New Strategies for Automated Random Testing

MIAN ASBAT AHMAD

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY of York
UNITED KINGDOM

17TH MARCH 2013

Abstract

This is the abstract text with the formula $v_L(t) = \int_{-\infty}^{t} \frac{di_L}{dt}$. This paper describes the computation of feature point correspondences using the spectra of a Hermitian property matrix. Firstly, a complex Laplacian (Hermitian) matrix is constructed from the Gaussian-weighted distances and the difference of SIFT angles between each pair of points in the two images to be matched. Matches are computed by comparing the complex eigenvectors of the Hermitian property matrices for the two point sets acquired from the two images. Secondly, we embed the complex modal structure within Carcassoni's iterative alignment method to render it more robust to rotation. Our method has been evaluated on both synthetic and real-world data.

Contents

List of Tables

List of Figures

Acknowledgements

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

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

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

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

Declaration

This thesis has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree other than Doctor of Philosophy of the University of York. This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by explicit references.

I hereby give consent for my thesis, if accepted, to be made available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed . . . . (candidate)

Date . . . .

Publications:

Some of the material contained in this thesis has appeared in the following published conference and workshop papers:

Kazakov, D. and Shahid, A. (2008). Extracting Multilingual Dictionaries for the teaching of CS and AI. In *4th UK Workshop on AI in Education* as part of the annual SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence, Cambridge, UK.

Kazakov, D. and Shahid, A. (2009). Unsupervised Construction of a Multilingual WordNet from Parallel Corpora. In *Workshop on Natural Language Processing methods and Corpora in Translation, Lexicography, and Language Learning (RANLP '09)*, Borovets, Bulgaria.

Shahid, A. and Kazakov, D. (2009). Automatic Multlingual Lexicon Generation using Wikipedia as a resource. In *Proceedings of the International Conference on Agents and Artificial Intelligence, (ICCART '09)*, Porto, Portugal.

Shahid, A. and Kazakov, D. (2010). Retrieving Lexical Semantics from Parallel Corpora. *Polibits, 5*, 25-28.

Shahid, A. and Kazakov, D. (2011). Using Multilingual Corpora to Extract Semantic Information. In *Proceedings of the Symposium on Learning Language Models from Multilingual Corpora, AISB'11 Convention*, York, UK.

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 reflected in my dissertation.

CHAPTER 1

Introduction

Software testing is the process of executing a software with specific test data followed by evaluation of the results to check whether it is working according to its specification or not [3]. The test passes if the output complies to its specification and fails otherwise. The success of testing correlates with the number of failures found in the Software Under Test (SUT): a test is more successful if it finds more faults. The primary focus of modern software companies is to achieve high quality in all phases of Software Development Life Cycle (SDLC) from requirements to implementation phase. Most companies spend 40 to 50 percent of the total project fund on testing [5]. It is interesting that program testing is used to show the presence of bugs, rather than absence of bugs [6]. Therefore the SUT that passes all the tests without returning a single failure does not guarantee that there is no fault. The testing process increases however the reliability and confidence of both the developers and the users in the tested product [7] [8] [9].

Random testing is a black-box testing technique in which the SUT is executed against ran- domly selected test data. Test results obtained are compared either against the oracle defined, using SUT specifications in the form of assertions or exceptions defined by the programming language. The rapid increase in software development in today?s modern world prompts the need for automated testing to ensure high quality. The generation of random test data is com- paratively cheap and does not require too much intellectual and computation efforts [10] [11]. It is for this reason that various researchers have recommended this strategy for incorporation in automatic testing tools [12]. YETI [13] [14], AutoTest [15] [16], QuickCheck [17], Randoop [18], JArtage [19] are a few of the most common tools based on random strategy.

1.1 Problem Description

1.1.1 Test input data is decisive

1.1.2 Selecting Fault finding input is challenging

1.2 Our Goals

1.3 Contributions

1.3.1 Dirt Spot Sweeping Random Strategy

1.3.2 Automated Discovery of Failure Domain

1.3.3 Directed Random Plus Strategy

1.4 Thesis Outline

CHAPTER 2

Literature Review

2.1 Software Testing

2.1.1 Categories of Software Testing

2.1.1.1 Black-box Testing

2.1.1.2 White-box Testing

2.1.2 Manual Testing

2.1.3 Automated Testing

2.1.3.1 Random Testing

2.1.3.2 Exhaustive Testing

2.2 Automated Random Testing

2.2.1 Test Data Generation

2.2.2 Test Execution

2.2.3 Test Oracle

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. Chan et al. (1996) 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. Chen (2008) 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 Chan et al. (2002), Feedback directed Random Test Generation Pacheco et al. (2007), Mirror Adaptive Random Testing Chen et al. (2003) and Quasi Random Testing Chen & Merkel (2005) 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 Oriol (2012) 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,

[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`

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 4.5 discusses the results. Section 3.7 presents related work and Section 3.8, concludes the study.

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 Chen et al. (2010). The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts Ciupa et al. (2009, 2008). It is mainly for this reason that various researchers have recommended random strategy for automated testing tools Ciupa et al. (2008). YETI Oriol & Tassis (2010); Oriol & Ullah (2010), AutoTest Leitner et al. (2007); Ciupa et al. (2007), QuickCheck Claessen & Hughes (2000a), Randoop Pacheco & Ernst (2007), JArtege Oriat (2004) 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 Myers & Sandler (2004) who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers, Ciupa et al. (2007); Duran & Ntafos (1981, 1984); Hamlet (1994); Ntafos (2001) 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 Ciupa et al.

(2007); Oriol (2012) 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 Leitner et al. (2007) 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 Beizer (1990) 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 Ciupa et al. (2008).

3.2.3 Dirt Spot Sweeping (DSS)

Chan et al. Chan et al. (1996) found that there are patterns of failure-causing inputs across the input domain. Figure 4.1 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 Chan et al. (2002); Chen et al. (2003); Chen & Merkel (2005), 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.

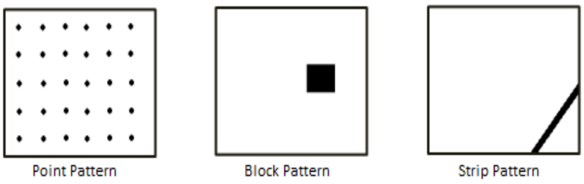


Figure 3.1: Failure patterns across input domain Chen (2008)

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.

