New Strategies for Automated Random Testing

Mian Asbat Ahmad

Department of Computer Science

The University of York

A thesis submitted for the degree of

Doctor of Philosophy

September 3, 2013

Abstract

This is where you write your abstract ...

Contents

List of Figures

List of Tables

.

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.

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.

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 everywhere. In our world today, software flies spacecraft, monitors power plants, manages stock exchange, assist surgeries, drives cars and design graphics. The margin for error in these mission-critical and safety-critical systems is so small that a minor fault can incur huge cost to the economy and miseries to the mankind [33]. Therefore, software development companies leave no stone unturned to ensure the reliability and accuracy of the software. This dissertation is a step further towards the reduction in overall cost of software testing by devising new improved and highly effective software testing techniques.

1.2 The Problems

In software testing, one is often confronted with the problem of selecting a test data set, from a large or often infinite domain, as exhaustive testing is not always applicable. Test data set is a subset of domain carefully selected to test the given software. Finding an adequate test data set is a crucial process in any testing technique as it aims to represent the whole domain and evaluate the given system under test (SUT) for structural or functional properties [40], [32]. Manual test data set generation is a time-consuming and laborious exercise [36]; therefore, automated test data set generation is always preferred. Test data generators are classified in to Pathwise, Goal-Oriented, Intelligent and Random [62]. Random test data generation generates test data set randomly from the whole domain. Unlike other approaches Random approach is simple, widely applicable, easy to implement in an automatic testing tool, fastest in computation, no overhead in choosing inputs and free from bias [19].

Despite the benefits random testing offers, its simplistic and non-systematic nature expose it to high criticism [61]. Myers & Sandler [42] mentioned it as "Probably the poorest methodology of all is random-input testing...". Where this statement is based on intuition and lacks any experimental evidence, it motivated the interest of research community to evaluate and improve random testing. Adaptive random testing [12], Restricted Random Testing [7], Feedback directed Random Testing [52], Mirror Adaptive Random Testing [13] and Quasi Random Testing [15] are few of the enhanced random testing techniques aiming to increase its fault finding ability.

Random testing is also considered weak in providing high code coverage [44], [23]. 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 [29].

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 display only the fault revealing test cases in the form of unit tests. However efforts are required to show the test results in more compact and user-friendly way.

1.3 Research Goals

The overall goal of this thesis is to develop new techniques for automated testing based on random strategy that addresses the above-mentioned problems. Particularly,

1. We aim to develop an automated random testing technique that is able to generate more fault-revealing test data. To achieve this we exploit the presence of fault clusters found in the form of block and strip fault domains inside the input domain of a given SUT. Thus we are able to find equal number of faults in fewer numbers of test cases than other random strategies.

2. We aim to develop a novel framework for finding the faults, their domains and the presentation of obtained results on a graphical chart inside the specified lower and upper bound. It considers the correlations of the fault and fault domain. It also gives a simplified and user-friendly report to easily identify the faulty regions across the whole domain.

3. We aim to develop another automated testing technique which focuses on increase in code coverage and generation of more fault-revealing data. To achieve this we utilises Daikon— an automated invariant detector that reports likely program invariant. An invariant is a property that holds at certain point or points in a program. With these invariants in hand we can restrict the random strategy to generate values around these critical points. Thus we are able to increase the code coverage and quick identification of faults.

1.4 Contributions

To achieve the research goals described in Section 1.3, we make the following specific contributions:

1.4.1 Dirt Spot Sweeping Random Strategy

Random testing is a simple and effective technique to find failures in complex programs. However, its efficiency reduces when the failures lie in contiguous locations across the input domain. To overcome the deficiency, we developed a new automated technique: Dirt Spot Sweeping Random (DSSR) strategy. 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. To evaluate the effectiveness of DSSR strategy a total of 60 classes (35,785 lines of code), each class with 30 x $\mathbf{10^5}$ calls, were tested by Random (R), Random+ (R+) and DSSR strategies. T-Test analysis showed significantly better performance of DSSR compared to R strategy in 17 classes and R+ strategy in 9 classes. In the remaining classes all the three strategies performed equally well. Numerically, the DSSR strategy found 43 and 12 more unique failures than R and R+ strategies respectively. This study comprehends that DSSR strategy will have a profound positive impact on the failure-finding ability of R and R+ testing.

