Automated Discovery and Analysis of Failure Domain with respect to Random Testing

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. All dynamic 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 which contain any discovered faults with the input values, that lead to the generation of the fault. These reports are later evaluated by debuggers to fix the discovered faults. It is important to note that most if not all the existing testing techniques identify the faults but ignores the failure domain containing the discovered fault.

Chan et al. [1] 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.

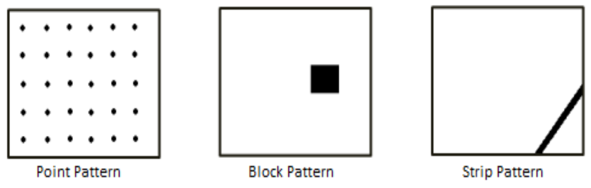

Figure 1: Failure domains across input domain [1]

In search of better testing results Chen et al., implemented the same idea in Adaptive Random Testing (ART) [4]. 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) [5], Feedback-Directed Random Testing (FDRT) [9], Restricted Random Testing (RRT) [2] and Quasi Random Testing (QRT) [6] are the strategies based on the same principle that found better results compared to ordinary random testing.

Significant research has been done to utilise the failure domains but their existence, nature and boundaries need further attention. In this article we present an automated random technique to discover and investigate the failure domains across the input domain. Having this 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. PROBLEMS AND SOLUTIONS

This paper address five. main problems in random testing. These are (1) Finding the whole domain of fault instead of only failing values, (2) Representation of fault values, (3) automation of the evaluation process, (4) Identification of fault domain for multi arguments methods and its representation, (5) Generation and classification of test values for Strings and complex (non scaler) arguments. This section elaborate each of the above mention problem and describe our proposed solution (if any) to them.

2.1 Finding the whole domain of fault instead of only failing values

Most of the testing tools take into account only the fault finding values with out giving due consideration to the domain in which the values exist.

2.2 Representation of fault values and fault domains

points: instead of dumping logs and more complex reports we describe the fault domains with the help of charts.

2.3 Automation of the testing process

We developed an automated system to test the system, generate the fault domain finding files, compile and execute them to find the fault domains if any. points: automation is achieved by combining the test tool and evaluation system e.g. yeti and ADFD.

2.4 Identification and representation of multi arguments data

I think it is beyond the scope of this study to identify and represent more then 3 arguments method because at the moment we can show only three diminutional charts.

2.5 Generation and classification of test values for string and complex (non scaler) arguments

It is difficult to find the domain for strings and complex (non scaler) data therefore they can be exempted from this study.

3. AUTOMATED DISCOVERY OF FAILURE DOMAIN

Automated Discovery of Failure Domain (ADFD) is the process in which a failure domain is identified in an automated fashion across the whole input domain for a specific fault in a given system. The whole process is made automated until the results are generated. 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 fault in the given SUT.
2. Automated generation of modules at run time according to the found fault.
3. Automated compilation and execution of the generated modules.
4. 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. We could select any automated testing system for finding the fault in the given SUT 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.

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.

3.1 Flow Chart of the process

The following flow chart clearly identify the workflow of the whole process and the various steps involved in the process.

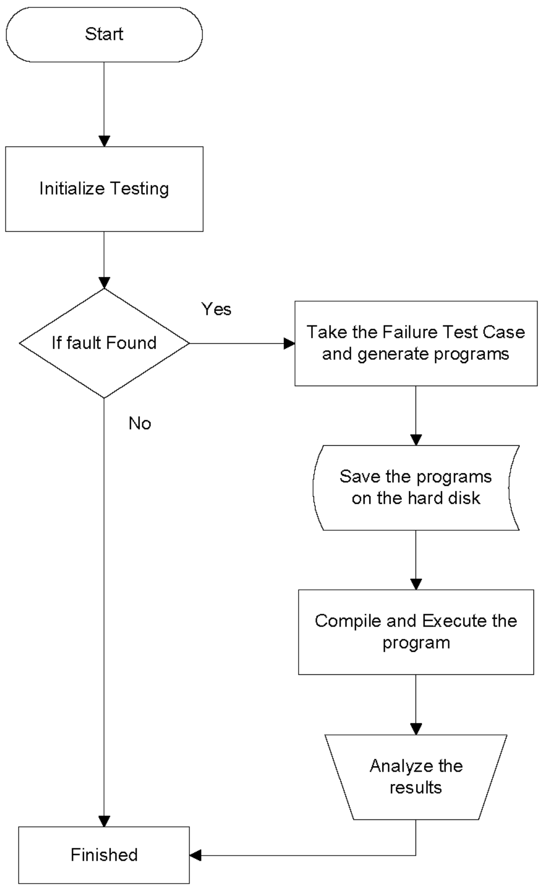
Start
Initialize Testing
If fault Found
Yes
No
Take the Failure Test Case and generate programs
Save the programs on the hard disk
Compile and Execute the program
Analyze the results
Finished

Figure 2: 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 [5][1][3] 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).

In step one each program was tested individually by YETI that discovered the existing faults and generated relevant program for each fault. The number of generated programs depend on the dimension of the program under test. Since all selected programs were two dimensional (containing two parameters) therefore two programs were generated for each found fault. Table 1 shows the programs generated and pass & fail patterns discovered for each fault. For each test entire integer range i.e. -2147483648 to 2147483647 was selected as input domain.

In step 2 the generated programs were executed in such a way that the variable parameter receives all the values between -2147483648 to 2147483647 while static parameter remain constant. Results of the execution are demonstrated with the help of the following graphs. The graphs only scales to 100 values and not for complete integer range only to keep the fault patterns visible and more readable.

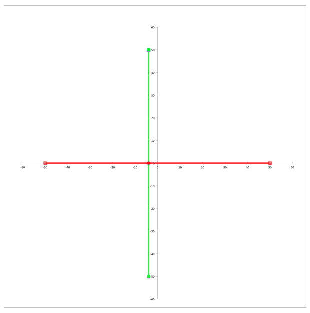

Figure 3: Point pattern failure domain

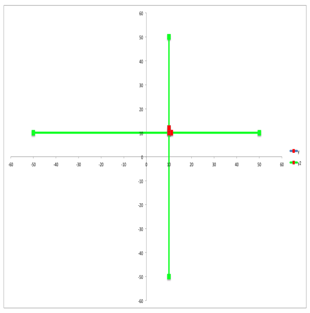

Figure 4: Block pattern failure domain

5. IMPLEMENTATION

The technique of automated discovery of failure domain is implemented in a tool called York Extensible Testing Infrastructure (YETI) [8]. 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 [7]. 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

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 1: Failure pattern with respect to failure domain

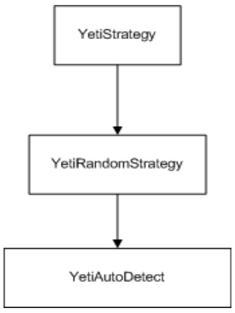

Figure 5: Class Hierarchy of automated discovery of failure domains in YETI

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] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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