# Automated Discovery of Failure Domain

Mian Asbat Ahmad and Manuel Oriol

***Abstract*—There are several automated random strategies of software testing based on the presence of point, block and strip failure domains inside the whole input domain. As yet no particular, fully automated test strategy has been developed for the discovery and evaluation of the failure domains. We therefore have developed Automated Discovery of Failure Domain (ADFD), a new random test strategy that finds the failures and their domains in a given system under test. It further provides visualization of the identified pass and fail domains. In this paper we describe ADFD strategy, its implementation in York Extensible Testing Infrastructure (YETI) and illustrate its working with the help of an example. We report on experiments in which we tested error-seeded one and two-dimensional numerical programs. Our experimental results show that for each program, ADFD strategy successfully performs identification of failures, failure domains and their representation on graphical chart.**

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

## I. Introduction

Testing is fundamental requirement to assess the quality of any software. Manual testing is laborious and error-prone; therefore emphasis is to use automated testing that significantly reduces the cost of software development process and its maintenance [1]. Most of the modern black-box testing techniques execute the SUT with specific input and compare the obtained results against the test oracle. A report is generated at the end of each test session containing any discovered faults and the input values which triggers the faults. Debuggers fix the discovered faults in the SUT with the help of these reports. The revised version of the system is given back to the testers to find more faults and this process continues till the desired level of quality, set in the test-plan, is achieved.

The fact that exhaustive testing for any non-trivial program is impossible, compels the testers to come up with some strategy of input selection from the whole input domain. Pure random is one of the possible strategies widely used in automated tools. It is intuitively simple and easy to implement [2], [3]. It involves minimum or no overhead in input selection and lacks human bias [4], [5]. While pure random testing has many benefits, there are some limitations as well, including low code coverage [6] and discovery of lower number of faults [7]. To overcome these limitations while keeping its benefits intact many researchers successfully refined pure random testing. Adaptive Random Testing (ART) is the most significant refinements of random testing. Experiments performed using ART showed up to 50% better results compared to the traditional/pure random testing [8]. Similarly Restricted Random Testing (RRT) [9], Mirror Adaptive Random Testing (MART) [10], Adaptive Random Testing for Object Oriented Programs (ARTOO) [2], Directed Adaptive Random Testing (DART) [11], Lattice-based Adaptive Random Testing (LART) [12] and Feedback-directed Random Testing (FRT) [13] are some of the variations of random testing aiming to increase the overall performance of pure random testing.

All the above-mentioned variations in random testing are based on the observation of Chan et al. [14] that failure causing inputs across the whole input domain form certain kinds of domains. They classified these domains into point, block and strip failure domain. In Fig. 1 the square box represents the whole input domain. The black points, block and strip area inside the box represent the faulty values whereas white area inside the box represent legitimate values for a specific system. They further suggested that the failure-finding ability of testing could be improved by taking into consideration these failure domains.

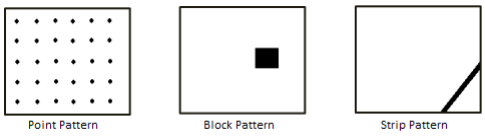



Figure. 1. Failure domains across input domain [4]

It is interesting that where many random strategies are based on the principle of contiguous failure domains inside the input domain, no specific strategy is developed to evaluate these failure domains. This paper describes a new test strategy called Automated Discovery of Failure Domain (ADFD), which not only finds the pass and fail input values but also finds their domains. The idea of identification of pass and fail domain is attractive as it provides an insight of the domains in the given SUT. Some important aspects of ADFD strategy presented in the paper include:

- Implementation of the new ADFD strategy in York Extensible Testing Infrastructure tool.
- Evaluation to assess ADFD strategy by testing classes with different failure domains.
- Decrease in overall test duration by identification of all the failure domains instead of a single instance of failure.
- Increase in test efficiency by helping debugger to keep in view all the failure occurrences when debugging.

The rest of this paper is organized as follows:
Section 2 describes the ADFD strategy. Section 3 presents implementation of the ADFD strategy. Section 4 explains the experimental results. Section 5 discusses the results. Section 6 presents the threats to validity. Section 7 presents related work and Section 8, concludes the study.

