To find the effectiveness of ADFD and ADFD+ techniques

Mian Asbat Ahmad and Manuel Oriol

***Abstract*—The achievement of up-to 50% better results by Adaptive Random Testing verses Random Testing ensures that the pass and fail domains across the input domain are highly useful and need due consideration during selection of test inputs. The Automated Discovery of Failure Domain (ADFD) and its successor Automated Discovery of Failure Domain+ (ADFD+) techniques, automatically find failures and their domains in a specified range and provides their visualisation. They can precisely detect the failure-domain of the identified failure in an effective way. Performing exhaustive testing in a limited region around the failure is the key to the success of ADFD and ADFD+ techniques.**

We performed an extensive experimental analysis of Java projects contained in Qualitas Corpus for finding the effectiveness of automated techniques (ADFD and ADFD+). The results obtained were analysed and cross-checked using manual testing. Furthermore the impact of nature, location, size, type and complexity of failure-domains on the testing techniques were also studied. The results provide insights into the effectiveness of automated techniques and a number of lessons for testing researchers and practitioners.

***Index Terms*—software testing, automated random testing, manual testing, ADFD, Daikon.**

I. Introduction

The input-domain of a given SUT can be divided into two sub-domains. The pass-domain, containing the values, for which the software behaves correctly and the failure-domain for which the software behaves incorrectly. Chan et al. [1] observed that input inducing failures are contiguous and form certain geometrical shapes in the input domain. They divided these into point, block and strip failure-domains as shown in Figure 1.

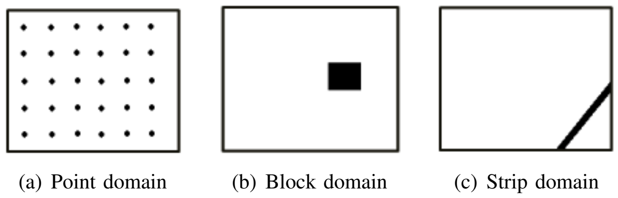
(a) Point domain (b) Block domain (c) Strip domain

Fig. 1. Failure domains across input domain [1]

Adaptive Random Testing (ART) exploited the existence of the failure-domains and resultantly achieved up to 50% better performance than random testing [2]. This was mainly attributed to the better distribution of input which increased the chance of selecting inputs from failure-domains. This insight motivated us to increase our understanding of failure-domains in production software.

Software testing cost is approximately half of the development cost. Testing software is expensive and it becomes tedious, laborious and error-prone if performed manually [3]. Automated testing is an alternative approach to manual testing. The case study reveals that the 150 hours of automated testing found more faults in complex .NET code than a test engineer finds in one year by manual testing [4].

We have developed two fully automated techniques ADFD [5] and ADFD+ [6], which effectively find failures and their domains in a specified range and provides visualisation of the pass and fail domain as either point, block or strip failure-domain. This is achieved in two steps: first, random testing is used to find the failure and secondly, exhaustive testing in a limited region around the detected failure is done to identify the failure domain. The ADFD strategy searches in one-dimensional and covers longer range than ADFD+ which is more effective in multi-dimensional and covers shorter range.

Both ADFD and ADFD+ techniques are formed by the combination of three separate tools i.e. YETI, Daikon and JFreeChart. York Extensible Testing Infrastructure [7] is used to test the program automatically using random strategy (ADFD and ADFD+). Daikon [8] observes the all the test execution and automatically generate invariants to show the failure-domains. Finally, JFreeChart [9] generates the graph from the test output to show the pass and failure-domains.

Software testing can be performed either automatically or manually. Both the techniques have their own advantages and limitations. The main advantage of automated testing is execution of large number of tests in little time, whereas manual testing utilizes the tester experience to concentrate on error-prone part of the SUT and generate target oriented test cases [10].

The rest of the paper is organized as follows: Section II presents an overview of ADFD+ technique. Section III evaluates and compares ADFD+ technique with Randoop. Section IV reveals results of the experiments. Section V discusses the results. Section VI presents the threats to validity. Section VII presents related work. Finally, Section VIII concludes the study.

II. Automated Discovery of Failure-domain

III. Automated Discovery of Failure-domain+

IV. Automated Discovery of Failure Domain+

V. Evaluation

To evaluate the presence, nature and type of failure-domains in production software we tested the main jar files of all the 106 projects in Qualitas Corpus []. The source code of the programs containing failure-domains were also evaluated manually to verify the conformance of automated results. Only one and two dimensional numerical programs were selected for evaluation. Every program was tested independently by ADFD, ADFD+ and manual testing. All the programs in which failure-domains were identified are presented in Table **??**. 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 [5], [11].

