Dirt Spot Sweeping Random Strategy: an upgraded version of Random+ Testing

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

Manuel Oriol
Department of Computer Science
University of York
York, United Kingdom
manuel.oriol@york.ac.uk

***Abstract*—Random testing is a simple but highly effective technique to find failures in complex programs. However, its efficiency reduces when the failures lie in contiguous locations across the input domain. To overcome the deficiency, we developed a new automated technique: Dirt Spot Sweeping Random (DSSR) strategy. It is based on the assumption that unique failures reside in contiguous blocks and stripes. When a failure is identified, the DSSR strategy selects neighbouring values for the subsequent tests. Resultantly, selected values sweep around the failure leading to the discovery of new failures in the vicinity. To evaluate the effectiveness of DSSR strategy a total of 60 classes (35,785 lines of code), each class with 30 x 10^5 calls, were tested by Random (R), Random+ (R+) and DSSR strategies. T-Test analysis showed significantly better performance of DSSR compared to R strategy in 17 classes and R+ strategy in 9 classes. In the remaining classes all the three strategies performed equally well. Numerically, the DSSR strategy found 43 and 12 more unique failures than R and R+ strategies respectively. This study comprehends that DSSR strategy will have a profound positive impact on the fault-finding ability of random testing.**

I. Introduction

The success of a software testing technique is mainly dependant on the number of faults it discovers in the SUT. An efficient testing process discovers the maximum number of faults in minimum possible test cases and time. Exhaustive testing is not always feasible, therefore, strategies in automated testing tools are developed with the aim to select more fault-finding test cases from the whole input domain.

Chan et al. [1] discovered that there are patterns of failure-causing inputs across the input domain. They divided these into point, block and strip patterns on the basis of their occurrence. Chen et al. [2] found that the performance of random testing can be increased by altering the technique of test case selection. Moreover, they found that the performance increased up to 50% when test input was selected evenly across the whole input domain. This was mainly attributed to the better distribution of input, which increased the chances of selecting inputs from failure patterns.

Based on the assumption that in a significant number of classes, failure domains are contiguous or very close by, the Dirt Spot Sweeping[1] is devised to give higher priority to the failure domains for identification of new failures efficiently. The DSSR strategy is implemented in the York Extensible Testing Infrastructure (YETI)[2], a random testing tool. To evaluate the effectiveness of DSSR strategy a total of 60 classes (35,785 lines of code) of 32 different projects from the Qualitas Corpus[3], each class with 30 x 10^5 calls, were tested by R, R+ and DSSR strategies.

This paper is organized as follows: Section II describes the DSSR strategy. Section III presents implementation of the DSSR strategy. Section IV explains the experimental setup. Section V reveals results of the experiments. Section VI discusses the results. Section VII presents related work and Section VIII concludes the study.

II. Dirt Spot Sweeping Random Strategy

Dirt Spot Sweeping Random (DSSR) strategy combines the random+ strategy with a Dirt Spot Sweeping (DSS) 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, failures reside in contiguous patterns. If this is true, DSS increases the performance of the test strategy. Before presenting the details of the DSSR strategy, it is pertinent to review briefly the R and the R+ strategy.

A. 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 [3]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational effort [4], [5]. It is mainly for this reason that various researchers have recommended random strategy in automated testing tools [5]. YETI [6], AutoTest [7], [8], QuickCheck [9], Randoop [10] and JArtege [11] are some

[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

of the most common automated testing tools based on random strategy.

Efficiency of random testing was made suspicious with the intuitive statement of Myers [12] who termed random testing as one of the poorest methods for software testing. However, experiments performed by various researchers [8], [13], [14], [15], [16] have proved experimentally that random testing is simple to implement, cost effective, efficient and free from human bias as compared to its rival techniques.

B. Random Plus Strategy (R+)

The random+ strategy [7] is an extension of the random strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding faults in the SUT. Boundary values are the values in the start and end of a particular type [17]. For instance, such values for `int` could be `MAX_INT`, `MAX_INT-1`, `MAX_INT-2`, `0`, `MIN_INT`, `MIN_INT+1`, and `MIN_INT+2`. Similarly, the tester might also add some other special values that are considered effective in finding faults in 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 a high priority (10%) than selection of random values because of more relevance and better chances of finding faults.

