Automated Discovery of Failure Domain

Mian Ahmad and Manuel Oriol

University of York, Department of Computer Science,
Deramore Lane, YO10 5GH YORK, United Kingdom

Abstract. There are many random test strategies based on the presence of point, block and strip fault domains across the whole input domain. As yet no particular, fully automated test strategy has been proposed for the evaluation of these fault domains. We therefore have developed Automated Discovery of Failure Domain (ADFD), a new random test strategy that finds the fault domain, if any, for a given System Under Test (SUT). Furthermore, it also provides visualization of the pass and fail domain found in the SUT. In this paper we describe ADFD strategy, its implementation in YETI and illustrate its working with the help of an example. We report on experiments in which we tested error seeded one and two-dimensional numerical programs with ADFD strategy. Our experimental results show that for each SUT, ADFD strategy successfully performs identification of faults domains and their representation on graphical chart.

Keywords: automated, software, testing, random, YETI, failure domain, point, block, strip

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 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 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 widely used in automated tools. It is intuitively simple and easy to implement [2], [3]. It involves minimum or no overhead in input selection and lacks human bias [4], [5]. While pure random testing has many benefits, there are some limitations as well,

including low code coverage [6] and discovery of lower number of faults [7]. To overcome these limitations many researchers successfully refined pure random testing while keeping its benefits intact. Adaptive Random Testing (ART) is the most significant refinements of random testing. Experiments performed using ART showed up to 50% better results compared to the traditional/pure random testing [8]. Similarly Restricted Random Testing (RRT) [9], Mirror Adaptive Random Testing (MART) [10], Adaptive Random Testing for Object Oriented Programs (ARTOO) [2], Directed Adaptive Random Testing (DART) [11], Lattice-based Adaptive Random Testing (LART) [12] and Feedback-directed Random Testing (FRT) [13] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

All the above-mentioned variations in random testing are based on the observation of Chan et. al., [14] that failure causing inputs across the whole input domain form certain kinds of domains. They classified these domains into point, block and strip fault domain. In Figure 1 the square box represents the whole input domain. The black point, block and strip area inside the box represent the faulty values while white area inside the box represent legitimate values for a specific system. They further suggested that the effectiveness of testing could be improved by taking into consideration these faulty domains.

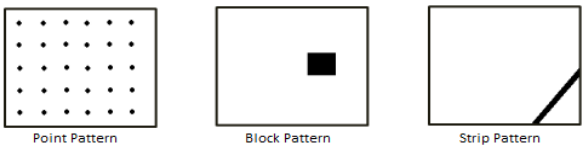


Fig. 1. Failure domains across input domain [14]

It is interesting that where many random strategies are based on the principle 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), which not only finds the pass and fail input values but also finds their domains. The idea of identification of pass and fail domain is attractive as it provides an insight of the domains in the given SUT. Some important aspects of ADFD strategy presented in the paper include:

- Implementation of the new ADFD strategy in York Extensible Testing Infrastructure (YETI) tool.
- Evaluation to assess ADFD strategy by testing classes with different fault domains.
- Decrease in overall test duration by identification of all the fault domains instead of a single instance of fault.
- Increase in test efficiency by helping debugger to keep in view all the fault occurrences when debugging.

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 the threats to validity. Section 7 presents related work and Section 8, concludes the study.

2 Automated Discovery of Failure Domain

Automated Discovery of Failure Domain (ADFD) strategy is proposed as improvement on R+ strategy with capability of finding faults as well as the fault domains. The output produced at the end of test session is a chart showing the passing value or range of values in green and failing value or range of values in red. The complete workflow of ADFD strategy is given in Figure 3.

The process is divided into five major steps given below and each step is briefly explained in the following paras.

