Automated Discovery of Failure Domain+ and Daikon to Analyse Failure Boundaries

Mian Asbat Ahmad
Department of Computer Science
The University of York
York, United Kingdom
mian.ahmad@york.ac.uk

Manuel Oriol
Department of Computer Science
The University of York
York, United Kingdom
manuel.oriol@york.ac.uk

ABSTRACT

This paper verifies the accuracy of invariants generated automatically by Daikon and suggests how to improve their quality. To achieve this, it uses a newly developed Automated Discovery of Failure Domain+ (ADFD+) technique. ADFD+ is a testing frame-work which after identifying a failure searches its surrounding to find its domain within the set range. The result obtained is presented graphically on a two-dimension chart.

Several error-seeded one and two-dimensional numerical programs with point, block and strip failure domain were evaluated independently for 30 times by both ADFD+ and Daikon. On analysis of results, it is found that where Daikon generates the correct invariants, it was not good enough to identify the exact failure boundaries.

It is concluded that the invariants generated by Daikon can be made further effective if the boundary values of the failure domain identified by ADFD+ are passed to the Daikon as test cases.

Keywords

software testing, automated random testing

1. INTRODUCTION

Testing is the most widely used and essential method in verification and validation process. Ample efforts have been made to improve its effectiveness and efficiency. Testing is effective if it finds maximum number of faults in minimum number of test cases. Testing is efficient if it executes maximum number of test-cases in minimum possible time. Upgrading existing and developing new improved test strategies can increase test effectiveness. While automating a single component or complete system increases its efficiency.

Boundary Value Analysis [10] and Daikon [3] are among the several approaches used to increase testing effectiveness and efficiency. In Boundary Value Analysis (BVA) technique test data from boundaries of domain are added because it is assumed that errors often reside along the boundaries. Similarly, Daikon is a tool, which saves precious testers time by automatically generating likely program invariants.

However, the approaches can adversely affect testing process if wrong boundaries or invariants are taken into consideration. It is therefore motivating to measure the degree of correctness of auto-generated invariants by Daikon in the case of point, block and strip failure domain. To assess this, we set up and performed several experiments and analysed the results derived from the error-seeded programs tested independently with ADFD+ and Daikon. ADFD+ stands for Automated Discovery of Failure Domain+ is an extended test framework based on our previous approach Automated Discovery of Failure Domain (ADFD). For more details of ADFD please see [1].

The main contributions of the article are:

- **ADFD+:** It extends our previously proposed ADFD strategy. The new strategy improves the search algorithm of ADFD and makes the report more intuitive.
- **Implementation of ADFD+:** The new ADFD+ strategy is implemented and integrated in the York Extensible Testing Infrastructure (YETI).
- **Evaluation:** It evaluate the reports generated by ADFD+ and Daikon about the boundaries of the failure domains in the error-seeded programs. It is found that where Daikon was able to find the failure, it was not able to identify its domain boundary as accurately as ADFD+.
- **Future work:** It gives ideas of further application of ADFD+, such as finding and plotting of failures in multi-dimensional non-numerical programs using multi-dimensional graphs.

2. PRELIMINARIES

A number of empirical evidence confirms that failure revealing test cases tend to cluster in contiguous regions across the input domain [4, 11, 12]. According to Chan et al. [2] the clusters are arranged in the form of point, block and strip failure domain. In the point domain the failure revealing inputs are stand-alone that are spread through out

the input domain. In block domain the failure revealing inputs are clustered in one or more contiguous areas. Finally, in strip domain the failure revealing inputs are clustered in one long elongated area. Figure 1 shows the failure domains in two-dimensional input domain.

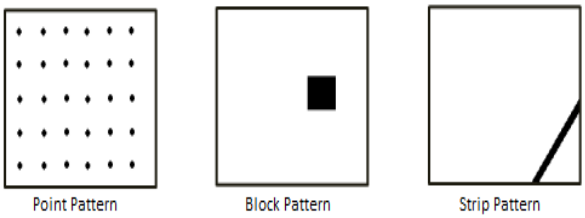


Figure 1: Failure domains across input domain [2]

3. AUTOMATED DISCOVERY OF FAILURE DOMAIN+

ADFD+ is an improved and extended form of our previously developed Automated Discovery of Failure Domain technique [1]. The ADFD+ is an automated framework that finds the failures and their domains within a specified range and present them on a graphical chart. Following are the main improvements of ADFD+ over ADFD.

