An Empirical Comparison of Fault-Detection Effectiveness and Cost of Exhaustive and Pair-Wise Testing

Shashvat A Thakor
Department of Computer Science
University of Maryland
College Park, MD 20742
shash@cs.umd.edu

August 11, 2008

Abstract

Graphical User Interface (GUI) is an integral part of the contemporary software. Testing GUI poses different challenges compared to those applications not involving GUI. The main reason is that GUI involves interactive input, where as the output produced is often graphical. One of the approaches in testing GUI is functional testing. The goal is to find the difference between the actual behavior and desired behavior mentioned in specification. A standard method of generating specification based functional tests is to partition the input space, and then select test data from each partition. One such systematic approach, the category partition method, is however exhaustive in nature. In order to reduce the number of test cases generated by the category partition method, pair-wise sampling can be used. Pair-wise sampling is based on the observation that most of the errors occur because of interaction of at the most two factors. In this paper we empirically compare fault-detection ability and the cost of applying category partition method (exhaustive) and pair-wise sampling. Our results, based on the subject applications, show that pair-wise sampling can be used at much lower cost and without losing much of the fault-finding ability.

Keywords:
Software testing, GUI testing, functional testing, category partition method, pair-wise sampling, empirical studies

1 Introduction

In recent times, most of the software applications are developed and maintained by several programmers, who are not only geographically distributed, but work on different parts of the application code. This scenario inadvertently introduces bugs, as the software is designed and constructed. As Glen Myers has stated in his book, “Testing is the process of executing a program with the intent of finding errors” [10].

Let us assume that we make sure there are five faults in a thousand lines of code (LOC), i.e. defect rate is 5 faults/ 1000 lines. Then also, a program with a million LOC will have approximately 5000 faults. Going by this analogy, Windows XP, having 45 million LOC, has 225,000 faults.

Software is being used everywhere, ranging from digital flight control systems in space shuttles to sonar system submarines, from radiation-therapy machine in medicine to Apple iPods. Hence, Software Quality Assurance (SQA) has become an integral and critical phase of any software development project [4].

Software testing is an important part of SQA, which involves evaluating the results generated under controlled conditions of system operation. The controlled conditions should include both normal and abnormal conditions. Software testing is governed towards ‘detection’ of anomalous software behavior. It is useful in improving correctness, reliability, usability, robustness and performance of the software.

Most of the contemporary software consists of Graphical User Interface (GUI). Testing issues in GUI are different than those of other software development. Interactive input and graphical output makes GUI testing difficult [16].

There are many approaches towards software testing, one of them being unit testing. A unit is considered to be the smallest testable part of the application. The goal is to verify whether the individual units of the entire source code are working properly. Another is functional testing, in which the goal is to find the discrepancies between the actual behavior of the implemented system’s functions and the desired behavior as described in the system’s functional specification.

A standard approach of generating specification based functional tests is first to partition the input domain of a function being tested, and then to select test data from each class of the partition. Category-partition method is one such systematic, specification-based method that uses partitioning to generate functional tests for complex software systems [2]. However, it is resource-intensive in nature. There is always the problem of combinatorial explosion when dealing with category partition method. A solution is to change the sampling strategy.

One general observation is that most of the faults are caused by interactions of at most two factors. An effective test case generation testing based on this observation is pair-wise (all-pairs) sampling. Pair-wise-generated test suites cover all combinations of two and therefore are much smaller than exhaustive ones yet still very effective in finding defects.

The purpose of this paper is to empirically compare fault-detection effectiveness and cost of testing based on category partition method (exhaustive) and pair-wise sampling. In Section 2, we discuss background and related work. Section 3 describes the design and the implementation of the empirical study. In Section 4, we interpret the results. Conclusion is presented in Section 5.

2 Background and Related Work

In this section we review software testing, testing GUI, category partition method, JUnit and pair-wise sampling, which form the basis of our work.

2.1 Software Testing

Software testing has become an ever growing field with the advent of various types of software used in modern times. Whether it is object-oriented, component based, concurrent, distributed, GUI or web based, each of them requires testing in order to uncover and correct the defects. Software testing can be exploited to improve usability, security, correctness and performance of software. It is a fact that testing takes more than 50% of the total cost of software development. However, the question “does a program P, obey specification S” is undecidable. It is theoretically impossible to develop a completely accurate technique. Two types of verification are associated with software testing [12]:

(1) Execution based verification
(2) Non-execution based verification.