A. Research questions

The following research questions have been addressed in the study:

1) *Is ADFD and ADFD+ techniques capable of correctly identifying failure-domains in production software?* The experimental results claiming the correct identification of ADFD and ADFD+ were based on the purpose build error-seeded programs []. To answer the question, we applied the two techniques to all the projects of Qualitas Corpus and examined the results.
2) *What is the frequency of existence of point, block and strip failure-domains in production software?*
3) *What are the types of identified failure-domains?* There are strategies []. that exploit the presence of block and strip failure-domain to get better results. Therefore identifying the presence of underlying failure-domains in production software can help in high quality of software testing. To answer the questions, we reviewed all the classes containing failure-domains manually, automatically and graphically.
4) *If the nature of identified failure-domains simple or complex?* An interesting point is to know what failure is responsible for a failure-domain and how difficult it is to identify that failure by manual testing. To answer this question, we studied the test logs and test output of the automated testing and the source code of the program manually to identify the cause and complexity of failures of failure-domains.
5) *If the invariants generated by Daikon correctly represent the failure domains?* Invariants generated by Daikon can identify the start and stop of the failure domain. To answer this question we compared the generated invariants with the source code and the failure-domain presented in graphical form.
6) *If the graph generated by ADFD correctly represent the pass and fail domains?* Both the ADFD and ADFD+ techniques generate graphs to represent failure-domains for simplicity. To answer the question we compared the generated graphs with the source code and the invariants generated by Daikon.
7) *If obtained results consistent with previous theoretical and practical results presented?* As per our knowledge, till now no specific study has been conducted to automatically identify the pass and fail domains however it has been claimed by some researchers [] that there exist more block and strip patterns then the point patterns.
8) *If the presence of a particular failure-domain can make it easy or hard to find using automated and manual techniques?* Failure-domain can reside in the form of point, block or strip shape in the input domain. To answer this question we analysed the source code of all the programs in which failure-domains were detected.

VI. Experimental setup

We extracted main "jar" file of each of the 106 packages in Qualitas Corpus and tested all the classes individually using ADFD and ADFD+ strategies. The machine took approximately 100 hours to process the experiments. We found 57 faulty classes from 25 packages out of the total 4500 classes from 106 packages. Only the failing one and two dimensional methods with arguments (int, long, float, double and short) were taken in to consideration. This is because at the moment ADFD and ADFD+ are not capable of drawing/handling more than two dimensions. 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$^{\text{TM}}$SE Runtime Environment [version 1.6.0_35]. The ADFD and ADFD+ executable files are available at https://code.google.com/p/yeti-test/downloads/list/.

VII. Results

Results of the experiments are given in Table 1, 2, 3 and 4. The 85 classes contain strip failure-domain as shown in Table 1. The 4 classes contain point failure-domain as shown in Table 2. The 2 classes contain block failure domain as shown in Table 3. The 2 classes contain two types of failure-domains i.e. AnnotationValue with both point and block failure-domain and Token with point and Strip failure-domain as shown in Table 4.

VIII. Experimental results

IX. Threats to validity

X. Related Work

XI. Conclusion

XII. Future Work

Acknowledgments. The authors are thankful to the Department of Computer Science, University of York for physical and financial support. Thanks are also extended to Prof. Richard Paige for his valuable guidance, help and generous support.

References

[1] F. Chan, T. Y. Chen, I. Mak, and Y.-T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[2] T. Y. Chen, "Adaptive random testing," *Eighth International Conference on Qualify Software*, vol. 0, p. 443, 2008.

[3] B. Beizer, *Software testing techniques (2nd ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990.

[4] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in. net with feedback-directed random testing," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 87–96.

[5] M. A. Ahmad and M. Oriol, "Automated discovery of failure domain," *Lecture Notes on Software Engineering*, vol. 02, no. 4, pp. 331–336, 2014.

[6] ——, "Automated discovery of failure domain," *Lecture Notes on Software Engineering*, vol. 03, no. 1, pp. 289–294, 2013.

[7] M. Oriol. (2011) York extensible testing infrastructure. Department of Computer Science, The University of York. [Online]. Available: http://www.yetitest.org/

[8] 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*, vol. 69, no. 1, pp. 35–45, 2007.

[9] D. Gilbert, "The jfreechart class library version 1.0. 9," 2008.

[10] A. Leitner, I. Ciupa, B. Meyer, and M. Howard, "Reconciling manual and automated testing: The autotest experience," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, ser. HICSS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 261a–. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2007.462

[11] 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*, april 2012, pp. 201 –210.

