Automated Discovery of Failure Domain+ and Daikon to Analyse 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

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms
Comparison, Verification,

Keywords
software testing, automated random testing

1. INTRODUCTION
Testing is the most widely used and essential method of software testing. Therefore, the aim of researchers is to improve the effectiveness and efficiency of software testing process. Testing effectiveness is the increase in number of faults found in a specific number of test cases, which is achieved by upgradation of existing and development of new improved testing techniques. While the improve in testing efficiency is the decrease in the total time taken by a software testing process, which is most of the time accomplished by automation of the whole or a specific part of software testing process, like test data generation, execution and oracle formation.

Daikon [] is a tool that automate part of the testing process by auto generation of likely program invariants. The invariants are processed and can be annotated in the source code of the program to facilitate the testing process. While the tool helps to increase the efficiency, testing effectiveness is also dependant on it because the invariants produced by the tool are evaluated in the test execution to decide a pass and fail test case.

It is interesting to find out if we are compromising on quality by increasing the test efficiency. Therefore we set up and ran various experiments and analysed the results derived from the same set of error seeded programs tested with and without the assistance of Daikon. For this purpose, we proposed a new strategy named Automated Discovery of Failure Domain+. It is an upgraded strategy based on the combination of our two previous strategies Dirt Spot Sweeping Random (DSSR) and Automated Discovery of Failure Domain (ADFD), which discovers the fault surrounding values and graphically plot the fault domains respectively [].

The main contribution of this article include:

* To propose develop and implement an updated ADFD+ strategy.

* To analyse the failure boundaries derived by ADFD+ and Daikon.

* A case study suggesting that boundaries are properly recognized by Daikon and ADFD+ or Daikon lake .... etc.

The rest of this paper is organised as follows:
Section 2 describes the ADFD+ strategy. Section **??** presents implementation of the ADFD+ strategy. Section 10 explains the experimental setup. Section **??** shows results of the experiments. Section 11 discusses the results. Section 12 presents related work and Section 13, concludes the study.

2. AUTOMATED DISCOVERY OF FAILURE DOMAIN+
Automated Discovery of Failure Domain+ (ADFD+) is a successor of our previously developed technique Automated Discovery of Failure Domain (ADFD). ADFD is an automated framework that finds the failures, their domains and present these on a graphical chart[**?**].

2.1 Workflow of ADFD+
Instead of front end give workflow. It will make more sense. Change the code of the program

2.2 Front-end of ADFD+
Front end of ADFD+ is showing in the Figure 1. Here the user had set range to 5 which means that ADFD+ strategy will search the region up to 5 values around the discovered failure. Here ADFD+ using random and random+ strategy found the first failure in the method under test when the argument x and y value were 10 and -3 respectively. As soon as the failure is discovered the strategy start for any other failures in the vicinity under the restricted range. The