Execution based verification deals with generating and executing test cases on the software. Testing to specification is black-box testing and testing to code is white-box testing. The key difference is only in the test case generation. In both the cases, specifications are used for verification of correctness.

Non-execution based verification is basically reviews by a team of experts. Code reading, walkthroughs (manual simulation by team leader) and inspections (narration by the developer) are also used.

2.2 Graphical User Interface (GUI)

Definition: A Graphical User Interface (GUI) is a hierarchical, graphical front-end to a software system that accepts as input user-generated and system-generated events, from a fixed set of events and produces deterministic graphical output. A GUI contains graphical objects; each object has a fixed set of properties. At any time during the execution of the GUI, these properties have discrete values, the set of which constitutes the state of the GUI [8].

Above definition should be extended to include other types of GUI, such as web-user interfaces having synchronization constraints among objects, non-deterministic GUIs etc.

There are several approaches while testing GUI software. The most popular approach is to perform no GUI testing at all [5]. But, this approach is not governed towards improving software quality. Another approach is to "bypass" the GUI and test the methods of the underlying business logic as if they were invoked by the GUI. It follows the idea of having "light" GUI software and incorporating important decisions in the business logic [6]. However, this approach requires major architectural changes and does not test the end-user software. Third approach is to do limited testing using manual GUI testing tools [1], [14]. Some examples are JFCUnit (an extension of JUnit) [20], capture/replay tools [3], which provide very little automation. DART [9] framework automates the GUI smoke testing by using model based testing technique.

2.3 JUnit

JUnit is a unit testing framework for Java programming language [17]. It is a simple, open source framework to write and run repeatable tests. Prominent features of JUnit are:

(1) Assertions to test the expected results

(2) Text fixtures in order to share common test data

(3) Test runners to run the tests

Figure 1 shows a sample JUnit test case.

```
public class HelloWord extends TestCase{
        public void testMultiplication(){
                //testing if 5*2 = 10
                assertEquals(10, 5*2);
        }
}
```

Figure 1: An example of a JUnit test case

2.4 Category Partition Method

```
1     Decompose the functional specification into functional units
      a. Characteristics of functional units
              • They can be tested independently
              • Examples
                    • A top level user command
                    • Function
2     Examine each functional unit
3         Identify parameters
              • Explicit input to functional unit
          and Environmental conditions
              • Characteristic of the system's state
4     Test Cases
      a. Specific values of parameters and environmental conditions
      b. Maximize the chances of finding errors
5     For each parameter and environmental condition
6         Find categories
              • Major property or characteristics
7         For each Category
8                 Find choices
9     Develop "formal test specification for each functional unit
      a. List of Categories
      b. Lists of choices within each category
10   Find Constraints
11   Produce a set of "test-frames"
```

Figure 2: Steps of Category Partition method

The Category Partition test-cases are generated using Cartesian products of the possible input values for each of the parameters. In Cartesian product every unit of a group is paired with every unit of every other group. Thus, all combinations of the inputs across all groups are obtained.

The category partition method results in the generation of large number of test cases. For a method having four parameters, and four different values for each of the parameters, 256 *(4^4)* number of test cases are generated.

According to [11], the main characteristics of category-partition method are:

(1) Test specification is a succinct and uniform representation of the test information of a function.

(2) Test specification is easily changeable depending upon requirement of more test cases and changes or errors in the original specification.

(3) Logical control over the test volume is provided.

(4) Generator tool provides enough tests for each function in an automated fashion taking into consideration improbable environment.

(5) Coverage and error detection are given importance.

Figure 2 describes the steps of the category partition method.

2.5 Pair-wise Sampling

The number of test cases is reduced by making use of a pair-wise sampling technique. It is a method of intelligently choosing a small number of combinations of values from a potentially astronomically large number of test cases, which satisfies the pair wise criterion, converting into a manageable set that still makes for an effective test suite in most situations [13].

Satisfying the pair-wise criterion means that for any two parameters p_1 and p_2 and any valid values v_1 for p_1 and v_2 for p_2, there is a test in which p_1 has the value v_1 and p_2 has the value v_2.

Essentially, pair-wise sampling is an effective test case generation technique that is based on the observation that most faults are caused by interactions of at the most two factors. Pair wise-generated test suites cover all combinations of the two factors and therefore are much smaller than exhaustive ones yet are still very effective in finding defects [18]. The necessary condition is that for each pair of input parameters, every combination of valid values of these two parameters should be covered.

