New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group
Department of Computer Science
University of York, UK

Nov. 2013

A thesis submitted for the degree of Doctor of Philosophy

Abstract

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

Contents

List of Figures

List of Tables

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

Acknowledgements

The years I spent working on my PhD degree at the University of York have undoubtedly been some of the most joyful and rewarding in my academic career. The institution provided me with everything I need to thrive: challenging research problems, excellent company, and a supportive environment. I am deeply grateful to the people who shared this experience with me and to those who made it possible.

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

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

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

I was funded by Departmental Overseas Research Scholarship (DORS), a financial support awarded to overseas students on the basis of outstanding academic ability and research potential. I am truly grateful to the Department of Computer Science for financial support that allowed me to concentrate on my research.

Chapter 1

Introduction

Software is a very important and essential component of computer system without which no task can be accomplished. Some softwares are developed for use in simple day to day operations while others are for highly complex processes in specialized fields like research and education, business and finance, defence and security, health and medicine, science and technology, aeronautics and astronautics, commerce and industry, information and communication, environment and safety etc. The ever increasing dependency of software expect us to believe that the software in use is reliable, robust, safe and secure. Unfortunately the performance of software in general is not what is expected. According to the National Institute of Standards and Technology (NIST) US companies alone bear $60 billion loss every year due to software failures and one-third of that can be eliminated by an improved testing infrastructure [31]. Humans are prone to errors and programmers are no exceptions. Maurice Wilkes [76], a British computer pioneer, stated that:

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

The margin of error in mission-critical and safety-critical systems is so small that a minor fault can lead to huge economic losses [39]. Therefore, software companies leave no stone unturned to ensure the reliability and accuracy of the software. Software testing is by far the most recognized and widely used technique to verify the correctness and ensure quality of the software [49, 77, 61, 31]. According to Myers et al. some software companies spend up to 50% cost of the total cost of software development and maintenance on testing [49].

In the IEEE standard glossary of software engineering terminology [2], testing is defined as "the process of exercising or evaluating a system or system component by manual

or automated means to verify that it satisfies specified requirements and actual results. This dissertation is a humble contribution to the literature on the subject, with the aim to reduce the overall cost of software testing by devising new, improved and effective automated software testing techniques based on random strategy.

Random testing is a process (Figure 1.1) in which generation of test data is random but according to requirements, specifications or any other test adequacy criteria. The given SUT is executed against the test data and results obtained are evaluated to determine whether the output produced satisfies the expected results.

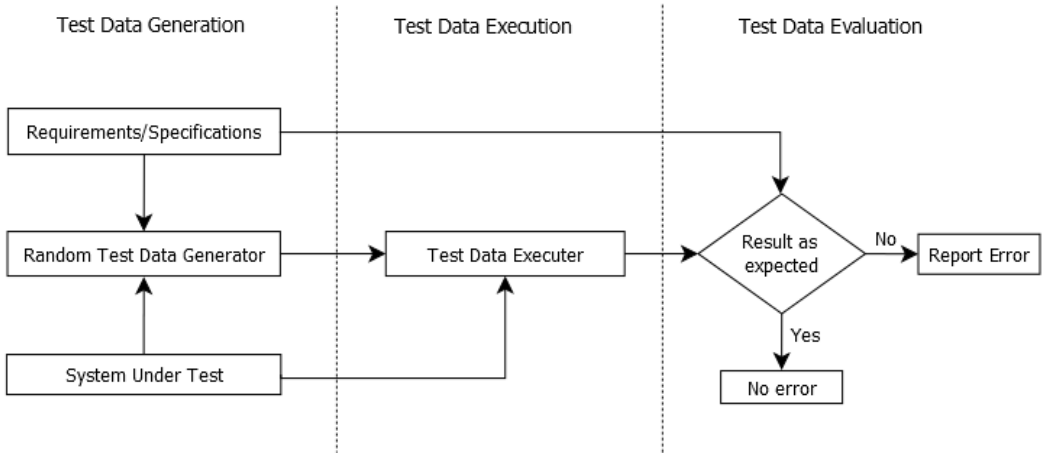

Figure 1.1: Three main phases of random testing

1.1 The Problems

Exhaustive testing of software is not always possible and the problem of selecting a test data set, from a large/infinite domain is often confronted. Test data set, as a subset of the whole domain, is carefully selected for testing the given software. Adequate test data set is a crucial factor in any testing technique because it represents the whole domain for evaluating the structural and/or functional properties [38, 46]. Miller and Maloney were the first who comprehensively described a systematic approach of test data set selection known as path coverage. They proposed that testers select the test data so that all paths of the SUT are executed at least once [47]. The implementation of the strategy resulted in higher standards of test quality and a large number of test strategies were developed afterwords including boundary value analysis and equivalence class.

Generating test data set manually is a time-consuming and laborious exercise [42]; Therefore, automated test data set generation is always preferred. Data generators can be

of different types i.e. Path-wise, Goal-Oriented, Intelligent or Random [75]. Random generator produces test data set randomly from the whole domain. Unlike other approaches random technique is simple, widely applicable, easy to implement, faster in computation, free from bias and costs minimum overhead [19]. According to Godefroid et al., "Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs" [34].

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [74]. Myers et al. [49] mentioned it as, "Probably the poorest methodology of all is random-input testing...". However, Ciupa et al. reported that the above stated statement is based on intuition and lacks any experimental evidence [20]. The criticism motivated the researchers to look into various aspects of random testing for evaluation and possible improvement. Adaptive random testing (ART) [13], Restricted Random Testing (RRT) [8], Feedback Directed Random Testing (FDRT) [59], Mirror Adaptive Random Testing (MART) [14] and Quasi Random Testing (QRT) [16] are a few of the enhanced random testing techniques reported in the literature.

Random testing is also considered weak in providing high code coverage [24, 51]. For example, in random testing when the conditional statement "*if (x == 25) then ...* " is exposed to execution then there is only one chance, of the "*then...*" part of the statement, to be executed out of 2^{32}. If x is an integer variable of 32 bit value [34].

Random testing is no exception when it comes to the complexity of understanding and evaluating test results. Modern testing techniques simplify results by truncating the lengthy log files and displaying only the fault revealing test cases in the form of unit tests. Further efforts are required to get the test results of random testing in more compact and user-friendly way.

1.2 Research Goals

The main goal of the research study is to develop new techniques for automated random testing with the aim to achieve the following objectives:

1. To develop a testing strategy with the capability to generate more fault-finding test data.

2. To develop a testing technique for finding faults, fault domains and presentation of results on a graphical chart within the specified lower and upper bound.

