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.

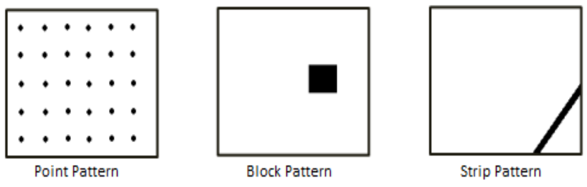


Figure 1: Failure domains across input domain [2]

All the above mention variations in random testing 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. In Figure 1 the square box represents the whole input domain while the black point, block and strip inside the box represent the faulty values across the input domain. They further suggested that the effectiveness of testing could be improved by taking into account the possible characteristics of failure causing inputs.

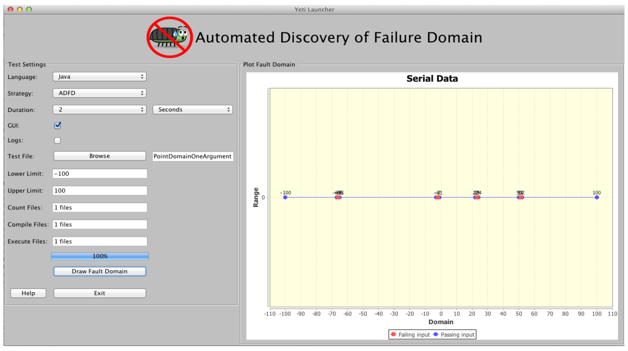
It is interesting that where many random strategies are based on the principal of contiguous fault domains inside the input domain no specific strategy is developed to evaluate these fault domains. This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD). The ADFD strategy not only finds the fault but also finds its domain. The idea of identification of fault domain is attractive as it provides an insight of the fault domains in the SUT. Additionally, the paper makes the following contributions:

- **Implementation:** It presents the implementation of the new ADFD strategy in York Extensible Testing Infrastructure (YETI) tool.
- **Evaluation:** Several classes are tested with ADFD strategy to evaluate its behaviour.
- **Decrease in Test Duration:** 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
- **Increase in Test Efficiency:** ADFD strategy helps debugger team as the debuggers will keep in view all the fault occurrences while

The rest of this paper is organized 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 to find not only the fault but its complete domain across the whole input 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. The work flow of ADFD strategy is given in Figure 3.

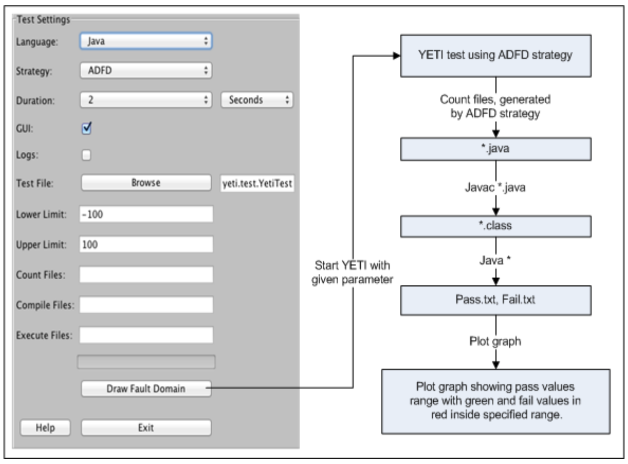


Language: Java
Strategy: ADFD
Duration: 2 Seconds
GUI: ☑
Logs: ☐
Test File: PointDomainOneArgument
Lower Limit: -100
Upper Limit: 100
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files
100%

Figure 2: Front-end 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



Test Settings
Browse
Draw Fault Domain
Help
Exit
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.
Start YETI with given parameter

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:

Figure 3: Working Flow of ADFD strategy

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. 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 4.

3.3 Example

For a concrete example to show how ADFD strategy in YETI proceeds. We suppose the following class is tested by YETI with ADFD strategy selected for evaluation. Note that for more clear visibility of the output graph generated

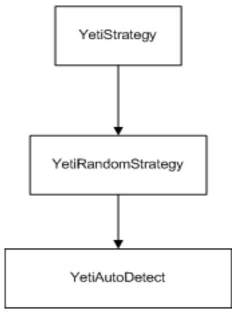


Figure 4: Class Hierarchy of automated discovery of failure domains in YETI

by ADFD strategy at the end of test session we decrease the values of lower and upper range to -70 and 70 from Integer.Min_Int and Integer.Max_Int respectively.

```
/**
 * 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();
    }
}
```

As soon as any one of the above four faults are discovered the ADFD strategy generate a dynamic program given in Appendix. This program is automatically compiled to get binary and then executed to find the pass and fail domain inside the specified range as shown in the Figure 5.