C. Dirt Spot Sweeping (DSS)

Chan et al. [1] found that there are patterns of failure-causing inputs across the input domain. They divided these patterns into three types called points, block and strip patterns (figure 1) and argued that a strategy has more chances of hitting the fault patterns if test cases are selected farther from each other. Other researchers [18], [19], [20], also tried to generate test cases further away from one another targeting these patterns and achieved better performance. Such increase in performance indicates that faults more often occur contiguous across the input domain. In DSS, if a value reveals fault

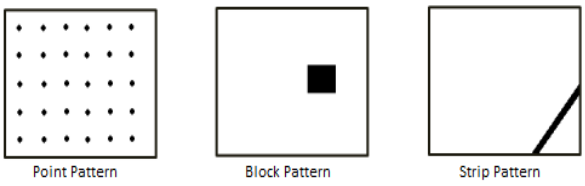


Fig. 1. Failure patterns across input domain [2]

from the block or strip pattern then for the selection of the next test value, DSS may not look farthest from the known value but picks the closest value to find another fault from the same region. DSSR strategy relies on DSS 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 DSS will add values from `47 to 53` to the list of interesting values. The addition of neighbouring values will explore other failures present in the block or strip domain of the SUT. The list of interesting values in DSSR strategy is dynamic and changes during the test execution of each program as against R+ where the list remains static.

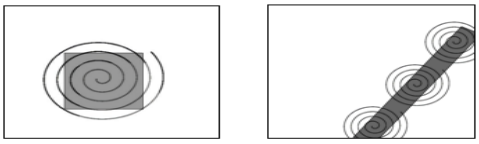


Fig. 2. DSSR strategy covering block and strip pattern

Figure 2, shows how DSS explores the failures residing in the block and strip patterns of a program. The coverage of block and strip pattern is shown in spiral form because first failure leads to second, second to third and so on till the end. In case the failure is positioned on the point pattern then the added values may not be effective because point pattern is only an arbitrary failure point in the whole input domain.

D. 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 [21]. 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.

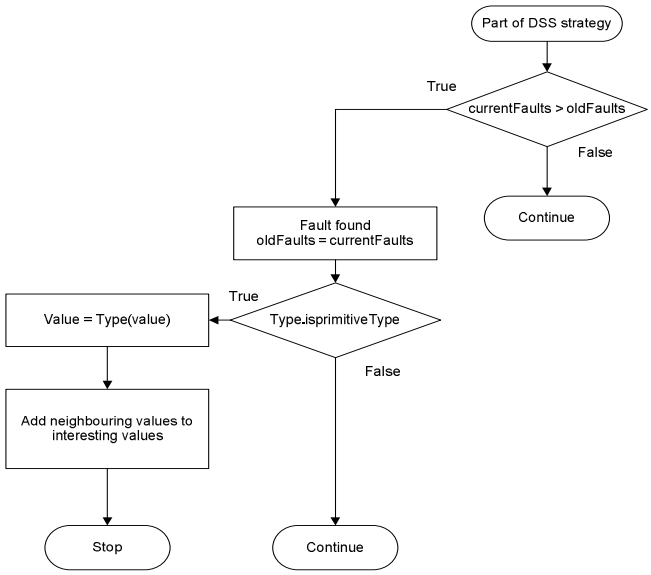


Fig. 3. Working mechanism of DSSR strategy

Both the variables `currentFaults` and `oldFaults` are initialised to 0 at the start of the test, when a failure occurs

the `currentFaults` value is incremented which make the condition true. The flowchart presented in Figure 3 depicts the case when failure is caused by a primitive type value. The DSSR strategy identifies its type and adds values only of that particular type to the list of interesting values. The resultant list of interesting values provides 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.

