Dirt Spot Sweeping Random Strategy

Mian Asbat Ahmad and Manuel Oriol

Abstract—**The paper presents a new and improved automated random testing technique named as Dirt Spot Sweeping Random (DSSR) strategy based on the rationale that, "when failures lies in the contiguous locations across the input domain, the effectiveness of random testing can be further improved through diversity of test cases". The DSSR strategy selects neighboring values for the subsequent tests on identification of failure. 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 105 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 failure-finding ability of R and R+ testing.**

Index Terms—**Software testing, automated random testing, ADFD.**

I. Introduction

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

Based on the assumption that in a significant number of classes, failure domains are contiguous, the Dirt Spot Sweeping Random [1] strategy 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 30×10^5 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.

Manuscript received February 5, 2014; revised April 9, 2014.

Mian Asbat Ahmad and Manuel Oriol are with the Department of Computer Science, University of York, YO10 5DD, UK (e-mail: mian.ahmad@york.ac.uk, manuel.oriol@york.ac.uk).

[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

II. Dirt Spot Sweeping Random Strategy

Dirt Spot Sweeping Random strategy combines the R+ 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. The generation of random test data is comparatively cheap and does not require too much intellectual and computational effort [3]. It is mainly for this reason that various researchers have recommended R strategy in automated testing tools. YETI [4], AutoTest [5], Randoop [6] and Jartege [7] are some of the most common automated testing tools based on R strategy. Experiments performed by various researchers [8], [9], [10] 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+)

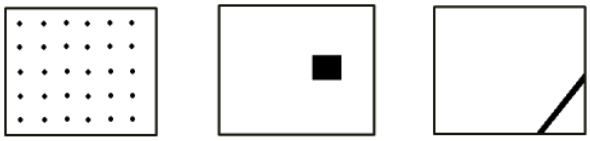
The random+ strategy [5] is an extension of the R strategy. It uses some special pre-defined values which can be simple boundary values or values that have high tendency of finding failures in the SUT. Boundary values are the values in the start and end of a particular type [11]. For instance, such values for *int* could be *MAX_INT, MAX_INT-1, MAX_INT-2, MAX_INT-3 0, MIN_INT, MIN_INT+1, MIN_INT+2,* and *MIN_INT+3*. The tester might also add some other special values that are considered effective in finding failures in the SUT. 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 failures.

C. Dirt Spot Sweeping (DSS)

Chan *et al*. [1] found that there are sub-domains of failure-causing inputs across the input domain. They divided these domains into three types called point, block and strip domains (Fig. 1) and argued that a strategy has more chances

DOI: 10.7763/LNSE.2014.V2.139

of hitting the failure domains if test cases are selected farther from each other. Other researchers [12]-[14] also tried to generate test cases further away from one another targeting these domains and achieved better performance.



a) Point domain b) Block domain c) Strip domain

Fig. 1. Failure domains across the input domain [2].

In DSS, if a value reveals failure from the block or strip domain 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 failure 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 neighboring 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 neighboring 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.

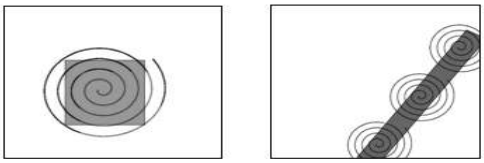


Fig. 2. DSSR strategy covering block and strip domain.

Fig. 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 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 [15]. 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.

Both the variables *currentFaults* and *oldFaults* are initialized to 0 at the start of the test, when a fault is found the *currentFaults* value is incremented which make the condition true. The flowchart presented in Fig. 3 depicts the case when fault 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 faults around the existing fault in the given SUT.

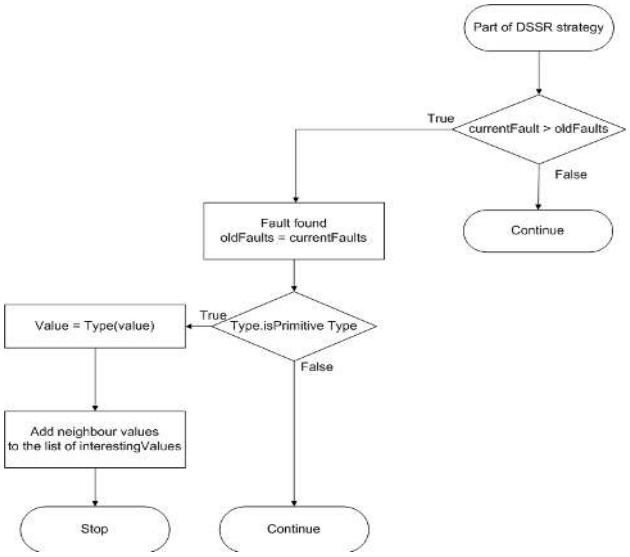


Fig. 3. Working mechanism of DSSR strategy.

