Analysis of Failure Domain by ADFD+ and Daikon

Mian Asbat Ahmad
Department of Computer Science
The 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

This paper verifies the accuracy of invariants generated automatically by Daikon and suggests how to improve their quality. To achieve this, it uses a newly developed Automated Discovery of Failure Domain+ (ADFD+) technique. ADFD+ is a testing frame-work which after identifying a failure searches its surrounding to find its domain within the set range. The result obtained is presented graphically on a two-dimension chart.

Several error-seeded one and two-dimensional numerical programs with point, block and strip failure domain were evaluated independently for 30 times by both ADFD+ and Daikon. On analysis of results, it is found that where Daikon generates the correct invariants, it was not good enough to identify the exact failure boundaries.

It is concluded that the invariants generated by Daikon can be made further effective if the boundary values of the failure domain identified by ADFD+ are passed to the Daikon as test cases.

Keywords

software testing, automated random testing

1. INTRODUCTION

Testing is an essential and most widely used method for verification and validation process. Efforts have been continuously made by researchers to make it more and more effective and efficient. Testing is effective when it finds maximum number of faults in minimum number of test cases and efficient when it executes maximum number of test cases in minimum possible time. Upgrading existing techniques and developing new test strategies focus on increasing test effectiveness while automating one or more components or complete system aims at increasing efficiency.

Boundary Value Analysis (BVA) is one of the technique used of increasing test effectiveness. In BVA test cases with boundary values are added to the test suite with the assumption that errors reside along the boundaries [8]. Daikon [4] is an automatic tool used to improve the efficiency. It saves testers time by automatically generating likely program invariants.

However, the two approaches can adversely affect the testing process if wrong boundaries or invariants are taken into consideration. It is therefore motivating to accurately identify the boundaries of the input domain in BVA and measure the degree of correctness of auto-generated invariants by Daikon in the case of point, block and strip failure domain. To analyse the failure domains the ADFD+ technique was developed and experiments were conducted by testing several error-seeded one and two-dimensional numerical programs with ADFD+ and Daikon. The results obtained were analysed and reported.

The main contributions of the study are:

- **ADFD+:** It is an extension of Automated Discovery of Failure Domain (ADFD) strategy developed by Ahmad and Oriol [1]. The new technique improves the search algorithm of ADFD and makes the report more intuitive (Section 3).
- **Implementation of ADFD+:** It is implemented and integrated in the York Extensible Testing Infrastructure (Section 3.2).
- **Evaluation:** The results generated by ADFD+ and Daikon about failure domains in the error-seeded programs are evaluated (Section 5). The results show that although Daikon generate invariant to identify the failure yet it is not able to identify the boundary of failure domain as accurately as ADFD+.
- **Future work:** ADFD+ can be extended to find and plot failure domains in multi-dimensional non-numerical programs (Section 10).

2. PRELIMINARIES

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [5, 9, 10]. According to Chan et al. [2] the clusters are arranged in the form of point, block and strip failure domain. In the point domain the failure revealing inputs are stand-alone, and spread through out the input domain. In block domain the failure revealing inputs are clustered in one or more contiguous areas. In strip domain the failure revealing inputs are clustered in one long elongated area. Figure 1 shows the failure domains in two-dimensional input domain.

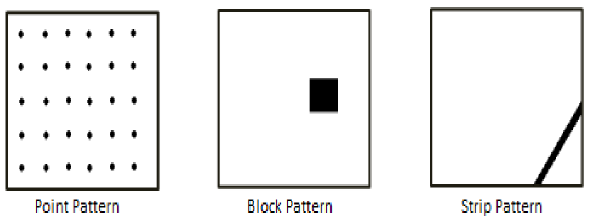


Figure 1: Failure domains across input domain [2]

3. AUTOMATED DISCOVERY OF FAILURE DOMAIN+

ADFD+ is an improved and extended form of ADFD strategy developed previously by Ahmad and Oriol [1]. It is an automated framework which finds the failures and their domains within a specified range and present them on a graphical chart.

