Automated Discovery of Failure Domain+ and Daikon to Analyse Boundaries

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 verify the accuracy of invariants generated automatically by Daikon and suggests how to improve their quality. To achieve this, it uses a newly developed technique called Automated Discovery of Failure Domain+ (ADFD+). It is a testing framework that after identifying a failure search its surrounding values to the specified range. The result obtained is presented graphically indicating pass and fail points.

Several error seeded two dimensional programs were tested in the experiments with point block and strip fault domain were evaluated first with ADFD+ and than with Daikon. On comparison of results from both the methods it is found that where Daikon generates the correct invariants, it was not good enough to identify the fault boundaries.

It is concluded that it will be highly effective if the fault boundary found by ADFD+ is passed to the Daikon in the first stage then the invariants generated by Daikon will correctly point to the fault boundary.

Keywords

software testing, automated random testing

1. INTRODUCTION

Testing is the most widely used and essential method of software testing. Ample efforts have been made to improve its effectiveness and efficiency. Testing is effective if it finds maximum number of faults in minimum number of test cases. Testing efficiency is the process to execute maximum number of test cases in minimum possible time. It is concluded that automation is the key to achieve both effectiveness and efficiency of software testing. Automation of a single part like data generation, execution, oracle or the whole system is highly beneficial in most of the cases. Some of the common automated testing tools include YETI [24], Randoop [32], Eclat [30], QuickCheck [13] and Jaretege [23].

Daikon [16] is one of the several tools, which uses machine learning technique to automatically generate likely invariants of the program written in C, C++, Java and Pearl. Daikon takes as input the program and few test cases written manually or generated by an automated tool. It executes the test cases on the program under test and observes the values that the program computes. At the end of the test session it reports the properties that were true over the observed executions. Daikon can process the generated invariants to mitigate non interesting and redundant invariants. Daikon can also inserts the generated invariants in to the source code as assertions. Daikon's output can be useful in understanding program, generating invariants, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistent data structures and checking the validity of data streams.

While the idea behind Daikon is attractive, it is interesting to see how much the auto generated invariants represent the fault domain residing in the program. To assess this, we set up performed several experiments and analysed the results derived from the error-seeded programs tested with Daikon and ADFD+. ADFD+ is a framework named Automated Discovery of Failure Domain+. It is based on our previous strategy Automated Discovery of Failure Domain (ADFD), which tries to find a fault, search the surrounding for more faults and graphically plot the fault domain if any [1].

The main contribution of the article are:

- **ADFD+:** It brings some changes to the previously proposed ADFD strategy. The new strategy improves the search algorithm of ADFD and make the report more intuitive.
- **Implementation of ADFD+:** The new ADFD+ strategy is implemented and integrated in the YETI tool.
- **Evaluation:** It evaluates the report generated by Daikon and ADFD+ about the boundaries known fault domains in the error-seeded programs. It is found that where Daikon was able to find the fault, it was not able to identify its domain boundary as accurately as ADFD+.
- **Future work:** It gives ideas of further application of ADFD+, such as finding and plotting faults in multi-dimensional programs using multi-dimensional graphs.

2. FAILURE PATTERNS

A number of empirical evidence confirms that fault revealing test cases tend to cluster in contiguous regions across the input domain [37, 17, 33]. According to Chan et al. [4] the clusters are arranged in the form of point, block and strip failure pattern. In the point pattern the failure revealing inputs are stand alone which are spread through out the input domain. In block pattern the failure revealing inputs are clustered in a one or more contiguous area. Finally, in strip pattern the failure revealing inputs are clustered in one long elongated area. Figure 1 shows the failure patterns in two-dimensional input domain.

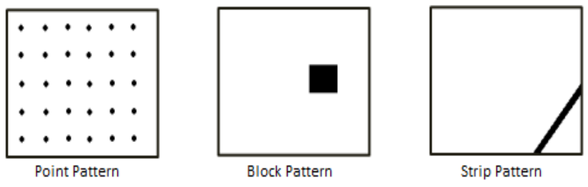


Figure 1: Failure patterns across input domain [4]

3. AUTOMATED DISCOVERY OF FAILURE DOMAIN+

ADFD+ is an improved and extended form of our previously developed Automated Discovery of Failure Domain [1]. The ADFD+ is an automated framework that finds the failures, their domains in a specified range and present them on a graphical chart. Following are the main differneces between ADFD and ADFD+.