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)
X is object of user defined class	Call its constructor recursively until empty or primitive values

TABLE I: Neighbouring values for primitive types and string

Table I presents the data types with the corresponding values to be added to the list of interesting values. 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.

E. Explanation of DSSR strategy on a concrete example

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

```
/**
* Calculate square of given number
* and verify result.
* @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 result1 >= num1; // 2
  assert Math.sqrt(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 that are prioritized for selection as test inputs. As the test starts, three faults are quickly discovered by DSSR strategy in the following order:

Fault 1: The strategy selects value `0` for variable `num1` in the first test case because `0` is available in the list of interesting values and its priority is higher than other values. This will cause Java to generate division by zero exception (1). After discovering the fault, the strategy adds its surrounding values to the list of interesting values i.e. `1, 2, -1, -2`.

Fault 2: After a few tests the DSSR strategy may select `Integer.MAX_VALUE` for variable `num1` from the list of interesting values leading to the discovery of the 2nd 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 `1` (Java language rule) to variable `result1` that will lead to the violation of the next assertion (2).

Fault 3: In the third test case the strategy may pick `-3` as a test value, which is added to the list of interesting values after the discovery of first fault. This may lead to the third fault where assertion (3) fails because the square root of `9` is `3` against the input value of -3.

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

III. Implementation of the DSSR strategy

Implementation of the DSSR strategy is made in the YETI, an open-source automated random testing tool. YETI, coded in Java, 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, capability to test programs using multiple strategies, high speed tests execution, real time logging, GUI support and auto generation of test report at the end of test session [22], [23].

YETI can be divided into three loosely coupled main parts: the core infrastructure, the language-specific bindings and the 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 calls 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 initialization. It also enables the user to control the probability of using null values and the percentage of newly created objects for each test session. In addition to GUI, YETI also provides extensive logs of the test session for more in-depth analysis.

IV. Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [7]. 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.

A. Research questions

For evaluating the DSSR strategy, the following research questions were formulated and addressed in this study:

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

B. Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [24]. The Qualitas Corpus is a curated collection of open source java projects designed with the aim of helping empirical research in software engineering. These projects have been collected in an organized form containing both the source and binary forms. Version 20101126 containing 106 open source java projects was used in our experiments. From 32 randomly selected projects, 60 classes were selected at random with the help of automated pseudo-random generator. The selected classes produced at least one fault and timed out within the testing session of not more than 10 minutes. Every class was tested thirty times by each strategy (R, R+, DSSR). Test details of the classes are presented in table II. Programs tested at random typically fail most of the times as a result of large number of calls. Therefore, it is necessary to cluster failures that likely represent the same fault. The traditional way is to compare the full stack traces and error types and use this as an equivalence class [8], [22] called a unique failure. The same concept of unique failure has been adapted in the present study.

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, Undeclared exceptions were considered as unique failures in accordance with previous studies [22].

All tests were 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.