The main improvements of ADFD+ over ADFD strategy are stated as follows.

- ADFD+ generates a single Java file dynamically at run time to plot the failure domains as compared to one Java file per failure in ADFD. This saves sufficient time and makes the execution process quicker.
- ADFD+ uses (x, y) vector series to represent failure domains as opposed to the (x, y) line series in ADFD. The vector series allows more flexibility and clarity to represent a failure and its domain.
- ADFD+ takes a single value as range with in which the strategy search for a failure domain whereas ADFD takes two values for lower and upper bound representing x and y-axis respectively.
- In ADFD+, the algorithm of dynamically generating Java file, created at run-time after a failure is detected, is made more simplified and efficient.
- In ADFD+, the failure domain is focused in the graph, which gives a clear view of, pass and fail points. The points are also labelled for clarification as shown in Figure 2.

3.1 Workflow of ADFD+

ADFD+ is a completely automatic process and all the user has to do is to specify the program to test and click the *DrawFaultDomain* button. The default value for range is set to 5, which means that ADFD+ will search 83 values around the failure. On clicking the button YETI is executed with ADFD+ strategy to search for a failure in two-dimensional program. On finding a failure the ADFD+ strategy creates a Java file which contains calls to the program on the failing value and its surrounding values within the specified range. The Java file is compiled and executed and the result is analysed to check for pass and fail values. Pass and fail values are stored in pass and fail text files respectively. At the end of test, all the values are plotted on the graph with pass values in blue and fail values in red colour as shown in Figure 2.

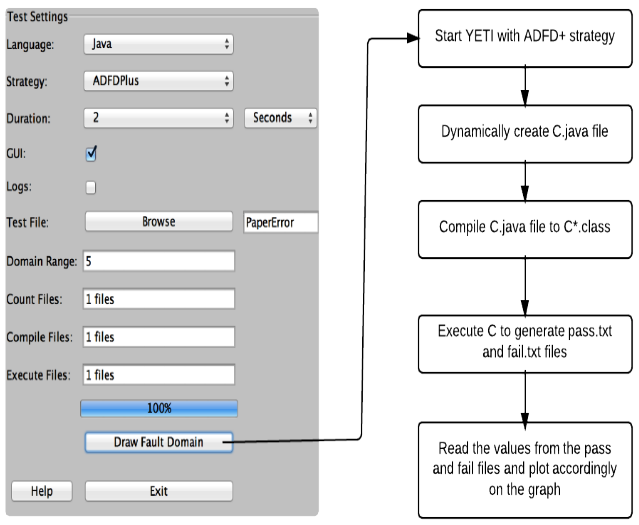

Test Settings
Language: Java
Strategy: ADFDPlus
Duration: 2 Seconds
GUI: ☑
Logs: ☐
Test File: Browse PaperError
Domain Range: 5
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files
100%
Draw Fault Domain
Help
Exit
Start YETI with ADFD+ strategy
Dynamically create C.java file
Compile C.java file to C*.class
Execute C to generate pass.txt and fail.txt files
Read the values from the pass and fail files and plot accordingly on the graph

