New Strategies for Automated Random Testing

Mian Asbat Ahmad

Enterprise Systems Research Group

Department of Computer Science

University of York, UK

March 2014

A thesis submitted for the degree of Doctor of Philosophy

Abstract

Software testing is the process of evaluating the quality of a software or its component.The thesis presents new techniques for improving the effectiveness of automated random testing, evaluates the efficiency of these techniques and proposes directions for future work.

The first technique, Dirt Spot Sweeping Random (DSSR) strategy is developed on the assumption that unique failures reside in contiguous block and strips. It starts by testing the code at random. When a failure is identified, the DSSR strategy selects the neighbouring input values for the subsequent tests. The selected values sweep around the identified failure leading to the discovery of new failures in the vicinity. This results in quick and efficient identification of new failures in SUT.

The second technique, Automated Discovery of Failure Domain (ADFD) is developed with the capability to find failure and the failure-domains in a given SUT and provides visualization of the identified pass and fail domains within a specified range in the form of a chart. The new technique is highly effective in testing and debugging and provides an easy to understand test report in the visualized form.

The third technique, Automated Discovery of Failure Domain+ (ADFD+) is an upgraded version of ADFD technique with respect to algorithm and graphical representation of failure domains. 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.

Contents

List of Figures

List of Tables

I feel it a great honour to dedicate my PhD thesis to my beloved parents for their significant contribution in achieving the goal of academic excellence.

Acknowledgements

The duration at the University of York for my PhD has been the most joyful and rewarding experience in my academic career. The institution provided me with everything I needed to thrive: challenging research problems, excellent company and supportive environment. I am deeply grateful to all those who shared this experience with me.

Several people have contributed to the completion of my PhD dissertation. The most prominent personality deserving due recognition is my worthy advisor, Dr. Manuel Oriol. Thank you Manuel for your endless help, valuable guidance, constant encouragement, precious advice, sincere and affectionate attitude.

I thank my assessor Prof. Dr. John Clark for his constructive feedback on various reports and presentations. I am highly indebted to Prof. Dr. Richard Paige for his generous help, cooperation and guidance throughout my research.

Thanks to my father Prof. Dr. Mushtaq A. Mian who provided a conducive environment, valuable guidance and crucial support at all levels of my educational career and to my beloved mother whose love, affection and prayers have been my most precious assets. I am also thankful to my brothers Dr. Ashfaq, Dr. Aftab, Dr. Ishaq, Dr. Afaq, and Dr. Ilyas who have been the source of inspiration for me to pursue higher studies. Last but not the least I am thankful to my dear wife Dr. Munazza Asbat for her company, help and cooperation throughout my stay at York.

I received Departmental Overseas Research Scholarship which is awarded to overseas students for higher studies on academic merit and research potential. I am truly grateful to the Department of Computer Science, University of York for financial support.

Chapter 1

ADFD+: An Automatic Testing Technique for Finding and Presenting Failure domains

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.

1.1 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.

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.

1.2 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.

1.2.1 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 1.2.

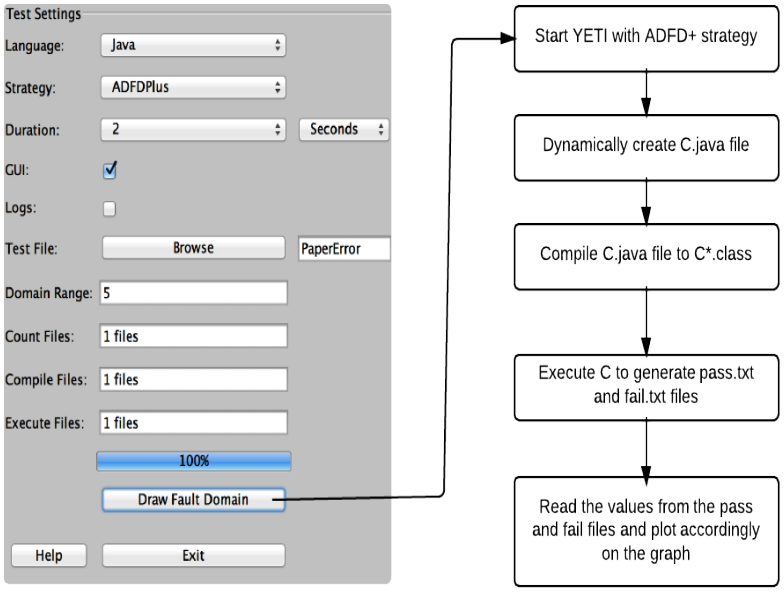


Figure 1.1: Workflow of ADFD+

1.2.2 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. The strategies section in YETI contains different strategies including random, ran-

dom+, DSSR and ADFD for selection according to specific needs. ADFD+ strategy is implemented in this section by extending the $YetiADFDStrategy$.

