ADFD+: An Automatic Tool for Finding and Presenting Failure domains

Mian Asbat Ahmad and Manuel Oriol

Abstract—**This paper presents Automated Discovery of Failure Domain+ (ADFD+), an improvement on our previously developed Automated Discovery of Failure Domain (ADFD) technique. ADFD+ is a random testing technique which after identifying a failure searches its surrounding to find its domain within the set range. The result obtained is graphically presented. To find the effectiveness of our Technique, 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 Randoop. Results indicated that ADFD+ can identify failure and failure-domain sufficiently quick and in fewer number of test cases as compared to Randoop. Additionally ADFD+ presents the failure and failure domains in a graphical form.**

Index Terms—**software testing, automated random testing, ADFD.**

I. 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.

This paper describes ADFD+, a technique for automatically finding failures, failure domains and their representation in Java programs. ADFD+ uses random+ testing [4], [5]. After identification of a failure, ADFD+ searches its surrounding to find its failure domain within the specified range. The result obtained is presented in a graphical form. The performance of ADFD+ is compared with a random testing tool Randoop [11]. The results obtained were analysed and reported.

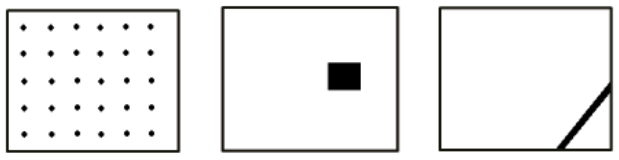
The main contributions of the study are:

- **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 III).
- **Implementation of ADFD+:** It is implemented and integrated in the York Extensible Testing Infrastructure [8] (Section III-B).
- **Evaluation:** The results generated by ADFD+ and Randoop about failure domains in the error-seeded programs are evaluated (Section VI). The results show that although ADFD+ outperform Randoop with respect to time and number of test cases to find a failure domain. Additionally ADFD+ presents the results graphically.
- **Future work:** ADFD+ can be extended to find and plot failure domains in multi-dimensional non-numerical programs (Section XI).

Manuscript received Feb 5, 2014; revised March 20, 2013. The authors are with the Department of Computer Science, University of York, YO10 5DD, UK (e-mail: mian.ahmad@york.ac.uk, manuel.oriol@york.ac.uk).

II. Preliminaries

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [6], [12], [13]. 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.



(a) Point domain (b) Block domain (c) Strip domain

Fig. 1. Failure domains across input domain [2]

III. 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.

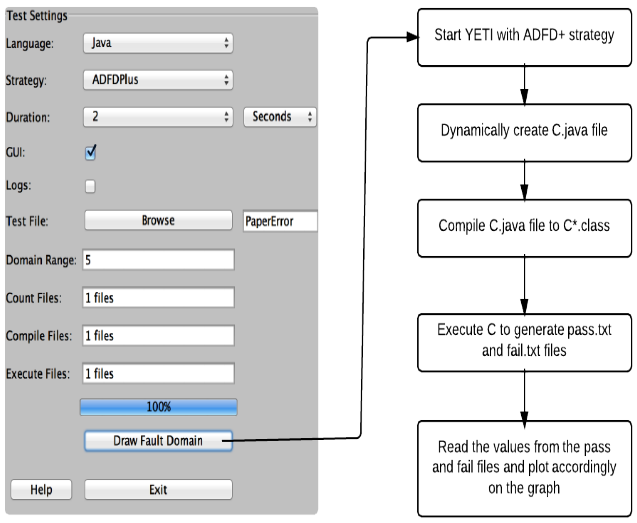
The main improvements of ADFD+ over ADFD strategy are stated as follows.

- 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.

