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 14, 2013

Abstract

This is where you write your abstract ...

Contents

List of Figures

List of Tables

.

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 [36], [29]. Manual test data set generation is a time-consuming and laborious exercise [32], therefore, automated test data set generation is always preferred. Test data generators are classified in to Pathwise, Goal-Oriented, Intellighent and Random [57]. 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 [16].

Despite the benefits random testing offers, its simplistic and non-systematic nature expose it to high criticism [56]. Myers & Sandler [38] 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 [7], Feedback directed Random Test Generation [48], 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 [40], [20]. 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 [26].

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. A successful test is one that finds a fault [38], 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" [22]. 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], [13], [25], [51], [53]

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

2.1.1 Software Testing Levels

Unit testing, integration testing and system testing [13] 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 [21]. 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 [44].

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 [24]

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 [18]. While manual testing is effective in some cases, in general, it is a laborious, time consuming, error-prone [54]. 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 [33]. 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 [50].

2.1.7 Test Oracle

Test oracles set the acceptable behaviour for test executions [51]. 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 [25]. 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 [55] 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. [28], 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.

Random Input
Executable Program
Output

Figure 2.1: Random Testing

It is quick and cheap to generate random test data as it don't require too much intellectual and computational efforts [14]. This capability makes it an ideal choice for implementation in automated testing tools [17]. 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 [37] 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 [52], [45].

Myers statement was not based on any experimental evidence and later on the experiments performed in [28], [18], [34] and [23] confirmed that random testing is as effective as any other systematic testing technique. The experiments in [23] 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 [27] also prove that random testing and partition testing are equally effective. Furthermore the study conducted by Ntafos [39] 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 [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.

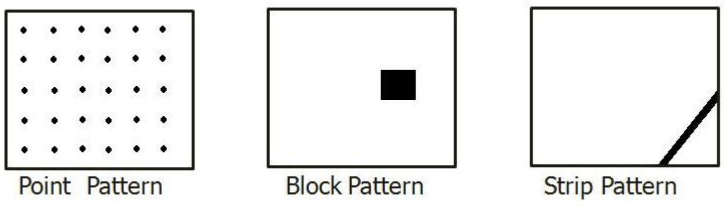


Figure 2.2: Patterns of failure causing inputs

In the figure 3.1 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.

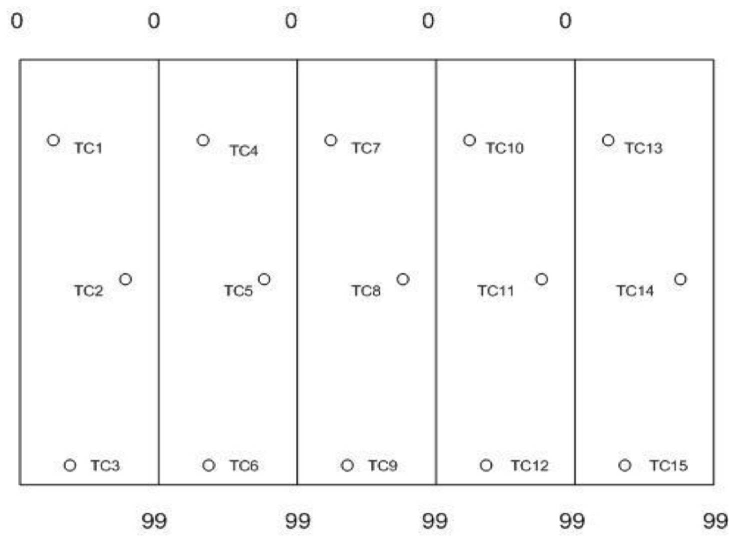


Figure 2.3: Mirror Adaptive Random Testing (0-500)

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., [49] 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 [47]. 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 [15]. 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 [35] 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 [15], 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. [16] 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

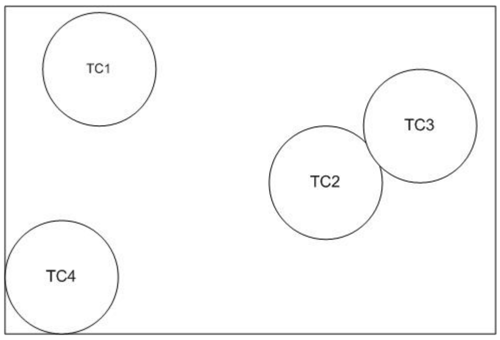


Figure 2.4: Input domain with exclusion zone around the selected test case

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 [47]. 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) [41]. 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 [46] 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 [19] is a light weight random testing tool that is developed specifically for testing of Haskell programs [30]. 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 [16]. 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 [31] 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 [5] 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.