TABLE I
TABLE DEPICTING CLASSES WITH STRIP FAILURE-DOMAINS FOUND BY ADFD AND ADFD+ AND MANUAL TESTING

S#	Class	Method	ADFD+	ADFD	Manual
1	LeadPipeInputStream	LeadPipeInputStream(i)	I >= 2147483140 I <= 2147483647	I	I > 698000000

2	BitSet	BitSet.of(i,j)	I <= -1, I >= -18,
			J <= 7, J >= -12
				I one of {-513, -1}
				J one of {-503, 507}
					I <= -1
					J != 0

3	ToolPallete	ToolPalette(i,j)	I <= -1, I >= -18	I one of { -515, -1}
				J one of {-509, 501}
					I <= -1,
					J any value

4	IntMap	idMap(i)	I != 0, I <= -1, I >= -18	I one of {-1, -512}	I <= -1
5	ExpressionFactory	expressionOfType(i)	I <= 13, I >= -7	I one of {-497, 513}	I >= -2147483648 I <= 2147483647
6	ArrayStack	ArrayStack(i)	I >= 2147483636 I <= 2147483647	I one of {2147483142, 2147483647}	I > 698000000
7	BinaryHeap	BinaryHeap(i)	I <= -2147483637 I >= -2147483648	I one of {-2147483648, -2147483142}	I <= 0
8	BondedFifoBuffer	BoundedFifoBuffer(i)	I <= -2147483639 I >= -2147483648	I one of {-505, 0}	I <= 0
9	FastArrayList	FastArrayList(i)	I <= -2147483641 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
10	StaticBucketMap	StaticBucketMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
11	PriorityBuffer	PriorityBuffer(i)	I != 0, I <= -1, I >= -14	I one of {-2147483647, -2147483142}	I <= 0

12	GenericPermuting	permutation(i,j)	I <= 0, I >= -18	I one of { -498, 0}
				I one of {2, 512}
					I <= 0, I >= 2
					J != 0

13	LongArrayList	LongArrayList(i)	I <= -2147483640 I >= -2147483648	I one of {-510, -1}	I <= -1
14	OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	I <= -1, I >= -17	I one of {-514, -1}	I <= -1
15	ByteVector	ByteVector(i)	I <= -2147483639 I >= -2147483648	I one of {-2147483648, -2147483141}	I <= -1
16	ElementFactory	newConstantCollection(i)	I >= 2147483636 I <= 2147483647	I one of {2147483141, 2147483647}	I > 698000000
17	IntIntMap	IntIntMap(i)	I <= -2147483638 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
18	ObjectIntMap	ObjectIntMap(i)	I >= 2147483640 I <= 2147483647	I one of {2147483591, 2147483647}	I > 698000000
	IntObjectMap	IntObjectMap(i)	I <= -1, I >= -17	I <= -1, I >= -518	I <= -1

19	ArchiveUtils	padTo(i,j)	I >= 2147483641
			I <= 2147483647
				I one of {-497, 513}
				J one of {2147483591, 2147483647}
					I any value
					J > 698000000

20	BloomFilter32bit	BloomFilter32bit(i,j)	I <= -1 I >= -18	I one of {-515, -1} J may be any value	I <-1 J <-1
21	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647} I >= -518	I > 698000000
22	ObjectCacheHashMap	ObjectCacheHashMap(i)	I <= -2147483641 I >= -2147483648	I one of {-512, 0}	I <= 0
23	ObjToIntMap	ObjToIntMap(i)	I <= -2147483636 I >= -2147483648	I one of {-2147483646, -2147483137}	I <= -1

24	PRTokeniser	isDelimiterWhitespace(i)	I <= -2
			I >= -18
				I one of {-509, -2}
				I one of {256, 501}
					I <= -2
					I >= 256

25	PdfAction	PdfAction(i)	I <= -2147483640
			I >= -2147483648
				I one of {-514, 0}
				I one of {6, 496}
					I <= 0
					I >= 6

26	PdfLiteral	PdfLiteral(i)	I <= -1, I >= -14	I one of {-511, -1}	I <= -1
27	PhysicalEnvironment	PhysicalEnvironment(i)	I <= -1, I >= -11	I one of {-2147483646, -2147483137}	I <= -1
28	IntegerArray	IntegerArray(i)	I >= 2147483636 I <= 2147483647	I one of {2147483587, 2147483647}	I > 698000000
29	AttributeMap	AttributeMap(i)	I <= -2147483639 I >= -2147483648	I one of {-514, 0}	I <= 0
30	ByteList	ByteList(i)	I <= -1, I >= -14	I one of {-513, -1}	I <= -1
31	WeakIdentityHashMap	WeakIdentityHashMap(i)	I >= 2147483636 I <= 2147483647	I one of {2147483140, 2147483647}	I >698000000

32	AmmoType	getMunitionsFor(i)	I <= -1
			I >= -17
				I one of {-514, -1}
				I one of {93, 496}
					I <= -1
					I >= 93

