Automated Discovery of Failure Domain+

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

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

ABSTRACT

This paper presents a new, enhanced and improved form of automated random testing: the Dirt Spot Sweeping Random (DSSR) strategy. This strategy is based on the assumption that faults and unique failures reside in contiguous blocks and stripes. The DSSR strategy starts as a regular random+ testing strategy — a random testing technique with preference for boundary values. When a failure is found, it increases the chances of using neighbouring values of the failure in subsequent tests, thus slowly sweeping values around the failures found in hope of finding failures of different kind in its vicinity.

The DSSR strategy is implemented in the YETI random testing tool. It is evaluated against random (R) and random+ (R+) strategies by testing 60 classes (35,785 line of code) with one million (10^5) calls for each session, 30 times for each strategy. The results indicate that for 31 classes, all three strategies find the same unique failures. We analysed the 29 remaining classes using t-tests and found that for 7 classes DSSR is significantly better than both R+ and R, for 8 classes it performs similarly to R+ and is significantly better than R, and for 2 classes it performs similarly to random and is better than R+. In all other cases, DSSR, R+ and R do not perform significantly differently. Numerically, the DSSR strategy finds 43 more unique failures than R and 12 more unique failures than R+.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Comparison, Verification,

Keywords

software testing, automated random testing

1. INTRODUCTION

Testing is the most widely used and essential method of software testing. Therefore, the aim of researchers is to improve the effectiveness and efficiency of software testing process. Testing effectiveness is the increase in number of faults found in a specific number of test cases, which is achieved by upgradation of existing and development of new improved testing techniques. While the improve in testing efficiency is the decrease in the total time taken by a software testing process, which is most of the time accomplished by automation of the whole or a specific part of software testing process, like test data generation, execution and oracle formation.

Daikon [] is a tool that automate part of the testing process by auto generation of likely program invariants. The invariants are processed and can be annotated in the source code of the program to facilitate the testing process. While the tool helps to increase the efficiency, testing effectiveness is also dependant on it because the invariants produced by the tool are evaluated in the test execution to decide a pass and fail test case.

It is interesting to find out if we are compromising on quality by increasing the test efficiency. Therefore we set up and ran various experiments and analysed the results derived from the same set of error seeded programs tested with and without the assistance of Daikon. For this purpose, we proposed a new strategy named Automated Discovery of Failure Domain+. It is an upgraded strategy based on the combination of our two previous strategies Dirt Spot Sweeping Random (DSSR) and Automated Discovery of Failure Domain (ADFD), which discovers the fault surrounding values and graphically plot the fault domains respectively [].

The rest of this paper is organised as follows:
Section 2 describes the ADFD+ strategy. Section 3 presents implementation of the ADFD+ strategy. Section 4 explains the experimental setup. Section 5 shows results of the experiments. Section 6 discusses the results. Section 7 presents related work and Section 8, concludes the study.

2. AUTOMATED DISCOVERY OF FAILURE DOMAIN+

The new software testing technique named, Dirt Spot Sweeping Random (DSSR) strategy combines the random+ strategy with a dirt spot sweeping functionality. It is based on two intuitions. First, boundaries have interesting values and using these values in isolation can provide high impact on test results. Second, faults and unique failures reside in contiguous block and strip pattern. If this is true, DSS increase the performance of the test strategy. Before presenting the details of the DSSR strategy, it is pertinent to review briefly the Random and the Random+ strategy.

2.1 Random Strategy (R)

The random strategy is a black-box testing technique in which the SUT is executed using randomly selected test data. Test results obtained are compared to the defined oracle, using SUT specifications in the form of contracts or assertions. In the absence of contracts and assertions the exceptions defined by the programming language are used as test oracles. Because of its black-box testing nature, this strategy is particularly effective in testing softwares where the developers want to keep the source code secret [?]. The generation of random test data is comparatively cheap and does not require too much intellectual and computational efforts [?, ?]. It is mainly for this reason that various researchers have recommended random strategy for automated testing tools [?]. YETI [?, ?], AutoTest [?, ?], QuickCheck [?], Randoop [?], JArtege [?] are some of the most common automated testing tools based on random strategy.

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

Programs tested at random typically fail a large number of times (there are a large number of calls), therefore, it is necessary to cluster failures that likely represent the same fault. The traditional way of doing it is to compare the full stack traces and error types and use this as an equivalence class [?, ?] called a unique failure. This way of grouping failures is also used for random+ and DSSR.

