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

May 5, 2013

Abstract

This is where you write your abstract ...

Contents

List of Figures

.

Acknowledgements

Several people have contributed to the completion of my PhD dissertation. However, the most prominent personality deserving due recognition is my worthy supervisor, 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 elder brothers Dr. Ashfaq, Dr. Aftab, Dr. Ishaq, Dr. Afaq and my sister Dr. Haleema who have been the source of inspiration for me to pursue higher studies. My immediate younger brother Dr. Ilyas and my younger sister Ayesha studying in the UK, deserve recognition for their help, well wishes and moral support. 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. 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 an outline of the thesis.

1.1 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 [34], [27]. Manual test data set generation is a time-consuming and laborious exercise [30], therefore, automated test data set generation is always preferred. Test data generators are classified in to Pathwise, Goal-Oriented, Intellighent and Random [54]. Random test data generation generate 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 [15].

Despite the benefits random testing offers, its simplistic and non-systematic nature

expose it to high criticism [53]. Myers & Sandler [36] 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 [9], Restricted Random Testing [6], Feedback directed Random Test Generation [45], Mirror Adaptive Random Testing [10] and Quasi Random Testing [12] 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 [38], [19]. 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 [24].

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.

1.2 Our 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 which 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 number 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 aims to increase 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.

1.3 Contributions

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

1.3.1 Dirt Spot Sweeping Random Strategy

Development of a new, enhanced and improved form of automated random testing: the Dirt Spot Sweeping Random (DSSR) strategy. This strategy is based on the assumption that faults and unique failures reside in contiguous blocks and stripes. The DSSR strategy starts as a regular random+ testing strategy a random testing technique with preference for boundary values. When a failure is found, it increases the chances of using neighbouring values of the failure in subsequent tests, thus slowly sweeping values around the failures found in hope of finding failures of different kind in its vicinity. The DSSR strategy is implemented in the YETI random testing tool. It is evaluated against random (R) and random+ (R+) strategies by testing 60 classes (35,785 line of code) with one million (105) calls for each session, 30 times for each strategy. The results indicate that for 31 classes, all three strategies find the same unique failures. We analysed the 29 remaining classes using t-tests and found that for 7 classes DSSR is significantly better than both R+ and R, for 8 classes it performs similarly to R+ and is significantly better than R, and for 2 classes it performs similarly to random and is better than R+. In all other cases, DSSR, R+ and R do not perform significantly differently. Numerically, the DSSR strategy finds 43 more unique failures than R and 12 more unique failures than R+.

1.3.2 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.

1.3.3 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 generate 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.

1.4 Thesis Outline

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. Test is more successful if it finds more errors in the software. Once errors are found in the SUT, the software is given back to the developers for removing the found

errors and after rectifying the mentioned errors the software is again handed over to the testers for retesting. The testing process starts from the very beginning of software development and remains continuos throughout the life of the software.

One thing to keep in mind is that program testing can be used to show the presence of bugs, but never to show the absence of bugs [20]. 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.

2.1.1 Software Test Plan

As proper planning is the key to success for many projects this is often also true with software testing. A software test plan is a well defined document that defines the goal, scope, method, resources and time schedule of the testing.

2.1.2 Software Testing Approaches

The testing process starts from the very beginning of the System Development Life Cycle (SDLC) and is carried out in the following two ways.

1. Static Testing
2. Dynamic Testing

2.1.3 Static Testing

The term static means still or non executable. Static Testing is the process in which software documentation/source code is checked for errors without any execution. All high quality softwares will always be accompanied by documentation in addition to software code. These include requirements, de- sign, technical, end user and marketing documentation. Reviews, walkthroughs or inspections are most commonly used for static tests. For instance it is necessary to do the static testing of user documentation for errors because software developed at the cost of millions of dollars have been neglected and abandoned for the only reason that end users were not able to find out the proper way to operate it to do their routine business. Users tend not to think I will figure out

how to operate this software; rather, they say, this software doesnt do what I need it to do, so I want the old system back [22].

2.1.4 Dynamic Testing

Dynamic means variable or changeable so Dynamic Testing is the process in which software code is executed and input is converted into output through processing. Results are analysed to find any error in the software. It is not necessary that dynamic testing start once the software is fully complete. Instead it can start from a single method/unit. Unit testing, integration testing, system testing, and acceptance test- ing are most commonly used as dynamic testing methods. Dynamic testing can be manual or automated. In manual testing the programmer develops the test cases which are executed by the developed software to find any error in processing or output. Similarly in automated testing the software or components of the software is given as input to testing software that automatically generates test cases and executes the SUT against them to find any errors. Manual testing typically consumes more time and resources than automated testing.