Manuscript received March 9, 2013; revised April 11, 2013.

Mian Asbat Ahmad is with the Department of Computer Science, University of York, YO10 5DD, UK (e-mail: mian.ahmad@york.ac.uk).

Manuel Oriol is with the Department of Computer Science, University of York, York, YO10 5DD, UK. (e-mail: manuel.oriol@york.ac.uk) and Industrial Software Systems, ABB Corporate Research, CH-5405 Baden-Daettwil, Switzerland.

## II. AUTOMATED DISCOVERY OF FAILURE DOMAIN

Automated Discovery of Failure Domain strategy is proposed as improvement on R+ strategy with capability of finding failures as well as the failure domains. The output produced at the end of test session is a chart showing the passing value or range of values in blue and failing value or range of values in red. The complete workflow of ADFD strategy is given in Fig. 2.

The process is divided into five major steps given below and each step is briefly explained in the following paras.

1. GUI front-end for providing input
2. Automated finding of failure
3. Automated generation of modules
4. Automated compilation and execution of modules to discover domains
5. Automated generation of graph showing domains

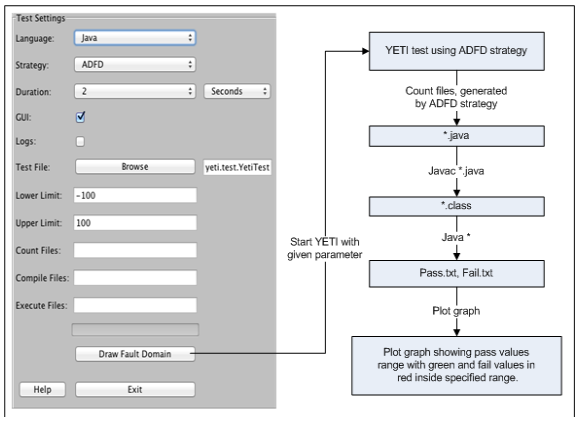



Figure. 2. Workflow of ADFD strategy

### A. GUI front-end for providing input

ADFD strategy is provided with an easy to use GUI front-end to get input from the user. It takes YETI specific input including language of the program, strategy, duration, enable or disable YETI GUI, logs and a program to test in the form of java byte code. In addition it also takes minimum and maximum values to search for failure domain in the specified range. Default range for minimum and maximum is Integer.MIN_INT and Integer.MAX_INT respectively.

### B. Automated finding of failure:

To find the failure domain for a specific failure, the first requirement is to identify that failure in the system. ADFD strategy extends R+ strategy and relies on R+ strategy to find the first failure. Random+ (R+) is an improvement over random strategy with preference to the boundary values to provide better fault-finding ability. ADFD strategy is implemented in YETI tool, which is famous for its simplicity, high speed, and proven ability of finding potentially hazardous faults in many systems [15], [16]. YETI is quick and can call up to one million instructions in one second on Java code. It is also capable of testing VB.Net, C, JML and CoFoJa beside Java programs.

### C. Automated generation of modules:

After a failure is found in the SUT, ADFD strategy generate complete new Java program to search for failure domains in the given SUT. These programs with ".java" extensions are generated through dynamic compiler API included in Java 6 under javax.tools package. The number of programs generated can be one or more, depending on the number of arguments in the test module i.e., for module with one argument one program is generated, for two argument two programs and so on. To track failure domain the program keeps one or more than one argument constant and only one argument variable in the generated program.

### D. Automated compilation and execution of modules to discover domains:

The java modules generated in previous step are compiled using "javac *" command to get their binary ".class" files. The "java *" command is applied to execute the compiled programs. During execution the constant arguments of the module remain the same but the variable argument receive all the values in range, from minimum to maximum, specified in the beginning of the test. After execution is completed we get two text files of "Pass.txt" and "Fail.txt". Pass file contains all the values for which the modules behave correctly while fail file contains all the values for which the modules fail.