Figure 2: Workflow of ADFD+

3.2 Implementation of ADFD+

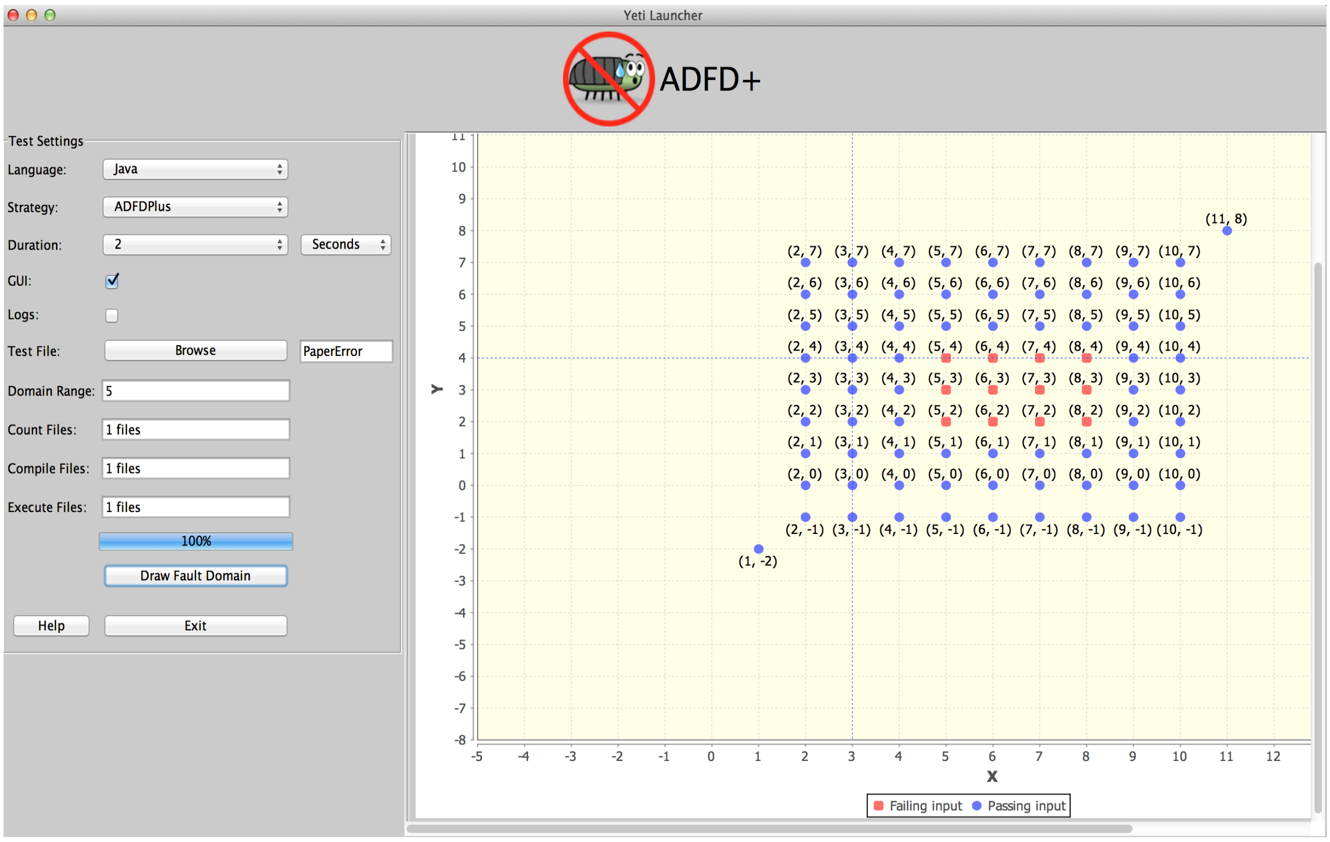
The ADFD+ technique is implemented in YETI. The tool YETI is available in open-source at `http://code.google.com/p/yeti-test/`. A brief overview of YETI is given with the focus on parts relevant to implementation of ADFD+ strategy.

YETI is a testing tool developed in Java that tests programs using random strategies in an automated fashion. 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 including core infrastructure for extendibility, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both strategies and languages sections have pluggable architecture to easily incorporate new strategies and languages making YETI a favourable choice to implement ADFD+ strategy. YETI is also capable of generating test cases to reproduce the failures found during the test session. The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. In strategies package, on top of the hierarchy, is an abstract class $YetiStrategy$, which is extended by $YetiRandomPlusStrategy$ and is further extended to get ADFD+ strategy.

