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 a labour-intensive job, therefore emphasis is on the use of automated testing that significantly reduce 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 values which causes the fault. These reports are later evaluated by debuggers to fix the discovered faults. The revised version of the system is given back to the testers to find more faults and this process is continue till a certain level of satisfaction is achieved.

Chan et al. [2] found domains of failure causing inputs across the whole input domain. They divided them into block, strip and point domain as shown in Figure 4. They further suggested that the effectiveness of proportional sampling strategy can be improved by taking into account the possible characteristics of failure causing inputs.

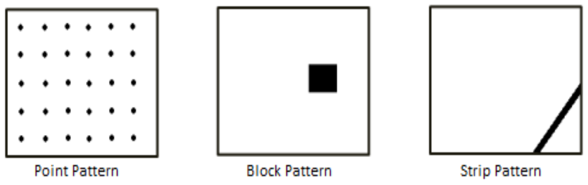


Figure 1: Failure domains across input domain [2]

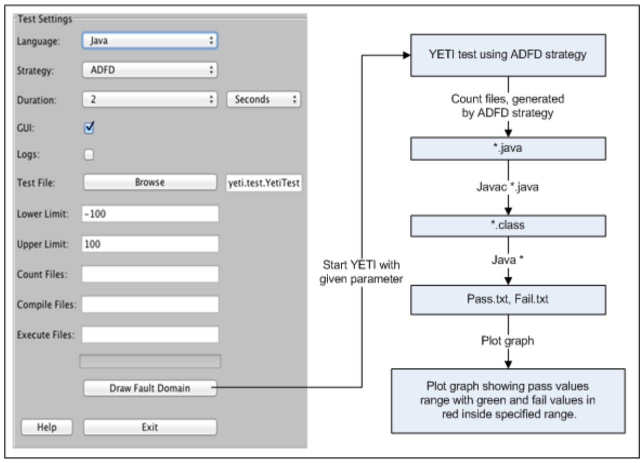
In search of better testing results Chen et al., implemented the same idea in Adaptive Random Testing (ART) [5]. In ART test data is generated by selecting test values farthest away from one another to increase the chance of hitting these faulty domains. The experiments performed using ART showed up to 50% better results as compared to the traditional/pure random testing which has no criteria for input selection. Mirror Adaptive Random Testing (MART) [6], Feedback-directed Random Testing (FDRT) [10], Restricted Random Testing (RRT) [3] and Quasi Random Testing (QRT) [7] are the strategies based on the same principle that found better results compared to ordinary random testing.

This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD), implemented in York Extensible Testing Infrastructure (YETI). In this strategy testing of SUT starts using Random+ (R+) strategy and after a fault is found the testing doesn't stop but remain continue until the discovery of complete fault domain. Identification of fault domain is helpful in two ways. Identifying all the fault domains and not a single instance of fault reduces testing times by reducing the back and forth of the project between testers and debuggers. Secondly it helps debugger as the debuggers will keep in view all the the fault occurrences while rectifying the fault and not just a single instance of fault. For simplification purpose ADFD strategy use 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 fault values. Additionally, ADFD test strategy can also be used to identify frequency of point, block and strip fault domain across the production softwares.

The rest of this paper is organised as follows:
Section 2 describes the ADFD strategy. Section **??** presents implementation of the ADFD strategy. Section **??** explains the experimental setup. Section **??** shows results of the experiments. Section **??** discusses the results. Section **??** presents related work and Section **??**, 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.


Test Settings
Language: Java
Strategy: ADFD
Duration: 2 Seconds
GUI: ✓
Logs: ☐
Test File: yeti.test.YetiTest
Lower Limit: -100
Upper Limit: 100
Count Files:
Compile Files:
Execute Files:

Browse
Draw Fault Domain
Help
Exit
Start YETI with given parameter
YETI test using ADFD strategy
Count files, generated by ADFD strategy
*.java
Javac *.java
*.class
Java *
Pass.txt, Fail.txt
Plot graph
Plot graph showing pass values range with green and fail values in red inside specified range.

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.

3. STRUCTURE OF AUTOMATED DISCOVERY OF FAILURE DOMAIN

4. MOTIVATING 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 [6][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.

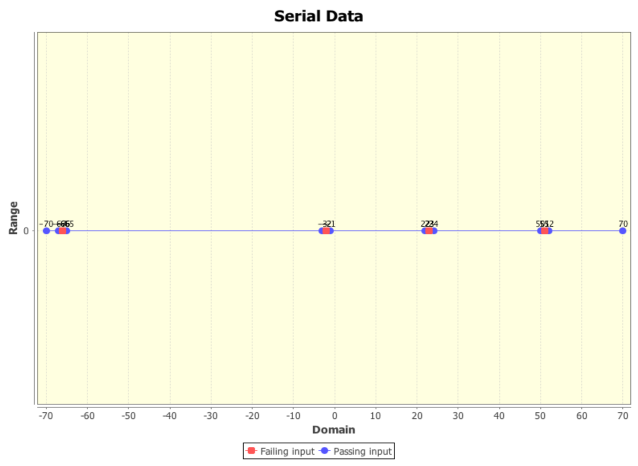


Figure 3: Point pattern failure domain

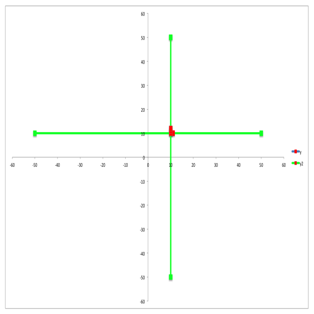


Figure 4: Block pattern failure domain

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 1: 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 2: Failure domain with respect to two dimensional program

5. IMPLEMENTATION

The technique of automated discovery of failure domain is implemented in a tool called York Extensible Testing Infrastructure (YETI) [9]. 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 to adjust multiple strategies and language-specific bindings to provide support for multiple languages [8]. The default test strategy for testing is simple random.

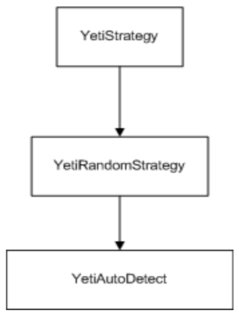


Figure 5: Class Hierarchy of automated discovery of failure domains in YETI

Strategies package contain all the strategies that can be selected for testing according to the specific needs. On top of the hierarchy is an abstract class YetiStrategy which is extended by YetiRandomStrategy and it is further extended to get YetiAutoDectect (ADFD) strategy as shown in figure 5.

6. 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.

7. 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.

8. 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. 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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