2.1.5 Methods of Dynamic Software Testing

There are three methods for testing the software dynamically.

1. Black-Box Testing
2. White-Box Testing
3. Grey-Box Testing

2.1.5.1 Black-Box Testing

Black-Box or Functional testing is the method in which the testers dont know about the structure of the software. Test cases are derived from the specifications of the SUT and tests pass only if the output checks, according to the specification no matter how it is internally processed by the software. The main emphasis of black-box testing is to check the functionality of the product.

2.1.5.2 White-Box Testing

White-Box or Structural testing is a method in which the testers must know about the complete structure of the software. Test cases are derived from the code structure of the SUT and the test is pass only if the results are correct according to the specification as well as the execution is carried out according to oracle. The main emphasis of the white-box testing is not only functionality but also code coverage [1].

2.1.5.3 Grey-Box Testing

Grey-Box testing is the combination of both black-box/functionality and white-box/structural testing. The tester knows about both the functionality and the internal structure of the SUT. Some of the test cases are based on the functionality and some of the test cases are based on the structure. Emphasis of grey-box testing is both on code coverage as well as functionality [49].

2.1.6 Software Testing Workflow

There are many software techniques like unit testing, integration testing, random testing, regression testing, system testing, acceptance testing, performance testing, load testing, stress testing, alpha testing, beta test etc. All testing techniques belong to black-box, white-box or grey-box approach. Each testing technique has its own strength and weaknesses but the technique in focus here is Random Testing.

We have explained software testing graphically with the help of plotting venn diagram on two dimensional axis. The positive x axis represent black-box while negative x axis represent white-box testing. Grey-box testing in the middle is represented by the overlapping of black-box and white-box testing. Similarly on positive y axis we have dynamic testing and on negative y axis we have static testing. Now if a test is black box and dynamic then the test will fall in 0 to 90 degree on the diagram and if the test is black-box and static then it will fall in 270 to 360 degree. On the other hand if the test is white-box and dynamic then it will fall in 90 to 180 degree and if the test is white-box and static then it will fall in 180 to 270 degrees.