A. 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 3.



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

Fig. 2. Workflow of ADFD+

B. 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.

C. Example to illustrate working of ADFD+

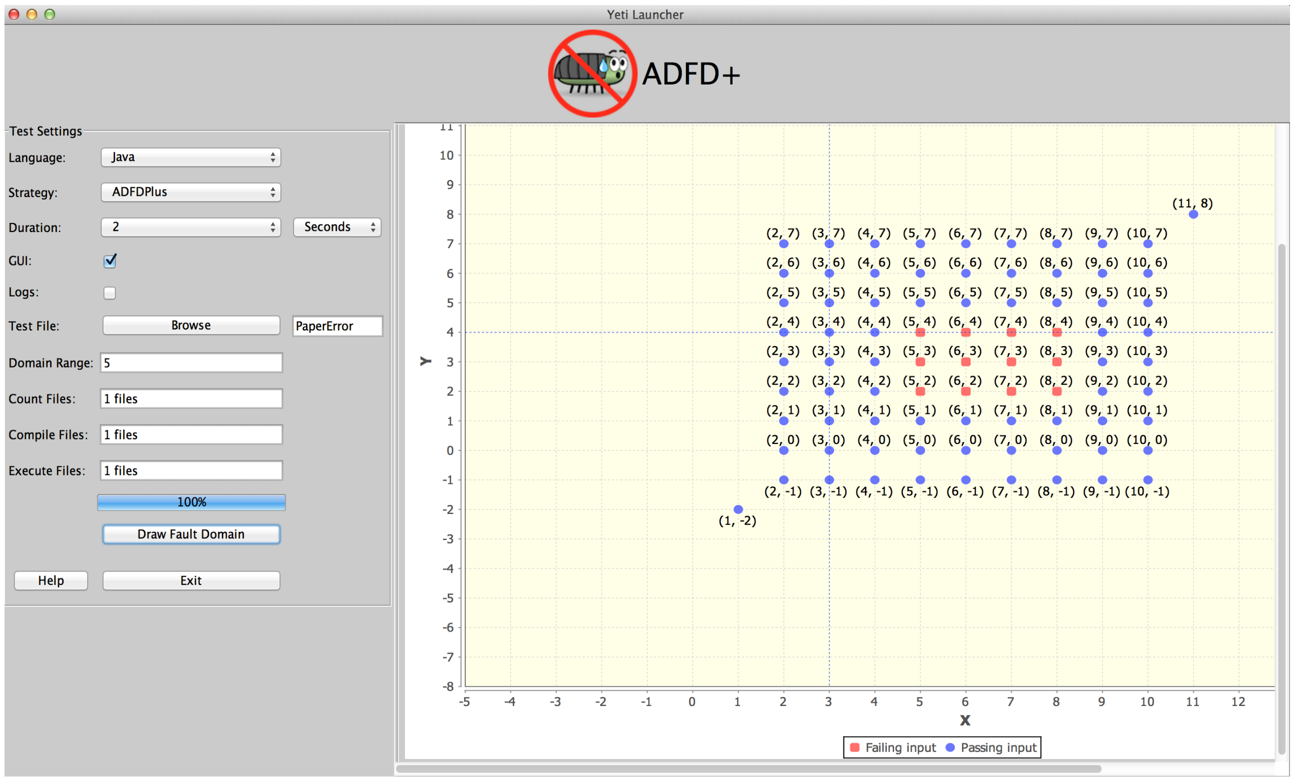
Suppose we have the following error-seeded class under test. From the program code, it can be easily noticed that an $ArithmeticException$ (DivisonByZero) failure is generated when the value of variable x ranges between 5 to 8 and the value of variable y between 2 to 4.

```java
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 $(5,2),(5,3),(5,4),(6,2),(6,4),(7,2),(7,4),(8,2),(8,3),(8,4)$.

IV. Randoop

Random tester for object oriented programs (Randoop) is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes a set of classes to test, time limit or number of tests to stop and and optionally a set of configuration files to assist testing as input. Randoop checks for are assertion violations, access violations and un-expected



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%

Fig. 3. The output of ADFD+ for the above code.

program termination in a given class. Its output a suite of JUnit for Java and NUnit for .Net program. 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 pre-defined pool in case of primitive types and as sequence of null values in case of reference type. It uses feedback mechanism to filter out duplicate test cases.

V. Research Questions

Beside the main question of finding the performance of ADFD+ with respect to time and number of test cases, the following research questions have also been addressed in this study:

1) **Efficiency:** How efficient is ADFD+, compared to Randoop, across different input domains?
2) **Effectiveness:** How effective is ADFD+, compared to Randoop, across different input domains?
3) **Failure-domains:** How the boundaries of a failure domains are presented by ADFD+ and Randoop?

VI. Evaluation

To evaluate the performance and efficiency of ADFD+, we followed the common practice [7], [10], [14] which is to compare it against another random testing tool. We used Randoop for comparison. We carried out testing of several error-seeded one and two dimensional numerical programs written in Java. Our subject programs were the same used in evaluation of ADFD [1]. The programs were divided in to two sets. Set A and B contains one and two-dimensional programs respectively. Each class was injected with at least one failure domain of point, block or strip nature. The values forming the failure domains are given in Table **??**. Every program was tested independently for 30 times by both ADFD+ and Randoop on the basis of time taken and number of tests used to find all the failures. The external parameters were kept constant in each test. 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], [9].

A. Experimental setup

All experiments were conducted with a 64-bit Mac OS X Mountain lion version 10.8.5 running on 2.7 GHz Intel Core i7 with 16 GB (1600 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35]. The ADFD+ Jar file is available at https://code.google.com/p/yeti-test/downloads/list/ and Randoop at https://randoop.googlecode.com/files/randoop.1.3.3.zip.

The following commands (1) and (2) were used to run the ADFD+ and Randoop. Both ADFD+ and Randoop were executed with default settings except that Randoop was provided with a seed value.

