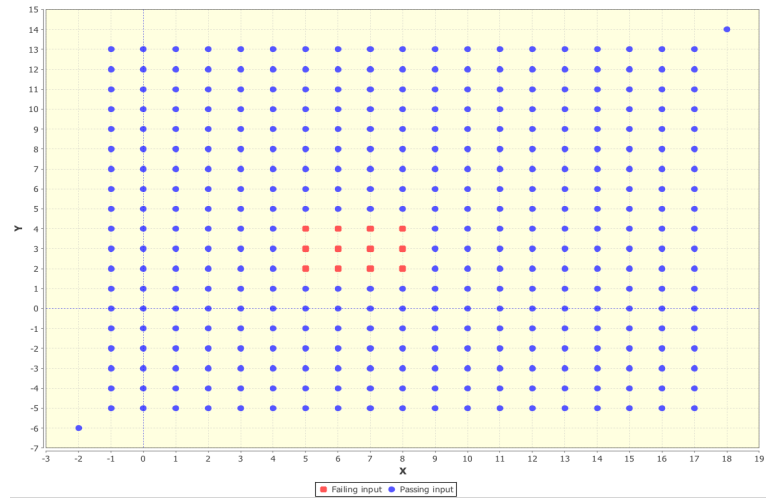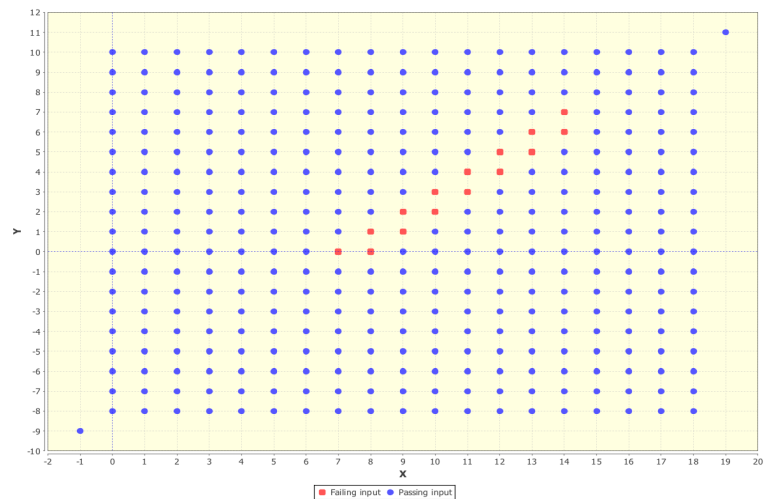
6.5 Discussion

The results indicate that ADFD$^+$ is a promising technique for finding failures and failure domains efficiently and effectively. It has the advantage of showing the results in graphical form. The pictorial representation of failure domains can serve to provide insight into the failure logic of the SUT.

In the initial set of experiments Randoop was executed for several minutes with default settings. This did not discover any faults. On analysis of the generated unit tests and Randoop's manual, it was found that the pool of values stored in Randoop database for *int* primitive type contains only 5 values including -1, 0, 1, 10 and 100. To enable Randoop to select different values, we supplied a configuration file with the option to generate random values between -500 and 500 for the test cases as all the seeded errors were in this range.

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

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

```java
public class ErrorClass {
  public static void errorMethod (int arg1, int arg2){
    if (arg1 == 10000) {
        abort();         /* error */
    }
    if (arg2 == -20000) {
        abort();         /* error */
    }
  }
}
```

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

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

6.6   Threats to Validity

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

6.7   Related Work

The increase in complexity of programs poses new challenges to researchers for finding more efficient and effective ways of software testing with user-friendly easy to understand test results. Adaptive Random Testing [24], Proportional random testing [1] and feedback-directed random testing [86] are some of the prominent upgraded versions of random testing with better performance. Automated random testing is simple to implement and capable of finding hitherto bugs in complex programs [2, 3]. ADFD$^+$ is a promising technique for finding failures and failure domains efficiently and effectively with the added advantage of presenting the output in graphical form showing point, block and strip failure domains.

Some previous research studies have reported work on identification, classification and visualisation of pass and fail domains [133, 134, 135]. This includes Xslice [134] is used to differentiate the execution slices of passing and failing part of the test in a visual form. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [135]. Hierarchical Multi Dimension Scaling (HMDS) describes a semi-automated procedure of classifying and plotting the faults [133]. A serious limitation of the above-mentioned tools is that they are not fully automated and require human intervention during execution. Moreover, these tools need the requirement of existing test cases to work on whereas ADFD$^+$ strategy generates test cases, discovers failures, identifies pass and fail domains and visualises the results in a graphical form operating in fully automated manner.

6.8 Summary

The newly developed ADFD$^+$ technique is distinct from other random testing techniques because it not only identifies failures but also discovers failure domains and presents them in an easily understandable graphical form. The chapter highlights the improved features of ADFD$^+$ in comparison with the ADFD technique presented in Chapter 5. The chapter then analyses and compares the experimental results of ADFD$^+$ and Randoop for point, block and strip failure domains. The ADFD$^+$ strongly discovered two orders of magnitude faster than Random. It also surpassed Randoop in terms of effectiveness by identifying the failure domains in 50% or less number of test cases. The better performance of ADFD$^+$ may be attributed mainly to its ability to dynamically change algorithm to exhaustive testing in a specified radius around the first identified failure as against Randoop which uses the same random algorithm continuously for searching failures.

Chapter 7

Evaluation of ADFD and ADFD$^+$ techniques

The newly developed ADFD and ADFD$^+$ techniques have been described in detail in the preceding chapters (5 and 6). Experimental evaluation of the two techniques through purpose built error-seeded numerical programs presented in the two chapters revealed that both techniques were capable of identifying the planted faults effectively. In this chapter, we have evaluated the precision of identifying the failure domains under the two techniques. For this purpose, we have incorporated Daikon in ADFD and ADFD$^+$. Daikon was selected on the basis of its capability to automatically generate invariants of failure domains, precisely point out the boundaries of failure domains and present the failure domains generated in more than two dimensional programs. We have performed extensive experimental analysis of real world Java projects contained in Qualitas Corpus. The results obtained were analysed and cross-checked using manual testing. The impact of nature, location, size, type and complexity of failure domains on the testing techniques were also studied [138].

7.1   Enhancement of the Techniques

Prior to experimental evaluation, new features were incorporated in ADFD and ADFD$^+$ techniques to: increase the code coverage, provide information about the identified failure and generate invariants of the detected failure domains as stated below:

1. The GUI is enabled to launch all the strategies defined in YETI from a single interface. For example: if the ADFD strategy is selected for testing the system automatically hides the field associated with ADFD+ (range value) and displays two fields of lower and upper bounds. On the other hand if the ADFD$^+$ strategy is selected for testing, the system automatically hides the two fields associated with the ADFD technique (lower and upper bounds) and displays a single field of range value.

2. Code coverage was increased by extending the techniques to support the testing of methods with `byte, short, long, double` and `float` type arguments while it was restricted to `int` type arguments only in the original techniques.

3. Invariants of the detected failure domains were automatically generated by integrating the tool Daikon into the two techniques. Daikon is an automated invariant detector that detects likely invariants in the program [139]. The generated invariants are displayed in GUI at the end of test execution.

4. Additional information was facilitated by adding the YETI generated failure finding test case to the GUI of the two techniques. The test case included type of failure, name of the failing class, name of the failing method, values causing the failure and line number of the code causing the failure.

5. The feature of screen capture is added to the new GUI. The user can click the screen capture button to capture the current screen during the testing process. It allows the user to capture multiple screen-shots at different intervals of testing for future reference.

