Finding Failure-domains: Automated Testing vs. Manual Testing

Mian Asbat Ahmad and Manuel Oriol

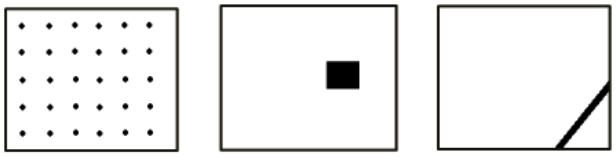
***Abstract*—The achievement of up-to 50% better results by Adaptive Random Testing verses Random Testing ensures that the pass and fail domains across the input domain are highly useful and need due consideration in selection of test inputs. The Automated Discovery of Failure Domain (ADFD) and its successor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

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

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

I. Introduction

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



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

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

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

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

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

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

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

The analysis of failures and failure-domains discovered in 57 classes from 25 open source Java projects of Qualitas Corpus through three different techniques—ADFD, ADFD+ and Manual testing—reveals that each is good at uncovering different type of failure-domains and each brings distinct contributions.

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

II. Automated Discovery of Failure Domain+

It is an improved version of ADFD technique developed earlier by Ahmad and Oriol [5]. The technique automatically finds failures, failure domains and present the results in graphical form. In this technique, the test execution is initiated by random+ and continues till the first failure is found in the SUT. The technique then copies the values leading to the failure and the surrounding values to the dynamic list of interesting values. The resultant list provides relevant test data for the remaining test session and the generated test cases are effectively targeted towards finding new failures around the existing failures in the given SUT.
The improvements made in ADFD+ over ADFD technique are stated as follows.

- ADFD+ generates a single Java file dynamically at run time to plot the failure domains as compared to one Java file per failure in ADFD. This saves sufficient time and makes the execution process quicker.
- ADFD+ uses (x, y) vector-series to represent failure domains as opposed to the (x, y) line-series in ADFD. The vector-series allows more flexibility and clarity to represent failure and failure domains.
- ADFD+ takes a single value for the radius within which the strategy searches for a failure domain whereas ADFD takes two values as lower and upper bounds representing x and y-axis respectively. This results in consumption of lower number of test cases for detecting failure domain.
- In ADFD+, the algorithm of dynamically generating Java file at run-time has been made simplified and efficient as compared to ADFD resulting in reduced overhead.
- In ADFD+, the point, block and strip failure domains generated in the output graph present a clear view of pass and fail domains with individually labelled points of failures as against a less clear view of pass and fail domains and lack of individually labelled points in ADFD.

A. Workflow of ADFD+

ADFD+ is a fully automatic technique requiring the user to select radius value and feed the program under test followed by clicking the $DrawFaultDomain$ button for test execution. As soon as the button is clicked, YETI comes in to play with ADFD+ strategy to search for failures in the program under test. On finding a failure, the strategy creates a Java file which contains calls to the program on the failing and surrounding values within the specified radius. The Java file is executed after compilation and the results obtained are analysed to separate pass and fail values which are accordingly stored in the text files. At the end of test, all the values are plotted on the graph with pass values in blue and fail values in red colour as shown in Figure **??**.

B. Implementation of ADFD+

The ADFD+ technique is implemented in YETI which is available in open-source at http://code.google.com/p/yeti-test/.