- ADFD+ generate a single Java file dynamically at run time to plot the failure domains instead of one Java file per failure as in ADFD. This saves a lot of execution time and makes the process much quicker.
- ADFD+ uses (x, y) vector series to represent failure pattern as opposed to the (x, y) line series in ADFD. The vector series allows more flexibility and clarity to represent a failure and its domain domain.
- ADFD+ takes a single value as range which specify a round region around the failure whereas takes two values for lower and upper bound representing x and y axis respectively.
- In ADFD+, the algorithm of dynamically generated Java file, which is created after an error is discovered, is made more simplified and efficient.
- In ADFD+, the failure domain is focused in the graph which gives a clear view of pass and fail points. The points are also labelled for simplification as shown in the Figure 2.

3.1 Workflow of ADFD+

ADFD+ is an automatic process and all the user has to do is to specify the program to test and click the *DrawFaultDomain* button. The default value for range is set to 5 which means that ADFD+ will search 25 values around the failure. On clicking the button YETI is executed with ADFD+ strategy to search for a failure. On finding a failure the ADFD+ strategy create a java file which contain calls to the program on the failing value and related values up to the specified range. The Java file is compiled and executed. The result is analysed to check for pass and failed values. Pass values are stored in pass file and fail values are stored in fail file. At the end of the values range, the values are plotted on the graph with pass values as green and fail values as red as shown in the Figure 2.

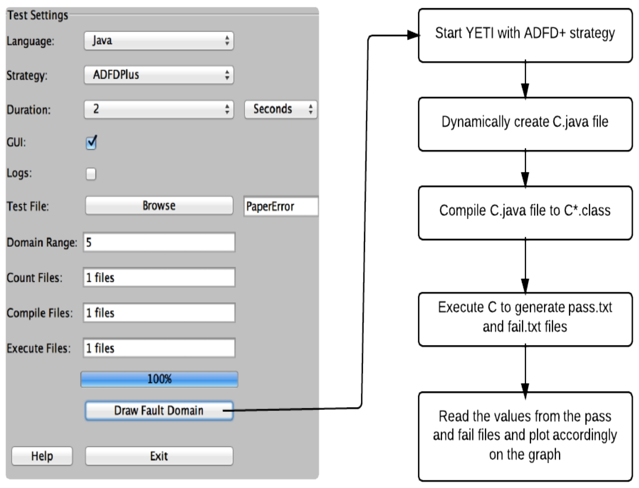

Test Settings
Language: Java
Strategy: ADFDPlus
Duration: 2 Seconds
GUI: ✓
Logs:
Test File: Browse PaperError
Domain Range: 5
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files
100%
Draw Fault Domain
Help
Exit
Start YETI with ADFD+ strategy
Dynamically create C.java file
Compile C.java file to C*.class
Execute C to generate pass.txt and fail.txt files
Read the values from the pass and fail files and plot accordingly on the graph

Figure 2: Workflow of ADFD+

3.2 Implementation of ADFD+

The ADFD+ strategy is implemented in a tool called York Extensible Testing Infrastructure (YETI). YETI is available in open-source at `http://code.google.com/p/yeti-test/`. In this section a brief overview of YETI is given with the focus on the parts relevant to the implementation of ADFD+ strategy. For verification of ADFD+ strategy in YETI, a program is used as an example to illustrate the working of ADFD+ strategy. Please refer to [24, 25, 27, 28] for more details on YETI.

YETI is a testing tool developed in Java that test programs using random strategies in an automated fashion. YETI meta-model is language-agnostic which enables it to test programs written in functional, procedural and object-oriented languages.

YETI consists of three main parts including core infrastructure for extendibility through specialization, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both the languages and strategies sections have a pluggable architecture to easily incorporate new strategies and languages making YETI a favorable choice to implement ADFD+ strategy. YETI is also capable of generating test cases to reproduce the faults found during the test session. The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. On top of the hierarchy in strategies, is an abstract class YetiStrategy, which is extended by YetiRandomPlusStrategy and is further, extended to get ADFD+ strategy.