3.3 Example to illustrate working of ADFD+

Suppose we have the following error-seeded class under test. From the program code, it can be easily noticed that an


Yeti Launcher

Test Settings
Language: Java
Strategy: ADFDPlus
Duration: 2 Seconds
GUI: ☑
Logs: ☐
Test File: PaperError
Domain Range: 5
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files
100%

Figure 3: The output of ADFD+ for the above code.

ArithmeticException (DivisonByZero) failure is generated when the value of variable x ranges between 5 to 8 and the value of variable y between 2 to 4.

```
public class Error {

  public static void Error (int x, int y){

    int z;

    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
         {
             z = 50/0;
         }
   }
}
```

On test execution, the ADFD+ strategy evaluates the class with the help of YETI and finds the first failure at x = 6 and y = 3. Once a failure is identified ADFD+ uses the surrounding values around it to find a failure domain. The range of surrounding values is limited to the value set by the user in the *DomainRange* variable. When the value of *DomainRange* is 5, ADFD+ evaluates total of 83 values of x and y around the found failure. All evaluated (x, y) values are plotted on a two-dimensional graph with red filled circles indicating fail values and blue filled circles indicating pass values. Figure 3 shows that the failure domain forms a block pattern and the boundaries of the failure are $(5,2),(5,3),(5,4),(6,2),(6,4),(7,2),(7,4),(8,2),(8,3),(8,4)$.

4. RANDOOP

Random tester for object oriented programs (Randoop) is the tool used for implementing FDRT technique [7]. Randoop is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes a set of classes, contracts, filters and time limit as input and gives a suite of JUnit for Java and NUnit for .Net program as output. Each unit test in a test suite is a sequence of method calls (hereafter referred as sequence). Randoop builds the sequence incrementally by randomly selecting a public method from the class under test and arguments for these methods are selected from the predefined pool in case of primitive types and as sequence of null values in case of reference type. Randoop maintains two sets called `ErrorSeqs` and `NonErrorSeqs` to record the feedback. It extends `ErrorSeqs` set in case of contract or filter violation and `NonErrorSeqs` set when no violation is recorded in the feedback. The use of this dynamic feedback evaluation at runtime brings an object to an interesting state. On test completion, `ErrorSeqs` and `NonErrorSeqs` are produced as JUnit/NUnit test suite. In terms of coverage and number of faults discovered, Randoop implementing FDRT was compared with JCrasher and JavaPathFinder and 14 libraries of both Java and .Net were evaluated [?]. The results showed that Randoop achieved more branch coverage and better fault detection than JCrasher.

5. COMPARISON OF ADFD+ AND RANDOOP

Because of using error-seeded one and two dimensional numerical programs, we were aware of the failure domain present in each program. The correct identification and presentation of the failure domain by ADFD+ prove the correct working of ADFD+. We then evaluated the same program by Daikon and plot its results. The unit test cases required by Daikon for generating invariants were generated using Randoop [7]. YETI being capable of generating the test cases is not used for this step to keep the second completely independent from first.

5.1 Research questions

The following research questions have been addressed in this study:

1. Is ADFD+ and Randoop capable of identifying the failure?
2. Is ADFD+ and Randoop capable of identifying the failure domain?
3. Is ADFD+ and Randoop capable of identifying the boundaries of the failure domain?

5.2 Experimental setup

To evaluate the performance of ADFD+ and Daikon, we carried out testing of several error-seeded one and two-dimensional numerical programs written in Java. The programs were divided in to two sets. Set A and B contains one and two-dimensional programs respectively. Each program was injected with at least one failure domain of point, block or strip nature. Every program was tested independently for 30 times by both ADFD+ and Daikon. The external parameters were kept constant in each test run and the initial test cases required by Daikon were generated by using an automated testing tool Randoop. The code for the programs under test is given in Appendix **??** while the test details are presented in Table **??**.