2.2 Random Plus Strategy (R+)

The random+ strategy [?] 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 on the start and end of a particular type. For instance, such values for `int` could be `MAX_INT`, `MAX_INT-1`, `MAX_INT-2`; `MIN_INT`, `MIN_INT+1`, `MIN_INT+2`. Similarly, the tester might also add some other special values that he considers effective in finding faults for the SUT. For example, if a program under test has a loop from -50 to 50 then the tester can add -55 to -45, -5 to 5 and 45 to 55 to the predefined list of special values. This static list of interesting values is manually updated before the start of the test and has slightly high priority than selection of random values because of more relevance and high chances of finding faults for the given SUT. These special values have high impact on the results, particularly for detecting problems in specifications [?].

2.3 Dirt Spot Sweeping (DSS)

Chan et al. [?] found that there are patterns of failure-causing inputs across the input domain. Figure 1 shows these patterns for two dimensional input domain. They divided these patterns into three types called points, block and strip patterns. The black area (points, block and strip) inside the box show the input which causes the system to fail while white area inside the box represent the genuine input. Boundary of the box (black solid line) surrounds the complete input domain and represents the boundary values. They argue that a strategy has more chances of hitting these fault patterns if test cases far away from each other are selected. Other researchers [?, ?, ?], also tried to generate test cases further away from one another targeting these patterns and achieved better performance. Such increase in performance indicate that faults more often occur contiguous across the input domain. In Dirt Spot Sweeping we propose that if a value reveals fault from the block or strip pattern then for the selection of the next test value, DSS may not look farthest away from the known value and rather pick the closest test value to find another fault from the same region.

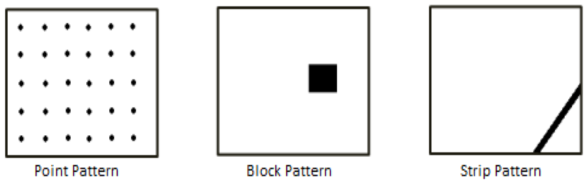


Figure 1: Failure patterns across input domain [?]

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

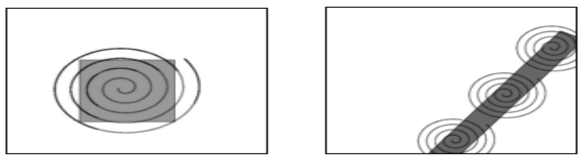


Figure 2: DSSR covering block and strip pattern

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.

2.4 Structure of the Dirt Spot Sweeping Random Strategy

The DSSR strategy continuously tracks the number of failures during the execution of the test. This tracking is done in a very effective way with zero or minimum overhead to keep the overhead up to bare minimum [?]. 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.

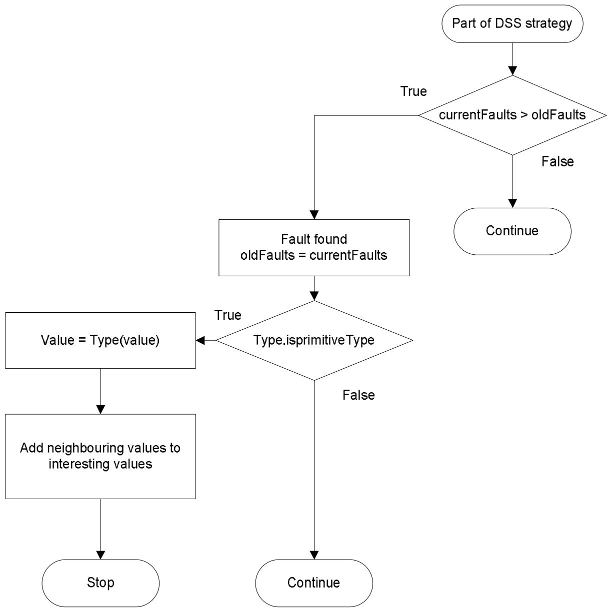


Figure 3: Working mechanism of DSSR Strategy

The flowchart presented in Figure 3 depicts that, when the failure finding value is of primitive type, the DSSR strategy identifies its type and add values only of that particular type to the list of interesting values. The resultant list of interesting values provide relevant test data for the remaining test session and the generated test cases are more targeted towards finding new failures around the existing failures in the given SUT.

Boundary and other special values that have a high tendency of finding faults in the SUT are added to the list of interesting values by random+ strategy prior to the start of test session where as in DSSR strategy the fault-finding and its surrounding values are added at runtime when a failure is found.

