Evaluation of ADFD and ADFD$^+$ techniques

Mian Asbat Ahmad and Manuel Oriol

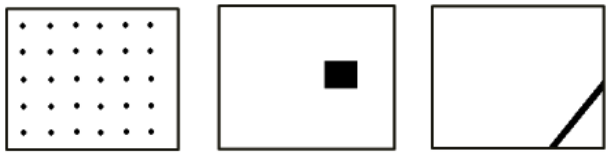
University of York, Department of Computer Science,
Deramore Lane, YO10 5GH YORK, United Kingdom

Abstract. The ever-increasing reliance on software-intensive system is driving research to discover software faults more efficiently. Despite intensive research, very few approaches have studied and used knowledge about fault domains to improve the testing or the feedback given to developers. The shortcoming was addressed by developing ADFD and ADFD$^+$ strategies presented in our previous publications. In the present study, the two strategies were enhanced by integration of the automatic testing tool Daikon and the precision of identifying failure domains was determined through extensive experimental evaluation of real world Java projects contained in Qualitas Corpus. The analyses of results, cross-checked by manual testing indicated that ADFD and ADFD$^+$ techniques are highly effective in providing assistance but are not an alternative to manual testing with the limited available resources.

Keywords: software testing, automated random testing, manual testing, ADFD, ADFD$^+$, Daikon

1 Introduction

The input-domain of a given System Under Test (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]

We have developed two fully automated techniques ADFD [3] and ADFD$^+$ [4], which effectively find failures and failure 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 York Extensible Testing Infrastructure (YETI), Daikon and JFreeChart have been used in combination for developing ADFD and ADFD$^+$ techniques. The YETI [5], Daikon [6] and JFreeChart [7] are used for testing the program, generating invariants and plotting the pass and fail domains respectively.

The rest of the paper is organized as follows: § 2 presents enhancement of the techniques. § 3 shows the difference in working mechanism of the two techniques by a motivating example. § 4 highlights the key research questions. § 5 describes the evaluation process comprising experiments, results and answers to the research questions. § 6 presents threats to validity while § 7 points out the related work. Finally, § 8 presents conclusion of the study.

2 Enhancement of the techniques

Prior to experimental evaluation, new features were incorporated in ADFD and ADFD$^+$ techniques to: increase the code coverage, provide information about the identified failure and generate invariants of the detected failure domains as stated below:

1. The GUI was enabled to launch all the strategies defined in YETI from a single interface. As an example, if ADFD strategy is selected for testing, the system automatically hides the field (range value) associated with ADFD$^+$ and displays two fields of lower and upper bounds. On the other hand if ADFD$^+$ strategy is selected for testing, the system automatically hides the two fields (lower and upper bounds) associated with ADFD technique and displays a single field of range value.
2. Code coverage was increased by extending the techniques to support the testing of methods with `byte, short, long, double` and `float` type arguments while it was restricted to `int` type arguments only in the original techniques.
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 [6]. The generated invariants are displayed in GUI at the end of test execution.
4. The screen capture button was added to the GUI to allow the user to capture multiple screen-shots at different intervals of testing for future reference.

3 Difference in working mechanism of the two techniques

Difference in working mechanism of ADFD and ADFD$^+$ for identification of failure domains is illustrated by testing a simple Java program (given below) with the two techniques. It is evident from the program code that failure is generated when the value of variable $x = \{4, 5, 6, 7 \text{ or } 8\}$ and the corresponding value of variable $y = \{2, 3 \text{ or } 4\}$. The total number of 12 failing instances form a block failure domain in the input domain.

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

The test output generated by ADFD technique is presented in Figure 2. The labelled graph shows 4 out of 12 failing values in red whereas the passing values are shown in blue. The generated invariants identify all but one failing value ($x = 4$). 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, name of the failing class, name of the failing method, values causing the failure and line number of the code causing failure.

The test output generated by ADFD$^+$ technique is presented in Figure 3. The labelled graph correctly shows all the 12 out of 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, name of the failing class, name of the failing method, values causing the failure and line number of the code causing failure.

The comparative results derived from execution of the two techniques on the developed program indicate that, ADFD$^+$ is more efficient than ADFD in identification of failures in two-dimensional programs. The 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 fewer resources than ADFD$^+$.

