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—This paper presents an enhanced and improved form of automated random testing, called Dirt Spot Sweeping Random (DSSR) strategy. DSSR strategy is a new strategy that combines ordinary random strategy and random plus strategy to achieve their combined benefits and additionally sweeps the dirt spots (Faulty patterns) in the program code for faults. It is based on two intuitions, first is that test values in boundaries of equivalence partition are interesting, using these test values in isolation can detect new faults in the system with fewer number of test executions, which produce high impact on test results while second is that faults reside in block and strip patterns inside the input domain therefore when a fault is found, using neighbouring values of the fault finding value can reveal more faults quickly which consequently increases the test performance.

I. Introduction

Success of software testing is based mainly on the number of faults it discovers in the Software Under Test (SUT). An efficient testing process would discover maximum number of faults in minimum possible time. Exhaustive testing where software is tested against all possible inputs is not feasible in most cases because of large amount of inputs, limited resources and strict time constraints. Therefore, strategies in automated software testing tools are developed with the aim to selects more of the fault finding test cases from the input domain for the given SUT. However, it is difficult to produce such targeted test cases because each system has its own requirements and functionality.

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. 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 get better results.

To produce more relevant and fault targeted test input data for random testing, we developed an enhanced technique called Dirt Spot Sweeping Random (DSSR) strategy. The following research issues about the DSSR strategy were considered and subsequently addressed:

1) To get highly efficient algorithm for coping with the combination of strategies including pure random, random plus and spot sweeping.
2) To get high number of unique faults in the SUT.
3) To get lower number of unique faults and higher number of similar faults in the SUT.
4) No improvement in test results of the SUT.
5) Consumption of processing time for DSSR strategy.

The rest of this paper is organised as follows. Sections, II to X describe Dirt Spot Sweeping Random strategy, Implementation of DSSR strategy, Experimental setup and analysis, evaluation of DSSR strategy, Experimental results, Unique faults found by DSSR strategy, discussion, conclusion and future work respectively.

II. Dirt Spot Sweeping Random Strategy

Dirt Spot Sweeping Random (DSSR) strategy is a new strategy developed during present study. It is combination of the two existing strategies i.e. pure random and random plus with the addition of one new strategy called spot sweeping. It 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 can reside in block and strip pattern thus using neighbouring values of the fault finding value can lead us to the next fault in the same block or strip. This might increase the performance of the test strategy in terms of executing fewer number of test cases with more number of faults. Each component is briefly explained as follows.

A. Random Testing Strategy

Random testing is a black-box testing technique in which the SUT is executed against randomly selected test data. Test results obtained are compared against the oracle defined using SUT specifications in the form of contracts or assertions. In the absence of contracts/assertions the exceptions defined by the programming language in which the program is developed are used as test oracle. According to Beizer [7], software performance is directly dependant on the combination of two main factors, correctness and robustness. Correctness is the expected behaviour of the software based on its specifications while robustness is the behaviour of the software which is not defined in its specifications. Since random testing generates test data randomly without any specific pattern therefore it effectively tests the performance of software by evaluating it for both correctness and robustness. This strategy because of its black-box testing nature 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 a few of the most common automated testing tools based on random strategy.

In the past, 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 in science there is no substitute for experimental analysis and later on various experiments performed by different researchers [15], [20], [21], [22] and [23] experimentally proved that random testing is simple to implement, cost effective, highly efficient and free from human bias compared to its rival techniques.

B. Random Plus Strategy

Random plus strategy [14] is extension of the pure random strategy. It uses some special pre-defined values which can be simple border 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, if input for a SUT is days of an year which is expressed in numbers from 1 to 365 then -2, -1, 0, 1, 2, 363, 364, 365, 366, 367 can be considered as border values as shown in Figure 1.

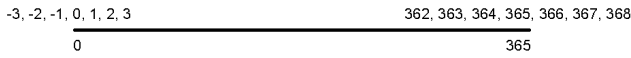-3, -2, -1, 0, 1, 2, 3   362, 363, 364, 365, 366, 367, 368

0   365

Fig. 1.   Boundary values for input domain from 1 to 365

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 its more relevance and high chances of finding faults for the given SUT. It is reported that these special values have high impact on the results particularly detecting problems in specifications [10].

