Dirt Spot Sweeping Random Strategy

Mian Asbat Ahmad
Department of Computer Science
The University of York
ma@cs.york.ac.uk

Manuel Oriol
Department of Computer Science
The University of York
manuel@cs.york.ac.uk

***Abstract*—In this article, an enhanced and improved form of random testing called Dirt Spot Sweeping Random (DSSR) Strategy is introduced. DSSR strategy not only combines ordinary random strategy and random plus strategy to achieve their combined benefits but also sweeps the dirt spots in the program code for faults. It is based on two intuitions, first is that boundaries have interesting values and using these values in isolation can produce high impact on test results while second is that faults reside in block and strip pattern thus using neighbouring values of the fault finding value can reveal more faults quickly which consequently increase the test performance. DSSR is implemented in an open source automated random testing tool called York Extensible Testing Infrastructure (YETI). Several experiments were performed on different open source Java programs from Qualitas Corpus and results showed that DSSR perform up to 30% better than pure random testing.**

***Keywords*-random; testing; automated testing; YETI;**

I. Introduction

The rapid increase in software development in today's modern world prompts the need for quick and efficient testing to ensure high quality in minimum consumption of time. To meet the chalange researchers are trying their best to develop new and improve the existing techniques to make them capable of finding more faults with few number of test cases. Random testing is one such technique which is highly efficient, consumes less computation power and can be easily automated.

Random testing is a black-box testing technique in which the Software Under Test (SUT) is executed against randomly selected test data. Test results obtained are compared either against the oracle defined, using SUT specifications in the form of assertions or exceptions defined by the programming language. The generation of random test data is comparatively cheap and does not require too much intellectual and computation efforts [**?**], [**?**]. It is mainly for this reason that various researchers have recommended this strategy for incorporation in automatic testing tools [**?**]. YETI [**?**], [**?**], AutoTest [**?**], [**?**], QuickCheck [**?**], Randoop [**?**], JArtage [**?**] are a few of the most common tools based on random strategy.

The 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 but in science there is no substitute for experimental analysis and later on various experiments performed by different researchers including [**?**], [**?**], [**?**] experimentally proved that random testing is simple to implement, cost effective, highly efficient and free from human bias as compared to the rival techniques.

The researchers further found that the performance of random testing can be further increased by slightly altering the technique of test case selection. In adaptive random testing [**?**] Chen et al found that the performance of random testing increases by up to 50% when test input is selected evenly which is spread across the whole input domain. Similarly Restricted Random Testing [**?**], Feedback directed Random Test Generation [**?**], Mirror Adaptive Random Testing [**?**] and Quasi Random Testing [**?**] also stress on the need of test case selection farthest away from one another for better results.

Chen et al. [**?**] further found that there are patterns of failure causing inputs across the input domain. They divided these patterns into three types called block, point and strip patterns. They also said that there were more chances of hitting the fault patterns if test cases far away from each other were selected. Various other researchers [**?**], [**?**], [**?**] also tried to generate test cases further away from one another targeting these patterns. Failure patterns are given in Figure **??**.

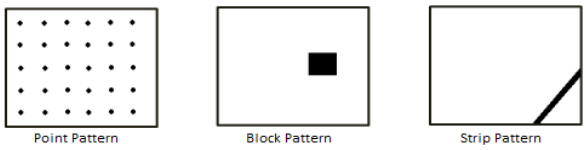

Figure 1. Failure patterns across input domain [**?**]

Random plus [**?**] is an updated version of the pure random strategy. It is a random strategy that uses some special predefined values which can be simple border

values or values that have high tendency of finding faults in the SUT. Boundary values [**?**] are the values on the start, end and middle of particular type. For instance, zero, Integer.MAX_VALUE and Integer.MIN_VALUE can be called as border values for Integer. Similarly the tester might also add some other special values that he consider effective for the current SUT. For example, if a program under test has a loop from 1 to 100 then the tester can add 100, 101, 99, 51, 50, 49, -1, 0 and 1 etc to the pre-defined list of special values in order to be selected for a test. It was found [**?**] that these special values have high impact on the results particularly detecting problems in specifications.

