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 upgraded version of ADFD technique with respect to algorithm and graphical representation of failure domains. The new algorithm searches for the failure-domain around the failure in a given radius as against ADFD which limits the search between lower and upper bounds. The ADFD+ graphical output is further improved by providing labelled graphs to make it easily understandable and user friendly. To find the effectiveness of ADFD+, it was compared with Daikon using error seeded programs. The ADFD+ correctly pointed out all the seeded failure domains while Daikon identified individual failures but was unable to discover the failure domains. Evaluating the effectiveness and efficiency of the newly developed ADFD+ technique, its performance is compared with that of a mature random testing tool Randoop. The comparative results obtained are reported.**

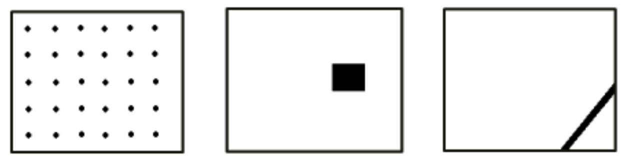
Index Terms—**software testing, automated random testing, ADFD.**

I. Introduction

Software testing is most widely used for verification and validation process. Efforts have been continuously made by researchers to make the testing process more and more effective and efficient. Testing is effective when it finds maximum number of faults in minimum number of test cases and it is efficient when maximum number of test cases are executed in minimum possible time. During up-gradation and development of testing techniques, focus is always on increasing the effectiveness by improving the algorithm and the efficiency by introducing partial or complete automation of the testing process.

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 domains. In the point domain the failure revealing inputs stand-alone and are evenly spread through out the input domain. In block domain the failure revealing inputs are contiguously clustered in one area. In strip domain the failure revealing inputs are clustered in one long elongated area. Figure 1 shows failure domains of the three types for two-dimensional program.

The paper describes ADFD+, an improved 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 the results on a graphical chart. Evaluating the effectiveness and efficiency of the newly developed ADFD+ technique, its performance is compared with that of a mature random testing tool Randoop [11]. The comparative results obtained are reported.



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

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

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).

This paper is organized as follows: Section II describes the ADFD+, improvement over ADFD, workflow, implementation and illustrate the working of ADFD+ with the help of an example. Section III describes the random testing tool Randoop, Section V explains the experimental setup. Section VI reveals results of the experiments. Section VII discusses the results. Section **??** presents related work, Section IX concludes the study and finally future work.

II. Automated Discovery of Failure Domain+

ADFD+ is an improved form of ADFD strategy developed previously by Ahmad and Oriol [1]. The technique automatically finds failures, failure domains and present the results in graphical form. In this technique, the test execution is initiated by random+ and continues till the first failure is found in the SUT. The technique then copies the values leading to the failure and the surrounding values to the dynamic list of interesting values. The resultant list provides relevant test data for the remaining test session and the generated test cases are more targeted towards finding new failures around the existing failures in the given SUT.

The improvements made in ADFD+ in comparison with 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 failure and failure domains.
- ADFD+ takes a single value as range within which the strategy searches for a failure domain whereas ADFD

takes two values as lower and upper bounds representing x and y-axis respectively.

- In ADFD+, the algorithm of dynamically generating Java file at run-time has been made simplified and efficient as compared to ADFD.
- In ADFD+, the failure domains generated in the output graph present a clear view of pass and fail domains with individually labelled points of failures as against a less clear view of pass and fail domains and lack of individually labelled points in ADFD.

A. *Workflow of ADFD+*

ADFD+ is a completely automatic technique requiring the user to specify program and domain range followed by clicking the $DrawFaultDomain$ button to execute testing. As soon as the button is clicked, YETI comes in to play with ADFD+ strategy to search for failures in the program under test. On finding a failure, the ADFD+ strategy creates a Java file which contains calls to the program on the failing and the surrounding values within the specified range. The Java file is executed after compilation and the results obtained are analysed to separate pass and fail values which are accordingly stored in the text files. 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.

