Chapter 1

Introduction

In this chapter we give a brief introduction and motivation for the research work presented in this thesis. After brief motivation, we commence by introducing the problems in random testing. We then describe the alternative approaches to overcome these problems, followed by our research goals and contributions. At the end of the chapter, we give the structure of the thesis.

1.1 Motivation

Software is a very important component of computer system, essentially required to accomplish a given task. Some softwares are developed for use in simple day to day operations whiles others are for highly complex processes in specialized fields like research and education, business and finance, defense and security, health and medicine, science and technology, aeronautics and astronautics, commerce and industry, information and communication, environment and safety etc. The margin of error in mission-critical and safety-critical systems is so small that a minor fault can lead to huge economic losses to the organization [45]. Therefore, software companies leave no stone unturned to ensure the reliability and accuracy of the software. 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 software testing techniques.

1.2 The Problem

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 [57], [44]. Generating test data set manually is a time-consuming and laborious exercise [49]. 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 [89]. 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 [25]. According to --------, “Random testing is a simple and well-known technique which can be remarkably effective in discovering software bugs” [DART].

Despite the benefits of random testing, its simplistic and non-systematic nature exposes it to high criticism [88]. Myers & Sandler [59] mentioned it as, “Probably the poorest methodology of all random-input testing...”. However, -------- consider that the above stated statement is based on intuition and lacks any experimental evidence [---]. The criticism motivated the research community to look into various aspects of random testing for evaluation and possible improvement. Adaptive Random Testing (ART) [13], Restricted Random Testing (RRT) [10], Feed-back Directed Random Testing (FDRT) [75], Mirror Adaptive Random Testing (MART) [14] and Quasi Random Testing (QRT) [18] are a few of the enhanced random testing techniques.

Random testing is also considered weak in providing high code coverage [63], [30]. For example, in random testing when the conditional statement *“if (x == 25) then …”* is exposed to execution then there is only one chance, of the “*then...*” part of the statement, to be executed out of 232. If x is an integer variable of 32 bit value [40].

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

1.3 Research Goals

The goals of the research study are to develop new techniques for automated testing, based on random strategy, with the aim to achieve the following objectives:

1. To develop a new testing strategy with the capability to generate more faultfinding test data.
2. To develop a new 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 faultfinding test data.

1.4 Contributions

The main contributions of the thesis research are stated below:

1.4.1 Dirt Spot Sweeping Random Strategy

The faultfinding ability of the 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 neighboring values for the subsequent tests. Resultantly, selected values sweep around the failure, leading to the discovery of new failures in the vicinity. Results indicated higher faultfinding ability of DSSR strategy as compared with R and R+ strategies.

1.4.2 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 study includes description of the ADFD strategy, its implementation in YETI. The strategy is practically illustrated by executing several programs of one and two dimensions. The experimental results proved that ADFD strategy automatically performed identification of faults and fault domains along with graphical representation in the form of chart.

1.4.3 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 indicated improved features of IGRS 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.

1.5 Structure of the Thesis

The rest of the thesis is organized as follows: In Chapter 2, a thorough review of the relevant literature is given. It includes a brief introduction followed types of software testing followed by and the techniques followed by automated random testing tools. Chapter 3 describes Dirt Spot Sweeping Random (DSSR) strategy, which is based on sweeping of fault clusters in the input domain. Chapter 4 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. Chapter 5 presents the new strategy developed with the focus on quick identification of faults and increase in code coverage with the help of assertions; Finally Chapter 7 summarizes contributions of the thesis research, discusses the strength and weaknesses of the study, gives conclusion and suggests avenues for future work.

Chapter 2:

Literature Review

Paul Ehrlich famous quote is, “To err is human, but to really foul things up you need a computer”. Since the programmers are ordinary human beings, it is most obvious that some errors remain in the software after its completion. Errors are not tolerated as they can cause great loss. According to the National Institute of Standard and Technology 2002, 10 report, software errors cost an estimated $59.5 billion loss to US economy annually. The destruction of the Mariner 1 rocket (1962) that cost $18.5 million was due to a simple formula coded incorrectly by a programmer. The Hartford Coliseum Collapse (1978) costing $70 million, Wall Street crash (1987) costing $500 billion, Failing of long division by Pentium (1993) costing $475 million, Ariane 5 Rocket disaster costing $500 million and many others are caused by minor errors in the software. To achieve high quality, the software has to satisfy rigorous stages of testing. The more complex and critical the software, the higher the requirements for software testing and the larger the damage caused if the bug remains 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 [42], where faults are defined as the errors made by people during software development [2].