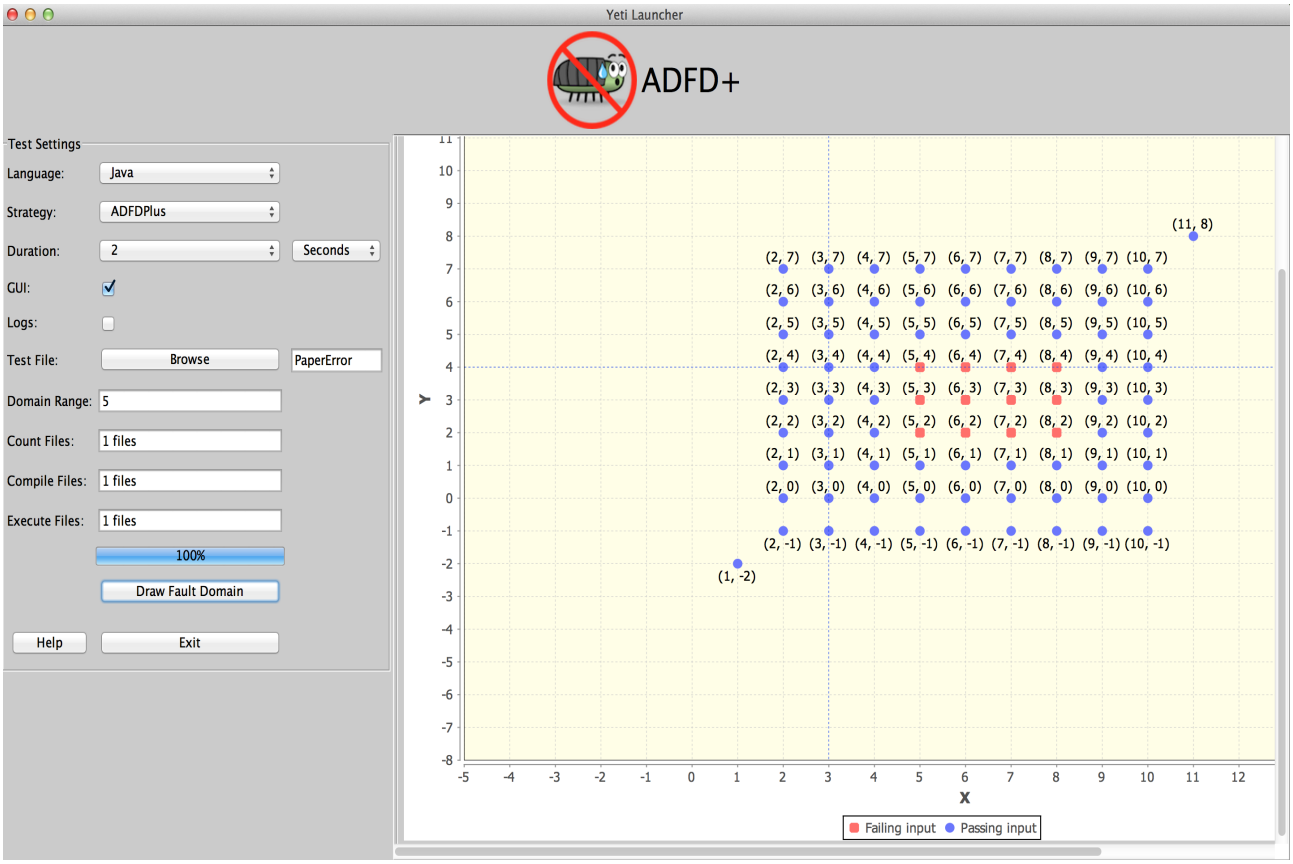


Figure 1.2: The output of ADFD+ for the above code.