Four of the above enhancements are visible from the front-end. As shown in Figure 7.1, the drop down menu for strategy field enables the tester to choose the appropriate strategy in the list for the test session. Secondly, the block failure domain is shown in graphical form and with the help of the automatic tool Daikon the failure domain is also shown by invariants (i one of {-1, 0, 1}, j == 0). Thirdly, the addition of YETI generated test case shows type of failure (RUNTIME EXCEPTION, java.lang.ArithmaticException: / by zero), name of the failing class (OneDimensionalBlockFailureDomain), name of the failing method (blockErrors), value causing the failure (1) and line number of the code causing failure (11). Fourthly, the provision of screen capture button allows the tester to store the record of each test.
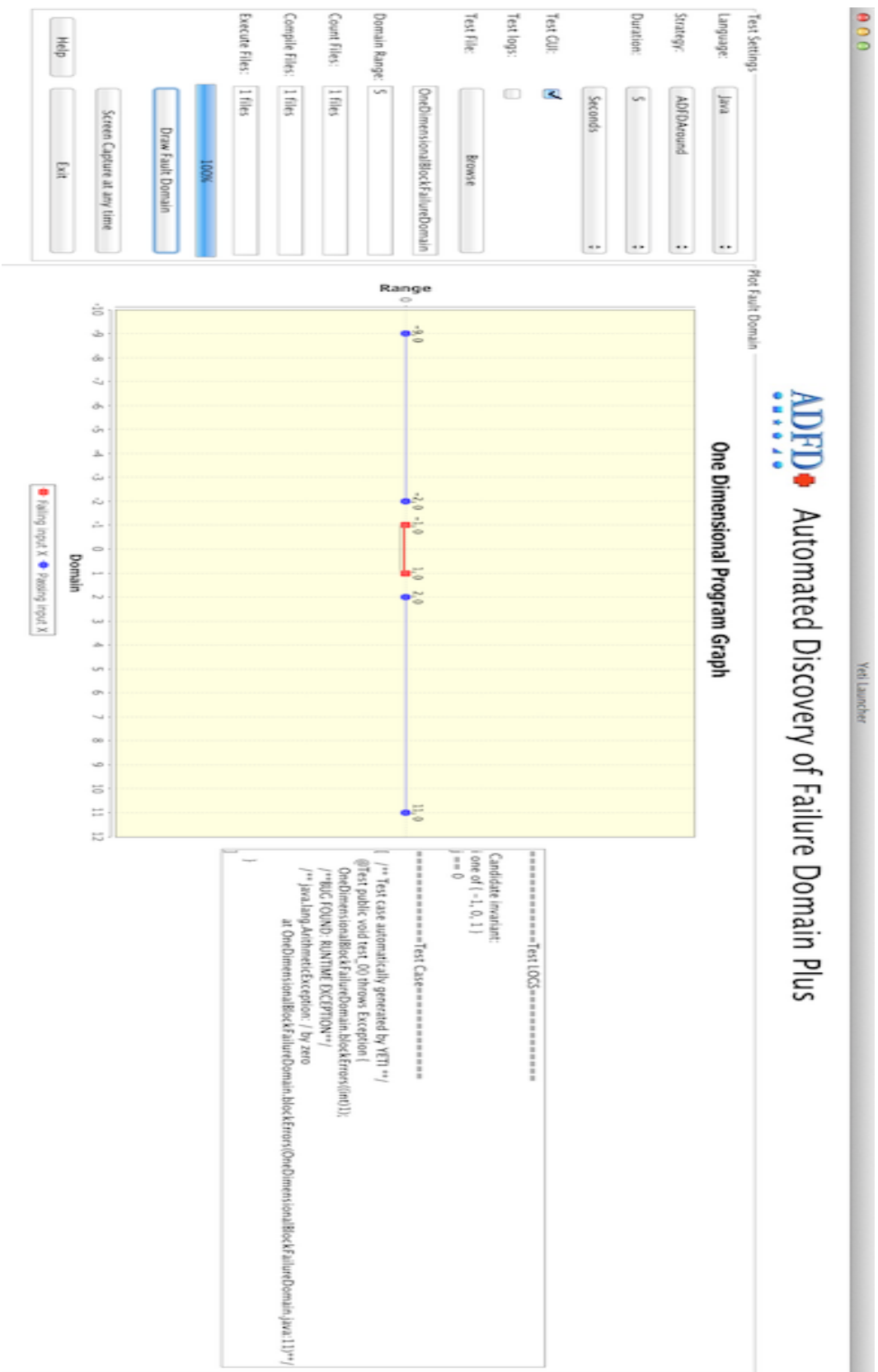
Figure 7.1: GUI front end of upgraded ADFD and ADFD$^+$

7.2 Daikon

Daikon is an automated tool that detects likely invariants at specific points in the program from its execution trace file [139]. The trace file records the effect of inputs to the program under observation during execution of the test cases. Daikon is capable of generating invariants for programs written in Java, C, C++, Perl and Eiffel. The tool helps programmers by identifying program properties, which must be preserved during modification of the program code. Daikon's output can also provide assistance on understanding, modifying and testing of programs that contain no explicit invariants.

Figure 7.2 presents the architecture of Daikon. To generate invariants for the original program, Daikon instruments the source code by inserting checks at various points in the program. The checks inserted do not change the original behaviour of the program in any way. On executing the instrumented program the check points collect values of variables accessible at that point and store them in the trace file. Then Daikon's inference engine analyses the trace file for any pattern which is true in all samples and reports it as a potential invariant. To avoid reporting false positive invariants, Daikon calculates the confidence it has that particular assertions really are invariants, and reports only those executing a set level of confidence.
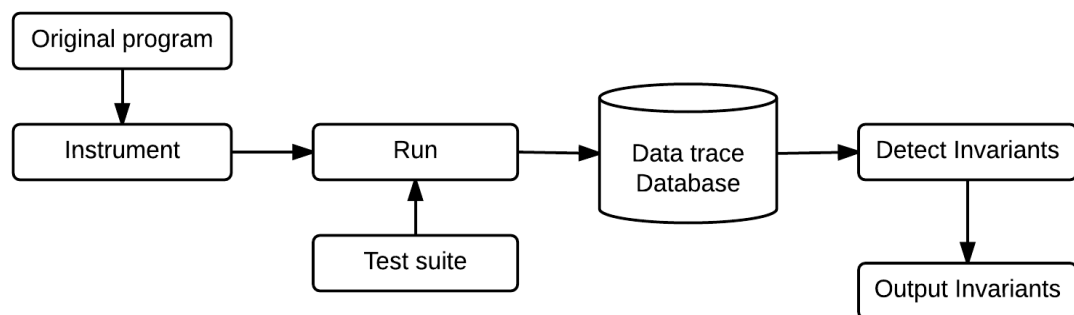
Figure 7.2: Architecture of Daikon [6]

7.2.1 Types of Invariants Detected by Daikon

Daikon is usually used to detect common types of likely invariants. However, it can be used to search for more specific invariants if it is tuned for the purpose by the user. The most

common types of invariants detected by Daikon are quoted below [6].

1. Invariants over any variable:

   (a) Constant value: x = a indicates the variable is a constant.

   (b) Uninitialized: x = uninit indicates the variable is never set.

   (c) Small value set: $x \in \{a,b,c\}$ indicates the variable takes only a small number of different values.

2. Invariants over a single numeric variable:

   (a) Range limits: $x \geq a$; $x \leq b$; and $a \leq x \leq b$ (printed as x in [a..b]) indicate the minimum and/or maximum value.

   (b) Nonzero: $x \neq 0$ indicates the variable is never set to 0.

   (c) Modulus: $x \equiv a$ mod b indicates that x mod $b \equiv a$ always holds.

   (d) Nonmodulus: $x \neq a$ mod b is reported only if x mod b takes on every value beside a.

3. Invariants over two numeric variables:

   (a) Linear relationship: y = ax + b.

   (b) Ordering comparison: $x \geq y$; $x \leq y$

   (c) Invariants over x + y: any invariant from the list of invariants over a single numeric variable.

   (d) Invariants over x - y: as for x + y

4. Invariants over a single sequence variable (arrays):

   (a) Range: minimum and maximum sequence values, ordered lexicographically.

   (b) Element ordering: whether the elements of each sequence are non-decreasing, non-increasing or equal.

   (c) Invariants over all sequence elements (treated as a single large collection): for example, all elements of an array are at least 100.

5. Invariants over two sequence variables:

   (a) Linear relationship: y = ax + b, elementwise.

   (b) Comparison: lexicographic comparison of elements.

   (c) Subsequence relationship.

7.3 Difference in Working Mechanism of the Two Techniques

The differences between the approaches of ADFD and ADFD$^+$ for identification of failure domains is illustrated by testing a simple Java program (given below) with the two techniques.

```java
/**
 * A program with block failure domain.
 * @author (Mian and Manuel)
 */
public class BlockErrorPlotTwoShort {
    public static void blockErrorPlot (int x, int y) {
        if ((x >= 4) && (x <= 8) && (y == 2)) {
            abort();          /* error */
        }
        if ((x >= 5) && (x <= 8) && (y == 3)) {
            abort();          /* error */
        }
        if ((x >= 6) && (x <= 8) && (y == 4)) {
            abort();          /* error */
        }
    }
}
```

As is evident from the program code, a failure is generated when the value of variable *x* = {*4, 5, 6, 7 or 8*} and the corresponding value of variable *y* = {*2, 3 or 4*}. The total number of 12 failing instances form a block failure domain in the input domain.