Being an integral part of Software Development Life Cycle (SDLC), the testing process is started from requirements phase since this is the starting point of all the software activities and continues throughout the life of the software. In traditional testing when testers finds a fault in the given SUT, the software is given back to the developers for removing the fault and after its rectification the software is handed back to the testers for retesting. It is important to understand the fact that “program testing can be used to show the presence of bugs, but never to show the absence of bugs” [25]. Which means SUT that passes all the tests without giving a single error is not guaranteed to contain no error. The testing process increases however the reliability and confidence of the users in the tested product.

2.1.1 Software Testing Levels

Unit testing, integration testing and system testing [16] are the three main levels of software testing defined in the literature. Unit testing evaluate a small piece of software code called units for faults. These units are combined together to form components and integration testing ensures that the integration points are working properly. Finally the components are combined to form a system and before production system testing is performed to make sure that it works as expected.

2.1.2 Software Testing Purpose

The primary purpose of software testing is identification of faults in the given SUT so that they can be corrected to achieve high quality. Ideally, maximum number of faults can be identified if software is tested exhaustively i.e. testing SUT against all possible combinations of input data, and comparing the obtained results to the expected results for assessment. However, exhaustive testing is not always possible in most of the test 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 a specific aspect of a SUT. For example, functionality testing is performed to check if one or more functions of a system are working correct or not. Structural testing analyze the code structure to generate test cases in order to evaluate paths of execution and identification of unreachable or dead code. In robustness testing the software behavior is observed in the case when it receive input that is outside of its expected input range. Stress and performance testing aim to test the response of software under high load and its ability to process different nature of tasks [24]. Finally, compatibility testing is performed to see the interaction of software with underlying operating system or hardware.

2.1.3 Software Testing Perspective

Testing activities can be split up into black-box and white-box testing on the basis of perspective taken. In black-box or functional testing the testers dont need to know about internal code structure of the SUT. Test cases are derived from the specifications and test passes if the output is according to expected output. Internal code structure of the SUT is not taken into any consideration [4]. Whereas in white-box or structural testing testers must know about the complete structure of the software and can modify it, if 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 [48].

2.1.4 Software Testing Execution

Test activities can be organized into static and dynamic testing on the basis test cases execution. In static testing test cases analyzed statically checked for errors without any execution. All high quality softwares are accompanied by documentation in addition to software code. These include requirements, design, technical, end-user and marketing documentation. Re- views, 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 through processing. Results are analyzed against expected results to find any error in the software. Unit testing, integration testing, system testing, and acceptance testing are most commonly used as dynamic testing methods [27]

2.1.5 Manual Testing

A software testing technique to find faults in a class or group of related classes, such that the tester must write the code by hand to create test cases and test oracle [21]. While manual testing is effective in some cases, in general, it is a laborious, time consuming, error-prone [58]. It further requires testers to have appropriate skills, experience and in depth knowledge of the under test software in order to evaluate it from different perspectives.

2.1.6 Automated Testing

A software testing technique to find faults in a class or group of related classes, such that the test cases and test oracle is generated automatically by a testing tool [37]. The tools can automate part of a test i.e. generation of test cases, execution of test cases and evaluation of results or the whole test process. The use of automated testing made it possible to test large volumes of code that would be otherwise impossible [54].

2.1.7 Test Oracle

Test oracles set the acceptable behavior for test executions [55]. All softwares testing techniques depend on the availability of a test oracle [3]. Designing test oracles for simple softwares may be straightforward, however, for relatively complex softwares it can be very cumbersome to decide whether a program execution returns a correct or incorrect result [28]. Different testing techniques tackle the oracle problem in various ways but some of the common issues include:

1. It is assumed that execution results are observable, so that they can be evaluated against the test oracle or the oracles are defined on the basis of these results.
2. An ideal test oracle would satisfy desirable properties of program specifications [3].
3. There is not a single oracle generation technique that satisfies all purposes. Weyuker [60]  argued that truly general test oracles are often unobtainable.

2.1.7.1 Random Testing

Random testing is a dynamic and black-box testing technique in which the software is tested with non-correlating or unpredictable test data from the specified input domain [7]. The input domain is a set of all possible inputs to the software under test. According to Richard H. [31], to conduct random testing, an input domain is defined, then test points are randomly taken from the whole input domain through a random number/test case generator. The program under test is executed on these points and the results obtained are compared to the program specifications. The test fails if any input leads to incorrect results or otherwise it is successful.

It is quick and cheap to generate random test data as it don’t require too much intellectual and computational efforts [17]. This capability makes it an ideal choice for implementation in automated testing tools [20]. In addition, no human intervention in data generation/selection makes it one of the most unbiased testing techniques.

Generating test cases with out using any background information makes it highly susceptible to criticism. Myers [41] intuitively mentioned random testing as one of the least effective testing technique. It is also criticized for generating many sets of tests that lead to the same state of the software. Furthermore, random testing can generate test inputs that violates requirements of the given SUT making it less effective [56], [49].