Two growth terms associated with pair-wise sampling are:

(1) Horizontal growth: Let T be a pair-wise test set for parameters p_1, p_2, …., p_{n-1}. Horizontal growth of T for parameter p_i is to extend each test in T by adding the value of p_i. (figure 3)

(2) Vertical growth: After applying horizontal growth, let T be a test set for p_1, p_2, …., p_i. Let π be the set of tests not covered by T. The vertical growth of T according to π is to construct new tests for pairs in π and add them to T. (figure 4)

```
    Input
    Test set: $T$ (a list having elements in arbitrary order)
    Parameter: $p_i$
    Procedure
1   Assume that the domain of $p_i$ contains valued $v_1$, $v_2$... and $v_q$;
2   $\pi$ = {pairs between values $p_i$ and valued $p_1$, $p_2$... and $p_{i-1}$};
3   if ($|T| < q$) {
4       for $1 \leq j \leq |T|$, extend the $j^{th}$ test in $T$ by adding value $v_j$ and remove
        from $\pi$ pairs covered by the extended test;
5   } else {
6       for $1 \leq j \leq q$, extend the $j^{th}$ test in $T$ by adding value $v_j$ and remove
        from $\pi$ pairs covered by the extended test;
7       for $q < j \leq |T|$, extend the $j^{th}$ test in T by adding one value of $p_i$ such
        that the resulting test covers the most number of pairs in $\pi$, and
        remove from $\pi$ pairs covered by the extended test;
    }
```

Figure 3: Algorithm ***IPO_H (T, P$_i$)*** for horizontal growth

However, there exists a problem with pair wise testing. If the domain of the input parameter is large, we get a very large number of tests. Suppose all the parameters have d values then we have to consider at least d^2. Thus, if each variable has 100 values, we have to consider 10,000 tests for pair wise testing, which is indeed a large number. One solution of this test explosion problem can be to consider partitions of the input domains, and selecting a representative value for each of them [13].

```
    Input
    Test set: T (a list having elements in arbitrary order)
    Set of tests not covered by T:  $\pi$
    Procedure
1   Let T' = { };
2   for each pair in $\pi$
3       assume that the pair contains values w of $p_k$, 1<k<i, and value u of $p_i$
4       if (T' contains a test with "-" as the value of $p_k$ and u as the value of  i)
5           notify this test by replacing the "-" with w;
6       } else {
7           add a new test to T' that has w as the value of $p_k$, u as the value of $p_i$,
            and "-" as the value of every other parameter;
        };
8    $T = T \cup T'$
```

Figure 4: Algorithm ***IPO_V (T, π)*** for vertical growth

3 Empirical Study

The goal of the study is to determine whether pair-wise sampling method, with its reduced number of test cases, is as effective as exhaustive method, with its large number of test cases generated using category partition method. The following steps have been performed:

(1) For each subject application, select relatively complex methods (say having 5 parameters); create JUnit test cases for these methods using the category-partition method. Reduce the number of test cases using pair-wise sampling.

(2) Examine each JUnit test cases and the method source code. Obtain a set of source-code changes that will cause each test case to fail [9].

(3) Use fault-seeding technique to artificially seed faults in the applications.

(4) Execute pair-wise and exhaustive test cases on the subject application. Compare the number of faults found and the cost of applying each method.

3.1 Subject Applications

We took into consideration several requirements while choosing the subject application. We wanted to have access to the source code, CVS development history, and bug reports.

We also wanted applications that were "GUI-intensive," i.e., ones without complex back-end code. The GUIs of such applications are typically static, i.e., not generated dynamically from back-end data. Finally, we wanted non-trivial applications, consisting of several windows and widgets.

The subject applications for our experiment are part of an open-source office suite developed at the Department of Computer Science of the University of Maryland by undergraduate students of the senior Software Engineering course [7, 9]. It is called TerpOffice and includes the applications TerpCalc, TerpWord, TerpPresent, TerpPaint and TerpSpreadSheet. TerpCalc is a scientific calculator with graphing capability; TerpWord is a word-processor with drawing capability; TerpPresent is used to prepare slides and present them online; TerpPaint is an imaging tool; TerpSpreadSheet is a compact spreadsheet program. They have been implemented using Java. Table 1 summarizes the characteristics (lines of code, number of classes, windows, widgets, methods and branches) of these applications. The widget counts shown include only those widgets on which events can be performed. Most of the code written for the implementation of each application is for the GUI. None of the applications have complex underlying "business logic", which makes seeding GUI faults (discussed later) easier because almost the entire code is for GUI and there is no need to distinguish it from the business-logic-code.