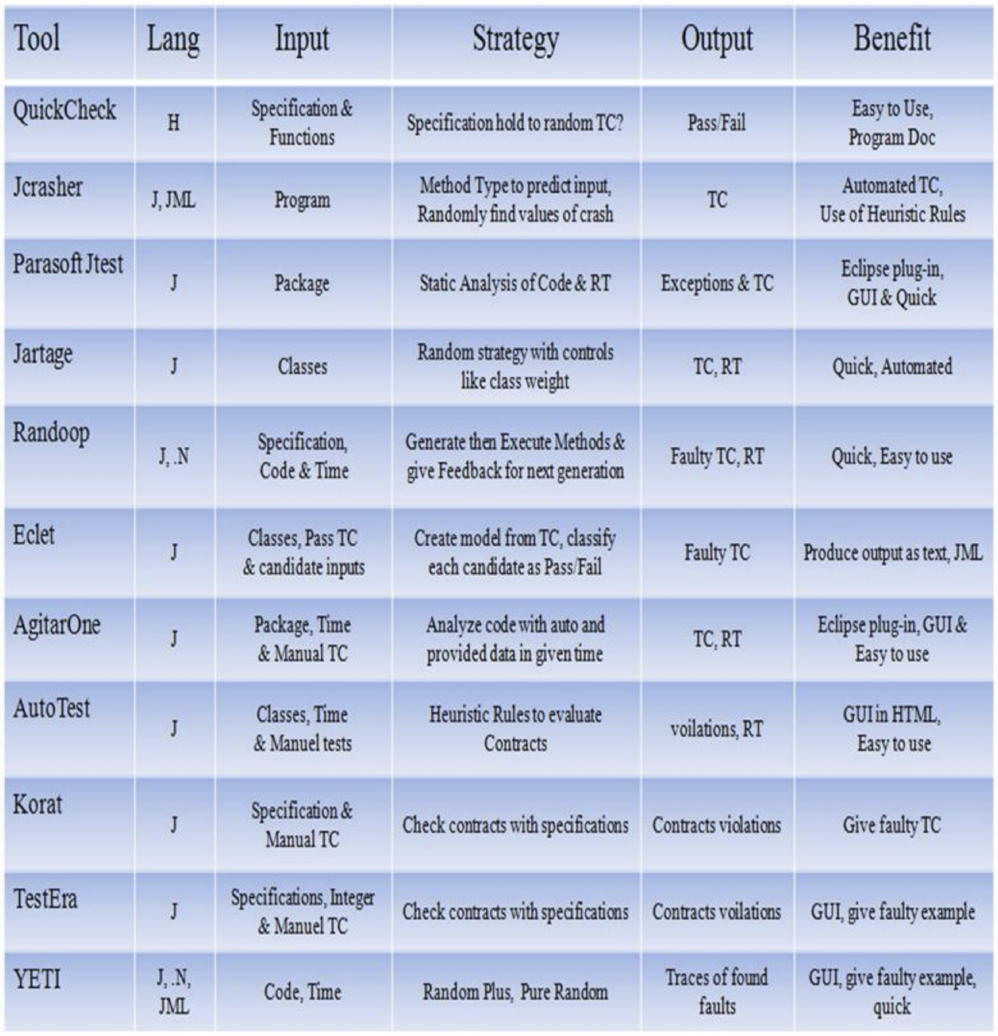
2.2.17 YETI

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 [42]. It can also be run in cloud for faster execution [43]. 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 2.5: Summary of automated testing tools

Chapter 3

Dirt Spot Sweeping Random Strategy

3.1 Introduction

The success of a software testing technique is mainly based on the number of faults it discovers in the Software Under Test (SUT). An efficient testing process discovers the maximum number of faults in a minimum possible time. Exhaustive testing, where software is tested against all possible inputs, is mostly not feasible because of the large size of the input domain, limited resources and strict time constraints. Therefore, strategies in automated software testing tools are developed with the aim to select more fault-finding test input from input domain for a given SUT. Producing such targeted test input is difficult because each system has its own requirements and functionality.

Chan et al. [6] discovered that there are patterns of failure-causing inputs across the input domain. They divided the patterns into point, block and strip patterns on the basis of their occurrence across the input domain. Chen et al. [9] found that the performance of random testing can be increased by slightly altering the technique of test case selection. In adaptive random testing, they found that the performance of random testing increases by up to 50% when test input is selected evenly across the whole input domain. This was mainly attributed to the better distribution of input which increased the chance of selecting inputs from failure patterns. Similarly Restricted Random Testing [7], Feedback directed Random Test Generation [48], Mirror Adaptive Random Testing [10] and Quasi Random Testing [12] stress the need for test case selection covering the whole input domain to get better results.

In this paper we take the assumption that for a significant number of classes failure domains are contiguous or are very close by. From this assumption, we devised the Dirt Spot Sweeping[1] Random (DSSR) strategy which starts as a random+ strategy — a random strategy focusing more on boundary values. When a new failure is found, it increases the chances of finding more faults using neighbouring values. As in previous studies [?] we approximate faults with unique failures. Since this strategy is an extension of random testing strategy, it has the full potential to find all unique failures in the program, but additionally we expect it to be faster at finding unique failures, for classes in which failure domains are contiguous, as compared with random (R) and random+ (R+) strategies.

We implemented the DSSR strategy in the random testing tool YETI[2]. To evaluate our approach, we tested 30 times each one of the 60 classes of 32 different projects from the Qualitas Corpus[3] with each of the three strategies R, R+ and DSSR. We observed that for 53% of the classes all three strategies find the same unique failures, for remaining 47% DSSR strategy perform up to 33% better than random strategy and up to 17% better than random+ strategy. We also validated the approach by comparing the significance of these results using t-tests and found out that for 7 classes DSSR was significantly better than both R+ and R, for 8 classes DSSR performed similarly to R+ and significantly better than R, while in 2 cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Numerically, the DSSR strategy found 43 more unique failures than R and 12 more unique failures than R+ strategy.