1.2.3 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 **??** shows that the failure domain forms a block pattern and the boundaries of the failure are $(5, 2), (5, 3), (5, 4), (6, 2), (6, 4), (7, 2), (7, 4), (8, 2), (8, 3), (8, 4)$.

1.3 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, **?**]. 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 1.1. 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, **?**].

1.3.1 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?

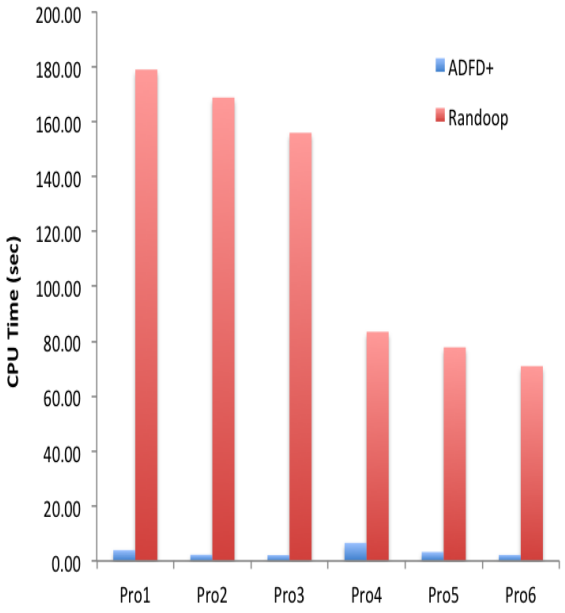


Figure 1.3: Time taken to find failure domains

1.3.2 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

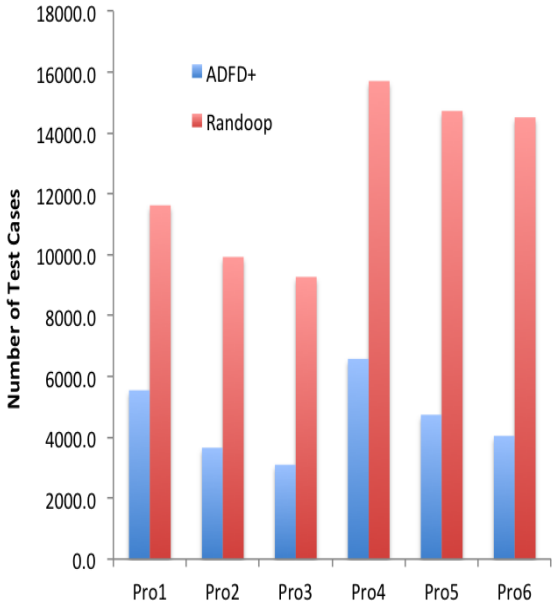


Figure 1.4: Test cases taken to find failure domains

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.