II. Dirt Spot Sweeping Random Strategy

Dirt Spot Sweeping Random (DSSR) strategy is a new random test startegy developed during this study. DSSR is the combination of boundary values, pure random and the spot sweeping. It is based on two intuition. Intuition No 1 is that boundaries has interesting values and using these values in isolation can provide high impact on test results. While intuition No 2 is that faults can reside in block and strip pattern thus using neighbouring values of the fault value can further increase the performance of the test strategy. One thing to note is that DSSR add border values before the test starts whereas neighbour values of the fault are added to the list of interesting values at run time when they are found during testing.

Initially, the DSSR strategy was not using boundary values and the list of interesting values was empty at the start of the test. In the earlier version the test had to start with random testing and once the fault was found in the system, it used to transfer the values surrounding a finding value to the list of interesting values. In this way the list of interesting values was populated and the strategy now had to look for values in the list before trying to get random value. The bottleneck in this strategy was that DSSR had to wait for the random testing to find the fault in the system. This bottleneck was removed by introducing border values in DSSR while keeping the remaining system the same. The border values are added to the list of interesting values before the test start thus the system not only selects purely random values but also check for values from boundary values, which increases the fault finding chances and consequently add more values to the list of interesting values [**?**].

Dirt Spot Sweeping is explained with the help of flowchart in Figure **??**. In this process the program continuously track the number of faults and once a fault is found in the SUT, the program not only copy the values of the test case but also copy its surrounding values to the variable list of interesting values. From the flowchart it is clear that if the fault finding value is of a primitive type then the strategy only add objects of that type to the list. Doing this increases size of the list of interesting values for testing thus providing more chances of finding faults.

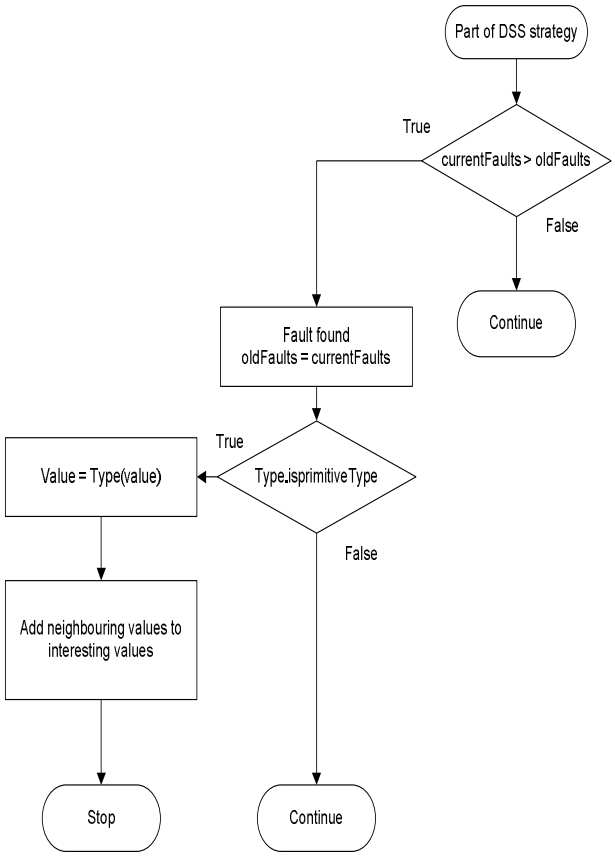

Figure 2. Working mechanism of DSSR Strategy

Pre-defined special values are added by random+ prior to testing but to sweep the failure pattern we need to add fault neighbouring values at run time after a fault is found in the system. The following table shows the values that are added to the list of interesting values when a fault is found by the test value X while X can be int, double, float, long, byte, short, char and String. All values are converted to their respective types before addition to the list of interesting values.

