ADFD+: An Automatic Testing Technique 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 presentation of failure domains. The new algorithm used in ADFD+ searches for failure domain around the failure in a given radius as against ADFD which limits the search between lower and upper bounds. This results in consumption of lower number of test cases for detecting failure domain. The output has been improved in ADFD+ to provide labelled graphs for depicting the results in easily understandable user friendly form. ADFD+ is compared with Randoop to find the comparative performance of the two techniques. The results indicate that ADFD+ is a promising technique for finding failure and failure domain efficiently and effectively. In comparison with Randoop, its efficiency is evident by taking two orders of magnitude less time and its effectiveness is shown by taking 50% or less number of test cases to discover failure domains. ADFD+ has the added advantage of presenting the output in graphical form showing point, block and strip domains visually as against Randoop which lacks graphical user interface.**

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 efficient when maximum number of test cases are executed in minimum possible time and it is effective when it finds maximum number of faults in minimum number of test cases. During up-gradation and development of testing techniques, focus is always on increasing the efficiency by introducing partial or complete automation of the testing process and the effectiveness by improving the algorithm.

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [1], [2]. According to Chan et al. [3] 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 strip. Figure 1 shows failure domains of the three types for two-dimensional program.

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).

To target failures and evaluate the failure domains we developed earlier ADFD technique [4]. The ADFD+, an improved version of ADFD, is a fully automatic technique which finds failures and failure domains within a specified radius and presents the results on a graphical chart. The efficiency and effectiveness of ADFD+ technique is evaluated by comparing its performance with that of a mature testing tool Random tester for object oriented programs (Randoop) [5]. The results generated by ADFD+ and Randoop for the error-seeded programs shows better performance of ADFD+ with respect to time and number of test cases to find failure domains. Additionally ADFD+ presents the results graphically showing identified point block and strip domains visually as against Randoop which lacks graphical user interface.

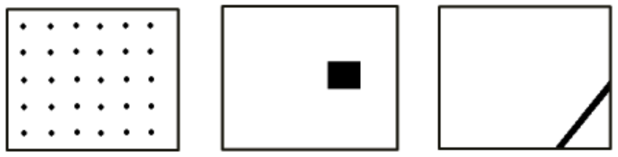

(a) Point domain (b) Block domain (c) Strip domain

Fig. 1. Failure domains across input domain [3]

The rest of the paper is organized as follows: Section II presents an overview of ADFD+ technique. Section III evaluates and compares ADFD+ technique with Randoop. Section IV reveals results of the experiments. Section V discusses the results. Section VI presents the threats to validity. Section VII presents related work. Finally, Section VIII concludes the study.

II. Automated Discovery of Failure Domain+

It is an improved version of ADFD technique developed earlier by Ahmad and Oriol [4]. 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 effectively targeted towards finding new failures around the existing failures in the given SUT.
The improvements made in ADFD+ over ADFD technique 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 for the radius 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. This results in consumption of lower number of test cases for detecting failure domain.
- In ADFD+, the algorithm of dynamically generating Java file at run-time has been made simplified and efficient as compared to ADFD resulting in reduced overhead.
- In ADFD+, the point, block and strip 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 fully automatic technique requiring the user to select radius value and feed the program under test followed by clicking the $DrawFaultDomain$ button for test execution. 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 strategy creates a Java file which contains calls to the program on the failing and surrounding values within the specified radius. 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.

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 for easily incorporating new strategies and languages making YETI a favourable choice for implementing ADFD+ strategy. YETI is also capable of generating test cases to reproduce the failures found during the test session.

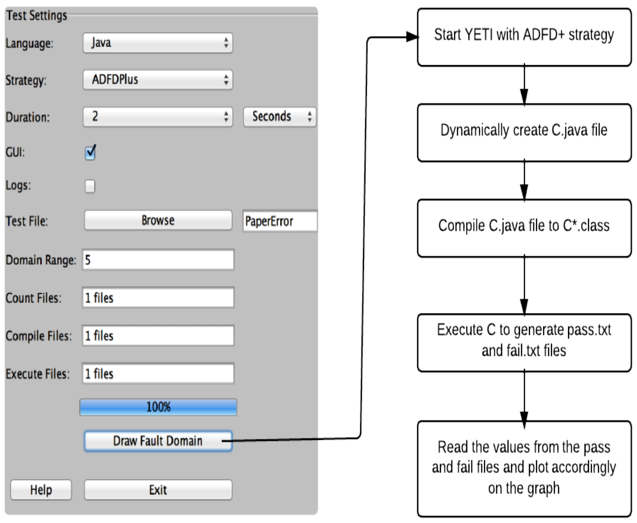

Test Settings
Language: Java
Strategy: ADFDPlus
Duration: 2 Seconds
GUI: ✓
Logs:
Test File: Browse PaperError
Domain Range: 5
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files
100%
Draw Fault Domain
Help
Exit
Start YETI with ADFD+ strategy
Dynamically create C.java file
Compile C.java 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+