C. Dirt Spot Sweeping

Chen et al. [1] found that there are patterns of failure-causing inputs across the input domain. Figure 2 show these patterns for two diminutional input domain. They divided these patterns into three types called block, point and strip patterns. Black points, boxes and strips inside the box show the input that causes the system to fail while white space inside the

box represent the non failure-causing or genuine input. Black square box represent the boundary values. They also argued that a strategy can get 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.
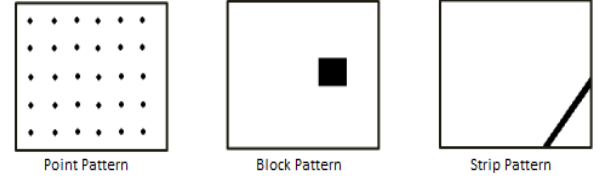

Fig. 2.   Failure patterns across input domain [1]

Spot sweeping is the part of DSSR strategy that come in to action when a fault is found in the system. On finding fault, it immediately add the value causing the fault and its neighbouring values to the already existing list of interesting values at run time. For example in a program if an int value of 50 causes a fault then spot sweeping will add values from 45 to 55 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 faults present in that strip. We can also say that in random plus the list of interesting values remain static/constant whereas in DSSR strategy the list of interesting values is dynamic and changes during the test execution of each program.
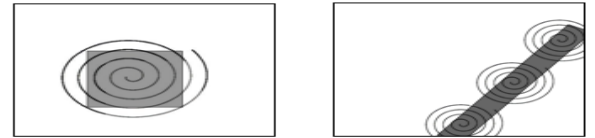

Fig. 3.   DSSR covering block and strip pattern

Figure 3 shows how spot sweeping explores the faults residing in the block and strip patterns of a program. The faults coverage from the block and strip 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 4. 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 0 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. From the flow-chart you can see that if the fault finding value is of primitive type then the DSSR finds the type of that primitive value and add values only of that particular type to the interesting values list. Addition of these values increases the size of the list of interesting values which 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.

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)

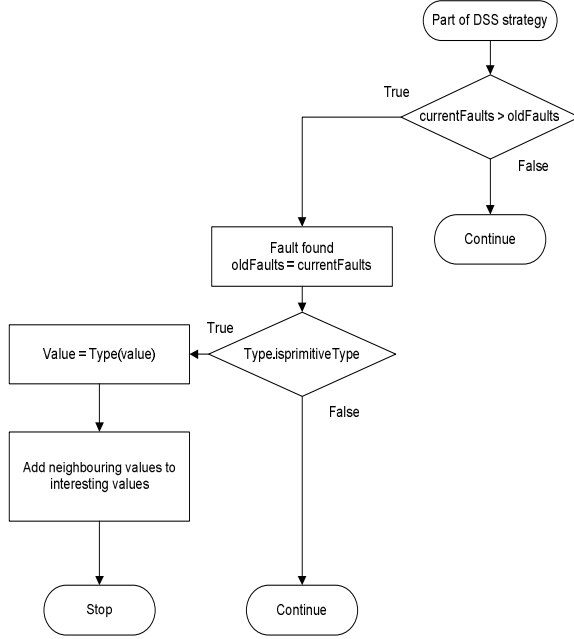TABLE I: Neighbouring values for primitive types and String

Fig. 4. 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 value and fault 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. 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 to the list of interesting values and vice versa.

E. Explanation of DSSR with an example program

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

```
/**
 *  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 is from -2,147,483,648 to 2,147,483,647. DSSR further select some values like 0, Integer.MIN_VALUE and Integer.MAX_VALUE as interesting values and increase its priority for selection as test values. When test start all 3 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: When DSSR strategy catch the first fault, it adds the fault generated and its surrounding values to the list of interesting values which includes 0, 1, 2, 3 and -1, -2, -3 in this case. For second test case DSSR strategy may pick -3 as a test value and it will lead us to the second fault where assertion (2) will fail because the square root of 9 will be +3 instead of the original value -3.

Fault 3: Few tests later DSSR strategy may select Integer.MAX_VALUE for variable "num1" which is also available in the list of interesting values and it will lead us to the 3rd fault because result1 will not be able to store the square of Integer.MAX_VALUE. Instead of the actual calculated square value Java will assign a negative value (Java language rule) to

variable result1 which will again lead to the violation of next assertion (3).

From the above execution process we can understand that, in this example, pre-defined values including border values, fault finding values and its surrounding values lead us quickly to the available faults and in less number of tests as compared to Random and Random plus strategy which takes longer to discover the same faults because they try to find faults randomly even when the first fault found is very close to the second.

III. IMPLEMENTATION OF DSSR STRATEGY

Instead of reinventing the wheel the DSSR strategy is implemented in an existing open-source automated random testing tool called YETI [25], [26]. YETI tool is developed in Java which 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.

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 way to select the class/module to test random selection of routine/method from the given module and get instances of the required type during testing. Most common strategies in this section are random, random plus, DSSR strategy is also added to the strategies section of YETI tool 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 5.


