Dirt Spot Sweeping Random Strategy

Mian Asbat Ahmad
Department of Computer Science
University of York
York, United Kingdom
Email: mian.ahmad@york.ac.uk

Manuel Oriol
ABB Corporate Research
Industrial Software Systems
Baden-Dattwil, Switzerland
Email: manuel.oriol@ch.abb.com

***Abstract*—While random testing has recently gained momentum, very little has been known on failure domains and their shape. This paper presents an enhanced and improved form of automated random testing, called the Dirt Spot Sweeping Random (DSSR) strategy. DSSR is a new strategy that takes the assumption that a number of failure domains are contiguous. DSSR starts as a regular random+ testing session — a random testing session with some preference for boundary values. When a failure is found, it increases the chances of using neighbouring values in subsequent tests, thus slowly sweeping around the failure found in hope of finding failures from a different kind in its vicinity.**

DSSR was implemented within the YETI random testing tool. We evaluate DSSR against random+ and pure random strategies by testing 80 classes with 10^5 calls for each session 30 times for each strategy. Our findings are that in 68% of the cases all three strategies all find the same faults, in 10% of cases random+ perfroms better, in 12% pure random performs better, and 10% DSSR performed better. Overall, DSSR also found 5% more failures than other strategies.

I. Introduction

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 amount of time. Exhaustive testing where software is tested against all possible inputs is, in most cases, not feasible because of the 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 the input domain for the 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. In another study, Chen et al. [1] 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 [2], Feedback directed Random Test Generation [3], Mirror Adaptive Random Testing [4] and Quasi Random Testing [5] also stress on the need of test case selection covering the whole input domain to improve results.

In this paper we take the assumption that in 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 strategy (DSSR) which starts as a random+ strategy — random strategy focusing more on boundary values. When it finds a new failure, it then increases the chances of testing using neighbouring values. Of course, since this strategy is also a random testing strategy, it has the potential to find all failures in the program, but we expect it to be faster at finding unique failures for classes in which failure domains are contiguous than the pure random (R) and random+ (R+) strategies.

We implemented DSSR as a strategy for the random testing tool YETI[2]. To evaluate our approach, we tested thirty times each one of 80 classes from the Qualitas Corpus[3] with each of the three strategies DSSR, R, and R+. We found that in 68% of the cases all three strategies all find the same faults, in 10% of cases random+ perfroms better, in 12% pure random performs better, and 10% DSSR performed better. Overall, DSSR also found 5% more failures than other strategies.

The rest of this paper is organized as follows:
Section II describes the DSSR strategy. Section II presents our implementation of the strategy. Section IV explains our experimental setup. Section V shows the results of our experiments. Section VI discusses the results. Section VII presents related work and we conclude in Section VIII.

II. Dirt Spot Sweeping Random Strategy

The Dirt Spot Sweeping Random (DSSR) strategy is a new strategy which combines the random+ strategy and with a dirt spot sweeping dunctionality. The strategy is based on two intuitions. One is that boundaries have interesting values and using these values in isolation can provide high impact on test results, the other is that faults reside in contiguous blocks and stripes. If this is the case, DSSR increases the performance of the test strategy in terms of executing fewer numbers of test

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

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

[3]http://...

cases with higher number of faults. Each strategy is briefly explained as follows.

A. Random (R)

The pure random strategy is a black-box testing technique in which the system under test (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 oracle. 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 [8]. The generation of random test data is comparatively cheap and does not require too much intellectual and computation efforts [9], [10]. It is mainly for this reason that various researchers have recommended this strategy for incorporation in automatic testing tools [11]. YETI [12], [13], AutoTest [14], [15], QuickCheck [16], Randoop [17], JArtage [18] 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 [19] who termed random testing as one of the poorest methods for software testing. however, experiments performed by various researchers, [15], [20], [21], [22] and [23] have experimentally proved that random testing is simple to implement, cost effective, highly efficient and free from human bias as compared to its rival techniques.

Because programs tested at random typically fail a large number of times (there are a large number of calls), 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 [15], [26] called a unique failures. This way of grouping failures is also used in random+ and DSSR.

B. Random Plus Strategy

The random+ strategy [14] is an extension of the pure 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 [7] 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`, `MIN_INT`, `MIN_INT+1`, `-1`, `0`, `1`.

Similarly the tester might also add some other special values that he considers effective in finding faults for the current 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, 45 to 55 etc. to the pre-defined list of special values in order to be selected for a test. 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 it is more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results particularly detecting problems in specifications [10].

C. Dirt Spot Sweeping

Chan et al. [6] found that there are patterns of failure-causing inputs across the input domain. Figure 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 also represents the boundary values. They also argue that a strategy has more chances of hitting these fault patterns if test cases far away from each other are selected. Other researchers, [2], [4] and [5], also tried to generate test cases further away from one another targeting these patterns and achieved higher performance.