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

C. 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 criticized and are not considered effective measuring techniques [2] while the F-measure has been often used by various researchers [25], [26]. In our initial experiments the F-measure was used to evaluate the efficiency. However it was realized that this was 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 [27].

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 favorable choice. In our experiments, performance of the strategy was measured by the maximum number of faults detected in the SUT by a particular number of test calls [28], [8]. This measurement is effective because it considers the performance of the strategy when all other factors are kept constant.

V. Results

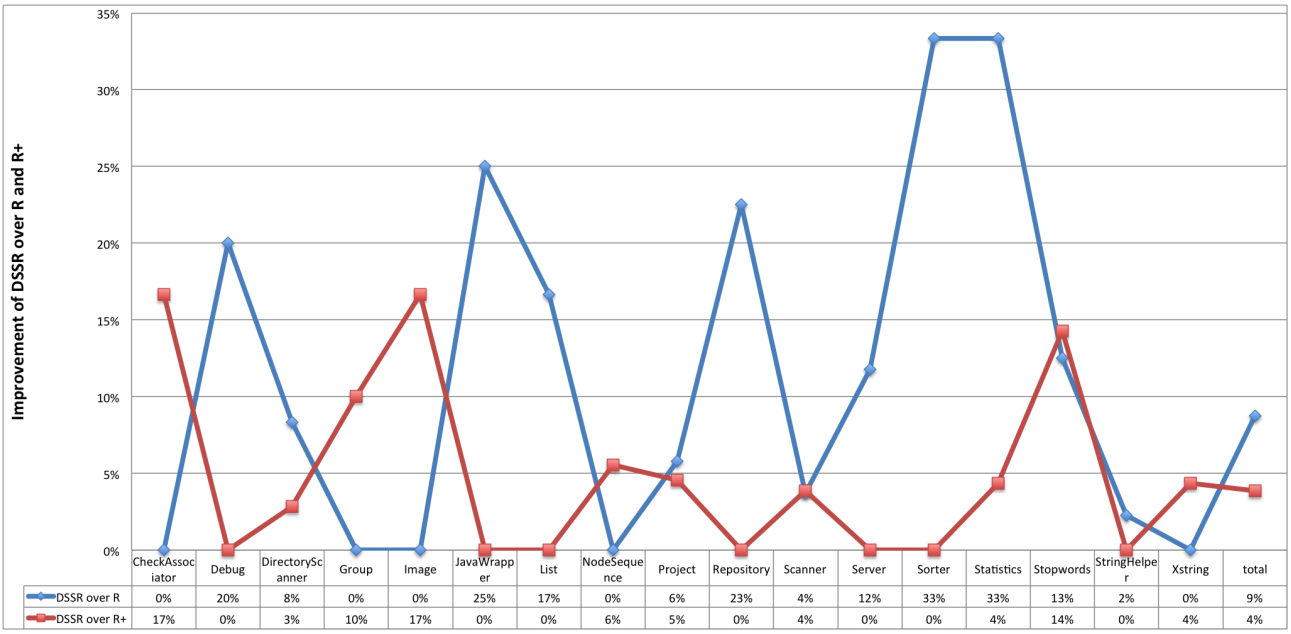
Results of the experiments including class name, Line of Code (LOC), mean value, maximum and minimum number of unique failures and relative standard deviation for each of the 60 classes tested by R, R+ and DSSR strategies are presented in Table II. Each strategy found an equal number of faults in 31 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 R (973) and R+ (1009), which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

A. Is there an absolute better strategy among R, R+ and DSSR strategies?

Based on our findings DSSR strategy performs better than R and R+ strategies. Figure 4 presents the average improvement of DSSR strategy over R and R+ strategies for 17 classes showing significant difference between DSSR and R,

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

TABLE II: Complete results for R, R+ and DSSR strategies. 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.



	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%

Fig. 4. Improvement of DSSR strategy over Random and Random+ strategy.

DSSR and R+. The blue line with diamond symbol shows performance of DSSR over R and the red line with square symbols depicts the performance of DSSR over R+ strategy. 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. The improvement of DSSR over R and R+ strategy is calculated by applying the formula (1) and (2) respectively.

$$\frac{Average faults_{(DSSR)} - Average faults_{(R)}}{Average faults_{(R)}} * 100 \quad (1)$$

$$\frac{Average faults_{(DSSR)} - Average faults_{(R+)}}{Average faults_{(R+)}} * 100 \quad (2)$$

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

T-tests applied to the data given in Table III showed significantly better performance of DSSR compared to R strategy in 17 classes and R+ strategy in 9 classes. In the remaining classes all the three strategies performed equally well. 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.

C. Can we pick the best default strategy between R, R+ and DSSR strategies?

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. However, further analysis of DSSR strategy in terms of code coverage and overhead is required before picking it as a default strategy.

VI. Discussion

In the present study, we evaluated DSSR strategy against R and R+ strategies under identical conditions. This is in accordance with the common practice to evaluate performance of an extended strategy by applying it in combination with other existing strategies to the same programs and comparing the results obtained [29], [30]. We used random testing as a baseline for comparison as advocated by Arcuri et al [31].

In our experiments we selected projects from the Qualitas Corpus [24] which is a collection of open source java programs maintained for independent empirical research. These projects are carefully selected and span across the whole set of java applications [22], [24]. The selection of programs in the current study is made through automated random process to avoid any bias and maintain representative selection of classes.

All the three strategies have the potential of finding failures. However, DSSR strategy found more number of unique failures than both R and R+ strategies as reflected in the results. This improved performance of the strategy can be attributed to the additional feature of Dirt Spot Sweeping in the DSSR strategy.

In DSSR strategy 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 DSSR strategy performs comparatively better when the number of faults is higher in the SUT. The logical explanation is that the higher the number of faults in the SUT, the more

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	