Myers statement was not based on any experimental evidence and later on the experiments performed in [31], [21], [38] and [26] confirmed that random testing is as effective as any other systematic testing technique. The experiments in [26] found that random testing can find subtle faults in a given SUT if run for large number of test cases. They argued that the simplicity and cost effectiveness of random testing can make it feasible to run large number of test cases as opposed to systematic testing which requires considerable time and resources for test case generation and execution. The empirical comparison [30] also proves that random testing and partition testing are equally effective. Furthermore the study conducted by Ntafos [43] concludes the effectiveness of random testing over proportional partition testing.

2.2 Variations in Random Testing

Different researchers tried various strategies to improve the performance of random testing. In order to better understand the topic we have studied each strategy in detail.

2.2.1 Adaptive Random Testing

Adaptive random testing (ART) [12] is based on the existence of failure patterns across the input domain detected by Chan et al [6]. They observed that failure inducing inputs in the whole input domain form certain geometrical patterns. They divided these patterns into point, block and strip fault patterns. Each one is described below.

In the figure 2.2 the square box indicates the whole input domain. The white space shows legitimate or faultless values while the black color points, block and strip inside each box indicate the point, block and strip fault patterns in the input domain.

1. Point pattern: In the point pattern failure inducing inputs 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 multiple failure inducing inputs lies in close vicinity to form a block in the input domain. Example of block pattern is failure caused by a statement if ( (num >10) && (num <20) ). Here 11 to 19 is a block of faults.
3. Strip pattern: In the strip pattern the failure inducing inputs 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.

The authors argued that ordinary random testing might generate test inputs lurking too close or too far from the fault inducing input and thus failing to discover it. To generate more faults targeted test inputs they suggested ART. ART is a modified version of ordinary random testing where test values are selected at random like before but evenly spread across the input domain. To achieve an even distribution of test cases across the input domain they used two sets. The executed set having the test cases that have been executed by the system and the candidate set that contain the random selected test cases from the bounded input domain as candidates for execution. Initially both the sets are kept 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 based on the criteria that it is far away from the last executed test case. Thus the whole input domain can be tested and their are more chances of generating test input from inside of the existing geometrical patterns.

In the experiments they used number of test cases required to detect first failure (F-measure) as a performance matrix instead of the traditional matrix i.e. probability of detecting at least one failure (P-measure) and expected number of failures detected (E-measure). Results of the experiments performed on published programs using ART showed up to 50% increase in the performance of than ordinary random testing. Results showed significant improvement, however, 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 evolve their work on ART to address some of these issues in [14] and [15].

2.2.2 Mirror Adaptive Random Testing

As discussed in the above section ART provide better results, however the increase in overhead due to extra computation to achieve even spread of test inputs makes it less cost effective. Mirror Adaptive Random Testing (MART) [13] is an innovative approach that uses mirror-partitioning technique to reduce the overhead of ART by decreasing the extra computation involved in ART.

In this technique, the input domain of the program under test is divided into n disjoint subdomains of equal size and shape. One of the subdomains is called source subdomain while all the others are termed as mirror subdomains. ART is then applied only to the source subdomain to select the test cases and from all other subdomains 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) are the leftmost and (u, v) are the rightmost top corner of the two dimensional rectangle. On splitting it into two subdomains we get {(0, 0), (u/2, v)} as source subdomain and {(u/2, 0), (u, v)} as mirror subdomain. Let suppose we get x and y test cases by applying ART to source subdomain, 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. Experimental results showed that the performance of MART is equal to ART with MART using only one quarter of the calculations of that of ART.

2.2.3 Restricted Random Testing

Restricted Random Testing (RRT) [8] is another approach, with small overhead in contrast to ART, to spread the the test cases more evenly across the input domain. 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 is 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 one. The radius of exclusion zone is constant around each test case and the area of each zone decreases with successive cases.

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 to maximum when the exclusion zone is raised to largest possible size. Normalized Restricted Random Testing [8] is an improvement over RRT by allowing the testers to have better information about the target exclusion rate (R) of RRT. 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).

2.2.4 Directed Automated Random Testing

Directed Automated Random Testing (DART) is a random testing technique proposed by Godefroid et al., [29]. Its main purpose was to overcome the cost and difficulty of manual testing while keeping its quality intact. It automates the whole testing process including generation of unit tests, test drivers/harness and assertions for functional correctness. The main functions of DART can be divided into 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/Harness: After identification of all the external interfaces of a given SUT, DART generate test drivers/harness 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 behavior dynamically at run time. The results obtained are analyzed in real time to systematically direct the test case execution along alternative path for maximum code coverage.

Directed Automated Random Testing algorithm is implemented in DART tool. It is a completely automatic tool and all it needs is a 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.