Subject Application	LOC	Classes	Windows	Widgets	Methods	Branches
TerpCalc	**9916**	**141**	**4**	**82**	**446**	**1306**
TerpPaint	**18376**	**219**	**10**	**200**	**644**	**1277**
TerpPresent	**44591**	**230**	**12**	**294**	**230**	**3099**
TerpSpreadSheet	**12791**	**125**	**9**	**145**	**579**	**1521**
TerpWord	**4893**	**104**	**11**	**112**	**236**	**452**
TOTAL	**90567**	**819**	**46**	**833**	**3549**	**7655**

Table 1: Subject Applications

3.2 Generating test cases

The Category Partition test-cases were generated using Cartesian products of the possible input values for each of the parameters. The Pair wise test cases with combinatorial approach are generated using the ‘AllPairs’ tool [19]. ‘AllPairs’ is a test design tool tailored for Windows but portable to a wide variety of platforms with some minor tweaks to the script file. It automates the "all pairs" test design technique.

3.3 Fault Seeding

Fault seeding is a well known technique to measure the ability of different methods to find faults. In fault seeding, subject applications are artificially introduced to several types of fault classes. The artificially seeded faults should be as close as possible to naturally occurring faults by the software developers. There should be adequate number of test cases that cover the faults. An adequate number of instances of each fault type should be seeded. The following fault classes were chosen for this study:

(1) Modify relational operator (>, <, >=, <=, ==, !=)

(2) Invert the condition statement

(3) Modify arithmetic operator (+, -, *, /, =, ++, -, +=, -=,*=, /=)

(4) Modify logical operator (&&, jj)

(5) Set/return different Boolean value (true, false)

(6) Invoke different (syntactically similar) method

(7) Set/return different attributes

(8) Modify bit operator (&, j, ^, &=, !=, ^=)

(9) Set/return different variable name

(10) Set/return different integer value

(11) Exchange two parameters in a method and

(12) Set/return different string value.

The parts, in which the faults should be induced, were examined manually. Let f_j and F be the number of opportunities to seed fault of class j and the sum of all the opportunities of all the fault classes respectively. Since a total of 20 faults were seeded in each application, for each fault class j, we had introduced $((\mathbf{f_j}/ \boldsymbol{F}) * \mathbf{20})$ instances of faults, because of which there were more instances of one fault class over the others. e.g. there were a lot of relational operators in the subject applications, hence a total of 47 instances of fault type 1 (modify relational operator) were seeded. Figure 5 shows the distribution of the 100 faults that were seeded in the applications. The x-axis shows the class (type) of the fault and the height of the columns shows the number of faults seeded.

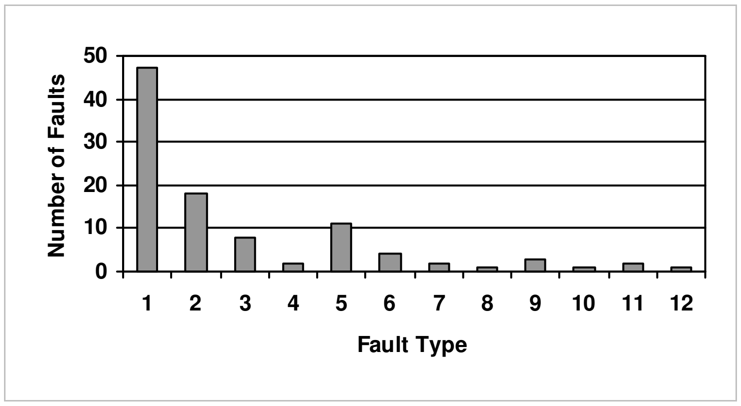

Figure 5: Classes of faults seeded

3.4 Implementation

We first used category partition method, in order to generate exhaustive test cases. For example, one of the classes of TerpCalc was TCGraph having the constructor TCGraph(). It sets the settings of the graph such as axis and checks for validity. TCGraph class continues the implementation of the graph window by implementing its actual graphing capability, in both setting appropriate values and painting the graph within its corresponding frame. This class also implements the saving feature of the graph, along with its copy mechanism to the clipboard.