1. GUI front-end for providing input
2. Automated finding of fault
3. Automated generation of modules
4. Automated compilation and execution of modules to discover domains
5. Automated generation of graph showing domains

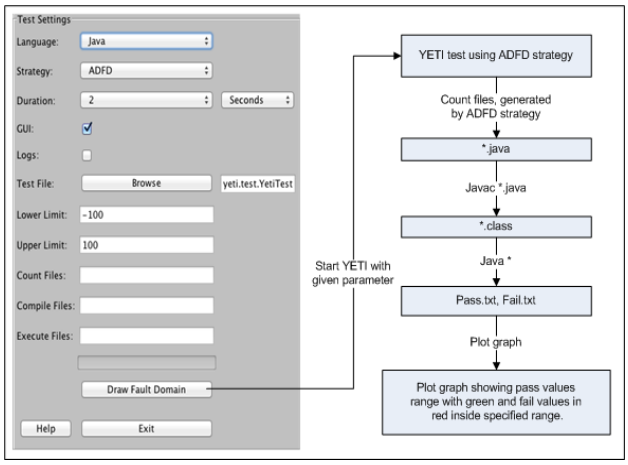


Fig. 2. Work flow of ADFD strategy

GUI front-end for providing input:

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

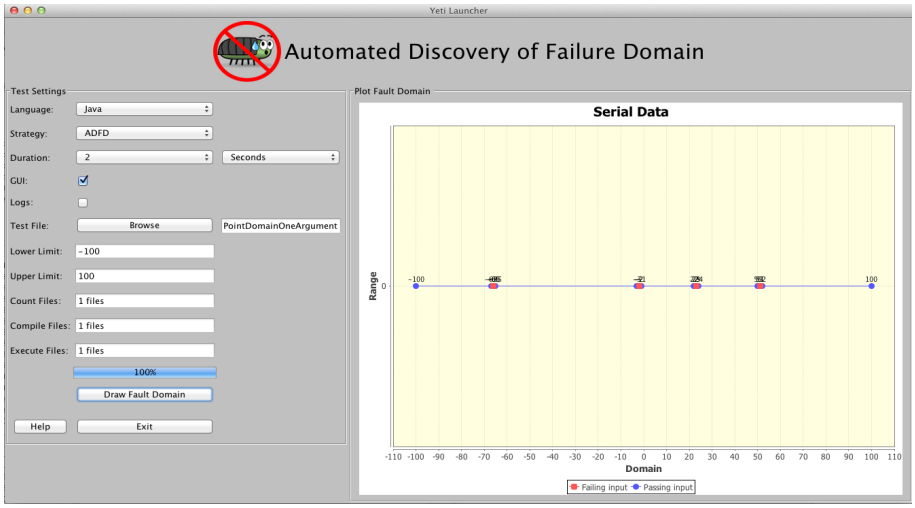


Fig. 3. Front-end of ADFD strategy

form of java byte code. In addition it also takes minimum and maximum values for the search for fault domain in the specified range. Default range for minimum and maximum is Integer.MIN_INT and Integer.MAX_INT respectively.

Automated finding of fault:

To find the failure domain for a specific fault, the first requirement is to identify that fault in the system. ADFD strategy extends R+ strategy and to find the fault ADFD strategy rely on R+ strategy. Random+ (R+) is an improvement over random strategy with preference to the boundary values to provide better fault finding ability. ADFD strategy is implemented in YETI tool which is famous for its simplicity, high speed and proven ability of finding potentially hazardous faults in many systems [15], [16]. 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 domains in the given SUT. These programs with ".java" extensions are generated through dynamic compiler API included in Java 6 under javax.tools package. The number of programs generated can be one or more, 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 the program keeps one or more than one argument constant and only one argument variable in the generated program.

Automated compilation and execution of modules to discover domains:

The java modules generated in previous step are compiled using "javac *" command to get their binary ".class" files. The "java *" command is applied to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument receive all the values, from minimum to maximum, specified in the beginning of the test. After execution is completed we get two text files of "Pass.txt" and "Fail.txt". Pass file contains all the values for which the modules behave correctly while fail file contains all the values for which the modules fail.

Automated generation of graph showing domains:

The values from the pass and fail files are used to plot (x, y) chart using JFreeChart. JFreeChart is a free open-source java library that helps developers to display complex charts and graphs in their applications [17]. Green colour lines with circle represents pass values while red colour line with squares represents the fail values. Resultant graph clearly depicts both the pass and fail domain across the specified input domain. The graph shows red points in case the program fails for only one value, blocks when the program fails for multiple values and strips when a program fails for a long range of values.