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

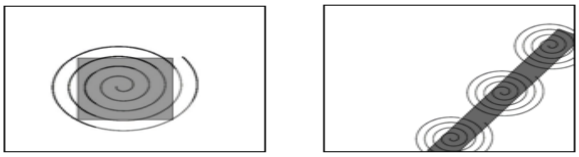


Figure 3.2: DSSR covering block and strip pattern

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 Leitner et al. (2009). 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.

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

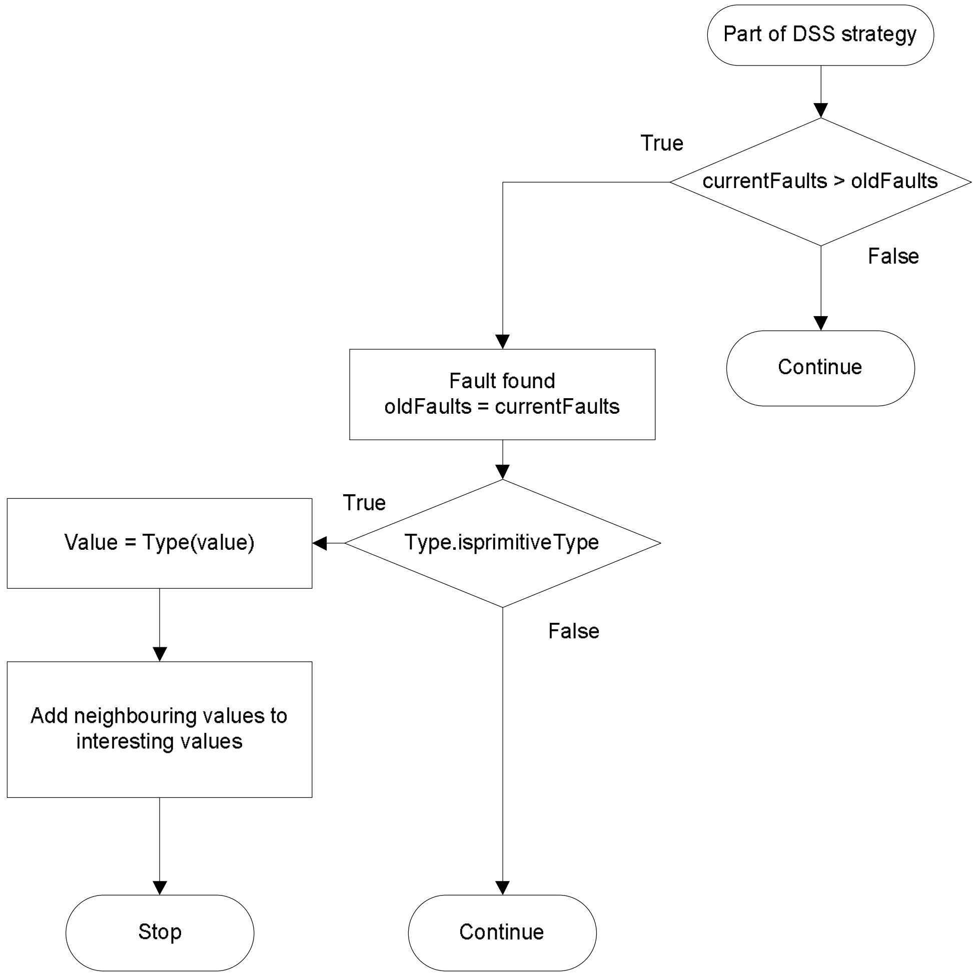


Figure 3.3: Working mechanism of DSSR Strategy

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.

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

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)

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 Oriol & Ullah (2010). A number of hitherto faults have successfully been found by YETI in various production softwares Oriol (2011, 2012).

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.

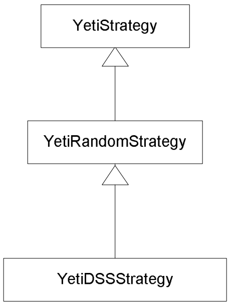


Figure 3.4: Class Hierarchy of DSSR in YETI

3.4 Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy Leitner et al. (2007). 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 Tempero et al. (2010). 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.[4] Because of the absence of the contracts and assertions in the code under test, Similar approach

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

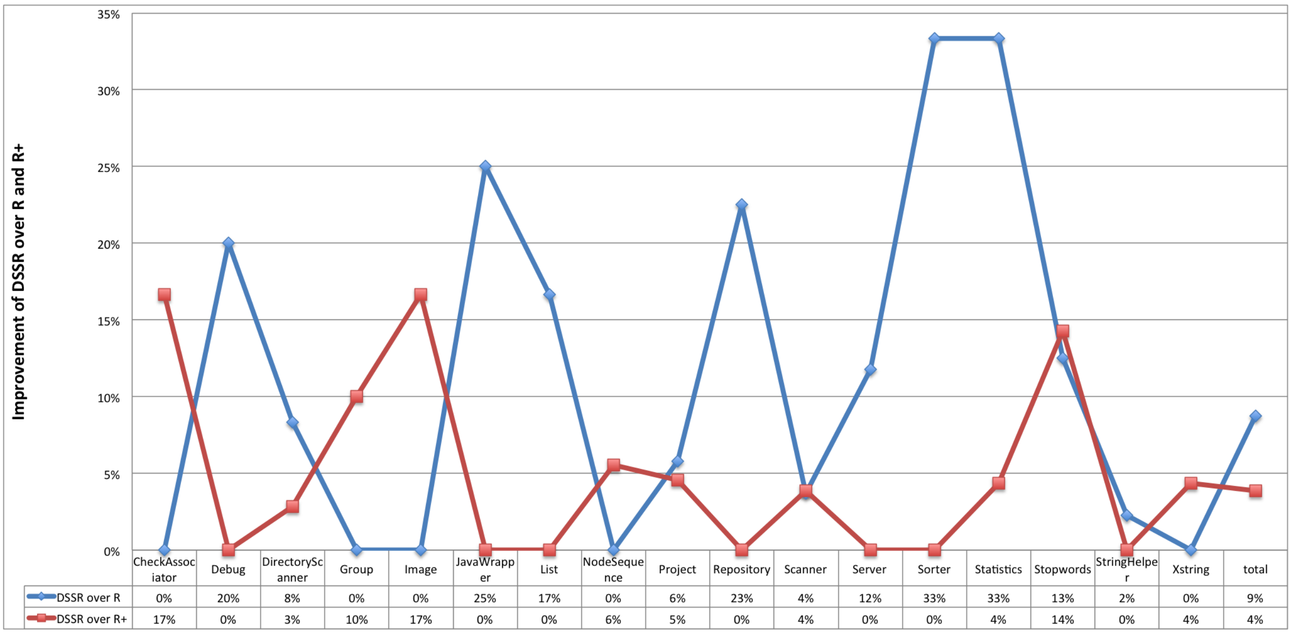
as used in previous studies Oriol (2012) is followed using undeclared exceptions to compute unique failures.