```
$ java -jar adfd_yeti.jar -------------(1)
```

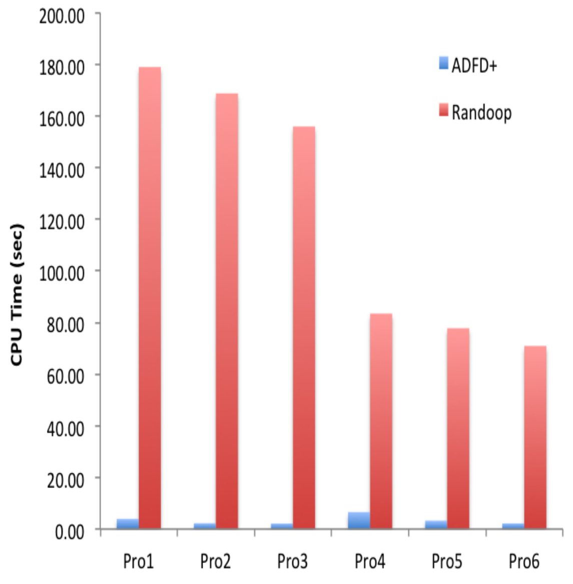


Fig. 4. Time taken to find failure domains

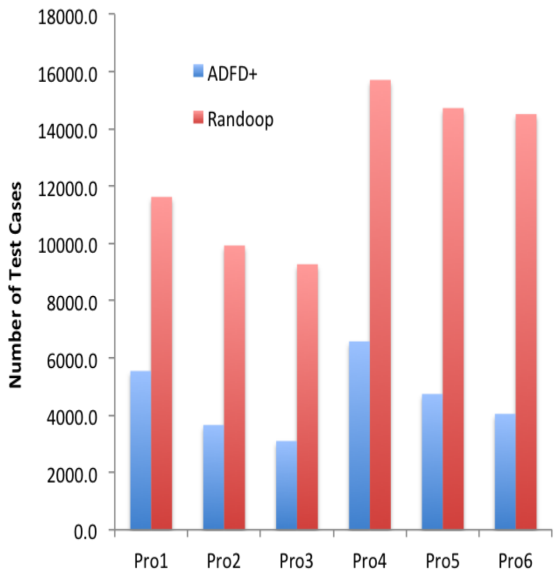


Fig. 5. Test cases taken to find failure domains