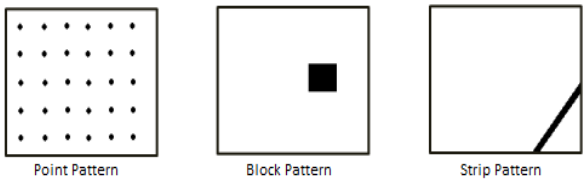

Fig. 1. Failure patterns across input domain [1]

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

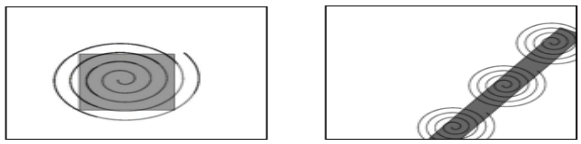

Fig. 2. DSSR covering block and strip pattern

Figure 2 shows how spot sweeping explores the faults residing in the block and strip patterns of a program. The faults coverage from the pattern is shown in spiral form because first fault will lead to second, second to third and so on till the end. In case the fault is positioned on the point pattern then the added values will not be very effective because point pattern is only an arbitrary fault point in the whole input domain.

D. Structure of Dirt Spot Sweeping Random Strategy

The DSSR strategy is explained with the help of flow-chart in Figure 3. In this process the strategy continuously

track the number of faults during the execution of the test session. To keep the system fast this tracking is done in a very effective way with zero or minimum overhead [24]. Execution of test is performed normally until a fault is found in the SUT. Then the program not only copy the value that lead to the fault, but also copy its surrounding values to the variable list of interesting values. It is evident from the flow-chart that if the fault finding value is of primitive type then the DSSR finds the type of the value and add values only of that particular type to the interesting values. Addition of these values increases the size of the list of interesting values that provide relevant test data for the remaining test session and the new generated test cases are more targeted towards finding new faults in the given SUT.

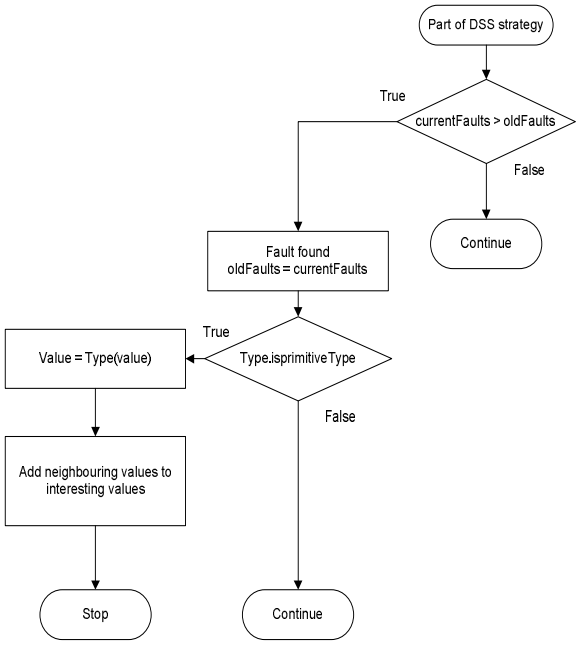

Fig. 3. Working mechanism of DSSR Strategy

Border values and other special values that have high tendency of finding faults in the SUT are added to the list by random plus strategy prior to the start of test session where as to sweep the failure pattern, fault-finding value and its surrounding values are added at run time after a fault is found. Table I contain the values that are added to the list of interesting values when a fault is found. In the table 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 to the list of interesting values and vice versa.

E. Explanation of DSSR with an example program

The concept of DSSR strategy is explained through a simple program seeded with at least three faults. The first is division by zero exception denoted by 1 while the second and third are in the form of assertion statements denoted by 2 and 3 in the following program. Below we describe how DSSR strategy perform execution when the following class is expose to testing.

Type	Values to be added
X is int, double, float, long, byte, short & char	X, X+1, X-1
X is String	X X + " " " " + X X.toUpperCase() X.toLowerCase() X.trim() X.substring(2) X.substring(1, X.length()-1)