1.3.3 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)
```

1.4 Experimental results

1.4.1 Efficiency

Figure **??** 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.

1.4.2 Effectiveness

Figure **??** 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.

1.4.3 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 1.5 and 1.6. 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.

1.5 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.

1.6 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.

1.7 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 [**?**], Proportional random testing [3] and feedback directed random testing [**?**] 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 [**?**, **?**]. 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 [8, 9, 10]. This includes Xslice [8] 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 [9]. Hierarchical Multi Dimension Scaling (HMDS) describes a semi-automated procedure of classifying and plotting the faults [10]. 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.

1.8 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.

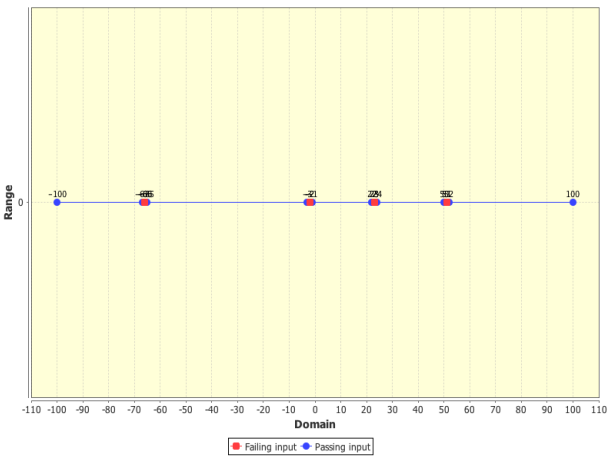
1.9 Future Work

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.

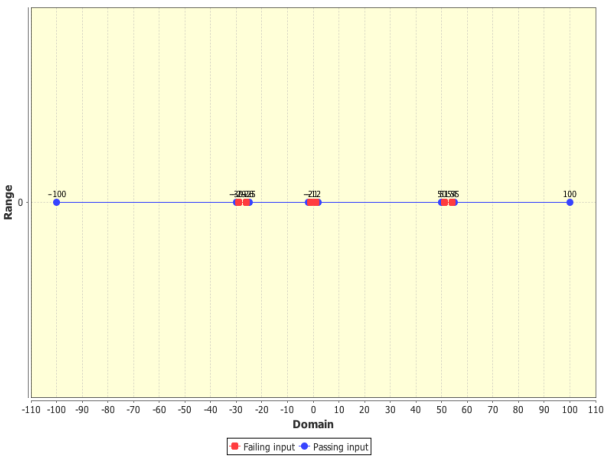
Appendix

Table 1.1: 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

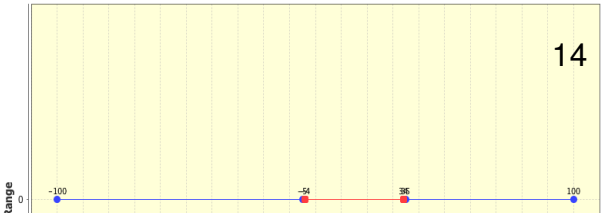
Dim	Point failure	Block failure	Strip failure
One	x = -66 x = -2 x= 51 x= 23	x = -1, 0, 1 x =-26 – -29 x = 51 – 54	x = -4 – 34
Two	x=2, y=10 x=4, y=10 x=7, y=10 x=9, y=10	x = 5, y = 2 x = 6, y = 2 x = 7, y = 2 x = 8, y = 2 x = 5, y = 3 x = 6, y = 3 x = 7, y = 3 x = 8, y = 3 x = 5, y = 4 x = 6, y = 4 x = 7, y = 4 x = 8, y = 4	x = 7, y = 0 x = 8, y = 0 x = 8, y = 1 x = 9, y = 1 x = 9, y = 2 x = 10, y = 2 x = 10, y = 3 x = 11, y = 3 x = 11, y = 4 x = 12, y = 4 x = 12, y = 5 x = 13, y = 6 x = 14, y = 6 x = 14, y = 7

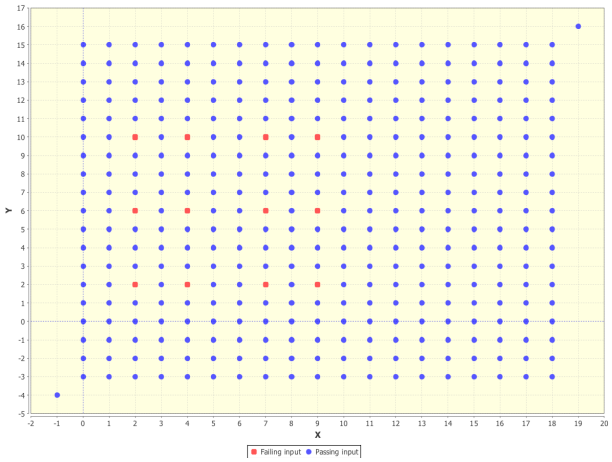


(a) Point failure domain in one-dimension

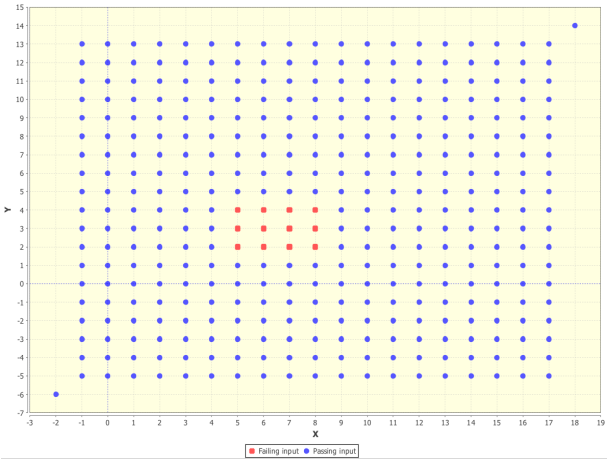


(b) Block failure domain in one-dimension

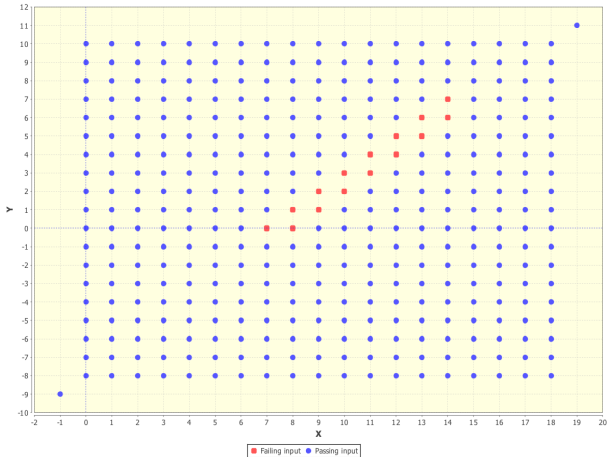




(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Figure 1.6: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

Chapter 2

Conclusions

The research study aimed at understanding the nature of failures in software, discovering how to leverage failure domain for finding more bugs and developing new improved automated random test strategies to achieve the desired objectives. The existing random test strategies find individual failures and do not focus on failure domain. The knowledge of failure along with failure domain is of great benefit to debuggers for quick and effective removal of failures.

Various aspects of failure-domains with respect to automated random testing were explored. The study focused on three main issues. To minimize the number of test cases required to discover a failure-domain was the first, to identify the pass and fail input domains and generate the result in graphical form was the second, to compare the developed strategy with the standard automated tool Daikon for finding the failure domain was the third focus of the study.
It was revealed in the study that the input inducing failures residue in contagious locations forming certain geometrical shapes in the input domain. These shapes can be divided in to point, block and strip domains. A set of techniques and tools have been developed for improving the effectiveness of automated random testing in finding failures and failure-domains.

The first technique, Dirt Spot Sweeping Random (DSSR) strategy is developed which starts by testing the program at random. When a failure is identified, the strategy selects the neighbouring input values for the subsequent tests. The selected values sweep around the identified failure leading to the discovery of new failures in the vicinity. This results in quick and efficient identification of failures in the software under test. The results stated in Chapter **??** showed that signifi-

cantly better performance of DSSR strategy as compared to random and random+ strategies.

The second technique, Automated Discovery of Failure Domain (ADFD) was developed with the capability to find failure and failure-domains in a given software and provides visualization of the identified pass and fail domains within a specified range in the form of a chart. The technique starts with random+ strategy to find the first failure. When a failure is identified, a new Java program is dynamically created at run-time which is then compiled and executed to search for failure-domains. The output of the program showing pass and fail domains is represented in the graphical form. The results stated in Chapter **??** showed that ADFD technique correctly identify the failure domains. The technique is highly effective in testing and debugging by providing an easy to understand test report in the visualized form.