1.4.2 Automated Discovery of Failure Domain

There are several automated random strategies of software testing based on the presence of point, block and strip fault domains inside the whole input domain. As yet no particular, fully automated test strategy has been developed for the discovery and evaluation of the fault domains. We therefore have developed Automated Discovery of Failure Domain, a new random test strategy that finds the faults and the fault domains in a given system under test. It further provides visualisation of the identified pass and fail domain. In this paper we describe ADFD strategy, its implementation in YETI and illustrate its working with the help of an example. We report on experiments in which we tested error seeded one and two-dimensional numerical programs. Our experimental results show that for each SUT, ADFD strategy successfully performs identification of faults, fault domains and their representation on graphical chart.

1.4.3 Invariant Guided Random+ Strategy

Acknowledgement of random testing being simple in implementation, quick in test case generation and free from any bias, motivated research community to do more for increase in performance, particularly, in code coverage and fault-finding ability. One such effort is Random+ — Ordinary random testing technique with addition of interesting values (border values) of high preference. We took a step further and developed Invariant Guided Random+ Strategy (IGRS). IGRS is an extended form of Random+ strategy guided by software invariants. Invariants from the given software under test are collected by Daikon— an automated invariant detector that reports likely invariant, prior to testing and added to the list of interesting values with high preference. The strategy generates more values around these critical program values. Experimental result shows that IGRS not only increase the code coverage but also find some subtle errors that pure Random and Random+ were either unable or may take a long time to find.

1.5 Structure of the Thesis

The rest of the thesis is organised as follows: In Chapter 2, we give a thorough review of the relevant literature. We commence by discussing a brief introduction of software testing and shed light on various techniques and types of software testing. Then, we extend our attention to automated random testing and the testing tools using random technique to test softwares. In Chapter 3, we present our first automated random strategy Dirt Spot Sweeping Random (DSSR) strategy based on sweeping faults from the clusters in the input domain. Chapter 4 describes our second automated random strategy which focus on dynamically finding the fault with their domains and its graphical representation. Chapter 5 presents the third strategy that focus on quick identification of faults and increase in coverage with the help of literals; Finally, in Chapter 7, we summarise the contributions of this thesis, discuss the weaknesses in the work, and suggest 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 the 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 continue 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 increase however the reliability and confidence of the users in the tested product.

Table 2.1: Parts of Software Testing [1], [16], [28], [55], [57]

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 3. Grey Box	1. Static 2. Dynamic

{table:addvalues

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 ensure 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 a 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 analyse 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 behaviour 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 blackbox and whitebox testing on the basis of perspective taken. In blackbox 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 whitebox 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 organised into static and dynamic testing on the basis test cases execution. In static testing test cases analysed 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. 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 through processing. Results are analysed 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 behaviour 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 straight forward, 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 technique.

Random Input
Executable Program
Output

Figure 2.1: Random Testing

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 criticised 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 prove that random testing and partition testing are equally effective. Furthermore the study conducted by Ntafos [43] conclude 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 colour points, block and strip inside each box indicate the point, block and strip fault patterns in the input domain.

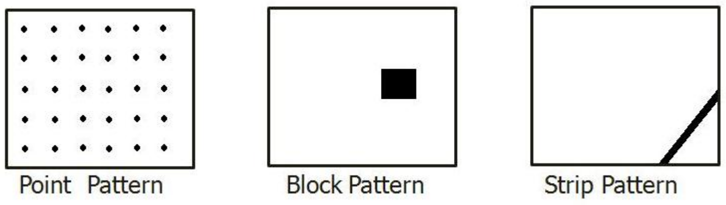


Figure 2.2: Patterns of failure causing inputs

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 a 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 may generate test inputs lurking too close or too far from the fault inducing input and thus failing to discover it. To generate more fault 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.