### E. Automated generation of graph showing domains:

The values from the Pass and Fail files are used to plot (x, y) chart-using JFreeChart. JFreeChart is a free open-source java library that helps developers to display complex charts and graphs in their applications [17]. Blue color lines with circle represent pass values whereas red color line with squares and diamonds represents the fail values. Resultant graph clearly depicts both the pass and fail domain across the specified input domain. The graph shows red points in case the program fails for only one value, blocks when the program fails for multiple values and strips when a program fails for a long range of values.

## III. IMPLEMENTATION

The ADFD strategy is implemented in a tool called York Extensible Testing Infrastructure (YETI). YETI is available in open-source at *http://code.google.com/p/yeti-test/*. In this section a brief overview of YETI is given with the focus on the parts relevant to the implementation of ADFD strategy. For verification of ADFD strategy in YETI, a program is used as an example to illustrate the working of ADFD strategy. Please refer to [15], [16], [18], [19] for more details on YETI tool.

### A. York Extensible Testing Infrastructure

YETI is a testing tool developed in Java that test programs using random strategies in an automated fashion. YETI meta-model is language-agnostic which enables it to test programs written in functional, procedural and object-oriented languages.

YETI consists of three main parts including core infrastructure for extendibility through specialization, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both the languages and strategies sections have a pluggable architecture to easily incorporate new strategies and languages making YETI a favorable choice to implement ADFD strategy. YETI is also capable of generating test cases to reproduce the faults found during the test session.

### B. ADFD strategy in YETI

The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. On top of the hierarchy in strategies, is an abstract class YetiStrategy, which is extended by YetiRandomPlusStrategy and is further, extended to get ADFD strategy.

### C. Example

For a concrete example to show how ADFD strategy in YETI proceeds, we suppose YETI tests the following class with ADFD strategy selected for testing. Note that for more clear visibility of the output graph, generated by ADFD strategy at the end of test session, we fix the values of lower and upper range by 70 from Integer.MIN_INT and Integer.MAX_INT.

```
/**
*  Point Failure Domain example for one argument
*  @author (Mian and Manuel)
*/
   public class PointDomainOneArgument{
      public static void pointErrors (int x){
         if (x == -66) abort ();
         if (x == -2) abort ();
         if (x == 51) abort ();
         if (x == 23) abort ();
      }
   }
```

As soon as any one of the above four faults are discovered the ADFD strategy generate a dynamic program.

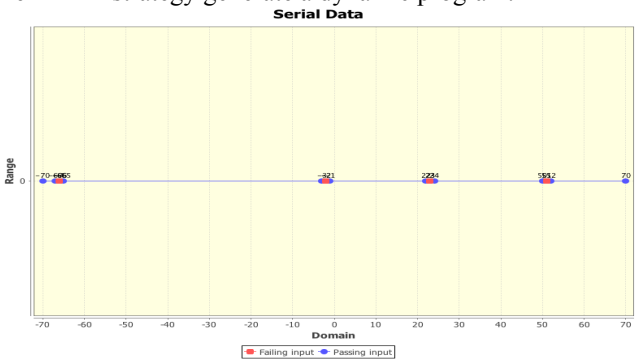



Figure 3. ADFD strategy plotting pass and fail domain of the above class

This program is automatically compiled to get binary file and then executed to find the pass and fail domains inside the specified range. The identified domains are plotted on two-dimensional graph. It is evident from the output presented in Fig. 3 that ADFD strategy not only finds all the faults but also the pass and fail domains.

## IV. Experimental Results

This section includes the experimental setup and results obtained after using ADFD strategy. Six numerical programs of one and two-dimension were selected. These programs were error-seeded in such a way to get all the three forms of failure domains including point, block and strip failure domains. Each selected program contained various combinations of one or more failure domains.

All experiments were performed on a 64-bit Mac OS X Lion Version 10.7.5 running on 2 x 2.66 GHz 6-Core Intel Xeon with 6.00 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java™ SE Runtime Env. [version 1.6.0 35].

To elucidate the results, six programs were developed so as to have separate program for one and two-dimension point, block and strip failure domains. The code of selected programs is given in Appendix (1-6). The experimental results are presented in Table 1 and described under the following three headings.

