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 verify the accuracy of invariants generated automatically by Daikon and suggests how to improve their quality. To achieve this, it uses a newly developed technique called Automated Discovery of Failure Domain+ (ADFD+). It is a testing framework that after identifying a failure search its surrounding values to the specified range. The result obtained is presented graphically indicating pass and fail points.

Several error seeded two dimensional programs were tested in the experiments with point block and strip fault domain were evaluated first with ADFD+ and than with Daikon. On comparison of results from both the methods it is found that where Daikon generates the correct invariants, it was not good enough to identify the fault boundaries.

It is concluded that it will be highly effective if the fault boundary found by ADFD+ is passed to the Daikon in the first stage then the invariants generated by Daikon will correctly point to the fault boundary.

Keywords
software testing, automated random testing

1. INTRODUCTION
Testing is the most widely used and essential method of software testing. 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 efficiency is the process to execute maximum number of test cases in minimum possible time. Test effectiveness can be increased by upgrading the existing and developing new improved test strategies. While test efficiency can be increased by automating a single component or complete system of software testing.

Boundary value analysis technique [], where test data from the boundaries of domain are selected assuming of high chances, and Daikon [], which is a tool to automatically generate likely program invariants are among the several ways of improving test efficiency and effectiveness.

However, the efforts can adversely affect if the wrong boundaries or invariants are taken into consideration. It is therefore interesting to see how much the auto generated invariants by Daikon represent the failure domain residing in the input domain. To assess this, we set up and performed several experiments and analysed the results derived from the error-seeded programs tested with Daikon and ADFD+. The ADFD+ is a framework named Automated Discovery of Failure Domain+. It is based on our previous strategy Automated Discovery of Failure Domain (ADFD), which tries to find a fault, search the surrounding for more faults and graphically plot the fault domain if any [1].

The main contribution of the article are:

- **ADFD+:** It brings some changes to the previously proposed ADFD strategy. The new strategy improves the search algorithm of ADFD and make the report more intuitive.
- **Implementation of ADFD+:** The new ADFD+ strategy is implemented and integrated in the YETI tool.
- **Evaluation:** It evaluates the report generated by Daikon and ADFD+ about the boundaries known fault domains in the error-seeded programs. It is found that where Daikon was able to find the fault, 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 faults in multi-dimensional programs using multi-dimensional graphs.

2. FAILURE DOMAINS
A number of empirical evidence confirms that fault revealing test cases tend to cluster in contiguous regions across the input domain [11, 4, 10]. According to Chan et al. [2] the clusters are arranged in the form of point, block and strip failure pattern. In the point pattern the failure revealing inputs are stand alone which are spread through out the input domain. In block pattern the failure revealing inputs are clustered in a one or more contiguous area. Finally, in

strip pattern the failure revealing inputs are clustered in one long elongated area. Figure 1 shows the failure patterns in two-dimensional input domain.

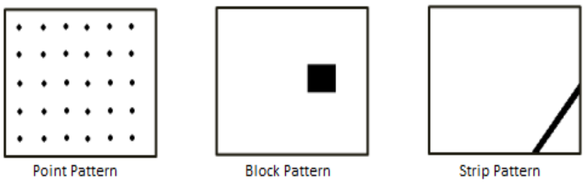

Figure 1: Failure domains across input domain [2]

3. AUTOMATED DISCOVERY OF FAILURE DOMAIN+

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

- ADFD+ generate 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 much quicker.
- ADFD+ uses (x, y) vector series to represent failure pattern 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 domain.
- ADFD+ takes a single value as range which specify a round region around the failure whereas takes two values for lower and upper bound representing x and y axis respectively.
- In ADFD+, the algorithm of dynamically generated Java file, which is created after an error is discovered, 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 the 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 25 values around the failure. On clicking the button YETI is executed with ADFD+ strategy to search for a failure. On finding a failure the ADFD+ strategy create a java file which contain calls to the program on the failing value and related values up to the specified range. The Java file is compiled and executed. The result is analysed to check for pass and failed values. Pass values are stored in pass file and fail values are stored in fail file. At the end of the values range, the values are plotted on the graph with pass values as green and fail values as red as shown in the Figure 2.