The rest of this paper is organised as follows:
Section 3.2 describes the DSSR strategy. Section 3.3 presents implementation of the DSSR strategy. Section 3.4 explains the experimental setup. Section 3.5 shows results of the experiments. Section 3.6 discusses the results. Section 3.7 presents related work and Section 3.8, concludes the study.

[1]The name refers to the cleaning robots strategy which insists on places where dirt has been found in large amount.

[2]http://www.yetitest.org

[3]http://www.qualitascorpus.com

3.2 Dirt Spot Sweeping Random Strategy

The new software testing technique named, Dirt Spot Sweeping Random (DSSR) strategy combines the random+ strategy with a dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, faults and unique failures reside in contiguous block and strip pattern. If this is true, DSS increase the performance of the test strategy. Before presenting the details of the DSSR strategy, it is pertinent to review briefly the Random and the Random+ strategy.

3.2.1 Random Strategy (R)

The random strategy is a black-box testing technique in which the SUT is executed using randomly selected test data. Test results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. In the absence of contracts and assertions the exceptions defined by the programming language are used as test oracles. Because of its black-box testing nature, this strategy is particularly effective in testing softwares where the developers want to keep the source code secret [**?**]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [14; 18]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [17]. YETI [43**?**], AutoTest [16; 33], QuickCheck [19], Randoop [49], JArtege [41] are some of the most common automated testing tools based on random strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [38] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, [16; 23**? ? ?**] have proved experimentally that random testing is simple to implement, cost effective, efficient and free from human bias as compared to its rival techniques.

Programs tested at random typically fail a large number of times (there are a large number of calls), therefore, it is necessary to cluster failures that likely represent the same fault. The traditional way of doing it is to compare the full stack traces and error types and use this as an equivalence class [16**?**] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

3.2.2 Random Plus Strategy (R+)

The random+ strategy [33] is an extension of the random strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values [?] are the values on the start and end of a particular type. For instance, such values for `int` could be `MAX_INT`, `MAX_INT-1`, `MAX_INT-2`; `MIN_INT`, `MIN_INT+1`, `MIN_INT+2`. Similarly, the tester might also add some other special values that he considers effective in finding faults for the SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the pre-defined list of special values. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [18].

3.2.3 Dirt Spot Sweeping (DSS)

Chan et al. [6] found that there are patterns of failure-causing inputs across the input domain. Figure 3.1 shows these patterns for two dimensional input domain. They divided these patterns into three types called points, block and strip patterns. The black area (points, block and strip) inside the box show the input which causes the system to fail while white area inside the box represent the genuine input. Boundary of the box (black solid line) surrounds the complete input domain and represents the boundary values. They argue that a strategy has more chances of hitting these fault patterns if test cases far away from each other are selected. Other researchers [7; 10; 12], also tried to generate test cases further away from one another targeting these patterns and achieved better performance. Such increase in performance indicate that faults more often occur contiguous across the input domain. In Dirt Spot Sweeping we propose that if a value reveals fault from the block or strip pattern then for the selection of the next test value, DSS may not look farthest away from the known value and rather pick the closest test value to find another fault from the same region.

ART_Patterns.png

Figure 3.1: Failure patterns across input domain [9]

Dirt spot sweeping is the part of DSSR strategy that comes into action when a failure is found in the system. On finding a failure, it immediately adds the value causing the failure and its neighbouring values to the existing list of interesting values. For example, in a program when the `int` type value of 50 causes a failure in the system then spot sweeping will add values from 47 to 53 to the list of interesting values. If the failure lies in the block or strip pattern, then adding it's neighbouring values will explore other failures present in the block or strip. As against random plus where the list of interesting values remain static, in DSSR strategy the list of interesting values is dynamic and changes during the test execution of each program.

block2.png

Figure 3.2: DSSR covering block and strip pattern

Figure 3.2 shows how DSS explores the failures residing in the block and strip patterns of a program. The coverage of block and strip pattern is shown in spiral form because first failure leads to second, second to third and so on till the end. In case the failure is positioned on the point pattern then the added values may not be effective because point pattern is only an arbitrary failure point in the whole input domain.

3.2.4 Structure of the Dirt Spot Sweeping Random Strategy

The DSSR strategy continuously tracks the number of failures during the execution of the test. This tracking is done in a very effective way with zero or minimum overhead to keep the overhead up to bare minimum [**?**]. The test execution is started by R+

strategy and continues till a failure is found in the SUT after which the program copies the values leading to the failure as well as the surrounding values to the variable list of interesting values.

flowchart1.pdf

Figure 3.3: Working mechanism of DSSR Strategy

The flowchart presented in Figure 3.3 depicts that, when the failure finding value is of primitive type, the DSSR strategy identifies its type and add values only of that particular type to the list of interesting values. The resultant list of interesting values provide relevant test data for the remaining test session and the generated test cases are more targeted towards finding new failures around the existing failures in the given SUT.

Boundary and other special values that have a high tendency of finding faults in the SUT are added to the list of interesting values by random+ strategy prior to the start of test session where as in DSSR strategy the fault-finding and its surrounding values are added at runtime when a failure is found.

Table 3.1 presents the values are added to the list of interesting values when a failure is found. In the table the test value is represented by X where X can be int, double, float, long, byte, short, char and String. All values are converted to their respective types before adding them to the list of interesting values.

