New Strategies for Automated Random Testing

Mian Asbat Ahmad

Doctor of Philosophy

Department of Computer Science

The University of York

September 2014

Abstract

Testing a software with all permutations and combinations of the inputs is not practically feasible because of the availability of infinite possible scenarios (a software can be/accept). Alternatively, a strategy is used to select a small subset of inputs from the input domain for testing the software. Random strategy is one possible/(of the) option which can generate comparatively cheap test inputs without requiring/(the need of) too much intellectual and computational efforts. The random generation of test inputs is quick and free from human bias.

However, without (making use of)/utilizing any available information to guide and generating test input arbitrarily may be effective in some cases but generally it results in a lot of vague or unneeded test inputs.

The strategy which uses a truly representative subset of the input domain increases the chances of detecting higher number of failure in the software.

This thesis presents new automated random testing strategies developed by manipulating the patterns of failure domains within the input domain. The strategies have been experimentally evaluated for effectiveness and efficiency. The characteristics of failure domains and their influence on the performance of the test strategies has been examined. The new Dirt Spot Sweeping Random (DSSR) strategy adds the value causing the failure and its neighbouring values to the list of interesting values for exploring the underlying failure domain. The comparative evaluation showed significantly better performance of DSSR over R and R+ strategies. The new Automated Discovery of Failure Domain (ADFD) strategy finds failure, failure domains and presents the pass and fail domains in graphical form. The results indicated highly effective performance of the strategy in identifying failure, failure domains and their graphical presentation. The new Automated Discovery of Failure Domain+ (ADFD+) strategy is the upgraded version of ADFD strategy with respect to algorithm and graphical representation of failure domains. On comparison with Randoop, ADFD+ strategy successfully detected all failures and failure domains as against Randoop which identified individual failures but was unable to detect the failure domains. To find the comparative effectiveness of ADFD and ADFD+ strategies, extensive experimental analysis of Java projects contained in Qualitas Corpus was

carried out in comparison with manual technique. The results revealed

Contents

List of Figures

List of Tables

Dedication

I feel it a great honour to dedicate my PhD thesis to my beloved parents, wife and daughter for their significant contribution in achieving the goal of academic excellence.

Acknowledgements

The duration at the University of York for my PhD has been the most joyful and rewarding experience in my academic career. The institution provided me with everything I needed to thrive: challenging research problems, excellent company and supportive environment. I am deeply grateful to all those who shared this experience with me.

Several people have contributed to the completion of my PhD dissertation. The most prominent personality deserving due recognition is my worthy advisor, Dr. Manuel Oriol. Thank you Manuel for your endless help, valuable guidance, constant encouragement, precious advice, sincere and affectionate attitude.

I thank my assessor Prof. Dr. John Clark for his constructive feedback on various reports and presentations. I am highly indebted to Prof. Dr. Richard Paige for his generous help, cooperation and guidance throughout my research.

Thanks to my father Prof. Dr. Mushtaq A. Mian who provided a conducive environment, valuable guidance and crucial support at all levels of my educational career and to my beloved mother whose love, affection and prayers have been my most precious assets. I am also thankful to my brothers Dr. Ashfaq, Dr. Aftab, Dr. Ishaq, Dr. Afaq, and Dr. Ilyas who have been the source of inspiration for me to pursue higher studies. Last but not the least I am thankful to my dear wife Dr. Munazza Asbat for her company, help and cooperation throughout my stay at York.

I obtained Departmental Overseas Research Scholarship which is awarded to overseas students for higher studies on academic merit and research potential. I am truly grateful to the Department of Computer Science, University of York for financial support.

Declaration

I declare that the contents of this thesis derive from my own original research between January 2010 and September 2014, the period during which I was registered for the degree of Doctor of Philosophy at University of York.

Contributions from this thesis have been published in the following papers:

- M. Ahmad and M. Oriol. Dirt Spot Sweeping Random strategy. *Poster presentation in fourth York Doctoral Symposium (YDS 2011)*, York, UK. October 2011.

- M. Ahmad and M. Oriol. Dirt Spot Sweeping Random strategy. *Proceedings of the International Conference on Software and Information Engineering (ICSIE)*, Singapore. Lecture Notes on Software Engineering, Volume 2(4), pp. 294-299, 2014.

  Based on research described in Chapter **??** in the thesis.

- M. Ahmad and M. Oriol. Automated Discovery of Failure Domain. *Proceedings of the International Conference on Software and Computer Applications (ICSCA)*, Paris, France. Lecture Notes on Software Engineering, Volume 1(3), pp. 289-294, 2014.

  Based on research described in Chapter **??** in the thesis.

- M. Ahmad and M. Oriol. ADFD+: An Automatic Testing Technique for Finding and Presenting Failure domains. *Proceedings of the International Conference on Software and Information Engineering (ICSIE)*, Singapore. Lecture Notes on Software Engineering, Volume 2(4), pp. 331-336, 2014.

  Based on research described in Chapter **??** in the thesis.

- M. Ahmad and M. Oriol. Finding the effectiveness of ADFD and ADFD+ techniques. *Proceedings of Seventh York Doctoral Symposium (YDS 2014)*, York, UK. October 2014. (in process)

  Based on research described in Chapter 1 in the thesis.

Chapter 1

Evaluation of ADFD and ADFD$^+$

The newly developed ADFD and ADFD$^+$ techniques have been described in detail in the preceding chapters (**??** and **??**). Experimental evaluation of the two techniques through purpose built error-seeded numerical programs presented in the two chapters revealed that both techniques were capable of identifying the planted faults effectively. In this chapter, we have evaluated the precision of identifying the failure domains in the two techniques. For this purpose, we have incorporated Daikon in ADFD and ADFD$^+$. Daikon was selected on the basis of its capability to automatically generate invariants of failure domains, precisely point out the boundaries of failure domains and present the failure domains generated in more than two dimensional programs. We have performed extensive experimental analysis of real world Java projects contained in Qualitas Corpus. 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 also studied.

1.1 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` type arguments while it was restricted to `int` type arguments only in the original techniques.
2. Additional information was facilitated by adding the YETI generated failure finding test case to the GUI of the two techniques. Test case includes the name of the failing class and method, values causing the failure and the line number of originating 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 [131]. The generated invariants are displayed in the GUI of the techniques at the end of test execution.
4. The GUI is enabled to launch all the strategies defined in YETI from a single interface. For example, when ADFD$^+$ strategy is selected for testing, the system automatically hides the two fields of lower and upper bounds and displays a single field of range value.
5. Feature of screen capture is added to the new GUI. The user can click the screen capture button to capture the current screen in the test folder. It allows the user to capture multiple screen-shots of results at different intervals for future reference.

Four of the above enhancements are visible from the front-end. As shown in Figure 1.1, the drop down menu for strategy field enables the tester to choose the appropriate strategy in the list for the test session. Secondly, the block failure domain is shown in graphical form and also as invariants due to incorporation of the automatic tool Daikon. Thirdly, the addition of YETI generated test case shows the type of failure, the first value causing the failure and the stack trace of the failure. Fourthly, the provision of screen capture button allows the tester to store the record of each test for record.