III. Motivating Example

We wrote a simple program to show how DSSR works and how special pre-defined values help in improving test performance.

```
public class Math {
```

Table I
NEIGHBOURING VALUES FOR PRIMITIVE TYPES AND STRING

Type	Values
X is int, double, float, long, byte short and char	X [[X-5 , X-1]] [[X+1 , X+5]]
X is String	X X + " " " " + X X.toUpperCase() X.toLowerCase() X.trim() X.substring(2) X.substring(1, X.length() - 1)

```
public void calc ( int a) {
//Square the value and assign it to result.
1.      int result1 = a * a;
//To check that the value of result is positive.
2.      assert result1 > a;
//To check that the revert of result is the received value.
3.      assert Math.sqrt(result1) = a;
4.      int result2 = result1 / a;
} }
```

From the above code we can see that only one primitive type is used and that is "int" and we also know that for "int" we have pre-defined values of 0, Integer.MAX_VALUE and Integer.MIN_VALUE in the list of interesting values. On starting the test we will immediately find the first fault.

Fault 1: The DSSR might select value 0 for variable "a" in the first test case because 0 is available in the list of interesting values. This will cause statement 1, 2, and 3 to pass but at statement 4 we will get a failure which is Division by zero.

Fault 2: When DSSR caught the failure it will add this and the surrounding values to the list of interesting values which includes 0, 1, 2, 3, 4, 5 and -1, -2, -3, -4, -5. Now for the second test case DSSR may pick -5 as a test value and it will take us to our second fault where assertion no 3 will fail because the square root of that number will be +5 instead of -5.

Fault 3: After a few test cases the DSSR might select Integer.MAX_VALUE for "a" which is also available in the list of interesting values and it will lead us to our 3rd fault because result1 will not be able to store the square of Integer.MAX_VALUE or instead of the actual square value Java will assign a negative value to result1 which will lead to the violation of assertion 2.

Thus we can see that, in this example, pre-defined values including border values, fault finding values and its surrounding values can find the faults quickly and in less number of tests.

IV. IMPLEMENTATION OF DSSR

The DSSR was implemented in YETI tool [**?**]. YETI is an automatic random testing tool developed in Java. It is an open source tool [25] capable of testing both procedural and object-oriented software. Its language-agnostic meta model enables it to test programs written in multiple languages including Java, C#, JML and .Net. The core features of YETI includes easy extensibility for future growth, speed of up to one million calls per minute on java code, real time logging, real time GUI support, ability to test programs using multiple strategies and auto generation of test report at the end of the test session.

YETI can be divided into three main sections that includes core infrastructure, startegies and language-specific bindings. The core infrastructure represents routines, a group of types and a pool of specific typed objects. The strategies section define the way to select the class/module to test, random selection of routine/method from that module and getting instance of the required type during testing. The third and final section language binding contains the code to make the call and process the results.

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

YETI also provide an interactive Graphical User Interface (GUI) where a user can see the progress of the current test in real time. Beside GUI YETI also provides extensive logs of the test session which are very helpful in fault tracking.

V. EXPERIMENTS AND ANALYSIS

In our experiments we tested 10 classes from the java.lang and java.util packages. Classes were selected randomly and each class was tested 10 times by pure random and DSSR strategy in order to have more consistent results. There is a handy option in YETI front end where user can right click any of the graph and save the data of the specific feature in a text file. The option was utilized to store the unique failures found in each experiment for further

analysis. Another feature called "Interesting value injection probability" gives control on the list of interesting values. This property set a specific number of calls, out of total calls, for which values should be selected from the list of interesting values. For all our experiments interesting value injection probability was set to 0.1, which means that 10% of the test cases will be selected from the list of interesting values while remaining 90% of the values will be selected randomly. Commands for executing the experiments using pure random and DSSR are as follows. Since pure random is the default strategy for testing in YETI therefore there is no need to mention that strategy in command.

java yeti.Yeti -java -testModules=java.lang.String -nTests=100000 -nologs -gui -DSS

java yeti.Yeti -java -testModules=java.lang.String -nTests=100000 -nologs -gui

Various options used in the above commands have the following meaning:
yeti.Yeti represents the package name (yeti) and the main class (YETI).
-java is used to show that the program under test is written in Java language.
-testModules points to the class under test, which is String in this case.
-nTests is used to execute specified number of test calls in that session, which is 100,000 in our case for both of the strategies.
-nologs is used so that logs are not printed on the screen during test execution to avoid extra load on processor.
-gui is used to show the real time test results in Graphical User Interface.