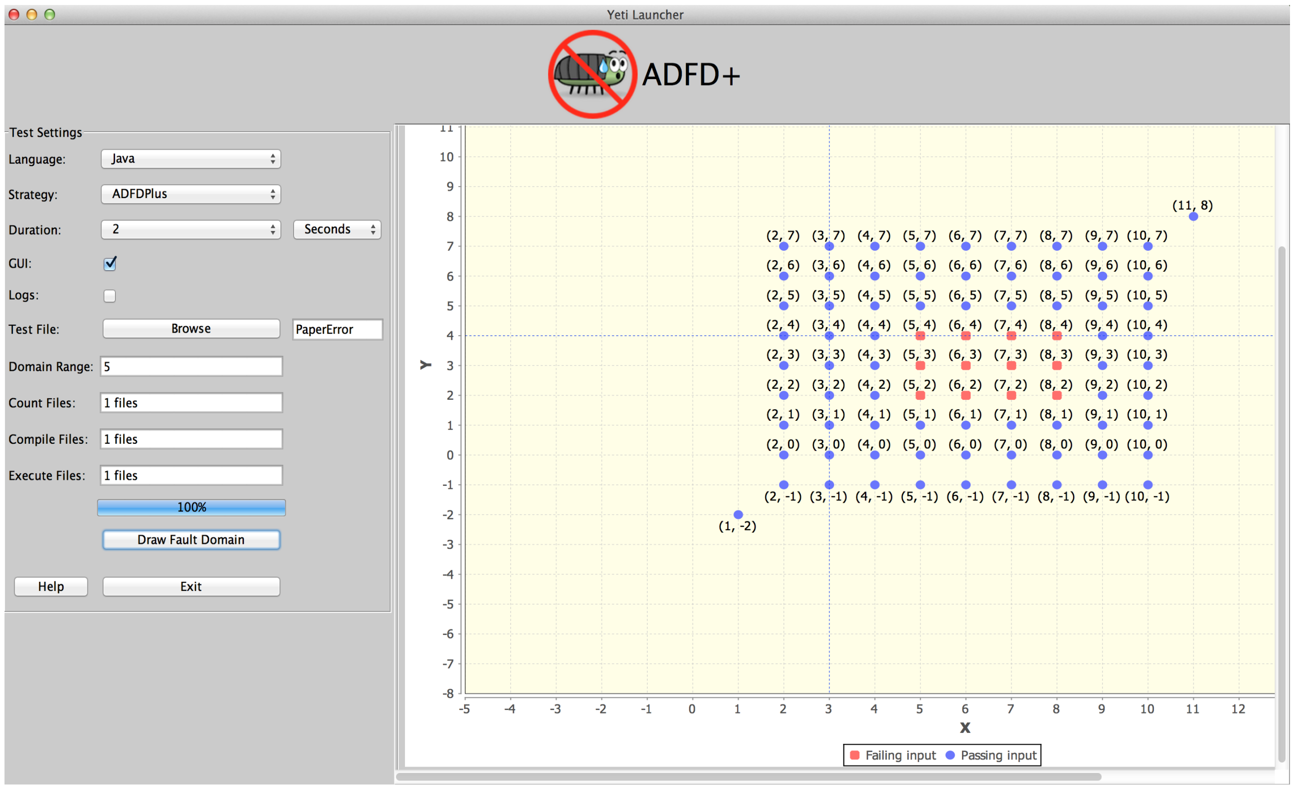
The strategies section in YETI contains different strategies including random, random+, DSSR and ADFD for selection according to specific needs. ADFD+ strategy is implemented in this section by extending the $YetiADFDStrategy$.

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.

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



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.