3. To develop a testing framework with focus on increase in code coverage along with generation of more fault-finding test data.

1.3 Contributions

The main contributions of the thesis research are stated below:

1.3.1 Dirt Spot Sweeping Random Strategy

The fault-finding ability of the random testing technique decreases when the failures lie in contiguous locations across the input domain. To overcome the problem, a new automated technique: Dirt Spot Sweeping Random (DSSR) strategy was developed. It is based on the assumption that unique failures reside in contiguous blocks and stripes. When a failure is identified, the DSSR strategy selects neighbouring values for the subsequent tests. Resultantly, selected values sweep around the failure, leading to the discovery of new failures in the vicinity. Results presented in Chapter **??** indicated higher fault-finding ability of DSSR strategy as compared with Random (R) and Random+ (R+) strategies.

1.3.2 Automated Discovery of Failure Domain

The existing random strategies of software testing discover the faults in the SUT but lack the capability of locating the fault domains. In the current research study, a fully automated testing strategy named, "Automated Discovery of Failure Domain (ADFD)" was developed with the ability to find the faults as well as the fault domains in a given SUT and provides visualization of the identified pass and fail domains in the form of a chart. The strategy is described, implemented in YETI, and practically illustrated by executing several programs of one and two dimensions in the Chapter **??**. The experimental results proved that ADFD strategy automatically performed identification of faults and fault domains along with graphical representation in the form of chart.

1.3.3 Invariant Guided Random+ Strategy

Another random test strategy named, "Invariant guided Random+ Strategy (IGRS)" was developed in the current research study. IGRS is an extended form of Random+ strategy guided by software invariants. Invariants from the given SUT are collected by Daikon— an automated invariant detector for reporting likely invariants and adding them to the SUT as assertions. The IGRS is implemented in YETI and generates values in compliance with the added assertions. Experimental results presented in Chapter **??** indicated improved features of IGR+S in terms of higher code coverage and identification of subtle errors that R, R+ and DSSR strategies were either unable to accomplish or required larger duration.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows:

Chapter 2 is a thorough review of the relevant literature, describing software testing and its common types. It then introduces the reader to random testing, describing various versions of random testing and the most commonly used automated testing tools based on random algorithms.

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

Chapter ?? presents the newly developed Automated Discovery of Fault Domains (ADFD) strategy, which focuses on dynamically finding the faults and domains along with their graphical representation. It is shown that the presence of fault domains across the input domain, which have not been so far identified, can in fact be identified and graphically represented using ADFD strategy.

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

Chapter ??is the concluding chapter, summarising the achievements of this research and discussing the extent to which our experiments have supported the hypothesis.

Appendix ??is the concluding chapter, summarising the achievements of this research and discussing the extent to which our experiments have supported the hypothesis.

Chapter 2

Literature Review

The very famous quote of, “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. Errors are not tolerated, particularly in software because a single error may cause a large upset in the system. According to the National Institute of Standard and Technology 2002, software errors cost an estimated $59.5 billion loss to US economy annually. The destruction of the Mariner 1 rocket (1962) costing $18.5 million was due to a small error in formula coded incorrectly by programmer. The Hartford Coliseum Collapse (1978) costing $70 million, Wall Street crash (1987) costing $500 billion, failing of long division by Pentium (1993) costing $475 million, Ariane 5 Rocket disaster costing $500 million are a few examples of losses caused by minor errors in the software. To achieve high quality, the software has to satisfy rigorous stages of testing. The more critical and complex the software, the higher the requirements for software testing and the larger the extent of damage caused as a result of an error in the software.

2.1 Software Testing

In the IEEE standard glossary of software engineering terminology [2], testing is defined as “the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements and actual results. A successful test is one that finds a fault [48], where fault is defined as, the error made by people during software development [2].

Being an integral part of Software Development Life Cycle (SDLC), the testing process starts from requirement phase and continues throughout the life of the software. In traditional testing when tester finds a fault in the given SUT, the software is returned to the developers for rectification and is consequently given back to the tester for retesting. It is important to note that, “program testing can be used to show the presence of bugs, but

never to show the absence of bugs" [27]. In other words, a SUT that passes all the tests without giving a single error is not guaranteed to contain no error. However, the testing process increases the reliability and confidence of users in the tested product.

Table 2.1: Parts of Software Testing [1, 17, 33, 65, 67]

Levels	Purpose	Perspective	Execution
1. Unit 2. Integration 3. System	1. Functionality 2. Structural 3. Robustness 4. Stress 5. Compatibility 6. Performance	1. White Box 2. Black Box	1. Static 2. Dynamic

2.1.1 Software Testing Levels

The three main levels of software testing defined in the literature include unit testing, integration testing and system testing [17]. Unit testing involves evaluation of piece-by-piece code and each piece is considered as independent unit. Units are combined together to form components. Integration testing ensures that the integration of units in a component is working properly. System testing is called out to make sure that the system formed by combination of components performs correctly to give the desired output.

2.1.2 Software Testing Purpose

The primary purpose of software testing is identification of faults in the given SUT for necessary correction in order to achieve high quality. Maximum number of faults can be identified if software is tested exhaustively. In exhausting testing SUT is checked against all possible combinations of input data, and the results obtained are compared with the expected results for assessment. Exhaustive testing is not always possible in most scenarios because of limited resources and infinite number of input values that software can take. Therefore, the purpose of testing is generally directed to achieve confidence in the system involved from a specific point of view. For example, functionality testing is performed to check functional aspect for working correctly. Structural testing analyses the code structure for generating test cases in order to evaluate paths of execution and identification of unreachable or dead code. In robustness testing the software behaviour is observed in the case when software receives input outside the expected input range. Stress and performance testing aims at testing the response of software under high load and its ability to

process different nature of tasks [25]. Finally, compatibility testing is performed to see the interaction of software with underlying operating system.

2.1.3 Software Testing Perspective

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

2.1.3.1 White-box testing

In white-box or structural testing, the testers must do need to know about the complete structure of the software and can do necessary modification, if so required. Test cases are derived from the code structure and test passes only if the results are correct and the expected code is followed during test execution [55]. Some of the most common White-box testing techniques are briefly defined:

Data Flow Analysis Data Flow Analysis is a testing technique that focuses on the input values by observing the behaviour of respective variables during the execution of the SUT [23]. In this technique a control flow graph (CFG), graphical representation of all possible states of program, of a SUT is drawn to determine the paths that might be traversed by a program during its execution. Test cases are generated and executed to verify its conformance with CFG on the basis of data.