Every class was evaluated through 10^5 calls in each test session of ADFD+. Due to the absence of contracts and assertions in the code under test, undeclared exceptions were taken as failures in accordance with the previous studies [1, 6]. All tests were performed with a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon processor with 6 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

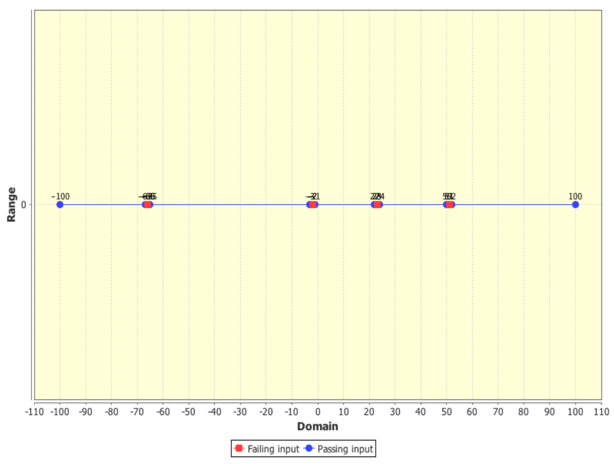
6. RESULTS

The results are split in to two sections for convenience. The first section (6.1) contains results of the experiments performed by ADFD+ strategy and the second section (6.2) contains results of Daikon on one and two dimensional numerical programs.

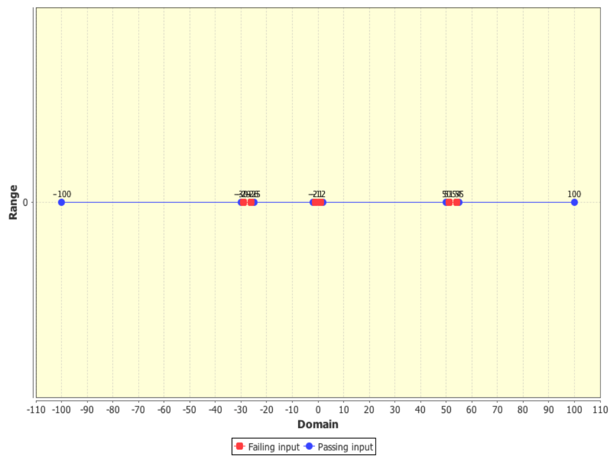
6.1 ADFD+ results

6.1.1 Test of one-dimension programs by ADFD+

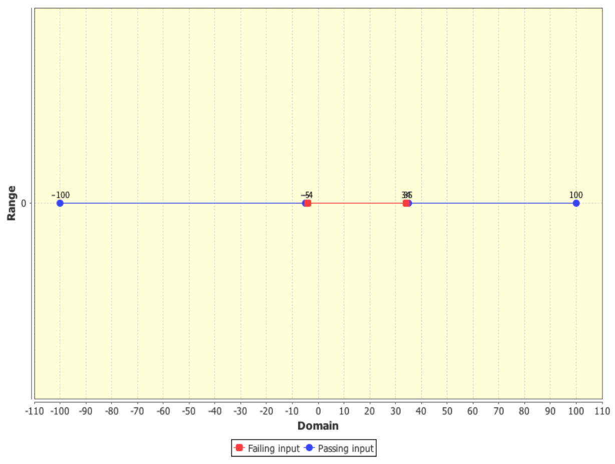
In each of the 30 experiments, The ADFD+ successfully discovered and plotted the failure domains for point, block and strip pattern as shown in the Figure 4. The lower and upper bound for each experiment are set to -100 and 100 respectively.