- ADFD+ generates a single Java file dynamically at run time to plot the failure domains instead of one Java file per failure as in ADFD. This saves a lot of execution time and makes the process quicker.
- ADFD+ uses (x, y) vector series to represent failure domains as opposed to the (x, y) line series in ADFD. The vector series allows more flexibility and clarity to represent a failure and its domain.
- ADFD+ takes a single value as range with in which the strategy search for a failure domain whereas ADFD takes two values for lower and upper bound representing x and y-axis respectively.
- In ADFD+, the algorithm of dynamically generated Java file, created at run-time, after a failure is detected is made more simplified and efficient.
- In ADFD+, the failure domain is focused in the graph, which gives a clear view of, pass and fail points. The points are also labelled for simplification as shown in Figure 2.

3.1 Workflow of ADFD+

ADFD+ is an automatic process and all the user has to do is to specify the program to test and click the *DrawFaultDomain* button. The default value for range is set to 5, which means that ADFD+ will search 83 values around the failure. On clicking the button YETI is executed with ADFD+ strategy to search for a failure in two-dimension program. On finding a failure the ADFD+ strategy creates a Java file which contain calls to the program on the failing value and its surrounding values within the specified range. The Java file is compiled and executed and the result is analysed to check for pass and fail values. Pass and fail values are stored in pass and fail text files respectively. At the end of test, all the values are plotted on the graph with pass values in blue and fail values in red colour as shown in the Figure 2.

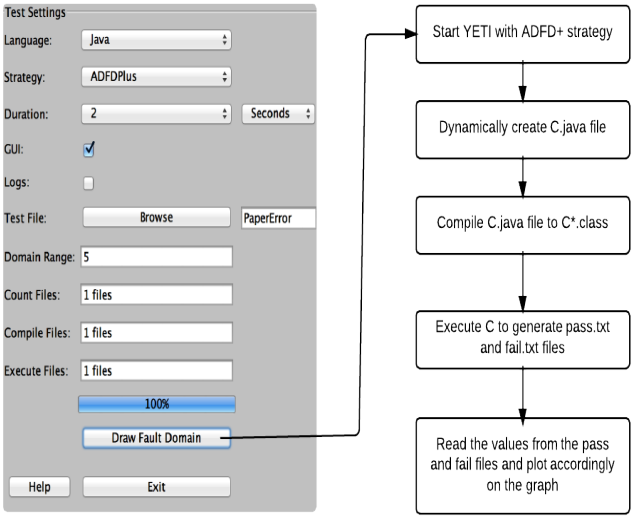


Figure 2: Workflow of ADFD+

3.2 Implementation of ADFD+

The ADFD+ technique is implemented in YETI. The tool YETI is available in open-source at `http://code.google.com/p/yeti-test/`. We give a brief overview of YETI 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 [5, 6, 8, 9] for more details on YETI.

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 favourable choice to implement ADFD+ strategy. YETI is also capable of generating test cases to reproduce the failures found during the test session. The strategies section in YETI contains all the strategies including random, random+ and DSSR to be selected for testing according to the specific needs. The default test strategy for testing is random. 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.

3.3 ADFD+ by an example

In this section we describe the working of ADFD+ with a motivating example. Suppose we have the following class name *Error* under test. According to the code, the value of