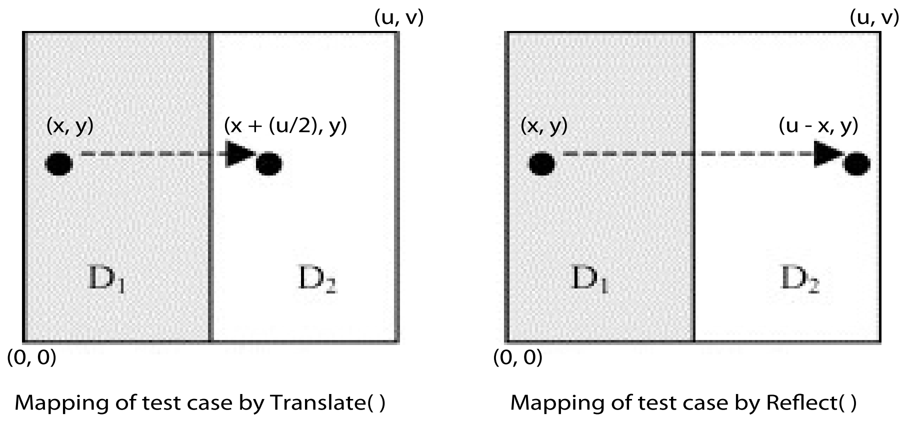

(u, v)
(x, y)
(x + (u/2), y)
D1
D2
(0, 0)
Mapping of test case by Translate( )
(u, v)
(x, y)
(u - x, y)
D1
D2
(0, 0)
Mapping of test case by Reflect( )

Figure 2.3: Mirror Adaptive Random Testing [13]

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 subdomain 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.

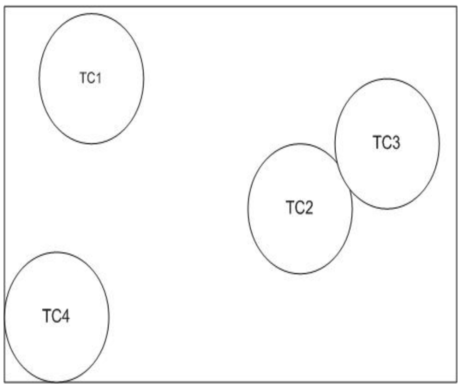


Figure 2.4: Input domain with exclusion zone around the selected test case

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 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 automate 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 initialises 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 instrument the given SUT at the start of the process in order to track its behaviour dynamically at run time. The results obtained are analysed in real time to systematically direct the test case execution along alternative path for maximum code coverage.

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 executables), 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 bring 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 between 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 [39] which 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 continue 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) which 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/undeclared 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.

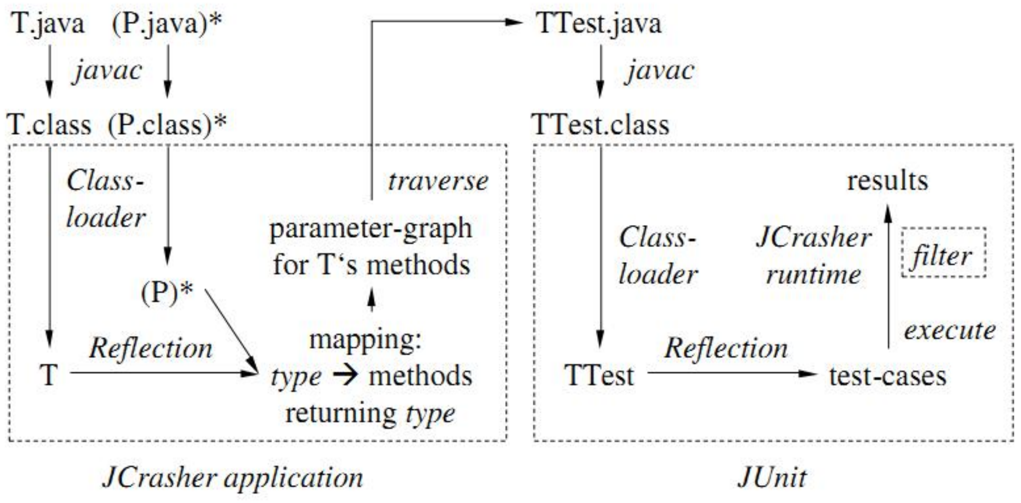


Figure 2.5: Process of robustness testing of java program with JCrasher [51]