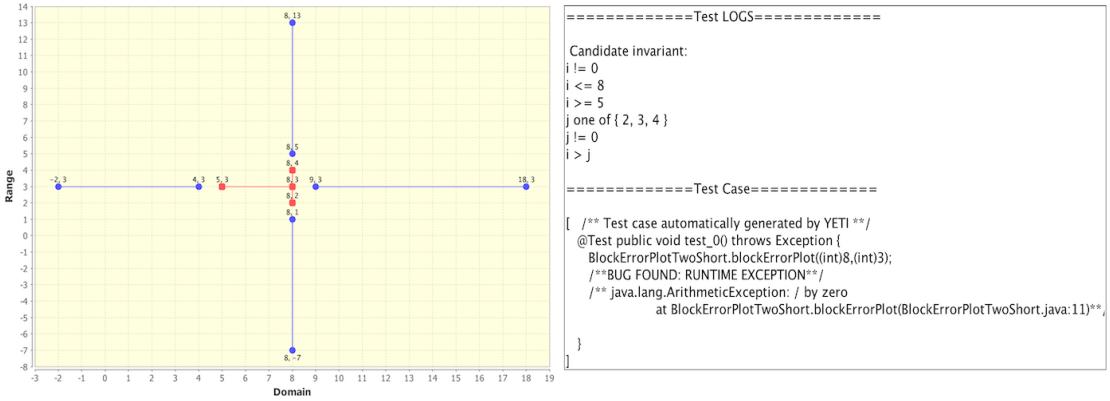


Fig. 2. Graph, Invariants and test case generated by ADFD for the given code

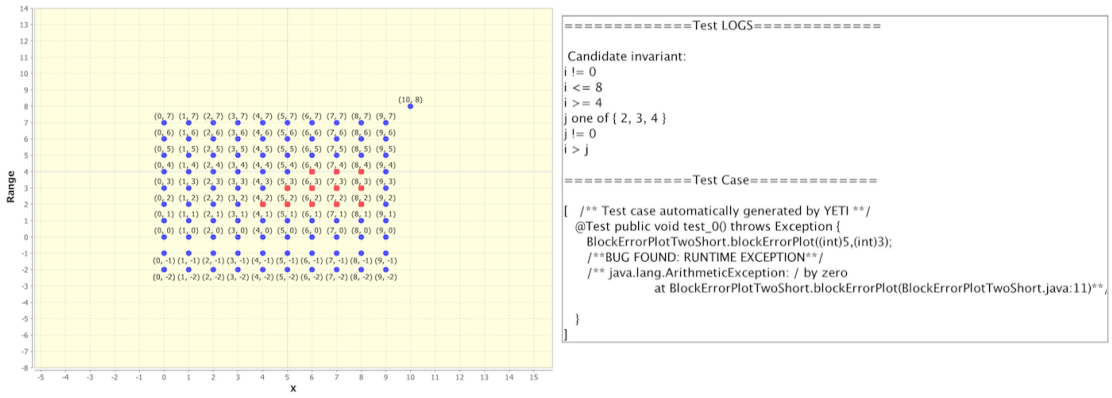


Fig. 3. Graph, Invariants and Test case generated by ADFD$^+$ for the given code

4 Research questions

The following research questions have been addressed in the study:

1. What is the relevance of ADFD and ADFD$^+$ techniques in identification and presentation of 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 automated 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 a 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 span across the whole set of Java applications and is specially built for empirical research which takes into account a large number of developmental models and programming styles. All packages included in Qualitas Corpus are open source with an easy access to the source code.

For experimental purpose, the main ".jar" file of each package was extracted to get the ".class" files as appropriate input for YETI. All 4000 classes were individually tested. The classes containing one and two-dimensional methods with arguments (int, long, float, byte, double and short) were selected for experimental analysis. Non-numerical arguments and more than two-dimensional methods were ignored because the two proposed techniques support the testing of 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 500 hours to perform the experiments completely. 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 [8], [3]. The source code of the programs containing failure domains were also evaluated manually to cross-examine the experimental results.

In accordance with Chan et al. [1], classification of failure domain into various types was based on the number of contiguous failures detected in the input-domain as shown in Table 1. If the contiguous failures detected range from 1 to 5, 6 to 49 or 50 and above the failure domain is classified as point, block or strip type respectively. If more than one type of domain are detected in a program, it is termed as mix type.