3 Implementation

We implemented ADFD strategy in a tool called York Extensible Testing Infrastructure (YETI). YETI is available in open-source at `http://code.google.com/p/yeti-test/`. In this section we give a brief overview of YETI focusing only on the parts relevant to the implementation of ADFD strategy. We further define integration of ADFD strategy in YETI and a program is used as an example to illustrate the working of ADFD strategy. For more details on YETI tool please see [18], [19], [20], [15], [16].

3.1 York Extensible Testing Infrastructure

YETI 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 functional, procedural and object oriented languages.

YETI consists 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 [18], [20]. Both the languages and strategies section has a pluggable architecture, to easily incorporate new strategies and languages, making it a conducive choice to implement ADFD strategy. YETI is capable of generating the test cases to reproduce the faults found during the current test session.

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.

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 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 (program 1). This program is automatically compiled to get binary and then executed to find the pass and fail domain inside the specified range as. Identified domains are plotted on two-dimensional graph. It is evident from output Figure 4 that use of ADFD strategy not only find all the faults but also pass and fail domains.

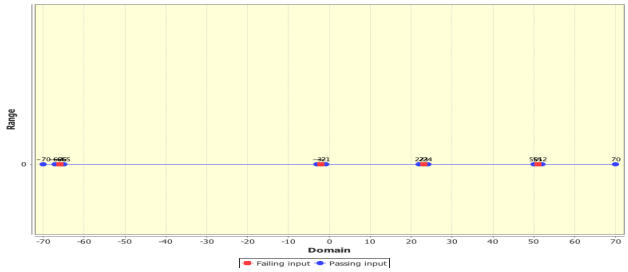


Fig. 4. ADFD strategy plotting pass and fault domain of the given class

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.

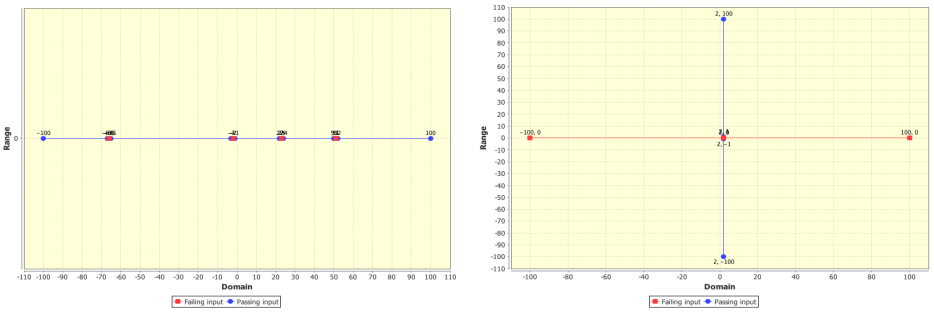
All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35].

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.

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

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

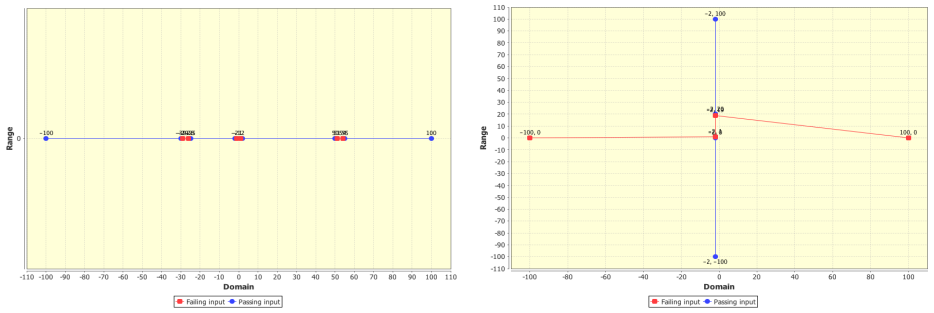
Point Fault Domain: Two separate programs P1 and P2 (Appendix) were tested with ADFD strategy in YETI to get the chart for point fault domain in one and two dimension program. Figure 5(a) represent point fault domain in one dimension whereas Figure 5(b) represent point fault domain in two dimension program. ADFD strategy present ranges for pass and fail values for each program in both text (Table 1, Serial No. 1) and graphical form (Figure 5(a) and 5(b)).



