Finding the Effectiveness of ADFD and ADFD+

Mian Asbat Ahmad and Manuel Oriol

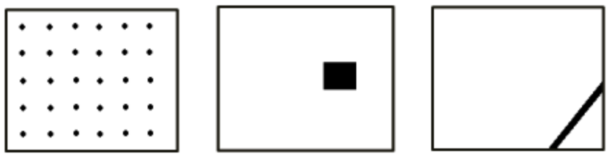
University of York, Department of Computer Science,
Deramore Lane, YO10 5GH YORK, United Kingdom

Abstract. The achievement of up-to 50% better results by Adaptive Random Testing (ART) in comparison with Random Testing (RT) ensures that failure-domains within the input domain are useful and need due consideration in selection of test inputs. Our previously developed two automated techniques (ADFD and ADFD+), automatically find failure and its domain in a specific range and present it graphically.
In this article, 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 [3]. 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 [4]. 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 [5].

We have developed two fully automated techniques ADFD [6] and ADFD+ [7], 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 [8] is used to test the program automatically with ADFD or ADFD+ strategy. The Daikon [9] checks all test executions and automatically generates invariants to present failure-domains quantitatively. The JFreeChart [10] facilitates graphical representation of the pass and fail domains.

The rest of the paper is organized as follows: Section 2 presents enhancement of the techniques. Section 3 shows the difference in working mechanism of the two techniques by a motivating example. Section 4 highlights the key research questions. Section 5 explains the evaluation process, which contain experiments, results and answers of research questions in view of experiment results. Section 6 presents the threats to validity. Section 7 presents related work. Finally, Section 8 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

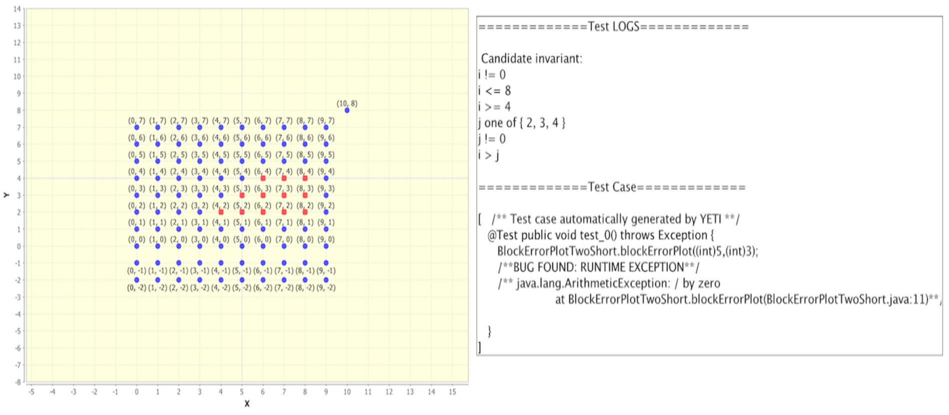
invariant detector that detects likely invariants in the program [9]. The generated invariants are displayed in the GUI of the techniques after completion of the test.

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;}
    }
}
```

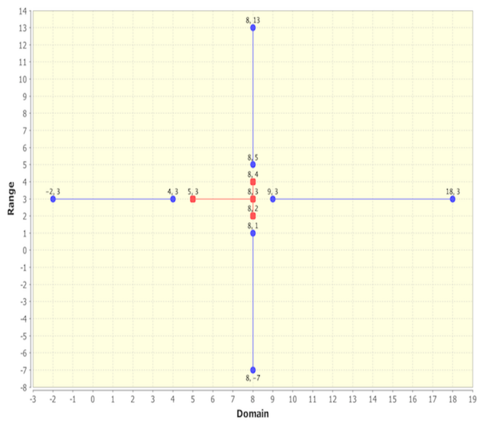
As evident from the program code that an *ArithmeticException* failure (divison by zero) is generated when the value of variable `x one of {4, 5, 6, 7, 8}` and the corresponding value of variable `y one of {2, 3, 4}`. The values form a block failure-domain in the input domain.

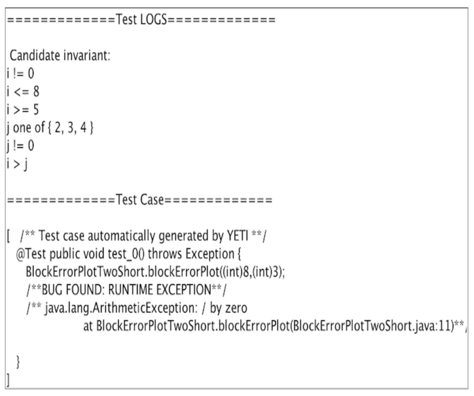