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™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 Chen (2008) and are not considered effective measuring techniques while the F-measure has been often used by various researchers Chen & Yu (1996); Chen et al. (2004). 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 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 Liu et al. (2012).

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 Pacheco et al. (2007); Ciupa et al. (2007, 2008). This measurement is effective because it considers the performance of the strategy when all other factors are kept constant.



	CheckAssociator	Debug	DirectoryScanner	Group	Image	JavaWrapper	List	NodeSequence	Project	Repository	Scanner	Server	Sorter	Statistics	Stopwords	StringHelper	Xstring	total
DSSR over R	0%	20%	8%	0%	0%	25%	17%	0%	6%	23%	4%	12%	33%	33%	13%	2%	0%	9%
DSSR over R+	17%	0%	3%	10%	17%	0%	0%	6%	5%	0%	4%	0%	0%	4%	14%	0%	4%	4%

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

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 Oriol & Ullah (2010). 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 Oriol (2012).

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 Pacheco & Ernst (2005); Csallner & Smaragdakis (2004); Claessen & Hughes (2000b). Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation Hamlet (1994). Some of the well known automated tools based on random strategy includes Jartege Oriat (2004), Eclat Pacheco & Ernst (2005), JCrasher Csallner & Smaragdakis (2004), AutoTest Ciupa et al. (2007, 2008) and YETI Oriol & Ullah (2010); Oriol (2012).

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed Chen et al. (2010); Chen & Merkel (2005); Chan et al. (2002); Chen et al. (2004, 2003). 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 Yoo & Harman (2012) 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 Leitner et al. (2007). These interesting values includes border values which have high tendency of finding faults in the given SUT Beizer (1990). Results obtained with R+ strategy show significant improvement over random strategy Leitner et al. (2007). 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 Gutjahr (1999); Duran & Ntafos (1984); Hamlet & Taylor (1990). Arcuri et al. Arcuri et al. (2012), 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 Tempero et al. (2010) 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 Oriol (2012); Tempero et al. (2010); Tempero (2008).

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

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+				DSS	
			Mean	Max	Min	R-STD	Mean	Max	Min	R-STD	Mean	Max
1	ActionTranslator	709	96	96	96	0	96	96	96	0	96	96
2	AjTypeImpl	1180	80	83	79	0.02	80	83	79	0.02	80	83
3	**Apriori**	**292**	**3**	**4**	**3**	**0.10**	**3**	**4**	**3**	**0.13**	**3**	**4**
4	BitSet	575	9	9	9	0	9	9	9	0	9	9
5	CatalogManager	538	7	7	7	0	7	7	7	0	7	7
6	**CheckAssociator**	**351**	**7**	**8**	**2**	**0.16**	**6**	**9**	**2**	**0.18**	**7**	**9**
7	**Debug**	**836**	**4**	**6**	**4**	**0.13**	**5**	**6**	**4**	**0.12**	**5**	**8**
8	**DirectoryScanner**	**1714**	**33**	**39**	**20**	**0.10**	**35**	**38**	**31**	**0.05**	**36**	**39**
9	DiskIO	220	4	4	4	0	4	4	4	0	4	4
10	DOMParser	92	7	7	3	0.19	7	7	3	0.11	7	7
11	Entities	328	3	3	3	0	3	3	3	0	3	3
12	EntryDecoder	675	8	9	7	0.10	8	9	7	0.10	8	9
13	EntryComparator	163	13	13	13	0	13	13	13	0	13	13
14	Entry	37	6	6	6	0	6	6	6	0	6	6
15	Facade	3301	3	3	3	0	3	3	3	0	3	3
16	FileUtil	83	1	1	1	0	1	1	1	0	1	1
17	Font	184	12	12	11	0.03	12	12	11	0.03	12	12
18	FPGrowth	435	5	5	5	0	5	5	5	0	5	5
19	Generator	218	17	17	17	0	17	17	17	0	17	17
20	**Group**	**88**	**11**	**11**	**10**	**0.02**	**10**	**4**	**11**	**0.15**	**11**	**11**
21	HttpAuth	221	2	2	2	0	2	2	2	0	2	2
22	**Image**	**2146**	**13**	**17**	**7**	**0.15**	**12**	**14**	**4**	**0.15**	**14**	**16**
23	InstrumentTask	71	2	2	1	0.13	2	2	1	0.09	2	2
24	IntStack	313	4	4	4	0	4	4	4	0	4	4
25	ItemSet	234	4	4	4	0	4	4	4	0	4	4
26	Itextpdf	245	8	8	8	0	8	8	8	0	8	8
27	**JavaWrapper**	**513**	**3**	**2**	**2**	**0.23**	**4**	**4**	**3**	**0.06**	**4**	**4**
28	JmxUtilities	645	8	8	6	0.07	8	8	7	0.04	8	8
29	**List**	**1718**	**5**	**6**	**4**	**0.11**	**6**	**6**	**4**	**0.10**	**6**	**6**
30	NameEntry	172	4	4	4	0	4	4	4	0	4	4
31	**NodeSequence**	**68**	**38**	**46**	**30**	**0.10**	**36**	**45**	**30**	**0.12**	**38**	**45**
32	NodeSet	208	28	29	26	0.03	28	29	26	0.04	28	29
33	PersistentBag	571	68	68	68	0	68	68	68	0	68	68
34	PersistentList	602	65	65	65	0	65	65	65	0	65	65
35	PersistentSet	162	36	36	36	0	36	36	36	0	36	36
36	**Project**	**470**	**65**	**71**	**60**	**0.04**	**66**	**78**	**62**	**0.04**	**69**	**78**
37	**Repository**	**63**	**31**	**31**	**31**	**0**	**40**	**40**	**40**	**0**	**40**	**40**
38	Routine	1069	7	7	7	0	7	7	7	0	7	7
39	RubyBigDecimal	1564	4	4	4	0	4	4	4	0	4	4
40	Scanner	94	3	5	2	0.20	3	5	2	0.27	3	5
41	**Scene**	**1603**	**26**	**27**	**26**	**0.02**	**26**	**27**	**26**	**0.02**	**27**	**27**
42	SelectionManager	431	3	3	3	0	3	3	3	0	3	3
43	**Server**	**279**	**15**	**21**	**11**	**0.20**	**17**	**21**	**12**	**0.16**	**17**	**21**
44	**Sorter**	**47**	**2**	**2**	**1**	**0.09**	**3**	**3**	**2**	**0.06**	**3**	**3**
45	Sorting	762	3	3	3	0	3	3	3	0	3	3
46	**Statistics**	**491**	**16**	**17**	**12**	**0.08**	**23**	**25**	**22**	**0.03**	**24**	**25**
47	Status	32	53	53	53	0	53	53	53	0	53	53
48	**Stopwords**	**332**	**7**	**8**	**7**	**0.03**	**7**	**8**	**6**	**0.08**	**8**	**8**
49	**StringHelper**	**178**	**43**	**45**	**40**	**0.02**	**44**	**46**	**42**	**0.02**	**44**	**45**
50	StringUtils	119	19	19	19	0	19	19	19	0	19	19
51	TouchCollector	222	3	3	3	0	3	3	3	0	3	3
52	Trie	460	21	22	21	0.02	21	22	21	0.01	21	22
53	URI	3970	5	5	5	0	5	5	5	0	5	5
54	WebMacro	311	5	5	5	0	5	6	5	0.14	5	7
55	XMLAttributesImpl	277	8	8	8	0	8	8	8	0	8	8
56	XMLChar	1031	13	13	13	0	13	13	13	0	13	13