Table 3.1: Neighbouring values for primitive types and String

Type	Values to be added
X is int, double, float, long, byte, short & char	X, X+1, X+2, X-1, X-2

X is String	X
	X + “ ”
	“ ” + X
	X.toUpperCase()
	X.toLowerCase()
	X.trim()
	X.substring(2)
	X.substring(1, X.length()-1)

3.2.5 Explanation of DSSR strategy on a concrete example

The DSSR strategy is explained through a simple program seeded with three faults. The first fault is a division by zero exception denoted by 1 while the second and third faults are failing assertion denoted by 2 and 3 in the given program below followed by description of how the strategy perform execution.

```
/**
* Calculate square of given number
* and verify results.
* The code contain 3 faults.
* @author (Mian and Manuel)
*/
public class Math1 {
 public void calc (int num1) {
  // Square num1 and store result.
  int result1 = num1 * num1;
  int result2 = result1 / num1; // 1
  assert Math.sqrt(result1) == num1; // 2
  assert result1 >= num1; // 3
 }
}
```

In the above code, one primitive variable of type `int` is used, therefore, the input domain for DSSR strategy is from `-2,147,483,648 to 2,147,483,647`. The strategy further select values (`0, Integer.MIN_VALUE` & `Integer.MAX_VALUE`) as interesting values which are prioritised for selection as inputs. As the test starts, three faults are quickly discovered by DSSR strategy in the following order.

Fault 1: The strategy select value `0` for variable `num1` in the first test case because `0` is available in the list of interesting values and therefore its priority is higher than other values. This will cause Java to generate division by zero exception (1).

Fault 2: After discovering the first fault, the strategy adds it and its surrounding values to the list of interesting values i.e. `0, 1, 2, 3 and -1, -2, -3` in this case. In the second test case the strategy may pick `-3` as a test value which may lead to the second fault where assertion (2) fails because the square root of `9` is `3` instead of the input value -3.

Fault 3: After a few tests the strategy may select `Integer.MAX_VALUE` for variable `num1` from the list of interesting values leading to discovery of the 3rd fault because int variable `result1` will not be able to store the square of `Integer.MAX_VALUE`. Instead of the actual square value Java assigns a negative value (Java language rule) to variable result1 that will lead to the violation of the next assertion (3).

The above process explains that including the border, fault-finding and surrounding values to the list of interesting values in DSSR strategy lead to the available faults quickly and in fewer tests as compared to random and random+ strategy. R and R+ takes more number of tests and time to discover the second and third faults because in these strategies the search for new unique failures starts again randomly in spite of the fact that the remaining faults are very close to the first one.

3.3 Implementation of the DSSR strategy

Implementation of the DSSR strategy is made in the YETI open-source automated random testing tool. YETI, coded in Java language, is capable of testing systems developed in procedural, functional and object-oriented languages. Its language-agnostic meta model enables it to test programs written in multiple languages including Java, C#, JML and .Net. The core features of YETI include easy extensibility for future growth, high speed (up to one million calls per minute on java code), real time logging, real time GUI support, capability to test programs with multiple strategies and auto generation of test report at the end of test session. For large-scale testing there is a cloud-enabled version of YETI, capable of executing parallel test sessions in Cloud [43]. A number of hitherto faults have successfully been found by YETI in various production softwares [**? ?**].

YETI can be divided into three decoupled main parts: the core infrastructure, language-specific bindings and strategies. The core infrastructure contains representation for routines, a group of types and a pool of specific type objects. The language specific bindings contain the code to make the call and process the results. The strategies define the procedure of selecting the modules (classes), the routines (methods) and generation of values for instances involved in the routines. By default, YETI uses the random strategy if no particular strategy is defined during test initialisation. It also enables the user to control the probability of using null values and the percentage of newly created objects for each test session. YETI provides an interactive Graphical User Interface (GUI) in which users can see the progress of the current test in real time. In addition to GUI, YETI also provides extensive logs of the test session for more in-depth analysis.

The DSSR strategy is an extension of YetiRandomPlusStrategy, an extended form of the YetiRandomStrategy. The class hierarchy is shown in Figure 3.4.

Figure 3.4: Class Hierarchy of DSSR in YETI

3.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [33]. General factors such as system software and hardware, YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability have been kept constant in the experiments.

3.4.1 Research questions

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

1. Is there an absolute best among R, R+ and DSSR strategies?
2. Are there classes for which any of the three strategies provide better results?
3. Can we pick the best default strategy between R, R+ and DSSR?

3.4.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [?]. The Qualitas Corpus is a curated collection of open source java projects built with the aim of helping empirical research on software engineering. These projects have been collected in an organised form containing the source and binary forms. Version 20101126, which contains 106 open source java projects is used in the current evaluation. In our experiments we selected 60 random classes from 32 random projects. All the selected classes produced at least one fault and did not time out with maximum testing session of 10 minutes. Every class is tested thirty times by each strategy (R, R+, DSSR). Name, version and size of the projects to which the classes belong are given in table 3.2 while test details of the classes is presented in table 3.3. Line of Code (LOC) tested per class and its total is shown in column 3 of table 3.3.

Every class is evaluated through 10^5 calls in each test session.[1] Because of the absence of the contracts and assertions in the code under test, Similar approach as used in previous studies [?] is followed using undeclared exceptions to compute unique failures.