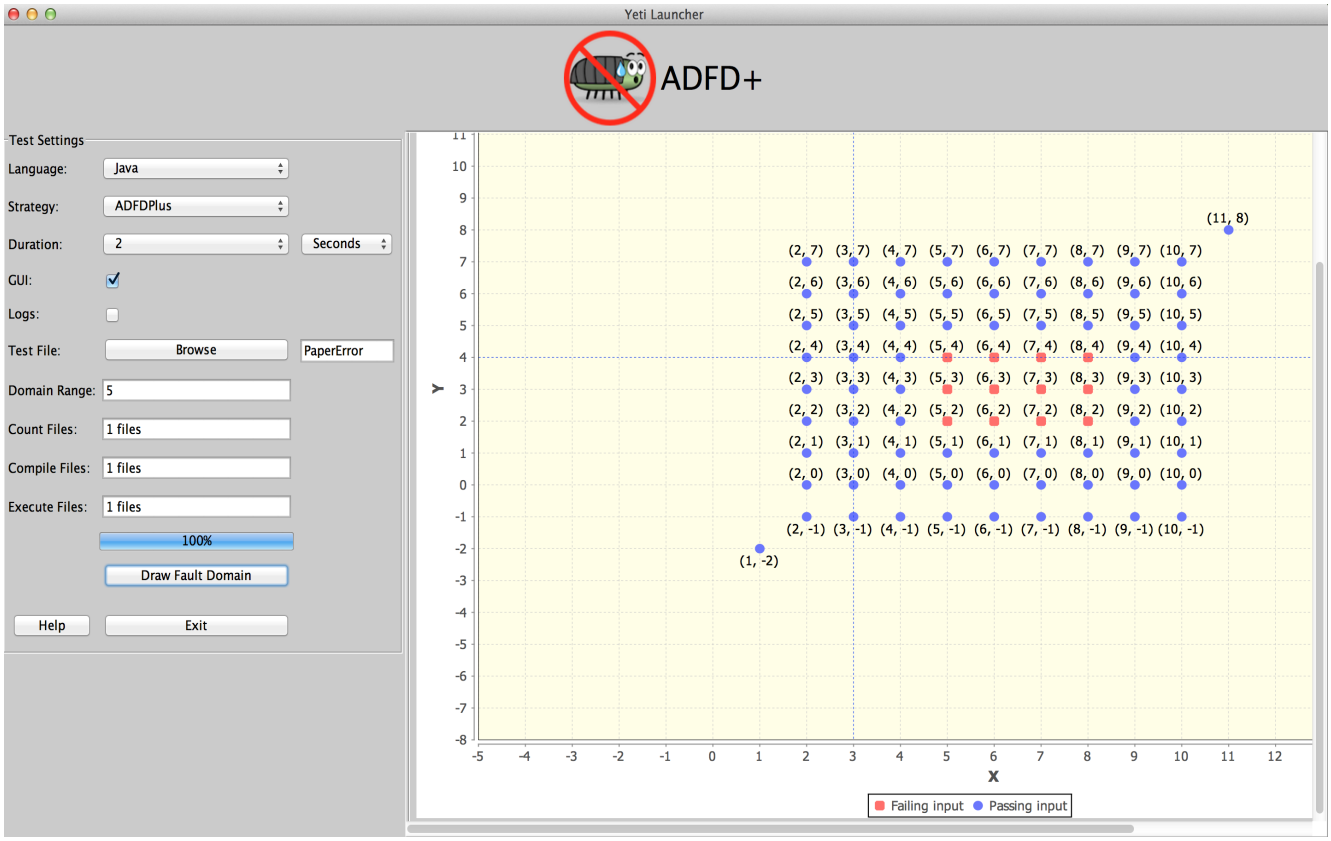


Figure 3: The output of ADFD+ for the above code.

variable x between 5 to 8 and the value of variable y between 2 to 4 triggers an *ArithmeticException* failure.

```
public class Error {

  public static void Error (int x, int y){

    int z;

    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
        {
            z = 50/0;
        }
   }
}
```

On execution, the ADFD+ strategy tests the class with the help of YETI and finds the first failure at x = 6 and y = 3. Once a failure is identified ADFD+ uses the surrounding values around it to find a failure domain. The range of surrounding values is limited to the value set by the user in the *DomainRange* variable. When the value of *DomainRange* is 5, ADFD+ evaluates total of 83 values of x and y around the found failure. All evaluated (x, y) values are plotted on a two-dimensional graph with red filled circles indicating fail values and blue filled circles indicating pass values. Figure 3 shows that the failure domain forms a block pattern and the boundaries of the failure are $(5,2), (5,3), (5,4), (6,2), (6,4), (7,2), (7,4), (8,2), (8,3), (8,4)$.