```
┌─────────────────┐
│  YetiStrategy   │
└─────────────────┘
         △
         │
┌─────────────────────┐
│ YetiRandomStrategy  │
└─────────────────────┘
         △
         │
┌─────────────────┐
│ YetiDSSStrategy │
└─────────────────┘
```
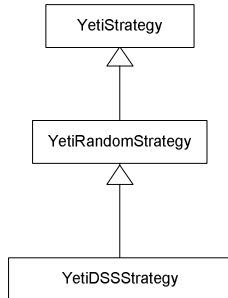
Fig. 5.   Class Hierarchy of DSSR in YETI

If no particular strategy is defined during test initialisation then YETI will use its default random plus strategy in which the user can control the probability of null values and the percentage of newly created objects. Both probabilities are set to 10% by default.

YETI also provide an interactive Graphical User Interface (GUI) where a 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 [12] and [13].

IV. EVALUATION

A. Use of Qualitas Corpus for Evaluation

To evaluate the performance of DSSR strategy we performed extensive testing of programs that cover full set of Java applications. Qualitas corpus [27] is a curated collection of such open source java projects built with the aim to help testers in empirical research. These projects are collected in an organised form containing both the source and binary form. The reason for including source code is that most of the binaries do not include infrastructure code like code for demonstrating aspects of the system, testing, installation and building or management tasks. In its latest version 20101126, it contain 106 open source java projects. It is available in two distributions. The release version "r" and the evolution version "e". The release version is compact size which contain only the recent version of the projects while the evolution version is more detailed which consists of more than 10 different versions of each project.

B. Evaluation by comparison under identical conditions

To get a clear view of DSSR performance we determined the comparative performance of DSSR strategy with pure random strategy and random plus strategy by applying them to similar systems under identical conditions. Both the general factors like system software, hardware and YETI specific factors like percentage of null values, percentage of newly created objects and Interesting value injection probability were kept constant for each test strategy. For all our experiments where applicable the interesting value injection probability was set to 0.5, which means that 50% of the test values will be selected from the list of interesting values while remaining 50% of the values will be selected randomly.

C. Minimising the effect of Randomness

Random testing is probably the only strategy whose not only input is random but its output is also random. It is not necessary that all the faults it found in one test run can also be found in the second test run. Therefore the performance of random strategy cannot be evaluated from a few test sessions. To avoid the random behaviour of random testing every class was tested at least 30 times each by pure random, random plus and DSSR strategy. This is achieved by creating a batch executable script with the handy feature of YETI called compact report that logs each test report to a file for later evaluation.

D. Performance measurement criteria

Various measures including F-measure, P-measure and E-measure have been used by researchers to find the effectiveness of the random test strategy. E-measure (Expected number of failures detected) and P-measure (Probability of detecting at least one failure) received criticism from researchers [1] and are not considered effective techniques for measuring efficiency of test strategy. F-measure (Number of test cases used to find the first fault) used by researchers [28], [29] is quite well known and initially we used it in our experiments to calculate the efficiency. After a few experiments we came to know that this was not the right choice because in some experiments the first strategy found first fault quickly than the second strategy but after the complete test session the first strategy found lower number of total faults than the second strategy. In our view it is not fair to prefer a strategy only because it found the first fault better without giving due consideration to the total number of faults. 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 }
}
```

From the literature we also found 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 found bug where it make though not for random testing but since now automated random testing tests the whole system and print all of the faults found therefore F-measure is not the favourable choice. Therefore in all 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 in our case was set to 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.

E. Experiments

Extensive experiments were carried out to evaluate the performance of DSSR strategy. Every class was tested 30 times by random, random plus and DSSR strategy. The total number of experiments performed are 100 x 30 x 30 x 30 = 9000. Each class was evaluated by 1 lac test cases in each experiment so The total number of test cases in 9000 experiments are 9000 x 1000000 = 9000000000. The automated oracle used for all experiments was the defined exception of the language because of the absence of the contracts and assertions in the code under test.

Commands for executing the experiments using pure random, random plus and DSSR strategies are as follows. Here Prog1 is the name of the class and nTests is the number of tests set to be executed during this experiment.

- java yeti.Yeti -java -testModules=Prog1 -nTests=10000 -nologs -gui -random.
- java yeti.Yeti -java -testModules=Prog1 -nTests=10000 -nologs -gui -randomPlus.
- java yeti.Yeti -java -testModules=Prog1 -nTests=10000 -nologs -gui -DSSR.

All tests were performed using 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. Furthermore, Java(SE) Runtime Environment [Version 6.1.7601] was used. It took 4 days (96 hours) for this machine to process all the experiments.

V. RESULTS

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 Pure random and DSSR performed equally better

	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.

	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

	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

	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

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

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.

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. Conclusion

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.

VIII. 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.

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] T. Y. Chen and F.-C. Kuo, "Is adaptive random testing really better than random testing," in *Proceedings of the 1st international workshop on Random testing*, ser. RT '06. New York, NY, USA: ACM, 2006, pp. 64–69. [Online]. Available: http://doi.acm.org/10.1145/1145735.1145745

[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