All tests are performed with a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon processor with 6 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java$^{\text{TM}}$SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

3.4.3 Performance measurement criteria

Various measures including the E-measure (expected number of failures detected), P-measure (probability of detecting at least one failure) and F-measure (number of test cases used to find the first fault) have been used by researchers to find the effectiveness of the random test strategy. The E-measure and P-measure have been heavily criticised [9] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [? ?]. In our initial experiments the F-measure is used to evaluate the efficiency. However it was realised that this is not the right choice. In some experiments a strategy found the first fault quickly than the other

[1]The total number of tests is thus $60 \times 30 \times 3 \times 10^5 = 540 \times 10^6\ tests.$

but on completion of test session that very strategy found lower number of total faults than the rival strategy. The preference given to a strategy by F-measure because it finds the first fault quickly without giving due consideration to the total number of faults is not fair [**?**].

The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered faults in one go therefore, F-measure is not the favourable choice. In our experiments, performance of the strategy is measured by the maximum number of faults detected in SUT by a particular number of test calls [16; 48**?**]. This measurement is effective because it considers the performance of the strategy when all other factors are kept constant.

3.5 Results

Results of the experiments including class name, Line of Code (LOC), mean value, maximum and minimum number of unique failures and relative standard deviation for each of the 60 classes tested using R, R+ and DSSR strategy are presented in Table 3.3. Each strategy found an equal number of faults in 31 classes while in the remaining 29 classes the three strategies performed differently from one another. The total of mean values of unique failures in DSSR (1075) is higher than for R (1040) or R+ (1061) strategies. DSSR also finds a higher number of maximum unique failures (1118) than both R (1075), and R+ (1106). DSSR strategy finds 43 and 12 more unique faults compared to R and R+ respectively. The minimum number of unique faults found by DSSR (1032) is also higher than for R (973) and R+ (1009) which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

3.5.1 Is there an absolute best among R, R+ and DSSR strategies?

Based on our findings DSSR is at least as good as R and R+ in almost all cases, it is also significantly better than both R and R+ in 12% of the classes. Figure 3.5 presents the average improvements of DSSR strategy over R and R+ strategy over the 17 classes for which there is a significant difference between DSSR and R or R+. The blue line with diamond symbol shows performance of DSSR over R and the red line with square symbols depicts the improvement of DSSR over R+ strategy. The classes where blue line with diamond symbols show the improvement of DSSR over R and red line with square symbols show the improvement of DSSR over R+.

The improvement of DSSR over R and R+ strategy is calculated by applying the formula (1) and (2) respectively.

$$\frac{Averagefaults_{(DSSR)} - Averagefaults_{(R)}}{Averagefaults_{(R)}} * 100 \tag{3.1}$$

$$\frac{Averagefaults_{(DSSR)} - Averagefaults_{(R+)}}{Averagefaults_{(R+)}} * 100 \tag{3.2}$$

The findings show that DSSR strategy perform up to 33% better than R and up to 17% better than R+ strategy. In some cases DSSR perform equally well with R and R+ but in no case DSSR performed lower than R and R+. Based on the results it can be stated that DSSR strategy is a better choice than R and R+ strategy.

3.5.2 Are there classes for which any of the three strategies provide better results?

T-tests applied to the data given in Table 3.4 show that DSSR is significantly better in 7 classes from R and R+ strategy, in 8 classes DSSR performed similarly to R+ but significantly higher than R, and in 2 classes DSSR performed similarly to R but significantly higher than R+. There is no case R and R+ strategy performed significantly better than DSSR strategy. Expressed in percentage: 72% of the classes do not show significantly different behaviours whereas in 28% of hte classes, the DSSR strategy performs significantly better than at least one of R and R+. It is interesting to note that in no single case R and R+ strategies performed better than DSSR strategy. We attribute this to DSSR possessing the qualities of R and R+ whereas containing the spot sweeping feature.

3.5.3 Can we pick the best default strategy between R, R+ and DSSR?

Analysis of the experimental data reveal that DSSR strategy has an edge over R and R+. This is because of the additional feature of Spot Sweeping in DSSR strategy.

In spite of the better performance of DSSR strategy compared to R and R+ strategies the present study does not provide ample evidence to pick it as the best default strategy because of the overhead induced by this strategy (see next section). Further study might give conclusive evidence.

3.6 Discussion

In this section we discuss various factors such as the time taken, effect of test duration, number of tests, number of faults in the different strategies and the effect of finding first fault in the DSSR strategy. **Time taken to execute an equal number of test cases:** The DSSR strategy takes slightly more time (up to 5%) than both pure random and random plus which may be due to maintaining sets of interesting values during the execution. We do not believe that the overhead can be reduced.

Effect of test duration and number of tests on the results: All three techniques have the same potential for finding failures. If testing is continued for a long duration then all three strategies will find the same number of unique failures and the results will converge. We suspect however that some of the unique failures will take an extremely long time to be found by using random or random+ only. Further experiments should confirm this point.

Effect of number of faults on results: We found that the DSSR strategy performs better when the number of faults is higher in the code. The reason seems to be that when there are more faults, their domains are more connected and DSSR strategy works better. Further studies might use historical data to pick the best strategy.