1.2 Daikon

Daikon is an automated tool that detects likely invariants at specific points in the program from execution trace file [131]. The trace file records effect of inputs to the program under observation during execution of the test cases. Daikon is capable of generating invariants for programs written in Java, C, C++, Perl and Eiffel. The tool helps programmers by identifying program properties, which must be preserved during modification of the program code. Daikon's output can also be used for assistance in understanding, modifying and testing a program that contains no explicit invariants.

Figure 1.2 presents the architecture of Daikon. To generate invariants Daikon instrument the source code of program by inserting checks at various points in the program. The added checks does not change the original behaviour of the program in any way. On executing the instrumented programs the check points collect values of variables accessible from that point and store them in the trace file. The Daikon's inference engine analyse the trace file for any patterns and reports any patters as invariants, which are observed to be true in all samples. To remove any false positive invariants, confidence of each invariant is calculated and only those invariants are reported which qualify a certain level of confidence.

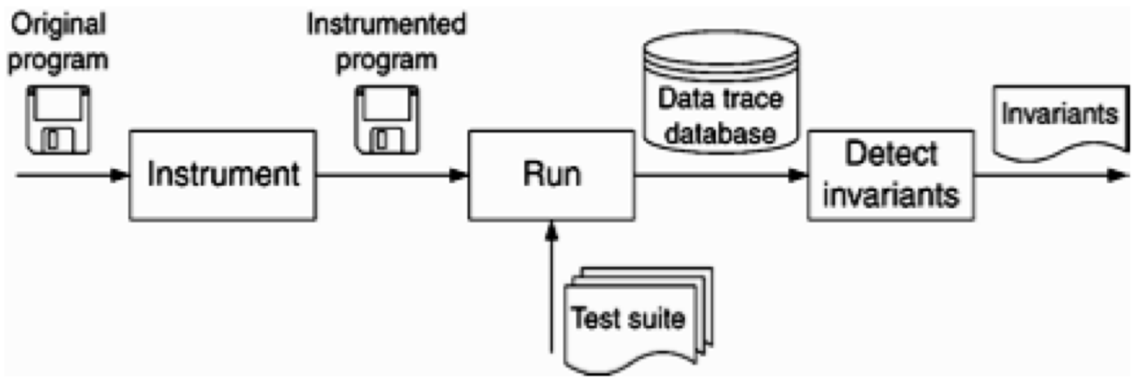
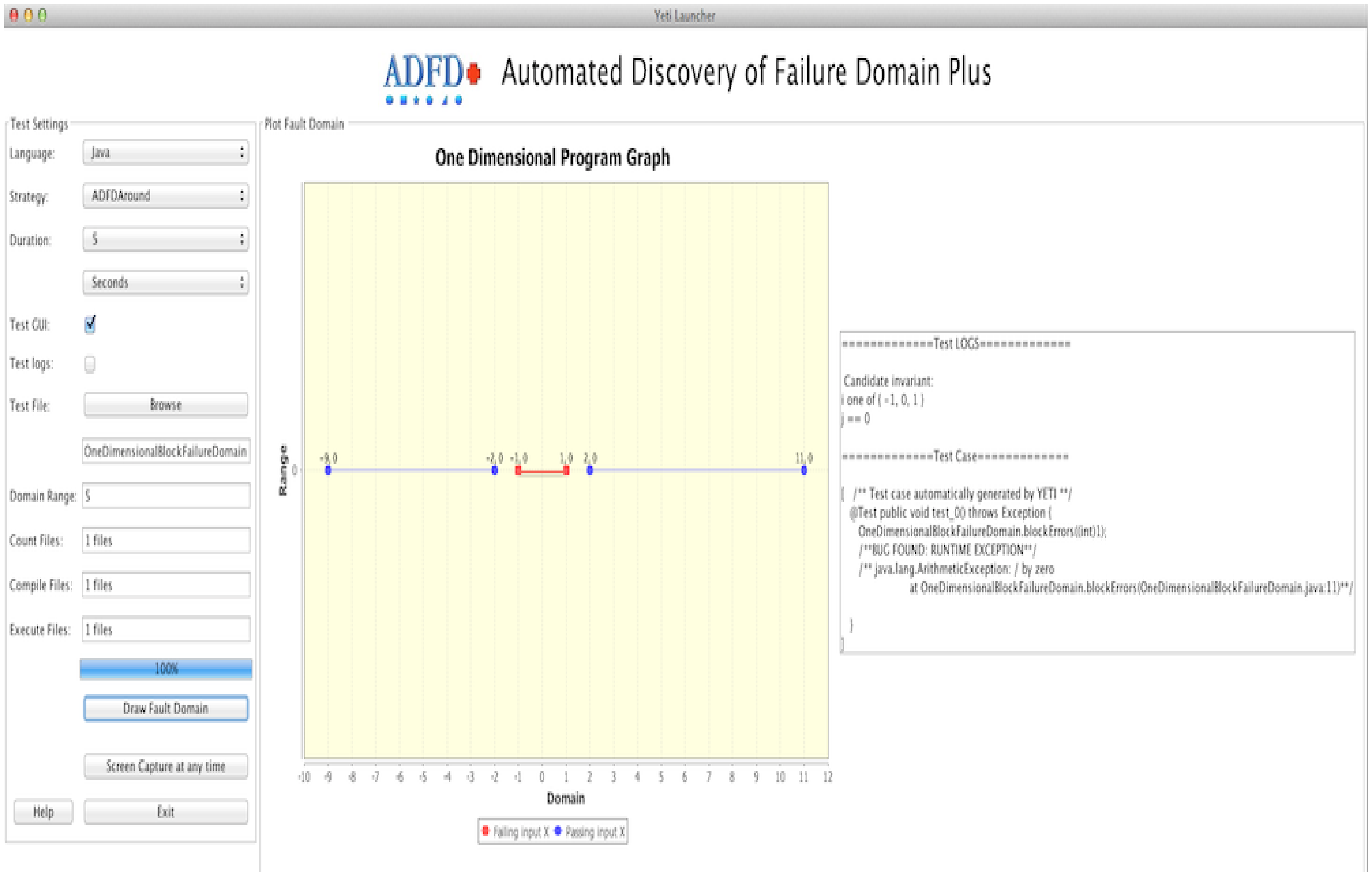


Figure 1.2: Architecture of Daikon [6]

1.2.1 Types of common invariants detected by Daikon

Daikon is capable of detecting common types of likely invariants. It also allows users to write more specific invariants and have Daikon check for them [6]. The most common types of invariants are quoted here from [6], page 102.