```
=============Test LOGS=============

 Candidate invariant:
i != 0
i <= 8
i >= 4
j one of { 2, 3, 4 }
j != 0
i > j

=============Test Case=============

[   /** Test case automatically generated by YETI **/
   @Test public void test_0() throws Exception {
     BlockErrorPlotTwoShort.blockErrorPlot((int)5,(int)3);
     /**BUG FOUND: RUNTIME EXCEPTION**/
     /** java.lang.ArithmeticException: / by zero
                 at BlockErrorPlotTwoShort.blockErrorPlot(BlockErrorPlotTwoShort.java:11)**/

   }
]
```

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.





```
=============Test LOGS=============

 Candidate invariant:
i != 0
i <= 8
i >= 5
j one of { 2, 3, 4 }
j != 0
i > j

=============Test Case=============

[   /** Test case automatically generated by YETI **/
   @Test public void test_0() throws Exception {
     BlockErrorPlotTwoShort.blockErrorPlot((int)8,(int)3);
     /**BUG FOUND: RUNTIME EXCEPTION**/
     /** java.lang.ArithmeticException: / by zero
                at BlockErrorPlotTwoShort.blockErrorPlot(BlockErrorPlotTwoShort.java:11)**/

   }
]
```

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+.

4 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?

5 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.

5.1 Experiments

In the present experiments we tested all 106 packages of Qualitas Corpus containing the total of 4000 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. Its all included packages are open source with an 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 [6][11]. 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$^{\text{TM}}$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/`.

5.2 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 3, 4, 5, 6. 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 as shown in Table 1. 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 1 to 5, the failure domain is classified as point. If failures detected as contiguous lie in the range of 6 to 49, the failure domain is classified as block. 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 50 contain strip failure domain,2 contain point failure domain, 1 contain block failure domain and 4 contain mix failure domain. Mix failure-domain includes the combination of two or more failure domain types including point & strip, point & block and point, block & strip. Invariants generated by manual and automated techniques, and analysis of the source code is also performed to differentiate the simplicity and complexity of the identified failure-domains as shown in Table 1. Further explanation is available in the Nature of failure-domain subsection. The key research questions identified in the previous section are individually addressed in the following.

Table 1. Results of the experiments

Failure domain	Contiguous failures	No. of classes	No. of failure-domains	Easy to Find FD by ADFD	Easy to Find FD by ADFD+	Easy to Find FD by MT	Hard to find FD by ADFD	Hard to find FD by ADFD+	Hard to find FD by ADFD+
Strip	50 or more	50	50	50	45	48	0	5	2
Point	between 1 and 5	2	2	2	2	2	0	0	0
Block	between 6 and 49	1	1	0	1	1	1	0	0
Mix									
	point and strip	3	3	3	0	2	0	3	1
	point and block	0	0	0	0	0	0	0	0
	point, block & strip	1	1	1	0	0	0	1	1
Total		57	57	57	48	53	1	9	4

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 3 - 6, 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, 50 classes showed strip failure domain, 2 point failure domain, 1 block failure domain and 4 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 were split into six categories on the basis of simplicity and complexity of failure-domains identified by each technique. The comparative results show that ADFD, ADFD+ and Manual testing can easily detect 56, 48 and 53 and difficultly detect 1, 9 and 4 failure domains respectively as shown in 1.