Dependence of DSSR strategy to find the first unique failure early enough: During the experiments we noticed that if a unique failure is not found quickly enough, there is no value added to the list of interesting values and then the test becomes equivalent to random+ testing. This means that better ways of populating failure-inducing values are needed for sufficient leverage to DSSR strategy. As an example, the following piece of code would be unlikely to fail under the current setting:

```
public void test(float value){
 if(value == 34.4445)   10/0;
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained from the constant pool in the class files of the SUT.

In the example above the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the unique failure right away.

DSSR strategy and coverage: Random strategies typically achieve high level of coverage [43]. It might also be interesting to compare R, R+ and DSSR with respect to the achieved coverage or even to use a DSSR variant that adds a new interesting value and its neighbours when a new branch is reached.

Threats to validity: As usual with such empirical studies, the present work might suffer from a non-representative selection of classes. The selection in the current study is however made through random process and objective criteria, therefore, it seems likely that it would be representative.

The parameters of the study might also have prompted incorrect results. But this is unlikely due to previous results on random testing [**?**].

3.7 Related Work

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [46**? ?**]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [**?**]. Some of the well known automated tools based on random strategy includes Jartege [41], Eclat [46], JCrasher [**?**], AutoTest [16; 17] and YETI [43**?**].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [7; 10; 12**? ?**]. Adaptive random testing (ART), Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror

ART and ART through dynamic partitioning increased the performance by reducing the overhead of ART. The main reason behind better performance of the strategies is that even spread of test input increases the chance of exploring the fault patterns present in the input domain.

A more recent research study [?] stresses on the effectiveness of data regeneration in close vicinity of the existing test data. Their findings showed up to two orders of magnitude more efficient test data generation than the existing techniques. Two major limitations of their study are the requirement of existing test cases to regenerate new test cases, and increased overhead due to "meta heuristics search" based on hill climbing algorithm to regenerate new data. In DSSR no pre-existing test cases are required because it utilises the border values from R+ and regenerate the data very cheaply in a dynamic fashion different for each class under test without any prior test data and with comparatively lower overhead.

The random+ (R+) strategy is an extension of the random strategy in which interesting values, beside pure random values, are added to the list of test inputs [33]. These interesting values includes border values which have high tendency of finding faults in the given SUT [?]. Results obtained with R+ strategy show significant improvement over random strategy [33]. DSSR strategy is an extension of R+ strategy which starts testing as R+ until a fault is found then it switches to spot sweeping.

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

In our experiments we selected projects from the Qualitas Corpus [?] which is a collection of open source java programs maintained for independent empirical research. The projects in Qualitas Corpus are carefully selected that spans across the whole set of java applications [? ? ?].

3.8 Conclusions

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases. We developed a new strategy named.

"DSSR strategy" as an extension of R+, based on the assumption that in a significant number of classes, failure domains are contiguous or located closely. The DSS strategy, a strategy which adds neighbouring values of the failure finding value to a list of interesting values, was implemented in the random testing tool YETI to test 60 classes, 30 times each, from Qualitas Corpus with each of the 3 strategies R, R+ and DSSR. The newly developed DSSR strategy uncovers more unique failures than both random and random+ strategies with a 5% overhead. We found out that for 7 (12%) classes DSSR was significantly better than both R+ and R, for 8 (13%) classes DSSR performed similarly to R+ and significantly better than R, while in 2 (3%) cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Overall, DSSR yields encouraging results and advocates to develop the technique further for settings in which it is significantly better than both R and R+ strategies.

Table 3.2: Name and versions of 32 Projects randomly selected from the Qualitas Corpus for the experiments

S. No	Project Name	Version	Size (MB)
1	apache-ant	1.8.1	59
2	antlr	3.2	13
3	aoi	2.8.1	35
4	argouml	0.30.2	112
5	artofillusion	281	5.4
6	aspectj	1.6.9	109.6