(a) Point failure domain in one-dimension



(b) Block failure domain in one-dimension



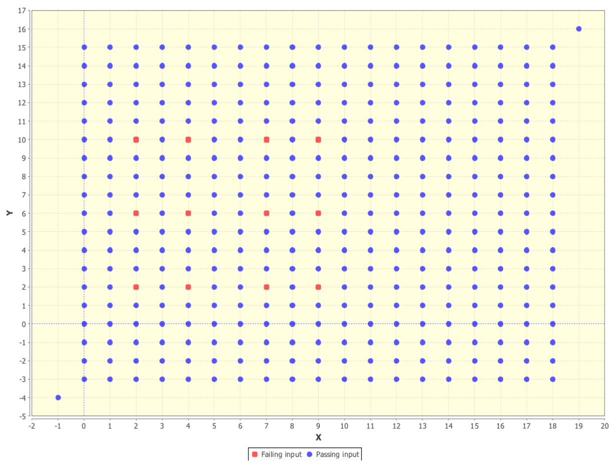
(c) Strip failure domain in one dimension

Figure 4: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

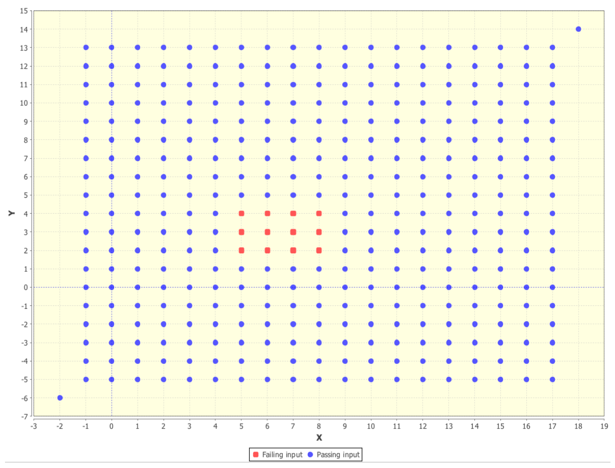
6.1.2 Test of two-dimension programs by ADFD+

Point failure	Block failure	Strip failure
x = -66	x = -1, 0, 1	x = -4 — 34
x = -2	x = -29, -28, -27, -26	
x = 51	x = 51, 52, 53, 54	
x = 23		

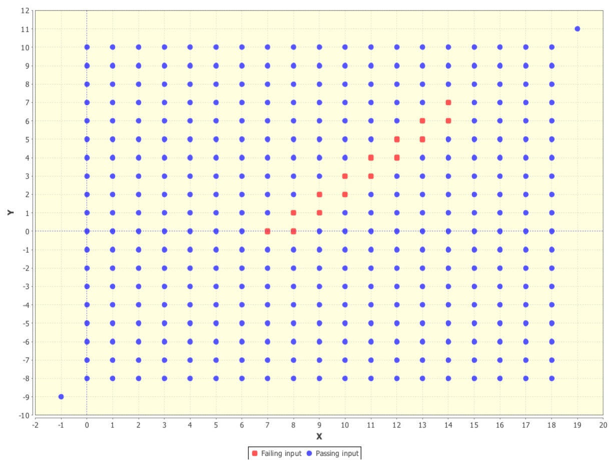
Table 1: Table depicting values of x and y arguments responsible for forming point, block and strip failure domain in Figure 4(a), Figure 4(b) and Figure= 4(c) respectively



(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Figure 5: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

In each of the 30 experiments, The ADFD+ once again successfully discovered and plotted the failure domain for point, block and strip failure domain as shown in the Figure 5. The range value for each experiment is set to 10. Labels are disabled in the charts given in Figure 5 for clarity purpose. The failure values in each of the point, block and strip failure domain is given in Table 3.

6.2 Daikon results

In initial set of experiments, Daikon failed to generate an invariant for identifying a single failure. On analysis It is found that Daikon rely on the existing set of test cases to

Point failure	Block failure	Strip failure
x = 2, y = 10	x = 5, y = 2	x = 7, y = 0
x = 4, y = 10	x = 6, y = 2	x = 8, y = 0
x = 7, y = 10	x = 7, y = 2	x = 8, y = 1
x = 9, y = 10	x = 8, y = 2	x = 9, y = 1
	x = 5, y = 3	x = 9, y = 2
	x = 6, y = 3	x = 10, y = 2
	x = 7, y = 3	x = 10, y = 3
	x = 8, y = 3	x = 11, y = 3
	x = 5, y = 4	x = 11, y = 4
	x = 6, y = 4	x = 12, y = 4
	x = 7, y = 4	x = 12, y = 5
	x = 8, y = 4	x = 13, y = 6
		x = 14, y = 6
		x = 14, y = 7