Table 1. Classification of failure domains

S. No	Type of failure domain	No of contiguous failures
1	Point	01 to 05
2	Block	06 to 49
3	Strip	50 & above
4	Mix	point & block point & strip point, block & strip

5.2 Results

The testing of 106 Java packages including 4000 classes, resulted in 25 packages containing 57 classes to have various types of failure domains. The details pertaining to project, class, method, dimension, line of code (LOC) and type of detected failure domains for each class are given in Table 3. Out of the total of 57 methods indicated in the table, 10 methods are two-dimensional while the remaining 47 methods are one-dimensional. A total number of 17262 lines of code spread across 57 classes in various proportions as shown in the table. The results obtained show that out of 57 classes 2 contain point failure domain, 1 contains block failure domain, 50 contain strip failure domain and 4 contain mix failure domain.

Effectiveness of ADFD and ADFD$^+$ techniques 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; 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 upon the detection of failure by random$^+$ strategy in YETI. Because only after a failure is identified, its neighbouring values are analysed according to the set range to plot the failure domain.

The generation of graph and invariants and the time of test execution directly depends on the range value, if the range value of a technique is greater, the presentation of failure domain is better and the execution time required is higher. This is due to the testing and handling of greater number of test cases when the range is set to a bigger level. Comparatively, ADFD requires fewer resources than ADFD$^+$ therefore it is less influenced by the range value.

Type and Frequency of Failure domains As evident from the results given in Table 4 - 7, all the three techniques (ADFD, ADFD$^+$ and Manual) detected the presence of strip, point and block types of failure domains in different frequencies. The results obtained show that out of 57 classes 2 contain point failure domain, 1 contains block failure domain, 50 contain strip failure domain and 4 contain mix failure domain. Mix failure domain includes the combination of two or more types of failure domains including point & block, point & strip and point, block & strip.

The discovery of higher number of strip 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 failure domain.

Nature of failure domain The nature of failure domain identified by two automatic techniques (ADFD and ADFD$^+$) and the manual technique was examined in terms of simplicity and complexity by comparing the invariants generated by automatic techniques with those of the manual technique. The results were split into six categories (2 categories per technique) on the basis of simplicity and complexity of failure domains identified by each of the three techniques. 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 Table 2. The analysis of generated invariants indicate that the failure domains which are simple in nature are easily detectable by both automated and manual techniques while the failure domains which are complex in nature are difficultly detectable by both automated and manual techniques.

Table 2. Simplicity and complexity of Failure Domains (FD) as found by 03 techniques

Type of failure domain	No. of classes	No. of FD	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 MT
Point	2	2	2	2	2	0	0	0
Block	1	1	0	1	1	1	0	0
Strip	50	50	50	45	48	0	5	2
Mix	0	0	0	0	0	0	0	0
	3	3	3	0	2	0	3	1
	1	1	1	0	0	0	1	1
Total	57	57	57	48	53	1	9	4

The simplicity of failure domain is illustrated by taking an example of ADFD, ADFD$^+$ and Manual Analysis in Table 7 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 $\{i \leq$ *-1,* $i \geq$ *-18*$\}$, by ADFD$^+$ are $\{i \leq$ *-1,* $i \geq$ *-512*$\}$ and by Manual Analysis are $\{i \leq$ *-1,* $i \geq$ *Integer.MIN_INT*$\}$. These results indicate maximum degree of representation of failure domain by Manual Analysis followed by ADFD and ADFD$^+$ respectively. This is mainly due to the bigger range value in manual analysis followed by ADFD and ADFD$^+$ respectively.

The complexity of failure domain is illustrated by taking an example of ADFD, ADFD$^+$ and Manual Analysis in Table 7 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 $\{$ $i \geq$ *698000000,* $i \leq$ *698000300*$\}$, by ADFD$^+$ are $\{$ $i \geq$ *2147483636,* $I \leq$ *MAX_INT*$\}$, by Manual analysis $\{$ $i \geq$ *698000000* $\}$. All the three strategies indicate the same failure but at different intervals. The ADFD$^+$ is unable to show the starting point of failure

due to its small range value. The ADFD easily discovers the breaking point due to its bigger range value while manual testing requires over 50 attempts to find the breaking point.

6 Threats to validity

All packages in Qualitas Corpus were tested by ADFD, ADFD$^+$ and Manual technique 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$^+$ are based on random$^+$ strategy having high preference for boundary values, therefore, most of the failures detected are from the boundaries of the 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.