(a) One dimension module (b) Two dimension module

Fig. 5. Chart generated by ADFD strategy presenting point fault domain

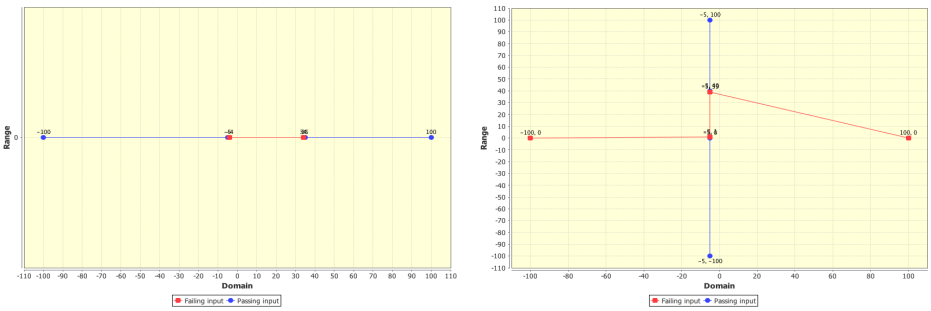
Block Fault Domain: Two programs P1 and P2 (Appendix) of one and two dimension are tested to get Figure 6(a) and 6(b) representing block fault domain. The pass and fail values for each block fault program is given in (Table 1, Serial No. 2).



(a) One dimension module (b) Two dimension module

Fig. 6. Chart generated by ADFD strategy presenting block fault domain

Strip Fault Domain: Two programs P1 and P2 (Appendix) of one and two dimension are tested to get Figure 7(a) and 7(b) representing strip fault domain. The pass and fail values for each strip fault program is given in (Table 1, Serial No. 3).



(a) One dimension module (b) Two dimension module

Fig. 7. Chart generated by ADFD strategy presenting Strip fault domain

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 start test execution. All the steps including Identification of fault, generation of dynamic java program to find domain of the identified fault, saving the program to a permanent media, compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automated without any human intervention.

In the experiments (section 4), the ADFD strategy effectively identified 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 overhead in terms of execution time associated with ADFD strategy is dependant mainly on the lower and upper bound. If the lower and upper bound is set to maximum range (i.e. maximum for int is Integer.Max_Int and minimum is Integer.Min_Int) then the test duration is maximum. It is rightly so because for identification of fault domain the program is executed for every input available

in the specified range. Similarly increasing the range also shrink the produced graph making it difficult to identify clearly point, block and strip domain unless they are of considerable size. Beside range factor, 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 Threats to Validity

The major external threat to the generalisation of the presented results is the selection of small set of error seeded programs of only primitive types such as integer. Future work should expand this study to more general purpose real world production application.

Another issue is that where we can plot numerical data easily as distinctive units, it is difficult to split into units and plot composite objects with many fields. Though work has been done to quantify composite objects into units on the basis of multiple features [21] which can make them easy to plot. However plotting composite objects is beyond the scope of this study.

An internal threat to validity include evaluating program with complex input arguments. ADFD strategy has so far only considered scaler (e.g. integer) of one and two dimensions that are easy to plot on the graph but plotting domain of programs with complex (non scaler) and more than two dimension argument is much more complicated.

Finally, plotting pass or fail values between a large input domain (e.g Integer.MIN_INT to Integer.MAX_INT) is difficult to adjust and don't give an easy to understand view. Although zoom option is available to zoom into the areas of interest on the graph.