III. Evaluation

For evaluating the efficiency and effectiveness, we compared ADFD+ with Randoop, following the common practice of comparison of the new tool with a mature random testing tool [6], [7], [8]. Testing of several error-seeded one and two dimensional numerical programs was carried out as per program code [4]. 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. The failure causing values are given in Table I. Every program was tested independently for 30 times by both ADFD+ and Randoop. Time taken and number of tests executed to find all failure domains 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 [4], [9].

A. Research questions

The following research questions have been addressed in the study for evaluating ADFD+ technique 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?

B. 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. Randoop uses feedback mechanism to filter out duplicate test cases.

C. 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 two commands were used to run the ADFD+ and Randoop respectively. Both tools were executed with

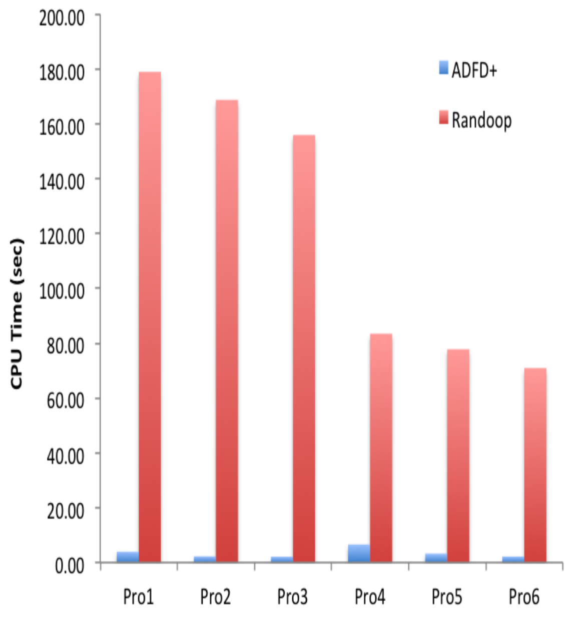


Fig. 4. Time taken to find failure domains

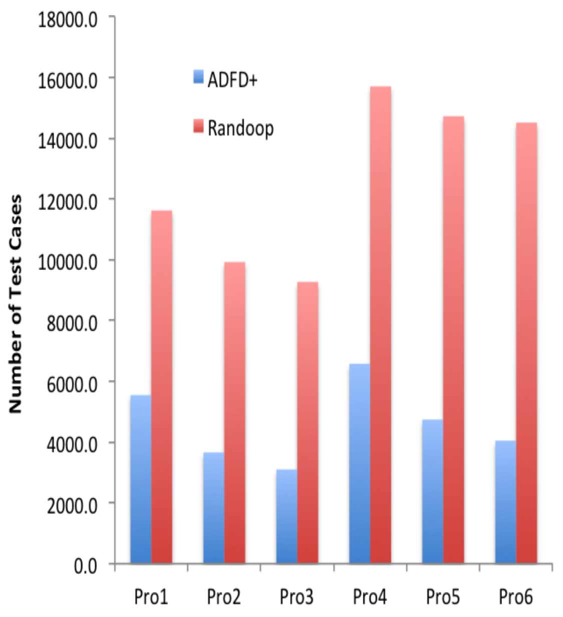


Fig. 5. Test cases taken to find failure domains

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)
```

IV. Experimental results

A. *Efficiency*

Figure 4 shows the comparative efficiency 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 to detect the failure domains. As shown in the figure ADFD+ showed extra ordinary efficiency by taking two orders of magnitude less time to discover failure domains as compared to Randoop.

This 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. To counter the contribution of YETI and assess the performance of ADFD+ by itself, the effectiveness of ADFD+ was compared with Randoop in terms of the number of test cases required to identify the failure domains without giving any consideration to the time consumed for completing the test session. The results are presented in the following section.

B. *Effectiveness*

Figure 5 shows the comparative effectiveness 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 number of test cases used by the tools to detect the failure domains. The figure shows higher effectiveness in case of ADFD+, amounting to 100% or more. The higher effectiveness of ADFD+ may be attributed to its working mechanism in comparison with Randoop for identifying failures. ADFD+ dynamically changes its algorithm to exhaustive testing in a specified radius around the failure as against Randoop which uses the same random algorithm for searching failures.

C. *Failure Domains*

The comparative results of the two tools with respect to presentation of the identified failure domains reveal better performance of ADFD+ by providing the benefit of presenting the failure domains in graphical form as shown in Figure 6 and 7. The user can also enable or disable the option of showing the failing values on the graph. In comparison Randoop lacks the ability of graphical presentation and the option of showing the failure domains separately and provides the results scattered across the textual files.

V. Discussion

The results indicated that ADFD+ is a promising technique for finding failure and failure domain efficiently and effectively. It has the added advantage of showing the results in graphical form. The pictorial representation of failure domains facilitates the debuggers to easily identify the underlying failure domain and its boundaries for troubleshooting.

In the initial set of experiments Randoop was executed for several minutes with default settings. The results indicated no identification of failures after several executions. On analysis of the generated unit tests and Randoop's manual, it was found that the pool of values stored in Randoop database for int primitive type contains only 5 values including -1, 0, 1, 10 and 100. To enable Randoop to select different values, we supplied a configuration file with the option to generate random values between -500 and 500 for the test cases as all the seeded errors were in this range.

As revealed in the results ADFD+ outperformed Randoop by taking two orders of magnitude less time to discover the failure domains. This was partially attributed to the very fast processing of YETI integrated with ADFD+. To counter the effect of YETI the comparative performance of ADFD+ and Randoop was determined in terms of the number of test cases required to identify the failure domains giving no consideration to the time taken for completing the test session. As shown in the results ADFD+ identified all failure domains in 50% or less number of test cases.

The ADFD+ was found quite efficient and effective in case of block and strip domains but not so in case of point domains where the failures lied away from each other as shown in the following code. This limitation of ADFD+ may be due to the search in vain for new failures in the neighbourhood of failures found requiring the additional test cases resulting in increased overhead.

```
public class Error {
  public static void Error (int x, int y){
    int z;
    if (x == 10000)
         {   z = 50/0;    }