Normally, program execution implies input of data, operations on it according to the defined algorithm, and output of results. This process can be viewed as a flow of data from input to output in which data may transform into several intermediate results before reaching its final state. In the process several errors can occur e.g. references may be made to variables that dont exist, values may be assigned to undeclared variables or the value of variables may be changed in an unexpected and undesired manner. It is the ordered use of data implicit in this process that is the central object of the technique to ensure that none of the aforementioned errors occur [32].

Control flow analysis Control flow Analysis is a testing technique which takes into consideration the control structure of a given SUT. Control structure is the order in which the individual statements, instructions or function calls are executed. In this technique a control flow graph (CFG), similar to the one required in data flow analysis, is drawn to determine the paths that might be traversed by a program during its execution. Test cases are generated and executed to verify its conformance with CFG on the basis of control. For example

to follow a specific path (also known as branch) between two or more choices at specific state. Efforts are made to ensure that the set of selected test cases execute all the possible control choices at least once. The effectiveness of the testing technique depends on controls measurement. Two of the most common measurement criteria defined by Vilkomir et al. are Decision/Branch coverage and Condition coverage [69].

Code-based fault injection testing A technique in which additional code is added to the code of the SUT at one or more locations to analyse its behaviour in response to the anomaly [71]. The process of code addition is called instrumentation which is usually performed before compilation and execution of software. The added code can be use for multiple reasons i.e. injection of fault to find the error handling behaviour of software, to determine the effectiveness of test procedure to check whether it discover the injected faults or to measure the code coverage achieved by the testing process.

2.1.3.2 Black-box testing

In black-box or functional testing, the testers do not need to know about internal code structure of the SUT. Test cases are derived from the specifications and test passes if the result is according to expected output. Internal code structure of the SUT is not taken into any consideration [4]. Some of the most common black-box testing techniques are briefly defined:

Use-case based testing A verification and validation technique that utilizes use-cases of the system to generate test cases. Use-case define functional requirements at a particular situation or condition of the system from actor's (user or external system) perspective. It consists of a sequence of actions to represent a particular behaviour of the system. A use case format include a brief description of the event, flow of events, preconditions, postconditions, extension points, context diagram and activity diagram. All the details required for test case is included in the use case therefore use case can be easily transformed into test case.

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

Partition testing A testing technique in which the input domain of a given SUT is divided into sub-domain according to some rule and then tests are conducted in each sub-domain.

The division in to sub-domain can be according to the requirements or specifications, structure of the code or according to the process by which the software was developed [35].

While the performance of partition testing is directly dependant on the quality of sub-domain [73], it is often however difficult to divide the domain into equal partitions. Therefore, another version of partition testing called Proportional sampling strategy: guidelines for software testing practitioners [7] is devised, in which the number of test cases selected from each partition is directly proportional to the size of the partition. Experiments performed by Ntafos [50] confirm the better performance of proportional partition testing over partition testing.

Boundary value analysis Boundary Value Analysis (BVA) is a testing technique which is based on the rationale that errors tends to occur near the extremities of the input variables. Therefore in BVA the data set consists of values which are selected from the borders. According to IEEE standards [62], boundary value is a value that corresponds to minimum or maximum input, internal or external value specified for a component or system.

The BVA technique is also used in conjunction with partition testing where test values are selected at the borders of each sub-domain instead of the whole input domain. The empirical analysis performed by Reid et al. [64] argue that BVA performs better in finding faults than partition testing. They also stated that like partition testing the performance of BVA is also dependant on the correct identification of partition and selection of boundary values.

Formal specification testing Formal specification is defined as "a mathematical based technique, which offers rigorous and effective way to model, design and analyse computer systems" [26, 37]. The mathematical structure allows formal specifications to be manipulated mechanically so that information contained within the specification can be isolated, transformed, assembled, and repackaged to be used as test cases. Furthermore, it also guarantee that the test frames are logical consequences of the specification [28]. Formal specification testing is effective because it is independent from the code of the SUT. Which means that no change is required in the test cases as long as the specifications are unchanged [33]. It uses the existing specification model to verify the test results and thus avoid the oracle problem [5].

2.1.4 Software Testing Execution

Testing process can be divided into static and dynamic phases on the basis of test execution. In static testing test cases are analysed statically for checking errors without test

execution. In addition to software code, high quality softwares are accompanied by necessary documentation. It includes requirements, design, technical, user manual marketing information. Reviews, walkthroughs or inspections are most commonly used techniques for static testing. In dynamic testing the software code is executed and input is converted into output. Results are analysed against expected outputs to find any error in the software. Unit testing, integration testing, system testing, and acceptance testing are most commonly used as dynamic testing methods [30].

2.1.5 Manual Testing

Manual testing is the technique of finding faults in software in which the tester writes the code by hand to create test cases and test oracle [21]. Manual testing may be effective in some cases but it is generally laborious, time consuming and error-prone [68]. Additionally, it requires that the testers must have appropriate skills, experience and sufficient knowledge of the SUT for evaluation from different perspectives.

2.1.6 Automated Testing

Automated testing is the technique of finding faults in a software in which the test cases or test oracle are generated automatically by a testing tool [43]. There are tools which can automate part of a test process like generation of test cases or execution of test cases or evaluation of results while other tools are available which can automate the whole testing process. The use of automated testing made it possible to test large volumes of code, which would have been impossible otherwise [63].

2.1.7 Test Oracle

Test oracles set the acceptable behaviour for test executions [3]. All software-testing techniques depend on the availability of test oracle [33]. Designing test oracle for ordinary software may be simple straightforward. However, for relatively complex software designing of oracle is quite cumbersome and requires special ways to overcome the oracle problem. Some of the common issues in the oracle problem include:

1. The assumption that the test results are observable and can be compared with the oracle.

2. An ideal test oracle would satisfy desirable properties of program specifications [3].

3. A specific oracle to satisfy all conditions is seldom available as rightly pointed out by Weyuker who ?states that truly general test oracles are often unobtainable Weyuker [72]

2.2 Random Testing

Random testing is a dynamic black-box testing technique in which the software is tested with non-correlating or unpredictable test data from the specified input domain [8]. The input domain is a set of all possible inputs to the software under test. According to Richard [36], in random testing, input domain is first identified, then test points are randomly taken from the whole input domain by means of random number generator. The program under test is executed on these points and the results obtained are compared with the program specifications. The test fails if the results are not according to the specifications and vice versa. Failure of any test results reflects error in the SUT.

Figure 2.1: Random Testing

Generating test data by random generator is quite economical and requires less intellectual and computational efforts [20]. This feature makes it an ideal choice for use in automated testing tools [—]. Moreover, no human intervention is involved in data generation that ensures an unbiased testing process.