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. Consider the following class with a simple failure domain detectable by all three techniques, 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.

6 Threats to validity

All packages in Qualitas Corpus were tested by ADFD, ADFD+ and Manual techniques 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 is related to the hardware and software resources. For example the OutOfMemoryError occured at the value of 6980000 on the machine executing the test. On another machine with different specification the failure revealing value can increase or decrease depending on the hardware and software.

It may be noted that all the non numerical and more than two dimensional methods were not considered in the experiments. The failures caught due to the error of non primitive type were also ignored because of the inability to present them. Therefore the results may reflect less number of failures.

7 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. [12] the boundary values located at the edge of domains have more chances of forming strip failure domain. Finelly [13] and Bishop [14] 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 [1].

Random testing is quick in execution and experimentally proven to detect errors in programs of various platforms including Windows [15], Unix16, Java Libraries citepacheco2005eclat, Heskell [16] and Mac OS [17]. Its ability to become fully automated makes it one of the best choice for automated testing tools [18][19]. AutoTest [20], Jcrasher [18], Eclat [19], Jartege [21], Randoop [22] and YETI [11][6][7] 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 [23].

Our previous studies ADFD [6] and ADFD+ [7] 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 [24][25]. Arcuri et al. [26], 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.

8 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.

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. Myers, G.J., Sandler, C., Badgett, T.: The art of software testing. John Wiley & Sons (2011)
4. Beizer, B.: Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA (1990)
5. 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

6. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **02**(4) (2014) 331–336
7. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **03**(1) (2013) 289–294
8. Oriol, M.: York extensible testing infrastructure (2011)
9. 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
10. Gilbert, D.: The jfreechart class library version 1.0. 9. (2008)
11. 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
12. White, L.J., Cohen, E.I.: A domain strategy for computer program testing. Software Engineering, IEEE Transactions on (3) (1980) 247–257
13. Finelli, G.B.: Nasa software failure characterization experiments. Reliability Engineering & System Safety **32**(1) (1991) 155–169
14. Bishop, P.G.: The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail). In: Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on, IEEE (1993) 98–107
15. Forrester, J.E., Miller, B.P.: An empirical study of the robustness of Windows NT applications using random testing. In: Proceedings of the 4th USENIX Windows System Symposium. (2000) 59–68
16. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices **46**(4) (2011) 53–64