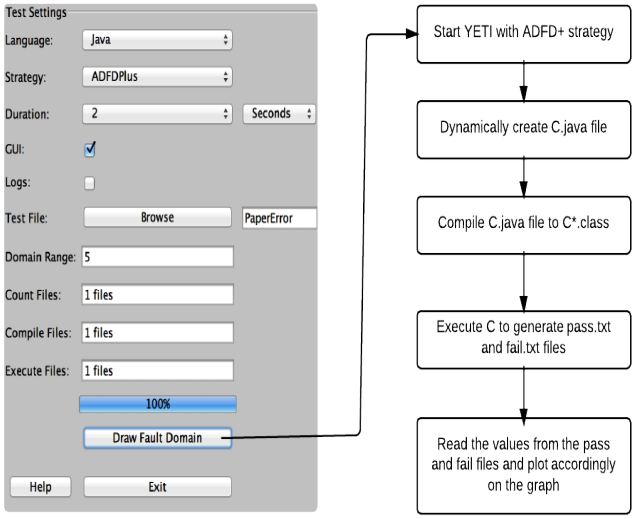


Fig. 2. Workflow of ADFD+

A brief overview of YETI is given with the focus on parts relevant to implementation of ADFD+ strategy.
YETI is a testing tool developed in Java for automatic testing of programs using random strategies. YETI meta-model is language-agnostic which enables it to test programs written in functional, procedural and object-oriented languages. YETI consists of three main parts including core infrastructure for extendibility, strategies section for adjustment of multiple strategies and languages section for supporting multiple languages. Both strategies and languages sections have pluggable architecture for easily incorporating new strategies and languages making YETI a favourable choice for implementing ADFD+ strategy. YETI is also capable of generating test cases to reproduce the failures found during the test session. The strategies section in YETI contains different strategies including random, random+, DSSR and ADFD for selection according to specific needs. ADFD+ strategy is implemented in this section by extending the $YetiADFDStrategy$.

C. Example to illustrate working of ADFD+

Suppose we have the following error-seeded class under test. It is evident from the program code that an $ArithmeticException$ (divison by zero) failure is generated when the value of variable x ranges between 5 to 8 and the value of variable y between 2 to 4.

```
public class Error {
  public static void Error (int x, int y){
    int z;
    if (((x>=5)&&(x<=8))&&((y>=2)&&(y<=4)))
          {
               z = 50/0;
          }
    }
}
```

At the beginning of the test, ADFD+ strategy evaluates the given class with the help of YETI and finds the first failure at x = 6 and y = 3. Once a failure is identified ADFD+ uses the surrounding values around it to find a failure domain. The radius of surrounding values is limited to the value set by the user in the $DomainRange$ variable. When the value of $DomainRange$ is set to 5, ADFD+ evaluates a total of 83 values of x and y around the found failure. All evaluated (x, y) values are plotted on a two-dimensional graph with red filled circles indicating fail values and blue filled circles indicating pass values. Figure **??** shows that the failure domain forms a block pattern and the boundaries of the failure are $(5, 2), (5, 3), (5, 4), (6, 2), (6, 4), (7, 2), (7, 4), (8, 2), (8, 3), (8, 4)$.

III. Evaluation

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

A. Research questions

The following research questions have been addressed in the study:

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

B. Experimental setup

All experiments were conducted with a 64-bit Mac OS X Mountain lion version 10.8.5 running on 2.7 GHz Intel Core i7 with 16 GB (1600 MHz DDR3) of RAM. YETI runs on top of the Java™SE Runtime Environment [version 1.6.0_35]. The ADFD+ Jar file is available at https://code.google.com/p/yeti-test/downloads/list/ and Randoop at https://randoop.googlecode.com/files/randoop.1.3.3.zip.

IV. Experimental results

A. Efficiency

Figure **??** shows the comparative efficiency of ADFD+ and Randoop. The $x - axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y - axis$ represents average time taken by the tools to detect the failure domains. As shown in the figure ADFD+ showed extra ordinary efficiency by taking two orders of magnitude less time to discover failure domains as compared to Randoop.

This may be partially attributed to the very fast processing of YETI, integrated with ADFD+. YETI is capable of executing 10^6 test calls per minute on Java code. To counter the contribution of YETI and assess the performance of ADFD+ by itself, the effectiveness of ADFD+ was compared with Randoop in terms of the number of test cases required to identify the failure domains without giving any consideration to the time consumed for completing the test session. The results are presented in the following section.

B. Effectiveness

Figure **??** shows the comparative effectiveness of ADFD+ and Randoop. The $x - axis$ represents one and two-dimensional programs with point, block and strip failure domains while the $y - axis$ represents average number of test

cases used by the tools to detect the failure domains. The figure shows higher effectiveness in case of ADFD+, amounting to 100% or more. The higher effectiveness of ADFD+ may be attributed to its working mechanism in comparison with Randoop for identifying failures. ADFD+ dynamically changes its algorithm to exhaustive testing in a specified radius around the failure as against Randoop which uses the same random algorithm for searching failures.

C. Failure Domains

The comparative results of the two tools with respect to presentation of the identified failure domains reveal better performance of ADFD+ by providing the benefit of presenting the failure domains in graphical form as shown in Figure **??** and **??**. The user can also enable or disable the option of showing the failing values on the graph. In comparison Randoop lacks the ability of graphical presentation and the option of showing the failure domains separately and provides the results scattered across the textual files.