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

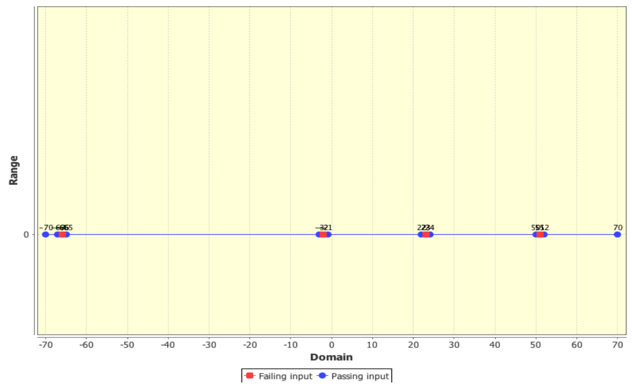


Figure 5: ADFD strategy plotting pass and fault domain of the given class

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

In this section we present the experimental setup and results of the several experiments performed using ADFD strategy. We selected 10 numerical programs of one and two dimension. These program are error seeded in such a way that they form all the three forms of fault domains that include point, block and strip fault domain. Each selected program contain various combinations of same or different fault domain. Code of the programs is given in Appendix.

All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35].

For clarification purpose we have taken a separate example program to represent each module. The code of selected programs is given in Appendix. Table 1 shows the results of the experiments. We can categorise the results in the following four parts.

Point Fault Domain: Two separate programs P1 and P2 (Appendix) were tested with ADFD strategy in YETI to get the chart for point fault domain in one and two dimension program. Figure 6 represent point fault domain in one dimension whereas Figure 7 represent point fault domain in two dimension program. ADFD strategy present ranges for pass and fail values for each program in both text (Table 1, Serial No. 1) and graphical form (Figure 6 and 7).

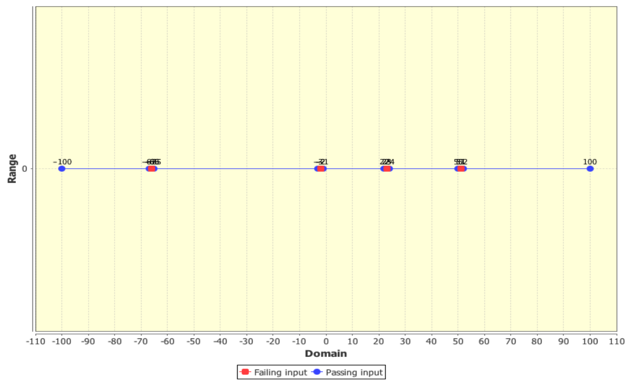


Figure 6: Chart generated by ADFD strategy presenting point fault domain in one dimension module

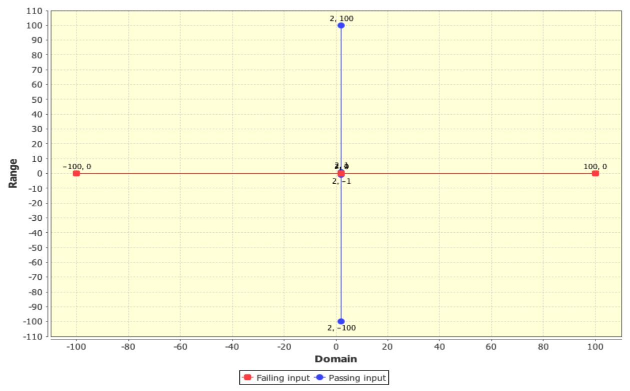


Figure 7: Chart generated by ADFD strategy presenting point fault domain in two dimension module

Block Fault Domain: Two programs P1 and P2 (Appendix) of one and two dimension are tested to get Figure 8 and 9 representing block fault domain. The pass and fail values for each block fault program is given in (Table 1, Serial No. 2).