Test Settings
Language: Java
Strategy: ADFDAround
Duration: 5 Seconds
Test GUI: ✓
Test logs:
Test File: OneDimensionalBlockFailureDomain
Domain Range: 5
Count Files: 1 files
Compile Files: 1 files
Execute Files: 1 files

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

Candidate invariant:
i one of { -1, 0, 1 }
j == 0

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

{  /** Test case automatically generated by YETI **/
  @Test public void test_0() throws Exception {
    OneDimensionalBlockFailureDomain.blockErrors((int)1);
    /**BUG FOUND: RUNTIME EXCEPTION**/
    /** java.lang.ArithmeticException: / by zero
                at OneDimensionalBlockFailureDomain.blockErrors(OneDimensionalBlockFailureDomain.java:11)**/

  }
}
```

Figure 1.1: GUI front end of upgraded ADFD and ADFD$^+$

1. Invariants over any variable:
    (a) Constant value: x = a indicates the variable is a constant.
    (b) Uninitialized: x = uninit indicates the variable is never set.
    (c) Small value set: $x \in \{a,b,c\}$ indicates the variable takes only a small number of different values.
2. Invariants over a single numeric variable:
    (a) Range limits: $x \geq a$; $x \leq b$; and $a \leq x \leq b$ (printed as x in [a..b]) indicate the minimum and/or maximum value.
    (b) Nonzero: $x \neq 0$ indicates the variable is never set to 0.
    (c) Modulus: $x \equiv a \bmod b$ indicates that x mod b $\equiv$ a always holds.
    (d) Nonmodulus: $x \neq a \bmod b$ is reported only if x mod b takes on every value beside a.
3. Invariants over two numeric variables:
    (a) Linear relationship: y = ax + b.
    (b) Ordering comparison.
    (c) Invariants over x + y: any invariant from the list of invariants over a single numeric variable.
    (d) Invariants over x - y: as for x + y
4. Invariants over a single sequence variable (arrays):
    (a) Range: minimum and maximum sequence values, ordered lexicographically.
    (b) Element ordering: whether the elements of each sequence are non-decreasing, non-increasing or equal.
    (c) Invariants over all sequence elements (treated as a single large collection): for example, all elements of an array are at least 100.
5. Invariants over two sequence variables:
    (a) Linear relationship: y = ax + b, elementwise.
    (b) Comparison: lexicographic comparison of elements.
    (c) Subsequence relationship.

1.3 Difference in working mechanism of the two techniques

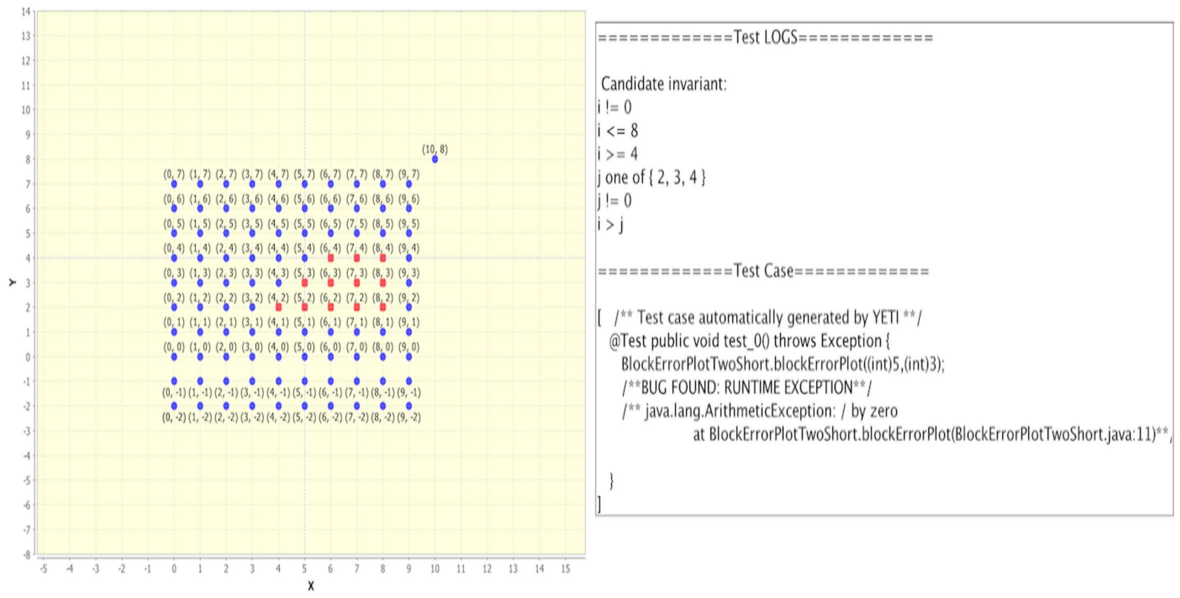
The difference with respect to the identification of failure domains is illustrated by testing a simple Java program (given below) with ADFD and ADFD$^+$ techniques.

```java
/**
* 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 */
        }
    }
}
```

As evident from the program code that a failure 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.

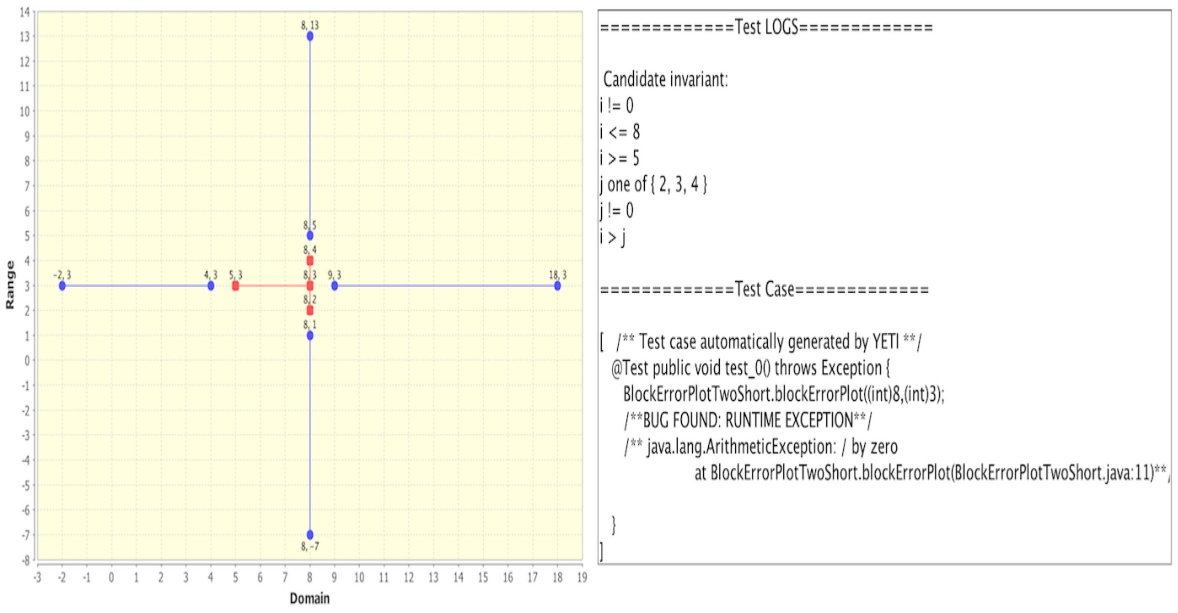
The test output generated by ADFD$^+$ technique is presented in Figure 1.3. The labelled graph correctly shows all the 12 out of 12 available failing values in red whereas the passing values are shown in blue. The invariants correctly represent the failure domain. The test case shows the type of failure, the values causing the first failure and the stack trace of the failure.

The test output generated by ADFD technique is presented in Figure 1.4. The labelled graph correctly shows the 4 out of 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.