We used category partition method on the constructor as shown in Figure 6. Using Cartesian product of input values, 4102 test cases were created, taking into consideration the environmental conditions. This set was reduced using ‘Allpairs’ tool. It took a text file of tab separated input values, and used pair-wise sampling, generating 34 test cases.

In order to seed faults, a comment ***/*FAULT## FAILURE INDUCING CODE */*** at line N was inserted in the method. The idea was to replace string of line N with “***FAILURE INDUCING CODE***”, which would cause the JUnit test case to fail. If the change required changes to multiple lines, then we replaced ***##*** with an integer; used the same value of the integer for all lines that were related to one failure. A Perl script was written to perform the string replacement automatically to avoid fault interaction.

TCGraph(expression, xmin, xmax, ymin, ymax, numGraphpts)

Parameter: expression
1. Type: String
2. Characteristics: length, blanks
3. Possible values: null, expression with one or more blanks, variable length expression
4. Error: null

Parameter: xmin
1. Type: Integer
2. Possible values: Zero, Positive, Negative
3. Error: Any other type of value like float, char, string

Parameter: xmax
1. Type: Integer
2. Possible values: Zero, Positive, Negative
3. Error: Any other type of value like float, char, string

Parameter: ymin
1. Type: Integer
2. Possible values: Zero, Positive, Negative
3. Error: Any other type of value like float, char, string

Parameter: ymax
1. Type: Integer
2. Possible values: Zero, Positive, Negative
3. Error: Any other type of value like float, char, string

Parameter: numGraphpts
1. Type: Integer
2. Possible values: Zero, Positive, Negative
3. Error: Any other type of value like float, char, string

Environment Conditions:
1. xmin < xmax
2. xmin >= xmax then xmin= -10 and xmax=10
3. ymin < ymax
4. ymin >= ymax & ymax != 0 then ymin = ymax = 0
5. numGraphpts < 25 implies numGraphpts=25

Constraints:
1. None

Input values to test case generation:

Expression	xmin	xmax	ymin	ymax	numGraphPts
$y= x^2+x+2$	-8	-8	0	0	-100
$y=x$	-4	-4	10	10	0
$y=x^2$	0	40	40	40	12
$y=x^3+1$	10	60	60	60	100

Figure 6: Category partition on TCGraph() constructor

A batch file was written in order to first run the Perl script to seed the faults in the application, then execute the test cases with the fault-seeded applications and finally to remove the seeded faults from the application to convert into original form.

3.5 Threats to Validity

Test validity can be considered as the degree of correlation between the test and a criterion. We considered two types of threats – external and internal.

Threats to external validity [15] are the conditions which restrict the ability to generalize the results of the experiments to industrial practice. Several such threats are identified in the study. Our subject applications and types of faults that we seed are the biggest threats to external validity. First, we have used five applications, which are part of TerpOffice suite developed by students, as our subject applications. These GUI applications do not represent the various possible GUIs that are used today. As mentioned previously, most of the code appearing in the application is written for GUI, i.e. applications are more GUI-intensive and have less business logic. So the results will be different from applications having complex business logic and simple GUI. Second, all the applications were developed in Java; hence the results may vary for non-Java applications. Third, the faults that were seeded represent a small subset of faults that are present in applications developed by students.

Threats to internal validity are conditions which affect the dependent variables of the experiment without the knowledge of the researcher. The biggest threats to internal validity are related to the way we seed the faults. We made an effort to introduce faults which were as close as possible to naturally occurring faults. Those faults which are not obvious for the GUI will not be detectable.

4 Results and Discussion

One of the original goals of the study was to examine the classes (types) of faults detected by category partition method and pair-wise sampling. Figure 7 summarizes the results. The x-axis shows the type/class of fault and the total height of the columns shows the number of faults seeded.

Since the category partition method is exhaustive, and takes into consideration every possible input values, it is able to find all the *reachable* faults of all the types. Hence the total height of the column also describes the number of faults detected by the category partition method for each class of fault. The part of the column, having wide upward diagonal pattern, shows the faults detected by test cases generated using pair-wise sampling. This result showed that the pair-wise sampling was able to detect all types of faults that were seeded; except for the fault type 11 (exchange two parameters in a method).