```
$ java randoop.main.Main gentests \
--testclass=OneDimPointFailDomain \
--testclass=Values --timelimit=100 ----(2)
```

VII. Experimental results

Results are split in to two sections depicting efficiency and effectiveness of the two tools.

A. *Efficiency Evaluation*

Figure 4 shows the efficiency measurement of the ADFD+ and Randoop against the selected programs. The x-axis represents one and two-dimensional programs with point, block and strip failure domain. The y-axis represents average time taken by both the tools to detect the whole failure domains. ADFD+ outperformed Randoop in the time taken to discover all forms of failure domains. Overall, this answers **RQ1** positively. The experimental analysis shows that there is a sufficient gain in efficiency when ADFD+ is used. The amount of gain in efficiency is up to two orders of magnitude.

It should be noted that the part of the gain may also be due to the fast processing of the underlying tool YETI, which is capable of executing 10^6 test calls per minute on Java code. Therefore, to find the performance of only ADFD+ we performed the second set of experiments to measure effectiveness.

B. *Effectiveness Evaluation*

Figure 5 shows the effectiveness measurement by ADFD+ and Randoop to find the whole failure domains. The x-axis represents one and two-dimensional programs with point, block and strip failure domain. The y-axis represents average number of test cases used by both the tools to detect the whole failure domains. The ADFD+ uses sufficiently lower number of test cases as compared to Randoop to find the whole failure domains. The amount of gain in performance is up to 50% or more. This provide a positive answer for **RQ2**.

C. *Failure Domains*

Both the tools found all the failure domains in specific time or using specific number of test cases. The ADFD+ also provide the added benefit of presenting the failure domains in graphical form as shown in Figure 6 and 7. The red and blue circle represents the failing and passing values respectively. The user can also enable or disable the option of showing the failing values on the graph. In Randoop there is no graphical representation or textual option to show failure domains separately. This provide a positive answer for **RQ3**.