Table 2: Table depicting values of x and y arguments responsible for forming point, block and strip failure domain in Figure 5(a), Figure 5(b) and Figure= 5(c) respectively

generate invariants. Since the test cases produced by Randoop did not generate any test case from failure domain therefore Daikon could not generate an assertion to point out the failure. To enable Randoop to generate test case from failure domain we directed it to produce random values between a limited range. In this way we were able to generate interesting test cases from Randoop. Test results of Randoop are shown in Table **??**

6.2.1 Test of one-dimensional programs by Daikon

6.2.2 Test of two-dimensional programs by Daikon

7. DISCUSSION

We have shown that ADFD+ is a promising technique to find a failure and using it as a focal point find the whole failure domain. We have also shown that ADFD+ can graphically draw the failure domain on a chart. The failure values are drawn in red and the pass values are drawn in green. The pictorial representation of failure domain helps in easily identifying the underlying pattern and its boundaries.

As a pilot study, we also ran an empirical study to evaluate several error-seeded programs. While it would be surprising if production programs produced much different results, it would be worthwhile to check.

More importantly, the implementation of ADFD+ for this pilot study has significant limitations in practice, as it requires only one and two dimensional numerical programs. Though it is not difficult to extend the approach to test more than two-dimensional programs containing other primitive types, it would however be difficult to plot them on the chart as the number of coordinates increases. The approach can also be extended to test object-oriented programs by implementing objects distance proposed by Ciupa et al. [3]. The details of such an implementation will take some effort.

The ADFD+ range value specifies how many values to test around the failure. The range can be set to any number before the test starts. The value of range is directly proportional to the time taken because the higher the range value the higher number of values to test. Higher range value also leads to a very large graph and the tester has to use the zoom feature of graph to magnify the failure region.

8. THREATS TO VALIDITY

The research study faces threats to external and internal validity. The threats to external validity are the same, which are common to most of the empirical evaluations i.e. to what degree the classes under test and test generation tool (Randoop) are representatives of true practice. The classes under test contains failure patterns in only one and two-dimensional input domain. The threats may be reduced to a greater extent in future experiments by taking several types of classes and different test generation tools.

The threat to internal validity includes annotation of invariants that can bias the results, which may have been caused by error-seeded classes used in our experiments. Internal threats may be avoided by taking real classes and failures in the experiments. Moreover, testing a higher number of classes will also increase the validity of the results.

9. CONCLUSION

Automated Discovery of Failure Domain+ (ADFD+) is distinctive from other random test strategies in the sense that it is not only limited to identifying a failure in the program. Instead, the failure is exploited to identify and graphically plot its failure domain.

In the first section, we describe ADFD+ in detail which is based on our previous approach ADFD [1]. We then describe the main improvements of ADFD+ over ADFD.

In the second section, we analysed and compared the results of the experiments performed by both ADFD+ and Daikon in the case of programs with point, block and strip failure domain.

We showed that Daikon lakes to accurately identify the failure boundary and therefore cannot generate invariants for such failures. We further explain why Daikon does not work well for boundary failures. The main reason we identified for this behaviour is Daikons dependence on initial set of test cases, which are required by Daikon for generating invariants. With increase in number of test suite or high quality test suite improves the performance of invariants.

10. FUTURE WORK

The current approach can be extended to a larger set of real world multi-dimensional programs, using real failure instead of error-seeded programs. However, to plot failure domains of complex multi-dimensional nature, more sophisticated graphical tools like Matlab will be required rather than JFreeChart used in the current study. This may not restrict the formation of new failure domains to point, block and strip failure domain in one and two-dimensional numerical programs.