The test output generated by the ADFD technique is presented in Figure 7.3. The labelled graph shows only 4 out of 12 failing values in red whereas the passing values are shown in blue. The generated invariants identify all but one failing value ($x = 4$). This is due to the fact that ADFD scans the values in one-dimension around the failure. The test case shows the type of failure, name of the failing class, name of the failing method, values causing the failure and line number of the code causing failure.

The test output generated by ADFD$^+$ technique is presented in Figure 7.4. The labelled graph correctly shows all the 12 out of 12 available failing values in red whereas the passing values are shown in blue. The invariants correctly represent the failure domain. The test case shows the type of failure, name of the failing class, name of the failing method, values causing the failure and line number of the code causing failure.
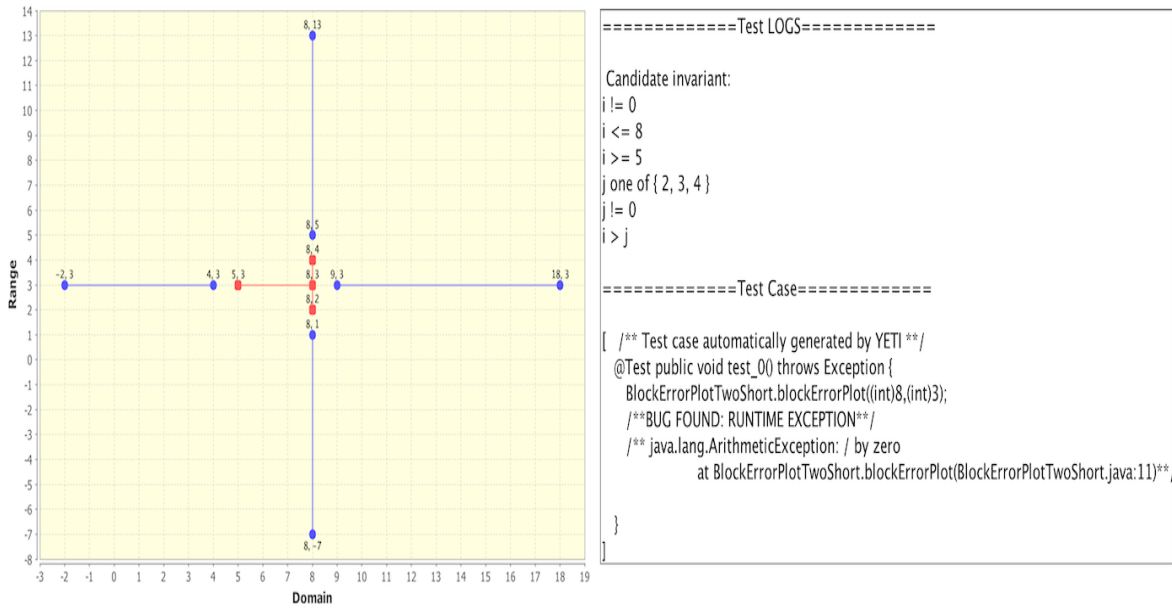
Figure 7.3: Graph, Invariants and test case generated by ADFD for the given code
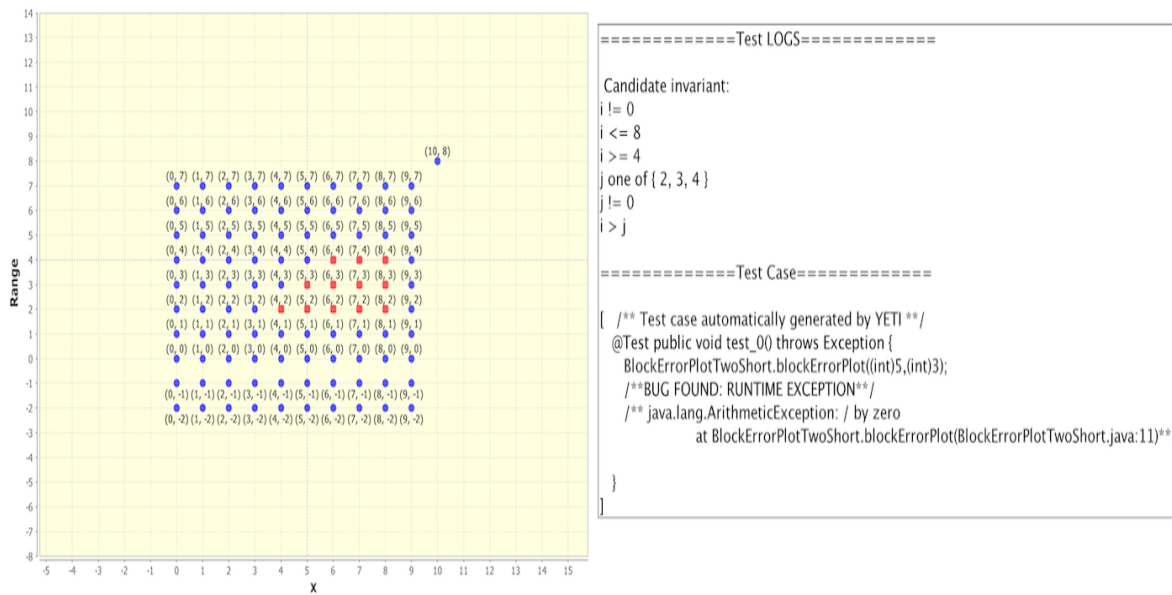
Figure 7.4: Graph, Invariants and Test case generated by ADFD$^+$ for the given code

The comparative results derived from execution of the two techniques on the developed program indicate that, ADFD$^+$ is more efficient than ADFD in identification of failures in two-dimensional programs. The ADFD and ADFD$^+$ performs equally well in one-dimensional

program, but ADFD covers a wider range around the first failure than ADFD$^+$ and is comparatively economical because it uses fewer resources (memory and CPU processing time) than ADFD$^+$.

7.4 Research Questions

The following research questions have been addressed in the study:

1. What is the relevance of ADFD and ADFD$^+$ techniques in identification and presentation of failure domains in production software?

2. What types and frequencies of failure domains exist in production software?

3. What is the nature of identified failure domains and how does this affect the automated testing techniques?

7.5 Evaluation

Experimental evaluation of ADFD and ADFD$^+$ techniques was carried out to determine: the effectiveness of the techniques in identifying and presenting the failure domains, the types and frequencies of failure domains, the nature of error causing a failure domain and the external validity of the results obtained.

7.5.1 Experiments

In the present experiments, we tested all 106 packages with no state of the Qualitas Corpus containing a total of 4000 classes. Qualitas Corpus was selected because it is a database of Java programs specially built for empirical research which takes into account a large number of developmental models and programming styles. All packages included in Qualitas Corpus are open source with an easy access to the source code.

For experimental purpose, the main ".jar" file of each package was extracted to get the ".class" files as appropriate input for YETI. All 4000 classes were individually tested. The classes containing one and two-dimensional methods with arguments (int, long, float, byte, double and short) were selected for experimental analysis. Non-numerical arguments and more than two-dimensional methods were ignored because the two proposed techniques support the testing of one and two dimensional methods with numerical arguments. Each test took 40 seconds on average to complete execution. The initial 5 seconds were used by YETI to find the first failure while the remaining 35 seconds were jointly consumed by ADFD/ADFD$^+$ technique, JFreeChart and Daikon to identify, draw graph and generate

invariants of the failure domains respectively. The machine took approximately 500 hours to perform the experiments completely. 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 [112, 108]. The source code of the programs containing failure domains were also evaluated manually to compare with the experimental results.

In accordance with Chan et al. [1], classification of failure domains into various types was based on the number of contiguous failures detected in the input-domain as shown in Table 7.1. If the contiguous failures detected range from 1 to 5, 6 to 49 or 50 and above the failure domain is classified as point, block or strip type respectively. If more than one type of domain is detected in a program, it is termed a mix type.

The ADFD and ADFD$^+$ executable files are available at https://code.google.com/p/yeti-test/downloads/list/. Daikon and JFreeChart can be seperately obtained from http://plse.cs.washington.edu/daikon/ and http://www.jfree.org/jfreechart/download.html respectively.