However, generating test cases with out using any background information makes random testing susceptible to criticism. It is criticized for generating many test cases ending up at the same state of software. It is further stated that, random testing generates test inputs that violates requirements of the given SUT making it less effective [66, 56]. Myers mentioned random testing as one of the least effective testing technique [48].

It is stated by ——— that Myers statement was not based on any experimental evidence. Later experiments performed by several researchers [36, 21, 44, 29] confirmed that random testing is as effective as any other testing technique. It was found experimentally [29] that random testing can also discover subtle faults in a given SUT when subjected to large number of test cases. It was pointed out that the simplicity and cost effectiveness of random testing makes it more feasible to run large number of test cases as opposed to systematic testing techniques which require considerable time and resources for test case generation and execution. The empirical comparison proves that random testing and partition testing are equally effective [35] A comparative study conducted by Ntafos [50]. concluded that random testing is more effective as compared to proportional partition testing.

2.3 Various versions of random testing

Researchers have tried various approaches to bring about improvement in the performance of random testing. The prominent modifications in random strategy are stated below:

2.3.1 Adaptive Random Testing

Adaptive random testing (ART), proposed by Chen et al. [13] is based on the previous work of Chan et al. [7]regarding the existence of failure patterns across the input domain. Chan et al observed that failure inducing inputs in the whole input domain form certain geometrical patterns and divided these into point, block and strip patterns described below.

1. Point pattern: In the point pattern inputs inducing failures are scattered across the input domain in the form of stand-alone points. Example of point pattern is the division by zero in a statement total = num1/num2; where num1, num2 and total are variables of type integer.

2. Block pattern: In the block pattern inputs inducing failures lie in close vicinity to form a block in the input domain. Example of block pattern is failure caused by the statement if ((num >10) && (num <20)). Here 11 to 19 are a block of faults.

3. Strip pattern: In the strip pattern inputs inducing failures form a strip across the input domain. Example of strip pattern is failure caused by a statement num1 + num2 = 20. Here multiple values of num1 and num2 can lead to the fault value 20.

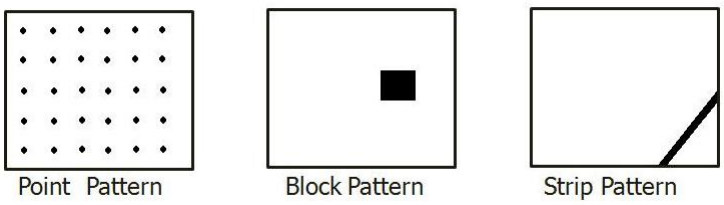

Figure 2.2: Patterns of failure causing inputs

In figure 2.2 the square boxes indicate the whole input domains. The white space in each box shows legitimate and faultless values while the black colour in the form of points, block and strip inside the respective box each box indicates the point, block and strip fault patterns.

Chen et al. [13] argued that ordinary random testing might generate test inputs lurking too close or too far from the input inducing failure and thus fails to discover the fault. To

generate more faults targeted test inputs they proposed Adaptive Random Testing (ART) as a modified version of ordinary random testing where test values are selected at random as usual but are evenly spread across the input domain by using two sets. The executed set comprises the test cases executed by the system and the candidate set includes the test cases to be executed by the system. Initially both the sets are empty. The first test case is selected at random from the candidate set and stored in executed set after execution. The second test case is then selected from the candidate set which is far away from the last executed test case. In this way the whole input domain is tested with greater chances of generating test input from the existing fault patterns.

In the experiments conducted by Chen et al. [13] the number of test cases required to detect first fault (F-measure) was used as a performance matrix instead of the traditional matrix P-measure and E-measure. Experimental results using ART showed up to 50% increase in performance compared to ordinary random testing. The authors admitted that the issues of increase overhead, spreading test cases across the input domain for complex objects and efficient ways of selecting candidate test cases still exist. Chen et al continued their work on ART to address some of these issues and proposed its upgraded version in [15] and [16].

2.3.2 Mirror Adaptive Random Testing

Mirror Adaptive Random Testing (MART) [14] is an improvement on ART by using mirror-partitioning technique to decrease the extra computation involved in ART and reduce the overhead.

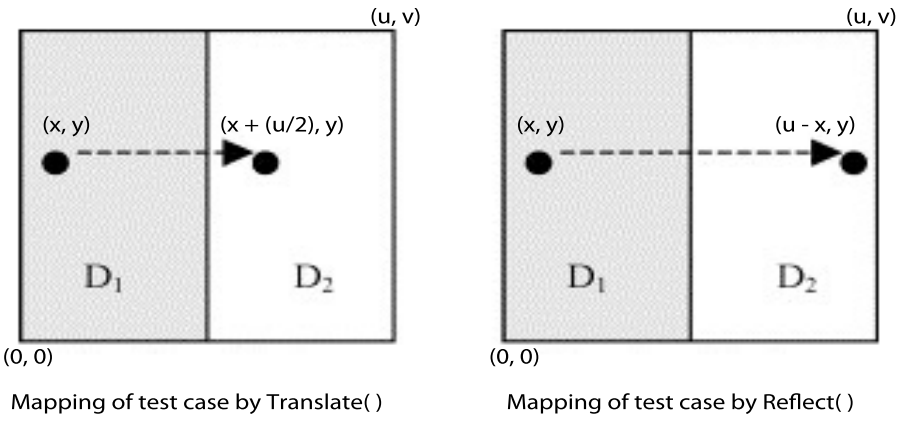

Figure 2.3: Mirror Adaptive Random Testing [14]

In this technique, the input domain of the program under test is divided into n disjoint sub-domains of equal size and shape. One of the sub-domains is called source sub-domain while all the others are termed as mirror sub-domains. ART is then applied only to the source sub-domain to select the test cases while from all other sub-domains test cases are selected by using mirror function. In MART (0, 0), (u, v) are used to represent the whole input domain where (0, 0) is the leftmost and (u, v) is the rightmost top corner of the two dimensional rectangle. On splitting it into two sub-domains we get (0, 0), (u/2, v) as source sub-domain and (u/2, 0), (u, v) as mirror sub-domain. Let suppose we get x and y test cases by applying ART to source sub-domain, now we can linearly translate these test cases to achieve the mirrored effect, i.e. (x + (u/2), y) as shown in the figure 2.3. Comparative study of MART with ART provide evidence of equally good results of the two strategies with MART having the added advantage of using only one quarter of the calculation as compared with ART and thereby reduces the overhead.