4. DAIKON

Daikon [3] is a tool, which uses machine-learning technique to automatically generate likely invariants of the program written in C, C++, Java and Pearl. Daikon takes as input the program and few test cases written manually or generated by an automated tool. It executes the test cases on the program under test and observes the values that the program computes. At the end of the test session it reports the properties that were true over the observed executions. Daikon can process the generated invariants to mitigate non-interesting and redundant invariants. Daikon can also inserts the generated invariants in to the source code as assertions. Daikon's output can be useful in understanding program, generating invariants, predicting incompatibilities in component integration, automating theorem proving, repairing inconsistent data structures and checking the validity of data streams.

5. EVALUATION

Because of using error-seeded one and two dimensional numerical programs, we were aware of the failure domain present in each program. The correct identification and presentation of the failure domain by ADFD+ prove the correct working of ADFD+. We then evaluated the same program by Daikon and plot its results. The unit test cases required by Daikon for generating invariants were generated using Randoop []. YETI being capable of generating the test cases is not used for this step to keep the second completely independent from first.

5.1 Research questions

For evaluating Daikon, the following research questions have been addressed in this study:

1. If Daikon is capable of generating invariants to identify the failure?

2. If Daikon is capable of generating invariants that identify the failure domain?

3. If Daikon is capable of correctly identifying the boundaries of the failure domain?

5.2 Experimental setup

To evaluate the invariants generated by Daikon in case of point, block and strip failure domain, we performed testing of several error seeded one and two-dimensional numerical programs written in Java. The programs were divided in to two sets. Set A and B contains one and two-dimensional programs. Each program is injected with point, block or strip failure domain. Every program is tested independently for 20 times by both ADFD+ and Daikon. The external parameters were kept constant in each test run and the initial test cases required by Daikon for generating invariants were generated by using an automated testing tool Randoop. The code for the programs under test is given in Appendix **??** while the test details are presented in Table **??**. The effect of increased number of test cases is also shown in column .. of Table **??**.

Because of the absence of the contracts and assertions in the code, similar approach as used in previous study [1, 7] is followed using undeclared exceptions to compute failures.

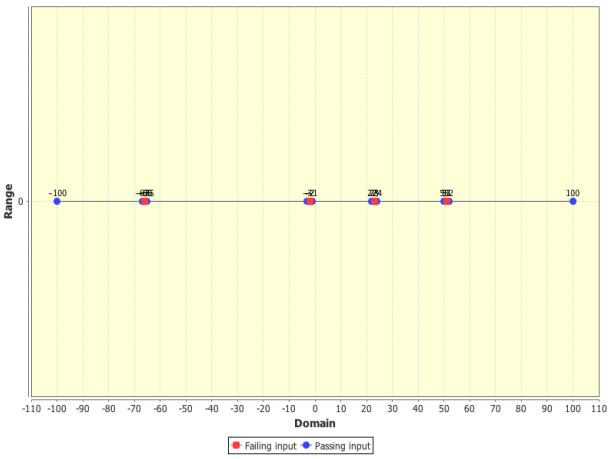
All tests are performed with a 64-bit Mac OS X Lion Version 10.7.4 running on 2 x 2.66 GHz 6-Core Intel Xeon processor with 6 GB (1333 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.6.0_35]. The machine took approximately 100 hours to process the experiments.

6. RESULTS

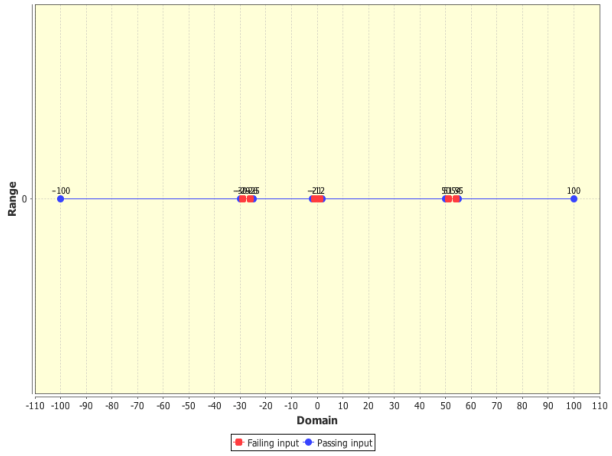
The results are split in to four sub-sections for convenience.