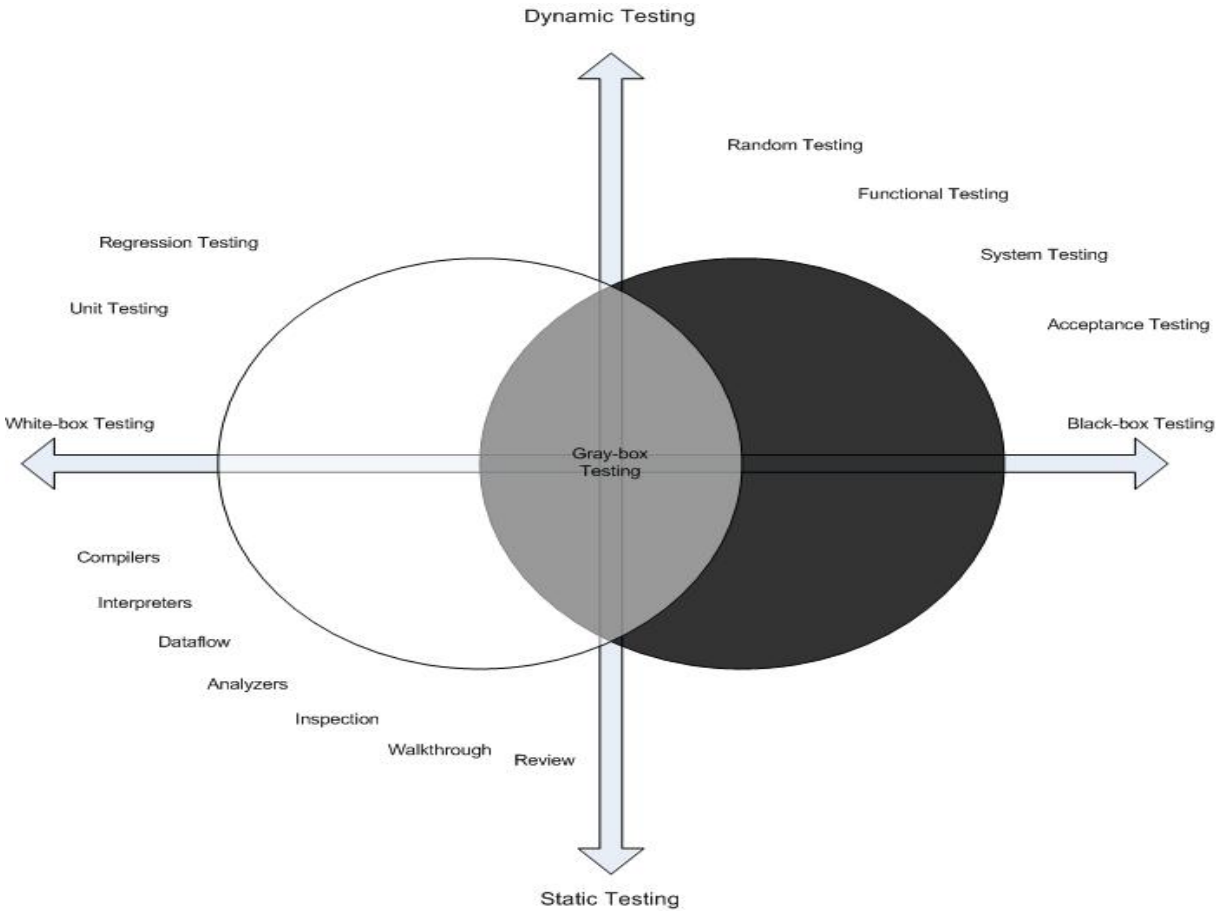

Figure 2.1: Software Testing Workflow

2.1.7 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 [17]. While manual testing is effective in some cases, in general, it is a laborious, time consuming, error-prone [51]. 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.

2.1.8 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 [31]. 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 [47].

2.1.8.1 Exhaustive Testing

A software testing technique in which a software is tested with all possible combination of inputs. This technique can prove conclusively that the software meet its specification however exhaustive testing is seldom feasible because of the large input domain or too many paths in a software code. Testers therefore are usually only able to use a small portion of a programs input domain to test a given software [7].

2.1.9 Test Oracle

Test oracles set the acceptable behaviour for test executions [48]. 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 [23]. 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 [52] argued that truly general test oracles are often unobtainable.

2.1.9.1 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 [6]. The input domain is a set of all possible inputs to the software under test. According to Richard H. [26], 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.

Figure 2.2: Random Testing

It is quick and cheap to generate random test data as it don't require too much intellectual and computational efforts [13]. This capability makes it an ideal choice for implementation in automated testing tools [16]. In addition, no human intervention in data generation/selection makes it one of the most unbiased testing technique.

Generating test cases with out using any background information makes it highly susceptible to criticism. Myers [35] 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 [50], [42].

Myers statement was not based on any experimental evidence and later on the experiments performed in [26], [17], [32] and [21] confirmed that random testing is as effective as any other systematic testing technique. The experiments in [21] 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 [25] also prove that random testing and partition testing are equally effective. Furthermore the study conducted by Ntafos [37] 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) [9] is based on the existence of failure patterns across the input domain detected by Chan et al [5]. 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.

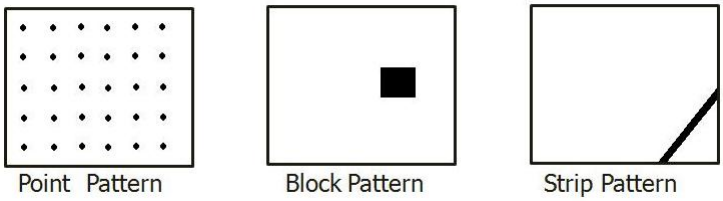

Figure 2.3: Patterns of failure causing inputs

In the figure 2.3 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.

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 [11] and [12].

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) [10] is an innovative approach that uses mirror partitioning technique to reduce the overhead of ART by decreasing the extra computation involved in ART.