2.3.3 Restricted Random Testing

Restricted Random Testing [9] is another approach to overcome the problem of extra overhead in ART. Restricted Random Testing (RRT) achieves this by creating a circular exclusion zone around the executed test case. A candidate is randomly selected from the input domain for the next test case. Before execution the candidate is checked and discarded if it lies inside the exclusion zone. This process repeats until a candidate laying outside the exclusion zone is selected. This ensures that the test case to be executed is well apart from the last executed test case. The radius of exclusion zone is constant around each test case and the area of each zone decreases with successive cases.

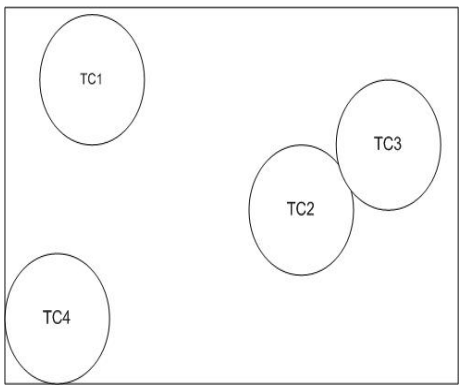

Figure 2.4: Input domain with exclusion zone around the selected test case

To find the effectiveness of RRT, the authors compared it with ART and RT on 7 out of the 12 programs evaluated by ART and MART. The experimental results showed that

the performance of RRT increases with the increase in the size of the exclusion zone and reaches the maximum level when the exclusion zone is raised to largest possible size. They further found that RRT is up to 55% more effective than ordinary random testing in terms of F-measure (Where F-measure is the total number of test cases required to find the first failure).

2.3.4 Directed Automated Random Testing

Godefroid et al., [34] proposed Directed Automated Random Testing (DART). The main features of DART can be divided into the following three parts:

1. Automated Interface Extraction: DART automatically identifies external interfaces of a given SUT. These interfaces include external variables, external functions and the user-specified main function, which initializes the program execution.

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

3. Dynamic Analysis of execution: The DART instruments the given SUT at the start of the process in order to track its behaviour dynamically at run time. The results obtained are analysed in real time to systematically direct the test case execution along alternative path for maximum code coverage.

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

2.3.5 Quasi Random Testing

Quasi-random testing (QRT) [16] is a testing technique which takes advantage of failure region contiguity by distributing test cases evenly with decreased computation. To achieve even spreading of test cases, QRT uses a class with a formula, that forms an s-dimensional cube in s-dimensional input domain and generates a set of numbers with small discrepancy and low dispersion. The set of numbers is then used to generate random test cases that are permuted to make them less clustered and more evenly distributed. An empirical study was conducted to compare the effectiveness of QRT with ART and RT. The empirical results of the experiments showed that in 9 out of 12 programs QRT found a fault quicker than ART and RT while there was no significant improvement in the remaining three programs.

2.3.6 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) [60] is a technique that generate unit test suite at random for object-oriented programs. As the name implies FDRT uses the feedback received from the execution of first batch of randomly selected unit test suite to generate next batch of more directed unit test suite. In this way redundant and illegal unit tests are eliminated incrementally from the test suite with the help of filtration and application of contracts. For example unit test that produce IllegalArgumentException on execution is discarded, because, selected argument used in this test was not according to the type of argument the method required.

2.3.6.1 Randoop: Feedback-directed Random Testing

The FDRT technique is implemented in Random tester for Object Oriented Programs (RANDOOP) tool [58]. RANDOOP is a fully automatic tool, capable of testing Java classes and .Net binaries which takes a set of classes (java or .Net executable), contracts, filters and time limit as input and gives output as a suite of JUnit and NUnit for Java and .Net programs respectively. Each unit test in a test suite is a sequence of method calls (hereafter referred as sequence). RANDOOP build the sequence incrementally by randomly selecting a public method from the class under test and arguments for these methods are selected from the predefined pool in case of primitive types and a sequence or null value in case of reference type. RANDOOP maintains two sets called ErrorSeqs and NonErrorSeqs to record the feedback. It extends ErrorSeqs set in case of contract or filter violation and NonErrorSeqs set when no violation is recorded in the feedback. The use of this dynamic feedback evaluation at runtime brings an object to very complex and interesting state. On test completion it produce ErrorSeqs and NonErrorSeqs as JUnit/NUnit test suite. In terms of coverage and number of faults discovered, RANDOOP implementing FDRT was compared with random testing of JCrasher and JavaPathFinder [70]. In the experiments 14 libraries of both Java and .Net were evaluated. The results showed that RANDOOP achieved more coverage than JCrasher in behavioural, branch coverage and faults detection. It can achieve on par coverage with systematic approaches like JavaPathFinder. RANDOOP also has an edge over model checking for its ability to easily search large input domains.

2.3.7 Object Distance and its application

To improve the performance of random testing the emphasis of ART was on the distance be- tween the test cases. But this distance was defined only for primitive data types like integers and other elementary input. Ciupa et al defined the parameters that can be used

to calculate distance between the composite programmer-defined types so that ART can be applicable to testing of todays object-oriented programs [18]. Two objects have more distance between them if they have more dissimilar properties. The parameters to specify the distance between the objects are dynamic types, values of its primitive and reference fields. Strings are treated as a directly usable values and Levenshtein distance [45] that is also known as edit distance is used as a distance criteria between the two strings. To implement object distance first all the distances of the objects are measured. Then two sets candidate- objects containing the all the objects ready to be run by the system and the used-objects set, which is initially empty. First object is selected randomly from the candidate-object set and is moved to used-object set when executed by the system. Now the second object selected from the candidate set for execution is the one with the biggest distance from the last executed object present in the used-object set. This process is continuing until the bug is found or the objects in the candidate-object set are finished.

2.3.7.1 ARTOO Tool

After the criteria to calculate the distance between the objects is defined [18], the same team implemented that model and performed several experiments to evaluate the proposed model. Adaptive Random Testing for Object Oriented (ARTOO) is a testing strategy, based on object distance, implemented in AutoTest tool [**?**]. ARTOO was implemented as a plug-in strategy in AutoTest. It only deals with creating and selecting inputs and all other functionality of the AutoTest was the same. Since ARTOO is based on object distance therefore the method for test input selection is to pick that object from the candidate set (A pool of objects that is a potential candidate to be executed by the system) that has the highest average distance in comparison to the objects already executed. In the experiments classes from EiffelBase library [**?**] were used. To evaluate ARTOO the same tests were also applied to directed random strategy (RAND). The outcome of the experiments showed that ARTOO finds the first bug with fewer test cases than RAND. The computation to select test case in ARTOO is more than RAND and therefore ARTOO takes more time to generate a test input. The experiments also found few of the bug found by ARTOO were not pointed out by RAND furthermore ARTOO is less sensitive to the variation of seed value than RAND.