The third technique, Automated Discovery of Failure Domain+ (ADFD+) is 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 was 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.

2.1 Lessons Learned

Research in the field of software testing has been in progress for more than three decades but only a handful of free and open source fully automated testing tools are available for software testing. The current study is in continuation of the research efforts to find improved testing techniques capable to identify failures and failure domains quickly, efficiently and effectively. In the following section, the lessons learned during the study are presented in the summarized form which may be of interest to the researchers pursuing future research.

Selection of performance measurement criteria

Among the three measuring techniques used for finding the effectiveness of random testing, E-measure and P-measure have been criticised [11] while F-measure has been often used by researchers [12, 13]. In our experiments, F-measure was initially used but its weakness was soon realised as stated in Section **??**. The F-measure is effective in traditional testing which counts the number of test cases used to find the first failure and the system is then handed over to developers for fixing the identified failure. Presently automated testing tools test the whole system and report all discovered failures in one go, thus F-measure is not the favourable choice. We addressed the issue by measuring the maximum number of failures detected in a particular number of test calls as the criterion for finding the effectiveness of the test strategy.

Test results in random testing keep on changing

In random testing, due to the random generation of test input, the results keep on changing even if all the test parameters and the program under test remain the same. Therefore the efficiency of one technique in comparison with the other becomes difficult. We addressed the issue by taking five steps. 1) Each experiment was repeated 30 times and the average was taken for comparison 2) In each experiment 10000 test cases were executed to minimize the random effect 3) Sufficiently large number of representative test samples (60) were taken for evaluation 4) Error seeded programs with known locations of faults were used to verify the results 5) The experimental results were statistically analysed to estimate the difference on statistical basis.

Higher computation increases overhead and decreases performance

Attempts to evolve new versions of random testing with higher fault finding ability usually result in increased computation, higher overhead and lower performance. We addressed the issue by developing new strategies which uses neighbouring values around the failure finding value for the subsequent tests. This approach saves the computation involved in generating suitable test values from the whole input domain.

Random testing coupled with exhaustive testing

Random testing generates test values from the input domain at random. Exhaustive testing although very effective yet it is not feasible for the whole input domain. The issue was addressed in our experiments by coupling the random testing with exhaustive testing. In our newly developed strategies the testing starts at random till a failure is identified and switches to exhaustive testing to select the values around the failure finding value in the specified range set by the tester. This results in quick identification of the neighbouring failures which may be difficult to find by using random strategy.

Easy to understand user friendly test output

Random testing is no exception when it comes to the complexity of understanding and evaluating test results. No random strategy seems to provide graphical representation of the failures and failure-domains. The issue of getting in easy to understand user friendly format has been addressed the present study. The ADFD strategy has been developed with the feature of giving the result output in the visualized graphical form. This feature has been further improved in the ADFD+ strategy which clarify and label individual failures and the failure domains in two-dimensional graph. The identification of failures and failure domains in graphical form helps debuggers keep in view all the occurrences while fixing the fault.

Auto-generation of primitive and user-defined data types

We noticed that auto-generation of user defined data type is more complex as compared to the primitive data type. We addressed the issue by creating objects of the classes under test and randomly calling the methods with random inputs in accordance with the parameter's space. The inputs were divided into primitive and user defined data types. For primitive data generation `Math.random()` method is used and for generation of user-defined data object of the class is created at run time as stated in Section. The approach adopted helps in achieving a fully automated testing system.