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

TABLE I: DATA TYPES AND VALUES TO BE ADDED

Data Type	Values added
X is int, double, float, long, byte, short & char	X, X+1, X+2, X+3, X-1, X-2, X-3
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

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 and verify result.
public class Math1 {
  public void calc (int num1) {
    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:

Failure 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, 3, -1, -2, -3.

Failure 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 rule) to variable result1 that will lead to the violation of the next assertion (2).

Failure 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.

TABLE II: COMPARATIVE PERFORMANCE OF R, R+ AND DSSR STRATEGIES

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

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 faults starts again randomly in spite of the fact that the remaining faults are very close to the first fault.

III. Implementation of 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 [16], [17].

IV. Evaluation

The DSSR strategy is experimentally evaluated by comparing its performance with that of R and R+ strategy [5]. 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 [18]. 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. 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 failure. The traditional way is to compare the full stack traces and error types and use this as an equivalence class [8], [16] 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. 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 [16].

C. Performance Measurement Criteria

The literature review revealed that the F-measure is used where testing stops after identification of the first failure and the system is given back to the developers to remove the failure. Currently automated testing tools test the whole system and print all discovered failures 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 failures detected in the SUT by a particular number of test calls [8], [19]. This measurement, similar to E-measure, is effective because it considers the performance of the strategy when all other factors are kept constant.

V. Results

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 failures compared to R and R+ respectively. The minimum number of unique failures 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.

TABLE III: T-Test Results of the Classes Showing Different Results

S. No	Class Name	T-Test Results			Interpretation
		DSSR, R	DSSR, R+	R, R+	
1	AjTypeImpl	1	1	1	
2	Apriori	0.03	0.49	0.16	
3	CheckAssociator	0.04	0.05	0.44	DSSR better
4	Debug	0.03	0.14	0.56	
5	DirectoryScanner	0.04	0.01	0.43	DSSR better
6	DomParser	0.05	0.23	0.13	
7	EntityDecoder	0.04	0.28	0.3	
8	Font	0.18	0.18	1	DSSR = R > R+
9	Group	0.33	0.03	0.04	
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	

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. Fig. 4 presents the average improvement

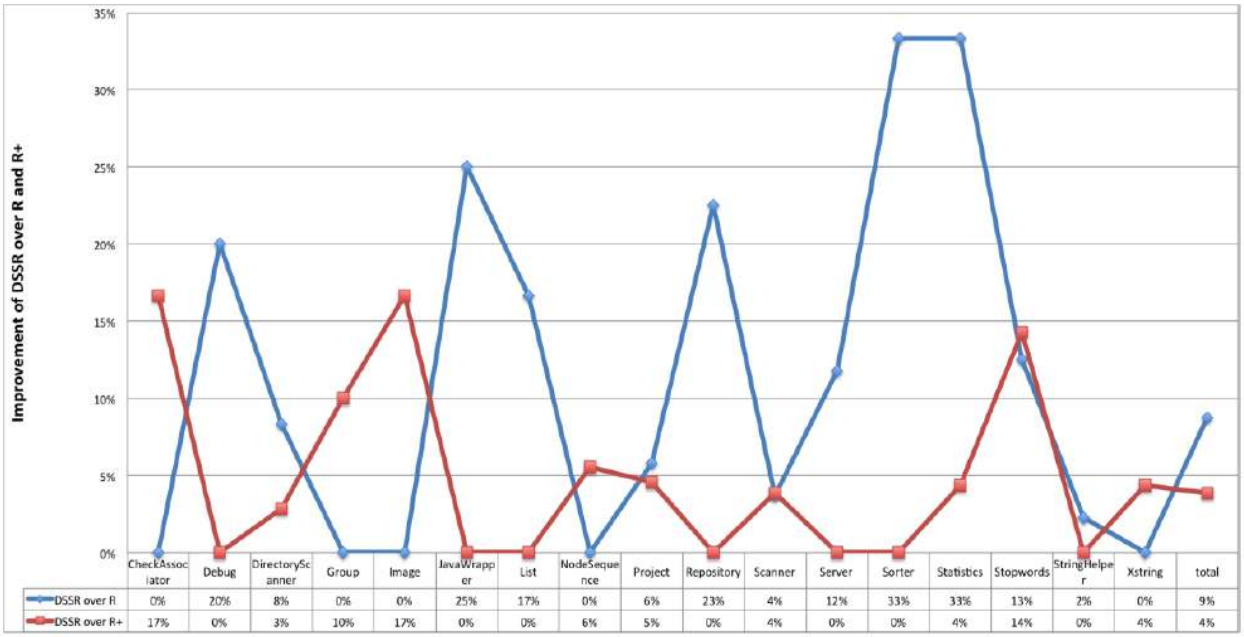
of DSSR strategy over R and R+ strategies for 17 classes showing substantial difference between DSSR and R, DSSR and R+. The blue line with diamond symbol shows the performance of DSSR over R and the red line with square symbols depicts the performance of DSSR over R+ strategy. The findings show that 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.