2.3.7.2 Experimental Assessment of RT for Object-Oriented Software

In this research the effect of various parameters involved in random testing and its effect on efficiency is evaluated by performing various experiments on Industrial-grade code base. Large-scale clusters of computers were used for 1500 hours of CPU time which resulted

in 1875 test sessions for 8 classes under test. [19] The finding of the experiments are 1. Version of random testing algorithm that is efficient for smaller testing timeout is equally efficient for higher testing timeouts. 2. The value of seed for random testing algorithm plays a vital role in finding the number of bugs in specific time. 3. Most of the bugs are found in the first few minutes of the testing sessions.

2.4 Tools for Automated Random Testing

From the literature we can find a number of open source and commercial testing tools that automatically generate unit tests. Each tool utilize different generation technique but the one we are interested in is random technique. We present the most well known tools.

2.4.1 JCrasher

JCrasher is an automatic robustness testing tool developed by Csallner and Smaragadakis [58]. JCrasher tries to crash the Java program with random input and exceptions thrown during the process are recorded. The exceptions are then compared with the list of acceptable standards, defined in advance as heuristics. The undefined runtime exceptions are considered as errors. Since users interact with programs through its public methods with different kind of inputs, therefore, JCrasher is designed to test only the public methods of the SUT with random inputs.

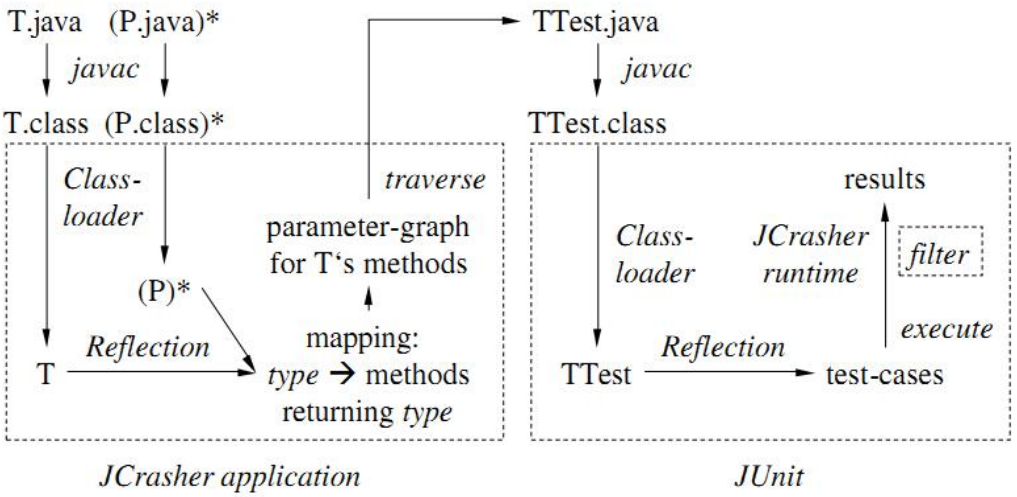

Figure 2.5: Illustration of robustness testing of Java program with JCrasher [58]

Figure 2.5 illustrates the working of JCrasher by testing a T.java program. The source file is first compiled using *javac* and the byte code obtained is passed as input to JCrasher.

The JCrasher, with the help of Java reflection [10], analyse all the methods declared by class T using methods transitive parameter types P to generate the most appropriate test data set which is written to a file *TTest.java*. The file is compiled and executed by JUnit. All the exceptions produced during test case executions are collected and compared with robustness heuristic for any violation which is reported as error.

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

2.4.2 Jartege

Jartege (Jawa random test generator) [52] is an automated testing tool that randomly generates unit tests for Java classes with contracts specified in Java Modelling Language (JML). The contracts include methods pre- and post-conditions and class invariants. Initially Jartege uses the contracts to eliminate irrelevant test cases and later on the same contracts serve as test oracle to differentiate between errors and false positives. Jartege uses simple random testing to test classes and generate test cases. In addition, it parametrise its random aspect in order to prioritise testing specific part of the class or to get interesting sequences of calls. The parameters include the following:

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

The Jartege technique evaluates a class by entry pre-condition and internal pre-conditions. Entry pre-conditions are the contracts to be met by the generated test data for testing the method while internal pre-conditions are the contracts which are inside the methods and their violation are considered as error either in the method or in the specification. The Jartege checks for errors in program code as well as in specifications and the Junit tests

produced by Jartege can be used later as regression tests. Its limitation is the requirement of prior existence of the program JML specifications.

2.4.3 Eclat

Eclat [57] is an automated testing tool which generates and classifies unit tests for Java classes. The process is accomplished in three main components. In the first component, it selects a small subset of test inputs, likely to reveal faults in the given SUT, from a large set.

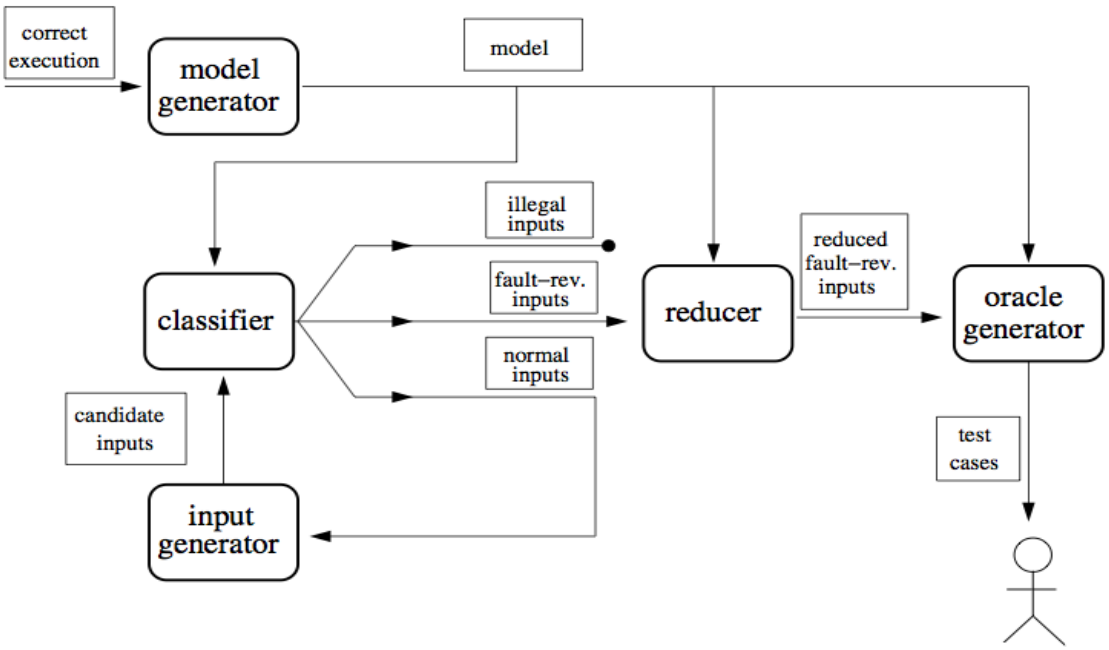

Figure 2.6: Main component of Eclat contributing to generate test input [57]