Table 1 presents the values are added to the list of interesting values when a failure is found. In the table the test value is represented by X where X can be int, double, float, long, byte, short, char and String. All values are converted to their respective types before adding them to the list of interesting values.

2.5 Explanation of DSSR strategy on a concrete example

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)

Table 1: Neighbouring values for primitive types and String

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 in the given program below followed by description of how the strategy perform execution.

```
/**
* Calculate square of given number
* and verify results.
* The code contain 3 faults.
* @author (Mian and Manuel)
*/
public class Math1 {
 public void calc (int num1) {
  // Square num1 and store result.
  int result1 = num1 * num1;
  int result2 = result1 / num1; // 1
  assert Math.sqrt(result1) == num1; // 2
  assert result1 >= num1; // 3
 }
}
```

In the above code, one primitive variable of type `int` is used, therefore, the input domain for DSSR strategy is from `-2,147,483,648 to 2,147,483,647`. The strategy further select values (`0, Integer.MIN_VALUE` & `Integer.MAX_VALUE`) as interesting values which are prioritised for selection as inputs. As the test starts, three faults are quickly discovered by DSSR strategy in the following order.

Fault 1: The strategy select value `0` for variable `num1` in the first test case because `0` is available in the list of interesting values and therefore its priority is higher than other values. This will cause Java to generate division by zero exception (1).

Fault 2: After discovering the first fault, the strategy adds it and its surrounding values to the list of interesting values i.e. `0, 1, 2, 3 and -1, -2, -3` in this case. In the second test case the strategy may pick `-3` as a test value which may lead to the second fault where assertion (2) fails because the square root of `9` is `3` instead of the input value -3.

Fault 3: After a few tests the strategy may select `Integer.MAX_VALUE` for variable `num1` from the list of interesting values leading to discovery of the 3rd fault because int variable `result1` will not be able to store the square of `Integer.MAX_VALUE`. Instead of the actual square value Java assigns a negative value (Java language rule) to variable result1 that will lead to the violation of the next assertion (3).

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

3. IMPLEMENTATION OF THE DSSR STRATEGY

Implementation of the DSSR strategy is made in the YETI open-source automated random testing tool. YETI, coded in Java language, is capable of testing systems developed in procedural, functional and object-oriented languages. Its language-agnostic meta model enables it to test programs written in multiple languages including Java, C#, JML and .Net. The core features of YETI include easy extensibility for future growth, high speed (up to one million calls per minute on java code), real time logging, real time GUI support, capability to test programs with multiple strategies and auto generation of test report at the end of test session. For large-scale testing there is a cloud-enabled version of YETI, capable of executing parallel test sessions in Cloud [?]. A number of hitherto faults have successfully been found by YETI in various production softwares [?, ?].

YETI can be divided into three decoupled main parts: the core infrastructure, language-specific bindings and strategies. The core infrastructure contains representation for routines, a group of types and a pool of specific type objects. The language specific bindings contain the code to make the call and process the results. The strategies define the procedure of selecting the modules (classes), the routines (methods) and generation of values for instances involved in the routines. By default, YETI uses the random strategy if no particular strategy is defined during test initialisation. It also enables the user to control the probability of using null values and the percentage of newly created objects for each test session. YETI provides an interactive Graphical User Interface (GUI) in which users can see the progress of the current test in real time. In addition to GUI, YETI also provides extensive logs of the test session for more in-depth analysis.

The DSSR strategy is an extension of YetiRandomPlusStrategy, an extended form of the YetiRandomStrategy. The class hierarchy is shown in Figure 4.

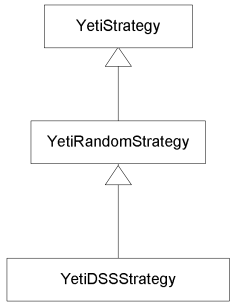


Figure 4: Class Hierarchy of DSSR in YETI

4. EVALUATION

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

4.1 Research questions

For evaluating the DSSR strategy, the following research questions have been addressed in this study:

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

4.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [?]. The Qualitas Corpus is a curated collection of open source java projects built with the aim of helping empirical research on software engineering. These projects have been collected in an organised form containing the source and binary forms. Version 20101126, which contains 106 open source java projects is used in the current evaluation. In our experiments we selected 60 random classes from 32 random projects. All the selected classes produced at least one fault and did not time out with maximum testing session of 10 minutes. Every class is tested thirty times by each strategy (R, R+, DSSR). Name, version and size of the projects to which the classes belong are given in table 2 while test details of the classes is presented in table 3. Line of Code (LOC) tested per class and its total is shown in column 3 of table 3.