6.1 Test of one-dimension programs by ADFD+

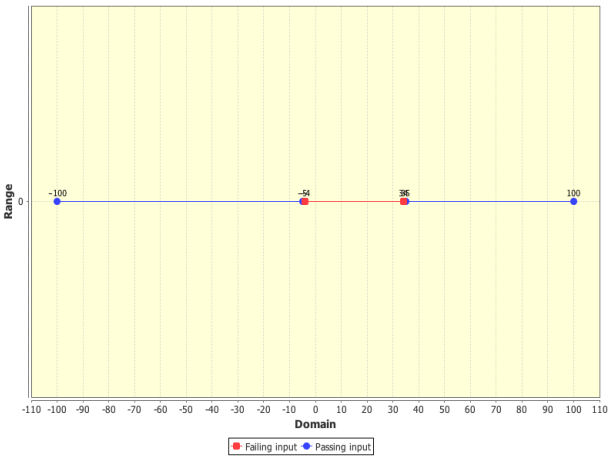
In each of the 20 experiments, The ADFD+ successfully discovered and plotted the failure domains for point, block and strip pattern as shown in the Figure **??**. The lower and upper bound for each experiment were kept — and — respectively.



(a) Point failure domain in one-dimension



(b) Block failure domain in one-dimension



(c) Strip failure domain in one dimension

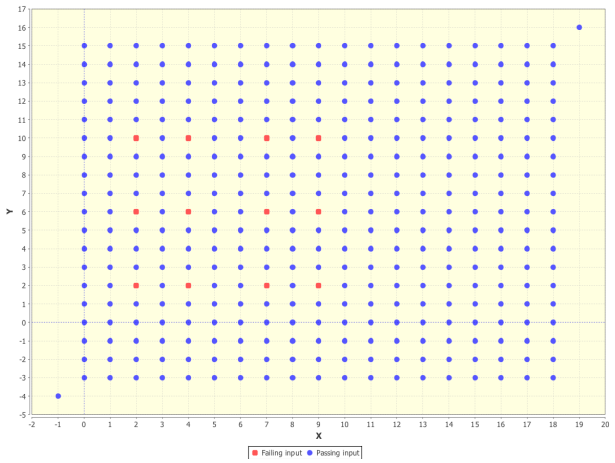
Figure 4: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

6.2 Test of one-dimension programs by Daikon

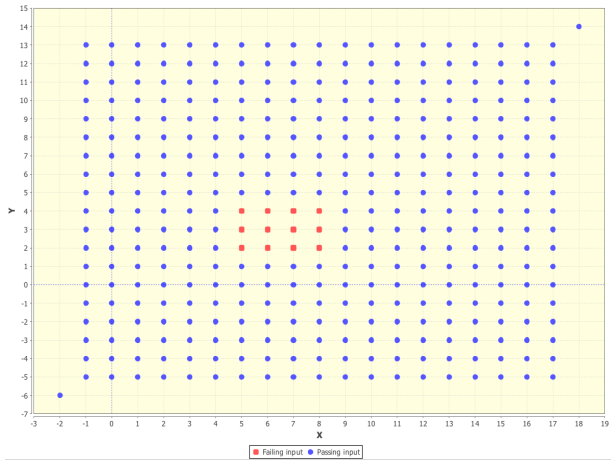
6.3 Test of two-dimension programs by ADFD+

In each of the 20 experiments, The ADFD+ once again successfully discovered and plotted the failure domain for point, block and strip failure domain as shown in the Figure **??**.