TABLE I: Neighbouring values for primitive types and String

```
/**
*  Calculate square of given number and verify results.
*  Code contain 3 faults.
*  @author (Mian and Manuel)
*  @version (1.1, 11/07/12)
*/

public class Math1 {
    public void calc (int num1) {

    // Square num1 and store result.
    int result1 = num1 * num1;

    int result2 = result1 / num1;..... Fault 1

    assert Math.sqrt(result1) == num1;..... Fault 2

    assert result1 >= num1;..... Fault 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 some values like 0, Integer.MIN_VALUE and Integer.MAX_VALUE as interesting values which are prioritized for selection as test values. As the test starts, three faults are quickly discovered by DSSR strategy in the following order.

Fault 1: The DSSR strategy might 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 for selection is higher than other values. This will cause Java to generate division by zero exception because any integer divided by zero is infinity.

Fault 2: After catching the first fault, the strategy adds it and its surrounding values to the list of interesting values which includes 0, 1, 2, 3 and -1, -2, -3 in this case. In the second test case DSSR strategy may pick -3 as a test value and lead to the second fault where assertion (2) will fail because the square root of 9 will be +3 instead of input value -3.

Fault 3: After few tests DSSR strategy may select Integer.MAX_VALUE for variable "num1" from the list of interesting values which will lead to the 3rd fault because result1

will not be able to store the square of Integer.MAX_VALUE. Instead of the actual square value Java will assign a negative value (Java language rule) to variable result1 that will lead to the violation of next assertion (3).

The above execution process explains that the pre-defined values including border values, fault-finding values and the surrounding values lead us to the available faults quickly and in small number of tests as compared to Random and Random plus strategy. Random and Random plus takes longer to discover the second and third fault because they again start searching for new faults randomly although the remaining faults are very close to the first one.

III. Implementation of DSSR strategy

To avoid reinventing the wheel, an existing open-source automated random testing tool called YETI [25], [26] is used for implementing the DSSR strategy. YETI is developed in Java and 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 includes easy extensibility for future growth, speed of up to one million calls per minute on java code, real time logging, real time GUI support, ability to test programs using multiple strategies and auto generation of test report at the end of the test session. A number of hitherto faults have successfully been found by YETI in various production softwares [26].

YETI can be divided into three main sections including core infrastructure, language-specific bindings and strategies. The core infrastructure represents 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 section defines the procedure of how to select the modules (classes) from the project, how to select routines (methods) from these modules and how to generate values for the instances involved inside these routines. Most common strategies are random and random plus while DSSR strategy is also added to this section with the class name YetiDSSRStrategy. It is extension of YetiRandomStrategy, which in itself is extension of an abstract class YetiStrategy. The class hierarchy is shown in Figure 4.

If no particular strategy is defined during test initialization then YETI uses Random plus strategy by default. To enable the user to control the probability of null values and the percentage of newly created objects.

YETI also provides an interactive Graphical User Interface (GUI) where user can see the progress of the current test in real time. Besides GUI, YETI also provides extensive logs of the test session, which are very helpful in fault tracking. For more details about YETI see references [13] and [12].

IV. Evaluation

To evaluate the DSSR strategy, we compare its performances to the performances of both pure random testing (R) and the random plus (R+) [13] strategy. General

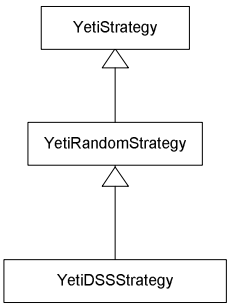

Fig. 4. Class Hierarchy of DSSR in YETI

factors such as system software and hardware as well as the YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability are respectively set to 0.5 and XXX.

A. Experiments

To evaluate the performances of the DSSR strategy we performed extensive testing of programs from the Qualitas Corpus [27]. 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 are collected in an organized form containing both the source and binary forms. The present evaluation uses version 20101126 which contains 106 open source java projects. We picked 80 classes at random from all classes in the Qualitas Corpus and test each of them thirty times with each strategy. Each class is evaluated through 10^5 calls in each testing session.[4] Because of the absence of the contracts and assertions in the code under test, similarly to previous approaches [26], we use undeclared exceptions to compute unique failures found.

All tests are performed using a 64-bit Mac OS X Version 10.7.4 running on Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz with 6.00 GB RAM. Yeti runs on top of the Java(SE) Runtime Environment [Version 6.1.7601]. The machine took approximately 100 hours to process the experimental data.

B. Performance measurement criteria

// WE NEED TO SHORTEN THIS Various measures including the E-measure, P-measure and F-measure have been used by researchers to find the effectiveness of the random test strategy. The E-measure (expected number of failures detected) and P-measure (probability of detecting at least one failure) were hevily criticized [1] and are not considered effective techniques for measuring efficiency of test strategy. The F-measure (number of test cases used to find the first fault) has been often used by researchers [28], [29]. In our initial experiments the F-measure was used to

[4]The total number of tests is thus $80 \times 30 \times 3 \times 10^5 = 720 \times 10^6$ $tests$.

evaluate the efficiency. Soon after a few experiments, it was realized that this was not the right choice because in some experiments the first strategy found first fault quickly than the second strategy but on the completion of test session the first strategy found lower number of total faults than the second strategy. Preference to a strategy only because it found the first fault better without giving due consideration to the total number of faults was not fair. Moreover, for random testing F-measure is quite unpredictable because its value can be easily increased by adding more narrow conditional statements in the SUT. For example in the following program it is difficult for random testing to generate the exact number (3.3338) quickly and therefore the F-measure will be high.