V. Discussion

The results indicated that ADFD+ is a promising technique for finding failure and failure domain efficiently and effectively. It has the added advantage of showing the results in graphical form. The pictorial representation of failure domains facilitates the debuggers to easily identify the underlying failure domain and its boundaries for troubleshooting.

In the initial set of experiments Randoop was executed for several minutes with default settings. The results indicated no identification of failures after several executions. On analysis of the generated unit tests and Randoop's manual, it was found that the pool of values stored in Randoop database for int primitive type contains only 5 values including -1, 0, 1, 10 and 100. To enable Randoop to select different values, we supplied a configuration file with the option to generate random values between -500 and 500 for the test cases as all the seeded errors were in this range.

As revealed in the results ADFD+ outperformed Randoop by taking two orders of magnitude less time to discover the failure domains. This was partially attributed to the very fast processing of YETI integrated with ADFD+. To counter the effect of YETI the comparative performance of ADFD+ and Randoop was determined in terms of the number of test cases required to identify the failure domains giving no consideration to the time taken for completing the test session. As shown in the results ADFD+ identified all failure domains in 50% or less number of test cases.

The ADFD+ was found quite efficient and effective in case of block and strip domains but not so in case of point domains where the failures lied away from each other as shown in the following code. This limitation of ADFD+ may be due to the search in vain for new failures in the neighbourhood of failures found requiring the additional test cases resulting in increased overhead.

```
public class Error {
  public static void Error (int x, int y){
    int z;
    if (x == 10000)
         {  z = 50/0;   }

    if (y == -2000)
         {  z = 50/0;   }
   }
}
```

The number of test cases to be undertaken in search of failures around the previous failure found is set in the range value by the user. The time taken by test session is directly proportional to the range value. Higher range value leads to larger graphical output requiring zoom feature which has been incorporated in ADFD+ for use when the need arise.

VI. Threats to validity

The study faces threats to external and internal validity. The external threats are common to most of the empirical evaluations. It includes the extent to which the programs under test the generation tools and the nature of seeded errors are representative of the true practice. The present findings will serve as foundation for future research studies needed to be undertaken with several types of classes, test generation tools and diversified nature of seeded errors in order to overcome the threats to external validity. The internal threats to validity includes error-seeded and limited number of classes used in the study. These may be avoided by taking real and higher number of classes in future studies.

VII. Related Work

The increase in complexity of programs poses new challenges to researchers for finding more efficient and effective ways of software testing with user friendly easy to understand test results. Adaptive Random Testing [2], Proportional random testing [1] and feedback directed random testing [12] are some of the prominent upgraded versions of random testing with better performance. Automated random testing is simple to implement and capable of finding hitherto bugs in complex programs [13], [14]. ADFD+ is a promising technique for finding failures and failure domains efficiently and effectively with the added advantage of presenting the output in graphical form showing point, block and strip domains.

Some previous research studies have reported work on Identification, classification and visualisation of pass and fail domains in the past [15], [16], [17]. This includes Xslice [15] is used to differentiate the execution slices of passing and failing part of a test in a visual form. Another tool called Tarantula uses colour coding to track the statements of a program during and after the execution of the test suite [16]. Hierarchical Multi Dimension Scaling (HMDS) describes a semi-automated procedure of classifying and plotting the faults [17]. A serious limitation of the above mentioned tools is that they are not fully automated and require human intervention during execution. Moreover these tools need the requirement

of existing test cases to work on where as ADFD+ strategy generates test cases, discovers failures, identifies pass and fail domains and visualises the results in a graphical form operating in fully automated manner.

VIII. Conclusion

The newly developed ADFD+ technique is distinct from other random testing techniques because it not only identifies failures but also discovers failure domains and provides the result output in easily understandable graphical form. The paper highlights the improved features of ADFD+ in comparison with ADFD technique previously developed by our team [5]. The paper then analyses and compares the experimental results of ADFD+ and Randoop for the point, block and strip failure domains. The ADFD+ demonstrated extra ordinary efficiency by taking less time to the tune of two orders of magnitude to discover the failure domains and it also surpassed Randoop in terms of effectiveness by identifying the failure domains in 50% or less number of test cases. The better performance of ADFD+ may be attributed mainly to its ability to dynamically change algorithm to exhaustive testing in a specified radius around the first identified failure as against Randoop which uses the same random algorithm continuously for searching failures.