**Point Failure Domain:** Two separate Java programs Program1 and Program2 given in Appendix (1, 2) were tested with ADFD strategy in YETI to get the findings for point failure domain in one and two-dimension program. Figure 4(a) present range of pass and fail values for point failure domain in one-dimension whereas Figure 4(b) present range of pass and fail values for point failure domain in two-dimension program. The ranges of pass and fail values for each program in point failure domain is given in (Table 1, Serial No. 1).

**Block Failure Domain:** Two separate Java programs Program3 and Program4 given in Appendix (3, 4) were tested with ADFD strategy in YETI to get the findings for block failure domain in one and two-dimension program. Figure 5(a) present range of pass and fail values for block failure domain in one-dimension whereas Figure 5(b) present range of pass and fail values for block failure domain in two-dimension program. The range of pass and fail values for each program in block failure domain are given in (Table 1, Serial No. 2).

**Strip Failure Domain:** Two separate Java programs Program5 and Program6 given in Appendix (5, 6) were tested with ADFD strategy in YETI to get the findings for strip failure domain in one and two-dimension program. Figure 6(a) present range of pass and fail values for strip failure domain in one-dimension whereas Figure 6(b) present range of pass and fail values for strip failure domain in two-dimension program. The ranges of pass and fail values for each program in strip fault domain is given in (Table 1, Serial No. 3).

## V. Discussion

ADFD strategy with a simple graphical user interface is a fully automated process to identify and plot the pass and fail domains on the chart. Since the default settings are all set to optimum, the user needs only to specify the module to be tested and click “Draw domains” button to start test execution. All the steps including Identification of failure, generation of dynamic java program to find domain of the identified failure, saving the program to a permanent media, compiling the program to get its binary, execution of binaries to get pass and fail domain and plotting these values on the graph are done completely automated without any human intervention.

In the experiments (section 5), the ADFD strategy effectively identified failures and failure domains in a program. Identification of failure domain is simple for one and two dimension numerical program but the difficulty increases as the program dimension increases beyond two. Similarly no clear boundaries are defined for non-numerical data therefore, it is not possible to plot domains for such data unless some boundary criteria is defined.

Table 1. Pass and Fail domain with respect to one and two-dimension programs

| S. No | Failure Domain | Module Dimension | Fault | Pass Domain | Fail Domains |
|---|---|---|---|---|---|
| 1 | Point | One | PFDOneA(i) | -100 to -67, -65 to -3, -1 to 50, 2 to 22, 24 to 50, 52 to 100 | -66, -2, 23, 51 |
| | | Two | PFDTwoA(2, i) | (2, 100) to (2, 1), (2, -1) to (2, -100) | (2, 0) |
| | | | PFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |
| 2 | Block | One | BFDOneA(i) | -100 to -30, -25 to -2, 2 to 50, 55 to 100 | -1 to 1, -29 to -24, 51 to 54 |
| | | Two | BFDTwoA(-2,i) | (-2, 100) to (-2, 20), (-2, -1) to (-2, -100) | (-2, 1) to (-2, 19), (-2, 0) |
| | | | BFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |
| 3 | Strip | One | SFDOneA(i) | -100 to -5, 35 to 100 | -4 to 34 |
| | | Two | SFDTwoA(-5, i) | (-5, 100) to (-5, 40), (-5, 0) to (-5, -100) | (-5, 39) to (-5, 1), (-5, 0) |
| | | | SFDTwoA(i, 0) | Nil | (-100, 0) to (100, 0) |