```
{
   if ( (value > 3.3337) && (value < 3.3339) )
   { 10/0 }
}
```

The literature review revealed that F-measure is used where testing stops after identification of first fault and the system is given back to the developers to remove the fault found. In such cases it make sense but now a days automated random testing tools test the whole system and print all of the faults found in one go therefore F-measure is not the favourable choice. Therefore in our experiments, performance of the strategy was measured in terms of finding maximum number of faults in a particular number of test calls [15], [3], [30] which was 10,0000 calls per class. This measurement was found effective because it clearly measured the performance of the strategy when all the other factors were kept constant.

V. Results

// HERE IS THE NEW STRUCTURE

Present the data with the graphs that Mian generated using Excel.

Add data about the number of classes that are best served with each strategy and which are equivalently in the caption of the figure and in the text.

Present basic statistical data (max min, mean, std deviation) about each strategy aggregated over all classes.

Present each of the 6 categories (best with R, best with R+, best with DSSR, R=R+¿DSSR, DSSR=R¿R+, and DSSR=R+¿R)

Conclusions: - we show that for roughly 10- overall DSSR finds more faults in the same time or number of tests (but this is marginally more than other methods)

Please check if there are more conclusions something with the standard deviation (is it higher, lower etc...) and the minimum as well.

needs formatting.

S. No	Class Name	Mean		Std. Dev		Min		max	
		X	Y	X	Y	X	Y	X	Y
1	Routine	1	1	0	0	1	1	1	1
2	Response	1	1	0	0	1	1	1	1
3	Repository	1.2	1	0	0	1.29	1	1.29	1
4	Rectangle	1	1	0	0	1	1	1	1
5	Project	1.05	1.03	1.29	1.6	1.06	1.03	1.09	1
6	ProjectFactory	1	1	0	0	1	1	1	1
7	PersistentSet	1	1	0	0	1	1	1	1
8	PersistentMap	1	1	0	0	1	1	1	1
9	PersistentList	1	1	0	0	1	1	1	1
10	PersistentBag	1	1	0	0	1	1	1	1
1 1	NodeSet	0.98	0.99	0	0	1	1	1	1
1 2	NodeSequence	0.96	0.99	1.26	0.9	1	1	0.95	0.97
1 3	NameEntry	1	1	0.89	0.81	1	1	1	1
1 4	MultiCast	1	1	0	0	1	1	1	1
1 5	Mat4	1	1	0	0	1	1	1	1
1 6	List	1.01	0.94	0	0	0.5	0.5	1	1
1 7	JMXUtilities	1.02	0.99	1.62	1.55	1.16	1	1	1
1 8	JavaWrapper	1.98	1	0.64	1.13	1.5	1	1	1
1 9	ItemSet	1	1	0.71	0.71	1	1	1	1
20	IntHolder	1	1	0	0	0	0	0	0
21	InstrumentTask	1.03	1.01	0	0	2	2	1	1
22	Image	0.91	1.04	0	0	0.71	1.25	0.83	1.07
23	HttpAuth	1	1	0.71	1	1	1	1	1
24	Group	1	1.09	0.81	0.69	1	2.75	1	1
25	Generator	1	1	0	0	1	1	1	1
26	FPGrowth	1	1	0	0	1	1	1	1
27	Font	1	1	0	0	1	1	1	1
28	FileUtil	1	1	0	0	1	1	1	1
29	Files	1	1	0	0	1	1	1	1
30	FileHandler	1	1	0	0	1	1	1	1
31	Facade	1	1	0	0	1	1	1	1
32	Entry	1	1	0	0	1	1	1	1
33	EntryComparator	1	1	0	0	1	1	1	1
34	EntityDecoder	1.02	1.00	1.02	0.96	1	1	1	1
35	Entities	1	1	0	0	1	1	1	1
36	DOMParser	1.03	1	0	0	0	1	1	1
37	DiskIO	1	1	0	0	1	1	1	1
38	Scanner	1.08	1	0.17	1.06	0	1.03	1	1.02
39	Debug	1.05	1.06	1.48	1.62	1	1	1.33	1.33
40	ColumbaClient	1	1	0	0	1	1	1	1
41	ClassLoaderLogM.	1	1	0	0	1	1	1	1
42	CheckAssociator	0.98	1.08	1.10	0.59	1	1	1.12	1
43	CatalogManager	1	1	0	0	1	1	1	1
44	Capabilities	1.08	0.90	1.10	0.90	1	1	1	1
45	BitSet	1	1	0	0	1	1	1	1
46	BaseColor	1	1	0	0	1	1	1	1
47	ArchiveUtil	1	1	0	0	1	1	1	1
48	Apriori	1.02	0.97	1.24	0.88	1	1	1	1
49	AntTypeDefinition	0.96	1.01	0.82	0.82	1	1	1	1
50	AjTypeImpl	0.99	0.99	0.73	0.72	1	1	1	1
51	AdminCore	1	1	0	0	1	1	1	1