7	axion	1.0-M2	13.3
8	azureus	1	99.3
9	castor	1.3.1	63.2
10	cayenne	3.0.1	4.1
11	cobertura	1.9.4.1	26.5
12	colt	1.2.0	40
13	emma	2.0.5312	7.4
14	freecs	1.3.20100406	11.4
15	hibernate	3.6.0	733
16	hsqldb	2.0.0	53.9
17	itext	5.0.3	16.2
18	jasml	0.10	7.5
19	jmoney	0.4.4	5.3
20	jruby	1.5.2	140.7
21	jsXe	04_beta	19.9
22	quartz	1.8.3	20.4
23	sandmark	3.4	18.8
24	squirrel-sql	3.1.2	61.5
25	tapestry	5.1.0.5	69.2
26	tomcat	7.0.2	24.1
27	trove	2.1.0	18.2
28	velocity	1.6.4	27.1
29	weka	3.7.2	107
30	xalan	2.7.1	85.4
31	xerces	2.10.0	43.4
32	xmojo	5.0.0	15

DssrImprove.png

Figure 3.5: Improvement of DSSR strategy over Random and Random+ strategy.

Table 3.3: Complete results for R, R+ and DSSR. Results present Serial Number (S.No), Class Name, Line of Code (LOC), mean, maximum number of faults, minimum number of faults and relative standard deviation for each Random (R), Random+ (R+) and Dirt Spot Sweeping Random (DSSR) strategies.

S. No	Class Name	LOC	R				R+				DSSR			
			Mean	Max	Min	R-STD	Mean	Max	Min	R-STD	Mean	Max	Min	R-STD
1	ActionTranslator	709	96	96	96	0	96	96	96	0	96	96	96	0
2	AjTypeImpl	1180	80	83	79	0.02	80	83	79	0.02	80	83	79	0.01
3	**Apriori**	**292**	**3**	**4**	**3**	**0.10**	**3**	**4**	**3**	**0.13**	**3**	**4**	**3**	**0.14**
4	BitSet	575	9	9	9	0	9	9	9	0	9	9	9	0
5	CatalogManager	538	7	7	7	0	7	7	7	0	7	7	7	0
6	**CheckAssociator**	**351**	**7**	**8**	**2**	**0.16**	**6**	**9**	**2**	**0.18**	**7**	**9**	**6**	**0.73**
7	**Debug**	**836**	**4**	**6**	**4**	**0.13**	**5**	**6**	**4**	**0.12**	**5**	**8**	**4**	**0.19**
8	**DirectoryScanner**	**1714**	**33**	**39**	**20**	**0.10**	**35**	**38**	**31**	**0.05**	**36**	**39**	**32**	**0.04**
9	DiskIO	220	4	4	4	0	4	4	4	0	4	4	4	0
10	DOMParser	92	7	7	3	0.19	7	7	3	0.11	7	7	7	0
11	Entities	328	3	3	3	0	3	3	3	0	3	3	3	0
12	EntryDecoder	675	8	9	7	0.10	8	9	7	0.10	8	9	7	0.08
13	EntryComparator	163	13	13	13	0	13	13	13	0	13	13	13	0
14	Entry	37	6	6	6	0	6	6	6	0	6	6	6	0
15	Facade	3301	3	3	3	0	3	3	3	0	3	3	3	0
16	FileUtil	83	1	1	1	0	1	1	1	0	1	1	1	0
17	Font	184	12	12	11	0.03	12	12	11	0.03	12	12	11	0.02
18	FPGrowth	435	5	5	5	0	5	5	5	0	5	5	5	0
19	Generator	218	17	17	17	0	17	17	17	0	17	17	17	0
20	**Group**	**88**	**11**	**11**	**10**	**0.02**	**10**	**4**	**11**	**0.15**	**11**	**11**	**11**	**0**
21	HttpAuth	221	2	2	2	0	2	2	2	0	2	2	2	0
22	**Image**	**2146**	**13**	**17**	**7**	**0.15**	**12**	**14**	**4**	**0.15**	**14**	**16**	**11**	**0.07**
23	InstrumentTask	71	2	2	1	0.13	2	2	1	0.09	2	2	2	0
24	IntStack	313	4	4	4	0	4	4	4	0	4	4	4	0
25	ItemSet	234	4	4	4	0	4	4	4	0	4	4	4	0
26	Itextpdf	245	8	8	8	0	8	8	8	0	8	8	8	0
27	**JavaWrapper**	**513**	**3**	**2**	**2**	**0.23**	**4**	**4**	**3**	**0.06**	**4**	**4**	**3**	**0.05**
28	JmxUtilities	645	8	8	6	0.07	8	8	7	0.04	8	8	7	0.04
29	**List**	**1718**	**5**	**6**	**4**	**0.11**	**6**	**6**	**4**	**0.10**	**6**	**6**	**5**	**0.09**
30	NameEntry	172	4	4	4	0	4	4	4	0	4	4	4	0
31	**NodeSequence**	**68**	**38**	**46**	**30**	**0.10**	**36**	**45**	**30**	**0.12**	**38**	**45**	**30**	**0.08**
32	NodeSet	208	28	29	26	0.03	28	29	26	0.04	28	29	26	0.03
33	PersistentBag	571	68	68	68	0	68	68	68	0	68	68	68	0
34	PersistentList	602	65	65	65	0	65	65	65	0	65	65	65	0
35	PersistentSet	162	36	36	36	0	36	36	36	0	36	36	36	0
36	**Project**	**470**	**65**	**71**	**60**	**0.04**	**66**	**78**	**62**	**0.04**	**69**	**78**	**64**	**0.05**
37	**Repository**	**63**	**31**	**31**	**31**	**0**	**40**	**40**	**40**	**0**	**40**	**40**	**40**	**0**
38	Routine	1069	7	7	7	0	7	7	7	0	7	7	7	0
39	RubyBigDecimal	1564	4	4	4	0	4	4	4	0	4	4	4	0
40	Scanner	94	3	5	2	0.20	3	5	2	0.27	3	5	2	0.25
41	**Scene**	**1603**	**26**	**27**	**26**	**0.02**	**26**	**27**	**26**	**0.02**	**27**	**27**	**26**	**0.01**
42	SelectionManager	431	3	3	3	0	3	3	3	0	3	3	3	0
43	**Server**	**279**	**15**	**21**	**11**	**0.20**	**17**	**21**	**12**	**0.16**	**17**	**21**	**12**	**0.14**
44	**Sorter**	**47**	**2**	**2**	**1**	**0.09**	**3**	**3**	**2**	**0.06**	**3**	**3**	**3**	**0**
45	Sorting	762	3	3	3	0	3	3	3	0	3	3	3	0
46	**Statistics**	**491**	**16**	**17**	**12**	**0.08**	**23**	**25**	**22**	**0.03**	**24**	**25**	**22**	**0.04**
47	Status	32	53	53	53	0	53	53	53	0	53	53	53	0
48	**Stopwords**	**332**	**7**	**8**	**7**	**0.03**	**7**	**8**	**6**	**0.08**	**8**	**8**	**7**	**0.06**
49	**StringHelper**	**178**	**43**	**45**	**40**	**0.02**	**44**	**46**	**42**	**0.02**	**44**	**45**	**42**	**0.02**
50	StringUtils	119	19	19	19	0	19	19	19	0	19	19	19	0
51	TouchCollector	222	3	3	3	0	3	3	3	0	3	3	3	0
52	Trie	460	21	22	21	0.02	21	22	21	0.01	21	22	21	0.01
53	URI	3970	5	5	5	0	5	5	5	0	5	5	5	0
54	WebMacro	311	5	5	5	0	5	6	5	0.14	5	7	5	0.28
55	XMLAttributesImpl	277	8	8	8	0	8	8	8	0	8	8	8	0
56	XMLChar	1031	13	13	13	0	13	13	13	0	13	13	13	0
57	XMLEntityManger	763	17	18	17	0.01	17	17	16	0.01	17	17	17	0
58	XMLEntityScanner	445	12	12	12	0	12	12	12	0	12	12	12	0
59	XObject	318	19	19	19	0	19	19	19	0	19	19	19	0
60	**XString**	**546**	**23**	**24**	**21**	**0.04**	**23**	**24**	**23**	**0.02**	**24**	**24**	**23**	**0.02**
	Total	35,785	1040	1075	973	2.42	1061	1106	1009	2.35	1075	1118	1032	1.82

Table 3.4: T-test results of the classes showing different results

S. No	Class Name	T-test Results			Interpretation
