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 process of software testing and test data generation is a labour-intensive, therefore emphasis is on the use of automated testing that significantly reduce the cost of software development process and its maintenance [1]. Most of the 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 system is given back to the testers to find more faults and this process is continue till a certain level of satisfaction is achieved. It is important to note that these techniques only identify a single instance of failure and do not focus on the failure domain.

This paper describes Automated Discovery of Failure Domain (ADFD), a framework based on auotmated random testing tool, York Extensible Testing Infrastructure (YETI) **??** for not only finding fault in java program but also its domain. To find the fault and its domain ADFD needs java method (byte code, .class file), lower and upper bound to limit the search for fault domain; for integer the default lower and upper bound is set to Integer.MIN_VALUE and Integer.MAX_VALUE. After a fault is found it search for the fault domain between the upper and lower bound. ADFD apply every value between the upper and lower bound to the method for finding a point, block or strip fault domain. The result obtained is produced in the form of graph at the end of the test session.

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.

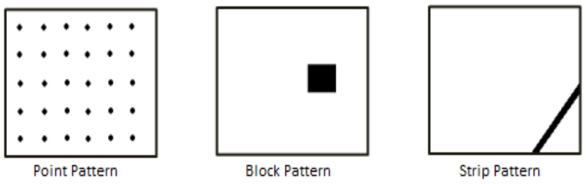


Figure 1: Failure domains across input domain [2]

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.

The main objective behind ADFD is to get an automated frame work that find the existence of fault and fault domain across the input domain, decrease debugging time and to discover any more faults missed by the testing system. Significant research has been done to utilise the failure domains but their existence, nature and boundaries need further attention. Having fault domain information prior to testing enables the tester to guide testing according to the failure domain of the SUT, for example pure random testing is more effective for point domain than block and strip domains where as ART, MART, FDRT, RRT and QRT are more effective for block and strip fault domains than point fault domain.

In Section II, we describe the automated technique of discovering failure domains and explain its structure and function with the help of a flowchart and motivating example. Section III presents its implementation in automated random testing tool called York Extensible Testing Infrastructure (YETI). Section IV and V report the experiments performed using the proposed technique and evaluate & discuss the obtained results. Section VI and VII discuss any threats to validity and related discussion. Finally we conclude in Section VIII.

2. AUTOMATED DISCOVERY OF FAILURE DOMAIN

Automated Discovery of Failure Domain (ADFD) strategy is a new technique that find a failure domain across the whole input domain in an automated fashion. The discovered failure domains are presented in the form of graphs at the end of the session. The process is divided into the following four major parts for simplification. Each part is explained below.

1. Use of automated testing tool to find the failure in the given SUT.
2. Setting of Lower and Upper bound
3. Automated generation of modules at run time according to the found failure.
4. Automated compilation and execution of the generated modules.
5. Analysis of results obtained and graphically plot the fault domain .

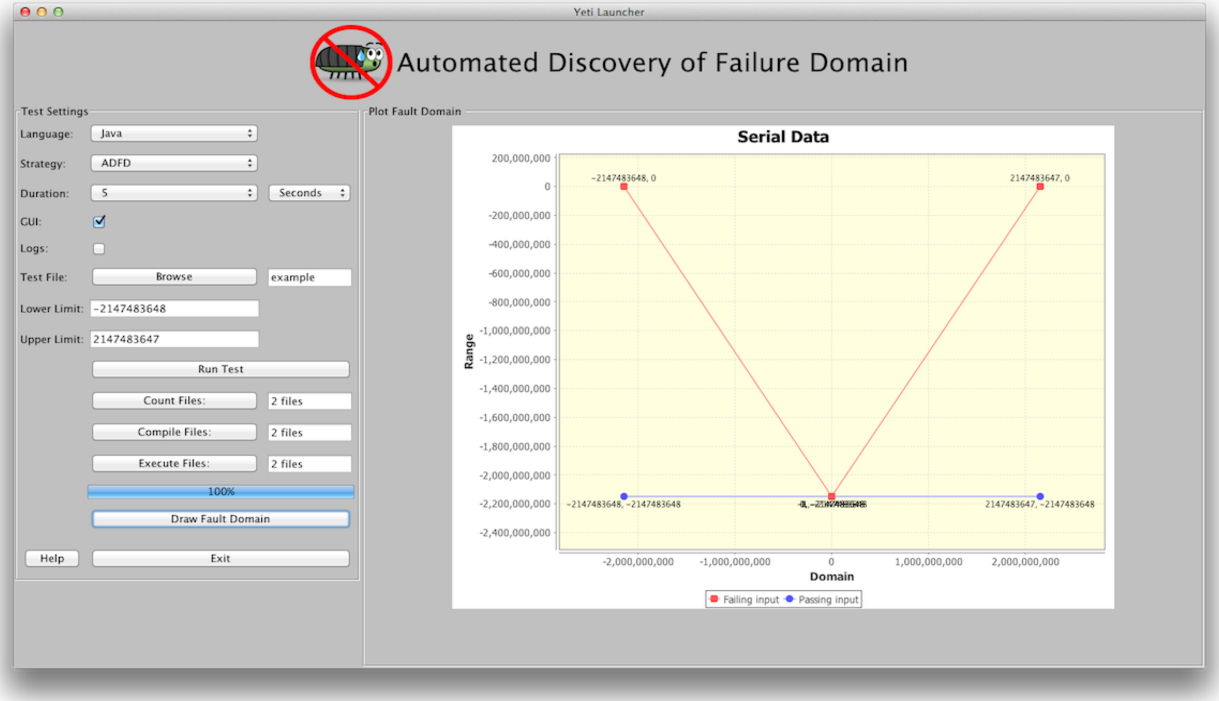
Automated testing to find the fault in given SUT:
To find the failure domain for a specific fault first we need to identify the fault in the system. This required fault can be identified by manual testing but we selected automated testing system because it can saves time, resource and produce quick results. For this purpose we could select any automated testing system because the research is focusing on the failure domains and not on the performance of finding faults and for that reason F-measure, E-measure and P-measure etc were not of particular concern. York Extensible Testing Infrastructure (YETI) is selected to test and find the fault in given SUT mainly for its simplicity, speed and proven capability of finding potentially hazardous faults in many systems. It is a complete automated testing system that is capable of calling millions of instructions in one second on Java code. It is also capable of testing VB.Net, C, JML and CoFoJa beside Java. YETI was executed with its default strategy of random testing for finding the fault in our experiments.