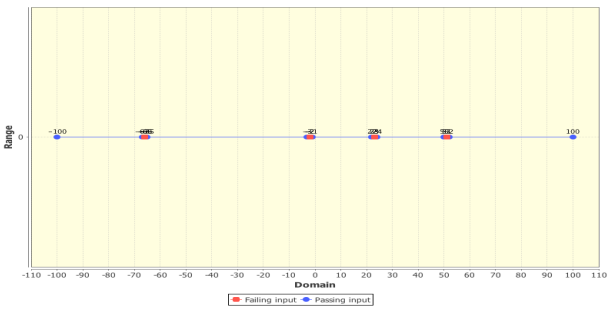



Fig. 4(a). One dimension module

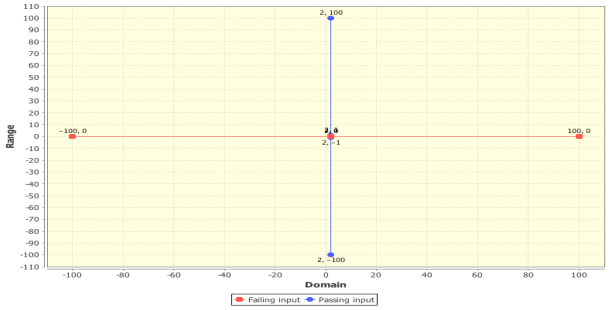



Fig. 4(b). Two dimension module

Fig. 4. ADFD strategy presenting point failure domain

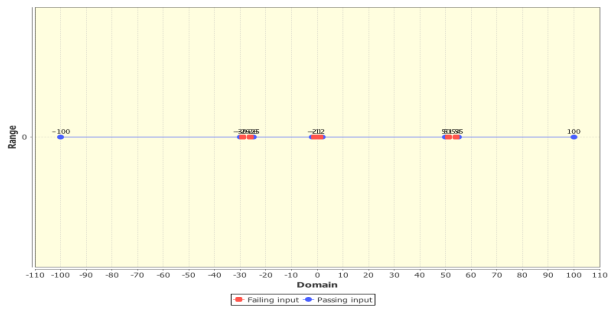



Fig. 5(a). One dimension module

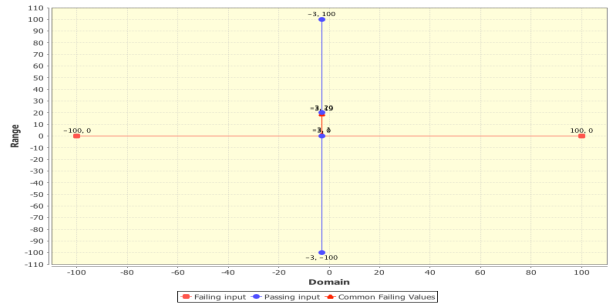



Fig. 5(b). Two dimension module

Fig. 5. ADFD strategy presenting block failure domain

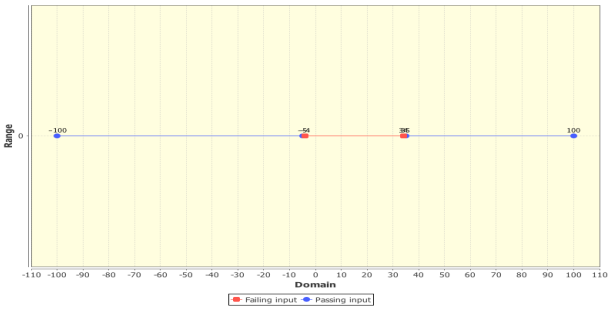



Fig. 6(a). One dimension module

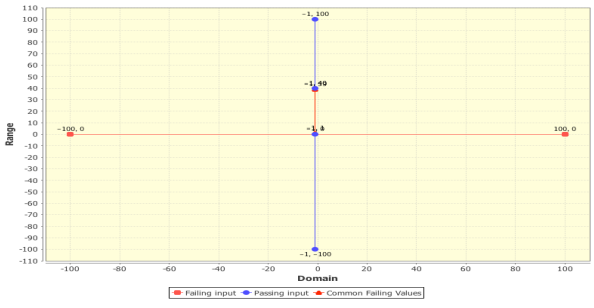



Fig. 6(b). Two dimension module

Fig. 6. ADFD strategy presenting strip failure domain

ADFD strategy initiate testing with random+ strategy to find the failure and later switch to brute-force approach to apply all the values between upper and lower bound for finding pass and fail domain. It is found that failures at boundary of the input domain can pass unnoticed through ordinary random test strategy but not from ADFD strategy as it scans all the values between lower and upper range.