17. Miller, B.P., Cooksey, G., Moore, F.: An empirical study of the robustness of macos applications using random testing. In: Proceedings of the 1st international workshop on Random testing, ACM (2006) 46–54
18. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Software: Practice and Experience **34**(11) (2004) 1025–1050
19. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. Springer (2005)
20. Ciupa, I., Pretschner, A., Leitner, A., Oriol, M., Meyer, B.: On the predictability of random tests for object-oriented software. In: Software Testing, Verification, and Validation, 2008 1st International Conference on, IEEE (2008) 72–81
21. Oriat, C.: Jartege: a tool for random generation of unit tests for java classes. In: Quality of Software Architectures and Software Quality. Springer (2005) 242–256
22. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for java. In: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, ACM (2007) 815–816
23. Oriol, M., Tassis, S.: Testing .NET code with YETI. In: Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on, IEEE (2010) 264–265
24. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. Software Engineering, IEEE Transactions on **SE-10**(4) (july 1984) 438 –444
25. Gutjahr, W.: Partition testing vs. random testing: the influence of uncertainty. Software Engineering, IEEE Transactions on **25**(5) (sep/oct 1999) 661 –674
26. Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: Theoretical results and practical implications. IEEE Transactions on Software Engineering **38** (2012) 258–277

Table 2. Table depicting results of ADFD and ADFD+

S#	Project	Class	Method	Dim	LOC	Failure domain
1	ant	LeadPipeInputStream	LeadPipeInputStream(i)	1	159	Strip
2	antlr	BitSet	BitSet.of(i,j)	2	324	Strip
3	artofillusion	ToolPallete	ToolPalette(i,j)	2	293	Strip
4	aspectj	AnnotationValue	whatKindIsThis(i)	1	68	**Mix**
		IntMap	idMap(i)	1	144	Strip
5	cayenne	ExpressionFactory	expressionOfType(i)	1	146	Strip
6	collections	ArrayStack	ArrayStack(i)	1	192	Strip
		BinaryHeap	BinaryHeap(i)	1	63	Strip
		BondedFifoBuffer	BoundedFifoBuffer(i)	1	55	Strip
		FastArrayList	FastArrayList(i)	1	831	Strip
		StaticBucketMap	StaticBucketMap(i)	1	103	Strip
		PriorityBuffer	PriorityBuffer(i)	1	542	Strip
7	colt	GenericPermuting	permutation(i,j)	2	64	Strip
		LongArrayList	LongArrayList(i)	1	153	Strip
		OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	1	47	Strip
8	drjava	Assert	assertEquals(i,j)	2	780	**Point**
		ByteVector	ByteVector(i)	1	40	Strip
9	emma	ClassLoaderResolver	getCallerClass(i)	1	225	Strip
		ElementFactory	newConstantCollection(i)	1	43	Strip
		IntIntMap	IntIntMap(i)	1	256	Strip
		ObjectIntMap	ObjectIntMap(i)	1	252	Strip
		IntObjectMap	IntObjectMap(i)	1	214	Strip
10	heritrix	ArchiveUtils	padTo(i,j)	2	772	Strip
		BloomFilter32bit	BloomFilter32bit(i,j)	2	223	Strip
11	hsqld	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	1	52	Strip
		ObjectCacheHashMap	ObjectCacheHashMap(i)	1	76	Strip
12	htmlunit	ObjToIntMap	ObjToIntMap(i)	1	466	Strip
		Token	typeToName(i)	1	462	**Mix**
13	itext	PRTokeniser	isDelimiterWhitespace(i)	1	593	Strip
		PdfAction	PdfAction(i)	1	585	Strip
		PdfLiteral	PdfLiteral(i)	1	101	Strip
14	jung	PhysicalEnvironment	PhysicalEnvironment(i)	1	503	Strip
15	jedit	IntegerArray	IntegerArray(i)	1	82	Strip
16	jgraph	AttributeMap	AttributeMap(i)	1	105	Strip
17	jruby	ByteList	ByteList(i)	1	1321	Strip
		WeakIdentityHashMap	WeakIdentityHashMap(i)	1	50	Strip
18	junit	Assert	assertEquals(i,j)	2	780	**Point**
19	megamek	AmmoType	getMunitionsFor(i)	1	268	Strip
		Board	getTypeName(i, j)	1	1359	**Mix**
