Finding the Effectiveness of ADFD and ADFD+

Mian Asbat Ahmad and Manuel Oriol

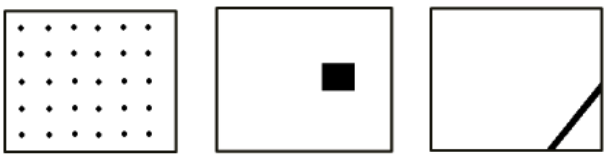
University of York, Department of Computer Science,
Deramore Lane, YO10 5GH YORK, United Kingdom

Abstract. The achievement of up-to 50% better results by Adaptive Random Testing (ART) verses Random Testing (RT) ensures that the pass and fail domains in the input domain are 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.
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. The impact of nature, location, size, type and complexity of failure-domains on the testing techniques were studied. The results provide insights into the effectiveness of automated techniques and a number of lessons for testing researchers and practitioners.

Keywords: software testing, automated random testing, manual testing, ADFD, Daikon

1 Introduction

The input-domain of a given SUT can be divided into two sub-domains. The pass-domain comprises of the values for which the software behaves correctly and the failure-domain comprises of the values for which the software behaves incorrectly. Chan et al. [1] observed that input inducing failures are contiguous and form certain geometrical shapes. They divided these into point, block and strip failure-domains as shown in Figure 1. Adaptive Random Testing achieved up to 50% better performance than random testing by taking into consideration the presence of failure-domains while selecting the test input [2].



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

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

The cost of software testing constitute about half of the total cost of software development. Software testing is an expensive but essential process which is particularly time consuming, laborious and error-prone when performed manually. Alternatively, automated software testing may involve higher initial cost but brings the key benefits of lower cost of production, higher productivity, maximum availability, greater reliability, better performance and ultimately proves highly beneficial for any organisation [3]. A case study conducted by Pacheco et al. 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 also provide visualisation of the pass and fail domains. The process is accomplished in two steps. In the first step, an upgraded random testing is used to find the failure. In the second step, exhaustive testing is performed in a limited region around the detected failure in order to identify the domains. The ADFD searches in one-dimension and covers longer range than ADFD+ which is more effective in multi-dimension and covers shorter range.

Three separate tools including YETI, Daikon and JFreeChart have been used in combination to develop ADFD and ADFD+ techniques. The York Extensible Testing Infrastructure [7] is used to test the program automatically with ADFD or ADFD+ strategy. The Daikon [8] checks all test executions and automatically generates invariants to present failure-domains quantitatively. The JFreeChart [9] facilitates graphical representation of the pass and fail domains.

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.

2 Enhancement of the techniques

Prior to conducting the experiments for comparative evaluation, the ADFD and ADFD+ techniques were enhanced to increase the code coverage, provide information about the identified failure and generate invariants of the detected failure-domains as stated below.

1. Code coverage was increased by extending the techniques to support the testing of methods with byte, short, long, double and float arguments while it was restricted to int type arguments only in the original techniques.
2. Additional information was facilitated by adding the YETI generated test case to the GUI of the two techniques. Test case includes the name of the failing method, values that caused the failure and stack trace of the failure.
3. Invariants of the detected failure-domains were automatically generated by integrating the tool Daikon in the two techniques. Daikon is an automated

invariant detector that detects likely invariants in the program [8]. The generated invariants are displayed in the GUI of the techniques after completion of the test.

3 Difference in working mechanism of the two techniques

The difference with respect to the identification of failure-domains is illustrated by testing a simple Java program (given below) with ADFD and ADFD+ techniques.

```
/**
* A program with block failure-domain.
* @author (Mian and Manuel)
*/
public class BlockErrorPlotTwoShort {
    public static void blockErrorPlot (int x, int y){
        int z;
        if ((x >= 4) && (x <= 8) && (y == 2))
            { z = 50/0;}
        if ((x >= 5) && (x <= 8) && (y == 3))
            { z = 50/0;}
        if ((x >= 6) && (x <= 8) && (y == 4))
            { z = 50/0;}
    }
}
```

As evident from the program code that an *ArithmeticException* failure (divison by zero) is generated when the value of variable `x one of {4, 5, 6, 7, 8}` and the corresponding value of variable `y one of {2, 3, 4}`. The values form a block failure-domain in the input domain.