A threat to validity is related to the hardware and software resources. For example, the `OutOfMemoryError` occurs at the value of 6980000 on the machine used for executing the test. On another machine with different specification the failure revealing value can increase or decrease depending on the hardware and software resources.

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

7 Related Work

Shape and location of failure domain within the input domain have been studied in the past. Similar to our findings, White et al. [9] reported that the boundary values have more chances of forming strip failure domain. Finally [10] and Bishop [11] found that failure causing inputs form a continuous region inside the input domain. Chan et al. revealed that failure causing values form 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 [12], Unix [13], Java Libraries [14], Heskell [15] and Mac OS [16]. Its potential to become fully automated makes it one of the best choice for developing automated testing tools [17], [14]. AutoTest [18], Jcrasher [17], Eclat [14], Jartege [19], Randoop [20] and YETI [8], [3], [4] are a few of the most common automated random testing tools used by the research community.

In our previous research publications, we have described the fully automated techniques ADFD [3] and ADFD$^+$ [4] for the discovery of failure domains and

have experimentally evaluated the performance with one and two-dimensional error-seeded numerical programs. The current study is a continuation of the previous work. It is aimed at the enhancement of the two techniques for evaluation of the precision of identifying failure domains by integrating Daikon with ADFD and ADFD$^+$.

Our current approach of evaluation is inspired from several studies in which random testing has been compared with other testing techniques to find the failure finding ability [22], [23], [24]. The automated techniques have been compared with manual techniques in the previous research studies [25], [26]. This study is of special significance because we compared the effectiveness of the techniques by identifying failure domains rather than individual failures considered in the previous studies.

8 Conclusion

Based on the results, it is concluded that the two automated techniques (ADFD and ADFD$^+$) are more effective in identifying and presenting complex (point and block) failure domains with minimal labour. The manual technique is more effective in identifying simple (long strip) failure domain but is tedious and labour intensive. The precision to identify failure domains can be increased by increasing the range value. The results indicate that the automated techniques can be highly effective in providing assistance to manual testing but are not an alternative to manual testing.

Acknowledgments The authors are thankful to the Department of Computer Science, University of York for academic and financial support. Thanks are also extended to Prof. Richard Paige and Prof. John Clark for their valuable guidance, help and cooperation.

References

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. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **02**(4) (2014) 331–336
4. Ahmad, M.A., Oriol, M.: Automated discovery of failure domain. Lecture Notes on Software Engineering **03**(1) (2013) 289–294
5. Oriol, M.: York extensible testing infrastructure (2011)
6. 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
7. Gilbert, D.: The jfreechart class library version 1.0. 9. (2008)

8. 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
9. White, L.J., Cohen, E.I.: A domain strategy for computer program testing. Software Engineering, IEEE Transactions on (3) (1980) 247–257
10. Finelli, G.B.: Nasa software failure characterization experiments. Reliability Engineering & System Safety **32**(1) (1991) 155–169
11. 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
12. 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
13. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the ACM **33**(12) (1990) 32–44
14. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. Springer (2005)
15. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. Acm sigplan notices **46**(4) (2011) 53–64
16. 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
17. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Software: Practice and Experience **34**(11) (2004) 1025–1050
18. 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
19. 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
20. 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
21. 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
22. Hamlet, D., Taylor, R.: Partition testing does not inspire confidence (program testing). IEEE Transactions on Software Engineering **16**(12) (1990) 1402–1411
23. Weyuker, E.J., Jeng, B.: Analyzing partition testing strategies. Software Engineering, IEEE Transactions on **17**(7) (1991) 703–711
24. Gutjahr, W.J.: Partition testing vs. random testing: The influence of uncertainty. Software Engineering, IEEE Transactions on **25**(5) (1999) 661–674
25. Leitner, A., Ciupa, I.: Reconciling manual and automated testing: the autotest experience. In: Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology, Technology (2007) 3–6
26. Ciupa, I., Meyer, B., Oriol, M., Pretschner, A.: Finding faults: Manual testing vs. random+ testing vs. user reports. In: Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on, IEEE (2008) 157–166

Appendix

Table 3. 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

Table 4. Classes with block failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
1	IntList	I $\leq$ -1, I $\geq$ -15 J = 0	I $\leq$ -1, I $\geq$ -509 J =0	I $\leq$ -1, I $\geq$ min_int J = 0