```
=============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)**/

   }
]
```

Figure 1.3: Graph, Invariants and Test case generated by ADFD$^+$ for the given code



```
=============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)**/

   }
]
```

Figure 1.4: Graph, Invariants and test case generated by ADFD for the given code

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

1.4 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 automated testing techniques?

1.5 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.

1.5.1 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, it is specially built for empirical research, which takes into account a large number of developmental models and programming styles. It's all included packages are open source with an easy access to the source code.

Since YETI tests the byte code only, 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 500 hours to perform the complete 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 [104, 129]. The source code of the programs containing failure domains were also evaluated manually to cross-examine the experimental results. All experiments were conducted with a 64-bit Mac OS X Mountain lion version 10.8.5 running on 2.7 GHz Intel Core i7 with 16 GB (1600 MHz DDR3) of RAM. YETI runs on top of the Java^{TM}SE Runtime Environment [version 1.7.0_45]. The ADFD and ADFD$^+$ executable files are available at `https://code.google.com/p/yeti-test/downloads/list/`. Daikon and JFreeChart can be seperately obtained from `http://plse.cs.washington.edu/daikon/` and `http://www.jfree.org/jfreechart/download.html` respectively.

1.5.2 Results

Among 106 packages, we found 25 packages containing 57 classes with different types of failure domains. The project name, class name, method name, number of arguments, line of code and the type of detected failure domains in each case is given in Table 1.2. In accordance with Chan et al. [1], classification of failure domain into various types is based on the number of contiguous failures detected in the input-domain as shown in Column 2 of Table 1.1. 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 1.1: Results of the experiments

Type of failure domain	No of 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$^+$
Point	01 to 05	2	2	2	2	2	0	0	0
Block	06 to 49	1	1	0	1	1	1	0	0
Strip	50 & above	50	50	50	45	48	0	5	2
Mix	point & block	0	0	0	0	0	0	0	0
	point & strip	3	3	3	0	2	0	3	1
	point, block & strip	1	1	1	0	0	0	1	1
Total	-	57	57	57	48	53	1	9	4

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 types of failure domains 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. Further explanation is available in the Section 1.5.2.3. The details of results, separated by the type of identified failure domains, are given in Table 1.3, 1.4, 1.5 and 1.6. 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 manual analysis. The key research questions identified in the previous section are individually addressed in the following.

1.5.2.1 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 the 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 fewer resources and is thus capable of handling greater range value as compared to ADFD$^+$ technique.

1.5.2.2 Type and Frequency of Failure domains

As evident from the results given in Table 1.3 - 1.6, 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 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.

1.5.2.3 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 the 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 Table 1.1.

The analysis of generated invariants indicated that the failure domains which are simple in nature are easily detectable by both automated and manual techniques. 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 $\{i \leq -1, i \geq -18\}$, by ADFD$^+$ are $\{i \leq -1, i \geq -512\}$ and by Manual Analysis are $\{i \leq -1, i \geq \mathit{Integer.MIN\_INT}\}$. These results indicate maximum degree of representation of failure domain by Manual Analysis followed by ADFD and ADFD$^+$ respectively.

As an example of complex failure, we consider the results of ADFD, ADFD+ and Manual Analysis in Table 1 for class ArrayStack. The OutOfMemoryError failure is detected due to the input of value to the method ArrayStack(i). The invariants generated by ADFD are $\{ i \geq 2147483636, I \leq 2147483647\}$, by ADFD+ are $\{ i \geq 698000000, i \leq 698000300\}$, by Manual analysis $\{ i \geq 698000000 \}$. All the three strategies indicate the same failure but at different intervals. ADFD+ is unable to show the starting point of failure. On increasing the range value ADFD easily discovered the breaking point while it took over 50 attempts to find the breaking point by Manual testing.

1.6 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 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.

Another threat to validity is related to the hardware and software resources. For example, the OutOfMemoryError occurs 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.

1.7 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. [132] the boundary values located at the edge of domains have more chances of forming strip failure domain. Finally [133] and Bishop [134] 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 [77], Unix [76], Java Libraries [3], Heskell [92] and Mac OS [78]. Its ability to become fully automated makes it one of the best choice for automated testing tools [2, 3]. AutoTest [112], Jcrasher [2], Eclat [3], Jartege [90], Randoop [84] and YETI [104, 129, 135] 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 [100].

In the previous studies, we described the fully automated techniques ADFD [129] and

ADFD$^+$ [135] for the discovery of failure domains and performed its experimental evaluation. The programs used in the evaluation were error-seeded one and two-dimensional numerical 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.

Our current approach of evaluation is inspired from several studies [40, 41, 118] in which random testing have been compared with other testing techniques to find their respective failure finding ability. The comparison of automated technique with manual technique, although not very common, has been performed previously [4, 50]. The feature which makes our research study unique is that, we compared the effectiveness of the techniques in identifying of failure domain and not an individual failure.

1.8 Summary

The chapter examined the effectiveness of ADFD and ADFD$^+$ techniques by analysing the comparative performance of the two techniques on production applications from Qualitas Corpus. The involved classes were further examined and cross-checked by manual testing. The results reveal that the two techniques can effectively identify and present all types of failure domains to a certain degree of accuracy. We further explain how the degree of accuracy can be increased in ADFD and ADFD$^+$ techniques. During analysis the impact of nature, type and size of failure domains on automated testing, it was found that the strip failure domain, with large size and low complexity are quickly identified by automated techniques whereas small sized and non trivial point and block failure domains are difficult to identify by both automated and manual techniques. Based on the results, it can be stated that automated techniques (ADFD and ADFD+) can be highly effective in assistance to manual testing but are not an alternative to manual testing.

Table 1.2: 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

Table 1.3: 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 $\geq$ -2147483648 J = 0

Table 1.4: 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 $\geq$ -2147483648, J = 0	I any value J = 0

Table 1.5: 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 $\geq$ -2147483648	I $\leq$ -2, I $\geq$ -510 I = {73, 156} I $\geq$ 162, I $\leq$ 500	I $\leq$ -2, I >-2147483648 I = 73, 156, I $\geq$ 162, I $\leq$ 2147483647
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 1.6: Classes with strip failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
1	LeadPipeInputStream	$I \geq 2147483140$ $I \leq 2147483647$	$I \geq 2147483143$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