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	

CHAPTER 4

Automated Discovery of Failure Domain Strategy

4.1 Introduction

Testing is fundamental requirement to assess the quality of any software. Manual testing is labour-intensive and error-prone; therefore emphasis is to use automated testing that significantly reduces the cost of software development process and its maintenance Beizer (1995). Most of the modern black-box testing techniques execute the System Under Test (SUT) with specific input and compare the obtained results against the test oracle. A report is generated at the end of each test session containing any discovered faults and the input values which triggers the faults. Debuggers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality, set in test plan, is achieved.

The fact that exhaustive testing for any non-trivial program is impossible, compels the testers to come up with some strategy of input selection from the whole input domain. Pure random is one of the possible strategies widely used in automated tools. It is intuitively simple and easy to implement Ciupa et al. (2008), Forrester & Miller (2000). It involves minimum or no overhead in input selection and lacks human bias Hamlet (1994), Linger (1993). While pure random testing has many benefits, there are some limitations as well, including low code coverage Offutt & Hayes (1996) and discovery of lower number of faults Chen & Yu (1994). To overcome these limitations while keeping its benefits intact many researchers successfully refined pure random testing. Adaptive Random Testing (ART) is the most significant refinements of random testing. Experiments performed using ART showed up to 50% better results compared to the traditional/pure random testing Chen (2008). Similarly Restricted Random Testing (RRT) Chan et al. (2002), Mirror Adaptive Random Testing (MART) Chen et al. (2004), Adaptive Random Testing for Object Oriented Programs (ARTOO) Ciupa et al. (2008), Directed Adaptive Random Testing (DART) Godefroid et al. (2005), Lattice-based Adaptive Random Testing (LART) Mayer (2005) and Feedback-directed Random Testing (FRT) Pacheco & Ernst (2007) are some of the variations of random testing aiming to increase the overall performance of pure random testing.

All the above-mentioned variations in random testing are based on the observation of Chan et. al., Chan et al. (1996) that failure causing inputs across the whole input domain form certain kinds of domains. They classified these domains into point, block and strip fault domain. In Figure 4.1 the square box represents the whole input domain. The black point, block and strip area inside the box represent the faulty values while white area inside the box represent legitimate values for a specific system. They further suggested that the fault finding ability of testing could be improved by taking into consideration these failure domains.

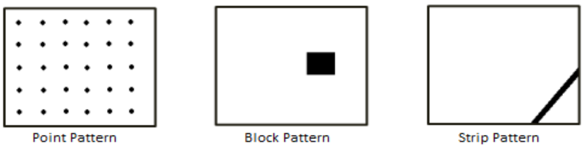


Figure 4.1: Failure domains across input domain Chan et al. (1996)

It is interesting that where many random strategies are based on the principle of contiguous fault domains inside the input domain, no specific strategy is developed to evaluate these fault domains. This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD), which not only finds the pass and fail input values but also finds their domains. The idea of identification of pass and fail domain is attractive as it provides an insight of the domains in the given SUT. Some important aspects of ADFD strategy presented in the paper include:

- Implementation of the new ADFD strategy in York Extensible Testing Infrastructure (YETI) tool.
- Evaluation to assess ADFD strategy by testing classes with different fault domains.
- Decrease in overall test duration by identification of all the fault domains instead of a single instance of fault.
- Increase in test efficiency by helping debugger to keep in view all the fault occurrences when debugging.

The rest of this paper is organized as follows:
Section 4.2 describes the ADFD strategy. Section 4.3 presents implementation

of the ADFD strategy. Section 4.4 explains the experimental results. Section 4.5 discusses the results. Section 4.6 presents the threats to validity. Section 4.7 presents related work and Section 4.8, concludes the study.

4.2 Automated Discovery of Failure Domain

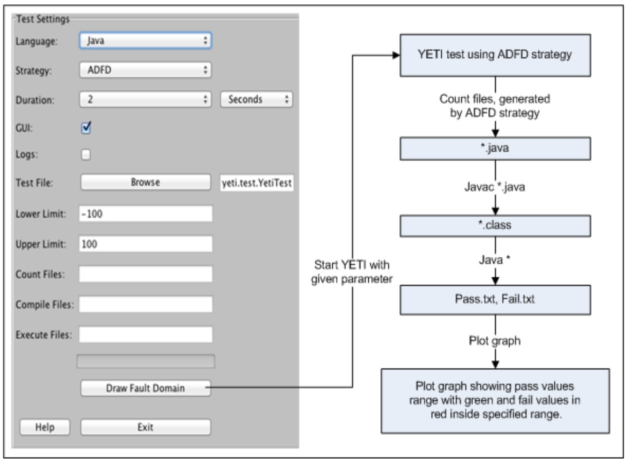
Automated Discovery of Failure Domain (ADFD) strategy is proposed as improvement on R+ strategy with capability of finding faults as well as the fault domains. The output produced at the end of test session is a chart showing the passing value or range of values in green and failing value or range of values in red. The complete workflow of ADFD strategy is given in Figure 4.3.

The process is divided into five major steps given below and each step is briefly explained in the following paras.