7 Related Work

Traditional random testing is quick, easy to implement and free from any bias. Although it enjoys many benefits the fault finding ability of traditional random testing is low and received some criticism [22], [6]. To overcome performance issues without compromising its benefits various researcher altered its algorithm (explained in Introduction section). Most of the alteration is based on the existence of failure domains across the input domain [14].

Identification, Classification of pass and fail domains across the input domain and visualisation of domains has received very little attention in the research community. Podgurski et. al., [23] proposes a semi-automated procedure only to classify similar faults and plot them using a hierarchical Multi Dimension Scaling (HMDS) algorithm. A tool named Xslice [24] differentiate visually the execution

slices of passing and failing part of a test. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [25].

Limitations of the above tools is that they are not fully automated and require human interaction during execution. Similarly these tools only concentrate on the already existing test cases where as ADFD strategy discover faults, identify pass and fault domains and visualise them in a fully automated manner.

8 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 arithmetic 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.

9 The References Section

1. Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley (1995)
2. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. (may 2008) 71 –80
3. Forrester, J.E., Miller, B.P.: 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, Berkeley, CA, USA, USENIX Association (2000) 6–6
4. Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, Wiley (1994) 970–978

5. Linger, R.C.: Cleanroom software engineering for zero-defect software. In: Proceedings of the 15th international conference on Software Engineering. ICSE '93, Los Alamitos, CA, USA, IEEE Computer Society Press (1993) 2–13
6. Offutt, A.J., Hayes, J.H.: A semantic model of program faults. SIGSOFT Softw. Eng. Notes **21**(3) (May 1996) 195–200
7. Chen, T., Yu, Y.: On the relationship between partition and random testing. Software Engineering, IEEE Transactions on **20**(12) (dec 1994) 977 –980
8. Chen, T.Y.: Adaptive random testing. Eighth International Conference on Qualify Software **0** (2008) 443
9. Chan, K.P., Chen, T.Y., Towey, D.: Restricted random testing. In: Proceedings of the 7th International Conference on Software Quality. ECSQ '02, London, UK, UK, Springer-Verlag (2002) 321–330
10. Chen, T., Merkel, R., Wong, P., Eddy, G.: Adaptive random testing through dynamic partitioning. In: Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on, IEEE (2004) 79–86
11. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Sigplan Notices. Volume 40., ACM (2005) 213–223
12. Mayer, J.: Lattice-based adaptive random testing. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM (2005) 333–336
13. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th international conference on Software Engineering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 75–84
14. Chan, F., Chen, T., Mak, I., Yu, Y.: Proportional sampling strategy: guidelines for software testing practitioners. Information and Software Technology **38**(12) (1996) 775–782
15. Oriol, M.: York extensible testing infrastructure (2011)
16. Oriol, M.: Random testing: Evaluation of a law describing the number of faults found. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. (april 2012) 201 –210
17. Gilbert, D.: The JFreeChart class library version 1.0.9: Developer's guide. Refinery Limited, Hertfordshire (2008)
18. Oriol, M., Tassis, S.: Testing .net code with yeti. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems. ICECCS '10, Washington, DC, USA, IEEE Computer Society (2010) 264–265
19. Oriol, M., Ullah, F.: Yeti on the cloud. In: Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops. ICSTW '10, Washington, DC, USA, IEEE Computer Society (2010) 434–437
20. Oriol, M.: The york extensible testing infrastructure (yeti). (2010)
21. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Object distance and its application to adaptive random testing of object-oriented programs. In: Proceedings of the 1st international workshop on Random testing. RT '06, New York, NY, USA, ACM (2006) 55–63
22. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. Wiley (2011)
23. Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Automated support for classifying software failure reports. In: Software Engineering, 2003. Proceedings. 25th International Conference on. (may 2003) 465 – 475
24. Agrawal, H., Horgan, J., London, S., Wong, W.: Fault localization using execution slices and dataflow tests. In: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on. (oct 1995) 143 –151

25. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, New York, NY, USA, ACM (2002) 467–477

Appendix:

Program 1 Program generated by ADFD on finding fault in SUT

```
/**
 * Dynamically generated code by ADFD strategy
 * after a fault is found in the SUT.
 * @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);
        }
    }
}
```