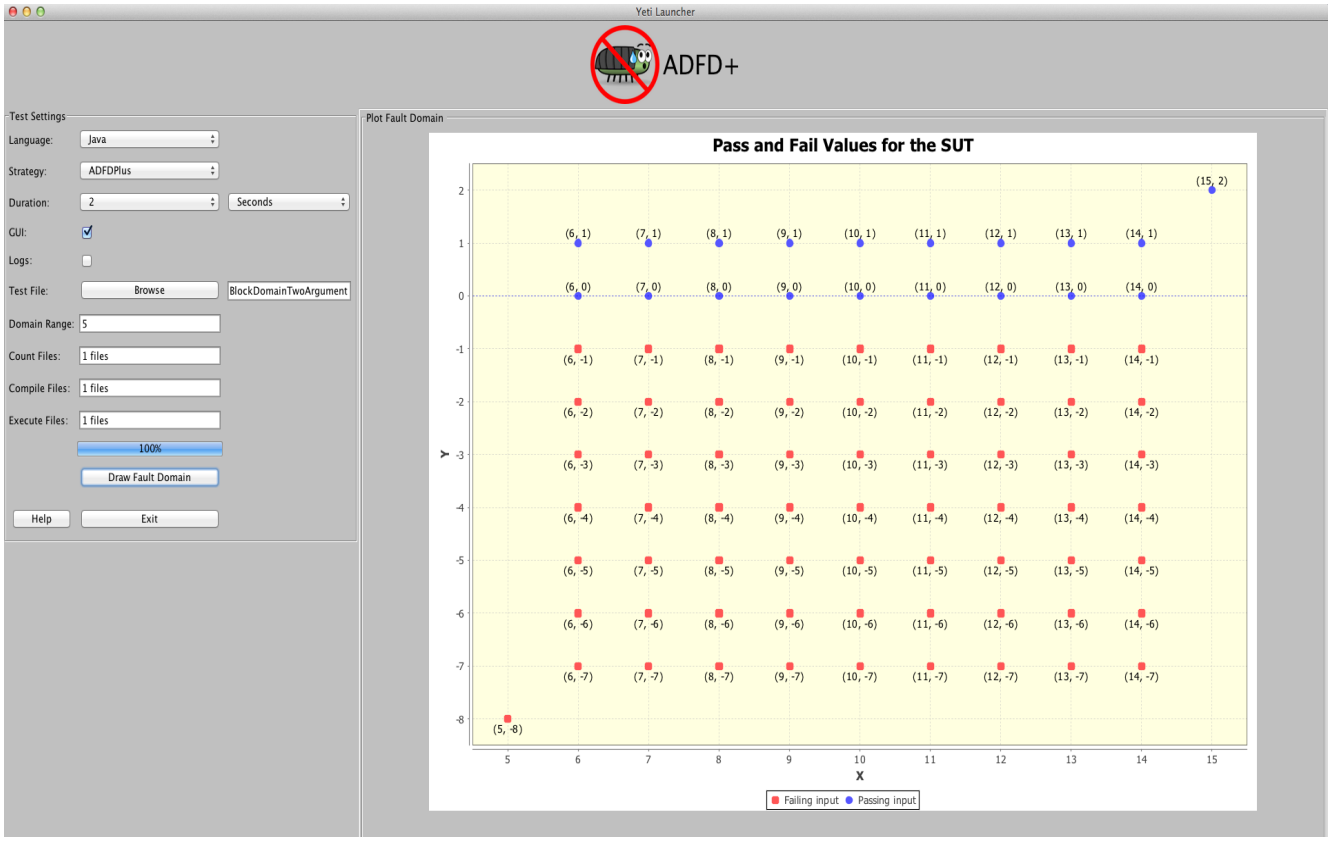

Figure 1: Frond-end of ADFD+ test framework.

figure clearly show the fail values with red dots and the correct values with green dot on the graph. T

ADFD+ is an extension of ADFD's algorithm with more accuracy to find and clarity to plot the failure domain on a graphical chart. Deriving failure domains using ADFD+ is a one click process and all the tester needs to input is the class to test and the range-value for which to search around the found failure.

2.3 Difference of ADFD and ADFD+

The workflow of ADFD and ADFD+ is the same except the following changes.

1) ADFD+ generate a single Java file 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.

2) ADFD+ uses vector series to represent failure pattern as opposed to the line series in ADFD. The vector series allows more flexibility and clarity to represent a failure domain.

```
/**
* Calculate square of given number
* and verify results.
* The code contain 3 faults.
* @author (Mian and Manuel)
*/
public class Math1 {
 public void calc (int num1) {
  // Square num1 and store result.
  int result1 = num1 * num1;
  int result2 = result1 / num1; // 1
  assert Math.sqrt(result1) == num1; // 2
  assert result1 >= num1; // 3
 }
}
```

The difference in representation of fault by ADFD and ADFD+ can be seen in figure .... Figure x is generated by ADFD with lower bound as ... and upper bound as ... While Figure Y is generated by ADFD+ with range ... for the same program given in appendix a.