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 reveals that DSSR strategy has an edge over R and R+. This is mainly because of the additional feature of DSS in DSSR strategy, which can identify new faults quickly in the case of different faults or helps in debugging by providing more faults revealing input in the case of single fault. However, further analysis of DSSR strategy in terms of code coverage and overhead is required before picking it as a default strategy.



	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 R and R+ 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 [20], [21]. We used random testing as a baseline for comparison as advocated by Arcuri *et al.* [22]. In our experiments we selected projects from the Qualitas Corpus [18], which is a collection of open source java programs maintained for independent empirical research. The selection of programs in the current study is made through 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 DSS in the DSSR strategy. In DSSR strategy no pre-existing test cases are required because it utilizes 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. 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 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 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 failures in complex programs and Java libraries [23], [24]. Its simplicity, ease of implementation and efficiency in generating test cases make it one of the best choices for test

automation [10]. Some of the well-known automated tools based on R strategy include Jartege [7], Eclat [23], JCrasher [24], AutoTest [5] and YETI [16]. In pursuit of better test results and lower overhead, many variations of R strategy have been proposed. Adaptive random testing (ART) [2], Quasi-random testing (QRT) [14] and Restricted Random testing (RRT) [12] achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror Adaptive Random Testing (MART) [13] and Adaptive Random Testing through dynamic partitioning [27] increased the performance by reducing the overhead of ART. The main reason behind better performance 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 [25] 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 failures 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 failures.

Acknowledgment

The authors are thankful to the Department of Computer Science, University of York for physical and financial support. Thanks are also extended to Prof. Richard Paige for his valuable guidance, help and generous support.

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," in *Proc. Eighth International Conference on Qualify Software*, 2008, pp. 443.

[3] 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.

[4] M. Oriol and S. Tassis, "Testing .net code with yeti," in *Proc. 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.

[5] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," presented at the 40th Annual Hawaii International Conference on System Sciences, ser. HICSS '07, Washington, DC, USA: IEEE Computer Society, 2007.

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

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

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Experimental assessment of random testing for object-oriented software," in *Proc. the 2007 International Symposium on Software Testing and Analysis, ser. ISSTA '07,* New York, NY, USA: ACM, 2007, pp. 84–94.

[9] 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.

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

[11] B. Beizer, *Software Testing Techniques,* 2nd ed., New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[12] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proc. the 7th International Conference on Software Quality*, *ser. ECSQ '02*, London, UK, UK: Springer-Verlag, 2002, pp. 321–330.

[13] T. Y. Chen, F. C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," in *Proc. the Third International Conference on Quality Software, ser. QSIC '03*, Washington, DC, USA: IEEE Computer Society, 2003, pp. 4.

[14] T. Y. Chen and R. Merkel, "Quasi-random testing," in *Proc. the 20th IEEE/ACM International Conference on Automated Software Engineering, ser. ASE '05,* New York, NY, USA: ACM, 2005.

[15] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol, "On the effectiveness of test extraction without overhead," in *Proc. the 2009 International Conference on Software Testing Verification and Validation,* Washington, DC, USA: IEEE Computer Society, 2009.

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

[17] M. Oriol and Mian, "York Extensible Testing Infrastructure," Sept 2011.

[18] 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 *Proc. 2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010, pp. 336-345.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. the 29th international conference on Software Engineering, ser. ICSE '07,* Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84.

[20] 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.

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

[22] 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.

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

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

[25] 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.

[26] T. Chen, R. Merkel, P. Wong, and G. Eddy, "Adaptive random testing through dynamic partitioning," in *Proc. Fourth International Conference on Quality Software,* sept. 2004, pp. 79–86.

Mian Asbat Ahmad is a PhD scholar at the Department of Computer Science, the University of York, UK. He completed his M(IT) and MS(CS) from Agric. University Peshawar, Pakistan in 2004 and 2009 respectively. His research interests include new automated random software testing strategies.

Manuel Oriol is a lecturer at the Department of Computer Science, the University of York, UK and a principal scientist at ABB Corporate Research, Industrial Software Systems, in Baden-Daettwil, Switzerland. He completed his PhD from University of Geneva and an M.Sc. from ENSEEIHT in Toulouse, France. His research interests include software testing, software engineering, middleware, dynamic software updates, software architecture and real-time systems.