52	ActionTranslator	1	1	0	0	1.01	1	1	1
53	RubyBigDecimal	1	1	0	0	1	1	1	1
54	Scanner	0.93	1.09	1.35	1.14	1	1	1	1
55	Scene	0.99	1	0.99	1	1	1	1	1
56	SelectionManager	1	1	0	0	1	1	1	1
57	Server	1.07	0.9	0.89	1.01	0.90	0.83	0.90	0.83
58	Sorter	1.52	1	0	0	3	1	1.5	1
59	Sorting	1	1	0	0	1	1	1	1
60	SSL	1	1	0	0	1	1	1	1
61	Statistics	1.58	1	0	1.2	0	1	0	1
62	Status	1	1	0	0	1	1	1	1
63	Stopwords	1.08	0.99	2.68	1.02	1	1	1	1
64	StringHelper	1	0.98	1.04	1	1.02	1	1	0.97
65	StringUtils	1	1	0	0	1	1	1	1
66	TextImpl	1	1	0	0	1	1	1	1
67	TouchCollector	1	1	0	0	1	1	1	1
68	Trie	0.99	0.99	0.48	0.59	1	1	1	1
69	URI	1	1	0	0	1	1	1	1
70	Utilities	1	1	0	0	1	1	1	1
71	WebMacro	1.01	1	0	1.43	1	1	1.4	1.16
72	XMLAttributesImpl	1	1	0	0	1	1	1	1
73	XMLChar	1	1	0	0	1	1	1	1
74	XMLEntityManager	0.99	1	1	1	0.94	1	0.94	1
75	XMLEntityScanner	1	1	0	0	1	1	1	1
76	XMLErrorReporter	1	1	0	0	1	1	1	1
77	XObject	1	1	0	0	1	1	1	1
78	XString	0.99	1	1.24	0.85	1	1	1	1

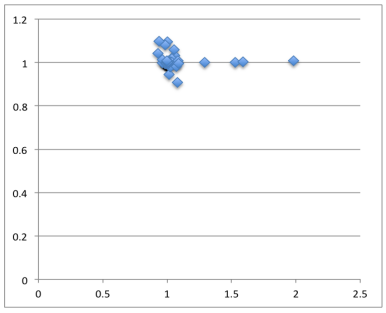

Fig. 5. Combined Mean.

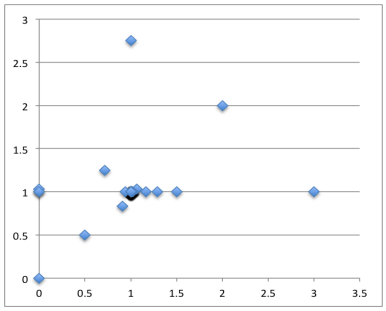

Fig. 6. Combined Min.

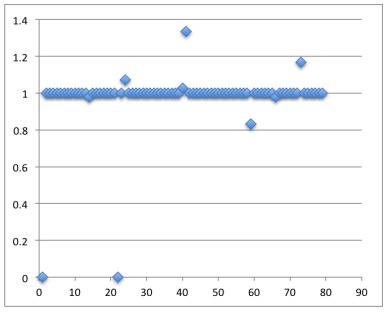

Fig. 7. Combined Max.

// END NEW STRUCTURE

Experimental finding indicate that in XX% experiments DSSR strategy performs better than pure random and random plus strategy, in YY% experiment pure random performs better, in ZZ% experiments DSSR and random plus are equivalent and in XYZ% random plus performs better.

No of Experiments	49
Mean	12.32
Median	5
Standard Deviation	17.73
Min No of Faults	0
Max No of Faults	68

TABLE II: 49 Experiments where each strategy performed equally better

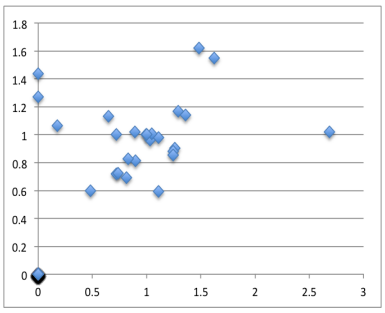

Fig. 8. Combined StdDev.

	RP = DSSR	PR
Mean	11	10.06
Median	11	11
Standard Deviation	0	2.42
Min No of Faults	11	4
Max No of Faults	11	11

TABLE III: 1 out of 80 Experiments where Pure random and DSSR performed equally better

	PR	DSSR	RP
Mean	16.78	16.65	16.6
Median	17	17	17
Standard Deviation	8.91	8.95	9.03
Min No of Faults	1	1	1
Max No of Faults	17	17	17

TABLE IV: 11 out of 80 Experiments where Pure Random (PR) strategy performed better than DSSR and Random Plus.

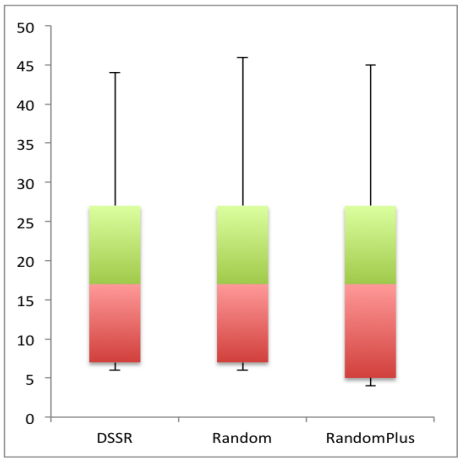

Fig. 9. Pure Random showing better results.

	PR	DSSR	RP
Mean	23.44	26.36	26.32
Median	12	12	12
Standard Deviation	15.81	14.70	14.85
Min No of Faults	0	4	4
Max No of Faults	45	45	46

TABLE V: 7 out of 80 Experiments where DSSR strategy performed better than Pure Random and Random Plus

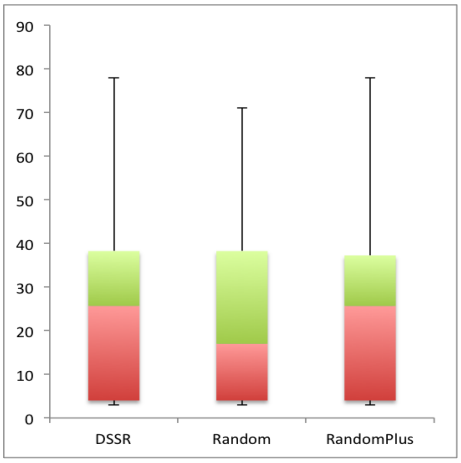

Fig. 10. DSSR showing better results.

	PR	DSSR	RP
Mean	26.32	26.68	26.95
Median	7	7.5	8
Standard Deviation	31.39	31.02	31.15
Min No of Faults	1	2	1
Max No of Faults	83	83	83

TABLE VI: 7 out of 80 Experiments where Random Plus strategy performed better than Pure Random and DSSR

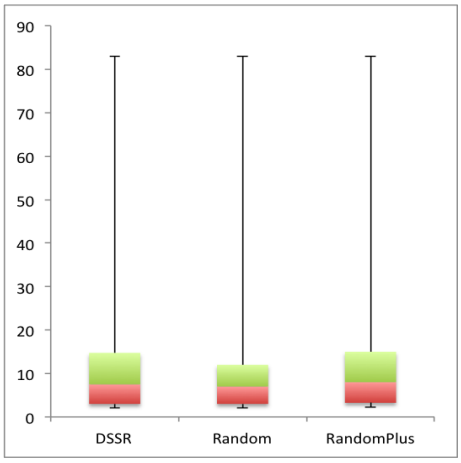

Fig. 11. Random plus showing better results.

	RP = DSSR	PR
Mean	13.76	11.18
Median	7	5
Standard Deviation	15.22	11.60
Min No of Faults	3	0
Max No of Faults	40	31

TABLE VII: 4 out of 80 Experiments where Random Plus and DSSR performed equally better

Note mian (write that there was no case Pure Random is equal to Random Plus.