Table 5. Classes with point failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
1	Assert	I != J	I != J	I != J
2	Assert	I $\leq$ 0, I $\geq$ 20 J = 0	I $\leq$ -2147483142, I $\geq$ min_int J = 0	I any value J = 0

Table 6. Classes with mix failure domains

S#	Class	Invariants by ADFD	Invariants by ADFD$^+$	Invariants by Manual
1	Board	I $\leq$ -1 I $\geq$ -18 J = 0	I $\geq$ -504, I $\leq$ -405, I $\geq$ -403, I $\leq$ -304, I $\geq$ -302, I $\leq$ -203, I $\geq$ -201, I $\leq$ -102, I $\geq$ -100, I $\leq$ -1 J = 0	I $\leq$ -910, I $\geq$ -908, I $\leq$ -809, I $\geq$ -807, I $\leq$ -708, I $\geq$ -706, I $\leq$ -607, I $\geq$ -605, I $\leq$ -506, I $\geq$ -504, I $\leq$ -405, I $\geq$ -403, I $\leq$ -304, I $\geq$ -302, I $\leq$ -203, I $\geq$ -201, I $\leq$ -102, I $\geq$ -100 I $\leq$ -1, J = 0
2	Variant	I $\geq$0, I $\leq$ 12	I $\geq$ 0, I $\leq$ 14, I $\geq$ 16 I $\leq$ 31, I $\geq$ 64, I $\leq$ 72	I $\geq$ 0, I $\leq$ 14, I $\geq$ 16 I $\leq$ 31, I $\geq$ 64, I $\leq$ 72
3	Token	I $\leq$ -2147483641 I $\geq$ min_int	I $\leq$ -2, I $\geq$ -510 I = {73, 156} I $\geq$ 162, I $\leq$ 500	I $\leq$ -2, I >min_int I = 73, 156, I $\geq$ 162, I $\leq$ max_int
4	AnnotationValue	I $\leq$ 85, I $\geq$ 92, I $\geq$ 98, I $\leq$ 100, I $\geq$ 102, I $\leq$ 104	I $\leq$63, I = {65, 69, 71, 72} I $\geq$ 75, I $\leq$ 82, I $\geq$ 84 I $\leq$ 89, I $\geq$ 92, I $\leq$ 98 I = 100, I $\geq$ 102, I $\leq$ 114 I $\geq$ 116	I $\leq$ 63, I = 65, 69, 71, 72 I $\geq$ 75, I $\leq$ 82, I $\geq$ 84 I $\leq$ 89, I $\geq$ 92, I $\leq$ 98 I = 100, I $\geq$ 102, I $\leq$ 114 I $\geq$ 116 and so on