3. DAIKON

3.1 Generation and Annotation of Invariants

4. EXPERIMENTAL SETUP

5. RESULTS

6. DISCUSSION

7. THREATS TO VALIDITY

8. CONCLUSION

9. FUTURE WORK

10. EVALUATION

The DSSR strategy is experimentally evaluated by comparing its performance with that of random and random+ strategy [**?**]. General factors such as system software and hardware, YETI specific factors like percentage of null values, percentage of newly created objects and interesting value injection probability have been kept constant in the experiments.

10.1 Research questions

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

1. Is there an absolute best among R, R+ and DSSR strategies?
2. Are there classes for which any of the three strategies provide better results?
3. Can we pick the best default strategy between R, R+ and DSSR?

10.2 Experiments

To evaluate the performance of DSSR we performed extensive testing of programs from the Qualitas Corpus [**?**]. The Qualitas Corpus is a curated collection of open source java projects built with the aim of helping empirical research on software engineering. These projects have been collected in an organised form containing the source and binary forms. Version 20101126, which contains 106 open source java projects is used in the current evaluation. In our experiments we selected 60 random classes from 32 random projects. All the selected classes produced at least one fault and did not time out with maximum testing session of 10 minutes. Every class is tested thirty times by each strategy (R, R+, DSSR). Name, version and size of the projects to which the classes belong are given in table 1 while test details of the classes is presented in table **??**. Line of Code (LOC) tested per class and its total is shown in column 3 of table **??**.

Every class is evaluated through 10^5 calls in each test session.[1] Because of the absence of the contracts and assertions in the code under test, Similar approach as used in previous studies [**?**] is followed using undeclared exceptions to compute unique 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.

10.3 Are there classes for which any of the three strategies provide better results?

T-tests applied to the data given in Table 2 show that DSSR is significantly better in 7 classes from R and R+ strategy, in 8 classes DSSR performed similarly to R+ but significantly higher than R, and in 2 classes DSSR performed similarly to R but significantly higher than R+. There is no case R and R+ strategy performed significantly better than DSSR strategy. Expressed in percentage: 72% of the classes do not show significantly different behaviours whereas in 28% of hte classes, the DSSR strategy performs significantly better than at least one of R and R+. It is interesting to note that in no single case R and R+ strategies performed better than DSSR strategy. We attribute this to DSSR possessing the qualities of R and R+ whereas containing the spot sweeping feature.

[1]The total number of tests is thus $60 \times 30 \times 3 \times 10^5 = 540 \times 10^6$ *tests*.

S. No	Project Name	Version	Size (MB)
1	apache-ant	1.8.1	59
2	antlr	3.2	13
3	aoi	2.8.1	35
4	argouml	0.30.2	112
5	artofillusion	281	5.4
6	aspectj	1.6.9	109.6
7	axion	1.0-M2	13.3
8	azureus	1	99.3
9	castor	1.3.1	63.2
10	cayenne	3.0.1	4.1
11	cobertura	1.9.4.1	26.5
12	colt	1.2.0	40
13	emma	2.0.5312	7.4
14	freecs	1.3.20100406	11.4
15	hibernate	3.6.0	733
16	hsqldb	2.0.0	53.9
17	itext	5.0.3	16.2
18	jasml	0.10	7.5
19	jmoney	0.4.4	5.3
20	jruby	1.5.2	140.7
21	jsXe	04_beta	19.9
22	quartz	1.8.3	20.4
23	sandmark	3.4	18.8
24	squirrel-sql	3.1.2	61.5
25	tapestry	5.1.0.5	69.2
26	tomcat	7.0.2	24.1
27	trove	2.1.0	18.2
28	velocity	1.6.4	27.1
29	weka	3.7.2	107
30	xalan	2.7.1	85.4
31	xerces	2.10.0	43.4
32	xmojo	5.0.0	15

Table 1: Name and versions of 32 Projects randomly selected from the Qualitas Corpus for the experiments

10.4 Can we pick the best default strategy between R, R+ and DSSR?

Analysis of the experimental data reveal that DSSR strategy has an edge over R and R+. This is because of the additional feature of Spot Sweeping in DSSR strategy.