The tool takes a software and a set of test cases for which the software runs properly. It then creates an operational model based on the correct software operations and apply the test data. If the operational pattern of execution of the test data differs from the model, the following three outcomes may be possible: (a) it sight a fault in the given SUT, (b) it might produce normal operations despite model violation, (c) it might be an illegal input that the program cannot handle. (note for author extend 2nd and 3rd component) In the second component of the process, reducer function is used to discard any redundant input, leaving only a single input per operational pattern. In third component the acquired test inputs are converted into test cases, by creation of oracle, to determine the success or failure of the test.

Eclat was compared with JCrasher by executing nine programs on both tools individually [58]. Based on the experimental results it was revealed that Eclat perform better than JCrasher. On the average, Eclat selected 5.0 inputs per run out of which 30% revealed faults, while JCrasher selected 1.13 inputs per run out of which 0.92% of those revealed faults. The limitation of Eclat is dependence on the initial pool of correct test cases and existence of any errors in the pool may lead to the creation of wrong operational model which will adversely affect the testing process.

2.4.4 QuickCheck

QuickCheck [22] is a lightweight random testing tool used for testing of Haskell programs [40]. Haskell is a functional programming language where programs are evaluated by using expressions rather than statements as in imperative programming. In Haskell most of the functions are pure except the IO functions, thus main focus of the tool is on testing pure functions. These are the functions which depend on its input parameters and make changes to them only. QuickCheck tool is designed to have a simple domain-specific language of testable specifications embedded in Haskell. This language is used to define expected properties of the functions under test - for example, reversing a list with single element must result in the same list.

The QuickCheck takes function to be tested and properties of the program defined by tester (Haskell functions) as input. The tool uses built-in random generator to generate effective test data, however, to get adequate coverage in the case of custom data types, the testers can also develop their own generator. On executing the function with test data, the tester defined properties must hold for the function to be correct. Any violation of the defined properties suggest error in the function.

2.4.5 Autotost

The Autotest tool, based on Formal Automated testing is used to test Eiffel language programs [19]. The Eiffel language uses the concept of contracts which is effectively utilized by Autotest - for example the auto generated input is filtered using pre-conditions and unwanted test input is discarded. The contracts are also used as test oracle to determine if the test is pass or fail. Beside automated testing the Autotest also allow the tester to manually write the test cases to target specific behaviour or section of the code. The Autotest takes a single method/class or cluster of methods/classes as inputs, it then automatically generate test input data according to the requirement of the methods or classes.

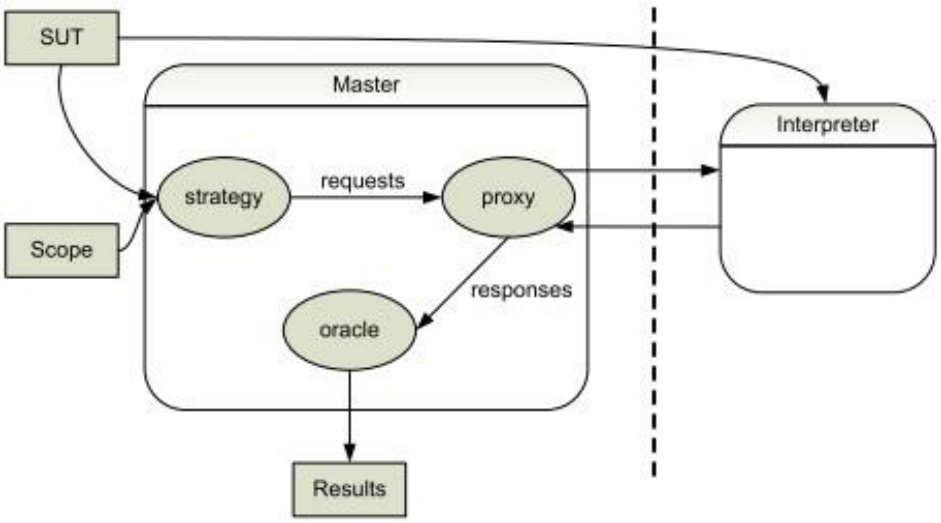

Figure 2.7: Architecture of Autotest

According to Figure 2.7 [43], the architecture of Autotest tool can be split into the following main parts:

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

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

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

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

2.4.6 TestEra

TestEra [41] is a novel framework for testing Java applications. All the tests are produced and executed in an automated fashion. Tests are conducted on the basis of

the method specifications [11]. TestEra takes methods specifications, integer value as a limit to the generated test cases and the method under test. It uses pre-conditions of a method from specifications to automatically generate test cases up to the specified limit. These test cases are then executed on the method and the result is compared against the postconditions (oracle) of that method. Any test case that fails to satisfy postcondition is considered as a fault. The complete error log is displayed in the Graphical User Inteface (GUI).

2.4.7 Korat

Korat [6] is a novel framework for automated testing of Java programs based on their formal specifications [12]. As the test start, it uses methods pre-condition to generate all non-isomorphic test cases up to a given size. It then executes each of the test case and compare the obtained results to the methods post-condition, which serves as an oracle to evaluate the correctness of each test case. Korat uses a black-box testing approach where it uses methods pre-conditions as predicates. After the Java predicates and finitization (that bounds the predicates input space) are defined, Korat systematically explore the predicates input space and generate 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 the inputs based on the fields accessed during executions.