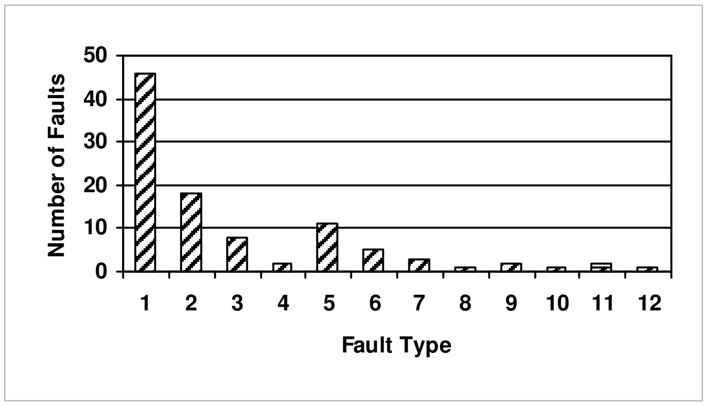

Figure 7: Classes of fault detected by Category partition method and pair-wise sampling

To determine why fault-type 11 was missed, we examined the input values and the seeded fault. In order to induce the fault of type 11, one method was modified as the following:

method (operator, valueRangeOne, valueRangeTwo)

$\downarrow$

method (operator, valueRangeOne, valueRangeOne)

Figure 8: Seeding fault type 11 - exchange two parameters in a method

This fault was not detected due to the following scenario:

(1) Fault was replacing operation (argumentOne ^ argumentTwo) by the operation (argumentOne ^ argumentOne)

(2) Pair-wise test-set = { (^, numOne, numOne), (^, numTwo, numTwo), (^, numThree, numThree) }

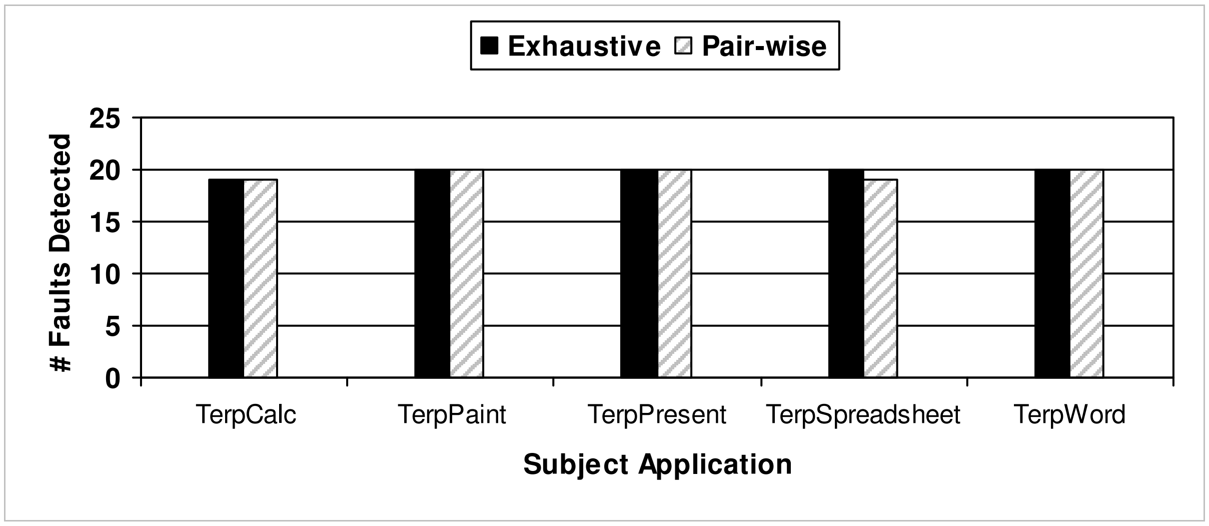

Figure 9: Fault-detection effectiveness of testing with exhaustive and pair-wise sampling

Figure 9 compares the faults found by test cases generated by the category partition and pair-wise sampling methods. The x-axis shows the subject applications and the y-axis shows the number of faults detected by each of the methods. The figure shows, that with the exception of TerpSpreadSheet, the pair-wise testing is as effective as category partition method.