    if (y == -2000)
         {   z = 50/0;    }
   }
}
```

The number of test cases to be undertaken in search of failures around the previous failure found is set in the range value by the user. The time taken by test session is directly proportional to the range value. Higher range value leads to larger graphical output requiring zoom feature which has been incorporated in ADFD+ for use when the need arise.

VI. Threats to validity

The study faces threats to external and internal validity. The external threats are common to most of the empirical evaluations. It includes the extent to which the programs under test the generation tools and the nature of seeded errors are representative of the true practice. The present findings will serve as foundation for future research studies needed to be undertaken with several types of classes, test generation tools and diversified nature of seeded errors in order to overcome the threats to external validity. The internal threats to validity includes error-seeded and limited number of classes used in the study. These may be avoided by taking real and higher number of classes in future studies.

VII. Related Work

The increase in complexity of programs poses new challenges to researchers for finding more efficient and effective ways of software testing with user friendly easy to understand test results. Adaptive Random Testing [10], Proportional random testing [3] and feedback directed random testing [11] are some of the prominent upgraded versions of random testing with better performance. Automated random testing is simple to implement and capable of finding hitherto bugs in complex programs [12], [13]. ADFD+ is a promising technique for finding failures and failure domains efficiently and effectively with the added advantage of presenting the output in graphical form showing point, block and strip domains.

Some previous research studies have reported work on Identification, classification and visualisation of pass and fail domains in the past [14], [15], [16]. This includes Xslice [14] is used to differentiate the execution slices of passing and failing part of a test in a visual form. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [15]. Hierarchical Multi Dimension Scaling (HMDS) describes a semi-automated procedure of classifying and plotting the faults [16]. A serious limitation of the above mentioned tools is that they are not fully automated and require human intervention during execution. Moreover these tools need the requirement of existing test cases to work on where as ADFD+ strategy generates test cases, discovers failures, identifies pass and fail domains and visualises the results in a graphical form operating in fully automated manner.