Table 7.1: Classification of failure domains

S. No	Type of failure domain	No of contiguous failures
1	Point	01 to 05
2	Block	06 to 49
3	Strip	50 & above
4	Mix	point & block point & strip point, block & strip

7.5.2 Results

The testing of 106 Java packages including 4000 classes, resulted in 25 packages containing 57 classes to have various types of failure domains. The details pertaining to project, class, method, dimension, line of code (LOC) and type of detected failure domains for each class are given in Table 7.2. Out of the total of 57 methods indicated in the table, 10 methods are two-dimensional while the remaining 47 methods are one-dimensional. The chosen subset of classes had a total number of 17262 lines of code spread across 57 classes in various proportions as shown in the table. The results obtained show that out of 57 classes 3 contain point failure domains, 1 contains a block failure domain, 50 contain strip failure domains and 3 contain mix failure domains. A mix failure domain includes the combination of two or more types of failure domains including point & block, point & strip and point, block & strip.

Table 7.2: Table depicting results of ADFD and ADFD$^+$

S#	Project	Class	Method	Dim.	LOC	Failure domain
1	ant	LeadPipeInputStream	LeadPipeInputStream(i)	1	159	Strip
2	antlr	BitSet	BitSet.of(i,j)	2	324	Strip
3	artofillusion	ToolPallete	ToolPalette(i,j)	2	293	Strip
4	aspectj	AnnotationValue	whatKindIsThis(i)	1	68	Mix
		IntMap	idMap(i)	1	144	Strip
5	cayenne	ExpressionFactory	expressionOfType(i)	1	146	Strip
6	collections	ArrayStack	ArrayStack(i)	1	192	Strip
		BinaryHeap	BinaryHeap(i)	1	63	Strip
		BondedFifoBuffer	BoundedFifoBuffer(i)	1	55	Strip
		FastArrayList	FastArrayList(i)	1	831	Strip
		StaticBucketMap	StaticBucketMap(i)	1	103	Strip
		PriorityBuffer	PriorityBuffer(i)	1	542	Strip
7	colt	GenericPermuting	permutation(i,j)	2	64	Strip
		LongArrayList	LongArrayList(i)	1	153	Strip
		OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	1	47	Strip
8	drjava	Assert	assertEquals(i,j)	2	780	Point
		ByteVector	ByteVector(i)	1	40	Strip
9	emma	ClassLoaderResolver	getCallerClass(i)	1	225	Strip
		ElementFactory	newConstantCollection(i)	1	43	Strip
		IntIntMap	IntIntMap(i)	1	256	Strip
		ObjectIntMap	ObjectIntMap(i)	1	252	Strip
		IntObjectMap	IntObjectMap(i)	1	214	Strip
10	heritrix	ArchiveUtils	padTo(i,j)	2	772	Strip
		BloomFilter32bit	BloomFilter32bit(i,j)	2	223	Strip
11	hsqld	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	1	52	Strip
		ObjectCacheHashMap	ObjectCacheHashMap(i)	1	76	Strip
12	htmlunit	ObjToIntMap	ObjToIntMap(i)	1	466	Strip
		Token	typeToName(i)	1	462	Mix
13	itext	PRTokeniser	isDelimiterWhitespace(i)	1	593	Strip
		PdfAction	PdfAction(i)	1	585	Strip
		PdfLiteral	PdfLiteral(i)	1	101	Strip
14	jung	PhysicalEnvironment	PhysicalEnvironment(i)	1	503	Strip
15	jedit	IntegerArray	IntegerArray(i)	1	82	Strip
16	jgraph	AttributeMap	AttributeMap(i)	1	105	Strip
17	jruby	ByteList	ByteList(i)	1	1321	Strip
		WeakIdentityHashMap	WeakIdentityHashMap(i)	1	50	Strip
18	junit	Assert	assertEquals(i,j)	2	780	Point
19	megamek	AmmoType	getMunitionsFor(i)	1	268	Strip
		Board	getTypeName(i, j)	1	1359	Mix
20	nekohtml	HTMLEntities	get(i)	1	63	Strip
21	poi	Variant	getVariantLength(i)	1	476	Block
		IntList	IntList(i,j)	2	643	Point
22	sunflow	QMC	halton(i,j)	2	32	Strip
		BenchmarkFramework	BenchmarkFramework(i,j)	2	24	Strip
		IntArray	IntArray(i)	1	47	Strip
23	trove	TDoubleStack	TDoubleStack(i)	1	120	Strip
		TIntStack	TIntStack(i)	1	120	Strip
		TLongArrayList	TLongArrayList(i)	1	927	Strip
24	weka	AlgVector	AlgVector(i)	1	424	Strip
		BinarySparseInstance	BinarySparseInstance(i)	1	614	Strip
25	xerces	SoftReferenceSymbolTable	SoftReferenceSymbolTable(i)	1	71	Strip
		SymbolHash	SymbolHash(i)	1	82	Strip
		SynchronizedSymbolTable	SynchronizedSymbolTable(i)	1	57	Strip
		XMLChar	isSpace(i)	1	169	Strip
		XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	1	96	Strip
		XML11Char	isXML11NCNameStart(i)	1	184	Strip
		AttributeList	AttributeList(i)	1	321	Strip

7.5.2.1 Effectiveness of ADFD and ADFD$^+$ techniques

The experimental results confirmed the effectiveness of the techniques by discovering all three types of failure domains (point, block and strip) across the input domain. The results obtained by applying the two automated techniques were verified: by manual analysis of the source code of all 57 classes; by cross checking the test case, the graph and the generated invariants of each class; and by comparing the invariants generated by automatic and manual techniques.

The identification of a failure domain by both ADFD and ADFD$^+$ is dependant upon the detection of failure by the random$^+$ strategy in YETI. It is due to the fact that neighbouring values of the failure are generated only after the first failure is detected by the random+ strategy.

The generation of graph and invariants and the time of test execution directly depends on the range value, if the range value of a technique is greater, the presentation of a failure domain is better and the execution time required is higher. This is due to the testing and handling of greater numbers of test cases when the range is set to a higher level. Comparatively, ADFD requires fewer resources than ADFD$^+$, therefore it is less influenced by the range value.

7.5.2.2 Type and Frequency of Failure domains

As evident from the results given in Table 7.3 - 7.6, all the three techniques (ADFD, ADFD$^+$ and Manual) detected the presence of strip, point and block types of failure domains in different frequencies. The results obtained show that out of 57 classes 3 contain point failure domains, 1 contains block failure domains, 50 contain strip failure domains and 3 contain mix failure domains. A mix failure domain includes the combination of two or more types of failure domains including point & block, point & strip and point, block & strip.

The discovery of higher number of strip failure domains may be attributed to the fact that a limited time of 5 seconds were set in YETI testing tool for searching the first failure. The ADFD and ADFD$^+$ strategies set in YETI for testing the classes are based on the random$^+$ strategy which gives high priority to boundary values, therefore, the search by YETI was prioritised to the boundary area where there were greater chances of occurrence of failures constituting strip failure domain.

Table 7.3: Class with block failure domain

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Manually generated Invariants
1	Variant	I $\geq$ 0, I $\leq$ 12	I $\geq$ 0, I $\leq$ 14, I $\geq$ 16 I $\leq$ 31, I $\geq$ 64, I $\leq$ 72	I $\geq$ 0, I $\leq$ 14, I $\geq$ 16 I $\leq$ 31, I $\geq$ 64, I $\leq$ 72

Table 7.4: Classes with point failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Manually generated Invariants
1	Assert	I == J	I == J	I == J
2	Assert	I $\leq$ 0, I $\geq$ 20 J = 0	I $\leq$ -2147483142, I $\geq$ min_int J = 0	I any value J = 0
3	IntList	I $\leq$ -1, I $\geq$ -15 J = 0	I $\leq$ -1, I $\geq$ -509 J =0	I $\leq$ -1, I $\geq$ min_int J = 0