Chapter 3

Future Work

This chapter presents the scope and potential of future work as an extension of the present research study. The topics suggested include: use of contracts and assertions to discover failure; introducing object distance in DSSR strategy to enhance its testing ability; measuring code coverage of the three newly developed strategies to assess additional aspect of performance; extension of ADFD+ strategy for testing non numerical data; enhancing the plotting ability of ADFD+ strategy to more than two-dimensional charts; introducing additional features in the user interface of ADFD+; extension of ADFD+ to apply it to the real world scenario; research on the prevalence of point block and strip failure domains; improvement in the DSSR strategy to reduce overhead.

Improvement in the DSSR strategy to reduce overhead

The DSSR strategy is an extension of random+ strategy based on the assumption that failure domains are contagious. The dirt spot sweeping feature of the strategy adds the neighbouring value of the failure finding value to the list of interesting values to cover the failure domain. This add 5% overhead to DSSR strategy compared to R+ strategy. In future studies the algorithm may be modified to decrease the overhead and make the strategy more effective.

Use of contracts and assertions to discover failure

The common practice to use undefined run-time exceptions of the programming language as test oracles in the absence of contracts and assertions was followed

in the study. It is worthwhile to study the fault-detection ability of an automated strategy in the presence of contracts and assertions. To generate explicit oracles a tool like Daikon may be integrated in the system for achieving the automatic generation of invariants and their annotation in to source code.

Introducing object distance in DSSR strategy to enhance its testing ability

The newly developed DSSR strategy add the neighbouring values for primitive type data and Strings. It has a limitation that no neighbouring values are added when the failure is found by a reference type data. It is suggested for future research work to extend the DSSR strategy by incorporating a suitable technique like Artoo to find the neighbouring objects for including these in the list of interesting values.

Measuring code coverage of the three newly developed strategies to assess additional aspect of performance

In spite of the fact that the strategies developed in the study generate more test cases from the surrounding area where a failure is discovered for better coverage, it is worthwhile to measure the code coverage achieved by the new strategies to ensure the effectiveness. The instrumentation technique may be applied to the software under test to achieve the desired objective.

Extension of ADFD+ strategy for testing non numerical data

The ADFD+ strategy tests numerical programs and can be used for testing the software of numerical data types in the real world context. The strategy may be extended to include testing of non numerical and reference data types to enable the strategy to test all types of data.

Enhancing the plotting ability of ADFD+ strategy to more than two-dimensional charts

The newly developed ADFD+ has the capability of graphical representation of results for one and two-dimensional numerical programs. It is worthwhile to extend the strategy so as to be capable of graphical representation of results for multi-dimensional numerical and non numerical programs.

Introducing additional features in the user interface of ADFD+

The user interface of ADFD+ provides a fully automated mechanism of testing the program, processing the results and visually representing the results in graphical form. The user interface may be extended in future to give choice to the tester for real time interaction, manual addition of test cases, showing thumbnail view of previous graphs and 3D support to present multi-dimensional arguments.

Extension of ADFD+ to apply it to the real world scenario

The newly developed ADFD+ strategy uses error-seeded programs for assessment of accuracy and effectiveness. This may likely expose it to external validity threat. Future studies may be undertaken in the real world scenario by including the feature of testing non numerical and reference data types so that there is no more threat to validity.

Research on the prevalence of point block and strip failure domains

In accordance with the reported literature, the three newly developed strategies used the concept of failure laying in point block and strip failure domains. It is worthwhile to undertake study for determining the prevalence and proportionate distribution of the failure domains in the input domain. This will improve the testing efficiency by giving due focus to the more prevalent types of failure domain.

Appendix A

A.1 Sample code to identify failure domains

Program 1 Program generated by ADFD on finding fault in SUT