IX. Future Work

The ADFD+ strategy is capable of testing numerical programs and needs to be extended for testing of non numerical and reference data types to enable it to test all types of data. ADFD+ has the capability of graphical presentation of results for one and two-dimensional numerical programs. It is worthwhile to extend the technique to enable it to present the results of multi-dimensional numerical and non numerical programs in the graphical form.

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

References

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

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

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

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

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

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

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

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

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

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

[11] M. Oriol, "Random testing: Evaluation of a law describing the number of faults found," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, april 2012, pp. 201 –210.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 75–84. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.37

[13] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for Java," *Software—Practice & Experience*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004.

[14] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *In 19th European Conference Object-Oriented Programming*, 2005, pp. 504–527.

[15] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*. IEEE, 1995, pp. 143–151.

[16] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*. ACM, 2002, pp. 467–477.

[17] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

TABLE I
TABLE DEPICTING RESULTS OF ADFD AND ADFD+

S#	Project	Class	Method	ADFD	ADFD+	Manual
1	ant	LeadPipeInputStream	LeadPipeInputStream(i)	I >= 2147483140 I <= 2147483647	I	I > 698000000
2	antlr	BitSet	BitSet.of(i,j)	I <= -1, i >= -18, j <= 7, j >= -12	I one of {-513, -1} J one of {-503, 507}	I <= -1 J != 0
3	artofillusion	ToolPallete	ToolPalette(i,j)	I <= -1, I >= -18	I one of { -515, -1} J one of {-509, 501}	I <= -1, J any value
		IntMap	idMap(i)	I != 0 I <= -1, I >= -18	I one of {-1, -512}	I <= -1
4	cayenne	ExpressionFactory	expressionOfType(i)	I <= 13, I >= -7	I one of {-497, 513}	I >= -2147483648 I <= 2147483647
5	collections	ArrayStack	ArrayStack(i)	I >= 2147483636 I <= 2147483647	I one of {2147483142, 2147483647}	I > 698000000
		BinaryHeap	BinaryHeap(i)	I <= -2147483637 I >= -2147483648	I one of {-2147483648, -2147483142}	I <= 0
		BondedFifoBuffer	BoundedFifoBuffer(i)	I <= -2147483639 I >= -2147483648	I one of {-505, 0}	I <= 0
		FastArrayList	FastArrayList(i)	I <= -2147483641 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
		StaticBucketMap	StaticBucketMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483140, 2147483647}	I > 698000000
		PriorityBuffer	PriorityBuffer(i)	I != 0 I <= -1, I >= -14	I one of {-2147483647, -2147483142}	I <= 0
6	colt	GenericPermuting	permutation(i,j)	I <= 0, I >= -18	I one of { -498, 0} I one of {2, 512}	I <= 0, I >= 2 j != 0
		LongArrayList	LongArrayList(i)	I <= -2147483640 I >= -2147483648	I one of {-510, -1}	I <= -1
		OpenIntDoubleHashMap	OpenIntDoubleHashMap(i)	I <= -1 I >= -17	I one of {-514, -1}	I <= -1
	drjava	ByteVector	ByteVector(i)	I <= -2147483639 I >= -2147483648	I one of {-2147483648, -2147483141}	I <= -1
	emma	ElementFactory	newConstantCollection(i)	I >= 2147483636 I <= 2147483647	I one of {2147483141, 2147483647}	I > 698000000
		IntIntMap	IntIntMap(i)	I <= -2147483638 I >= -2147483648	I one of {-2147483644, -2147483139}	I <= -1
		ObjectIntMap	ObjectIntMap(i)	I >= 2147483640 I <= 2147483647	I one of {2147483591, 2147483647}	I > 698000000
		IntObjectMap	IntObjectMap(i)	I <= -1 I >= -17	I <= -1 I >= -518	I <= -1
7	heritrix	ArchiveUtils	padTo(i,j)	I >= 2147483641 I <= 2147483647	I one of {-497, 513} j one of {2147483591, 2147483647}	I any value J > 698000000
		BloomFilter32bit	BloomFilter32bit(i,j)	I <= -1 I >= -18	I one of {-515, -1} J may be any value	I <-1 J <-1
8	hsqld	IntKeyLongValueHashMap	IntKeyLongValueHashMap(i)	I >= 2147483635 I <= 2147483647	I one of {2147483590, 2147483647} I >= -518	I > 698000000
		ObjectCacheHashMap	ObjectCacheHashMap(i)	I <= -2147483641 I >= -2147483648	I one of {-512, 0}	I <= 0
9	htmlunit	ObjToIntMap	ObjToIntMap(i)	I <= -2147483636 I >= -2147483648	I one of {-2147483646, -2147483137}	I <= -1
10	itext	PRTokeniser	isDelimiterWhitespace(i)	I <= -2 I >= -18	I one of {-509, -2} I one of {256, 501}	I <= -2 I >= 256
		PdfAction	PdfAction(i)	I <= -2147483640 I >= -2147483648	I one of {-514, 0} I one of {6, 496}	I <= 0 I >= 6
		PdfLiteral	PdfLiteral(i)	I <= -1 I >= -14	I one of {-511, -1}	I <= -1
11	jung	PhysicalEnvironment	PhysicalEnvironment(i)	I <= -1 I >= -11	I one of {-2147483646, -2147483137}	I <= -1
12	jedit	IntegerArray	IntegerArray(i)	I >= 2147483636 I <= 2147483647	I one of {2147483587, 2147483647}	I > 698000000
13	jgraph	AttributeMap	AttributeMap(i)	I <= -2147483639 I >= -2147483648	I one of {-514, 0}	I <= 0
14	jruby	ByteList	ByteList(i)	I <= -1 I >= -14	I one of {-513, -1}	I <= -1
		WeakIdentityHashMap	WeakIdentityHashMap(i)	I >= 2147483636 I <= 2147483647	I one of {2147483140, 2147483647}	I >698000000
15	megamek	AmmoType	getMunitionsFor(i)	I <= -1 I >= -17	I one of {-514, -1} I one of {93, 496}	I <= -1 I >= 93
16	poi	IntList	IntList(i,j)	I <= -1 I >= -15	I one of {-1, -509} j one of 0	I <= -1 j = 0
17	sunflow	QMC	halton(i,j)	I <= -1, I >= -12 J <= -1, J >= -15	I <= -1, I >= -508 j <= 499, j >= -511	I <= -1 j any value
		BenchmarkFramework	BenchmarkFramework(i,j)	I != 0 I <= -1, I >= -13	I one of {-501, -1}	I <= -1
		IntArray	IntArray(i)	I <= -1 I >= -16	I one of {-2147483650, -2147483141}	I <= -1
18	trove	TDoubleStack	TDoubleStack(i)	I <= -1 I >= -13	I one of {-511, -1}	I <= -1
		TIntStack	TIntStack(i)	I <= -1 I >= -12	I one of {-2147483648, -2147483144}	I <= -1