1. GUI front-end for providing input
2. Automated finding of fault
3. Automated generation of modules
4. Automated compilation and execution of modules to discover domains
5. Automated generation of graph showing domains

GUI front-end for providing input:

ADFD strategy is provided with an easy to use GUI front-end to get input from the user. It takes YETI specific input including language of the program, strategy, duration, enable or disable YETI GUI, logs and a program to test in the form of java byte code. In addition it also takes minimum and maximum values to search for fault domain in the specified range. Default range for minimum and



Test Settings

Language: Java

Strategy: ADFD

Duration: 2 Seconds

GUI: ☑

Logs: □

Test File: Browse yeti.test.YetiTest

Lower Limit: -100

Upper Limit: 100

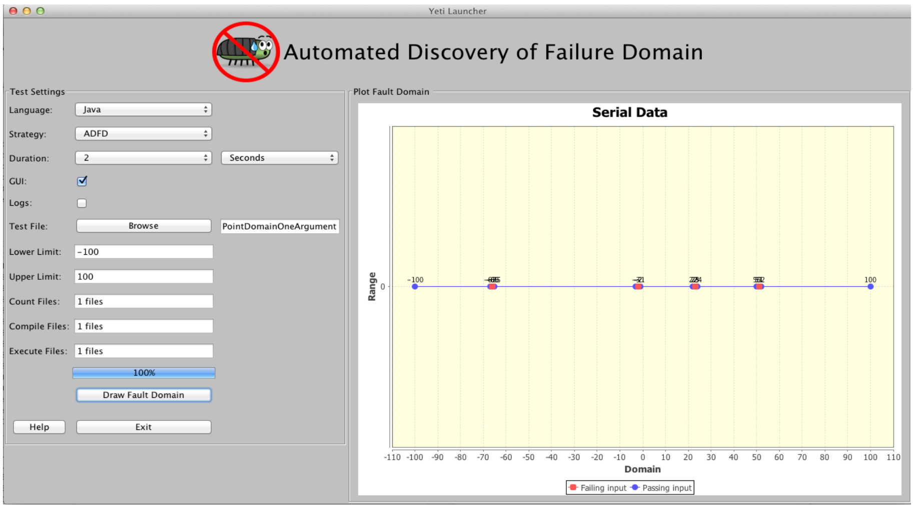
Count Files:

Compile Files:

Execute Files:

Draw Fault Domain
Help
Exit
Start YETI with given parameter
YETI test using ADFD strategy
Count files, generated by ADFD strategy
*.java
Javac *.java
*.class
Java *
Pass.txt, Fail.txt
Plot graph
Plot graph showing pass values range with green and fail values in red inside specified range.

Figure 4.2: Work flow of ADFD strategy



Yeti Launcher

Automated Discovery of Failure Domain

Test Settings

Language: Java

Strategy: ADFD

Duration: 2 Seconds

GUI: ☑

Logs: □

Test File: Browse PointDomainOneArgument

Lower Limit: -100

Upper Limit: 100

Count Files: 1 files

Compile Files: 1 files

Execute Files: 1 files

100%

Figure 4.3: Front-end of ADFD strategy

maximum is Integer.MIN_INT and Integer.MAX_INT respectively.

Automated finding of fault:

To find the failure domain for a specific fault, the first requirement is to identify that fault in the system. ADFD strategy extends R+ strategy and rely on R+ strategy to find the first fault. Random+ (R+) is an improvement over random strategy with preference to the boundary values to provide better fault finding ability. ADFD strategy is implemented in YETI tool which is famous for its simplicity, high speed and proven ability of finding potentially hazardous faults in many systems Oriol (2011), Oriol (2012). YETI is quick and can call up to one million instructions in one second on Java code. It is also capable of testing VB.Net, C, JML and CoFoJa beside Java.

Automated generation of modules:

After a fault is found in the SUT, ADFD strategy generate complete new Java program to search for fault domains in the given SUT. These programs with ".java" extensions are generated through dynamic compiler API included in Java 6 under javax.tools package. The number of programs generated can be one or more, depending on the number of arguments in the test module i.e. for module with one argument one program is generated, for two argument two programs and so on. To track fault domain the program keeps one or more than one argument constant and only one argument variable in the generated program.

Automated compilation and execution of modules to discover domains:

The java modules generated in previous step are compiled using "javac *" command to get their binary ".class" files. The "java *" command is applied to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument receive all the values in range, from minimum to maximum, specified in the beginning of the test. After execution is completed we get two text files of "Pass.txt" and "Fail.txt". Pass file contains all the values for which the modules behave correctly while fail file contains all the values for which the modules fail.

Automated generation of graph showing domains:

The values from the pass and fail files are used to plot (x, y) chart using JFreeChart. JFreeChart is a free open-source java library that helps developers to display complex charts and graphs in their applications Gilbert (2008). Green colour lines with circle represents pass values while red colour line with squares represents the fail values. Resultant graph clearly depicts both the pass and fail domain across the specified input domain. The graph shows red points in case the program fails for only one value, blocks when the program fails for multiple values and strips when a program fails for a long range of values.

4.3 Implementation

The ADFD strategy is implemented in a tool called York Extensible Testing Infrastructure (YETI). YETI is available in open-source at `http://code.google.com/p/yeti-test/`. In this section a brief overview of YETI is given with the focus on the parts relevant to the implementation of ADFD strategy. For integration of ADFD strategy in YETI, a program is used as an example to illustrate the working of ADFD strategy. Please refer to Oriol (2011), Oriol (2012), Oriol & Ullah (2010), Oriol & Tassis (2010), Oriol (2010) for more details on YETI tool.

4.3.1 York Extensible Testing Infrastructure

YETI is a testing tool developed in Java that test programs using random strategies in an automated fashion. YETI meta-model is language-agnostic which enables it to test programs written in functional, procedural and object-oriented languages.

YETI consists of three main parts including core infrastructure for extendibility through specialisation, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both the languages and strategies sections have a pluggable architecture to easily incorporate new strategies and languages making YETI a favourable choice to implement ADFD strategy. YETI is also capable of generating test cases to reproduce the faults found during the test session.

4.3.2 ADFD strategy in YETI

The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. On top of the hierarchy in strategies, is an abstract class YetiStrategy, which is extended by YetiRandomPlusStrategy and it is further extended to get ADFD strategy.