TABLE III: T-test results of the classes showing different results

connected are their fault domains, increasing the chances of finding failures by Dirt Spot Sweeping.

As pointed out earlier, DSSR strategy relies on R+ strategy to identify the first unique failure. We noticed in our experiments that discovery of first unique failure in the early stage of the test triggers the activation of Dirt Spot Sweeping resulting in quick finding of the failures in the SUT. The process is delayed when the identification of the first unique failure is not accomplished in the early stage by R+ strategy. The limitation of the new strategy may be addressed in the future studies by avoiding the reliance of DSSR strategy on R+ for the discovery of first failure.

VII. Related Work

Random testing used in the current study is a popular technique with simple algorithm but a proven method to find subtle faults in complex programs and Java libraries [32], [33]. Its simplicity, ease of implementation and efficiency in generating test cases make it one of the best choice for test automation [15]. Some of the well known automated tools based on random strategy includes Jartege [11], Eclat [32], JCrasher [33], AutoTest [5] and YETI [22].

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 [7]. These interesting values includes border values which have high tendency of finding faults in the given SUT [17]. Results obtained with R+ strategy show significant improvement over random strategy [7].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed. Adaptive random testing (ART) [3], Quasi-random testing (QRT) [20] and Restricted Random testing (RRT) [18] achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror Adaptive Random Testing (MART) [19] and Adaptive Random Testing through dynamic partitioning [19] 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 failure patterns present in the input domain.

A more recent research study [34] stresses 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 using "meta heuristics search" based on hill climbing algorithm to regenerate new data.

VIII. Conclusion

In the present study, we developed a new Dirt Spot Sweeping Random strategy as an upgraded version of the R+ strategy based on the Dirt Spot Sweeping feature, which strengthens the ability of finding more faults in lower number of tests. The DSSR strategy is particularly more effective when the failure domain forms block and strip patterns across the input domain. In addition, the findings of the present study indicated significantly better performance of DSSR in comparison with R and R+ strategies with respect to finding higher number of faults.

References

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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–.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

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

[16] S. C. Ntafos, "On comparisons of random, partition, and proportional partition testing," *IEEE Trans. Softw. Eng.*, vol. 27, pp. 949–960, October 2001.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] M. Oriol, "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.

[23] ——. (2011) York extensible testing infrastructure. Department of Computer Science, The University of York.

[24] 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.

[25] 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.

[26] 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.

[27] H. Liu, F.-C. Kuo, and T. Y. Chen, "Comparison of adaptive random testing and random testing under various testing and debugging scenarios," *Software: Practice and Experience*, vol. 42, no. 8, pp. 1055–1074, 2012.

[28] 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.

[29] W. Gutjahr, "Partition testing vs. random testing: the influence of uncertainty," *Software Engineering, IEEE Transactions on*, vol. 25, no. 5, pp. 661 –674, sep/oct 1999.

[30] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence [program testing]," *Software Engineering, IEEE Transactions on*, vol. 16, no. 12, pp. 1402 –1411, dec 1990.

[31] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, pp. 258–277, 2012.

[32] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *In 19th European Conference Object-Oriented Programming*, 2005, pp. 504–527.

[33] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[34] S. Yoo and M. Harman, "Test data regeneration: generating new test data from existing test data," *Softw. Test. Verif. Reliab.*, vol. 22, no. 3, pp. 171–201, May 2012.