Table 7.5: Classes with mix failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Manually generated Invariants	Failure Domains
1	Board	I $\leq$ -1 I $\geq$ -18 J = 0	I $\geq$ -504, I $\leq$ -405, I $\geq$ -403, I $\leq$ -304, I $\geq$ -302, I $\leq$ -203, I $\geq$ -201, I $\leq$ -102, I $\geq$ -100, I $\leq$ -1 J = 0	I $\leq$ -910, I $\geq$ -908, I $\leq$ -809, I $\geq$ -807, I $\leq$ -708, I $\geq$ -706, I $\leq$ -607, I $\geq$ -605, I $\leq$ -506, I $\geq$ -504, I $\leq$ -405, I $\geq$ -403, I $\leq$ -304, I $\geq$ -302, I $\leq$ -203, I $\geq$ -201, I $\leq$ -102, I $\geq$ -100 I $\leq$ -1, J = 0	Strip & Point
2	Token	I $\leq$ -2147483641 I $\geq$ min_int	I $\leq$ -2, I $\geq$ -510 I = {73, 156} I $\geq$ 162, I $\leq$ 500	I $\leq$ -2, I > min_int I = {73, 156} I $\geq$ 162, I $\leq$ max_int	Point & Strip
3	AnnotationValue	I $\leq$ 85, I $\geq$ 92, I $\geq$ 98, I $\leq$ 100, I $\geq$ 102, I $\leq$ 104	I $\leq$ 63, I = {65, 69, 71, 72} I $\geq$ 75, I $\leq$ 82, I $\geq$ 84 I $\leq$ 89, I $\geq$ 92, I $\leq$ 98 I = 100, I $\geq$ 102, I $\leq$ 114 I $\geq$ 116	I $\leq$ 63, I = {65, 69, 71, 72} I $\geq$ 75, I $\leq$ 82, I $\geq$ 84 I $\leq$ 89, I $\geq$ 92, I $\leq$ 98 I = 100, I $\geq$ 102, I $\leq$ 114 I $\geq$ 116 and so on	Point & block

Table 7.6: Classes with strip failure domains. The value of radius is 10 and 505 for ADFD and ADFD$^+$ respectively.

S#	Class	First failure	Invariants by ADFD$^+$	First failure	Invariants by ADFD	Manually generated Invariants
1	LeadPipe-InputStrea	2147483640	I $\geq$ 2147483630 I $\leq$ max_int	2147483645	I $\geq$ 2147483140 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
2	BitSet	-8 -2	I $\leq$ -1, I $\geq$ -18, J $\leq$ 7, J $\geq$ -12	-7 2	I $\leq$ -1, I $\geq$ -513 J $\geq$ -503, J $\leq$ 507	I $\leq$ -1, I $\geq$ min_int J any value
3	ToolPallete	-8 -5	I $\leq$ -1, I $\geq$ -18 J $\geq$ 3, J $\geq$ -15	-10 -4	I $\leq$ -1, I $\geq$ -515 J $\geq$ -509, J $\leq$ 501	I $\leq$ -1, I $\geq$ min_int J any value
4	IntMap	-8	I $\leq$ -1, I $\geq$ -18	-7	I $\leq$ -1, I $\geq$ -512	I $\leq$ -1, I $\geq$ min_int
5	Expression-Factory	3	I $\leq$ 13, I $\geq$ -7	8	I $\geq$ -497, I $\leq$ 513	I $\geq$ min_int I $\leq$ max_int
6	ArrayStack	2147483646	I $\geq$ 2147483636 I $\leq$ max_int	2147483647	I $\geq$ 2147483142 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
7	BinaryHeap	-2147483647	I $\leq$ -2147483637 I $\geq$ min_int	-2147483647	I $\leq$ -2147483142 I $\geq$ min_int	I $\leq$ 0 I $\geq$ min_int
8	Bonded-FifoBuffer	-2147483648	I $\leq$ -2147483638 I $\geq$ min_int	0	I $\geq$ -505, I $\leq$ 0	I $\leq$ 0 I $\geq$ min_int
9	FastArrayList	-2147483641	I $\leq$ -2147483631 I $\geq$ min_int	-2147483644	I $\leq$ -2147483139 I $\geq$ min_int	I $\leq$ -1 I $\geq$ min_int
10	Static-BucketMap	2147483645	I $\geq$ 2147483635 I $\leq$ max_int	2147483645	I $\geq$ 2147483140 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
11	PriorityBuffer	-4	I $\leq$ -1, I $\geq$ -14	-2147483647	I $\leq$ -2147483142 I $\geq$ -2147483648	I $\leq$ 0 I $\geq$ min_int
12	Generic-Permuting	-8	I $\leq$ 0, I $\geq$ -18	7	I $\geq$ -498, I $\leq$ 0 I $\geq$ 2, I $\leq$ 512	I $\leq$ 0, I $\geq$ min_int I $\geq$ 2, I $\leq$ max_int
13	LongArrayList	-2147483641	I $\leq$ -2147483631 I $\geq$ min_int	-5	I $\leq$ -1, I $\geq$ -510	I $\leq$ -1 I $\geq$ min_int
14	OpenIntDouble-HashMap	-7	I $\leq$ -1, I $\geq$ -17	-9	I $\leq$ -1, I $\geq$ -514	I $\leq$ -1, I $\geq$ min_int
15	ByteVector	-2147483648	I $\leq$ -2147483638 I $\geq$ min_int	-2147483646	I $\leq$ -2147483141 I $\geq$ min_int	I $\leq$ -1 I $\geq$ min_int
16	ElementFactory	2147483646	I $\geq$ 2147483636 I $\leq$ max_int	2147483646	I $\geq$ 2147483141 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
17	IntIntMap	-2147483648	I $\leq$ -2147483638 I $\geq$ min_int	-2147483644	I $\leq$ -2147483139 I $\geq$ min_int	I $\leq$ -1 I $\geq$ min_int
18	ObjectIntMap	2147483647	I $\geq$ 2147483637 I $\leq$ max_int	2147484096	I $\geq$ 2147483591 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
19	IntObjectMap	-7	I $\leq$ -1, I $\geq$ -17	-14	I $\leq$ -1, I $\geq$ -518	I $\leq$ -1, I $\geq$ min_int
20	ArchiveUtils	2147483641 2147483646	I $\geq$ 2147483631 I $\leq$ max_int J $\geq$ 2147483636 J $\leq$ max_int	8	I $\geq$ -497 I $\leq$ 513 J $\geq$ 2147483591 J $\leq$ max_int	I any value J $>$ 698000000
21	Bloom-Filter32bit	-10	I $\leq$ -1, I $\geq$ -18 J may be any value	-10	I $\leq$ -1, I $\geq$ -515 J may be any value	I $<$-1 J $<$-1
22	IntKeyLong-ValueHashMap	2147483645	I $\geq$ 2147483635 I $\leq$ max_int	2147484095	I $\geq$ 2147483590 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
23	ObjectCache-HashMap	-2147483647	I $\leq$ -2147483637 I $\geq$ min_int	-7	I $\geq$ -512, I $\leq$ 0	I $\leq$ 0 I $\geq$ min_int
24	ObjToIntMap	-2147483646	I $\leq$ -2147483636 I $\geq$ min_int	-2147484151	I $\leq$ -2147483646 I $\geq$ min_int	I $\leq$ -1 I $\geq$ min_int
25	PRTokeniser	-10	I $\leq$ -2, I $\geq$ -18	-4	I $\leq$ -2, I $\geq$ -509 I $\geq$ 256, I $\leq$ 501	I $\leq$ -2 , I $\geq$ min_int I $\geq$ 256 , I $\leq$ max_int
26	PdfAction	-2147483645	I $\leq$ -2147483635 I $\geq$ min_int	-9	I $\leq$ 0, I $\geq$ -514 I $\geq$ 6, I $\leq$ 496	I $\leq$ 0, I $\geq$ min_int I $\geq$ 6, I $\leq$ max_int
27	PdfLiteral	-4	I $\leq$ -1, I $\geq$ -14		I $\leq$ -1, I $\geq$ -511	I $\leq$ -1, I $\geq$ min_int
28	Physical-Environment	-1	I $\leq$ -1, I $\geq$ -11	-2147484151	I $\leq$ -2147483646 I $\geq$ min_int	I $\leq$ -1, I $\geq$ min_int
29	IntegerArray	2147483646	I $\geq$ 2147483636 I $\leq$ max_int	2147484092	I $\geq$ 2147483587 I $\leq$ max_int	I $>$ 698000000 I $\leq$ max_int
30	AttributeMap	-2147483648	I $\leq$ -2147483638 I $\geq$ min_int	-9	I $\leq$ 0, I $\geq$ -514	I $\leq$ 0 I $\geq$ min_int
31	ByteList	-4	I $\leq$ -1, I $\geq$ -14	-8	I $\leq$ -1, I $\geq$ -513	I $\leq$ -1, I $\geq$ min_int
32	WeakIdentity-HashMap	2147483646	I $\geq$ 2147483636 I $\leq$ max_int	2147483645	I $\geq$ 2147483140 I $\leq$ max_int	I $>$698000000 I $\leq$ max_int
33	AmmoType	-7	I $\leq$ -1, I $\geq$ -17	-9	I $\leq$ -1, I $\geq$ -514 I $\geq$ 93, I $\leq$ 496	I $\leq$ -1, I $\geq$ min_int I $\geq$ 93, I $\leq$ max_int
34	QMC	-2 -5	I $\leq$ -1, I $\geq$ -12 J $\leq$ -1, J $\geq$ -15	-3 -6	I $\leq$ -1, I $\geq$ -508 J $\leq$ 499, J $\geq$ -511	I $\leq$ -1, I $\geq$ min_int J any value
35	Benchmark-Framework	-3	I $\leq$ -1, I $\geq$ -13	-3	I $\leq$ -1, I $\geq$ -508	I $\leq$ -1, I $\geq$ min_int