4.3.3 Example

For a concrete example to show how ADFD strategy in YETI proceeds, we suppose YETI tests the following class with ADFD strategy selected for testing. Note that for more clear visibility of the output graph generated by ADFD strategy at the end of test session, we fix the values of lower and upper range by 70 from Integer.MIN_INT and Integer.MAX_INT.

```
/**
 * Point Fault Domain example for one argument
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{
    public static void pointErrors (int x){
            if (x == -66)
                abort();

            if (x == -2)
                abort();

            if (x == 51)
                abort();

            if (x == 23)
                abort();
    }
}
```

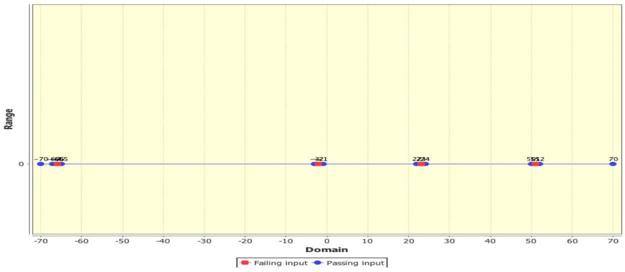


Figure 4.4: ADFD strategy plotting pass and fault domain of the given class

As soon as any one of the above four faults are discovered the ADFD strategy generate a dynamic program given in Appendix **??** (1). This program is automatically compiled to get binary file and then executed to find the pass and fail domains inside the specified range. The identified domains are plotted on two-dimensional graph. It is evident from the output presented in Figure 4.4 that ADFD strategy not only finds all the faults but also the pass and fail domains.

4.4 Experimental Results

This section includes the experimental setup and results obtained after using ADFD strategy. Six numerical programs of one and two-dimension were selected. These programs were error-seeded in such a way to get all the three forms of fault domains including point, block and strip fault domains. Each selected program contained various combinations of one or more fault domains.

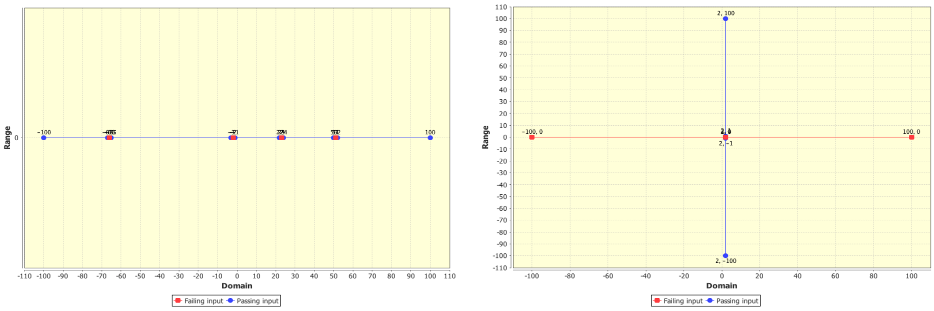
All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java™SE Runtime Environment [version 1.6.0_35].

To elucidate the results, six programs were developed so as to have separate program for one and two-dimension point, block and strip fault domains. The code of selected programs is given in Appendix **??** (2-7). The experimental results are presented in table **??** and described under the following three headings.

S. No	Fault Domain	Module Dimension	Specific Fault	Pass Domain	Fail Domain
1	Point	One	PFDOneA(i)	-100 to -67, -65 to -3, -1 to 50, 2 to 22, 24 to 50, 52 to 100	-66, -2, 23, 51
		Two	PFDTwoA(2, i)	(2, 100) to (2, 1), (2, -1) to (2, -100)	(2, 0)
			PFDTwoA(i, 0)	Nil	(-100, 0) to (100, 0)
2	Block	One	BFDOneA(i)	-100 to -30, -25 to -2, 2 to 50, 55 to 100	-1 to 1, -29 to -24, 51 to 54,
		Two	BFDTwoA(-2, i)	(-2, 100) to (-2, 20), (-2, -1) to (-2, -100)	(-2 , 1) to (-2, 19), (-2, 0)
			BFDTwoA(i, 0)	Nil	(-100, 0) to (100, 0)
3	Strip	One	SFDOneA(i)	-100 to -5, 35 to 100	-4, 34
		Two	SFDTwoA(-5, i)	(-5, 100) to (-5, 40), (-5, 0) to (-5, -100)	(-5, 39) to (-5, 1), (-5, 0)
			SFDTwoA(i, 0)	Nil	(-100, 0) to (100, 0)

Table 4.1: Pass and Fail domain with respect to one and two dimensional program

Point Fault Domain: Two separate Java programs Pro2 and Pro3 given in Appendix **??** (2, 3) were tested with ADFD strategy in YETI to get the findings for point fault domain in one and two-dimension program. Figure 4.5(a) present range of pass and fail values for point fault domain in one-dimension whereas Figure 4.5(b) present range of pass and fail values for point fault domain in two-dimension program. The range of pass and fail values for each program in point fault domain are given in (Table 4.1, Serial No. 1).

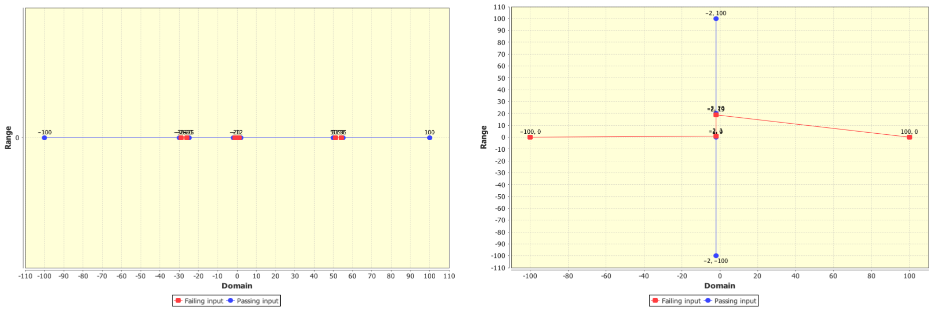


(a) One dimension module

(b) Two dimension module

Figure 4.5: Chart generated by ADFD strategy presenting point fault domain

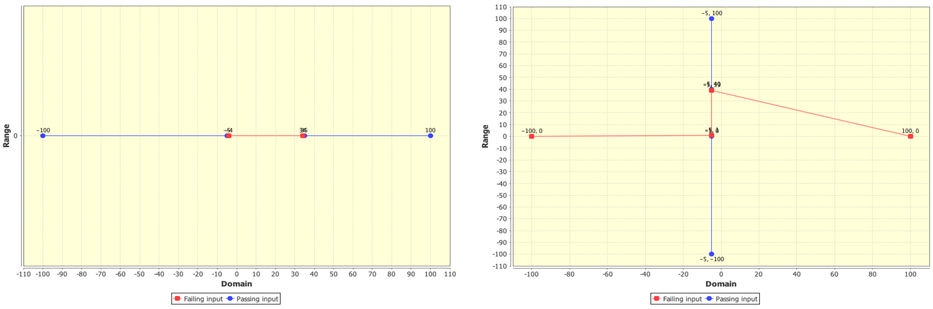
Block Fault Domain: Two separate Java programs Pro4 and Pro5 given in Appendix **??** (4, 5) were tested with ADFD strategy in YETI to get the findings for block fault domain in one and two-dimension program. Figure 4.6(a) present range of pass and fail values for block fault domain in one-dimension whereas Figure 4.6(b) present range of pass and fail values for block fault domain in two-dimension program. The range of pass and fail values for each program in block fault domain are given in (Table 4.1, Serial No. 2).