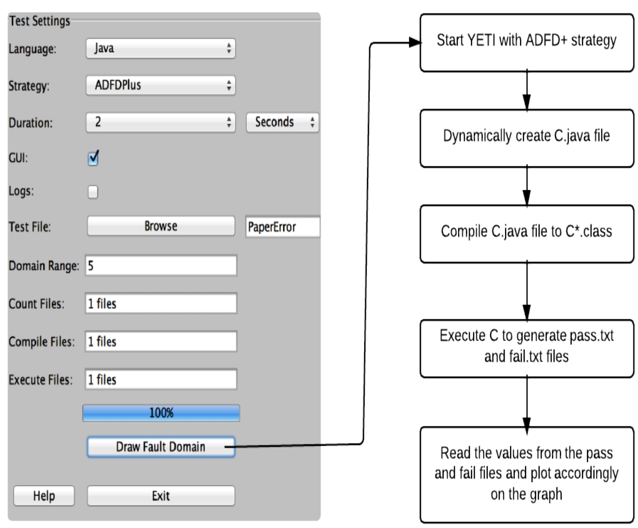


Fig. 2. Workflow of ADFD+

B. *Implementation of ADFD+*

The ADFD+ technique is implemented in YETI which 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 for automatic testing of programs 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 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 specific needs.

