Automated Discovery of Failure Domain

Mian Asbat Ahmad
Department of Computer Science
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

Many research studies in the random testing literature refer to point, block and strip fault domains across the input domain of a system. A number of new strategies have also been devised on this principle claiming better results. However, no study was conducted to graphically show their existence and the frequency of each faulty domain in real production application.

In this research we study fault domains and check to which type of domains they belong. Our experimental results show that in 60% cases faults form point domain, while block and strip domain form 20% each. We also checked what relation exists between fault domains traced back to only one fault: are they contiguous, separate, or marginally adherent.

This study allows for a better understanding of fault domains and assumptions made on the strategies for testing code. We applied our results by correlating our study with three random strategies: random, random+ and DSSR.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools, Failure Domains, Random testing, Automated Testing*

1. INTRODUCTION

Testing is fundamental requirement to assess the quality of any software. Manual testing is labour-intensive and error-prone; therefore emphasis is to use automated testing that significantly reduces the cost of software development process and its maintenance [1]. Most modern black-box testing techniques execute the System Under Test (SUT) with specific input and compare the obtained results against the test oracle. A report is generated at the end of each test session containing any discovered faults and the input values which triggers the faults. Debuggers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality, set in test plan, is achieved.

The fact that exhaustive testing for any non-trivial program is impossible, compels the testers to come up with some strategy of input selection from the whole input domain. Pure random is one of the possible strategies that are widely used in automated tools. It is intuitively simple, easy to implement, minimum or no overhead in input selection and lack of bias [8, 9, 11, 12]. While pure random testing has many benefits there are also some limitations that include low code coverage [14] and discovery of lower number of faults [5]. To overcome these limitations many researchers successfully refined pure random testing while keeping its benefits intact. Most significant refinement of random testing is Adaptive Random Testing (ART) [6]. The experiments performed using ART showed up to 50% better results as compared to the traditional/pure random testing which has no criteria for input selection. Similarly Restricted Random Testing (RRT), Mirror Adaptive Random Testing (MART), Adaptive Random Testing for Object Oriented Programs (ARTOO), Directed Adaptive Random Testing (DART), Lattice-based Adaptive Random Testing (LART) and Feedback-directed Random Testing (FRT) [3, 4, 8, 10, 13, 17] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

All the above mention variations are based on the observation that failure causing inputs across the whole input form certain kinds of domains [2]. They divided them into point, block and strip fault domain. The square box represents the whole input domain while the black point, block and strip represent the faulty values inside the input domain as shown in Figure 2. They further suggested that the effectiveness of testing could be improved by taking into account the possible characteristics of failure causing inputs.

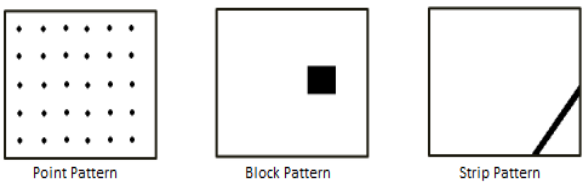

Figure 1: Failure domains across input domain [2]

This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD), implemented in York Extensible Testing Infrastructure (YETI). In ADFD strategy testing of SUT starts using Random+ (R+) strategy and when a fault is found in the SUT the ADFD strategy finds not only the values that trigger the fault but also its domain. Identification of fault domain is helpful in at least two ways. First is the Identification of the fault domain instead of a single instance of fault decreases testing times by reducing the back and forth of the project between testers and debuggers. Secondly it helps debugger team as the debuggers will keep in view all the fault occurrences while rectifying the fault and not just a single instance of that particular fault. For simplification purpose ADFD strategy uses GUI front end given in figure **??** and the output in the form of fault domains are presented graphically using (x, y) chart along with the pass and fail values.

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 results. Section 5 discusses the results. Section 6 presents related work and Section 7, concludes the study.

2. AUTOMATED DISCOVERY OF FAILURE DOMAIN

Automated Discovery of Failure Domain (ADFD) is a new test strategy where testing of SUT starts using Random+ (R+) strategy and when a fault is discovered the strategy shift its focus on the found fault to identify its domain. The output produced at the end of test session is an (x,y) chart showing the passing value or range of values in green line and failing value or range of values in red line.

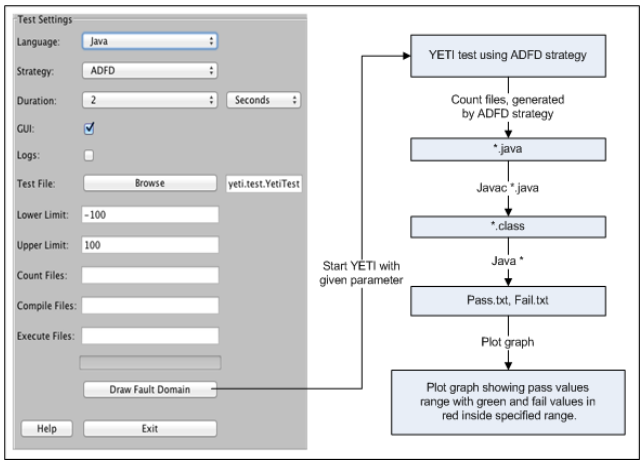

Figure 2: Working Flow of ADFD strategy

The process is divided into the following four major parts for simplification. Each part is explained below.