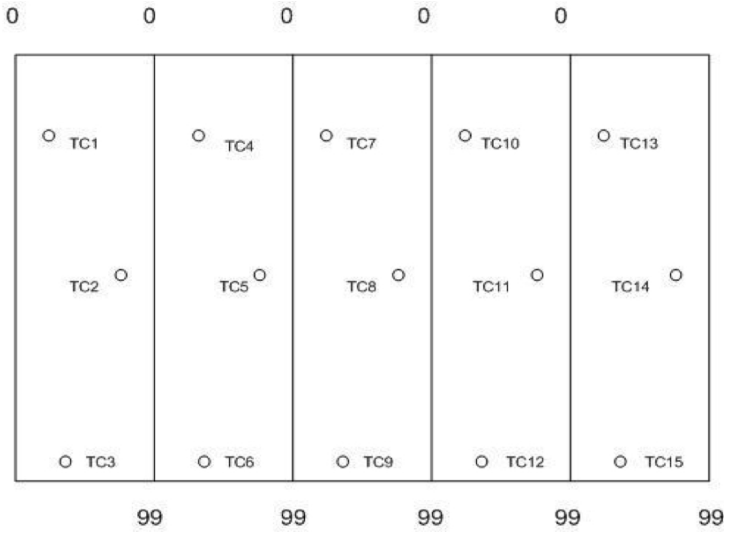

Figure 2.4: Mirror Adaptive Random Testing (0-500)

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 ???. 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 Directed Automated Random Testing

2.2.4 Quasi Random Testing

Quasi-random testing [12] is a technique developed to obtain an even better distribution of test cases across the input domain in less computation time with respect to ART. From various experiments Chen et al found out that the failure causing inputs not only form specific pattern but these patterns are continuous as well. Quasi-random testing don't restrict random selection of test cases like ART or RRT rather it uses a class with a formula. This formula forms an s-dimensional cube in s-dimensional input domain and produces number with Quasi sequence (a sequence of numbers that have small discrepancy and low dispersion) for an s-dimensional input domain. These sequence of numbers are then used by the Quasi approach to select the test cases from s-dimensional input domain. For performance analysis the author compared Quasi approach with ART and random testing. Results showed that the approach is better than random testing but not than ART.

2.2.5 Feedback-directed Random Test Generation

In a bid to improve random testing Pacheco et al., [46] developed a technique which produces unit tests randomly for object oriented programs which are later used for testing the units of the SUT. It is an incremental approach in which unit tests are created and executed against a set of contracts and filters. The feedback obtained from this execution serve as a basis for a sequence of new unit tests. The feedback of the unit test indicate that it is useful to create new input but if it is redundant or illegal like it throws IllegalArgumentEexception error then they are discarded and no unit test of similar nature is created based on its feedback. Thus it only selects unit tests which can be effective in finding bugs or can be used for regression testing. Results of the experiments adopting the technique of Feedback-directed random test generation shows that it can be more productive in code coverage and error detection than systematic and undirected random test generation.

2.2.6 Randoop: Feedback-directed Random Testing

Randoop stands for RANdom tester for Object Oriented Programs [44]. It tests software by using the principle of feedback-directed random test generation to produce unit tests for java and .NET. Randoops input is a set of classes that is to be tested within a certain time and optionally a set of contracts that extend the existing default contracts. After processing the input according to the method of feedback-directed random testing it give two test suites as output. One is contract voilating tests and the other is regression tests.

2.2.6.1 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 [14]. 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 [33] 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.

2.2.6.2 ARTOO Tool

After the criteria to calculate the distance between the objects is defined [14], 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.

2.2.6.3 Experimental Assessment of Random Testing 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. [15] 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.

2.2.7 Restricted Random Testing

Motivated from Adaptive random testing, aim of Restricted Random Testing (RRT) is the same that is selection of test cases from the input domain such that the whole input domain is represented [?]. The plan to achieve an even selection of test cases from the input domain is accomplished by forming an exclusion zone around the first random selected test case.

The next random test case then must be selected outside of this exclusion zone. It

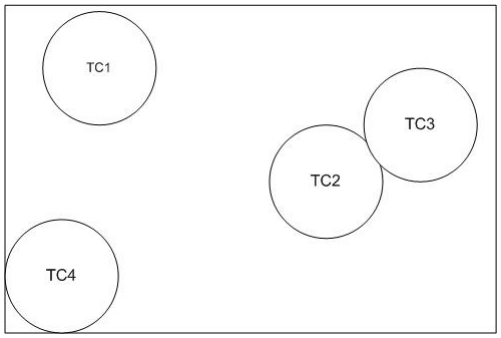

Figure 2.5: Input domain with exclusion zone around the selected test case

makes sure that there is enough distance between the two test cases. The exclusion zone is fixed around each test case and the area of each zone decreases with successive cases. Experimental results of seven error seeded program indicated that RRT is 55% more effective than ordianry/undirected random testing in terms of f-measure (Where f-measure is the total number of test cases required to find the first failure).