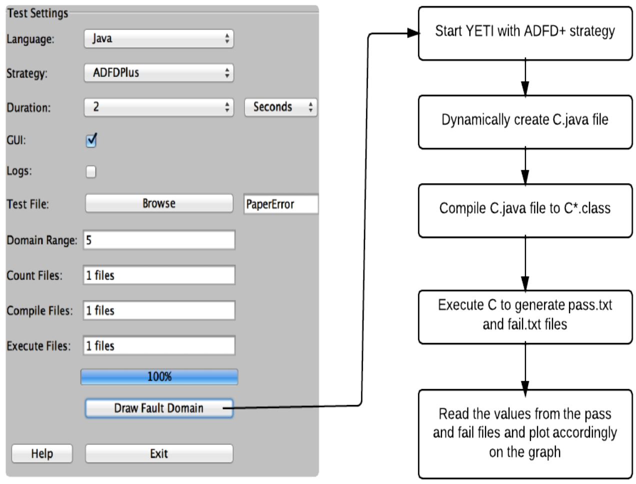

Figure 2: Workflow of ADFD+

3.2 Implementation of ADFD+

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 [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 favorable choice to implement ADFD+ strategy. YETI is also capable of generating test cases to reproduce the faults 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 variable x between 5 to 8 and the value of variable y between 2 to 4 triggers a failure.

```
public class Error {
```

```java
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 discovered 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 dot indicating a failing value and green dot indicating the passing value. Figure 3 clearly show 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 Experiments

To evaluate the Daikon performance for failure domain, we performed testing of several error seeded one and two dimensional numerical programs written in Java. All the selected programs were seeded with at least one failure domain of point, block or strip nature. Every program is tested twenty times by both ADFD+ and Daikon. The program code is given in Appendix **??** while the test details is presented in Table **??**.

Every class is evaluated through 10^5 calls in each test session of ADFD+. Because of the absence of the contracts and assertions in the code under test, similar approach as used in previous studies [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

for point, block and strip of one dimensional program. Use the same programs of ADFD, same figures but analyse it again on Daikon. because ADFD and ADFD+ behave in the same way for one dimension.

For point block and strip of two dimensional programs. Use adfd+ system.

7. DISCUSSION

8. THREATS TO VALIDITY

The threats to external validity is 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 include annotation of invariants that can bias our results. Error seeded classes selected in our implementation, might cause such effects. The threat could also be reduced by taking in to consideration the real failures in real classes. 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

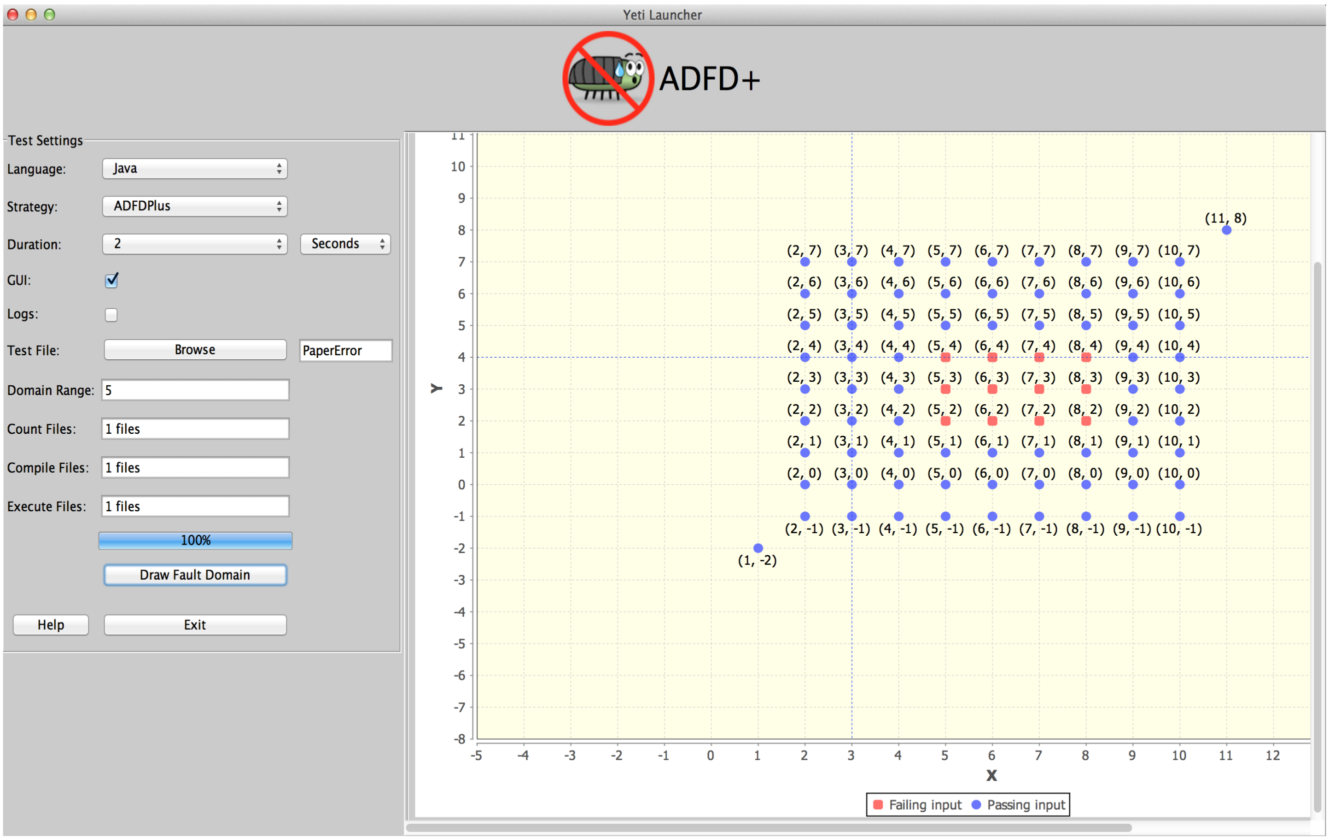

Figure 3: The output of ADFD+ for the above code.

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] 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.

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