2	BitSet	$I \leq -1, I \geq -18,$ $J \leq 7, J \geq -12$	$I \leq -1, I \geq -513$ $J \geq -503, J \leq 507$	$I \leq -1, I \geq -2147483648$ J any value
3	ToolPallete	$I \leq -1, I \geq -18$ $J \leq 3, J \geq -15$	$I \leq -1, I \geq -515$ $J \geq -509, J \leq 501$	$I \leq -1, I \geq -2147483648$ J any value
4	IntMap	$I \leq -1, I \geq -18$	$I \leq -1, I \geq -512$	$I \leq -1, I \geq -2147483648$
5	ExpressionFactory	$I \leq 13, I \geq -7$	$I \geq -497, I \leq 513$	$I \geq -2147483648$ $I \leq 2147483647$
6	ArrayStack	$I \geq 2147483636$ $I \leq 2147483647$	$I \geq 2147483142$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
7	BinaryHeap	$I \leq -2147483637$ $I \geq -2147483648$	$I \leq -2147483142$ $I \geq -2147483648$	$I \leq 0$ $I \geq -2147483648$
8	BondedFifoBuffer	$I \leq -2147483639$ $I \geq -2147483648$	$I \geq -505, I \leq 0$	$I \leq 0$ $I \geq -2147483648$
9	FastArrayList	$I \leq -2147483641$ $I \geq -2147483648$	$I \leq -2147483644,$ $I \geq -2147483139$	$I \leq -1$ $I \geq -2147483648$
10	StaticBucketMap	$I \geq 2147483635$ $I \leq 2147483647$	$I \geq 2147483140$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
11	PriorityBuffer	$I \leq -1, I \geq -14$	$I \leq -2147483142$ $I \geq -2147483647$	$I \leq 0$ $I \geq -2147483648$
12	GenericPermuting	$I \leq 0, I \geq -18$	$I \geq -498, I \leq 0$ $I \geq 2, I \leq 512$	$I \leq 0, I \geq -2147483648$ $I \geq 2, I \leq 2147483647$
13	LongArrayList	$I \leq -2147483640$ $I \geq -2147483648$	$I \leq -1, I \geq -510$	$I \leq -1$ $I \geq -2147483648$
14	OpenIntDoubleHashMap	$I \leq -1, I \geq -17$	$I \leq -1, I \geq -514$	$I \leq -1, I \geq -2147483648$
15	ByteVector	$I \leq -2147483639$ $I \geq -2147483648$	$I \leq -2147483141$ $I \geq -2147483648$	$I \leq -1$ $I \geq -2147483648$
16	ElementFactory	$I \geq 2147483636$ $I \leq 2147483647$	$I \geq 2147483141$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
17	IntIntMap	$I \leq -2147483638$ $I \geq -2147483648$	$I \leq -2147483644$ $I \geq -2147483139$	$I \leq -1$ $I \geq -2147483648$
18	ObjectIntMap	$I \geq 2147483640$ $I \leq 2147483647$	$I \geq 2147483591$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
19	IntObjectMap	$I \leq -1, I \geq -17$	$I \leq -1, I \geq -518$	$I \leq -1, I \geq -2147483648$
20	ArchiveUtils	$I \geq 2147483641$ $I \leq 2147483647$ $J \geq 2147483639$ $J \leq 2147483647$	$I \geq -497$ $I \leq 513$ $J \geq 2147483591$ $J \leq 2147483647$	I any value $J > 698000000$
21	BloomFilter32bit	$I \leq -1, I \geq -18$ J may be any value	$I \leq -1, I \geq -515$ J may be any value	$I < -1$ $J < -1$
22	IntKeyLongValueHashMap	$I \geq 2147483635$ $I \leq 2147483647$	$I \geq 2147483590$ $I \leq 2147483647$	$I > 698000000$ $I \leq 2147483647$
23	ObjectCacheHashMap	$I \leq -2147483641$ $I \geq -2147483648$	$I \geq -512, I \leq 0$	$I \leq 0$ $I \geq -2147483648$
24	ObjToIntMap	$I \leq -2147483636$ $I \geq -2147483648$	$I \leq -2147483646$ $I \geq -2147483648$	$I \leq -1$ $I \geq -2147483648$
25	PRTokeniser	$I \leq -2$ $I \geq -18$	$I \leq -2, I \geq -509$ $I \geq 256, I \leq 501$	$I \leq -2$, $I \geq -2147483648$ $I \geq 256$, $I \leq 2147483647$
26	PdfAction	$I \leq -2147483640$ $I \geq -2147483648$	$I \leq 0, I \geq -514$ $I \geq 6, I \leq 496$	$I \leq 0, I \geq -2147483648$ $I \geq 6, I \leq 2147483647$
27	PdfLiteral	$I \leq -1, I \geq -14$	$I \leq -1, I \geq -511$	$I \leq -1, I \geq -2147483648$

Table 1.6: Classes with strip failure domains

S#	Class	Invariants by ADFD$^+$	Invariants by ADFD	Invariants by Manual
28	PhysicalEnvironment	$I \le -1, I \ge -11$	$I \le -2147483646$ $I \ge -2147483648$	$I \le -1,$ $I \ge -2147483648$
29	IntegerArray	$I \ge 2147483636$ $I \le 2147483647$	$I \ge 2147483587$ $I \le 2147483647$	$I > 698000000$ $I \le 2147483647$
30	AttributeMap	$I \le -2147483639$ $I \ge -2147483648$	$I \le 0, I \ge -514$	$I \le 0$ $I \ge -2147483648$
31	ByteList	$I \le -1, I \ge -14$	$I \le -1, I \ge -513$	$I \le -1, I \ge -2147483648$
32	WeakIdentityHashMap	$I \ge 2147483636$ $I \le 2147483647$	$I \ge 2147483140$ $I \le 2147483647$	$I > 698000000$ $I \le 2147483647$
33	AmmoType	$I \le -1$ $I \ge -17$	$I \le -1, I \ge -514$ $I \ge 93, I \le 496$	$I \le -1, I \ge -2147483648$ $I \ge 93, I \le 2147483647$
34	QMC	$I \le -1, I \ge -12$ $J \le -1, J \ge -15$	$I \le -1, I \ge -508$ $J \le 499, J \ge -511$	$I \le -1, I \ge -2147483648$ J any value
35	BenchmarkFramework	$I \le -1, I \ge -13$	$I \le -1, I \ge -508$	$I \le -1, I \ge -2147483648$
36	IntArray	$I \le -1, I \ge -16$	$I \le -2147483650$ $I \ge -2147483141$	$I \le -1$ $I \ge -2147483648$
37	TDoubleStack	$I \le -1, I \ge -13$	$I \le -1, I \ge -511$	$I \le -1, I \ge -2147483648$
38	TIntStack	$I \le -1, I \ge -12$	$I \le -2147483648$ $I \ge -2147483144$	$I \le -1$ $I \ge -2147483648$
39	TLongArrayList	$I \le -1, I \ge -16$	$I \le -2147483648$ $I \ge -2147483141$	$I \le -1,$ $I \ge -2147483648$
40	AlgVector	$I \le -1, I \ge -15$	$I \le -1, I \ge -511$	$I \le -1, I \ge -2147483648$
41	BinarySparseInstance	$I \le -1, I \ge -15$	$I \le -1, I \ge -506$	$I \le -1, I \ge -2147483648$
42	SoftReferenceSymbolTable	$I \ge 2147483635$ $I \le 2147483647$	$I \ge 2147483140$ $I \le 2147483647$	$I > 698000000$ $I \le 2147483647$
43	HTMLEntities	$I \le - 1$ $I \ge -17$	$I \ge -504, I \le -405,$ $I \ge -403, I \le -304,$ $I \ge -302, I \le -203,$ $I \ge -201, I \le -102,$ $I \ge -100, I \le -1$	$I \le -809, I \le -607, I \ge -605,$ $I \le -506, I \ge -504, I \le -405,$ $I \ge -403, I \le -304, I \ge -302,$ $I \le -203, I \ge -201, I \le -102,$ $I \ge -100, I \le -1$
44	SymbolHash	$I \le -1, I \ge -16$	$I \le -2147483592$ $I \ge -2147483648$	$I \le -1,$ $I \ge -2147483648$
45	SynchronizedSymbolTable	$I \le -2147483140$ $I \ge -2147483648$	$I \le -2147483592,$ $I \ge -2147483648$	$I \le -1, I \ge -2147483648$
46	XMLChar	$I \le -1, I \ge -12$	$I \le -1, I \ge -510$	$I \le -1, I \ge -2147483648$
47	XMLGrammarPoolImpl	$I \le -1, I \ge -13$	$I \le -2147483137$ $I \ge -2147483648$	$I \le -1,$ $I \ge -2147483648$
48	XML11Char	$I \le -1, I \ge -16$	$I \le -1, I \ge -512$	$I \le -1, I \ge -2147483648$
49	AttributeList	$I \ge 2147483635$ $I \le 2147483647$	$I \ge 2147483590$ $I \le 2147483647$	$I > 698000000$ $I \le 2147483647$
50	ClassLoaderResolver	$I \ge 2,$ $I \le 18$	$I \ge 500, I \le -2$ $I \ge 2, I \le 505$	$I \le -2, I > -2147483648$ $I \ge 2, I \le 2147483647$

Glossary

Branch:	Conditional transfer of control from one statement to another in the code.
Correctness:	Ability of software to perform according to the given specifications.
Dead code	Unreachable code in program that cannot be executed.
Defect:	Generic term referring to fault or failure.
Detection:	Difference between observed and expected behaviour of program.
Effectiveness:	Number of defects discovered in the program by a testing technique.
Efficiency:	Number of defects discovered per unit time by a testing technique.
Error:	Mistake or omission in the software.
Failure:	Malfunction of a software.
Fault:	Any flaw in the software resulting in lack of capability or failure.
Instrumentation:	Insertion of additional code in the program for analysis.
Invariant:	A condition which must hold true during program execution.
Isolation:	Identification of the basic cause of failure in software.
Path:	A sequence of executable statements from entry to exit point in software.
Postcondition:	A condition which must be true after execution.
Precondition:	A condition which must be true before execution.
Robustness:	The degree to which a system can function correctly with invalid inputs.
Test case:	An artefact that delineates the input, action and expected output.
Test coverage:	Number of instructions exercised divided by total number of instructions expressed in percentage.

Test execution:	The process of executing test case.
Test oracle:	A mechanism used to determine whether a test has passed or failed.
Test Plan:	A document which defines the goal, scope, method, resources and time schedule of testing.
Test specification:	The requirements which should be satisfied by test cases.
Test strategy:	The method which defines the procedure of testing of a program.
Test suite:	A set of one or more test cases.
Validation:	Assessment of software to ensure satisfaction of customer requirements.
Verification:	Checking of software for verification of working properly.

Appendix

Error-seeded code to evaluate ADFD and ADFD+

Program 1 Point domain with One argument