1. Providing Input From GUI Front-end
2. Automated Fault Finding
3. Automated generation of modules
4. Automated compilation and execution of modules
5. Automated Generation of Graph

Providing Input From GUI Front-end:
ADFD strategy is provided with an easy to use GUI front-end to get input from the user. It takes YETI specific input including language of the program, strategy, duration, enable or disable YETI GUI, logs and a program to test in the form of java byte code. In addition it also takes minimum and maximum values for restricting ADFD strategy to search for fault domain in the specified range. Default range for minimum and maximum range is Integer.MIN_INT and Integer.MAX_INT respectively.

Automated Fault Finding:
To find the failure domain for a specific fault first we need to identify that fault in the system. ADFD strategy uses random+ strategy — random strategy with preference to the boundary values for better performance to find the fault. ADFD strategy is implemented in York Extensible Testing Infrastructure (YETI). The ADFD strategy is implemented in automated testing tool YETI for its simplicity, speed and proven capability of finding potentially hazardous faults in many systems **??**. YETI is quick and can call up to one million instructions in one second on Java code. It is also capable of testing VB.Net, C, JML and CoFoJa beside Java.

Automated generation of modules:
After a fault is found in the SUT, ADFD strategy generate complete new Java program to search for fault domain in the given SUT. These program with .java extension are generated through dynamic compiler API included with Java 6 under javax.tools package. The number of programs generated can be one or many depending on the number of arguments in the test module i.e. for module with one argument one program is generated, for two argument two programs and so on. To track fault domain we kept one or more argument constant and one argument variable in the generated program.

Automated compilation and execution of modules:
The java programs generated in previous step are compiled using javac command to get their binary (.class) files. After that the (java *) command is executed to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument receive all the values specified at the start from minimum to maximum. After execution is complete we get two text files of pass and fail. Pass file contain all the values for which the module behave correctly while Fail file contain the values for which the modules failed.

Automated Generation of Graph:
The values from the pass and fail files are plotted on an (x y) graph using a free open source JFreeChart. For one argument program the y component is kept constant. The pass values are represented with green lines while the fault values are represented using red line on the chart. Resultant graph clearly depicts the domain of the fault. The graph shows red points in case the program fails for only one value, blocks

for failing multiple values and strip for failing long range of values.

current_faults: INTEGER old_faults: INTEGER ... current_faults := track_number_of_errors old_faults := 0 if (current_faults > old_faults) then old_faults := current_faults Generate a program dynamically with the following algorithm. ... continue testing

Table 1: Algorithm 1 and 2

3. IMPLEMENTATION

We implemented ADFD strategy in a tool called York Extensible Testing Infrastructure (YETI) [16]. YETI is available in open-source at http://code.google.com/p/yeti-test/. In this section we give a brief overview of YETI, integration of ADFD strategy in YETI and a simple example to illustrate the working of ADFD strategy.

3.1 York Extensible Testing Infrastructure

It is a testing tool developed in Java that test programs in an automated fashion using random strategies. YETI meta model is language-agnostic which enables it to test programs written in multiple languages that include Java, C#, JML, .Net and Pharo. YETI consist of three main parts that include the core infrastructure responsible for extendibility through specialisation, the strategies section to adjust multiple strategies and language-specific bindings to provide support for multiple languages [15].

3.2 ADFD strategy in YETI

Strategies package in YETI contain all the strategies including random, random+ and DSSR that can be selected for testing according to the specific needs. The default test strategy for testing is simple random. On top of the hierarchy is an abstract class YetiStrategy which is extended by YetiRandomStrategy and it is further extended to get ADFD strategy as shown in figure 3.

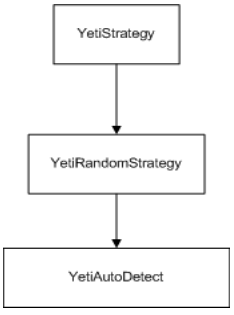

Figure 3: Class Hierarchy of automated discovery of failure domains in YETI

3.3 Example

The goal of ADFD is to find the fault in the SUT and its existence across the complete domain in an automated way. This helps the developers to debug the code keeping in view its every occurrence that may otherwise go unnoticed. Published programs from literature [7][2][4] of point, block and strip failure patterns are tested to explain the working of ADFD . These programs were translated in to java language for this experiment (See appendix 1 for more details).

```
/**
* Point Fault Domain example
* for one argument
* @author (Mian and Manuel)
*/
public class PointDomainOneArgument{

    public static void pointErrors (int x){
            if (x == -66 )
                abort();

            if (x == -2 )
                abort();

            if (x == 51 )
                abort();

            if (x == 23 )
                abort();
    }
}
```

ADFD can be activated by typing the command java -jar ADFD.jar. After the GUI of ADFD is launched we need to specify yeti specific values that include language of the program under test, strategy for the current test session, duration of test session (minutes or milli-second), display YETI GUI or not and display real time logs or not. Next we browse to select the file for testing and the run button starts testing the file with YETI tool.