S. No	Fault Domain	Module Dimension	Specific Fault	Pass Domain	Fail Domain
1	Point	One	PFDOneA(i)	-100 - -67, -65 - -3, -1 - 50, 2 - 22, 24 - 50, 52 - 100	-66, -2, 23, 51
		Two	PFDTwoA(2, i)	(2, 100) - (2, 1), (2, -1) - (2, -100)	(2, 0)
			PFDTwoA(i, 0)	Nil	(-100, 0) - (100, 0)
2	Block	One	BFDOneA(i)	-100 - -30, -25 - -2, 2 - 50, 55 - 100	-1 - 1, -29 - -24, 51 - 54,
		Two	BFDTwoA(-2, i)	(-2, 100) - (-2, 20), (-2, -1) - (-2, -100)	(-2 , 1) - (-2, 19), (-2, 0)
			BFDTwoA(i, 0)	Nil	(-100, 0) - (100, 0)
3	Strip	One	SFDOneA(i)	-100 - -5, 35 - 100	-4, 34
		Two	SFDTwoA(-5, i)	(-5, 100) - (-5, 40), (-5, 0) - (-5, -100)	(-5, 39) - (-5, 1), (-5, 0)
			SFDTwoA(i, 0)	Nil	(-100, 0) - (100, 0)

Table 1: Pass and Fail domain with respect to one and two dimensional program

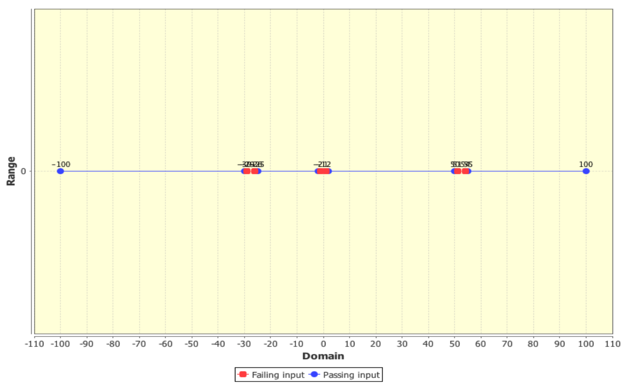


Figure 8: Chart generated by ADFD strategy presenting block fault domain in one dimension module

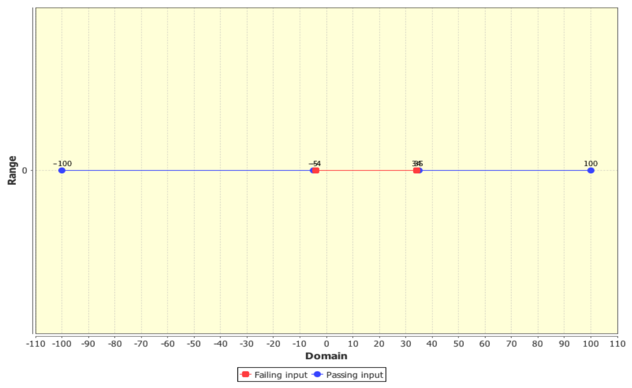


Figure 10: Chart generated by ADFD strategy presenting strip fault domain in one dimension module

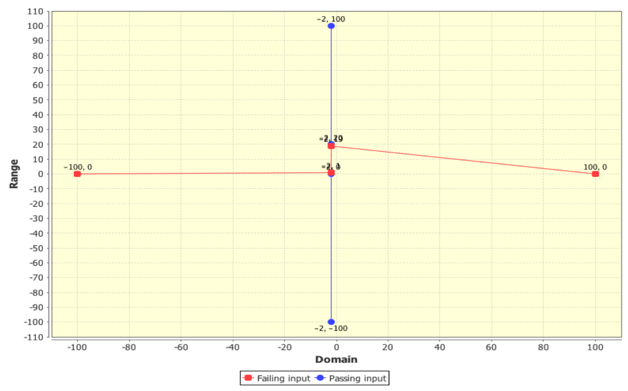


Figure 9: Chart generated by ADFD strategy presenting block fault domain in two dimension module