Every class is evaluated through 10^5 calls in each test session.[1] Because of the absence of the contracts and assertions in the code under test, Similar approach as used in previous studies [?] is followed using undeclared exceptions to compute unique failures.

All tests are performed with a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon processor

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

S. No	Project Name	Version	Size (MB)
1	apache-ant	1.8.1	59
2	antlr	3.2	13
3	aoi	2.8.1	35
4	argouml	0.30.2	112
5	artofillusion	281	5.4
6	aspectj	1.6.9	109.6
7	axion	1.0-M2	13.3
8	azureus	1	99.3
9	castor	1.3.1	63.2
10	cayenne	3.0.1	4.1
11	cobertura	1.9.4.1	26.5
12	colt	1.2.0	40
13	emma	2.0.5312	7.4
14	freecs	1.3.20100406	11.4
15	hibernate	3.6.0	733
16	hsqldb	2.0.0	53.9
17	itext	5.0.3	16.2
18	jasml	0.10	7.5
19	jmoney	0.4.4	5.3
20	jruby	1.5.2	140.7
21	jsXe	04_beta	19.9
22	quartz	1.8.3	20.4
23	sandmark	3.4	18.8
24	squirrel-sql	3.1.2	61.5
25	tapestry	5.1.0.5	69.2
26	tomcat	7.0.2	24.1
27	trove	2.1.0	18.2
28	velocity	1.6.4	27.1
29	weka	3.7.2	107
30	xalan	2.7.1	85.4
31	xerces	2.10.0	43.4
32	xmojo	5.0.0	15

Table 2: Name and versions of 32 Projects randomly selected from the Qualitas Corpus for the experiments

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.

4.3 Performance measurement criteria

Various measures including the E-measure (expected number of failures detected), P-measure (probability of detecting at least one failure) and F-measure (number of test cases used to find the first fault) have been used by researchers to find the effectiveness of the random test strategy. The E-measure and P-measure have been heavily criticised [?] and are not considered effective measuring techniques while the F-measure has been often used by various researchers [?, ?]. In our initial experiments the F-measure is used to evaluate the efficiency. However it was realised that this is not the right choice. In some experiments a strategy found the first fault quickly than the other but on completion of test session that very strategy found lower number of total faults than the rival strategy. The preference given to a strategy by F-measure because it finds the first fault quickly without giving due consideration to the total number of faults is not fair [?].

The literature review revealed that the F-measure is used where testing stops after identification of the first fault and the system is given back to the developers to remove the fault. Currently automated testing tools test the whole system and print all discovered faults in one go therefore, F-measure is not the favourable choice. In our experiments, performance of the strategy is measured by the maximum number of faults detected in SUT by a particular number of test calls [?, ?, ?]. This measurement is effective because it considers the performance of the strategy when all other factors are kept constant.

5. RESULTS

Results of the experiments including class name, Line of Code (LOC), mean value, maximum and minimum number of unique failures and relative standard deviation for each of the 60 classes tested using R, R+ and DSSR strategy are presented in Table 3. Each strategy found an equal number of faults in 31 classes while in the remaining 29 classes the three strategies performed differently from one another. The total of mean values of unique failures in DSSR (1075) is higher than for R (1040) or R+ (1061) strategies. DSSR also finds a higher number of maximum unique failures (1118) than both R (1075), and R+ (1106). DSSR strategy finds 43 and 12 more unique faults compared to R and R+ respectively. The minimum number of unique faults found by DSSR (1032) is also higher than for R (973) and R+ (1009) which attributes to higher efficiency of DSSR strategy over R and R+ strategies.

5.1 Is there an absolute best among R, R+ and DSSR strategies?

Based on our findings DSSR is at least as good as R and R+ in almost all cases, it is also significantly better than both R and R+ in 12% of the classes. Figure 5 presents the average improvements of DSSR strategy over R and R+ strategy over the 17 classes for which there is a significant difference between DSSR and R or R+. The blue line with diamond symbol shows performance of DSSR over R and the red line with square symbols depicts the improvement of DSSR over R+ strategy. The classes where blue line with diamond symbols show the improvement of DSSR over R and red line with square symbols show the improvement of DSSR over R+.

The improvement of DSSR over R and R+ strategy is calculated by applying the formula (1) and (2) respectively.