In 5 second YETI found one fault out of the above 4 faults. The ADFD strategy in YETI generate a source file (C*.java) at the end of the test session. This file contain the code that searches for fault domains. The count button count the number of files. ADFD create the number of files on the basis of the number of arguments in the method under test. For one argument one method is created and for two argument two methods are created.

The next button is compile which compile the generated files and generate the byte code (.class files). The execute button execute the byte code and test the method under test for all the values between upper and lower bound. At the end of execution it generates two files (pass.txt and fail.txt). Pass file contain all the values for which the method performed correctly while fail file contain all the values for which the method under test fail.

The draw fault domain button reads the pass and fail files and plot them on the x, y graph where red line with squares show the failing values while the blue line with square shapes

show the passing values.

From the figure we can see that the use of ADFD not only found all the faults but from the graph we can also know that the program follows a point domain of failure.

4. EXPERIMENTAL RESULTS

4.1 Experimental setup

4.2 Results

5. DISCUSSION

6. RELATED WORK

7. CONCLUSION

One conclusion is that ARDT helps in exploring new faults or you can say new failure test cases because if you see figure 3 (a, b, c) it gives 3 range of values for which the program fails.

Doing this also saves time in debugging because in ordinary testing the testing stops as soon as the fault is discovered and once the fault is removed by the developers the testing starts again. But here the develop debug the program for all the range instead of single fault value thus saving multiple steps.

Debugging can also be made more efficient because the debugger will have the list of all the values for which the program fail therefore he will be in a more better position to rectify the faults and test them against those special values before doing any further testing.

We also found that the block and strip pattern are most common in arithmatic programs where as point pattern are more frequently found in general programs.

This study will also let us know the reality of failure patterns and its existence across the programs.

8. ACKNOWLEDGMENTS

This section is optional; it is a location for you to acknowledge grants, funding, editing assistance and what have you. In the present case, for example, the authors would like to thank Gerald Murray of ACM for his help in codifying this *Author's Guide* and the **.cls** and **.tex** files that it describes.

9. REFERENCES

[1] B. Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley, 1995.

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

[3] K. Chan, T. Chen, and D. Towey. Restricted random testing. *Software QualityŮECSQ 2002*, pages 321–330, 2006.

[4] 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. IEEE, 2004.

[5] T. Chen and Y. Yu. On the relationship between partition and random testing. *Software Engineering, IEEE Transactions on*, 20(12):977 –980, dec 1994.

[6] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008.

[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] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo. In *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, pages 71 –80, may 2008.

[9] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4*, WSS'00, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

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

[12] R. C. Linger. Cleanroom software engineering for zero-defect software. In *Proceedings of the 15th international conference on Software Engineering*, ICSE '93, pages 2–13, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[13] J. Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 333–336. ACM, 2005.

[14] A. J. Offutt and J. H. Hayes. A semantic model of program faults. *SIGSOFT Softw. Eng. Notes*, 21(3):195–200, May 1996.

[15] 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, pages 264–265, Washington, DC, USA, 2010. IEEE Computer Society.

[16] M. Oriol and F. Ullah. Yeti on the cloud. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, pages 434–437, Washington, DC, USA, 2010. IEEE Computer Society.

[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, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

APPENDIX

S. No	Failure Pattern	Specific Fault	Pass Pattern	Fail Pattern
1	Point	Prog1.test1(i)	00 to 00	0
			00 to 00	
2	Block	Prog2.test2(i)	00 to 00	0
			00 to 00	
3	Strip	Prog3.test3(i)	00 to 00	0
			00 to 00	

Table 2: Failure domain with respect to one dimensional program

S. No	Failure Pattern	Specific Fault	Pass Pattern	Fail Pattern
1	Point	Prog1.test1(-4,i)	-2147483648 to -1	0
			1 to 2147483647	
		Prog1.test1(i,0)	None	-2147483648 to 2147483647
2	Block	Prog2.test2(i,10)	-2147483648 to 9	10, 11
			12 to 2147483647	
		Prog2.test2(10,i)	-2147483648 to 9	10, 11, 12
			13 to 2147483647	
3	Strip	Prog3.test3(i,4)	-2147483648 to -2147483641	-2147483640 to -214783637
			-2147483636 to 7	8 to 11
			12 to 2147483647	
		Prog3.test3(10,i)	-217483648 to 1	2 to 9
			10 to 2147483647	

Table 3: Failure domain with respect to two dimensional program