All tests were performed using 64-bit Microsoft Windows 7 Enterprise Service Pack 1 running on Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz with 4.00 GB RAM. Furthermore, Java(SE) Runtime Environment [Version 6.1.7601] was used.
Each test is explained with the help of a table and a figure. Table shows the results of the 10 tests performed using random and DSSR strategy while the figure shows the summary of all the tests. Each experiment is briefly explained.

A. Performance Measurement

Various measures including F-measure, P-measure and E-measure have been used by researchers to find the effectiveness of the test strategy in random testing. Initially the F-measure (Number of test cases used to find the first fault) was used [**?**] [**?**] but after several experiments it became evident that this was not the right choice because in some experiments a strategy found first fault quickly than the second but after the complete test session first strategy found lower number of total faults than the second competitor. Similarly F-measure of random strategy can be easily increased to high number by including conditional statements in a program under test. For example in the following program it is slightly difficult to generate the exact number quickly for a random testing as compared to other test strategies.

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

Therefore in all our experiments performance of the strategy was measured in terms of finding maximum number of faults in a particular number of test calls [**?**] [**?**] [**?**] which in our case was set to 100,000 calls per class. These number of test calls were kept constant for both pure random and DSSR strategies in all the experiments. This measurement was found effective because it clearly measure the performance of the strategy when all the other factors were kept constant.

VI. Evaluation

To evaluate the new DSSR strategy we performed extensive experiments in which the performance of DSSR was compared with pure random testing. Performance is measured in terms of the ability of a strategy to find maximum number of faults in fixed number of tests. Number of tests were kept constant at 500 while additionally time to execute 500 tests by each strategy was also measured. To perform testing free from any bias we selected 45 java projects from Qualitas Corpus [**?**].

VII. Results & Discussions

VIII. Conclusion

The conclusion goes here. this is more of the conclusion

Acknowledgment

The authors would like to thank... more thanks here

References

[1] 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.

[2] 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), pp. 157–166, IEEE Computer Society, 2008.

[3] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: adaptive random testing for object-oriented software," in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (New York, NY, USA), pp. 71–80, ACM, 2008.

[4] 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*, ICECCS '10, (Washington, DC, USA), pp. 264–265, IEEE Computer Society, 2010.

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

[6] 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*, HICSS '07, (Washington, DC, USA), pp. 261a–, IEEE Computer Society, 2007.

[7] 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*, ISSTA '07, (New York, NY, USA), pp. 84–94, ACM, 2007.

[8] 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*, ICFP '00, (New York, NY, USA), pp. 268–279, ACM, 2000.

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

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

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

[12] J. W. Duran and S. Ntafos, "A report on random testing," in *Proceedings of the 5th international conference on Software engineering*, ICSE '81, (Piscataway, NJ, USA), pp. 179–183, IEEE Press, 1981.

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

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

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

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

[17] 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*, ICSE '07, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2007.

[18] 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*, QSIC '03, (Washington, DC, USA), p. 4, IEEE Computer Society, 2003.

[19] T. Y. Chen and R. Merkel, "Quasi-random testing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, (New York, NY, USA), pp. 309–312, ACM, 2005.

[20] T. Y. Chen and F.-C. Kuo, "Is adaptive random testing really better than random testing," in *Proceedings of the 1st international workshop on Random testing*, RT '06, (New York, NY, USA), pp. 64–69, ACM, 2006.

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

[22] C. Kaner, "Software testing as a social science," tech. rep., Florida Institute of Technology, July 2004.

[23] 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, pp. 109 –119, feb 1996.

[24] 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*, pp. 146 – 153, sept. 2004.

[25] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer, "On the predictability of random tests for object-oriented software," in *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, (Washington, DC, USA), pp. 72–81, IEEE Computer Society, 2008.