```
/**
 * Point Fault Domain example for one argument
 * @author (Mian and Manuel)
 */
public class PointDomainOneArgument{

    public static void pointErrors (int x){
        if (x == -66 )
            x = 5/0;

        if (x == -2 )
            x = 5/0;

        if (x == 51 )
            x = 5/0;

        if (x == 23 )
            x = 5/0;
    }
}
```

Program 2 Point domain with two argument

```
/**
 * Point Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class PointDomainTwoArgument{

    public static void pointErrors (int x, int y){
        int z = x/y;
    }

}
```

Program 3 Block domain with one argument

```
/**
 * Block Fault Domain example for one arguments
 * @author (Mian and Manuel)
 */

public class BlockDomainOneArgument{

public static void blockErrors (int x){

    if((x > -2) && (x < 2))
        x = 5/0;

    if((x > -30) && (x < -25))
        x = 5/0;

    if((x > 50) && (x < 55))
        x = 5/0;

   }
}
```

Program 4 Block domain with two argument

```
/**
 * Block Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class BlockDomainTwoArgument{

    public static void blockErrors (int x, int y){

        if(((x > 0)&&(x < 20)) || ((y > 0) && (y < 20))){
        x = 5/0;
        }

    }

}
```

Program 5 Strip domain with One argument

```
/**
 * Strip Fault Domain example for one argument
 * @author (Mian and Manuel)
 */

public class StripDomainOneArgument{

    public static void stripErrors (int x){

        if((x > -5) && (x < 35))
            x = 5/0;
     }
}
```

Program 6 Strip domain with two argument

```
/**
 * Strip Fault Domain example for two arguments
 * @author (Mian and Manuel)
 */
public class StripDomainTwoArgument{

    public static void stripErrors (int x, int y){

        if(((x > 0)&&(x < 40)) || ((y > 0) && (y < 40))){
        x = 5/0;
        }

    }

}
```

Program generated by ADFD on finding fault in SUT

```
/**
 * Dynamically generated code by ADFD strategy
 * after a fault is found in the SUT.
 * @author (Mian and Manuel)
 */
import java.io.*;
import java.util.*;

public class C0
{
    public static ArrayList<Integer> pass = new ArrayList<Integer>();
    public static ArrayList<Integer> fail = new ArrayList<Integer>();
    public static boolean startedByFailing = false;
    public static boolean isCurrentlyFailing = false;
    public static int start = -80;
    public static int stop = 80;

    public static void main(String []argv){
        checkStartAndStopValue(start);
        for (int i=start+1;i<stop;i++){
            try{
                PointDomainOneArgument.pointErrors(i);
                if (isCurrentlyFailing)
                {
                    fail.add(i-1);
                    fail.add(0);
                    pass.add(i);
                    pass.add(0);
                    isCurrentlyFailing=false;
                }
            }
            catch(Throwable t) {
                if (!isCurrentlyFailing)
                {
                    pass.add(i-1);
                    pass.add(0);
                    fail.add(i);
                    fail.add(0);
                    isCurrentlyFailing = true;
                }
            }
        }
        checkStartAndStopValue(stop);
        printRangeFail();
        printRangePass();
    }