One important observation from the graph is that, both the methods were not able to detect one fault in TerpCalc. As mentioned earlier, category partition method was able to find all the *reachable* faults. This seeded fault was never detected by the category partition method as well as pair-wise sampling, as it was not reachable. In TCGraph() of TerpCalc application we induced fault type 1, by modifying the code as following:

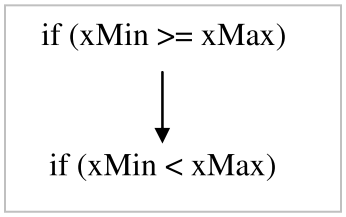

Figure 10: Seeding fault type 1 – modify the relational operator in a method

However, the only caller method TCGraphFrame() always set xMin= -10 and xMax = 10 when creating new TCGraph object. Hence, the condition was never checked.

We consider the cost as a linear function of the number of test cases executed in order to detect faults. Figure 11 compares the number of test cases executed with exhaustive and pair-wise sampling by subject application. The x-axis shows the subject applications and y-axis shows the number of test cases executed in hundreds.

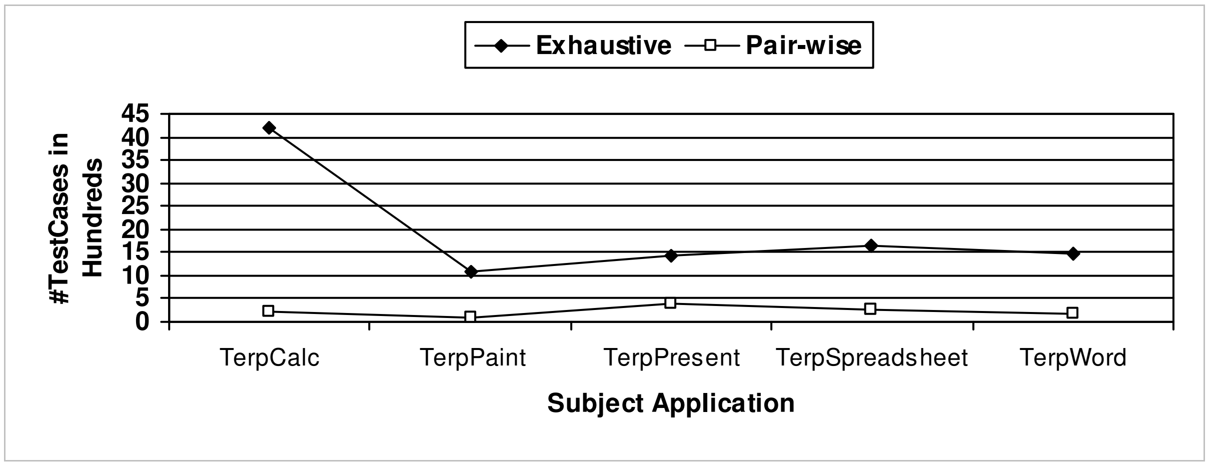

Figure 11: Test-suite size by subject application

Figure 12 shows the percentage reduction in the number of test cases when using pair-wise sampling compared to the exhaustive one (category-partition method). It is evident from the graph that we are able to achieve at least 70% of test-suite size reduction for all the applications.