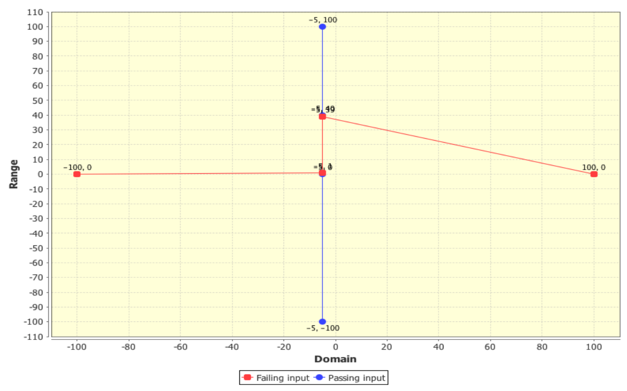


Figure 11: Chart generated by ADFD strategy presenting strip fault domain in two dimension module

Hybrid Fault Domain in one and —- Dimension

Strip Fault Domain: Two programs P1 and P2 (Appendix) of one and two dimension are tested to get Figure 10 and 11 representing strip fault domain. The pass and fail values for each strip fault program is given in (Table 1, Serial No. 3).

5. DISCUSSION

ADFD strategy with a simple graphical user interface is a completely automated process to identify and plot the pass and fault domains on the chart. Since the default settings are all set to optimum the user needs only to specify the module to be tested and click ”plot domain” button to plot domains.

ADFD strategy can effectively identify faults and faults domain in a program. Identification of fault domain is simple for one and two dimension numerical program but the difficulty increases as the program dimension increases beyond two. Similarly no clear boundaries are defined for non numerical data therefore it is not possible to plot domains for non numerical data unless some boundary criteria is defined.

ADFD strategy initiate testing with random+ strategy to find the fault and later switch to brute-force testing to apply all the values between upper and lower bound for finding pass and fault domain.

The main factor dependant on test execution time is the lower and upper bound. If the lower and upper bound is set to maximum or whole input domain then the test duration is maximum. Beside this test duration is also influenced by the identification of the fault and the complexity of module under test.

ADFD strategy can help the debuggers in two ways. First, it reduces the to and from movement of the project between the testers and debuggers as it identity all the faults in one go. Second, it identify locations of all fault domains across the input domain in a user friendly way helping debugger to fix the fault keeping in view its all occurrences.

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

```
/**
 * Dynamically generated code by ADFD strategy
 * @author (Mian and Manuel)
 */
import java.io.*;
import java.util.*;

public class C0
{
 public static ArrayList<Integer> pass = new ArrayList<Integer>();
 public static ArrayList<Integer> fail = new ArrayList<Integer>();
 public static boolean startedByFailing = false;
 public static boolean isCurrentlyFailing = false;
 public static int start = -80;
 public static int stop = 80;

 public static void main(String []argv){
  checkStartAndStopValue(start);
  for (int i=start+1;i<stop;i++){
   try{
PointDomainOneArgument.pointErrors(i);
   if (isCurrentlyFailing)
  {
fail.add(i-1);
fail.add(0);
pass.add(i);
pass.add(0);
  isCurrentlyFailing=false;
  }
 }
  catch(Throwable t) {
  if (!isCurrentlyFailing)
  {
pass.add(i-1);
pass.add(0);
fail.add(i);
fail.add(0);
isCurrentlyFailing = true;
 }
 }
 }
 checkStartAndStopValue(stop);
  printRangeFail();
  printRangePass();
  }

 public static void printRangeFail() {
   try {
   File fw = new File("Fail.txt");
   if (fw.exists() == false) {
     fw.createNewFile();
   }
     PrintWriter pw = new PrintWriter(new FileWriter (fw, true));
     for (Integer i1 : fail) {
      pw.append(i1+"\n");
   }
   pw.close();
   }
   catch(Exception e) {
   System.err.println(" Error : e.getMessage() ");
   }
 }
```

```
  public static void printRangePass() {
   try {
   File fw1 = new File("Pass.txt");
   if (fw1.exists() == false) {
     fw1.createNewFile();
   }
     PrintWriter pw1 = new PrintWriter(new FileWriter (fw1, true));   for (Integer i2 : pass) {
      pw1.append(i2+"\n");
   }
   pw1.close();
   }
   catch(Exception e) {
   System.err.println(" Error : e.getMessage() ");
   }
 }
   public static void checkStartAndStopValue(int i) {
   try {
   PointDomainOneArgument.pointErrors(i);
 pass.add(i);
pass.add(0);
  }
  catch (Throwable t) {
  startedByFailing = true;
  isCurrentlyFailing = true;
  fail.add(i);
 fail.add(0);
 }
 }
}
```