From the table **??** we can see that in 49 out of 80 experiments all three strategies found equal number of faults. Which means that each strategy is capable enough to find all the faults in one lac test calls. In 11 out of remaining 31 classes Pure Random found highest number of faults. In 7 experiments DSSR found the highest number of faults while in same number of cases Random Plus performed better than DSSR and Pure Random. In 5 experiments both DSSR and Random Plus performed equally well while in only 1 experiment PR and DSSR performed better than Random Plus. Table VIII include names of all the faults that DSSR strategy found in respective classes. Duplicate faults were removed for simplicity.

Class Name	Unique Fault Name
java.lang.ProcessBuilder	NullPointerException YetiSecurityException ArrayIndexOutOfBoundsException
java.lang.ClassLoader	NoClassDefFoundError IndexOutOfBoundsException
java.lang.Character	StringIndexOutOfBoundsException ArrayIndexOutOfBoundsException IllegalArgumentException
java.util.Scanner	IllegalArgumentException NoSuchElementException PatternSyntaxException IllegalStateException StringIndexOutOfBoundsException InputMismatchException UnsupportedOperationException
java.util.Properties	NullPointerException ClassCastException
java.util.Calendar	ArrayIndexOutOfBoundsException IllegalArgumentException
java.lang.Double	NullPointerException
java.lang.Thread	IllegalArgumentException IllegalThreadStateException NoSuchMethodError
java.lang.String	PatternSyntaxException IndexOutOfBoundsException StringIndexOutOfBoundsException
java.lang.Collections	ClassCasteException IndexOutOfBoundsException UnsupportedOperationException IllegalArgumentException OutOfMemoryError

TABLE VIII: Unique faults found by DSS in respective class

VI. Discussion

Performance of DSSR strategy, Random strategy and Random plus strategy in terms of finding faults: Analysis of results revealed that DSSR performs better than random and random plus in programs with block and strip pattern of faults. However, since not all the programs contain faults in the form of block and strip patterns therefore the results do not show a significant performance change.

Time taken by DSSR strategy, Random strategy and Random plus strategy to execute tests: To execute equal number of test cases, DSSR strategy took slightly more execution time than pure random and random plus test strategy. It is not unusual and we were expecting similar behaviour because pure random algorithm selects random input of

the required type with minimum calculation and therefore its process is very quick. On the other hand random plus and DSSR strategy performs additional computation when it maintains the list of interesting values and selects the correct type test values from the list when required. The desired process of adding values to the list and selecting the required values from the list consumes extra time which is the main reason that DSSR strategy takes a little extra time. Thus in executing tests random strategy, random plus and DSSR strategy comes first, second and third respectively.

Effect of test duration in terms of time and number of tests on test results: We found that test duration increases either because of increase in time or number of test cases which results in improving the performance of DSSR strategy than random and random plus. It is because when test duration or number of tests increases, the list of interesting values also increases and in turn DSSR strategy get enough relevant values in the list of interesting values and can easily pick one from the list instead of selecting it randomly or from static list of random plus.