(a) One dimension module (b) Two dimension module

Figure 4.6: Chart generated by ADFD strategy presenting block fault domain

Strip Fault Domain: Two separate Java programs Pro6 and Pro7 given in Appendix **??** (6, 7) were tested with ADFD strategy in YETI to get the findings for strip fault domain in one and two-dimension program. Figure 4.7(a) present range of pass and fail values for strip fault domain in one-dimension whereas Figure 4.7(b) present range of pass and fail values for strip fault domain in two-dimension program. The range of pass and fail values for each program in strip fault domain are given in (Table 4.1, Serial No. 3).



(a) One dimension module (b) Two dimension module

Figure 4.7: Chart generated by ADFD strategy presenting Strip fault domain

4.5 Discussion

ADFD strategy with a simple graphical user interface is a fully automated process to identify and plot the pass and fault domains on the chart. Since the default settings are all set to optimum the user needs only to specify the module to be tested and click “Draw fault domain” button to start test execution. All the steps including Identification of fault, generation of dynamic java program to find domain of the identified fault, saving the program to a permanent media, compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automated without any human intervention.

In the experiments (section 4.4), the ADFD strategy effectively identified faults and faults domain in a program. Identification of fault domain is simple for one and two dimension numerical program but the difficulty increases as the program dimension increases beyond two. Similarly no clear boundaries are defined for non-numerical data therefore it is not possible to plot domains for non-numerical data unless some boundary criteria is defined.

ADFD strategy initiate testing with random+ strategy to find the fault and later switch to brute-force strategy to apply all the values between upper and lower bound for finding pass and fault domain. It is found that faults at boundary of the input domain can pass unnoticed through ordinary random test strategy but not from ADFD strategy as it scan all the values between lower and upper range.

The overhead in terms of execution time associated with ADFD strategy is dependent mainly on the lower and upper bound. If the lower and upper bound is set to maximum range (i.e. minimum for int is Integer.MIN_INT and maximum Integer.MAX_INT) then the test duration is maximum. It is rightly so because for identification of fault domain the program is executed for every input available in the specified range. Similarly increasing the range also shrinks the produced graph making it difficult to identify clearly point, block and strip domain unless they are of considerable size. Beside range factor, test duration is also influenced by the identification of the fault and the complexity of module under test.

ADFD strategy can help the debuggers in two ways. First, it reduces the to and from movement of the project between the testers and debuggers as it identity all the faults in one go. Second, it identifies locations of all fault domains across the input domain in a user-friendly way helping debugger to fix the fault keeping in view its all occurrences.

4.6 Threats to Validity

The major external threat to the use of ADFD strategy on commercial scale is the selection of small set of error-seeded programs of only primitive types such as integer used in the experiments. However, the present study will serve as foundation for future work to expand it to general-purpose real world production application containing scalar and non-scalar data types.

Another issue is the easy plotting of numerical data in the form of distinctive units, because it is difficult to split the composite objects containing many fields into units for plotting. Some work has been done to quantify composite objects into units on the basis of multiple featuresCiupa et al. (2006),to facilitate easy plotting. Plotting composite objects is beyond the scope of the present study. However, further studies are required to look in to the matter in depth.

Another threat to validity includes evaluating program with complex and more than two input arguments. ADFD strategy has so far only considered scalar data of one and two-dimensions. However, plotting domain of programs with complex non-scalar and more than two dimension argument is much more complicated and needs to be taken up in future studies.

Finally, plotting the range of pass or fail values for a large input domain (Integer.MIN_INT to Integer.MAX_INT) is difficult to adjust and does not give a clearly understandable view on the chart. Therefore zoom feature is added to the strategy to zoom into the areas of interest on the chart.

4.7 Related Works

Traditional random testing is quick, easy to implement and free from any bias. In spite of these benefits, the lower fault finding ability of traditional random testing is often criticised Offutt & Hayes (1996), Myers et al. (2011). To overcome the performance issues without compromising on its benefits, various researchers have altered its algorithm as explained in section 1. Most of the alterations are based on the existence of faults and fault domains across the input domain Chan et al. (1996).

Identification, classification of pass and fail domains and visualisation of domains have not received due attention of the researchers. Podgurski et. al., Podgurski et al. (2003) proposed a semi-automated procedure to classify similar faults and plot them by using a Hierarchical Multi Dimension Scaling (HMDS) algorithm. A tool named Xslice Agrawal et al. (1995) visually differentiates the execution slices of passing and failing part of a test. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite Jones et al. (2002). A serious limitation of the above mentioned tools is that they are not fully automated and require human interaction during execution. Moreover these tools are based on the already existing test cases where as ADFD strategy generate test cases, discover faults, identify pass and fault domains and visualise them in a fully automated manner.

4.8 Conclusion

Results of the experiments (section 4), based on applying ADFD strategy to error-seeded numerical programs provide, evidence that the strategy is highly effective in identifying the faults and plotting pass and fail domains of a given SUT. It further suggests that the strategy may prove effective for large programs. However, it must be confirmed with programs of more than two-dimension and different non-scalar argument types. ADFD strategy can find boundary faults quickly as against the traditional random testing, which is either, unable or takes comparatively long time to discover the faults.

The use of ADFD strategy is highly effective in testing and debugging. It provides an easy to understand test report visualising pass and fail domains. It reduces the number of switches of SUT between testers and debuggers because all the faults are identified after a single execution. It improves debugging efficiency as the debuggers keep all the instances of a fault under consideration when debugging the fault.

CHAPTER 5

Directed Random Plus Strategy

CHAPTER 6

Conclusion

References

Agrawal, H., Horgan, J., London, S., & Wong, W. (1995). Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, (pp. 143 –151).

Arcuri, A., Iqbal, M. Z., & Briand, L. (2012). Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering*, *38*, 258–277.

Beizer, B. (1990). *Software testing techniques (2nd ed.)*. New York, NY, USA: Van Nostrand Reinhold Co.

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley.

Chan, F., Chen, T., Mak, I., & Yu, Y. (1996). Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, *38*(12), 775 – 782.