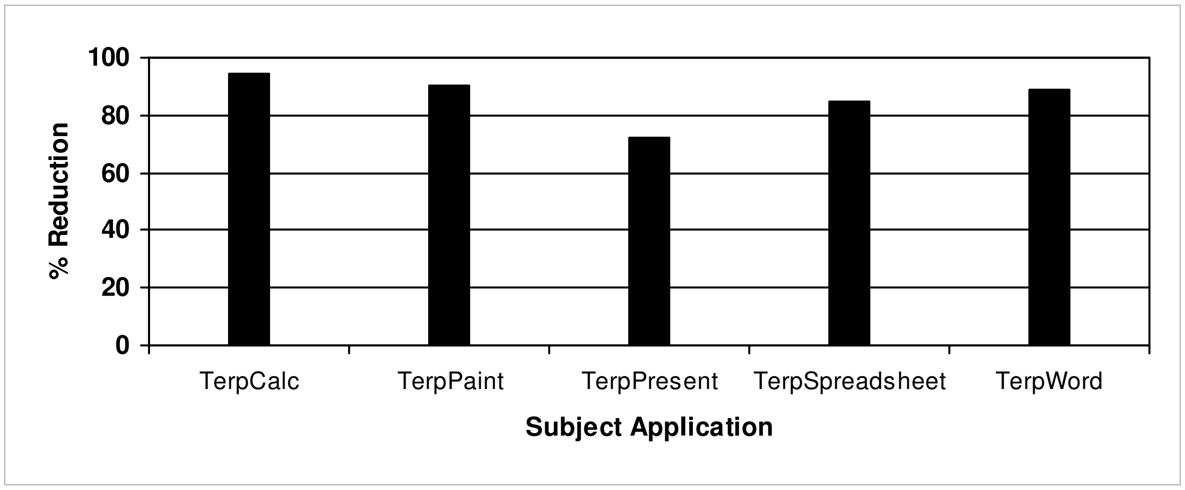

Figure 12: Percentage reduction in the test-suite size by subject application

5 Conclusion

We have explored the fault-detecting effectiveness of category partition (exhaustive) method and pair-wise sampling. The entire process has been feasible in terms of execution time, storage and the manual effort. We have empirically shown that, for the subject applications, pair-wise sampling is almost as effective as category partition method in finding seeded faults at much lower cost.

However, what we have presented here is an application-code based approach. While testing GUI one often wants to come up with an *interaction* (a sequence of events), that reveals faults by producing an outcome that is different from the original code. Such an interaction is not produced by a code-based empirical study. That is the reason we believe capture/replay tools, DART framework are more effective than a procedure involving generation of test cases using pair-wise sampling in detecting faults in GUI.

6 References

[1] M. Finsterwalder, "Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment," Proceedings of Second International Conference on eXtreme Programming and Flexible Processes in Software Eng., pp. 114-117, May 2001

[2] R. Hamlet, "Introduction to special section on software testing", Communications of the ACM June 1988, Volume 31 Issue 6

[3] J.H. Hicinbothom and W.W. Zachary, "A Tool for Automatically Generating Transcripts of Human-Computer Interaction," Proceedings of Human Factors and Ergonomics Society 37th Annual Meeting, p. 1042, 1993

[4] Nancy G. Leveson, "Software safety: why, what, and how", ACM Computing Surveys (CSUR) June 1986, Volume 18 Issue 2.

[5] Brian Marick, "When should a test be automated?" In Proceedings of the 11th International Software/Internet Quality Week, May 1998

[6] B. Marick, "Bypassing the GUI," Software Testing and Quality Engineering Magazine, pp. 41-47, Sept. 2002.

[7] Scott McMaster, Atif Memon, "Call-Stack Coverage for GUI Test Suite Reduction," IEEE Transactions on Software Engineering, vol. 34, no. 1, pp. 99-115, Jan., 2008

[8] Atif Memon, “A Comprehensive Framework For Testing Graphical User Interfaces”, PhD Thesis, 2001

[9] Atif M. Memon, Qing Xie, “Studying the Fault-Detection Effectiveness of GUI Test Cases for Rapidly Evolving Software”, IEEE Transactions on Software Engineering, vol. 31, no. 10, 2005, pp. 884-896

[10] G. Myers, “The art of software testing”, Wiley, 1979

[11] T. J. Ostrand, M. J. Balcer, “The category-partition method for specifying and generating functional tests”, Communications of the ACM June 1988, Volume 31 Issue 6.

[12] Stephen R. Schach, “Testing: principles and practice”, ACM Computing Surveys, (CSUR) March 1996, Volume 28 Issue 1.

[13] Kuo-Chung Tai; Yu Lei, “A test generation strategy for pair-wise testing”, Software Engineering, IEEE Transactions on, Volume: 28 Issue: 1, Jan. 2002, Page(s): 109 -111.

[14] L. White, H. AlMezen, and N. Alzeidi, “User-Based Testing of GUI Sequences and Their Interactions,” Proceedings of 12th International Symposium on Software Reliability Eng., pp. 54-63, 2001

[15] Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C., Regnell, B., and Wesslen, A. “Experimentation in software engineering: an introduction.” Kluwer Academic Publishers, Norwell, MA, USA, 2000

[16] Qing Xie, Atif M. Memon, “Designing and comparing automated test oracles for GUI-based software applications,” ACM Transactions on Software Engineering and Methodology, vol. 16, no. 1, 2007

[17] http://junit.sourceforge.net/doc/faq/faq.htm

[18] http://www.pairwise.org

[19] http://www.satisfice.com/tools/pairs.zip

[20] “JUnit, Testing Resources for Extreme Programming,” http://junit.org/news/extension/gui/index.htm , 2004.