Effect of number of faults on results: We also found that DSSR strategy performs better when the number of faults are more in the code. The reason is that when a fault is found in the code, DSSR strategy adds the neighbouring values of the fault finding value to the list of interesting values. Doing this increases the list of interesting values and the strategy is provided with more relevant test data resulting in higher chance of finding faults.

Can Pure Random and Random Plus Testing perform better than DSSR strategy: The experimental results indicated that pure random and random plus testing can perform better than DSSR strategy if the SUT contain point pattern of failures rather than block and strip pattern. It is due to the fact that in such cases faults don't lay in the neighbourhood of found fault and adding neighbouring values of the founded fault dont make any impact on performance therefore the extra computational time becomes a liability.

DSSR strategy Dependance on Random and Random Plus Testing: During the experiments we found that if the fault finding value is not in the list of interesting values then the test is dependant on random testing. In that case DSSR strategy has to wait for random testing to find the first fault and only then DSSR strategy will add its neighbouring values to the list of interesting values.

VII. Related Work

VIII. 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 and shorter execution time. The experimental findings revealed that DSSR strategy was up to 20% more effective in finding faults as compared to random strategy and up to 10 % more effective than random plus strategy. The DSSR strategy not only gave more consistent results but it proved more effective in terms of detecting faults as compared to random and random plus testing.

Improvement in performance of DSSR strategy over random strategy was achieved by taking advantage of Random Plus and fault neighbouring values. Random plus incorporated not only border values but it also added values having higher chances of finding faults in the SUT to the list of interesting values.

The DSSR strategy is highly effective in case of systems containing block and strip pattern of failure across the input domain.

Due to the additional steps of scanning the list of interesting values for better test values and addition of fault finding test value and its neighbour values, the DSSR strategy takes up to 5% more time to execute equal number of test cases than pure random and random plus.

In the current version of DSSR strategy, it might depend on random or random plus strategy for finding the first fault if the fault test value was not in the list of interesting values. Once the first fault is found only then DSSR strategy could make an impact on the performance of test strategy.

The limitation of random plus strategy is that it maintains a static list of interesting values which remains the same for each program under test, and can be effective in many cases but not always. The better approach will be to have a dynamic list of interesting values that is automatically updated for every program which can be achieved by adding the program literals and its surrounding values to the list of interesting values prior to starting every new test session.

IX. Future Work

From the research we came to know that random testing is not very good in generating a test value when the scope of a variable is too narrow as in the following example.