C. *Example to illustrate working of ADFD+*

Suppose we have the following error-seeded class under test. It is evident from the program code that an $ArithmeticException$ (divison by zero) 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;
        }
    }
}
```

At the beginning of the test, ADFD+ strategy evaluates the given 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 set to 5, ADFD+ evaluates a 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 $(5, 2), (5, 3), (5, 4), (6, 2), (6, 4), (7, 2), (7, 4), (8, 2), (8, 3), (8, 4)$.

III. Randoop

Random tester for object oriented programs (Randoop) is a fully automatic tool, capable of testing Java classes and .Net binaries. It takes as input a set of classes, time limit or number of tests and optionally a set of configuration files to assist testing. Randoop checks for assertion violations, access violations and un-expected program termination in a given class. Its output is 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 public methods from the class under test. 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.

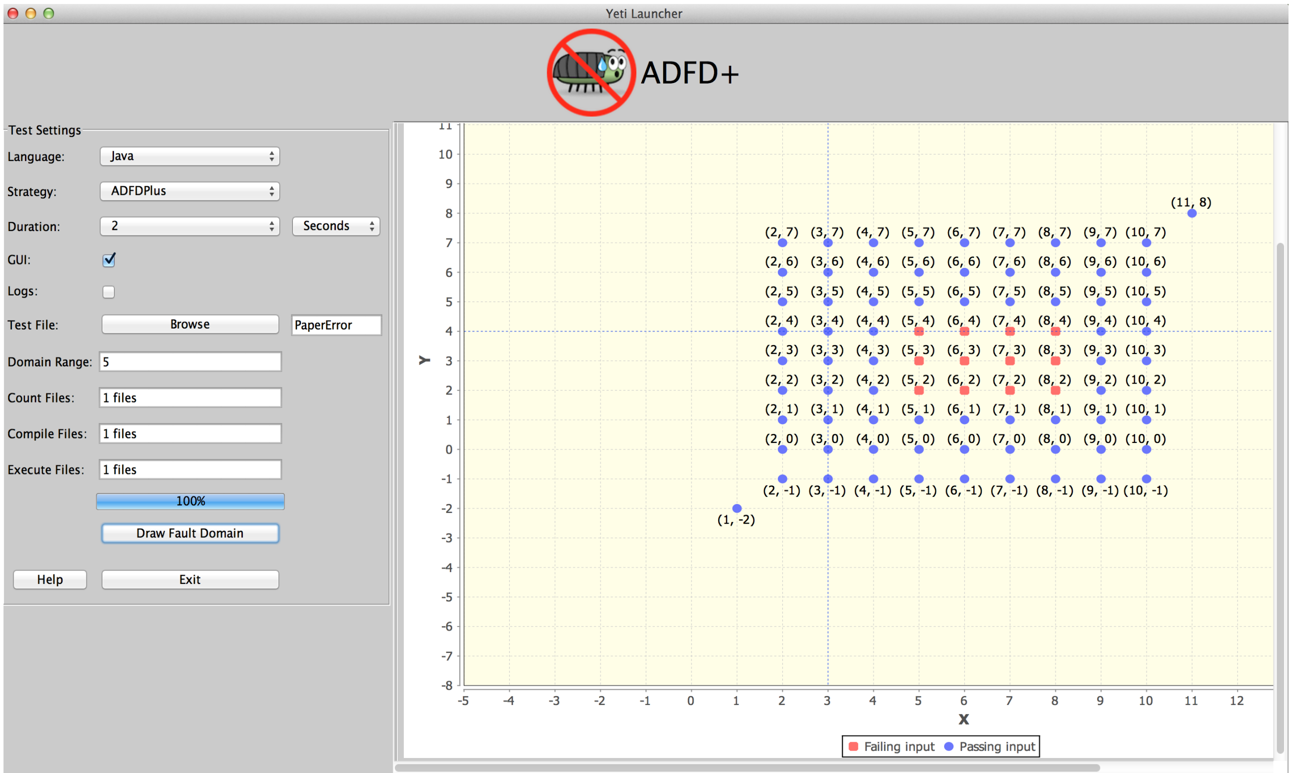


Fig. 3. The output of ADFD+ for the above code.

Randoop uses feedback mechanism to filter out duplicate test cases.

IV. Research Questions

The following research questions have been addressed in the study for evaluating ADFD+ strategy with respect to efficiency, effectiveness and presentation of failure domains:

1) How efficient is ADFD+ as compared to Randoop?
2) How effective is ADFD+ as compared to Randoop?