    public static void printRangeFail() {
        try {
            File fw = new File("Fail.txt");
            if (fw.exists() == false) {
                fw.createNewFile();
            }
            PrintWriter pw = new PrintWriter(new FileWriter (fw, true));
            for (Integer i1 : fail) {
                pw.append(i1+"\n");
            }
            pw.close();
        }
        catch(Exception e) {
            System.err.println(" Error : e.getMessage() ");
        }
    }
    public static void printRangePass() {
        try {
            File fw1 = new File("Pass.txt");
            if (fw1.exists() == false) {
                fw1.createNewFile();
            }
            PrintWriter pw1 = new PrintWriter(new FileWriter (fw1, true));
            for (Integer i2 : pass) {
                pw1.append(i2+"\n");
            }
            pw1.close();
        }
        catch(Exception e) {
            System.err.println(" Error : e.getMessage() ");
        }
    }
    public static void checkStartAndStopValue(int i) {
        try {
            PointDomainOneArgument.pointErrors(i);
            pass.add(i);
            pass.add(0);
        }
        catch (Throwable t) {
            startedByFailing = true;
            isCurrentlyFailing = true;
            fail.add(i);
            fail.add(0);
        }
    }
}
```

References

[1] FT Chan, Tsong Yueh Chen, IK Mak, and Yuen-Tak Yu. Proportional sampling strategy: guidelines for software testing practitioners. *Information and Software Technology*, 38(12):775–782, 1996.

[2] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[3] Carlos Pacheco and Michael D Ernst. *Eclat: Automatic generation and classification of test inputs*. Springer, 2005.

[4] Andreas Leitner and Ilinca Ciupa. Reconciling manual and automated testing: the autotest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, pages 3–6. Technology, 2007.

[5] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 22–31. IEEE, 2001.

[6] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *Software Engineering, IEEE Transactions on*, 27(2):99–123, 2001.

[7] Maurice Wilkes. *Memoirs of a computer pioneer*. Massachusetts Institute of Technology, 1985.

[8] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, pages 40–52. ACM, 2004.

[9] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, 7007(011), 2002.

[10] Ron Patton. *Software testing*, volume 2. Sams Indianapolis, 2001.

[11] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[12] William E. Howden. A functional approach to program testing and analysis. *Software Engineering, IEEE Transactions on*, (10):997–1005, 1986.

[13] Thomas J McCabe. *Structured testing*, volume 500. IEEE Computer Society Press, 1983.

[14] Joan C Miller and Clifford J Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63, 1963.

[15] Bogdan Korel. Automated software test data generation. *Software Engineering, IEEE Transactions on*, 16(8):870–879, 1990.

[16] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 84–94. ACM, 2007.

[17] T. Y. Chen. Adaptive random testing. *Eighth International Conference on Qualify Software*, 0:443, 2008.

[18] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[20] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[21] Lee J. White. Software testing and verification. *Advances in Computers*, 26(1):335–390, 1987.

[22] Simson Garfinkel. History's worst software bugs. *Wired News, Nov*, 2005.

[23] NY. American National Standards Institute. New York, Institute of Electrical, and Electronics Engineers. *Software Engineering Standards: ANSI/IEEE Std 729-1983, Glossary of Software Engineering Terminology*. Inst. of Electrical and Electronics Engineers, 1984.

[24] Robert T Futrell, Linda I Shafer, and Donald F Shafer. *Quality software project management*. Prentice Hall PTR, 2001.

[25] Ashfaque Ahmed. *Software testing as a service*. CRC Press, 2010.

[26] Luciano Baresi and Michal Young. Test oracles. *Techn. Report CISTR-01*, 2:9, 2001.

[27] Marie-Claude Gaudel. Software testing based on formal specification. In *Testing Techniques in Software Engineering*, pages 215–242. Springer, 2010.

[28] Elaine J Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[29] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

[30] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[31] Julie Cohen, Daniel Plakosh, and Kristi L Keeler. Robustness testing of software-intensive systems: Explanation and guide. 2005.

[32] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[33] Lori A Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *Software Engineering, IEEE Transactions on*, 15(11):1318–1332, 1989.

[34] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

[35] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[36] Jean Arlat. *Validation de la sûreté de fonctionnement par injection de fautes, méthode- mise en oeuvre- application*. PhD thesis, 1990.

[37] Jeffrey M Voas and Gary McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[38] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[39] Frank Armour and Granville Miller. *Advanced use case modeling: software systems*. Pearson Education, 2000.

[40] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[41] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703–711, 1991.

[42] Simeon Ntafos. On random and partition testing. In *ACM SIGSOFT Software Engineering Notes*, volume 23, pages 42–48. ACM, 1998.

[43] Muthu Ramachandran. Testing software components using boundary value analysis. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 94–98. IEEE, 2003.

[44] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std*, 610121990:121990, 1990.

[45] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64–73. IEEE, 1997.

[46] Michael R Donat. Automating formal specification-based testing. In *TAPSOFT'97: Theory and Practice of Software Development*, pages 833–847. Springer, 1997.

[47] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE'07*, pages 85–103. IEEE, 2007.

[48] Ian Sommerville. *Software Engineering: Pearson New International Edition*. Pearson Education Limited, 2013.

[49] Richard E Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[50] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 157–166. IEEE, 2008.

[51] G Venolia, Robert DeLine, and Thomas LaToza. Software development at microsoft observed. *Microsoft Research, TR*, 2005.

[52] Jan Tretmans and Axel Belinfante. Automatic testing with formal methods. 2000.

[53] ECMA ECMA. 367: Eiffel analysis, design and programming language. *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr*, 2005.

[54] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[55] GT Leavens, E Poll, C Clifton, Y Cheon, C Ruby, D Cok, and J Kiniry. Jml reference manual (draft), 2005.

[56] Reto Kramer. icontract-the java tm design by contract tm tool. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 295–307. IEEE, 1998.

[57] Mark Richters and Martin Gogolla. On formalizing the uml object constraint language ocl. In *Conceptual Modeling–ER98*, pages 449–464. Springer, 1998.

[58] Zhenyu Huang. Automated solutions: Improving the efficiency of software testing, 2003.

[59] CV Ramamoorthy and Sill-bun F Ho. Testing large software with automated software evaluation systems. In *ACM SIGPLAN Notices*, volume 10, pages 382–394. ACM, 1975.

[60] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[61] Insang Chung and James M Bieman. Automated test data generation using a relational approach.

[62] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(1):63–86, 1996.

[63] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software Testing Verification and Reliability*, 9(4):263–282, 1999.

[64] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, 2(3):223–226, 1976.

[65] Phil McMinn. Search-based software testing: Past, present and future. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 153–163. IEEE, 2011.

[66] Bryan F Jones, H-H Sthamer, and David E Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[67] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[68] David L. Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM systems journal*, 22(3):229–245, 1983.