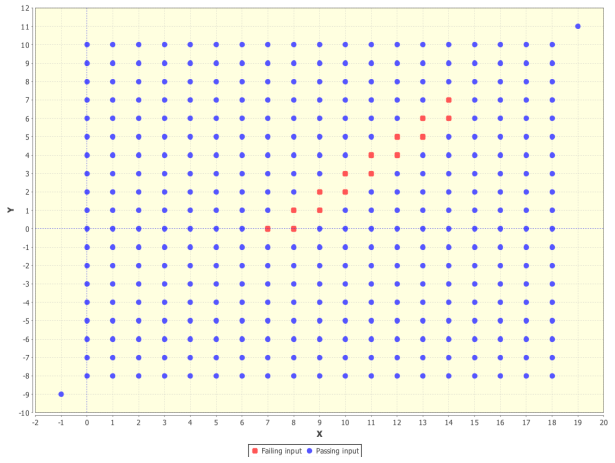
The range for each experiment were kept —-.



(a) Point failure domain in two-dimension



(b) Block failure domain in two-dimension



(c) Strip failure domain in two-dimension

Figure 5: Pass and fail values of plotted by ADFD+ in three different cases of two-dimension programs

6.4 Test of two-dimension programs by Daikon

7. DISCUSSION

We have shown that ADFD+ is a promising technique to find a failure and using it as a focal point find the whole failure domain. We have also shown that ADFD+ can graphically draw the failure domain on a chart. The failure values are drawn in red and the pass values are drawn in green. The pictorial representation of failure domain helps in easily identifying the underlying pattern and its boundaries.

As a pilot study, we also ran an empirical study to evaluate several error-seeded programs. While it would be surprising if production programs produced much different results, it would be worthwhile to check.

More importantly, the implementation of ADFD+ for this pilot study has significant limitations in practice, as it requires only one and two dimensional numerical programs. Though it is not difficult to extend the approach to test more than two-dimensional programs containing other primitive types, it would however be difficult to plot them on the chart as the number of coordinates increases. The approach can also be extended to test object-oriented programs by implementing objects distance proposed by Ciupa et al. []. The details of such an implementation will take some effort.

The ADFD+ range value specifies how many values to test around the failure. The range can be set to any number before the test starts. The value of range is directly proportional to the time taken because the higher the range value the higher number of values to test. Higher range value also leads to a very large graph and the tester has to use the zoom feature of graph to magnify the failure region.

8. THREATS TO VALIDITY

The threats to external validity are the same, common to all empirical evaluation i.e. the degree to which the tested classes and external test generation tool (Randoop) are representative of true practice. The classes are very simple concerning failure domain in only one and two-dimensional input domain. The threats could be reduced by experiments on various types of classes and different auto test generation tools. The main threat to internal validity includes annotation of invariants that can bias our results. Error seeded classes selected in our implementation, might cause such effects. Taking in to consideration the real failures in real classes could also reduce the threat. Furthermore, testing a higher number of classes would naturally have increased the reliability of the results.

9. CONCLUSION

10. FUTURE WORK

We aim to extend the current approach to a larger set of real world multi-dimension programs, using real failure instead of error-seeded programs. Moreover, to plot failure domains of complex multi-dimension shapes we would also require more sophisticated graphical tool like Matlab than JFreeChart. This could also result in the formation of new failure domains of different nature instead of the only point, block and strip failure domain in one and two dimension numerical programs.

11. ACKNOWLEDGMENTS

The authors thank the Department of Computer Science, University of York for its financial support with the Departmental Overseas Research Scholarship (DORS) award. We also thanks to Richard Page for his valuable help and generous support.

12. REFERENCES

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

[2] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

[3] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

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

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

[6] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201–210. IEEE, 2012.

[7] M. Oriol. Random testing: Evaluation of a law describing the number of faults found. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 201 –210, april 2012.

[8] M. Oriol and S. Tassis. Testing. net code with yeti. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 264–265. IEEE, 2010.

[9] M. Oriol and F. Ullah. Yeti on the cloud. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 434–437. IEEE, 2010.

[10] J. Radatz, A. Geraci, and F. Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[11] C. Schneckenburger and J. Mayer. Towards the determination of typical failure patterns. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*, pages 90–93. ACM, 2007.

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