		DSSR, R	DSSR, R+	R, R+	
1	AjTypeImpl	1	1	1	
2	Apriori	**0.03**	0.49	0.16	
3	CheckAssociator	**0.04**	**0.05**	0.44	DSSR better
4	Debug	**0.03**	0.14	0.56	
5	DirectoryScanner	**0.04**	**0.01**	0.43	DSSR better
6	DomParser	**0.05**	0.23	0.13	
7	EntityDecoder	**0.04**	0.28	0.3	
8	Font	0.18	0.18	1	
9	Group	0.33	**0.03**	**0.04**	DSSR = R ¿ R+
10	Image	**0.03**	**0.01**	0.61	DSSR better
11	InstrumentTask	0.16	0.33	0.57	
12	JavaWrapper	**0.001**	0.57	0.004	DSSR = R+ ¿ R
13	JmxUtilities	0.13	0.71	0.08	
14	List	**0.01**	0.25	**0**	DSSR = R+ ¿ R
15	NodeSequence	0.97	**0.04**	**0.06**	DSSR = R ¿ R+
16	NodeSet	**0.03**	0.42	0.26	
17	Project	**0.001**	0.57	**0.004**	DSSR better
18	Repository	**0**	1	**0**	DSSR = R+ ¿ R
19	Scanner	1	**0.03**	**0.01**	DSSR better
20	Scene	**0**	**0**	1	DSSR better
21	Server	**0.03**	0.88	**0.03**	DSSR = R+ ¿ R
22	Sorter	**0**	0.33	**0**	DSSR = R+ ¿ R
23	Statistics	**0**	0.43	**0**	DSSR = R+ ¿ R
24	Stopwords	**0**	0.23	**0**	DSSR = R+ ¿ R
25	StringHelper	**0.03**	0.44	0.44	DSSR = R+ ¿ R
26	Trie	0.1	0.33	0.47	DSSR better
27	WebMacro	0.33	1	0.16	
28	XMLEntityManager	0.33	0.33	0.16	
29	XString	0.14	**0.03**	0.86	

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. vi, 7

[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. 9

[4] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995. 8

[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. 11, 23, 26

[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, 10, 23, 26, 37

[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. 20

[9] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008. v, 2, 11, 23, 27, 33

[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, 13, 23, 26, 37

[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. 12

[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, 12, 14, 23, 26, 37

[13] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994. vi, 7

[14] 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. 10, 25

[15] 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. 15

[16] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of*

the 2007 international symposium on Software testing and analysis, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM. 1, 16, 19, 25, 34, 37

[17] 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. 10, 25, 37

[18] 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. 9, 10, 25, 26

[19] 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, 25

[20] 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

[21] Julie Cohen, Daniel Plakosh, and Kristi L Keeler. Robustness testing of software-intensive systems: Explanation and guide. 2005. 8

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

[23] 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. 10, 25

[24] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978. 9

[25] Marie-Claude Gaudel. Software testing based on formal specification. In *Testing Techniques in Software Engineering*, pages 215–242. Springer, 2010. vi, 7, 9

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

[27] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, dec 1990. 11

[28] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994. 10

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

[30] 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

[31] 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. 20

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

[33] 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. 9, 25, 26, 32, 38

[34] 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. 10

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

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

[37] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. 10

[38] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. 2, 6, 25

[39] Simeon Ntafos. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998. 11

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

[41] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. *CoRR*, abs/cs/0412012, 2004. 18, 25, 37

[42] 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

[43] Manuel Oriol and Faheem Ullah. Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:434–437, 2010. 21, 25, 31, 37

[44] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002. 8

[45] Carlos Pacheco. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009. 10

[46] 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. 18, 37

[47] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007. 15, 17

[48] 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, 23, 34

[49] 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. 14, 25

[50] 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. 9

[51] 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. vi, 7, 9

[52] 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. 10

[53] 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. vi, 7

[54] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 1999. 9

[55] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982. 10

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

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