2.2.5 Quasi Random Testing

Quasi-random testing (QRT) [15] is a testing technique that takes advantage of failure region contiguity by distributing test cases evenly similar to ART but with decreased computation. Chan et al after the analysis of faults in various experiments found that the fault patterns across the input domain are continuous. 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 generate sequence of numbers that have small discrepancy and low dispersion. These sequence of numbers are then used to generate random test cases that are permuted to make them less clustered and more even than RT. An empirical study was conducted to compare the effectiveness of QRT with ART and RT. The 12 numerical programs picked for experiments were the same used to evaluate ART. The empirical results of the experiments showed that in 9 out of 12 programs QRT on average finds a fault quickly than ART and RT while in the remaining three programs the improvement is insignificant.

2.2.6 Feedback-directed Random Testing

Feedback-directed Random Testing (FDRT) [53] 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 approach 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, randomly selected argument used in this test was not according to the type of argument the method required.

2.2.6.1 Randoop: Feedback-directed Random Testing

The FDRT technique is implemented in RANDOOP tool [51]. RANdom tester for Object Oriented Programs (RANDOOP) is a fully automatic tool, capable of testing Java classes and .Net binaries. RANDOOP takes as input a set of classes (java or .Net executable), contracts, filters and the time limit after which the testing process stops. Its output is 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 if 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. To find the effectiveness of the strategy, in terms of coverage and number of faults discovered, the authors compared RANDOOP implementing FDRT with random testing of JCrasher and JavaPathFinder [59]. 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.

2.2.7 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 today’s 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 [39] 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.

2.2.7.1 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 [16]. 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 [17] 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.

2.2.7.2 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.

2.2.8 JCrasher

JCrasher is first of the three automatic testing tools developed by Csallner C. and Smaragadakis Y. [51]. As the name suggests JCrasher tries to crash the Java program with random input and any exceptions thrown during the process are recorded. All exceptions are then compared to the list of acceptable exception, which are defined in advance as heuristics; any undefined/un- declared runtime exceptions are considered errors. Since programs interact with the world through its public methods and they are also exposed to different kind of inputs, therefore, JCrasher tests only these methods with random inputs.

Figure 2.6 illustrate the working of JCrasher by testing a java program namely T.java. The source file is first compiled using javac and the obtained byte code is passed into JCrasher. The JCrasher using Java reflection [9] analyze all the methods declared by class T and by their transitive parameter types P to generate the most appropriate test data set. The test data set is written to a file TTest.java that is compiled and executed by JUnit. All the exceptions produced during test case executions are collected and compared with robustness heuristic to check for violation. Any violated test case is reported as error.

JCrasher is a pioneer to perform fully automatic testing from test case generation, execution, and filtration to reporting the results. One of its novelties is that it generates test case as JUnit files that can be easily read and can be 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.

2.2.9 Jartege

Jartege (Jawa random test generator) [45] 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, however, it facilitate to parameterise its random aspect in order to prioritise testing specific part of the class or to get interesting sequences of calls. These includes:

1. To define the operational profile of the classes i.e. the likely use of the class under test by other classes.
2. To define the weight of each class and method under test and give test priority to the one’s with highest weight and skip those with null weight.
3. To control the creation of newly created objects with creation probability functions. Low probability means creation of fewer objects and more reusability for different operations while high probability means numerous new objects with less reusability.  The Jartege technique evaluate class by entry pre-condition and internal pre-condition. Entry pre-conditions are the contracts which must be met by the generated test data to test the method while internal pre-conditions are the ones which are inside the methods and their violation are considered error either in method or in the specification. The benefits of Jartege is that it checks for errors in both program and specifications and the Junit tests produced by Jartege can be used later as regression tests. Its minor shortcoming is that the SUT JML specifications must exist or may be written manually in order to be tested by Jartege.  2.2.10 Eclat  Eclat [50] testing tool automatically generates and classify unit tests for Java classes. The process can be divided into three main components. In the first component, it selects small subset of test inputs, likely to reveal faults in the given SUT, from a large set of test inputs.

The tool takes 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 applies the test data to it. As a result any inputs whose operational pattern of execution differs from the operational model are (1) likely to reveal fault in the given SUT, (2) Likely to produce normal operations despite violating the model, (3) illegal input that the program is not required to handle. In the second component, reducer function is used to discard any redundant input, leaving only one input per operational pattern. The third and final component facilitates automated testing by converting the acquired test inputs into test cases and creation of oracle to determine whether the test succeeds or fail.

To measure the effectiveness, the researchers tested 9 programs on both Eclat and JCrasher [51]. The experimental results revealed that Eclat outperformed JCrasher. On average, Eclat selected 5.0 inputs per run, and 30% of those revealed a fault. While JCrasher selected 1.13 inputs per run, and 0.92% of those revealed a fault. The shortcoming of Eclat is its dependence on the initial pool of correct test cases, which is usually written manually, and existence of any errors in it can propagate and affect the whole testing process.