TABLE II
TABLE DEPICTING RESULTS OF ADFD AND ADFD+

S#	Project	Class	Method	ADFD	ADFD+
1	drjava	Assert	assertEquals(i,j)		I != j
2	junit	Assert	assertEquals(i,j)		I != j
	megamek	Board	getTypeName(i)		I ¡= -910 I ¿= -908 I ¡= -809 I ¿= -807 I ¡= -708 I ¿= -706 I ¡= -607 I ¿= -605 I ¡= -506 I ¿= -504 I
3	nekohtml	HTMLEntities	get(i)		I ¡= -910 I ¿= -908 I ¡= -809 I ¿= -807 I ¡= -708 I ¿= -706 I ¡= -607 I ¿= -605 I ¡= -506 I ¿= -504 I

TABLE III
TABLE DEPICTING RESULTS OF ADFD AND ADFD+

S#	Project	Class	Method	ADFD
1	aspectj	AnnotationValue	whatKindIsThis(i)	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 ¿= 11
2	htmlunit	Token	typeToName(i)	

TABLE IV
TABLE DEPICTING RESULTS OF ADFD AND ADFD+

S#	Project	Class	Method	ADFD	ADFD+	Manual
1	emma	ClassLoaderResolver	getCallerClass(i)		I ¡= -2 I ¿= 2	
2	poi	Variant	getVariantLength(i)		I ¿= 0 I ¡= 14 I ¿= 16 I ¡= 31 I ¿= 64 I ¡= 72	