The overhead in terms of execution time associated with ADFD strategy is dependent mainly on the lower and upper bound. If the lower and upper bound is set to maximum range (i.e., minimum for "int" is Integer.MIN_INT and maximum Integer.MAX_INT) then the test duration is maximum. It is rightly so because for identification of failure domain the program is executed for every input available in the specified range. Similarly increasing the range also shrinks the produced graph making it difficult to identify clearly point, block and strip domain unless they are of considerable size. Beside range factor, test duration is also influenced by the identification of the failure and the complexity of module under test.

ADFD strategy can help the debuggers in two ways. First, it reduces the to and from movement of the project between the testers and debuggers as it identify all the failures in one go. Second, it identifies locations of all failure domains across the input domain in a user-friendly way helping debugger to fix the failure keeping in view its all occurrences.

## VI. Threats to Validity

The major external threat to the use of ADFD strategy on commercial scale is the selection of small set of error-seeded programs of only integer type used in the experiments. However, the present study will serve as foundation for future work to expand it to general-purpose real world production application containing scalar and non-scalar data types.

Another issue is the easy plotting of numerical data in the form of distinctive units, because it is difficult to split the composite objects containing many fields into units for plotting. Some work has been done to quantify composite objects into units on the basis of multiple features [20], to facilitate easy plotting. Plotting composite objects is beyond the scope of the present study. However, further studies are required to look in to the matter in depth.

Another threat to validity includes evaluating program with complex and more than two input arguments. ADFD strategy has so far only considered scalar data of one and two-dimensions. However, plotting domain of programs with complex non-scalar and more than two dimension argument is much more complicated and needs to be taken up in future studies.

Finally, plotting the range of pass or fail values for a large input domain (Integer.MIN_INT to Integer.MAX_INT) is difficult to adjust and does not give a clearly understandable view on the chart. Although zoom feature is added to the strategy to zoom into the areas of interest on the chart.

## VII. Related Works

Traditional random testing is quick, easy to implement and free from any bias. In spite of these benefits, the lower fault finding ability of traditional random testing is often criticized [6], [21]. To overcome the performance issues without compromising on its benefits, various researchers have altered its algorithm as explained in section 1. Most of the alterations are based on the existence of failures and failures domains across the input domain [14].

Identification, classification of pass and fail domains and visualization of domains have not received due attention of the researchers. Podgurski et al. [22] proposed a semi-automated procedure to classify similar faults and plot them by using a Hierarchical Multi Dimension Scaling (HMDS) algorithm. A tool named Xslice [23] visually differentiates the execution slices of passing and failing part of a test. Another tool Tarantula uses color-coding to track the statements of a program during and after the execution of the test suite [24]. A serious limitation of the above-mentioned tools is that they are not fully automated and require human interaction during execution. Moreover these tools are based on the already existing test cases where as ADFD strategy generate test cases, discover failures, identify pass and fail domains and visualize them in a fully automated manner.

## VIII. Conclusion

Results of the experiments (section 4), based on applying ADFD strategy to error-seeded numerical programs provide, evidence that the strategy is highly effective in identifying the failures and plotting pass and fail domains of a given program. It further suggests that the strategy may prove effective for large programs. However, it must be confirmed with programs of more than two-dimension and different non-scalar argument types. ADFD strategy can find boundary failures quickly as against the traditional random testing, which is either, unable or takes comparatively long time to discover such failures.

The use of ADFD strategy is highly effective in testing and debugging. It provides an easy to understand test report visualizing pass and fail domains. It reduces the number of switches of SUT between testers and debuggers because all the failures are identified after a single execution. It improves debugging efficiency as the debuggers keep all the instances of a failure under consideration when debugging the fault.