3) How failure domains are presented by ADFD+ as compared to Randoop?

V. Evaluation

For evaluating the effectiveness and efficiency, we compared ADFD+ with Randoop, following the common practice of comparison of the new tool with a mature random testing tool [7], [10], [14]. Testing of several error-seeded one and two dimensional numerical programs was carried out as per program code already published in the article [1]. The programs were divided in to set A and B containing 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 Randoop. Time taken and number of tests executed to find all failures were used as criteria for efficiency and effectiveness respectively. 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™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 two commands were used to run the ADFD+ and Randoop respectively. Both tools were executed with default settings, however, Randoop was provided with a seed value as well.

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

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

VI. Experimental Results

A. Efficiency

Figure 4 shows the efficiency measurement of ADFD+ and Randoop. The $x-axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y-axis$ represents average time taken by the tools

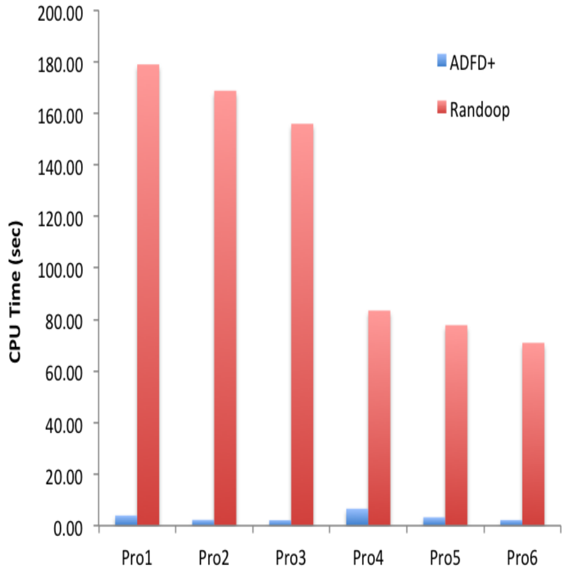


Fig. 4. Time taken to find failure domains

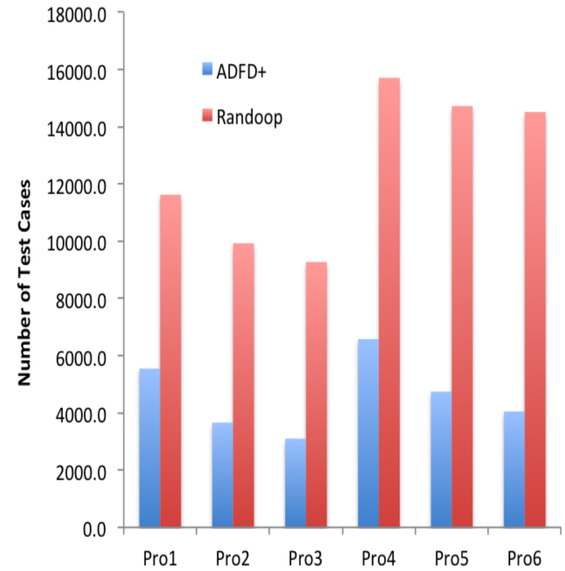


Fig. 5. Test cases taken to find failure domains