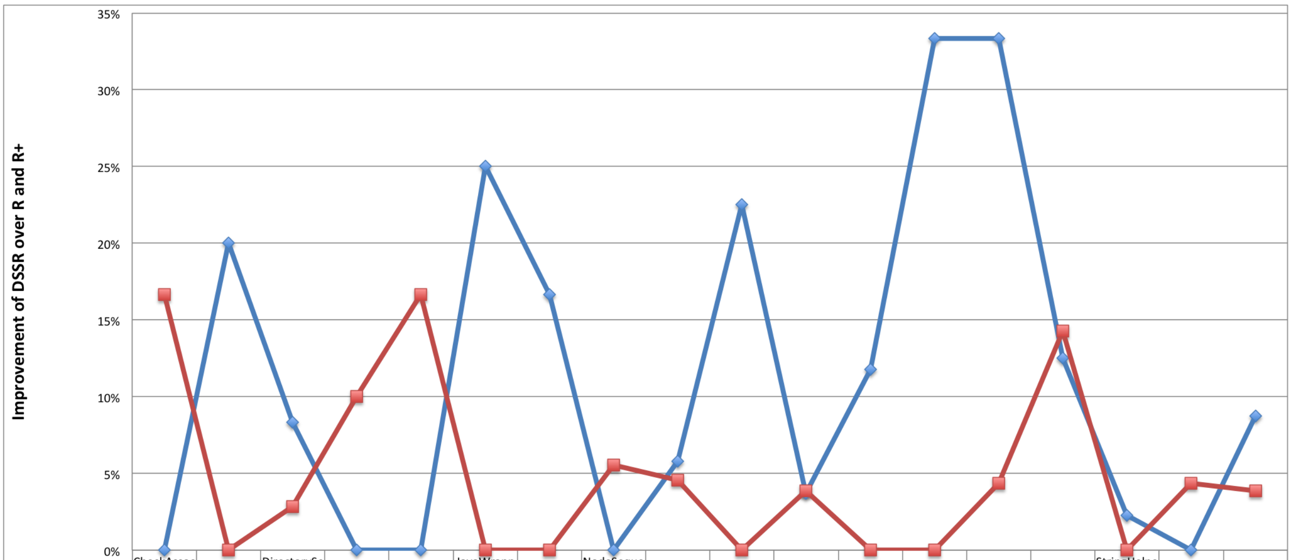
$$\frac{Averagefaults_{(DSSR)} - Averagefaults_{(R)}}{Averagefaults_{(R)}} * 100 \qquad (1)$$

$$\frac{Averagefaults_{(DSSR)} - Averagefaults_{(R+)}}{Averagefaults_{(R+)}} * 100 \qquad (2)$$

The findings show that DSSR strategy perform up to 33% better than R and up to 17% better than R+ strategy. In some cases DSSR perform equally well with R and R+ but in no case DSSR performed lower than R and R+. Based on the results it can be stated that DSSR strategy is a better choice than R and R+ strategy.

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 3: Complete results for R, R+ and DSSR. Results present Serial Number (S.No), Class Name, Line of Code (LOC), mean, maximum number of faults, minimum number of faults and relative standard deviation for each Random (R), Random+ (R+) and Dirt Spot Sweeping Random (DSSR) strategies.



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

Figure 5: Improvement of DSSR strategy over Random and Random+ strategy.

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

T-tests applied to the data given in Table 4 show that DSSR is significantly better in 7 classes from R and R+ strategy, in 8 classes DSSR performed similarly to R+ but significantly higher than R, and in 2 classes DSSR performed similarly to R but significantly higher than R+. There is no case R and R+ strategy performed significantly better than DSSR strategy. Expressed in percentage: 72% of the classes do not show significantly different behaviours whereas in 28% of hte classes, the DSSR strategy performs significantly better than at least one of R and R+. It is interesting to note that in no single case R and R+ strategies performed better than DSSR strategy. We attribute this to DSSR possessing the qualities of R and R+ whereas containing the spot sweeping feature.

5.3 Can we pick the best default strategy between R, R+ and DSSR?

Analysis of the experimental data reveal that DSSR strategy has an edge over R and R+. This is because of the additional feature of Spot Sweeping in DSSR strategy.

In spite of the better performance of DSSR strategy compared to R and R+ strategies the present study does not provide ample evidence to pick it as the best default strategy because of the overhead induced by this strategy (see next section). Further study might give conclusive evidence.