11. ACKNOWLEDGMENTS

Program dimension	Failure domain	Test time	Random range	Total TC	Pass TC	Fail TC
One	Point	60 sec	-100 to 100	82	79	3
One	Block	60 sec	-100 to 100	82	75	7
One	Strip	60 sec	-100 to 100	82	62	22
Two	Point	60 sec	-100 to 100	6563	6521	42
Two	Block	60 sec	-100 to 100	5902	5057	845
Two	Strip	60 sec	-100 to 100	6226	3896	2330

Table 3: Randoop Results

Technique	Dimension	Test cases	Point failure	Block failure	Strip failure
ADFD+	One	N/A			
Daikon	One	10			
Daikon	One	20			
ADFD+	Two	N/A			
Daikon	Two	10			
Daikon	Two	20			

Table 4: Table depicting values of failure points identified by ADFD+ Daikon

The authors thank the Department of Computer Science, University of York for its financial support with the Departmental Overseas Research Scholarship (DORS) award. We also thanks to Richard Page for his valuable help and generous support.

12. REFERENCES

[1] M. A. Ahmad and M. Oriol. Automated discovery of failure domain. *Lecture Notes on Software Engineering*, 03(1):289–294, 2013.

[2] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

[3] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st international workshop on Random testing*, pages 55–63. ACM, 2006.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[5] G. B. Finelli. Nasa software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[6] M. Oriol. 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*, pages 201 –210, april 2012.

[7] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[8] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[9] C. Schneckenburger and J. Mayer. Towards the determination of typical failure patterns. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90–93. ACM, 2007.

[10] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.

Mian Asbat Ahmad is a PhD scholar at the Department of Computer Science, the University of York, UK. He completed his M(IT) and MS(CS) from Agric. University Peshawar, Pakistan in 2004 and 2009 respectively. His research interests include new automated random software testing strategies.

Manuel Oriol is a lecturer at the Department of Computer Science, the University of York, UK. He completed his PhD from University of Geneva and an MSc from ENSEEIHT in Toulouse, France. His research interests include software testing, software engineering, middleware, dynamic software updates, software architecture and real-time systems.

Program 2 Point domain with One argument

```
/**
 * Point Fault Domain example for one argument
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{
    public static void pointErrors (int x){
        if (x == -66 )
            x = 5/0;

        if (x == -2 )
            x = 5/0;

        if (x == 51 )
            x = 5/0;

        if (x == 23 )
            x = 5/0;
    }
}
```

Program 3 Point domain with two argument

```
/**
 * Point Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{

    public static void pointErrors (int x, int y){
        int z = x/y;
    }

}
```

Program 4 Block domain with one argument

```
/**
 * Block Fault Domain example for one arguments
 * @author (Mian and Manuel)
 */

public class BlockDomainOneArgument{

public static void blockErrors (int x){

    if((x > -2) \&\& (x < 2))
        x = 5/0;

    if((x > -30) \&\& (x < -25))
        x = 5/0;

    if((x > 50) \&\& (x < 55))
        x = 5/0;

   }
}
```

Program 5 Block domain with two argument

```
/**
 * Block Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class BlockDomainTwoArgument{

    public static void pointErrors (int x, int y){

        if(((x > 0)\&\&(x < 20)) || ((y > 0) \&\& (y < 20))){
        x = 5/0;
        }

    }

}
```

Program 6 Strip domain with One argument

```
/**
 * Strip Fault Domain example for one argument
 * @author (Mian and Manuel)
 */

public class StripDomainOneArgument{

    public static void stripErrors (int x){

        if((x > -5) \&\& (x < 35))
            x = 5/0;
     }
}
```

Program 7 Strip domain with two argument

```
/**
 * Strip Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class StripDomainTwoArgument{

    public static void pointErrors (int x, int y){

        if(((x > 0)\&\&(x < 40)) || ((y > 0) \&\& (y < 40))){
        x = 5/0;
        }

    }

}
```