VIII. Conclusion

The newly developed ADFD+ technique is distinct from other random testing techniques because it not only identifies failures but also discovers failure domains and provides the result output in easily understandable graphical form. The paper highlights the improved features of ADFD+ in comparison with ADFD technique previously developed by our team [4]. The paper then analyses and compares the experimental results of ADFD+ and Randoop for the point, block and strip failure domains. The ADFD+ demonstrated extra ordinary efficiency by taking less time to the tune of two orders of magnitude to discover the failure domains and it also surpassed Randoop in terms of effectiveness by identifying the failure domains in 50% or less number of test cases. The better performance of ADFD+ may be attributed mainly to its ability to dynamically change algorithm to exhaustive testing in a specified radius around the first identified failure as against Randoop which uses the same random algorithm continuously for searching failures.

IX. Future Work

The ADFD+ strategy is capable of testing numerical programs and needs to be extended for testing of non numerical and reference data types to enable it to test all types of data. ADFD+ has the capability of graphical presentation of results for one and two-dimensional numerical programs. It is worthwhile to extend the technique to enable it to present the results of multi-dimensional numerical and non numerical programs in the graphical form.

Acknowledgments. The authors are thankful to the Department of Computer Science, University of York for physical and financial support. Thanks are also extended to Prof. Richard Paige for his valuable guidance, help and generous support.

References

[1] G. B. Finelli, "Nasa software failure characterization experiments," *Reliability Engineering & System Safety*, vol. 32, no. 1, pp. 155–169, 1991.

[2] 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*. ACM, 2007, pp. 90–93.

[3] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[4] M. A. Ahmad and M. Oriol, "Automated discovery of failure domain," *Lecture Notes on Software Engineering*, vol. 03, no. 1, pp. 289–294, 2013.