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] analyse 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, filtration to reporting the results. One of its novelty 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:

- To define the operational profile of the classes i.e. the likely use of the class under test by other classes.
- 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.
- 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 short coming 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.

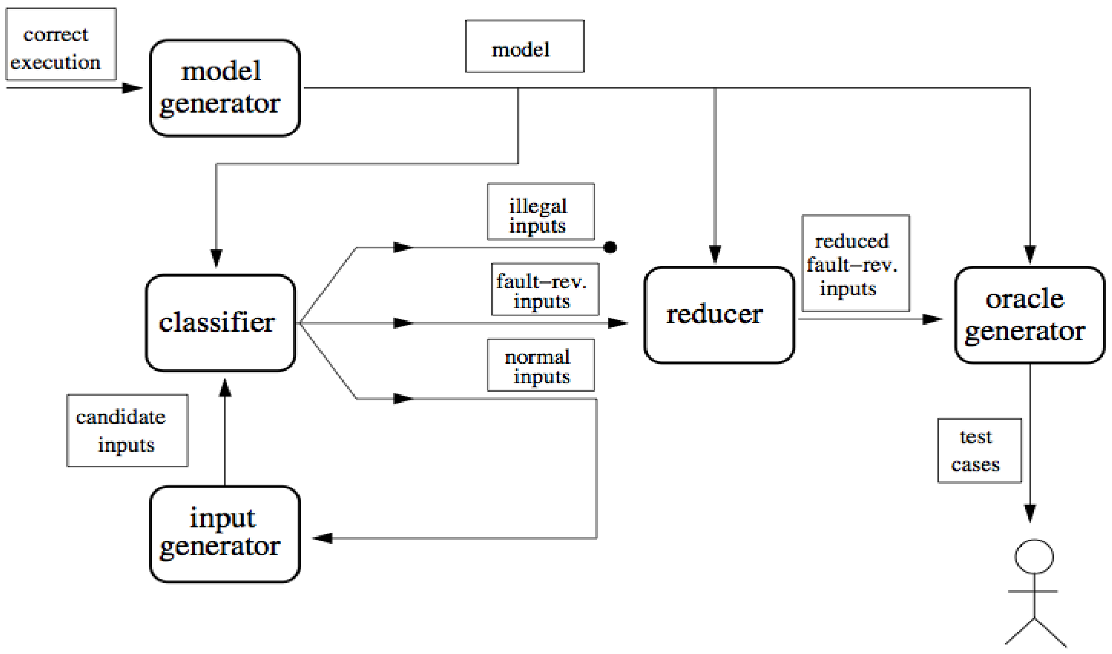


Figure 2.6: Main component of Eclat contributing to generate test input [50]

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 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 facilitate 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 short coming 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.

2.2.11 QuickCheck

QuickCheck [22] is a lightweight random testing tool that can be used for testing of Haskell programs [34]. Haskell is a functional programming language where programs are evaluated using expressions rather than statements as in the case of imperative programming. Since in Haskell most of the functions are pure other than IO functions, therefore it focuses on testing pure functions. These are the functions which depends upon its input parameters and make changes to them only.

QuickCheck takes as input the testers defined properties of the program called Haskell functions and the program to be tested. The tool uses built-in random generator to generate test data or the tester may provide custom define test data generator. Any generated test inputs that satisfies Haskell functions are declared as valid tests input. To cope with oracle problem, the authors have designed a simple domain-specific language of testable specifications embedded in Haskell. The tester use it to define expected properties of the functions under test. QuickCheck then checks and declare a fault in the function where a test case violate these properties.