In spite of the better performance of DSSR strategy compared to R and R+ strategies the present study does not provide ample evidence to pick it as the best default strategy because of the overhead induced by this strategy (see next section). Further study might give conclusive evidence.

11. DISCUSSION

S. No	Class Name	T-test Results			Interpretation
		DSSR, R	DSSR, R+	R, R+	
1	AjTypeImpl	1	1	1	
2	Apriori	**0.03**	0.49	0.16	
3	CheckAssociator	**0.04**	**0.05**	0.44	DSSR better
4	Debug	**0.03**	0.14	0.56	
5	DirectoryScanner	**0.04**	**0.01**	0.43	DSSR better
6	DomParser	**0.05**	0.23	0.13	
7	EntityDecoder	**0.04**	0.28	0.3	
8	Font	0.18	0.18	1	
9	Group	0.33	**0.03**	**0.04**	DSSR = R > R+
10	Image	**0.03**	**0.01**	0.61	DSSR better
11	InstrumentTask	0.16	0.33	0.57	
12	JavaWrapper	**0.001**	0.57	0.004	DSSR = R+ > R
13	JmxUtilities	0.13	0.71	0.08	
14	List	**0.01**	0.25	**0**	DSSR = R+ > R
15	NodeSequence	0.97	**0.04**	**0.06**	DSSR = R > R+
16	NodeSet	**0.03**	0.42	0.26	
17	Project	**0.001**	0.57	**0.004**	DSSR better
18	Repository	**0**	1	**0**	DSSR = R+ > R
19	Scanner	1	**0.03**	**0.01**	DSSR better
20	Scene	**0**	**0**	1	DSSR better
21	Server	**0.03**	0.88	**0.03**	DSSR = R+ > R
22	Sorter	**0**	0.33	**0**	DSSR = R+ > R
23	Statistics	**0**	0.43	**0**	DSSR = R+ > R
24	Stopwords	**0**	0.23	**0**	DSSR = R+ > R
25	StringHelper	**0.03**	0.44	0.44	DSSR = R+ > R
26	Trie	0.1	0.33	0.47	DSSR better
27	WebMacro	0.33	1	0.16	
28	XMLEntityManager	0.33	0.33	0.16	
29	XString	0.14	**0.03**	0.86	

Table 2: T-test results of the classes showing different results

In this section we discuss various factors such as the time taken, effect of test duration, number of tests, number of faults in the different strategies and the effect of finding first fault in the DSSR strategy. **Time taken to execute an equal number of test cases:** The DSSR strategy takes slightly more time (up to 5%) than both pure random and random plus which may be due to maintaining sets of interesting values during the execution. We do not believe that the overhead can be reduced.

Effect of test duration and number of tests on the results: All three techniques have the same potential for finding failures. If testing is continued for a long duration then all three strategies will find the same number of unique failures and the results will converge. We suspect however that some of the unique failures will take an extremely long time to be found by using random or random+ only. Further experiments should confirm this point.

Effect of number of faults on results: We found that the DSSR strategy performs better when the number of faults is higher in the code. The reason seems to be that when there are more faults, their domains are more connected and DSSR strategy works better. Further studies might use historical data to pick the best strategy.

Dependence of DSSR strategy to find the first unique failure early enough: During the experiments we noticed that if a unique failure is not found quickly enough, there is no value added to the list of interesting values and then the test becomes equivalent to random+ testing. This means that better ways of populating failure-inducing values are needed for sufficient leverage to DSSR strategy. As an example, the following piece of code would be unlikely to fail under the current setting:

```
public void test(float value){
 if(value == 34.4445)   10/0;
}
```

In this case, we could add constant literals from the SUT to the list of interesting values in a dynamic fashion. These literals can be obtained from the constant pool in the class files of the SUT.

In the example above the value 34.4445 and its surrounding values would be added to the list of interesting values before the test starts and the DSSR strategy would find the unique failure right away.