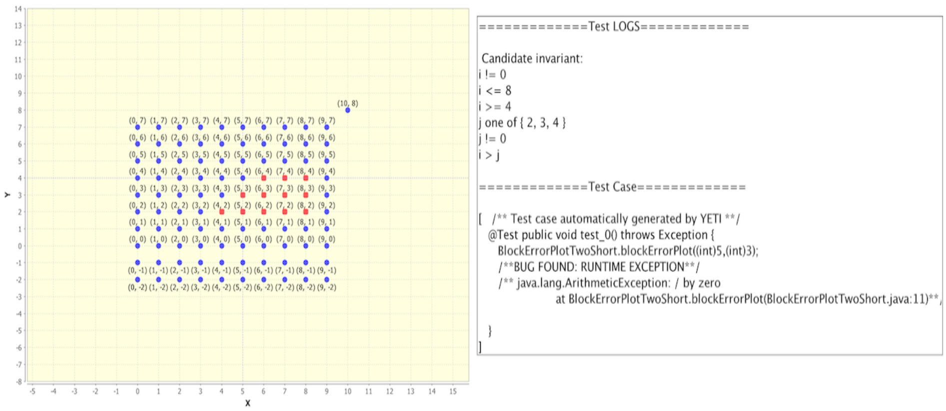


Fig. 2. Graph, Invariants and Test case generated by ADFD+

The test output generated by ADFD+ technique is presented in Figure 2. The labelled graph correctly shows all the 12/12 available failing values in red whereas the passing values are shown in blue. The invariants correctly represent the failure-domain. The test case shows the type of failure, the values causing the first failure and the stack trace of the failure.

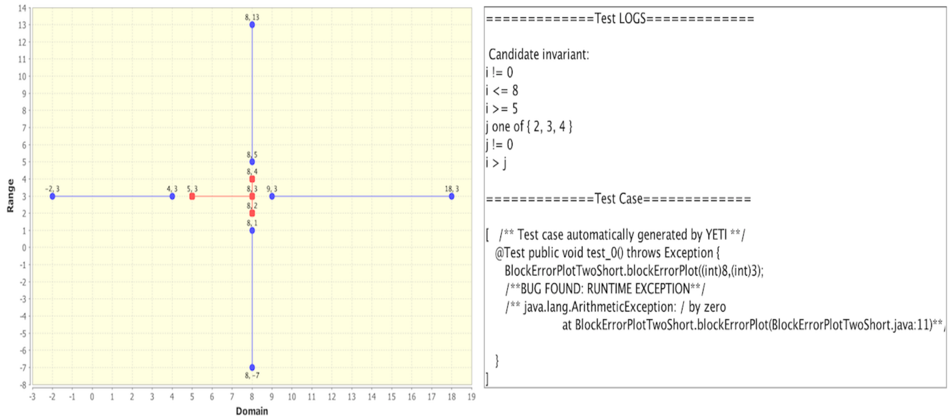


Fig. 3. Graph, Invariants and test case generated by ADFD

The test output generated by ADFD technique is presented in Figure 3. The labelled graph correctly shows the 4/12 available failing values in red whereas the passing values are shown in blue. The invariants identify all but one failing values ($x = 4$). This is due to the fact that ADFD scans the values in one dimension around the failure. The test case shows the type of failure, the values causing the first failure and the stack trace of the failure.

The comparative results derived from the execution of the two techniques on the selected program indicate that ADFD+ is more efficient than ADFD in identification of failures in two dimensional program. ADFD and ADFD+ performs equally well in one-dimensional program but ADFD covers more range around the first failure than ADFD+ and is comparatively economical because it uses less resources than ADFD+.

3.1 Research questions

The following research questions have been addressed in the study:

1. Can ADFD and ADFD+ techniques identify and present failure-domains in production software?

2. What types and frequencies of failure-domains exist in production software?

3. Is the failure-domain caused by simple or complex error and how it influences the testing?

4. What is the external validity of the results obtained?

4 Evaluation

Experimental evaluation of ADFD and ADFD+ techniques was carried out to determine: the effectiveness of the techniques in identifying and presenting the failure-domains, the types and frequencies of failure-domains, the nature of error causing failure-domain and the external validity of the results obtained.

4.1 Experiments

In the present experiments we tested all 106 packages of Qualitas Corpus containing the total of 4500 classes. Qualitas Corpus was selected because it is a database of Java programs that spans across the whole set of Java applications, it is specially built for empirical research which takes into account a large number of developmental models and programming styles and it includes all packages that are open source with easy access to the source code.

Since YETI tests the byte code only therefore the main ".jar" file of each package was extracted to get the ".class" files. Each class was individually tested. The one and two dimensional methods with arguments (int, long, float, byte, double and short) of each class were selected for experimental testing. Non numerical arguments and more than two dimensional methods were ignored because the two proposed techniques support the one and two dimensional methods with numerical arguments. Each test took 40 seconds on the average to complete the execution. The initial 5 seconds were used by YETI to find the first failure while the remaining 35 seconds were jointly consumed by ADFD/ADFD+ technique, JFreeChart and Daikon to identify, draw graph and generate invariants of the failure-domains respectively. The machine took approximately 100 hours to perform the experiments. 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][10]. The source code of the programs containing failure-domains were also evaluated manually to cross-examine the experimental results. 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™SE Runtime Environment [version 1.7.0_45]. The ADFD and ADFD+ executable files are available at `https://code.google.com/p/yeti-test/downloads/list/`.

4.2 Results

Among 106 packages we found 25 packages containing 57 classes with different types of failure-domains. Based on the type of failure-domains the results are presented in table 1, 2, 3, 4. The information available in the table includes the class showing failure domain, the method involved, the invariants generated by ADFD and ADFD+ (automatic techniques) and by manual analysis.

Classification of failure-domains into strip, point, block and mix types is based on the degree of contiguity of failures detected in the input-domain. If failures detected as contiguous are 50 or more, the failure-domain is classified as strip. If failures detected as contiguous lie in the range of 6 to 49, the failure domain is classified as block. If failures detected as contiguous lie in the range of 1 to 5, the failure domain is classified as point. If more than one type of failure domains are detected in the input domain, the domain is classified as mix.

The results obtained show that out of 57 classes 48 contain strip failure domain, 4 contain point failure domain, 2 contain block failure domain and 2 contain mix failure domain.

4.3 Effectiveness of ADFD and ADFD+ techniques

Experimental results indicate that both ADFD and ADFD+ techniques were effective in identifying and presenting all three types including point, block and strip failure-domains. The correctness of identification and presentation of failure-domains is checked by three different methods: (1) Manual Analysis of the source code of all the 57 classes containing failure-domains was performed to check the existence of underlying failure-domain. (2) The test case, graph and invariants generated by the two techniques were compared with each other for any discrepancy. (3) The invariants generated by ADFD, ADFD+ and manual techniques were compared with each other.

The value of range plays an important role in the identification and presentation of failure-domain because both the generation of graph and invariants starts from failing value and stops at range value. Since the range value of ADFD+ is less as compared to the ADFD therefore ADFD presented the failure domain effectively than ADFD+.

For example consider the following code under test. If the range value of ADFD is from -100 to 100 and the range value for ADFD+ is from -10 to 10 then the invariants generated to represent the failure domain by ADFD will be $ioneof\{-1, -100\}$ while for ADFD+ they will be $ioneof\{-1, -10\}$. Similarly the invariants generated to represent the failure-domain manually will be $i <= -1$. The presentation can be further improved if the value of range is extended to Integer.MIN_INT and Integer.MAX_INT .

```
/**
* A program with strip failure-domain.
* @author (Mian and Manuel)
*/
```

```
public class StripErrorPlot {
    public static void stripErrorPlot (int x){
        int a[] = new int[x];
    }
}
```

With all the effectiveness of automated techniques we still believe that ADFD and ADFD+ cannot be used as replacement of manual testing however it should be used to assist the manual testing for achieving higher quality.

4.4 Type and Frequency of Failure-domains

The ADFD, ADFD+ and Manual techniques discovered all three types of failure-domains. The results show that the frequency of strip failure domain is highest than point and block failure-domains. The results also show that the frequency of point failure-domain is more than block failure domains. Finally in two classes we found mix failure-domains.

It may be noted that YETI, which is used to find the first failure, is executed only for five seconds which uses ADFD and ADFD+ testing strategies. Both the strategies are based on random+ strategy which gives high priority to boundary values. It may be possible that the high number of strip failure-domains are detected because most of the failures are found at the boundaries.

4.5 Nature of failure-domains

To study the nature of failure-domains discovered by ADFD and ADFD+ techniques we performed manual analysis of the code of each faulty class and generated the invariants. On comparison we found 3 cases. i.e easy to find, hard to find and impossible to find.

Easy to find were those in which negativearraysizeexceptin. hard to find were those like IndexArrayOutOfBoundsException. Impossible to find were those in which finding failure is easy but finding the cut over point is very difficult. like OutOfMemoryError.

4.6 External validity of Results

The external validity is the degree to which the subject packages are representative of true practice. We have evaluated our approach by testing all the packages in Qualitas Corpus to minimize any threat to external validity. The Qualitas Corpus contains packages of different functionality, size, maturity and modification histories suggesting generalization of the the experimental findings.

5 Threats to validity

As in the case of any experimental study, the results are considered conclusive only if the programs tested represent general behaviour. We have tried to minimize this threat by choosing all the classes from all the projects included in the purpose built repository of Qualitas Corpus.

YETI with ADFD and ADFD+ strategies selected is executed only for 5 seconds to find a failure in the SUT. Since both the test strategies are based on random+ strategies which have high precedence for boundary values therefore most of the failures detected are boundary failures. Despite the fact that the failure-domains remained the same for a specific failure detected, It is likely possible that increasing the test duration may lead to the discovery of new failures exhibiting different behaviour.

Another threat to validity may be related to hardware and software resources. For example the OutOfMemoryError occured at the value of 6980000 on the machine executing the test. On another machine the failure revealing value can increase or decrease depending on the hardware specifications and system load.

The errors due to strings etc were ignored because they cannot be drawn on the graph. Therefore the results may reflect less number of failures.

6 Related Work

7 Conclusion

8 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.

9 The References Section

1. Chan, F., Chen, T.Y., Mak, I., Yu, Y.T.: Proportional sampling strategy: guidelines for software testing practitioners. Information and Software Technology **38**(12) (1996) 775–782
2. Chen, T.Y.: Adaptive random testing. Eighth International Conference on Qualify Software **0** (2008) 443
3. Beizer, B.: Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA (1990)
4. Pacheco, C., Lahiri, S.K., Ball, T.: Finding errors in. net with feedback-directed random testing. In: Proceedings of the 2008 international symposium on Software testing and analysis, ACM (2008) 87–96
5. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **02**(4) (2014) 331–336

6. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **03**(1) (2013) 289–294
7. Oriol, M.: York extensible testing infrastructure (2011)
8. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Science of Computer Programming **69**(1) (2007) 35–45
9. Gilbert, D.: The jfreechart class library version 1.0. 9. (2008)
10. 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, IEEE (2012) 201–210

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 <= -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 <= -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 <= -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 <= -1, I >= -12	I one of {-510, -1}	I <= -1
46	XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	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 1. Classes with strip failure-domains

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 2. Classes with point failure-domains

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 3. Classes with block failure-domains

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

Table 4. Classes with mix failure-domains