33	IntList	IntList(i,j)	I <= -1 I >= -15	I one of {-1, -509} j one of 0	I <= -1 j = 0
34	QMC	halton(i,j)	I <= -1, I >= -12 J <= -1, J >= -15	I <= -1, I >= -508 j <= 499, j >= -511	I <= -1 J any value
35	BenchmarkFramework	BenchmarkFramework(i,j)	I != 0, I <= -1, I >= -13	I one of {-501, -1}	I <= -1
36	IntArray	IntArray(i)	I <= -1, I >= -16	I one of {-2147483650, -2147483141}	I <= -1
37	TDoubleStack	TDoubleStack(i)	I <= -1, I >= -13	I one of {-511, -1}	I <= -1
38	TIntStack	TIntStack(i)	I <= -1, I >= -12	I one of {-2147483648, -2147483144}	I <= -1
39	TLongArrayList	TLongArrayList(i)	I <= -1, I >= -16	I one of {-2147483648, -2147483141}	I <= -1
40	AlgVector	AlgVector(i)	I <= -1, I >= -15	I one of {-511, -1}	I <= -1
41	BinarySparseInstance	BinarySparseInstance(i)	I <= -1, I >= -15	I one of {-506, -1}	I <= -1
42	SoftReferenceSymbolTable	SoftReferenceSymbolTable(i)	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
43	SymbolHash	SymbolHash(i)	I <= -1, I >= -16	I one of {-2147483648, -2147483592}	I <= -1
44	SynchronizedSymbolTable	SynchronizedSymbolTable(i)	I <= -2147483140 I >= -2147483648	I one of {-2147483648, -2147483592}	I <= -1
45	XMLChar	isSpace(i)	I != 0, I <= -1, I >= -12	I one of {-510, -1}	I <= -1
46	XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	I != 0, I <= -1, I >= -13	I one of {-2147483648, -2147483137}	I <= -1
47	XML11Char	isXML11NCNameStart(i)	I <= -1, I >= -16	I one of { -512, -1}	I <= -1
48	AttributeList	AttributeList(i)	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647}	I > 698000000

TABLE II
Table depicting classes with point failure-domains found by ADFD and ADFD+ and Manual testing

S#	Class	Method	ADFD+	ADFD	Manual
1	Assert	assertEquals(i,j)	I != J	I != J	I != J
2	Board	getTypeName(i)	I <= -1 I >= -18	I >= -504, I <= -405, I >= -403 I <= -304, I >= -302, I <= -203 I >= -201, I <= -102, I >= -100 I <= -1	I <= -910, I >= -908, I <= -809, I >= -807, I <= -708, I >= -706, I <= -607, I >= -605, I <= -506, I >= -504, I <= -405, I >= -403, I <= -304, I >= -302, I <= -203, I >= -201, I <= -102, I >= -100, I <= -1
3	HTMLEntities	get(i)	I <=- 1 I >= -17	I >= -504, I <= -405, I >= -403 I <= -304, I >= -302, I <= -203 I >= -201, I <= -102, I >= -100 I <= -1	I <= -910, I >= -908, I >= -807, I <= -708, I >= -706, I <= -809, I <= -607, I >= -605, I <= -506, I >= -504, I <= -405, I >= -403, I <= -304, I >= -302, I <= -203, I >= -201, I <= -102, I >= -100, I <= -1
4	Assert	assertEquals(i,j)	I <= 0, I >= 20 J <= 18, j >= -2	I one of {-2147483648, -2147483142} J one of {-501, 509}	I != J

TABLE III
Table depicting classes with block failure-domains found by ADFD and ADFD+ and Manual testing

S#	Class	Method	ADFD+	ADFD	Manual
1	AnnotationValue	whatKindIsThis(i)	I <= 85, I >= 92, I >= 98 I = 100, I >= 102, I <= 104	I ¡= 63, I = {65, 69, 71, 72} I >= 75, I ¡= 82, I >= 84 I <= 89, I >= 92, I ¡= 98 I = 100, I >= 102, I <= 114 I >= 116	I <= 63, I = 65, 69, 71, 72 I >= 75, I <= 82, I >= 84 I <= 89, I >= 92, I <= 98 I = 100, I >= 102, I <= 114 I >= 116
2	Token	typeToName(i)	I <= -2147483641 I >= -2147483648	I one of {-510, -2} I = {73, 156} I one of {162, 500}	I <= -2, I = 73, 156, I >= 162

TABLE IV
Table depicting classes with mix failure-domains found by ADFD and ADFD+ and Manual testing

S#	Class	Method	ADFD	ADFD+	Manual
1	ClassLoaderResolver	getCallerClass(i)	I >= 2, I <= 18	I >= 500, I <= -2 I >= 2, I <= 505	I <= -2, I >= 2
2	Variant	getVariantLength(i)	I >=0, I <= 12	I >= 0, I <= 14, I >= 16 I <= 31, I >= 64, I <= 72	I >= 0, I <= 14, I >= 16 I <= 31, I >= 64, I <= 72