## References


[1] Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley (1995)

[2] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo. In: Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on. (may 2008) 71 –80

[3] Forrester, J.E., Miller, B.P.: An empirical study of the robustness of windows nt applications using random testing. In: Proceedings of the 4th conference on USENIX Windows Systems Symposium - Volume 4. WSS'00, Berkeley, CA, USA, USENIX Association (2000) 6–6

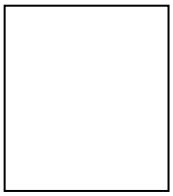
[4] Hamlet, R.: Random testing. In: Encyclopedia of Software Engineering, Wiley (1994) 970–978

[5] Linger, R.C.: Cleanroom software engineering for zero-defect software. In: Pro- ceedings of the 15th international conference on Software Engineering. ICSE '93, Los Alamitos, CA, USA, IEEE Computer Society Press (1993) 2–13

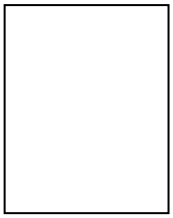
[6] Offutt, A.J., Hayes, J.H.: A semantic model of program faults. SIGSOFT Softw. Eng. Notes 21(3) (May 1996) 195–200

[7] Chen, T., Yu, Y.: On the relationship between partition and random testing. Software Engineering, IEEE Transactions on 20(12) (dec 1994) 977 –980

[8] Chen, T.Y.: Adaptive random testing. Eighth International Conference on Qualify Software 0 (2008) 443
[9] Chan, K.P., Chen, T.Y., Towey, D.: Restricted random testing. In: Proceedings of the 7th International Conference on Software Quality. ECSQ '02, London, UK, UK, Springer-Verlag (2002) 321–330
[10] Chen, T., Merkel, R., Wong, P., Eddy, G.: Adaptive random testing through dynamic partitioning. In: Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on, IEEE (2004) 79–86
[11] Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Sigplan Notices. Volume 40., ACM (2005) 213–223
[12] Mayer, J.: Lattice-based adaptive random testing. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ACM (2005) 333–336
[13] Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of the 29th international conference on Software Engi- neering. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 75–84
[14] Chan, F., Chen, T., Mak, I., Yu, Y.: Proportional sampling strategy: guidelines for software testing practitioners. Information and Software Technology 38(12) (1996) 775–782
[15] Oriol, M.: York extensible testing infrastructure (2011)
[16] Oriol, M.: 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) 201 –210
[17] Gilbert, D.: The JFreeChart class library version 1.0.9: Developer's guide. Refinery Limited, Hertfordshire (2008)
[18] Oriol, M., Tassis, S.: Testing .net code with yeti. In: Proceedings of the 2010 15th IEEE International Conference on Engineering of Complex Computer Systems. ICECCS '10, Washington, DC, USA, IEEE Computer Society (2010) 264–265
[19] Oriol, M., Ullah, F.: Yeti on the cloud. In: Proceedings of the 2010 Third Inter- national Conference on Software Testing, Verification, and Validation Workshops. ICSTW '10, Washington, DC, USA, IEEE Computer Society (2010) 434–437
[20] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Object distance and its application to adaptive random testing of object-oriented programs. In: Proceedings of the 1st international workshop on Random testing. RT '06, New York, NY, USA, ACM (2006) 55–63
[21] Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. Wiley (2011)
[22] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B.: Au- tomated support for classifying software failure reports. In: Software Engineering, 2003. Proceedings. 25th International Conference on. (may 2003) 465 – 475
[23] Agrawal, H., Horgan, J., London, S., Wong, W.: Fault localization using execution slices and dataflow tests. In: Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on. (oct 1995) 143 –151
[24] Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE '02, New York, NY, USA, ACM (2002) 467–477



**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 and a Principal Scientist at ABB Corporate Research, Industrial Software Systems, in Baden-Daettwil, Switzerland. He completed his PhD from University of Genevar 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

**Program 1:** Point domain with One argument

```
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 2:** Point domain with Two argument

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

**Program 3:** Block domain with one argument

```
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 4:** Block domain with two argument

```
public class BlockDomainTwoArgument{
    public static void pointErrors (int x, int y){
        if(((x > 0)&&(x < 20)) || ((y > 0) && (y < 20)))
            x = 5/0;
    }
}
```

**Program 5:** Strip domain with One argument

```
public class StripDomainOneArgument{
    public static void stripErrors (int x){
        if((x > -5) && (x < 35))
            x = 5/0;
    }
}
```

**Program 6:** Strip domain with two argument

```
public class StripDomainTwoArgument{
    public static void pointErrors (int x, int y){
        If(((x > 0) && (x < 40)) || ((y > 0) && (y < 40)))
            x = 5/0;
    }
}
```