3.3 ADFD+ by an example

In this section we describe the working of ADFD+ with a motivating example. Suppose we have the following class name *Error* under test. According to the code, the value of variable x between 5 to 8 and the value of variable y between 2 to 4 triggers a failure.

```
public class Error {

  public static void Error (int x, int y){

    int z;

    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
         {
             z = 50/0;
         }
   }
}
```

On execution, the ADFD+ strategy tests the class with the help of YETI and finds a failure at x = 6 and y = 3. Once a failure is discovered ADFD+ uses the surrounding values around the failure to find a failure domain. The range of surrounding values is limited to the value set by the user in the *DomainRange* variable. When the value of *DomainRange* is 5, ADFD+ evaluates total of 83 values of x and y around the found failure. All evaluated (x, y) values are plotted on a two dimensional graph with red dot indicating a failing value and green dot indicating the passing value as shown in the Figure 3.

4. EXPERIMENTAL SETUP

5. RESULTS

6. DISCUSSION

7. THREATS TO VALIDITY

8. CONCLUSION

9. FUTURE WORK

10. EVALUATION

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [21]. 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.

10.1 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?

10.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [36]. 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 1 while test details of the classes is presented in table **??**. Line of Code (LOC) tested per class and its total is shown in column 3 of table **??**.

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 [26] 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 with 6 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

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

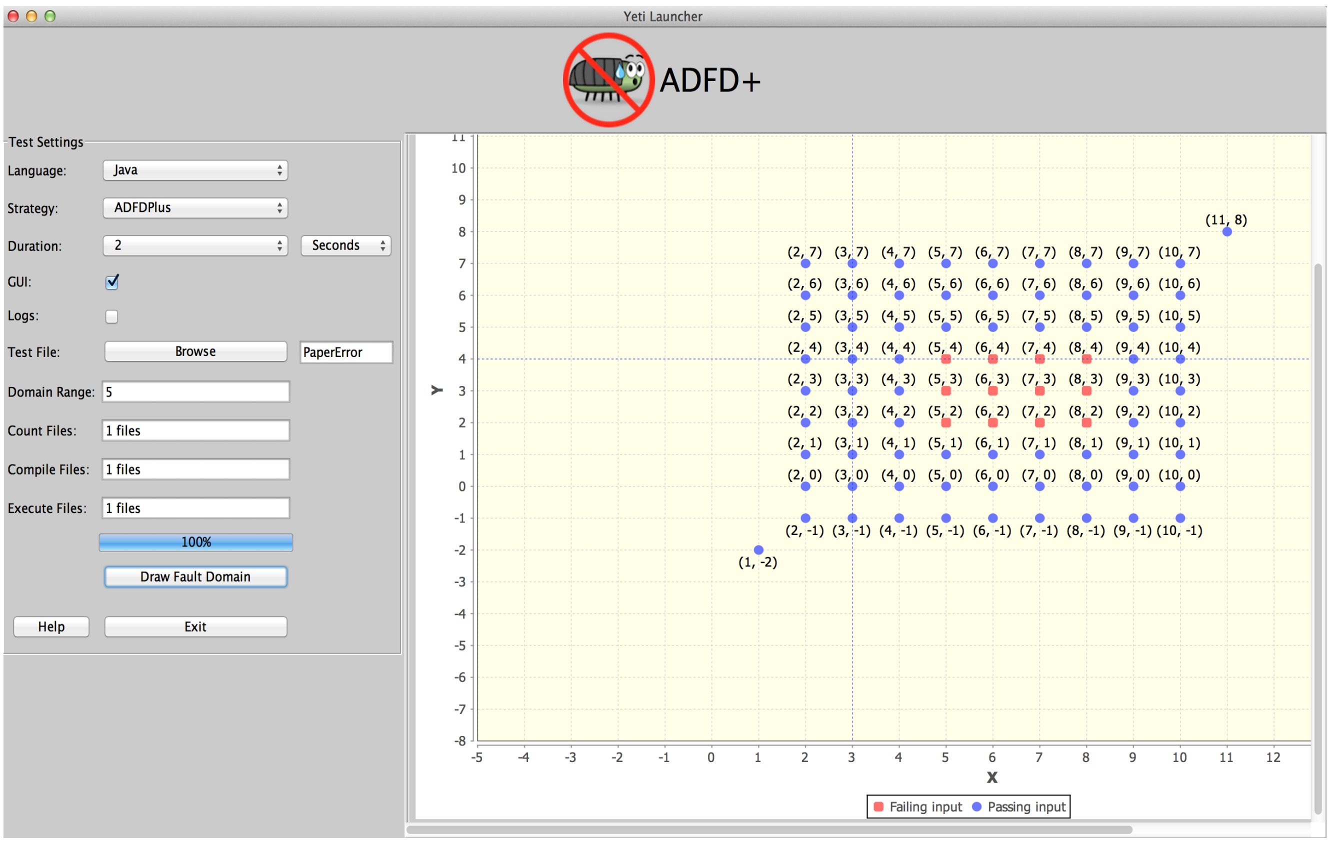
T-tests applied to the data given in Table 2 show that DSSR is significantly better in 7 classes from R and R+ strategy, in 8 classes DSSR performed similarly to R+ but significantly higher than R, and in 2 classes DSSR performed similarly to R but significantly higher than R+. There is no case R and 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.