```
{
   if(value == 34.4445)
   { 10/0 }
}
```

We also know that if the fault finding value is not in the list than DSSR has to wait for random testing to generate the fault finding value and only after that DSSR strategy will add that value and its surrounding values to the list of interesting values. To decrease the dependancy of DSSR strategy on random and random plus strategy, further work is in progress to add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained either from .java or .class files of the SUT. We are also working to add neighbouring values of the literals to the list of interesting values.

Thus if we have the above example then the value 34.4445 and its surrounding values will be added to the list of interesting values before the test starts and DSSR strategy will no more be dependent on random testing to find the first fault. Finally, it will also be interesting to evaluate the DSSR strategy in terms of coverage because the newly added values are most suitable for test cases and therefore can increase branch coverage.

X. Acknowledgment

The auther would like to acknowldge with thanks the Department of Computer Science, University of York for the financial support extended as Departmental Overseas Research Scholarship (DORS) which enabled him to continue higher studies leading to PhD. We extend our sincere thanks and appreciation to Prof. Richard Page, Head of the Enterprise System's group of the Department of Computer Science for his valuable help and generous support.

References

[1] T. Y. Chen, "Adaptive random testing," *Eighth International Conference on Qualify Software*, vol. 0, p. 443, 2008.

[2] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proceedings of the 7th International Conference on Software Quality*, ser. ECSQ '02. London, UK, UK: Springer-Verlag, 2002, pp. 321–330. [Online]. Available: http://portal.acm.org/citation.cfm?id=645341.650287

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[4] 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*, ser. QSIC '03. Washington, DC, USA: IEEE Computer Society, 2003, p. 4. [Online]. Available: http://portal.acm.org/citation.cfm?id=950789.951282

[5] T. Y. Chen and R. Merkel, "Quasi-random testing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 309–312. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101957

[6] F. Chan, T. Chen, I. Mak, and Y. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775 – 782, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0950584996011032

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

[8] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, pp. 60–66, January 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1663656.1663914

[9] 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*, vol. 9999, no. 9999, pp. 1–7, 2009. [Online]. Available: http://www3.interscience.wiley.com/journal/122498617/abstract

[10] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner, "Finding faults: Manual testing vs. random+ testing vs. user reports," in *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 157–166. [Online]. Available: http://portal.acm.org/citation.cfm?id=1474554.1475420

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: adaptive random testing for object-oriented software," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368099

[12] M. Oriol and S. Tassis, "Testing .net code with yeti," in *Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems*, ser. ICECCS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 264–265. [Online]. Available: http://dx.doi.org/10.1109/ICECCS.2010.58

[13] M. Oriol and F. Ullah, "Yeti on the cloud," *Software Testing Verification and Validation Workshop, IEEE International Conference on*, vol. 0, pp. 434–437, 2010.

[14] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, ser. HICSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 261a–. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.462

[15] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 84–94. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273476

[16] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279. [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[17] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion, Montreal, Canada*. ACM, Oct. 2007.

[18] C. Oriat, "Jartege: a tool for random generation of unit tests for java classes," *CoRR*, vol. abs/cs/0412012, 2004.

[19] G. J. Myers and C. Sandler, *The Art of Software Testing*. John Wiley & Sons, 2004.

[20] J. W. Duran and S. Ntafos, "A report on random testing," in *Proceedings of the 5th international conference on Software engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 179–183. [Online]. Available: http://portal.acm.org/citation.cfm?id=800078.802530

[21] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *Software Engineering, IEEE Transactions on*, vol. SE-10, no. 4, pp. 438 –444, july 1984.

[22] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[23] S. C. Ntafos, "On comparisons of random, partition, and proportional partition testing," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 949–960, October 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=505464.505469

[24] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol, "On the effectiveness of test extraction without overhead," in *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 416–425. [Online]. Available: http://dl.acm.org/citation.cfm?id=1547558.1548228

[25] M. Oriol. (2011) York extensible testing infrastructure. Department of Computer Science, The University of York. [Online]. Available: http://www.yetitest.org/

[26] ——, "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*, april 2012, pp. 201 –210.

[27] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble, "Qualitas corpus: A curated collection of java code for empirical studies," in *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[28] T. Chen and Y. Yu, "On the expected number of failures detected by subdomain testing and random testing," *Software Engineering, IEEE Transactions on*, vol. 22, no. 2, pp. 109 –119, feb 1996.

[29] T. Y. Chen, F.-C. Kuo, and R. Merkel, "On the statistical properties of the f-measure," in *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, sept. 2004, pp. 146 – 153.

[30] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 72–81. [Online]. Available: http://portal.acm.org/citation.cfm?id=1381305.1382069