20	nekohtml	HTMLEntities	get(i)	1	63	Strip
21	poi	Variant	getVariantLength(i)	1	476	**Mix**
		IntList	IntList(i,j)	2	643	**Block**
22	sunflow	QMC	halton(i,j)	2	32	Strip
		BenchmarkFramework	BenchmarkFramework(i,j)	2	24	Strip
		IntArray	IntArray(i)	1	47	Strip
23	trove	TDoubleStack	TDoubleStack(i)	1	120	Strip
		TIntStack	TIntStack(i)	1	120	Strip
		TLongArrayList	TLongArrayList(i)	1	927	Strip
24	weka	AlgVector	AlgVector(i)	1	424	Strip
		BinarySparseInstance	BinarySparseInstance(i)	1	614	Strip
25	xerces	SoftReferenceSymbolTable	SoftReferenceSymbolTable(i)	1	71	Strip
		SymbolHash	SymbolHash(i)	1	82	Strip
		SynchronizedSymbolTable	SynchronizedSymbolTable(i)	1	57	Strip
		XMLChar	isSpace(i)	1	169	Strip
		XMLGrammarPoolImpl	XMLGrammarPoolImpl(i)	1	96	Strip
		XML11Char	isXML11NCNameStart(i)	1	184	Strip
		AttributeList	AttributeList(i)	1	321	Strip

S#	Class	Invariants by ADFD+	Invariants by ADFD	Invariants by Manual
1	LeadPipeInputStream	I >= 2147483140 I <= 2147483647	I >= 2147483143 I <= 2147483647	I > 698000000 I <= 2147483647
2	BitSet	I <= -1, I >= -18, J <= 7, J >= -12	I <= -1, I >= -513 J >= -503, J <= 507	I <= -1, I >= -2147483648 J any value
3	ToolPallete	I <= -1, I >= -18 J <= 3, J >= -15	I <= -1, I >= -515 J >= -509, J <= 501	I <= -1, I >= -2147483648 J any value
4	IntMap	I <= -1, I >= -18	I <= -1, I >= -512	I <= -1, I >= -2147483648
5	ExpressionFactory	I <= 13, I >= -7	I >= -497, I <= 513	I >= -2147483648 I <= 2147483647
6	ArrayStack	I >= 2147483636 I <= 2147483647	I >= 2147483142 I <= 2147483647	I > 698000000 I <= 2147483647
7	BinaryHeap	I <= -2147483637 I >= -2147483648	I <= -2147483142 I >= -2147483648	I <= 0 I >= -2147483648
8	BondedFifoBuffer	I <= -2147483639 I >= -2147483648	I >= -505, I <= 0	I <= 0 I >= -2147483648
9	FastArrayList	I <= -2147483641 I >= -2147483648	I <= -2147483644, I >= -2147483139	I <= -1 I >= -2147483648
10	StaticBucketMap	I >= 2147483635 I <= 2147483647	I >= 2147483140 I <= 2147483647	I > 698000000 I <= 2147483647
11	PriorityBuffer	I <= -1, I >= -14	I <= -2147483142 I >= -2147483647	I <= 0 I >= -2147483648
12	GenericPermuting	I <= 0, I >= -18	I >= -498, I <= 0 I >= 2, I <= 512	I <= 0, I >= -2147483648 I >= 2, I <= 2147483647
13	LongArrayList	I <= -2147483640 I >= -2147483648	I <= -1, I >= -510	I <= -1 I >= -2147483648
14	OpenIntDoubleHashMap	I <= -1, I >= -17	I <= -1, I >= -514	I <= -1, I >= -2147483648
15	ByteVector	I <= -2147483639 I >= -2147483648	I <= -2147483141 I >= -2147483648	I <= -1 I >= -2147483648
16	ElementFactory	I >= 2147483636 I <= 2147483647	I >= 2147483141 I <= 2147483647	I > 698000000 I <= 2147483647
17	IntIntMap	I <= -2147483638 I >= -2147483648	I <= -2147483644 I >= -2147483139	I <= -1 I >= -2147483648
18	ObjectIntMap	I >= 2147483640 I <= 2147483647	I >= 2147483591 I <= 2147483647	I > 698000000 I <= 2147483647
19	IntObjectMap	I <= -1, I >= -17	I <= -1, I >= -518	I <= -1, I >= -2147483648
20	ArchiveUtils	I >= 2147483641 I <= 2147483647 J >= 2147483639 J <= 2147483647	I >= -497 I <= 513 J >= 2147483591 J <= 2147483647	I any value J > 698000000
21	BloomFilter32bit	I <= -1, I >= -18 J may be any value	I <= -1, I >= -515 J may be any value	I <-1 J <-1
22	IntKeyLongValueHashMap	I >= 2147483635 I <= 2147483647	I >= 2147483590 I <= 2147483647	I > 698000000 I <= 2147483647
23	ObjectCacheHashMap	I <= -2147483641 I >= -2147483648	I >= -512, I <= 0	I <= 0 I >= -2147483648
24	ObjToIntMap	I <= -2147483636 I >= -2147483648	I <= -2147483646 I >= -2147483648	I <= -1 I >= -2147483648
25	PRTokeniser	I <= -2 I >= -18	I <= -2, I >= -509 I >= 256, I <= 501	I <= -2 , I >= -2147483648 I >= 256 , I <= 2147483647