2.2.12 Autotost

Based on Formal Automated testing AutoTest is a tool used for testing of Eiffel programs [19]. The Eiffel language use the concept of contracts (pre-conditions, postconditions and class invariants). Input can be a single class, method or a set of classes which is then processed by AutoTest to generate test cases. It generates both primitive and object type test cases. All the generated test cases are kept in a pool and then randomly a test case is selected from it for execution. A user can set the features of the AutoTest options include: Number of test cases to generate, whether to monitor pre or post condition, order of testing and the initial values of the primitives variables.

2.2.13 TestEra

TestEra [35] 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 [10]. 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).

2.2.14 Korat

Korat [5] is a novel framework for automated testing of Java programs based on their formal specifications [11]. 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.

2.2.15 YETI

York Extensible Testing Infrastructure (YETI) is an automated tool for testing Java, JML and .NET assemblies [46]. 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 [47]. The following sections briefly describe internal working and execution of YETI tool.

2.2.16 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.

2.2.17 Commnad-line Options

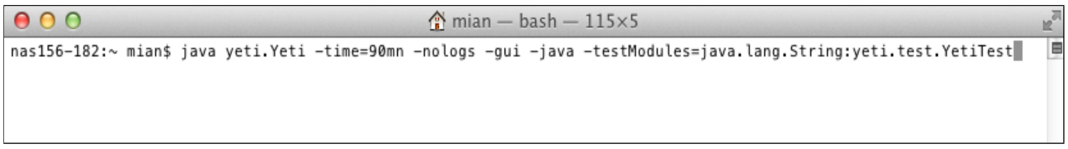
While YETI GUI launcher has been developed during this research study, to take maximum benefit of the available options one still need to launch YETI from CLI mode. These command-line options are case insensitive and can be provided as input to the tool in CLI mode. For example, to save processing power 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.

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

2.2.18 YETI Execution

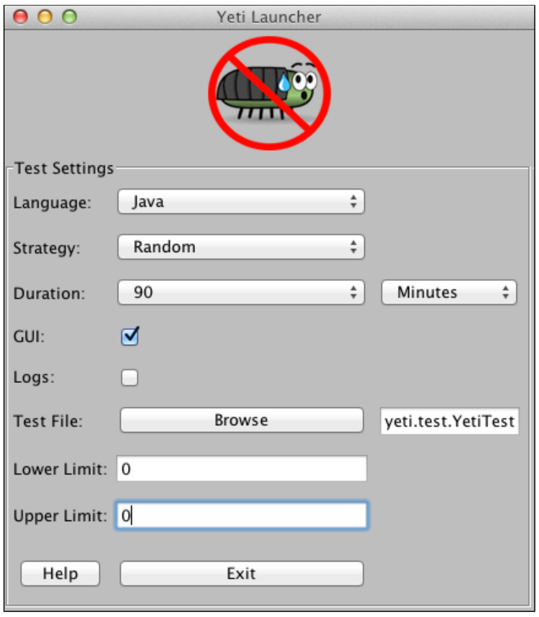
YETI being developed in Java is highly portable and can easily run on any operating system with Java Virtual Machine (JVM). YETI can be executed from both command line and GUI. To build and execute YETI, it is necessary to specify the project and all the .jar library files particularly javassist.jar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



```
nas156-182:~ mian$ java yeti.Yeti -time=90mn -nologs -gui -java -testModules=java.lang.String:yeti.test.YetiTest
```

Figure 2.7: 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.