Table 7.6: Classes with strip failure domains. The value of radius is 10 and 505 for ADFD and ADFD$^+$ respectively.

S#	Class	First failure	Invariants by ADFD$^+$	First failure	Invariants by ADFD	Manually generated Invariants
36	IntArray	-6	I ≤ -1, I ≥ -16	-2147484157	I ≤ -2147483652 I ≥ min_int	I ≤ -1 I ≥ min_int
37	TDoubleStack	-3	I ≤ -1, I ≥ -13	-6	I ≤ -1, I ≥ -511	I ≤ -1, I ≥ min_int
38	TIntStack	-2	I ≤ -1, I ≥ -12	-2147483648	I ≥ -2147483143 I ≤ min_int	I ≤ -1 I ≥ min_int
39	TLongArrayList	-6	I ≤ -1, I ≥ -16	-2147483646	I ≥ -2147483141 I ≤ min_int	I ≤ -1, I ≥ min_int
40	AlgVector	-5	I ≤ -1, I ≥ -15	-6	I ≤ -1, I ≥ -511	I ≤ -1, I ≥ min_int
41	BinarySparse-Instance	-5	I ≤ -1, I ≥ -15	-1	I ≤ -1, I ≥ -506	I ≤ -1, I ≥ min_int
42	SoftReference-SymbolTable	2147483645	I ≥ 2147483635 I ≤ max_int	2147483645	I ≥ 2147483140 I ≤ max_int	I > 698000000 I ≤ max_int
43	HTMLEntities	-7	I ≤- 1, I ≥ -17	-1	I ≥ -504, I ≤ -405, I ≥ -403, I ≤ -304, I ≥ -302, I ≤ -203, I ≥ -201, I ≤ -102, I ≥ -100, I ≤ -1	I ≤ -809, I ≤ -607, I ≥ -605, I ≤ -506, I ≥ -504, I ≤ -405, I ≥ -403, I ≤ -304, I ≥ -302, I ≤ -203, I ≥ -201, I ≤ -102, I ≥ -100, I ≤ -1
44	SymbolHash	-6	I ≤ -1, I ≥ -16	-2147483640	I ≤ -2147483135 I ≥ min_int	I ≤ -1, I ≥ min_int
45	Synchronized-SymbolTable	-2147483134	I ≤ -2147483144 I ≥ min_int	-2147483642	I ≤ -2147483137, I ≥ min_int	I ≤ -1, I ≥ min_int
46	XMLChar	-2	I ≤ -1, I ≥ -12	-5	I ≤ -1, I ≥ -510	I ≤ -1, I ≥ min_int
47	XMLGrammar-PoolImpl	-3	I ≤ -1, I ≥ -13	-2147483642	I ≤ -2147483137 I ≥ min_int	I ≤ -1, I ≥ min_int
48	XML11Char	-6	I ≤ -1, I ≥ -16	-7	I ≤ -1, I ≥ -512	I ≤ -1, I ≥ min_int
49	AttributeList	2147483645	I ≥ 2147483635 I ≤ max_int	2147483647	I ≥ 2147483142 I ≤ max_int	I > 698000000 I ≤ max_int
50	ClassLoader-Resolver	8	I ≥ 2, I ≤ 18	-5	I ≥ 500, I ≤ -2 I ≥ 2, I ≤ 505	I ≤ -2, I >min_int I ≥ 2, I ≤ max_int

7.5.2.3 Nature of failure domain

The nature of failure domain identified by two automatic techniques (ADFD and ADFD$^+$) and the manual technique was examined in terms of simplicity and complexity by comparing the invariants generated by automatic techniques with those of the manual technique. The results were split into six categories (2 categories per technique) on the basis of simplicity and complexity of failure domains identified by each of the three techniques. The comparative results show that ADFD, ADFD$^+$ and Manual testing can easily detect 56, 48 and 53 and difficultly detect 1, 9 and 4 failure domains respectively as shown in Table 7.7.

The analysis of generated invariants indicates that the failure domains which are simple in nature are easily detectable by both automated and manual techniques while the failure domains which are complex in nature are not easily detected by either automated or manual techniques.

The failure domain formed by class BitSet as shown in Table 7.6 is an example of a simple failure domain. The negativeArray failure is detected due to the input of negative value to the method bitSet.of(i). The invariants generated by ADFD are {$i \leq$ *-1, i $\geq$ -18*}, by ADFD$^+$

Table 7.7: Simplicity and complexity of Failure Domains (FD) as found by three techniques

Type of failure domain	No. of classes	No. of FD	Easy to find FD by ADFD	Easy to find FD by ADFD$^+$	Easy to find FD by MT	Hard to find FD by ADFD	Hard to find FD by ADFD$^+$	Hard to find FD by MT
Point	3	3	3	3	3	0	0	0
Block	1	1	0	1	1	1	0	0
Strip	50	50	50	45	48	0	5	2
Mix	3	6	5	4	4	1	2	2
Total	57	60	58	53	56	2	7	4

are $\{i \leq \text{-1}, i \geq \text{-512}\}$ and by Manual Analysis are $\{i \leq \text{-1}, i \geq Integer.MIN\_INT\}$. These results indicate maximum degree of representation of failure domain by Manual Analysis followed by ADFD and ADFD$^+$ respectively. This is mainly due to the bigger range value in manual analysis followed by ADFD and ADFD$^+$ respectively.

The complexity of failure domain is illustrated by taking an example of ADFD, ADFD+ and Manual Analysis in Table 7.6 for class ArrayStack. The `OutOfMemoryError` failure is detected due to the input of value to the method ArrayStack(i). The invariants generated by ADFD are $\{ i \geq 698000000, i \leq 698000300\}$, by ADFD+ are $\{ i \geq 2147483636, I \leq MAX\_INT\}$, by Manual analysis $\{ i \geq 698000000 \}$. All the three strategies indicate the same failure but at different intervals. The ADFD+ is unable to show the starting point of failure due to its small range value. The ADFD easily discovers the breaking point due to its bigger range value while manual testing requires over 50 attempts to find the breaking point.

7.6 Threats to Validity

All packages in Qualitas Corpus were tested by ADFD, ADFD$^+$ and manually in order to minimize the threats to external validity. The Qualitas Corpus contains packages of different functionality, size, maturity and modification histories. Another limitation of the evaluation is that all the experiments were carried on software with no state.

YETI using an ADFD/ADFD$^+$ strategy was executed only for 5 seconds to find the first failure in the given SUT. Since both ADFD and ADFD$^+$ are based on the random$^+$ strategy

having high preference for boundary values, therefore, most of the failures detected are from the boundaries of the input domain. It is quite possible that increasing the test duration of YETI may lead to the discovery of new failures with different failure domains.

A threat to validity is related to the hardware and software resources. For example, the `OutOfMemoryError` occurs at the value of 6980000 on the machine used for executing the test. On another machine with different specification the failure revealing value can increase or decrease depending on the hardware and software resources.

Non-numerical and methods with more than two parameters were not considered in the experiments. The failures caught due to error of non-primitive types were also ignored because of the inability of the techniques to present them graphically.