```
/**
 * Dynamically generated code by ADFD strategy
 * after a fault is found in the SUT.
 * @author (Mian and Manuel)
 */
import java.io.*;
import java.util.*;

public class C0
{
    public static ArrayList<Integer> pass = new ArrayList<Integer>();
    public static ArrayList<Integer> fail = new ArrayList<Integer>();
    public static boolean startedByFailing = false;
    public static boolean isCurrentlyFailing = false;
    public static int start = -80;
    public static int stop = 80;

    public static void main(String []argv){
        checkStartAndStopValue(start);
        for (int i=start+1;i<stop;i++){
            try{
                PointDomainOneArgument.pointErrors(i);
                if (isCurrentlyFailing)
                {
                    fail.add(i-1);
                    fail.add(0);
                    pass.add(i);
                    pass.add(0);
                    isCurrentlyFailing=false;
                }
            }
            catch(Throwable t) {
                if (!isCurrentlyFailing)
                {
```

```
                pass.add(i-1);
                pass.add(0);
                fail.add(i);
                fail.add(0);
                isCurrentlyFailing = true;
            }
        }
    }
    checkStartAndStopValue(stop);
    printRangeFail();
    printRangePass();
}

public static void printRangeFail() {
    try {
        File fw = new File("Fail.txt");
        if (fw.exists() == false) {
            fw.createNewFile();
        }
        PrintWriter pw = new PrintWriter(new FileWriter (fw, true));
        for (Integer i1 : fail) {
            pw.append(i1+"\n");
        }
        pw.close();
    }
    catch(Exception e) {
        System.err.println(" Error : e.getMessage() ");
    }
}
public static void printRangePass() {
    try {
        File fw1 = new File("Pass.txt");
        if (fw1.exists() == false) {
            fw1.createNewFile();
        }
        PrintWriter pw1 = new PrintWriter(new FileWriter (fw1, true));
        for (Integer i2 : pass) {
            pw1.append(i2+"\n");
        }
        pw1.close();
    }
    catch(Exception e) {
        System.err.println(" Error : e.getMessage() ");
    }
}
public static void checkStartAndStopValue(int i) {
    try {
        PointDomainOneArgument.pointErrors(i);
        pass.add(i);
        pass.add(0);
    }
    catch (Throwable t) {
        startedByFailing = true;
        isCurrentlyFailing = true;
        fail.add(i);
```

```
            fail.add(0);
        }
    }
}
```

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;
        }
```

```
        }

    }
```

References

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

[2] Christoph Schneckenburger and Johannes 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.

[3] FT Chan, Tsong Yueh Chen, IK Mak, and Yuen-Tak Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

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

[5] Carlos Pacheco and Michael 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.

[6] Catherine 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.

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

[8] Hiraral Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[9] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[10] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE, 2003.

[11] Tsong Yueh Chen, Hing Leung, and IK Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making*, pages 320–329. Springer, 2005.

[12] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of the f-measure. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 146–153. IEEE, 2004.

[13] Tsong Yueh Chen and Yuen Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109–119, 1996.