Table 7: Classes with strip failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
1	LeadPipeInputStream	I $\geq$ 2147483140	I $\geq$ 2147483143	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
2	BitSet	I $\leq$ -1, I $\geq$ -18,	I $\leq$ -1, I $\geq$ -513	I $\leq$ -1, I $\geq$ min_int
		J $\leq$ 7, J $\geq$ -12	J $\geq$ -503, J $\leq$ 507	J any value
3	ToolPallete	I $\leq$ -1, I $\geq$ -18	I $\leq$ -1, I $\geq$ -515	I $\leq$ -1, I $\geq$ min_int
		J $\leq$ 3, J $\geq$ -15	J $\geq$ -509, J $\leq$ 501	J any value
4	IntMap	I $\leq$ -1, I $\geq$ -18	I $\leq$ -1, I $\geq$ -512	I $\leq$ -1, I $\geq$ min_int
5	ExpressionFactory	I $\leq$ 13, I $\geq$ -7	I $\geq$ -497, I $\leq$ 513	I $\geq$ min_int
				I $\leq$ max_int
6	ArrayStack	I $\geq$ 2147483636	I $\geq$ 2147483142	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
7	BinaryHeap	I $\leq$ -2147483637	I $\leq$ -2147483142	I $\leq$ 0
		I $\geq$ min_int	I $\geq$ min_int	I $\geq$ min_int
8	BondedFifoBuffer	I $\leq$ -2147483639	I $\geq$ -505, I $\leq$ 0	I $\leq$ 0
		I $\geq$ min_int		I $\geq$ min_int
9	FastArrayList	I $\leq$ -2147483641	I $\leq$ -2147483644,	I $\leq$ -1
		I $\geq$ min_int	I $\geq$ -2147483139	I $\geq$ min_int
10	StaticBucketMap	I $\geq$ 2147483635	I $\geq$ 2147483140	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
11	PriorityBuffer	I $\leq$ -1, I $\geq$ -14	I $\leq$ -2147483142	I $\leq$ 0
			I $\geq$ -2147483647	I $\geq$ min_int
12	GenericPermuting	I $\leq$ 0, I $\geq$ -18	I $\geq$ -498, I $\leq$ 0	I $\leq$ 0, I $\geq$ min_int
			I $\geq$ 2, I $\leq$ 512	I $\geq$ 2, I $\leq$ max_int
13	LongArrayList	I $\leq$ -2147483640	I $\leq$ -1, I $\geq$ -510	I $\leq$ -1
		I $\geq$ min_int		I $\geq$ min_int
14	OpenIntDoubleHashMap	I $\leq$ -1, I $\geq$ -17	I $\leq$ -1, I $\geq$ -514	I $\leq$ -1, I $\geq$ min_int
15	ByteVector	I $\leq$ -2147483639	I $\leq$ -2147483141	I $\leq$ -1
		I $\geq$ min_int	I $\geq$ min_int	I $\geq$ min_int
16	ElementFactory	I $\geq$ 2147483636	I $\geq$ 2147483141	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
17	IntIntMap	I $\leq$ -2147483638	I $\leq$ -2147483644	I $\leq$ -1
		I $\geq$ min_int	I $\geq$ -2147483139	I $\geq$ min_int
18	ObjectIntMap	I $\geq$ 2147483640	I $\geq$ 2147483591	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
19	IntObjectMap	I $\leq$ -1, I $\geq$ -17	I $\leq$ -1, I $\geq$ -518	I $\leq$ -1, I $\geq$ min_int
20	ArchiveUtils	I $\geq$ 2147483641	I $\geq$ -497	I any value
		I $\leq$ max_int	I $\leq$ 513	
		J $\geq$ 2147483639	J $\geq$ 2147483591	J $>$ 698000000
		J $\leq$ max_int	J $\leq$ max_int	
21	BloomFilter32bit	I $\leq$ -1, I $\geq$ -18	I $\leq$ -1, I $\geq$ -515	I <-1
		J may be any value	J may be any value	J <-1
22	IntKeyLongValueHashMap	I $\geq$ 2147483635	I $\geq$ 2147483590	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
23	ObjectCacheHashMap	I $\leq$ -2147483641	I $\geq$ -512, I $\leq$ 0	I $\leq$ 0
		I $\geq$ min_int		I $\geq$ min_int
24	ObjToIntMap	I $\leq$ -2147483636	I $\leq$ -2147483646	I $\leq$ -1
		I $\geq$ min_int	I $\geq$ min_int	I $\geq$ min_int
25	PRTokeniser	I $\leq$ -2	I $\leq$ -2, I $\geq$ -509	I $\leq$ -2 , I $\geq$ min_int
		I $\geq$ -18	I $\geq$ 256, I $\leq$ 501	I $\geq$ 256 , I $\leq$ max_int
26	PdfAction	I $\leq$ -2147483640	I $\leq$ 0, I $\geq$ -514	I $\leq$ 0, I $\geq$ min_int
		I $\geq$ min_int	I $\geq$ 6, I $\leq$ 496	I $\geq$ 6, I $\leq$ max_int
27	PdfLiteral	I $\leq$ -1, I $\geq$ -14	I $\leq$ -1, I $\geq$ -511	I $\leq$ -1, I $\geq$ min_int
28	PhysicalEnvironment	I $\leq$ -1, I $\geq$ -11	I $\leq$ -2147483646	I $\leq$ -1,
			I $\geq$ min_int	I $\geq$ min_int
29	IntegerArray	I $\geq$ 2147483636	I $\geq$ 2147483587	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
30	AttributeMap	I $\leq$ -2147483639	I $\leq$ 0, I $\geq$ -514	I $\leq$ 0
		I $\geq$ min_int		I $\geq$ min_int
31	ByteList	I $\leq$ -1, I $\geq$ -14	I $\leq$ -1, I $\geq$ -513	I $\leq$ -1, I $\geq$ min_int
32	WeakIdentityHashMap	I $\geq$ 2147483636	I $\geq$ 2147483140	I >698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
33	AmmoType	I $\leq$ -1	I $\leq$ -1, I $\geq$ -514	I $\leq$ -1, I $\geq$ min_int
		I $\geq$ -17	I $\geq$ 93, I $\leq$ 496	I $\geq$ 93, I $\leq$ max_int
34	QMC	I $\leq$ -1, I $\geq$ -12	I $\leq$ -1, I $\geq$ -508	I $\leq$ -1, I $\geq$ min_int