7.7 Related Work

In previous studies, researchers have done work to analyse the shape and location of failure domain in the input domain. Similar to our findings, White et al. [140] reported that the boundary values located at the edge of input domains have more chances of forming strip failure domain. Finally [141] and Bishop [142] found that failure causing inputs form a continuous region inside the input domain. Chan et al. revealed that failure causing values form certain geometrical shapes in the input domain, they classified the failure domains into point, block and strip failure domains [1].

Random testing is quick in execution and experimentally proven to detect errors in programs of various platforms including Windows [79], Unix [78], Java Libraries [3], Heskell [94] and Mac OS [80]. Its potential to become fully automated makes it one of the best choice for developing automated testing tools [2, 3]. AutoTest [120], Jcrasher [2], Eclat [3], Jartege [92], Randoop [86] and YETI [112, 108, 109] are a few of the most common automated random testing tools used by the research community. YETI is loosely coupled, highly flexible and is easily extendible [103].

In Chapters 5 and 6, we have described the fully automated techniques ADFD [108] and ADFD$^+$ [109] for the discovery of failure domains and have experimentally evaluated the performance with one and two-dimensional error-seeded numerical programs. The current study is a continuation of the previous work. It is aimed at the enhancement of the two techniques for evaluation of the precision of identifying failure domains by integrating Daikon with ADFD and ADFD+.

Our current approach of evaluation is inspired by several studies in which random testing has been compared with other testing techniques to find the failure finding ability [41, 42, 126]. The automated techniques have been compared with manual techniques in the previous research studies [4, 51]. This study is of special significance because we

compared the effectiveness of the techniques by identifying failure domains rather than individual failures considered in the previous studies.

7.8 Summary

The chapter evaluates the precision of identifying failure domains by the enhanced ADFD and ADFD$^+$ techniques integrated with the automatic tool Daikon. Extensive experimental analysis of real world Java projects contained in Qualitas Corpus were performed. The results obtained were analysed and cross-checked with the results of manual testing. The results reveal that the two techniques can effectively identify and present all types of failure domains (graphically by JFreeChart and as invariants by Daikon) to a certain level of precision. It is also evident that the level of precision of identifying failure domains can be further increased graphically and invariantly by increasing the range value in the two techniques. The analysis revealed that the strip failure domain having large size and low complexity are quickly identified by the automated techniques whereas the point and block failure domains having small size and higher complexity are difficult to identify by the automated and manual techniques. Based on the results, it can also be stated that automated techniques (ADFD and ADFD+) can be highly effective in providing assistance to manual testing but are not an alternative to the manual testing.

Chapter 8

Conclusions

The present research study aims at understanding the nature of failures in software, manipulating failure domains for finding more bugs and developing new improved automated random test strategies. The existing random test strategies find individual failures and do not focus on failure domains. Knowledge of failures as well as failure domains is highly beneficial to developers for quick and effective removal of underlying flaws in the code.

We developed three new techniques to find failures and failure domains: Dirt Spot Sweeping Random (DSSR), Automated Discovery of Failure Domain (ADFD) and Automated Discovery of Failure Domain$^+$ (ADFD$^+$) techniques. We evaluated each technique independently. ADFD and ADFD$^+$ have the potential to present a developer with the failure domain graphically and as an invariant by using JFreeChart and Daikon respectively. We also evaluated the two automated techniques independently against manual inspection of the code.

The study revealed that the input inducing failures reside in contiguous locations forming certain geometrical shapes in the input domain. These shapes can be divided into point, block and strip domains. A set of techniques has been developed for improving the effectiveness of automated random testing to find failures and failure domains.

The first technique, Dirt Spot Sweeping Random (DSSR) strategy starts by testing the program at random$^+$. When a failure is identified, the strategy selects the neighbouring input values for the subsequent tests. The selected values sweep around the identified failure leading to the discovery of new failures in the vicinity. This results in a quick and efficient identification of failures in the software under test. The results stated in Chapter 4 showed that DSSR performs significantly better than random and random$^+$ strategies.

The second technique, Automated Discovery of Failure Domain (ADFD) finds failure and failure domains in a given piece of software and provides visualization of the identified pass and fail domains within a specified range in the form of a chart. The technique starts with a

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

The third technique, Automated Discovery of Failure Domain$^+$ (ADFD$^+$) is an upgraded version of the ADFD technique with respect to the algorithm and graphical representation of the failure domain. The new algorithm searches for the failure domain around the failure in a given radius as against ADFD which limits the search between lower and upper bounds. The graphical output of ADFD$^+$ is further improved to provide labelled graphs for making it easily understandable and user-friendly. To find the effectiveness of ADFD$^+$, it is compared with Randoop using error seeded programs. The results in Chapter 6 reveal that ADFD$^+$ correctly points out all the seeded failure domains while Randoop identifies individual failures but is unable to discover the failure domains.

Finally, on the basis of comparative analysis of automated techniques (ADFD, ADFD$^+$) and manual technique, it is revealed that both the techniques can effectively identify and present types of failure domains to a certain degree of accuracy. However, the manual technique is more effective in identifying the simple (long strip) failure domain but is tedious and labour intensive. The two automated techniques are more effective in identifying and presenting complex (point and block) failure domains with minimal labour. The precision of identifying the failure domains by automated techniques can be increased by increasing the range values, provided that more resources are dedicated. Based on the results in Chapter 7, we can state that automated techniques (ADFD and ADFD$^+$) can be highly effective in providing assistance to manual testing but are not an alternative to manual testing

8.1   Lessons Learned

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

Number of failures detected per unit test as performance measurement criteria for random testing

Among the three measuring criteria usually used for finding the effectiveness of random testing, the E-measure and P-measure have been criticised [24, 85, 114] whereas the F-measure has been often used by researchers [117, 118]. In our experiments, the F-measure was initially used but its weakness was soon realised as stated in Section 4.3.3. The F-measure is effective in traditional testing and counts the number of test cases used to find the first failure. The system is then handed over to developers for fixing the identified failure. Automated testing tools test the whole system and report all discovered failures in one go, thus the F-measure is not a favourable choice. We addressed the issue by measuring the number of failures detected in a particular number of test calls as the criterion for finding the effectiveness of the testing technique.

Test results in random testing are stochastic

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

Testing of neighbouring values around the failure value decreases computation

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

Random testing coupled with exhaustive testing

Random testing generates test values at random as against exhaustive testing where the whole input domain is tested. Although exhaustive testing is quite effective it is not usually feasible for a larger domain because of infinite test values. The issue is addressed by coupling random testing with exhaustive testing. In our newly developed strategies, random

testing is used until a failure is identified; it then switches to exhaustive testing to select the values around the failure finding value in the sub-domain set by the tester. This provides the partial benefit of exhaustive testing in random testing and results in quick identification of the neighbouring failures which may be difficult to find by random testing alone.

Easy to understand user-friendly test output

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

Auto-generation of primitive and user-defined data types

We noticed that auto-generation of user-defined data types is more complex as compared to the primitive data types. We addressed the issue by creating objects of the classes under test and randomly calling the methods with random inputs in accordance with the parameter's space. The inputs are divided into primitive type and user-defined data type. For primitive data generation, `Math.random()` method is used and for generation of user-defined data, an object of the class is created at run time as stated in Section 3.2.4. The approach adopted helps in achieving a fully automated testing system.

Integration of Daikon in ADFD and ADFD$^+$ techniques

Daikon is integrated in the two automated techniques for generating invariants of failure domains to add the feature of presenting the results in textual form besides the existing available graphical form of presentation. Plotting multi-dimensional methods graphically is very hard, therefore, Daikon was introduced, which automatically detects and generates likely program invariants in multi-dimensional programs. Besides providing information about the failure domain, integration of Daikon gives two additional benefits: (1) It helps in cross checking the results with the graphical form; (2) It works as a backup method to generate invariants of failure domains in the situation when graphical presentation of the failure domain is not possible.

Effectiveness of ADFD and ADFD$^+$ techniques in production programs

The experimental analysis of production applications from Qualitas Corpus show that the automated techniques (ADFD and ADFD$^+$) are highly effective in identifying failure domains. The effectiveness of automated techniques can be enhanced by increasing the range value provided that more resources are dedicated.

Nature of failure domain

The experimental evaluation performed on Qualitas Corpus indicates that strip failure domains exist more frequently than point and block failure domains in production software. This is attributed to large size of the strip failure domains which provides more chances of choosing a value from strip failure domains when random values are generated during testing. The results also indicate that point and block failure domains are hard to discover by manual testing as compared with automated testing. This is due to the scattered nature of point and block failure domains which require large number of test cases for detection.