to detect the whole failure domains. ADFD+ outperformed Randoop in the time taken to discover all forms of failure domains. The Figure 4 shows extra ordinary efficiency in case of ADFD+, amounting up to two orders of magnitude, which may be partially attributed to the very fast processing of YETI integrated with ADFD+. YETI is capable of executing 10^6 test calls per minute on Java code.

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**.

VII. 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.

VIII. 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.

IX. 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.

X. 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.

XI. 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.

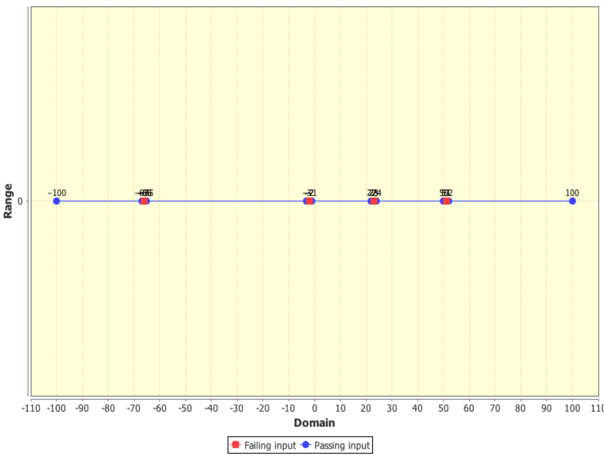
[13] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.

[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.

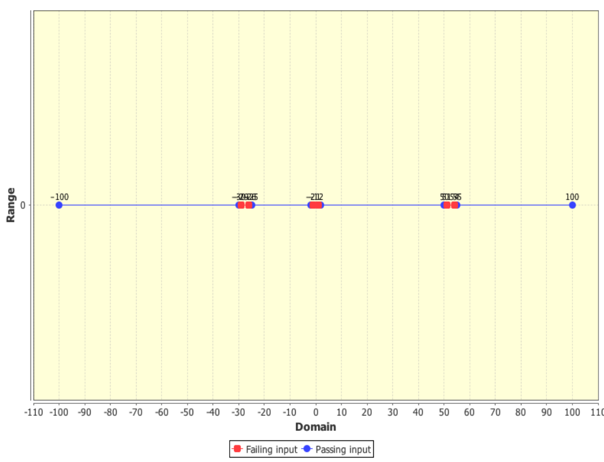
Mian Asbat Ahmad is a PhD scholar at the Department of Computer Science, the University of York, UK. He completed his M(IT) and MS(CS) from Agric. University Peshawar, Pakistan in 2004 and 2009 respectively. His research interests include new automated random software testing strategies.

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.

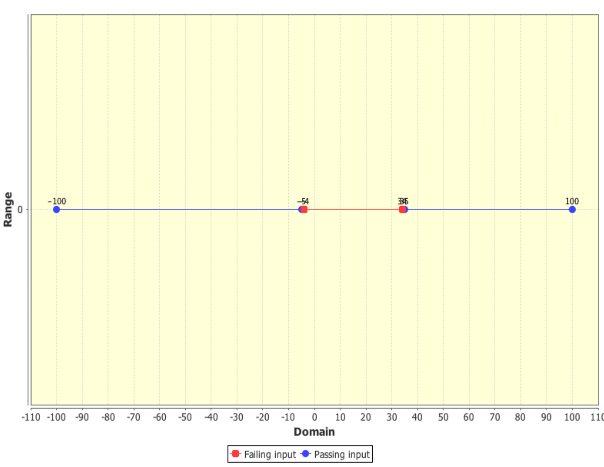
Appendix



(a) Point failure domain in one-dimension

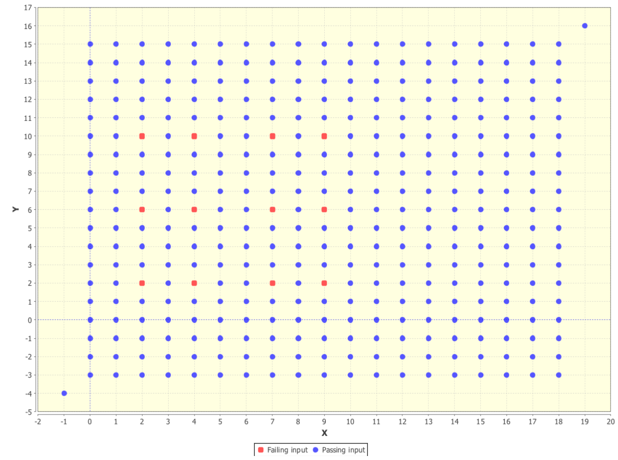


(b) Block failure domain in one-dimension

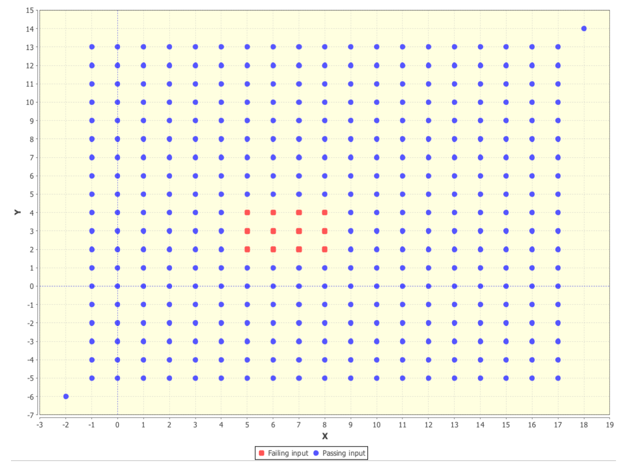


(c) Strip failure domain in one dimension

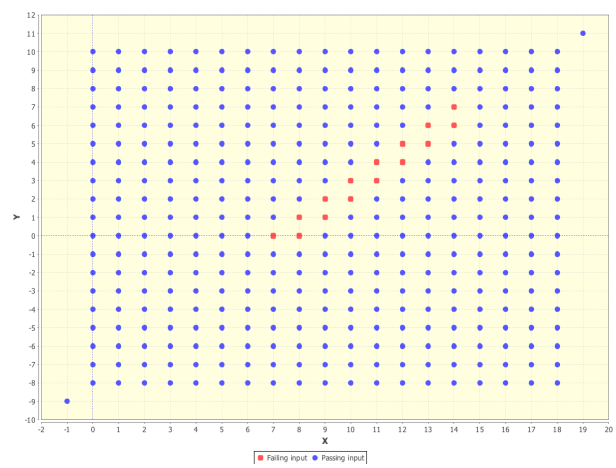
Fig. 6. Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs



(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