10.4 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.

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


Yeti Launcher

ADFD+

Test Settings

Language: Java

Strategy: ADFDPlus

Duration: 2 Seconds

GUI: ☑

Logs: ☐

Test File: Browse PaperError

Domain Range: 5

Count Files: 1 files

Compile Files: 1 files

Execute Files: 1 files

100%

Figure 3: The output of ADFD+ for the above code.

11. 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 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 [29]. 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 stud-

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

ies, 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 [26].

12. RELATED WORK

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [31, 14, 12]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [20]. Some of the well known automated tools based on random strategy includes Jartege [22], Eclat [31], JCrasher [14], AutoTest [10, 11] and YETI [29, 26].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [8, 9, 5, 6, 7]. 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 [38] 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 [21]. These interesting values includes border values which have high tendency of finding faults in the given SUT [3]. Results obtained with R+ strategy show significant improvement over random strategy [21]. 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 [18, 15, 19]. Arcuri et al. [2], 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 [35] which is a collection of open source java programs maintained for independent empirical research. The

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

projects in Qualitas Corpus are carefully selected that spans across the whole set of java applications [26, 36, 34].

13. 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.

14. REFERENCES

[1] M. A. Ahmad and M. Oriol. Automated discovery of failure domain. *Lecture Notes on Software Engineering*, 03(1):289–294, 2013.

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

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

[4] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

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

[6] T. Chen, R. Merkel, P. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 79 – 86, sept. 2004.

[7] 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, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

[8] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83:60–66, January 2010.

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

[10] 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, pages 84–94, New York, NY, USA, 2007. ACM.

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

[12] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, Sept. 2000.

[13] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[14] C. Csallner and Y. Smaragdakis. Jcrasher: An automatic robustness tester for Java. *Software—Practice & Experience*, 34(11):1025–1050, Sept. 2004.

[15] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438 –444, july 1984.

[16] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[17] G. B. Finelli. Nasa software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[18] W. Gutjahr. Partition testing vs. random testing: the influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661 –674, sep/oct 1999.

[19] D. Hamlet and R. Taylor. Partition testing does not inspire confidence [program testing]. *Software Engineering, IEEE Transactions on*, 16(12):1402 –1411, dec 1990.

[20] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[21] 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, pages 261a–, Washington, DC, USA, 2007. IEEE Computer Society.

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

[23] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality*, pages 242–256. Springer, 2005.

[24] M. Oriol. York extensible testing infrastructure, 2011.

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

[26] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201 –210, april 2012.

[27] M. Oriol and S. Tassis. Testing. net code with yeti. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 264–265. IEEE, 2010.

[28] M. Oriol and F. Ullah. Yeti on the cloud. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 434–437. IEEE, 2010.

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

[30] C. Pacheco and M. D. Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

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

[32] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[33] C. Schneckenburger and J. Mayer. Towards the determination of typical failure patterns. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90–93. ACM, 2007.

[34] E. Tempero. An empirical study of unused design decisions in open source java software. In *Software Engineering Conference, 2008. APSEC '08. 15th Asia-Pacific*, pages 33 –40, dec. 2008.

[35] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[36] E. Tempero, S. Counsell, and J. Noble. An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, pages 3–12, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[37] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.

[38] S. Yoo and M. Harman. Test data regeneration: generating new test data from existing test data. *Softw. Test. Verif. Reliab.*, 22(3):171–201, May 2012.