Chapter 9

Future Work

The chapter presents the scope and potential of future work related to the present study. The following areas are suggested:

Introducing object distance in the DSSR strategy to enhance its testing ability

The DSSR strategy has a limitation of not adding the neighbouring values when the failure is found by an object e.g. String. Future research is desirable for extending the strategy by incorporating a suitable technique like ARTOO to find the neighbouring objects of the failure finding object and add these to the list of interesting values.

Reducing the overhead of the DSSR strategy for better performances

The DSSR strategy adds neighbouring values of the failure finding value to the list of interesting values. This adds up to 5% overhead to the strategy. The algorithm needs up-grading to achieve better performance by reducing the overhead.

Extension of the ADFD and ADFD$^+$ strategies for testing non-numerical and more than two-dimensional programs

The two strategies need to be extended to facilitate testing and graphical presentation of results for non-numerical and programs with more then two inputs.

Deterministic nature of the ADFD and ADFD$^+$ strategies

Both the techniques ADFD and ADFD$^+$ are deterministic in nature. They describe how a domain is traversed once a failure is discovered in it. This also suggests that a variety of seeding techniques other than random generation can also be used.

Research on the prevalence of point, block and strip failure domains

The present study has revealed the abundance of strip failure domains as compared to point and block types within the input domain. It is worthwhile to further explore and determine the prevalence and proportionate distribution of the failure domains in the input domain of all available data types instead of only primitive data types.

Exemption of detected failure from second test execution

The ADFD and ADFD$^+$ techniques stop searching for new failures after detection of the first failure and start working on exploring the related failure domain. In the next test execution, the strategy may pick an identical failure lying on the same or different location making it a futile exercise. Efforts are required to incorporate the feature of exempting identified failures from the next test execution so that new failures and failure domains are discovered each time.

Glossary

Branch: Conditional transfer of control from one statement to another in the code.

Correctness: Ability of software to perform according to the given specifications.

Dead code Unreachable code in program that cannot be executed.

Defect: Generic term referring to fault or failure.

Detection: Difference between observed and expected behaviour of program.

Effectiveness: Number of defects discovered in the program by a testing technique.

Efficiency: Number of defects discovered per unit time by a testing technique.

Error: Mistake or omission in the software.

Failure: Malfunction of a software.

Fault: Any flaw in the software resulting in lack of capability or failure.

Instrumentation: Insertion of additional code in the program for analysis.

Invariant: A condition which must hold true during program execution.

Isolation: Identification of the basic cause of failure in software.

Path: A sequence of executable statements from entry to exit point in software.

Postcondition: A condition which must be true after execution.

Precondition: A condition which must be true before execution.

Robustness: The degree to which a system can function correctly with invalid inputs.

Test case: An artefact that delineates the input, action and expected output.

Test coverage: Number of instructions exercised divided by total number of instructions expressed in percentage.

Test execution: The process of executing test case.

Test oracle: A mechanism used to determine whether a test has passed or failed.

Test Plan: A document which defines the goal, scope, method, resources and time schedule of testing.

Test specification: The requirements which should be satisfied by test cases.

Test strategy: The method which defines the procedure of testing of a program.

Test suite: A set of one or more test cases.

Validation: Assessment of software to ensure satisfaction of customer requirements.

Verification: Checking of software for verification of working properly.

Appendix

Error-seeded code to evaluate ADFD and ADFD+

Program 1 Point domain with One argument

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

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

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

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

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

Program 2 Point domain with two argument

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

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

}
```

Program 3 Block domain with one argument

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


public class BlockDomainOneArgument{

public static void blockErrors (int x){

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

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

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

    }
}
```

Program 4 Block domain with two argument

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

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

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

    }

}
```

Program 5 Strip domain with One argument

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

public class StripDomainOneArgument{

    public static void stripErrors (int x){

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

Program 6 Strip domain with two argument

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

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

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

    }

}
```

Program generated by ADFD on finding fault in SUT

```java
/**
 * 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) {
```

```java
                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);
        }
    }
}
```

References

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

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

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

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

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

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 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] William E. Howden. A functional approach to program testing and analysis. *IEEE Transactions on Software Engineering*, (10):997–1005, 1986.

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

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

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

[16] 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.

[17] Edsger Wybe Dijkstra. *Notes on structured programming*. Technological University Eindhoven Netherlands, 1970.

[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] 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.

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

[22] 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.

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

[24] 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.

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

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

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

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

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

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

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

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

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

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

[35] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[36] Jean Arlat. *Validation de la sûreté de fonctionnement par injection de fautes, méthode- mise en oeuvre- application*. PhD thesis, 1990.

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

[38] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer, IEEE*, 30(4):75–82, 1997.

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

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

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

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

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

[44] Muthu Ramachandran. Testing software components using boundary value analysis. In *Proceedings of 29th Euromicro Conference*, pages 94–98. IEEE, 2003.

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

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

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

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

[49] Ian Sommerville. *Software Engineering: Pearson New International Edition*. Pearson Education Limited, 2013.

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

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

[52] G. Venolia, Robert DeLine, and Thomas LaToza. Software development at microsoft observed. *Microsoft Research, TR*, 2005.

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

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

[55] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

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

[57] Reto Kramer. iContract-the Java design by contract tool. In *Proceedings of Tools 26 - USA 98, IEEE Computer Society*, pages 295–307. IEEE, 1998.

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

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

[60] 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.

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

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

[63] 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.

[64] Bogdan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th international conference on Software engineering*, pages 71–80. IEEE Computer Society, 1996.

[65] 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.

[66] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[67] Phil McMinn. Search-based software testing: Past, present and future. In *International Workshop on Search-Based Software Testing (SBST 2011)*, pages 153–163. IEEE, 2011.

[68] Bryan F. Jones, H-H Sthamer, and David E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

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

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

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

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

[73] 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.

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

[75] 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.

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

[77] 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.

[78] 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.

[79] 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.

[80] 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.

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

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

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

[84] 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.

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

[86] 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.

[87] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .NET with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 87–96. ACM, 2008.

[88] 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.

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

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

[91] 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.

[92] 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.

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

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

[95] 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.

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

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

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

[99] 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.

[100] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software Engineering ESEC/FSE 99*, pages 285–302. Springer, 1999.

[101] 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.

[102] Manuel Oriol. YETI: York Extensible Testing Infrastructure. 2011.

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

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

[105] Vasileios Dimitraiadis. Testing JML code with YETI. Master's thesis, Department of Computer Science, The University of York, September 2009.

[106] Muneeb Waseem Khawaja. YETI: increase branch coverage. Master's thesis, Department of Computer Science, The University of York, September 2010.

[107] Mian Asbat Ahmad and Manuel Oriol. Dirt Spot Sweeping Random Strategy. *Lecture Notes on Software Engineering*, 2(4), 2014.

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

[109] Mian A. Ahmad and Manuel Oriol. ADFD+: An automatic testing technique for finding and presenting failure domains. *Lecture Notes on Software Engineering*, 02(4):331–336, 2014.

[110] 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.

[111] 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.

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

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

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

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

[116] Stéphane Ducasse, Manuel Oriol, and Alexandre Bergel. Challenges to support automated random testing for dynamically typed languages. In *Proceedings of the International Workshop on Smalltalk Technologies*, page 9. ACM, 2011.

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

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

[119] 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.

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

[121] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Good random testing. In *Reliable Software Technologies-Ada-Europe 2004*, pages 200–212. Springer, 2004.

[122] 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.

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

[124] 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.

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

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

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

[128] 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.

[129] 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.

[130] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[131] 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.

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

[133] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 465–475. IEEE, 2003.

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

[135] 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.

[136] Per Runeson, Carina Andersson, Thomas Thelin, Anneliese Andrews, and Tomas Berling. What do we know about defect detection methods?[software testing]. *Software, IEEE*, 23(3):82–90, 2006.

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

[138] Mian A. Ahmad and Manuel Oriol. Evaluation of ADFD and ADFD+ techniques. In *Seventh York Doctoral Symposium on Computer Science & Electronics*, page 47, 2014.

[139] 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.

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

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

[142] Peter G Bishop. The Variation of Software Survival Time for Different Operational Input Profiles (or why you can wait a long time for a big bug to fail). In *23rd International Symposium on Fault-Tolerant Computing, 1993. FTCS-23*, pages 98–107. IEEE, 1993.