Language: Java
Strategy: Random
Duration: 90 Minutes
GUI: ☑
Logs: ☐
Test File: Browse yeti.test.YetiTest
Lower Limit: 0
Upper Limit: 0

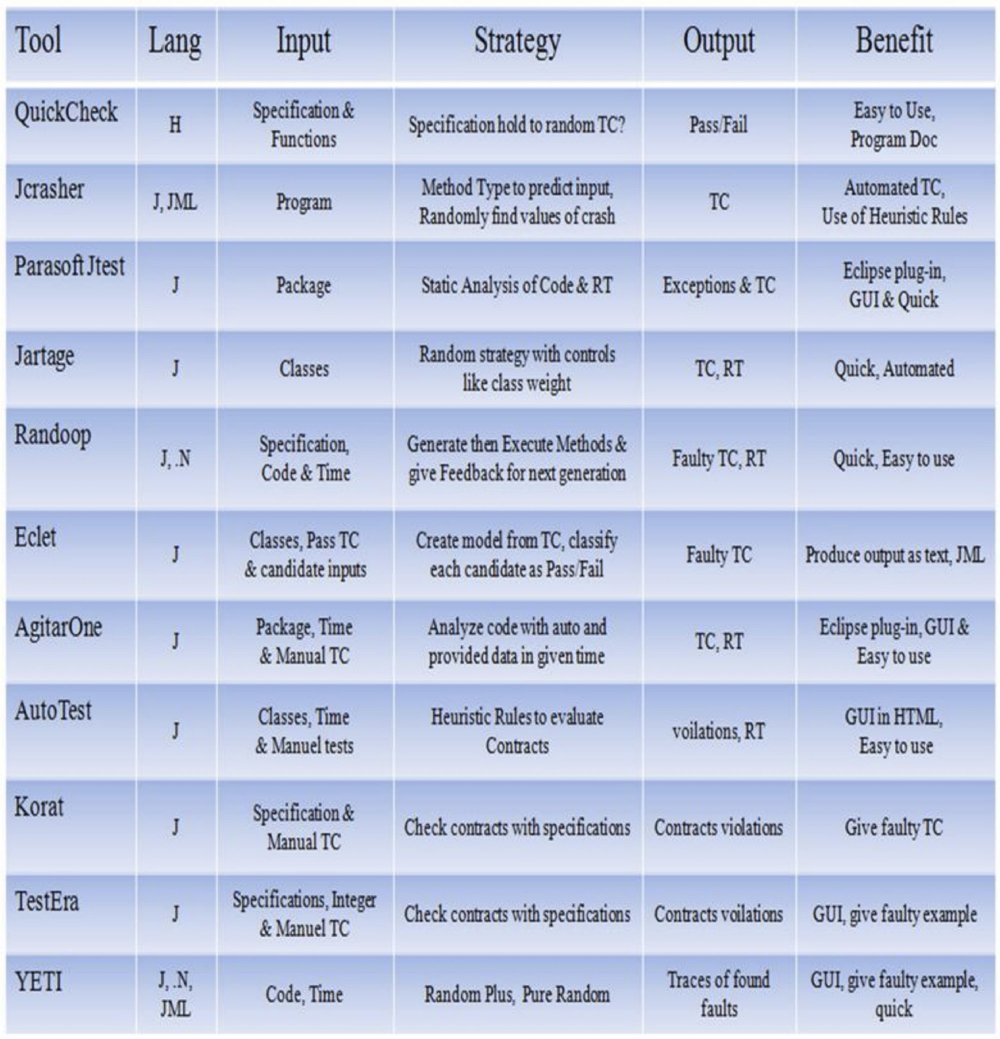
Figure 2.8: Command to launch YETI from GUI

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

2.2.19 YETI Report

2.2.20 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.