2.2.8 JCrasher

JCrasher is an automatic testing tool that uses a random testing technique to test java classes/programs [44]. The main features of JCrasher are: 1. The randomly created test cases are according to the type and parameters of the methods under test. 2. It uses special heuristics rules, after the execution of the test cases, to see whether the given excep- tions are real bugs or the generated input violated the pre-conditions of the program. 3. To clarify the testing from any old tests JCrasher make it sure that every test run on a clean state. 4. JCrasher also produces test cases for JUnit that can be integrated into IDEs like Eclipse. To use JCrasher we have to supply set of Java classes in byte code and testing time. JCrasher analyzes the classes and create test cases randomly with the same type and same parameter list. These test cases are only for public methods of the classes and they check for any system crash. List of exceptions is obtained as a result of execution of test cases which are differentiated as bugs and precondition violations by the input.

2.2.9 JArtage

Jartege (JAwa Random TEst GEnerator) is a tool that randomly generates unit tests for classes specified with JML (Java Modeling Language) [39]. The specification of Java classes with JML serves two pur- poses. First, all the test cases generated by Jartege have to verify the conditions defined by JML and thus irrelevant test cases are eliminated. Secondly these JML specifications are also used as oracles. Apart from the JML specification which are made by hand it automates the whole testing process which include test case generation, execution, comparing it against oracle and using the generated test cases for future regression testing.

2.2.10 Eclat

Eclat [43] is a tool that automatically generates unit tests for Java. Eclat can be executed from both command line or from IDE where it can be installed as a plug-in. [28]. Eclat selects a sub-set of test inputs from a large domain, that is likely to reveal fault in the SUT. Eclat takes a correct execution of the SUT and on the basis of it creates an operational model. It then selects only these test inputs from the input domain which fail to comply with the model. A Reducer function removes the redundant test inputs and the remaining test inputs are likely to discover faults in the SUT. Based on the operational model it also produces an automated oracle. Various experiments results shows that Eclats is very effective in finding faults and the ratio of finding faults and test inputs is almost same.

2.2.11 JTest

Parasoft Jtest is a commercial tool that automatically generates and execute unit tests. It can be easily integrated to Java IDEs like Eclipse where it provide two main functionalities, i.e. Static Analysis, Unit testing and code coverage. [25] In static analysis Jtest takes a complete project or set of classes as input and compares it with a list of built-in rules. The statement violating any of these rules is an error. It also suggests probable fixes for the detected fault. For unit testing it takes a class as an input and processes a number of scenarios against it to generate and execute unit tests. Once unit tests are executed they become the part of regression test for future reference. Jtest

also shows the code coverage of the program by colour coding the statements that are not executed by the unit tests.

2.2.12 QuickCheck

QuickCheck [18] is a light weight random testing tool that is developed specifically for testing of Haskell programs [28]. Haskell is a functional programming language where programs are evaluated using expressions rather than statements as in case of imperative programming. Therefore in this process the tester defines certain expressions for the functions that must hold for a large number of test cases to be correct. These test cases are generated automatically through generator function which can be set by the tester to generate random test cases or according to specific criteria. After processing all the generated test cases any test case that causes the expression to become false is considered faults.

2.2.13 AgitarOne

AgitarOne is a commercial tool that automatically generates unit tests. It has a Junit Generator engine that can create 25,000 lines or more of Junit per hour [29]. It can be easily integrated into famous IDE like Eclipse. It takes as input, classes under test, time and optionally any knowledge or test cases that has a positive influence on the performance of the testing process. The generated Junit tests can be run from the same IDE and can also be used for later regression testing. The GUI interface is called a dashboard which provides in depth knowledge of the tests conducted, failures detected, alerts and the archieves of the tests conducted earlier. It also shows the coverage obtained after executing the Junits against the code under test.

2.2.14 Autotost

Based on Formal Automated testing AutoTest is a tool used for testing of Eiffel programs [15]. 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.