VIII. 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 pictorial representation of failure domain helps in easily identifying the underlying domain and its boundaries, which can be helpful to developers in debugging.

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.

IX. 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.

X. 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.

XI. 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.

XII. Acknowledgments

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.

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] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 84–94, New York, NY, USA, 2007. ACM.

[5] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 157–166. IEEE, 2008.

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

[7] C. Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality*, pages 242–256. Springer, 2005.

[8] M. Oriol. York extensible testing infrastructure, 2011.

[9] 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.

[10] C. Pacheco and M. D. Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

[11] 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.

[12] 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.

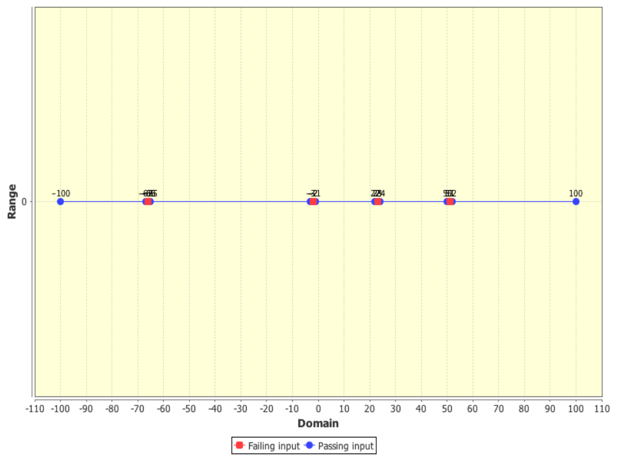
[13] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.

[14] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

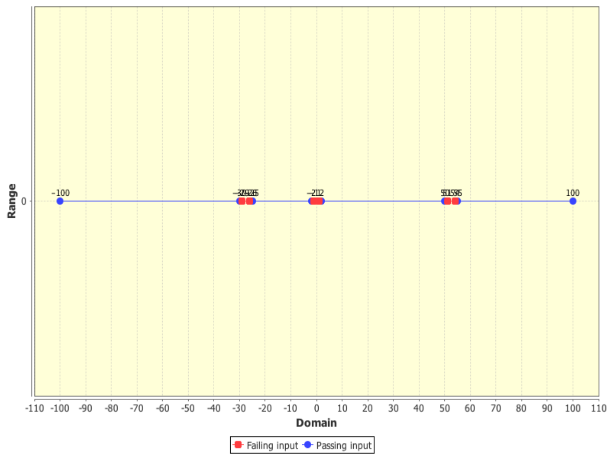
Mian Asbat Ahmad is a PhD scholar at the Department of Computer Science, the University of York, UK. He completed his M(IT) and MS(CS) from Agric. University Peshawar, Pakistan in 2004 and 2009 respectively. His research interests include new automated random software testing strategies.

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.

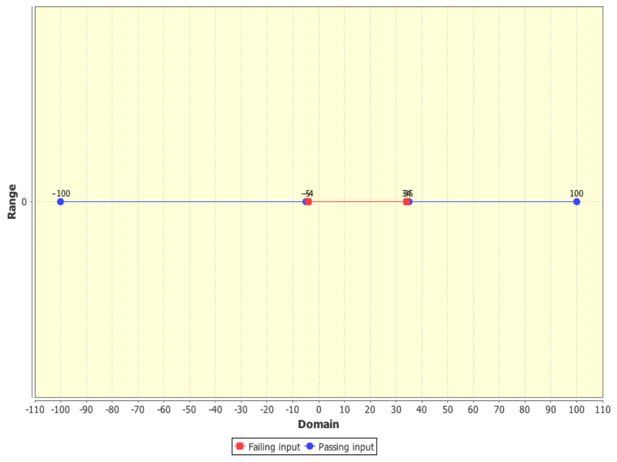
Appendix



(a) Point failure domain in one-dimension

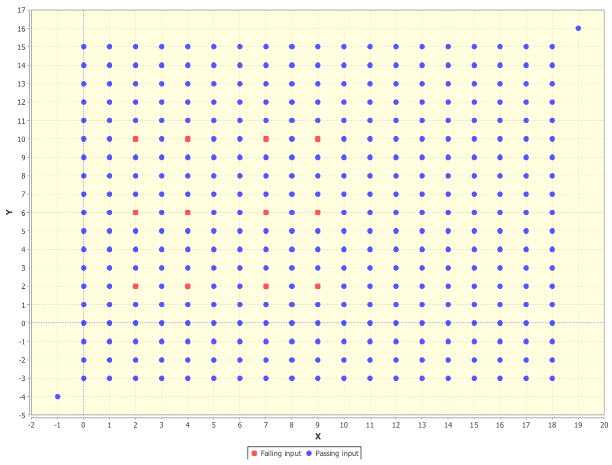


(b) Block failure domain in one-dimension

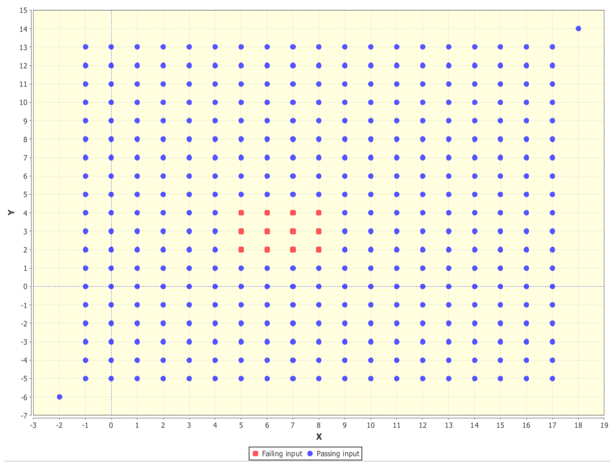


(c) Strip failure domain in one dimension

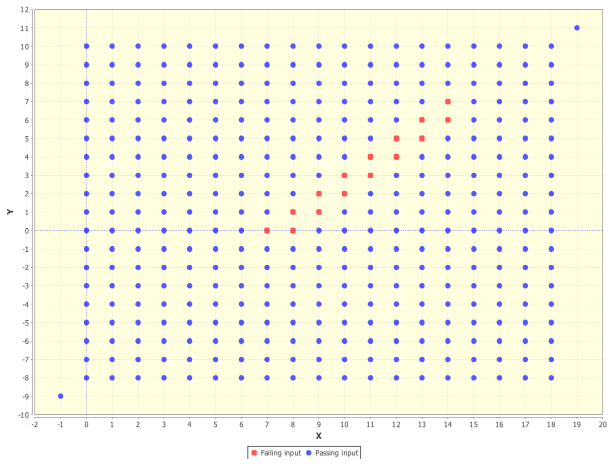
Fig. 6. Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs



(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Fig. 7. Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs