Finding the Effectiveness of ADFD and ADFD+

Mian Asbat Ahmad and Manuel Oriol

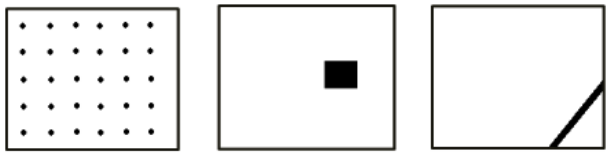
University of York, Department of Computer Science,
Deramore Lane, YO10 5GH YORK, United Kingdom

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. [**?**] 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 [**?**].



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

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

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 [**?**]. 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 [**?**].

We have developed two fully automated techniques ADFD [**?**] and ADFD+ [**?**], 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 [**?**] is used to test the program automatically with ADFD or ADFD+ strategy. The Daikon [**?**] checks all test executions and automatically generates invariants to present failure-domains quantitatively. The JFreeChart [**?**] 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 [**?**]. 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.

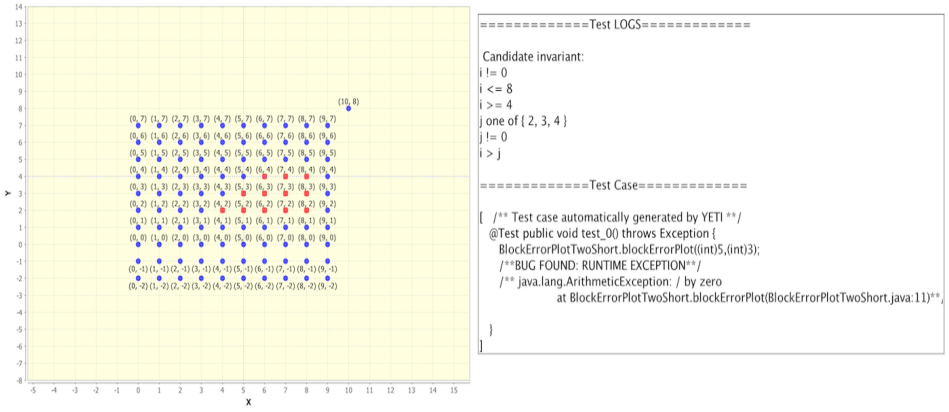


Fig. 2. Graph, Invariants and Test case generated by ADFD+

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.

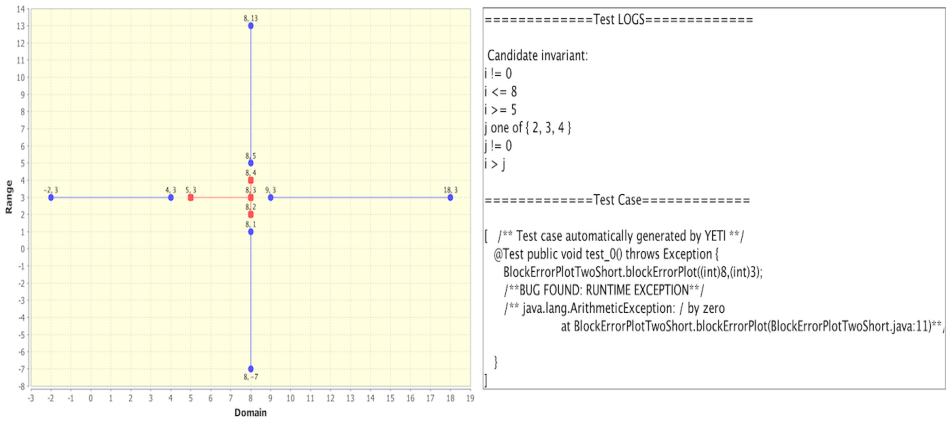


Fig. 3. Graph, Invariants and test case generated by ADFD

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. What is the nature of identified failure-domain and how it affects the testing techniques?

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 [**?**][**?**]. 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^{TM}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 2, 3, 4, 5. 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.

Effectiveness of ADFD and ADFD+ techniques: The effectiveness of ADFD and ADFD+ techniques for identifying failure-domains in production software was demonstrated. The experimental results confirmed the effectiveness of the techniques by discovering all three types of failure-domains (point, block and strip) across the input domain. The results obtained by applying the two automated techniques were verified: by manual analysis of the source code of all 57 classes containing failure domains; by cross checking the test case, the graph and the generated invariants of each class; by comparing the invariants generated by automatic and manual techniques.

The identification of failure domain by both ADFD and ADFD+ is dependant on the identification of failure by ADFD and ADFD+ strategy in YETI. Because only after a failure is identified, its neighbour values according to the set range are analysed and failure domain of the failure is plotted.

The generation of graph and invariants depends on range value, the greater the range value of a technique the better is the presentation of failure domain. The generation of graph and invariants starts from the minimum range value and ends at the maximum range value around the detected failure value. The ADFD requires less resources and is thus capable of handling greater range value as compared to ADFD+.

Type and Frequency of Failure-domains: As evident from the results given in table 1 - 4, all the three techniques (ADFD, ADFD+ and Manual) detected the presence of strip, point and block types of failure domains in different frequencies. Out of 57 classes containing failure domains, 47 classes showed strip failure domain, 4 point failure domain, 2 point failure domain and 2 mix failure domains.

The discovery of higher number of strip type of failure domains may be attributed to the fact that a limited time of 5 seconds were set in YETI testing tool for searching the first failure. The ADFD and ADFD+ strategies set in YETI for testing the classes are based on random+ strategy which gives high priority to boundary values, therefore the search by YETI was prioritised to the boundary area where there were greater chances of occurrence of failures constituting strip type of failure domain.

Nature of failure-domain: The nature of failure domain as identified by automatic techniques (ADFD and ADFD+) and Manual technique was examined in terms of simplicity and complexity by comparing the invariants generated by the

automatic techniques with the manual technique. The results indicated that x1 failure domains were simple in nature for automated techniques as against x2 for manual technique while y1 failure domains were simple in nature for automated technique as against y2 for manual technique.

The analysis of generated invariants indicated that the failure domains which are simple in nature are easily detectable by both automated and manual techniques irrespective of the type of failure domain (Strip, point, block). It was further indicated that the failure domains which are complex in nature are difficultly detectable by both automated and manual techniques. Both types are explained with the help of following examples.

As an example we consider the results of ADFD, ADFD+ and Manual Analysis in Table 1 for class BitSet. The negativeArray failure is detected due to the input of negative value to the method bitSet.of(i). The invariants generated by ADFD are $\{i <= -1, i >= -18\}$, by ADFD+ are $\{i <= -1, i >= -512\}$ and by Manual Analysis are $\{i <= -1, i >= Integer.MIN\_INT\}$. These results indicate maximum degree of representation of failure-domain by Manual Analysis followed by ADFD and ADFD+ respectively.

As an example we consider the results of ADFD, ADFD+ and Manual Analysis in Table 1 for class ArrayStack. The OutOfMemoryError failure is detected due to the input of value to the method ArrayStack(i). The invariants generated by ADFD are $\{i >= 2147483636, I <= 2147483647\}$, by ADFD+ are $\{i >= 2147483142, i <= 2147483647\}$, by Manual analysis $\{i >= 698000000\}$.

5 Threats to validity

All packages in Qualitas Corpus were tested by ADFD, ADFD+ and Manual techniques to minimize in order to minimize the threats to external validity. The Qualitas Corpus contains packages of different: functionality, size, maturity and modification histories.

YETI using ADFD/ADFD+ strategy was executed only for 5 seconds to find the first failure in the given SUT. Since both ADFD and ADFD+ strategies are based on random+ strategy having high preference for boundary values therefore most of the failures detected are from the boundaries of input domain. It is quite possible that increasing the test duration of YETI may lead to the discovery of new failures with different failure domain.

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

In previous work, researchers have done some work to study the shape and location of the failure-domain in the input domain. According to White et al. [?] the boundary values located at the edge of domains have more chances of forming strip failure domain. Finelly [?] and Bishop [?] found that failure causing inputs form a continuous region inside the input domain. Chan et al. reveal that failure causing values form certain geometrical shapes in the input domain, they classified the failure-domains into point, block and strip failure domains [?].

Random testing is quick in execution and experimentally proven to detect errors in programs of various platforms including Windows [?], Unix16, Java Libraries citepacheco2005eclat, Heskell [?] and Mac OS [?]. Its ability to become fully automated makes it one of the best choice for automated testing tools [?][?]. AutoTest [?], Jcrasher [?], Eclat [?], Jartege [?], Randoop [?] and YETI [?][?][?] are few of the most common automated random testing tools used by research community. YETI is loosely coupled, highly flexible and allows easy extensibility as reported previously [?].

Our previous studies ADFD [?] and ADFD+ [?] describes fully automated techniques for the discovery of failure domains and evaluate it experimentally. The programs used in evaluation were error-seeded one and two dimensional programs. This work is a direct continuation of our previous work to further contributes to this line of research by extending the techniques with support of Daikon, manual analysis and testing of production software from Qualitas Corpus.

A common practice to evaluate the effectiveness of an extended technique is to compare the results obtained by applying the new and existing techniques to identical programs [?][?]. Arcuri et al. [?], stresses on the use of random testing as a baseline for comparison with other testing techniques. We followed the procedure and evaluated ADFD, ADFD+ and Manual testing under identical conditions.

7 Conclusion

Failures within the input domain are contiguous and form point, block and strip failure-domains. Existing automated testing tools, such as JCrasher and Jartege, search for individual failure ignoring the failure-domain. We have developed ADFD and ADFD+ techniques for identification of failure-domains and its presentation by graph and invariants. We have conducted automated and manual experiments that evaluate the effectiveness of our techniques on detecting and presenting the failure-domains in production software contained in Qualitas Corpus. The results show that the two techniques can effectively identify and present the failure-domains to certain degree of accuracy. We further explain how the degree of accuracy can be increased in ADFD and ADFD+ techniques.

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.

Table 1. Table depicting results of ADFD and ADFD+

S#	Project	Class	Method	Dim	LOC	Failure-domain
1	ant	LeadPipeInputStream	LeadPipeInputStream(i)	one		Strip
2	antlr	BitSet	BitSet.of(i,j)	two		Strip
3	artofillusion	ToolPallete	ToolPalette(i,j)	two		Strip
4	aspectj	AnnotationValue	whatKindIsThis(i)	one		
		IntMap	idMap(i)	one		Strip
5	cayenne	ExpressionFactory	expressionOfType(i)	one		Strip
6	collections	ArrayStack	ArrayStack(i)	one		Strip
		BinaryHeap	BinaryHeap(i)	one		Strip
		BondedFifoBuffer	BoundedFifoBuffer(i)	one		Strip
		FastArrayList	FastArrayList(i)	one		Strip
		StaticBucketMap	StaticBucketMap(i)	one		Strip
		PriorityBuffer	PriorityBuffer(i)	one		Strip
7	colt	GenericPermuting	permutation(i,j)	two		Strip
		LongArrayList	LongArrayList(i)	one		Strip
		OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	one		Strip
8	drjava	Assert	assertEquals(i,j)	two		
		ByteVector	ByteVector(i)	one		Strip
9	emma	ClassLoaderResolver	getCallerClass(i)	one		
		ElementFactory	newConstantCollection(i)	one		Strip
		IntIntMap	IntIntMap(i)	one		Strip
		ObjectIntMap	ObjectIntMap(i)	one		Strip
		IntObjectMap	IntObjectMap(i)	one		Strip
10	heritrix	ArchiveUtils	padTo(i,j)	two		Strip
		BloomFilter32bit	BloomFilter32bit(i,j)	two		Strip
11	hsqld	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	one		Strip
		ObjectCacheHashMap	ObjectCacheHashMap(i)	one		Strip
12	htmlunit	ObjToIntMap	ObjToIntMap(i)	one		Strip
		Token	typeToName(i)	one		
13	itext	PRTokeniser	isDelimiterWhitespace(i)	one		Strip
		PdfAction	PdfAction(i)	one		Strip
		PdfLiteral	PdfLiteral(i)	one		Strip
14	jung	PhysicalEnvironment	PhysicalEnvironment(i)	one		Strip
15	jedit	IntegerArray	IntegerArray(i)	one		Strip
16	jgraph	AttributeMap	AttributeMap(i)	one		Strip
17	jruby	ByteList	ByteList(i)	one		Strip
		WeakIdentityHashMap	WeakIdentityHashMap(i)	one		Strip
18	junit	Assert	assertEquals(i,j)	two		
19	megamek	AmmoType	getMunitionsFor(i)	one		Strip
		Board	getTypeName(i)	one		Strip
20	nekohtml	HTMLEntities	get(i)	one		
21	poi	Variant	getVariantLength(i)	one		
		IntList	IntList(i,j)	two		Strip
22	sunflow	QMC	halton(i,j)	two		Strip
		BenchmarkFramework	BenchmarkFramework(i,j)	two		Strip
		IntArray	IntArray(i)	one		Strip
23	trove	TDoubleStack	TDoubleStack(i)	one		Strip
		TIntStack	TIntStack(i)	one		Strip
		TLongArrayList	TLongArrayList(i)	one		Strip
24	weka	AlgVector	AlgVector(i)	one		Strip
		BinarySparseInstance	BinarySparseInstance(i)	one		Strip
25	xerces	SoftReferenceSymbolTable	SoftReferenceSymbolTable(i)	one		Strip
		SymbolHash	SymbolHash(i)	one		Strip
		SynchronizedSymbolTable	SynchronizedSymbolTable(i)	one		Strip
		XMLChar	isSpace(i)	one		Strip
		XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	one		Strip
		XML11Char	isXML11NCNameStart(i)	one		Strip
		AttributeList	AttributeList(i)	one		Strip

S#	Class	Invariants by ADFD+	Invariants by ADFD	Invariants by Manual
1	LeadPipeInputStream	I >= 2147483140 I <= 2147483647	I one of {2147483143, 2147483647}	I > 698000000
2	BitSet	I <= -1, I >= -18, J <= 7, J >= -12	I one of {-513, -1} J one of {-503, 507}	I <= -1 J != 0
3	ToolPallete	I <= -1, I >= -18	I one of { -515, -1} J one of {-509, 501}	I <= -1, J any value
4	IntMap	I <= -1, I >= -18	I one of {-1, -512}	I <= -1
5	ExpressionFactory	I <= 13, I >= -7	I one of {-497, 513}	I >= -2147483648 I <= 2147483647
6	ArrayStack	I >= 2147483636 I <= 2147483647	I one of {2147483142, 2147483647}	I > 698000000
7	BinaryHeap	I <= -2147483637 I >= -2147483648	I one of {-2147483648, -2147483142}	I <= 0
8	BondedFifoBuffer	I <= -2147483639 I >= -2147483648	I one of {-505, 0}	I <= 0
9	FastArrayList	I <= -2147483641 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
10	StaticBucketMap	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
11	PriorityBuffer	I <= -1, I >= -14	I one of {-2147483647, -2147483142}	I <= 0
12	GenericPermuting	I <= 0, I >= -18	I one of { -498, 0} I one of {2, 512}	I <= 0, I >= 2 J != 0
13	LongArrayList	I <= -2147483640 I >= -2147483648	I one of {-510, -1}	I <= -1
14	OpenIntDoubleHashMap	I <= -1, I >= -17	I one of {-514, -1}	I <= -1
15	ByteVector	I <= -2147483639 I >= -2147483648	I one of {-2147483648, -2147483141}	I <= -1
16	ElementFactory	I >= 2147483636 I <= 2147483647	I one of {2147483141, 2147483647}	I > 698000000
17	IntIntMap	I <= -2147483638 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
18	ObjectIntMap	I >= 2147483640 I <= 2147483647	I one of {2147483591, 2147483647}	I > 698000000
	IntObjectMap	I <= -1, I >= -17	I <= -1, I >= -518	I <= -1
19	ArchiveUtils	I >= 2147483641 I <= 2147483647	I one of {-497, 513} J one of {2147483591, 2147483647}	I any value J > 698000000
20	BloomFilter32bit	I <= -1 I >= -18	I one of {-515, -1} J may be any value	I <-1 J <-1
21	IntKeyLongValueHashMap	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647} I >= -518	I > 698000000
22	ObjectCacheHashMap	I <= -2147483641 I >= -2147483648	I one of {-512, 0}	I <= 0
23	ObjToIntMap	I <= -2147483636 I >= -2147483648	I one of {-2147483646, -2147483137}	I <= -1
24	PRTokeniser	I <= -2 I >= -18	I one of {-509, -2} I one of {256, 501}	I <= -2 I >= 256
25	PdfAction	I <= -2147483640 I >= -2147483648	I one of {-514, 0} I one of {6, 496}	I <= 0 I >= 6
26	PdfLiteral	I <= -1, I >= -14	I one of {-511, -1}	I <= -1
27	PhysicalEnvironment	I <= -1, I >= -11	I one of {-2147483646, -2147483137}	I <= -1
28	IntegerArray	I >= 2147483636 I <= 2147483647	I one of {2147483587, 2147483647}	I > 698000000
29	AttributeMap	I <= -2147483639 I >= -2147483648	I one of {-514, 0}	I <= 0
30	ByteList	I <= -1, I >= -14	I one of {-513, -1}	I <= -1
31	WeakIdentityHashMap	I >= 2147483636 I <= 2147483647	I one of {2147483140, 2147483647}	I >698000000
32	AmmoType	I <= -1 I >= -17	I one of {-514, -1} I one of {93, 496}	I <= -1 I >= 93
33	IntList	I <= -1 I >= -15	I one of {-1, -509} j one of 0	I <= -1 j = 0
34	QMC	I <= -1, I >= -12 J <= -1, J >= -15	I <= -1, I >= -508 j <= 499, j >= -511	I <= -1 J any value
35	BenchmarkFramework	I <= -1, I >= -13	I one of {-501, -1}	I <= -1
36	IntArray	I <= -1, I >= -16	I one of {-2147483650, -2147483141}	I <= -1
37	TDoubleStack	I <= -1, I >= -13	I one of {-511, -1}	I <= -1
38	TIntStack	I <= -1, I >= -12	I one of {-2147483648, -2147483144}	I <= -1
39	TLongArrayList	I <= -1, I >= -16	I one of {-2147483648, -2147483141}	I <= -1
40	AlgVector	I <= -1, I >= -15	I one of {-511, -1}	I <= -1
41	BinarySparseInstance	I <= -1, I >= -15	I one of {-506, -1}	I <= -1
42	SoftReferenceSymbolTable	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
43	SymbolHash	I <= -1, I >= -16	I one of {-2147483648, -2147483592}	I <= -1
44	SynchronizedSymbolTable	I <= -2147483140 I >= -2147483648	I one of {-2147483648, -2147483592}	I <= -1
45	XMLChar	I <= -1, I >= -12	I one of {-510, -1}	I <= -1
46	XMLGrammarPoolImpl	I <= -1, I >= -13	I one of {-2147483648, -2147483137}	I <= -1
47	XML11Char	I <= -1, I >= -16	I one of { -512, -1}	I <= -1
48	AttributeList	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647}	I > 698000000

Table 2. Classes with strip failure-domains

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

Table 3. Classes with point failure-domains

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

Table 4. Classes with block failure-domains

S#	Class	Invariants by ADFD	Invariants by ADFD+	Invariants by Manual
1	ClassLoaderResolver	I >= 2,	I >= 500, I <= -2	I <= -2, I >= 2
		I <= 18	I >= 2, I <= 505	
2	Variant	I >=0, I <= 12	I >= 0, I <= 14, I >= 16	I >= 0, I <= 14, I >= 16
			I <= 31, I >= 64, I <= 72	I <= 31, I >= 64, I <= 72

Table 5. Classes with mix failure-domains