[5] 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*. ACM, 2007, pp. 815–816.

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

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

[8] 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*. Springer, 2005, pp. 365–381.

[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*, april 2012, pp. 201 –210.

[10] T. Y. Chen, "Adaptive random testing," *Eighth International Conference on Qualify Software*, vol. 0, p. 443, 2008.

[11] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[12] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[13] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *In 19th European Conference Object-Oriented Programming*, 2005, pp. 504–527.

[14] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. IEEE, 1995, pp. 143–151.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*. ACM, 2002, pp. 467–477.

[16] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

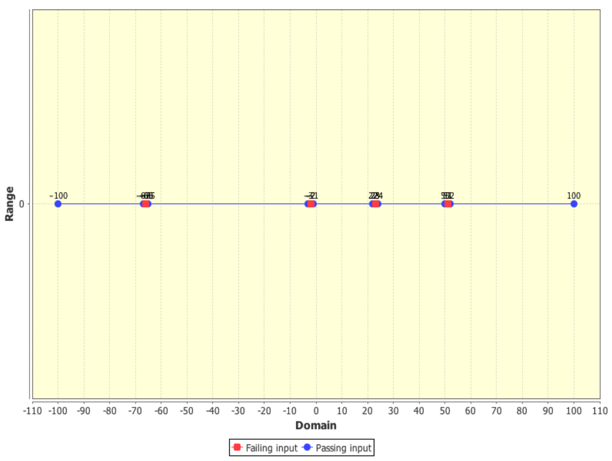
Mian Asbat Ahmad is a PhD scholar at the Department of Computer Science, the University of York, UK. He completed his M(IT) and MS(CS) from Agric. University Peshawar, Pakistan in 2004 and 2009 respectively. His research interests include new automated random software testing strategies.

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.

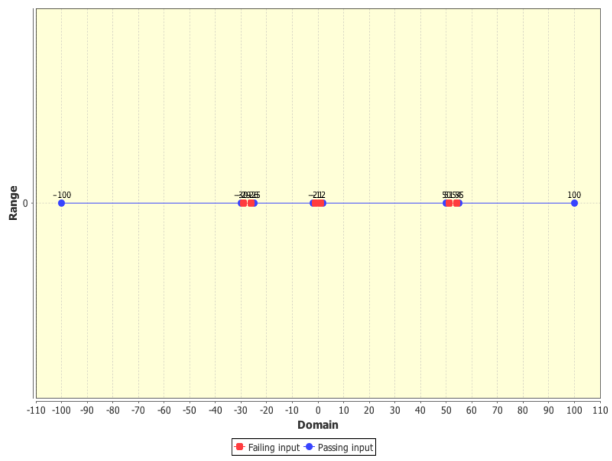
Appendix

TABLE I
TABLE DEPICTING VALUES OF X AND Y ARGUMENTS FORMING POINT, BLOCK AND STRIP FAILURE DOMAIN IN FIGURE 6(A), 6(B), 6(C) AND FIGURE 7(A), 7(B), 7(C) RESPECTIVELY

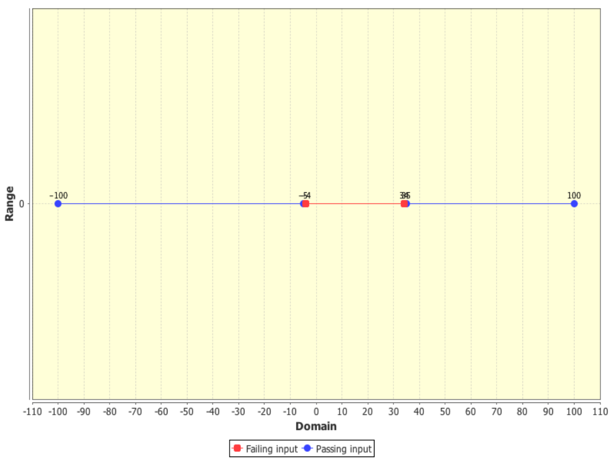
Dim	Point failure	Block failure	Strip failure
One	x = -66	x = -1, 0, 1	x = -4 – 34
	x = -2	x =-26 – -29	
	x= 51	x = 51 – 54	
	x= 23		
Two	x=2, y=10	x = 5, y = 2	x = 7, y = 0
	x=4, y=10	x = 6, y = 2	x = 8, y = 0
	x=7, y=10	x = 7, y = 2	x = 8, y = 1
	x=9, y=10	x = 8, y = 2	x = 9, y = 1
		x = 5, y = 3	x = 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



(a) Point failure domain in one-dimension

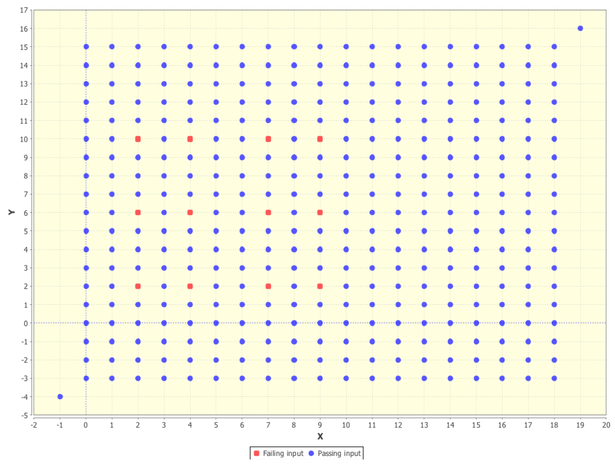


(b) Block failure domain in one-dimension

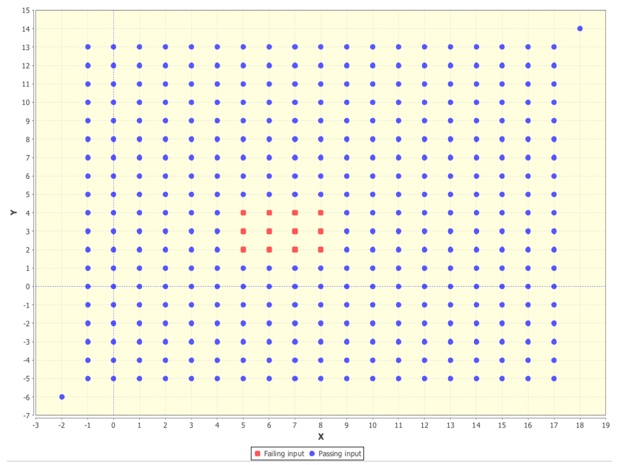


(c) Strip failure domain in one dimension

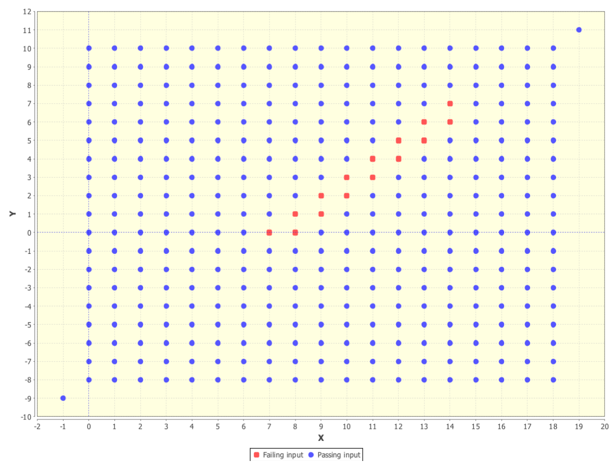
Fig. 6. Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs



(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