Tool	Lang	Input	Strategy	Output	Benefit
QuickCheck	H	Specification & Functions	Specification hold to random TC?	Pass/Fail	Easy to Use, Program Doc
Jcrasher	J, JML	Program	Method Type to predict input, Randomly find values of crash	TC	Automated TC, Use of Heuristic Rules
Parasoft Jtest	J	Package	Static Analysis of Code & RT	Exceptions & TC	Eclipse plug-in, GUI & Quick
Jartage	J	Classes	Random strategy with controls like class weight	TC, RT	Quick, Automated
Randoop	J, .N	Specification, Code & Time	Generate then Execute Methods & give Feedback for next generation	Faulty TC, RT	Quick, Easy to use
Eclet	J	Classes, Pass TC & candidate inputs	Create model from TC, classify each candidate as Pass/Fail	Faulty TC	Produce output as text, JML
AgitarOne	J	Package, Time & Manual TC	Analyze code with auto and provided data in given time	TC, RT	Eclipse plug-in, GUI & Easy to use
AutoTest	J	Classes, Time & Manuel tests	Heuristic Rules to evaluate Contracts	voilations, RT	GUI in HTML, Easy to use
Korat	J	Specification & Manual TC	Check contracts with specifications	Contracts violations	Give faulty TC
TestEra	J	Specifications, Integer & Manuel TC	Check contracts with specifications	Contracts voilations	GUI, give faulty example
YETI	J, .N, JML	Code, Time	Random Plus, Pure Random	Traces of found faults	GUI, give faulty example, quick

Figure 2.9: Summary of automated testing tools

2.3 Conclusion

Appdx A

and here I put a bit of postamble ...

Appdx B

and here I put some more postamble ...

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. v, 6

[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. 5

[3] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2, 2001. 8

[4] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. 7

[5] 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. 20

[6] 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. 9

[7] 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. 2, 8

[8] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Normalized restricted random testing. In *Reliable Software TechnologiesAda-Europe 2003*, pages 368–381. Springer, 2003. 12

[9] 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. 16

[10] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. *SIGSOFT Softw. Eng. Notes*, 24(6):285–302, 1999. 19

[11] Juei Chang and Debra J Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software EngineeringESEC/FSE99*, pages 285–302. Springer, 1999. 20

[12] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008. 2, 9

[13] 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. iv, 2, 11

[14] 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. 11

[15] 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. 2, 11, 13

[16] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994. v, 6

[17] I Ciupa, A Pretschner, M Oriol, A Leitner, and B Meyer. On the number and nature of faults found by random testing. *Software Testing Verification and Reliability*, 9999(9999):1–7, 2009. 8

[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. 14, 15

[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. 2, 15, 19

[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. 8

[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. 7, 9

[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. 19

[23] 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. 2

[24] Julie Cohen, Daniel Plakosh, and Kristi L Keeler. Robustness testing of software-intensive systems: Explanation and guide. 2005. 7

[25] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972. 6

[26] 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. 9

[27] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978. 7

[28] Marie-Claude Gaudel. Software testing based on formal specification. In *Testing Techniques in Software Engineering*, pages 215–242. Springer, 2010. v, 6, 8

[29] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005. 2, 12

[30] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, dec 1990. 9

[31] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994. 8, 9

[32] William E Howden. A functional approach to program testing and analysis. *Software Engineering, IEEE Transactions on*, (10):997–1005, 1986. 1

[33] 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. 1

[34] 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. 19

[35] 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. 19

[36] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990. 1

[37] 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. 8

[38] 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. 9

[39] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. 10(8):707–710, 1966. 15

[40] Thomas J McCabe. *Structured testing*, volume 500. IEEE Computer Society Press, 1983. 1

[41] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. 9

[42] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. 2, 5

[43] Simeon Ntafos. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998. 9

[44] A. Jefferson Offutt and J. Huffman Hayes. A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, 21(3):195–200, May 1996. 2

[45] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004. 17

[46] 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. 20

[47] Manuel Oriol and Faheem Ullah. Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:434–437, 2010. 20

[48] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002. 7

[49] Carlos Pacheco. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009. 9

[50] 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. iv, 17, 18

[51] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007. iv, 14, 16, 18

[52] 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. 2

[53] 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. 13

[54] 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. 8

[55] 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. v, 6, 8

[56] 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. 9

[57] 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. v, 6

[58] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 1999. 7

[59] 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. 14

[60] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. 8

[61] Lee J. White. Software testing and verification. *Advances in Computers*, 26(1):335–390, 1987. 2

[62] Wikipedia. Plagiarism — Wikipedia, the free encyclopedia, 20013. [Online; accessed 23-Mar-2013]. 2