26	PdfAction	I <= -2147483640 I >= -2147483648	I <= 0, I >= -514 I >= 6, I <= 496	I <= 0, I >= -2147483648 I >= 6, I <= 2147483647
27	PdfLiteral	I <= -1, I >= -14	I <= -1, I >= -511	I <= -1, I >= -2147483648
28	PhysicalEnvironment	I <= -1, I >= -11	I <= -2147483646 I >= -2147483648	I <= -1, I >= -2147483648
29	IntegerArray	I >= 2147483636 I <= 2147483647	I >= 2147483587 I <= 2147483647	I > 698000000 I <= 2147483647
30	AttributeMap	I <= -2147483639 I >= -2147483648	I <= 0, I >= -514	I <= 0 I >= -2147483648
31	ByteList	I <= -1, I >= -14	I <= -1, I >= -513	I <= -1, I >= -2147483648
32	WeakIdentityHashMap	I >= 2147483636 I <= 2147483647	I >= 2147483140 I <= 2147483647	I >698000000 I <= 2147483647
33	AmmoType	I <= -1 I >= -17	I <= -1, I >= -514 I >= 93, I <= 496	I <= -1, I >= -2147483648 I >= 93, I <= 2147483647
34	QMC	I <= -1, I >= -12 J <= -1, J >= -15	I <= -1, I >= -508 J <= 499, J >= -511	I <= -1, I >= -2147483648 J any value
35	BenchmarkFramework	I <= -1, I >= -13	I <= -1, I >= -508	I <= -1, I >= -2147483648
36	IntArray	I <= -1, I >= -16	I <= -2147483650 I >= -2147483141	I <= -1 I >= -2147483648
37	TDoubleStack	I <= -1, I >= -13	I <= -1, I >= -511	I <= -1, I >= -2147483648
38	TIntStack	I <= -1, I >= -12	I <= -2147483648 I >= -2147483144	I <= -1 I >= -2147483648
39	TLongArrayList	I <= -1, I >= -16	I <= -2147483648 I >= -2147483141	I <= -1, I >= -2147483648
40	AlgVector	I <= -1, I >= -15	I <= -1, I >= -511	I <= -1, I >= -2147483648
41	BinarySparseInstance	I <= -1, I >= -15	I <= -1, I >= -506	I <= -1, I >= -2147483648
42	SoftReferenceSymbolTable	I >= 2147483635 I <= 2147483647	I >= 2147483140 I <= 2147483647	I > 698000000 I <= 2147483647
43	HTMLEntities	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 <= -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
44	SymbolHash	I <= -1, I >= -16	I <= -2147483592 I >= -2147483648	I <= -1, I >= -2147483648
45	SynchronizedSymbolTable	I <= -2147483140 I >= -2147483648	I <= -2147483592, I >= -2147483648	I <= -1, I >= -2147483648
46	XMLChar	I <= -1, I >= -12	I <= -1, I >= -510	I <= -1, I >= -2147483648
47	XMLGrammarPoolImpl	I <= -1, I >= -13	I <= -2147483137 I >= -2147483648	I <= -1, I >= -2147483648
48	XML11Char	I <= -1, I >= -16	I <= -1, I >= -512	I <= -1, I >= -2147483648
49	AttributeList	I >= 2147483635 I <= 2147483647	I >= 2147483590 I <= 2147483647	I > 698000000 I <= 2147483647
50	ClassLoaderResolver	I >= 2, I <= 18	I >= 500, I <= -2 I >= 2, I <= 505	I <= -2, I >-2147483648 I >= 2, I <= 2147483647

S#	Class	Invariants by ADFD+	Invariants by ADFD	Invariants by Manual
1	Assert	I != J	I != J	I != J
2	Assert	I <= 0, I >= 20 J = 0	I <= -2147483142, I >= -2147483648, J = 0	I any value J = 0

Table 4. Classes with point failure-domains

S#	Class	Invariants by ADFD+	Invariants by ADFD	Invariants by Manual
2	IntList	I <= -1, I >= -15 J = 0	I <= -1, I >= -509 J =0	I <= -1, I >= -2147483648 J = 0

Table 5. Classes with block failure-domains

S#	Class	Invariants by ADFD	Invariants by ADFD+	Invariants by Manual
1	Board	I <= -1 I >= -18 J = 0	I >= -504, I <= -405, I >= -403, I <= -304, I >= -302, I <= -203, I >= -201, I <= -102, I >= -100, I <= -1 J = 0	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, J = 0
2	Variant	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
3	Token	I <= -2147483641 I >= -2147483648	I <= -2, I >= -510 I = {73, 156} I >= 162, I <= 500	I <= -2, I >-2147483648 I = 73, 156, I >= 162, I <= 2147483647
4	AnnotationValue	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 and so on

Table 6. Classes with mix failure-domains