DSSR strategy and coverage: Random strategies typically achieve high level of coverage [**?**]. It might also be interesting to compare R, R+ and DSSR with respect to the achieved coverage or even to use a DSSR variant that adds a new interesting value and its neighbours when a new branch is reached.

Threats to validity: As usual with such empirical studies, the present work might suffer from a non-representative selection of classes. The selection in the current study is however made through random process and objective criteria, therefore, it seems likely that it would be representative.

The parameters of the study might also have prompted incorrect results. But this is unlikely due to previous results on random testing [**?**].

12. RELATED WORK

Random testing is a popular technique with simple algorithm but proven to find subtle faults in complex programs and Java libraries [?, ?, ?]. Its simplicity, ease of implementation and efficiency in generating test cases make it the best choice for test automation [?]. Some of the well known automated tools based on random strategy includes Jartege [?], Eclat [?], JCrasher [?], AutoTest [?, ?] and YETI [?, ?].

In pursuit of better test results and lower overhead, many variations of random strategy have been proposed [?, ?, ?, ?, ?]. Adaptive random testing (ART), Quasi-random testing (QRT) and Restricted Random testing (RRT) achieved better results by selecting test inputs randomly but evenly spread across the input domain. Mirror ART and ART through dynamic partitioning increased the performance by reducing the overhead of ART. The main reason behind better performance of the strategies is that even spread of test input increases the chance of exploring the fault patterns present in the input domain.

A more recent research study [?] stresses on the effectiveness of data regeneration in close vicinity of the existing test data. Their findings showed up to two orders of magnitude more efficient test data generation than the existing techniques. Two major limitations of their study are the requirement of existing test cases to regenerate new test cases, and increased overhead due to "meta heuristics search" based on hill climbing algorithm to regenerate new data. In DSSR no pre-existing test cases are required because it utilises the border values from R+ and regenerate the data very cheaply in a dynamic fashion different for each class under test without any prior test data and with comparatively lower overhead.

The random+ (R+) strategy is an extension of the random strategy in which interesting values, beside pure random values, are added to the list of test inputs [?]. These interesting values includes border values which have high tendency of finding faults in the given SUT [?]. Results obtained with R+ strategy show significant improvement over random strategy [?]. DSSR strategy is an extension of R+ strategy which starts testing as R+ until a fault is found then it switches to spot sweeping.

A common practice to evaluate performance of an extended strategy is to compare the results obtained by applying the new and existing strategy to identical programs [?, ?, ?]. Arcuri et al. [?], stress on the use of random testing as a baseline for comparison with other test strategies. We followed the procedure and evaluated DSSR strategy against R and R+ strategies under identical conditions.

In our experiments we selected projects from the Qualitas Corpus [?] which is a collection of open source java programs maintained for independent empirical research. The projects in Qualitas Corpus are carefully selected that spans across the whole set of java applications [?, ?, ?].

13. CONCLUSIONS

The main goal of the present study was to develop a new random strategy which could find more faults in lower number of test cases. We developed a new strategy named. "DSSR strategy" as an extension of R+, based on the assumption that in a significant number of classes, failure domains are contiguous or located closely. The DSS strategy, a strategy which adds neighbouring values of the failure finding value to a list of interesting values, was implemented in the random testing tool YETI to test 60 classes, 30 times each, from Qualitas Corpus with each of the 3 strategies R, R+ and DSSR. The newly developed DSSR strategy uncovers more unique failures than both random and random+ strategies with a 5% overhead. We found out that for 7 (12%) classes DSSR was significantly better than both R+ and R, for 8 (13%) classes DSSR performed similarly to R+ and significantly better than R, while in 2 (3%) cases DSSR performed similarly to R and significantly better than R+. In all other cases, DSSR, R+ and R do not seem to perform significantly differently. Overall, DSSR yields encouraging results and advocates to develop the technique further for settings in which it is significantly better than both R and R+ strategies.