[69] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Normalized restricted random testing. In *Reliable Software TechnologiesAda-Europe 2003*, pages 368–381. Springer, 2003.

[70] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.

[71] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 71–80. IEEE, 2008.

[72] Carlos Pacheco. *Directed random testing*. PhD thesis, Massachusetts Institute of Technology, 2009.

[73] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332. ACM, 2007.

[74] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420. ACM, 2007.

[75] Joe W Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183. IEEE Press, 1981.

[76] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[77] Justin E Forrester and Barton P Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows System Symposium*, pages 59–68, 2000.

[78] Barton P Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *Proceedings of the 1st international workshop on Random testing*, pages 46–54. ACM, 2006.

[79] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239. IEEE, 1998.

[80] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631. IEEE, 2007.

[81] Tsong Yueh Chen, F-C Kuo, Robert G Merkel, and Sebastian P Ng. Mirror adaptive random testing. *Information and Software Technology*, 46(15):1001–1010, 2004.

[82] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Restricted random testing. In *Software QualityECSQ 2002*, pages 321–330. Springer, 2006.

[83] Tsong Yueh Chen and Robert Merkel. Quasi-random testing. *Reliability, IEEE Transactions on*, 56(3):562–568, 2007.

[84] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.

[85] Carlos Pacheco, Shuvendu K Lahiri, and Thomas Ball. Finding errors in. net with feedback-directed random testing. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 87–96. ACM, 2008.

[86] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the 1st international workshop on Random testing*, pages 55–63. ACM, 2006.

[87] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[88] Bertrand Meyer, Jean-Marc Nerson, and Masanobu Matsuo. Eiffel: object-oriented design for software engineering. In *ESEC'87*, pages 221–229. Springer, 1987.

[89] Patrick Chan, Rosanna Lee, and Douglas Kramer. *The Java Class Libraries, Volume 1: Supplement for the Java 2 Platform, Standard Edition, V 1.2*, volume 1. Addison-Wesley Professional, 1999.

[90] Catherine Oriat. Jartege: a tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality*, pages 242–256. Springer, 2005.

[91] Willem Visser, Corina S Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[92] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[93] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.

[94] Ilinca Ciupa, Andreas Leitner, et al. Automatic testing of object-oriented software. In *In Proceedings of SOFSEM 2007 (Current Trends in Theory and Practice of Computer Science*. Citeseer, 2007.

[95] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. *ACM SIGSOFT Software Engineering Notes*, 26(5):62–73, 2001.

[96] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The alloy constraint analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733. IEEE, 2000.

[97] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 123–133. ACM, 2002.

[98] Juei Chang and Debra J Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Software EngineeringESEC/FSE99*, pages 285–302. Springer, 1999.

[99] Sarfraz Khurshid and Darko Marinov. Checking Java implementation of a naming architecture using TestEra. *Electronic Notes in Theoretical Computer Science*, 55(3):322–342, 2001.

[100] Manuel Oriol and Sotirios Tassis. Testing .NET code with YETI. In *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, pages 264–265. IEEE, 2010.

[101] Manuel Oriol and Faheem Ullah. YETI on the cloud. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 434–437. IEEE, 2010.

[102] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[103] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[104] Manuel 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*, pages 201–210. IEEE, 2012.

[105] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 75–84. IEEE, 2007.

[106] Andreas Leitner, Alexander Pretschner, Stefan Mori, Bertrand Meyer, and Manuel Oriol. On the effectiveness of test extraction without overhead. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 416–425. IEEE, 2009.

[107] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[108] Stéphane Ducasse, Manuel Oriol, and Alexandre Bergel. Challenges to support automated random testing for dynamically typed languages. In *Proceedings of the International Workshop on Smalltalk Technologies*, page 9. ACM, 2011.

[109] Tsong Yueh Chen, Fei-Ching Kuo, and Robert Merkel. On the statistical properties of the f-measure. In *Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on*, pages 146–153. IEEE, 2004.

[110] Tsong Yueh Chen and Yuen Tak Yu. On the expected number of failures detected by subdomain testing and random testing. *Software Engineering, IEEE Transactions on*, 22(2):109–119, 1996.

[111] Huai Liu, Fei-Ching Kuo, and Tsong Yueh Chen. Comparison of adaptive random testing and random testing under various testing and debugging scenarios. *Software: Practice and Experience*, 42(8):1055–1074, 2012.

[112] Ilinca Ciupa, Alexander Pretschner, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. On the predictability of random tests for object-oriented software. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 72–81. IEEE, 2008.

[113] Kwok Ping Chan, Tsong Yueh Chen, and Dave Towey. Good random testing. In *Reliable Software Technologies-Ada-Europe 2004*, pages 200–212. Springer, 2004.

[114] A Jefferson Offutt and J Huffman Hayes. A semantic model of program faults. In *ACM SIGSOFT Software Engineering Notes*, volume 21, pages 195–200. ACM, 1996.

[115] Tsong Yueh Chen, Robert G Merkel, G Eddy, and PK Wong. Adaptive random testing through dynamic partitioning. In *QSIC*, pages 79–86, 2004.

[116] Shin Yoo and Mark Harman. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability*, 22(3):171–201, 2012.

[117] Joe W Duran and Simeon C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, (4):438–444, 1984.

[118] Walter J Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *Software Engineering, IEEE Transactions on*, 25(5):661–674, 1999.

[119] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on*, 38(2):258–277, 2012.

[120] Ewan Tempero, Steve Counsell, and James Noble. An empirical study of overriding in open source java. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*, pages 3–12. Australian Computer Society, Inc., 2010.

[121] Ewan Tempero. An empirical study of unused design decisions in open source Java software. In *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*, pages 33–40. IEEE, 2008.

[122] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[123] Johannes Mayer. Lattice-based adaptive random testing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 333–336. ACM, 2005.

[124] D Gilbert. The JFreeChart class library version 1.0. 9: Developers guide. *Refinery Limited, Hertfordshire*, 48, 2008.

[125] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 465–475. IEEE, 2003.

[126] Hiraral Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[127] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[128] Per Runeson, Carina Andersson, Thomas Thelin, Anneliese Andrews, and Tomas Berling. What do we know about defect detection methods?[software testing]. *Software, IEEE*, 23(3):82–90, 2006.

[129] Mian A Ahmad and Manuel Oriol. Automated discovery of failure domain. *Lecture Notes on Software Engineering*, 03(1):289–294, 2013.

[130] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381. Springer, 2005.

[131] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[132] Lee J White and Edward I Cohen. A domain strategy for computer program testing. *Software Engineering, IEEE Transactions on*, (3):247–257, 1980.

[133] George B Finelli. NASA software failure characterization experiments. *Reliability Engineering & System Safety*, 32(1):155–169, 1991.

[134] Peter G Bishop. 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*, pages 98–107. IEEE, 1993.

[135] Mian A Ahmad and Manuel Oriol. Automated discovery of failure domain. *Lecture Notes on Software Engineering*, 03(1):289–294, 2013.