6. DISCUSSION

In this section we discuss various factors such as the time taken, effect of test duration, number of tests, number of faults in the different strategies and the effect of finding first fault in the DSSR strategy. **Time taken to execute an equal number of test cases:** The DSSR strategy takes slightly more time (up to 5%) than both pure random and random plus which may be due to maintaining sets of interesting values during the execution. We do not believe that the overhead can be reduced.

Effect of test duration and number of tests on the results: All three techniques have the same potential for finding failures. If testing is continued for a long duration then all three strategies will find the same number of unique failures and the results will converge. We suspect however that some of the unique failures will take an extremely long time to be found by using random or random+ only. Further experiments should confirm this point.

Effect of number of faults on results: We found that the DSSR strategy performs better when the number of faults is higher in the code. The reason seems to be that when there are more faults, their domains are more connected and DSSR strategy works better. Further studies might use historical data to pick the best strategy.

Dependence of DSSR strategy to find the first unique failure early enough: During the experiments we noticed that if a unique failure is not found quickly enough, there is no value added to the list of interesting values and then the test becomes equivalent to random+ testing. This means that better ways of populating failure-inducing values are needed for sufficient leverage to DSSR strategy. As an example, the following piece of code would be unlikely to fail under the current setting:

```
public void test(float value){
 if(value == 34.4445)   10/0;
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained from the constant pool in the class

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 4: T-test results of the classes showing different results

files of the SUT.

In the example above the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the unique failure right away.

DSSR strategy and coverage: Random strategies typically achieve high level of coverage [?]. It might also be interesting to compare R, R+ and DSSR with respect to the achieved coverage or even to use a DSSR variant that adds a new interesting value and its neighbours when a new branch is reached.

Threats to validity: As usual with such empirical studies, the present work might suffer from a non-representative selection of classes. The selection in the current study is however made through random process and objective criteria, therefore, it seems likely that it would be representative.

The parameters of the study might also have prompted incorrect results. But this is unlikely due to previous results on random testing [?].

7. RELATED WORK

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

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [?, ?, ?, ?, ?]. Adaptive random testing (ART), Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror ART and ART through dynamic partitioning increased the performance by reducing the overhead of ART. The main reason behind better performance of the strategies is that even spread of test input increases the chance of exploring the fault patterns present in the input domain.

A more recent research study [?] stresses on the effectiveness of data regeneration in close vicinity of the existing test data. Their findings showed up to two orders of magnitude more efficient test data generation than the existing techniques. Two major limitations of their study are the requirement of existing test cases to regenerate new test cases, and increased overhead due to "meta heuristics search" based on hill climbing algorithm to regenerate new data. In DSSR no pre-existing test cases are required because it utilises the border values from R+ and regenerate the data very cheaply in a dynamic fashion different for each class under test without any prior test data and with comparatively lower overhead.

The random+ (R+) strategy is an extension of the random strategy in which interesting values, beside pure random values, are added to the list of test inputs [?]. These interesting values includes border values which have high tendency of finding faults in the given SUT [?]. Results obtained with R+ strategy show significant improvement over random strategy [?]. DSSR strategy is an extension of R+

strategy which starts testing as R+ until a fault is found then it switches to spot sweeping.

A common practice to evaluate performance of an extended strategy is to compare the results obtained by applying the new and existing strategy to identical programs [?, ?, ?]. Arcuri et al. [?], stress on the use of random testing as a baseline for comparison with other test strategies. We followed the procedure and evaluated DSSR strategy against R and R+ strategies under identical conditions.

In our experiments we selected projects from the Qualitas Corpus [?] which is a collection of open source java programs maintained for independent empirical research. The projects in Qualitas Corpus are carefully selected that spans across the whole set of java applications [?, ?, ?].

8. CONCLUSIONS

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases. We developed a new strategy named. "DSSR strategy" as an extension of R+, based on the assumption that in a significant number of classes, failure domains are contiguous or located closely. The DSS strategy, a strategy which adds neighbouring values of the failure finding value to a list of interesting values, was implemented in the random testing tool YETI to test 60 classes, 30 times each, from Qualitas Corpus with each of the 3 strategies R, R+ and DSSR. The newly developed DSSR strategy uncovers more unique failures than both random and random+ strategies with a 5% overhead. We found out that for 7 (12%) classes DSSR was significantly better than both R+ and R, for 8 (13%) classes DSSR performed similarly to R+ and significantly better than R, while in 2 (3%) cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Overall, DSSR yields encouraging results and advocates to develop the technique further for settings in which it is significantly better than both R and R+ strategies.