2.2.15 TestEra

TestEra [29] 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 [8]. 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 post-conditions (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).

2.2.16 Korat

Korat [4] is an automated testing tool for Java programs that generates and execute test cases for a method based on its formal specification. To generate test cases for a method Korat makes use of its pre-condition. It then executes the generated test cases against the method specifications. Korat uses JML for specifications. In order to generate test cases for a method Korat constructs new methods that return a Boolean value (Java Predicate) from its pre-conditions. When given these Java predicates Korat generates all non isomorphic input for which the return value of predicate is true. To check correctness of the method, Korat executes the test cases on that method and analyzes the output with the post conditions of the method (oracle). A fault in a method under test throws an exception to indicate the violation of the post-condition.

2.2.17 YETI

The final tool we discuss is YETI (the York Extensible Testing Infrastructure) [33], which is entirely automated and freely available as open-source. It can be used for testing programs written in Java, JML, C, command-line and .Net [40]. It can also

be run in cloud for faster execution [41]. It is implemented in Java and uses random technique for testing. It has GUI which makes it easy to diagnose problems at runtime. It is able to call up to one million calls per minute on Java code. YETI oracle is language dependant. If the specifications are available, YETI checks the code against the specifications for any inconsistency. In case of programs having assertions YETI interprets violations as failures and in case there is no specifications or assertions, YETI performs robustness testing and considers undeclared run- time exceptions as faults. Errors-revealing test cases are reproduced at the end of each testing session. Experiments conducted with YETI showed significant number of bugs in Java.lang class (45 faults) and Itext (120 faults).

2.2.18 Tools for Automated Random Testing

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

2.3 Conclusion

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 2.6: Summary of automated testing tools

Chapter 3

Dirt Spot Sweeping Random Strategy

3.1 First Section

nd now I begin my second chapter here ...

3.2 Second Section

nd here I write more ...

3.2.1 first subsection in the Second Section

... and some more ...

3.2.2 second subsection in the Second Section

... and some more ...

3.2.3 third subsection in the Second Section

... and some more ...

Appdx A

and here I put a bit of postamble ...

Appdx B

and here I put some more postamble ...

References

[1] B. B. Agarwal, S. P. Tayal, and M. Gupta. *Software Engineering and Testing: An Introduction*. Jones & Bartlett Publishers, 1 edition, March 2009. 9

[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. 6

[3] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2, 2001. 11

[4] 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. 22

[5] 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. 13

[6] 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, 12

[7] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Normalized restricted random testing. In *Reliable Software TechnologiesAda-Europe 2003*, pages 368–381. Springer, 2003. 11

[8] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. *SIGSOFT Softw. Eng. Notes*, 24(6):285–302, 1999. 22

[9] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008. 2, 13

[10] 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. 2, 15

[11] 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. 14

[12] 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, 14, 16

[13] 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. 12

[14] 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. 17

[15] 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. 1, 18, 21

[16] 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. 12

[17] 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. 10, 12

[18] 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. 21

[19] 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

[20] Edsger W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming, pages 1–82. Academic Press Ltd., London, UK, UK, 1972. 7

[21] 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. 12

[22] Gerald D. Everett and Raymond McLeod Jr. *Software Testing: Testing Across the Entire Software Development Life Cycle*. Wiley-IEEE Computer Society Pr, 2007. 8

[23] Marie-Claude Gaudel. Software testing based on formal specification. In *Testing Techniques in Software Engineering*, pages 215–242. Springer, 2010. 11

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

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

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

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

[28] 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. 21

[29] 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. 22

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

[31] 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. 11

[32] 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. 12

[33] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. 10(8):707–710, 1966. 17

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

[35] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. 12

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

[37] Simeon Ntafos. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998. 13

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

[39] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004. 20

[40] 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. 22

[41] Manuel Oriol and Faheem Ullah. Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:434–437, 2010. 23

[42] Carlos Pacheco. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009. 12

[43] 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. 20

[44] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007. 17, 19

[45] 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

[46] 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. 16

[47] 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. 11

[48] 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. 11

[49] Roman Savenkov. *How to Become a Software Tester*. Roman Savenkov, 1 edition, November 2008. 9

[50] 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. 12

[51] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 1999. 10

[52] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. 11

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

[54] Wikipedia. Plagiarism — Wikipedia, the free encyclopedia, 20013. [Online; accessed 23-Mar-2013]. 1