2.5 YETI

York Extensible Testing Infrastructure (YETI) is an automated tool for testing Java, JML and .NET assemblies [53]. YETI execute the program under test with random generated but type-correct inputs and declare a fault if the response is an unexpected exception or a contract violation. YETI has been designed with an emphasis on extensibility. Its three main parts: the core infrastructure, strategies and language bindings are loosely coupled to easily accommodate new languages and strategies. To keep the process fully automated YETI uses two approaches for oracle (pass/fail judgement). If available, YETI uses code contracts as oracle if not it uses undeclared runtime exceptions of the underlying language as oracle. The test cases revealing errors are reproduced at the end of each test session for unit and regression testing. Other prominent features of YETI include its Graphical User Interface (GUI) for

user friendliness and ability to distribute large testing tasks in cloud for parallel execution [54]. The following sections briefly describe internal working and execution of YETI tool.

2.5.1 Construction of Test Cases

YETI construct test cases at random by creating objects of the class under test and randomly calling its methods with inputs according to its parameter's-space. Strategy section contains seven different strategies and inputs to the tested methods is defined by one of the selected strategy. To completely automate the data generation YETI split input values into two types i.e. primitive data types and user defined classes. For Java primitive data types, which includes short, byte, char, int, float, double, long etc., YETI uses its own built-in random value generation library. However, in the case of user defined classes where objects data type is a user defined class YETI calls its constructor to generate object of that class at run time. It may be possible that the constructor require another object and in this case YETI will recursively calls the constructor of that object. This process is continued until the an object with blank constructor, constructor with only primitive types (type 1) or the set level of recursion is reached.

2.5.2 Command-line Options

While YETI GUI launcher has been developed during this research study, to take maximum benefit of the available options one still need to launch YETI from CLI mode. These command-line options are case insensitive and can be provided as input to the tool in CLI mode. For example, to save processing power command line option -nologs can be used to bypass real-time logging. The following table describes few of the most common command-line options available in YETI.

2.5.3 YETI Execution

YETI being developed in Java is highly portable and can easily run on any operating system with Java Virtual Machine (JVM). YETI can be executed from both command line and GUI. To build and execute YETI, it is necessary to specify the project and all the .jar library files particularly javassist.jar in the CLASSPATH or JVM would not be able to find and execute it. There are several options available as discussed

Table 2.2: YETI command line options

Levels	Purpose
-java	Test programs coded in Java
-jml	Test programs coded in JML
-dotnet	Test programs coded in .NET
-ea	To check code assertions
-nTests	Specify number of tests after which the test stops
-time	Specify time in seconds or minutes after which the test stops
-testModules	Specify one or more modules to test
-rawlogs	Prints real time logs during test
-nologs	Omit real time logs and print end result only
-yetiPath	Specify path to the test modules
-gui	Show test session in GUI
-DSSR	Specify Dirt Spot Sweeping Random strategy for this session
-ADFD	Specify Automated Discovery of Failure Domain strategy for this session
-random	Specify random test strategy for this session
-randomPlus	Specify random plus test strategy for this session
-randomPlusPeriodic	Specify random plus periodic test strategy for this session
-nullProbability	Specify probability of inserting null as input value
-newInstanceProability	Specify probability of inserting new object as input value

in section xxx??? to accommodate specific needs but the typical command to invoke YETI is given in figure **??**. In this particular command YETI tests java.lang.String and yeti.test.YetiTest modules, for details of other options please see section xxx????.

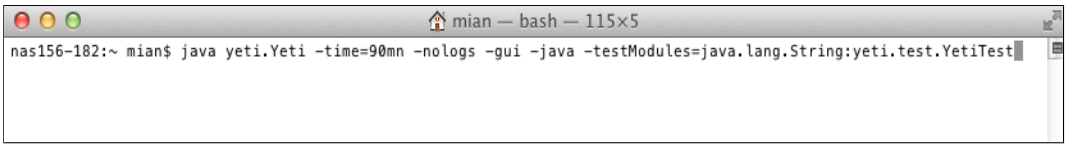

Figure 2.8: Command to launch YETI from CLI

Alternately, runnable jar file by the name YetiLauncher is also available to launch YETI from GUI. However, till the writing of this thesis, the GUI version of YETI only supports the basic options of YETI. Figure xxx??? shows the equivalent of above command in GUI mode.

As a result of both the above commands YETI launch its own GUI window and start testing the assigned programs.

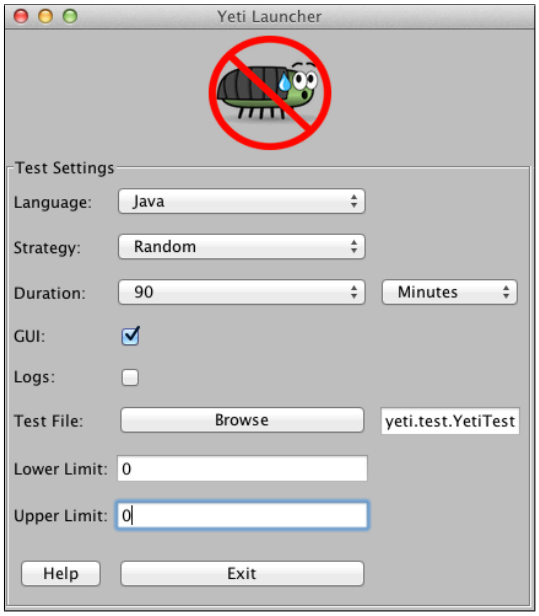

Figure 2.9: Command to launch YETI from GUI

2.5.4 YETI Report

2.5.5 Summary of automated testing tools

2.6 Conclusion

Table 2.3: Summary of automated testing tools

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

References

[1] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.

[2] 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.

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

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

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

[6] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM.

[7] 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.

[8] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, ECSQ '02, pages 321–330, London, UK, UK, 2002. Springer-Verlag.

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

[10] 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.

[11] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. *SIGSOFT Softw. Eng. Notes*, 24(6):285–302, 1999.

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

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

[14] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng. Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC '03, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

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

[16] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 309–312, New York, NY, USA, 2005. ACM.

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

[18] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 55–63, New York, NY, USA, 2006. ACM.

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

[20] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[21] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 157–166, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

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

[24] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437–444, 1997.

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

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

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

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

[29] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

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

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

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

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

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

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

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

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

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

[39] 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.

[40] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[41] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-Based testing of java programs using SAT. *Automated Software Engineering*, 11:403–434, 2004. 10.1023/B:AUSE.0000038938.10589.b9.

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

[43] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 261a–, Washington, DC, USA, 2007. IEEE Computer Society.

[44] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[57] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, pages 504–527, 2005.

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

[59] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[60] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

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

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

[63] 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.

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

[65] Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th international conference on Software engineering*, pages 105–118. ACM, 1992.

[66] 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.

[67] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*, pages 285–288. IEEE, 1998.

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

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

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

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

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

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

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

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

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

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