Chan, K. P., Chen, T. Y., & Towey, D. (2002). Restricted random testing. In *Proceedings of the 7th International Conference on Software Quality*, ECSQ

’02, (pp. 321–330)., London, UK, UK. Springer-Verlag.

Chen, T., Merkel, R., Wong, P., & Eddy, G. (2004). Adaptive random testing through dynamic partitioning. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, (pp. 79 – 86).

Chen, T. & Yu, Y. (1994). On the relationship between partition and random testing. *Software Engineering, IEEE Transactions on*, *20*(12), 977 –980.

Chen, T. & Yu, Y. (1996). On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, *22*(2), 109 –119.

Chen, T. Y. (2008). Adaptive random testing. *Eighth International Conference on Qualify Software*, *0*, 443.

Chen, T. Y., Kuo, F.-C., & Merkel, R. (2004). On the statistical properties of the f-measure. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, (pp. 146 – 153).

Chen, T. Y., Kuo, F. C., Merkel, R. G., & Ng, S. P. (2003). Mirror adaptive random testing. In *Proceedings of the Third International Conference on Quality Software*, QSIC ’03, (pp.4̃)., Washington, DC, USA. IEEE Computer Society.

Chen, T. Y., Kuo, F.-C., Merkel, R. G., & Tse, T. H. (2010). Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, *83*, 60–66.

Chen, T. Y. & Merkel, R. (2005). Quasi-random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE ’05, (pp. 309–312)., New York, NY, USA. ACM.

Ciupa, I., Leitner, A., Oriol, M., & Meyer, B. (2006). 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, (pp. 55–63)., New York, NY, USA. ACM.

Ciupa, I., Leitner, A., Oriol, M., & Meyer, B. (2007). Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, (pp. 84–94)., New York, NY, USA. ACM.

Ciupa, I., Leitner, A., Oriol, M., & Meyer, B. (2008). Artoo: adaptive random testing for object-oriented software. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (pp. 71–80)., New York, NY, USA. ACM.

Ciupa, I., Meyer, B., Oriol, M., & Pretschner, A. (2008). Finding faults: Manual testing vs. random+ testing vs. user reports. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, (pp. 157–166)., Washington, DC, USA. IEEE Computer Society.

Ciupa, I., Pretschner, A., Leitner, A., Oriol, M., & Meyer, B. (2008). On the predictability of random tests for object-oriented software. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, (pp. 72–81)., Washington, DC, USA. IEEE Computer Society.

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

Claessen, K. & Hughes, J. (2000a). Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, (pp. 268–279)., New York, NY, USA. ACM.

Claessen, K. & Hughes, J. (2000b). Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, *35*(9), 268–279.

Csallner, C. & Smaragdakis, Y. (2004). Jcrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, *34*(11), 1025–1050.

Duran, J. W. & Ntafos, S. (1981). A report on random testing. In *Proceedings* *of the 5th international conference on Software engineering*, ICSE '81, (pp. 179–183)., Piscataway, NJ, USA. IEEE Press.

Duran, J. W. & Ntafos, S. C. (1984). An evaluation of random testing. *Software Engineering, IEEE Transactions on*, *SE-10*(4), 438 –444.

Forrester, J. E. & Miller, B. P. (2000). An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, (pp. 6–6)., Berkeley, CA, USA. USENIX Association.

Gilbert, D. (2008). *The JFreeChart class library version 1.0.9: Developer's guide*. Hertfordshire: Refinery Limited.

Godefroid, P., Klarlund, N., & Sen, K. (2005). Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, (pp. 213–223). ACM.

Gutjahr, W. (1999). Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, *25*(5), 661 –674.

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

Hamlet, R. (1994). Random testing. In *Encyclopedia of Software Engineering*, (pp. 970–978). Wiley.

Jones, J. A., Harrold, M. J., & Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (pp. 467–477)., New York, NY, USA. ACM.

Leitner, A., Ciupa, I., Meyer, B., & Howard, M. (2007). Reconciling manual and automated testing: The autotest experience. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, (pp. 261a–)., Washington, DC, USA. IEEE Computer Society.

Leitner, A., Pretschner, A., Mori, S., Meyer, B., & Oriol, M. (2009). On the effectiveness of test extraction without overhead. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, (pp. 416–425)., Washington, DC, USA. IEEE Computer Society.

Linger, R. C. (1993). Cleanroom software engineering for zero-defect software. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, (pp. 2–13)., Los Alamitos, CA, USA. IEEE Computer Society Press.

Liu, H., Kuo, F.-C., & Chen, T. Y. (2012). Comparison of adaptive random testing and random testing under various testing and debugging scenarios. *Software: Practice and Experience*, *42*(8), 1055–1074.

Mayer, J. (2005). Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, (pp. 333–336). ACM.

Myers, G. J. & Sandler, C. (2004). *The Art of Software Testing*. John Wiley & Sons.

Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. Wiley.

Ntafos, S. C. (2001). On comparisons of random, partition, and proportional partition testing. *IEEE Trans. Softw. Eng.*, *27*, 949–960.

Offutt, A. J. & Hayes, J. H. (1996). A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, *21*(3), 195–200.

Oriat, C. (2004). Jartege: a tool for random generation of unit tests for java classes. *CoRR*, *abs/cs/0412012*.

Oriol, M. (2010). The york extensible testing infrastructure (yeti).

Oriol, M. (2011). York extensible testing infrastructure.

Oriol, M. (2012). Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, (pp. 201 –210).

Oriol, M. & Tassis, S. (2010). Testing .net code with yeti. In *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ICECCS '10, (pp. 264–265)., Washington, DC, USA. IEEE Computer Society.

Oriol, M. & Ullah, F. (2010). Yeti on the cloud. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, *0*, 434–437.

Pacheco, C. & Ernst, M. D. (2005). Eclat: Automatic generation and classification of test inputs. In *In 19th European Conference Object-Oriented Programming*, (pp. 504–527).

Pacheco, C. & Ernst, M. D. (2007). Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM.

Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, (pp. 75–84)., Washington, DC, USA. IEEE Computer Society.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., & Wang, B. (2003). Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, (pp. 465 – 475).

Tempero, E. (2008). An empirical study of unused design decisions in open source java software. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, (pp. 33 –40).

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., & Noble, J. (2010). Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*

(APSEC2010).

Tempero, E., Counsell, S., & Noble, J. (2010). An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, (pp. 3–12)., Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Yoo, S. & Harman, M. (2012). Test data regeneration: generating new test data from existing test data. *Softw. Test. Verif. Reliab.*, *22*(3), 171–201.