Table 7: Classes with strip failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
		J $\leq$ -1, J $\geq$ -15	J $\leq$ 499, J $\geq$ -511	J any value
35	BenchmarkFramework	I $\leq$ -1, I $\geq$ -13	I $\leq$ -1, I $\geq$ -508	I $\leq$ -1, I $\geq$ min_int
36	IntArray	I $\leq$ -1, I $\geq$ -16	I $\leq$ -2147483650	I $\leq$ -1
			I $\geq$ -2147483141	I $\geq$ min_int
37	TDoubleStack	I $\leq$ -1, I $\geq$ -13	I $\leq$ -1, I $\geq$ -511	I $\leq$ -1, I $\geq$ min_int
38	TIntStack	I $\leq$ -1, I $\geq$ -12	I $\leq$ min_int	I $\leq$ -1
			I $\geq$ -2147483144	I $\geq$ min_int
39	TLongArrayList	I $\leq$ -1, I $\geq$ -16	I $\leq$ min_int	I $\leq$ -1,
			I $\geq$ -2147483141	I $\geq$ min_int
40	AlgVector	I $\leq$ -1, I $\geq$ -15	I $\leq$ -1, I $\geq$ -511	I $\leq$ -1, I $\geq$ min_int
41	BinarySparseInstance	I $\leq$ -1, I $\geq$ -15	I $\leq$ -1, I $\geq$ -506	I $\leq$ -1, I $\geq$ min_int
42	SoftReferenceSymbolTable	I $\geq$ 2147483635	I $\geq$ 2147483140	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
43	HTMLEntities	I $\leq$- 1	I $\geq$ -504, I $\leq$ -405,	I $\leq$ -809, I $\leq$ -607, I $\geq$ -605,
		I $\geq$ -17	I $\geq$ -403, I $\leq$ -304,	I $\leq$ -506, I $\geq$ -504, I $\leq$ -405,
			I $\geq$ -302, I $\leq$ -203,	I $\geq$ -403, I $\leq$ -304, I $\geq$ -302,
			I $\geq$ -201, I $\leq$ -102,	I $\leq$ -203, I $\geq$ -201, I $\leq$ -102,
			I $\geq$ -100, I $\leq$ -1	I $\geq$ -100, I $\leq$ -1
44	SymbolHash	I $\leq$ -1, I $\geq$ -16	I $\leq$ -2147483592	I $\leq$ -1,
			I $\geq$ min_int	I $\geq$ min_int
45	SynchronizedSymbolTable	I $\leq$ -2147483140	I $\leq$ -2147483592,	I $\leq$ -1, I $\geq$ min_int
		I $\geq$ min_int	I $\geq$ min_int	
46	XMLChar	I $\leq$ -1, I $\geq$ -12	I $\leq$ -1, I $\geq$ -510	I $\leq$ -1, I $\geq$ min_int
47	XMLGrammarPoolImpl	I $\leq$ -1, I $\geq$ -13	I $\leq$ -2147483137	I $\leq$ -1,
			I $\geq$ min_int	I $\geq$ min_int
48	XML11Char	I $\leq$ -1, I $\geq$ -16	I $\leq$ -1, I $\geq$ -512	I $\leq$ -1, I $\geq$ min_int
49	AttributeList	I $\geq$ 2147483635	I $\geq$ 2147483590	I $>$ 698000000
		I $\leq$ max_int	I $\leq$ max_int	I $\leq$ max_int
50	ClassLoaderResolver	I $\geq$ 2,	I $\geq$ 500, I $\leq$ -2	I $\leq$ -2, I $>$min_int
		I $\leq$ 18	I $\geq$ 2, I $\leq$ 505	I $\geq$ 2, I $\leq$ max_int