Setting Lower and Upper bound:
One can specify the lower and upper bound to limit the search. ADFD will supply only the values enclosed in the upper and lower bound. By default this range is set to Integer.MAX_VALUE and Integer.MIN_VALUE.

Automated generation of modules at run time according to the fault found:
After finding the fault in the given SUT through an automated system we needed to write one, two or many modules based on the condition of the found fault. These modules are later executed to explore the nature of the failure domain for the found fault. To keep this process automated we used Java feature of generating dynamic code which is available in javax.tools package in Java language. We added additional functionality in YETI and now when the fault is found by YETI in the given SUT it stops testing and dynamically generate the required modules with .java extension and saves it to a file for further execution. This generated module can have one or more argument constant and one argument as variable.

Automated compilation and execution of these modules:
The testing process stops after the generated modules are written to a file with .java extension on some permanent media. A script is executed and the .java files produced earlier are fist compiled to get the binary .class file and then executed. During execution the static arguments of the module remain the same but the variable argument receive all the values of the whole input domain of that particular type. Once the execution of the generated modules is completed the results are produced by the system on the standard output.

Generation of results and analysis:
All the generated modules are executed by the system and the results obtained are saved and analysed. On the basis of analysis of these results we can identify that which particular domain that fault belongs to. If the module output a single value after few thousands of values then the fault belongs to the point domain. If it fails for a pool of value at some specific interval then it belongs to the block domain and if it fails to a large pool of continuos values than it belongs to a strip domain.



Language: Java
Strategy: ADFD
Duration: 5 Seconds
GUI: ☑
Logs: □
Test File: Browse example
Lower Limit: -2147483648
Upper Limit: 2147483647
Run Test
Count Files: 2 files
Compile Files: 2 files
Execute Files: 2 files
100%
Draw Fault Domain
Help
Exit

Figure 2: GUI of ADFD strategy

3. 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.

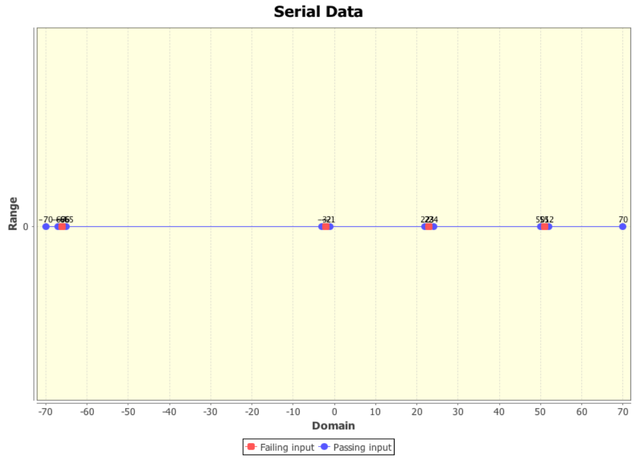


Figure 3: Point pattern failure domain

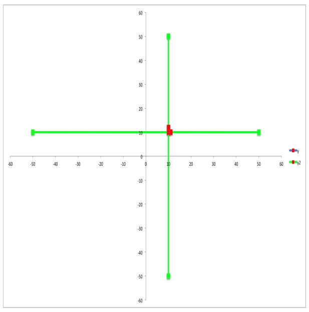


Figure 4: Block pattern failure domain

4. 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.

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.

5. 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.

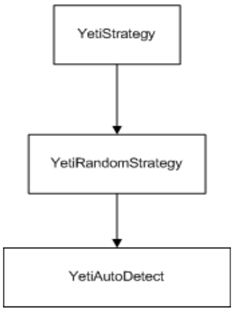


Figure 5: Class Hierarchy of automated discovery of failure domains in YETI

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.

6. 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.

7. 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

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

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