Black Box Testing Technique: How All Pair Testing Covers Maximum Number of Software Defects

Shwetha M S
Unisys Global Services, Bengaluru, India

Dr. Girija Kasal, Professor, Dept of CSE, RNSIT, Bangalore

***Abstract*—Pair-Wise testing is a classic combinatorial test design technique, which is adoptable to any of the methodology from Waterfall to Agile methodology to save cost. No developer can say the code is bug free. Tester is one who never satisfied with the amount of testing is done. Tester always in application there is always a bug. For example, even in IPhone, Amazon and Flipkart there are many bugs.**

In testing, we could see four fundamental challenges. Complete testing is impossible. Testers misallocates resources because they fall for the company process myths. Test groups operates under multiple missions often conflicting rarely articulated. Test group often lack skilled programmers and a vision of appropriate project that would keep programming testers challenged. Why complete testing is impossible? Reasons are: Test every possible input to every variable, Test every possible combinations of input to every combination of variable, Test every possible sequence through the program and Test every hardware, software configuration including configuration of servers not under your control. Test every way in which the user might try to use the program.

This leads to motivation to pairwise testing. It is determined that 98% of reported software defect is recalled medical devices could have been detected by testing all pairs of parameters settings.

This paper focuses on Black Box technique, explained how All Pair testing covers maximum number of software defects with minimum Test cases, and thus saves time and cost of Software Development Life cycle. All Pair testing technique is applicable for different kind of testing such as Unit testing, Integration Testing, System Testing and Regression testing etc.,

Keywords*— *SDLC, All-Pair testing,Combinatorial testing, Software testing, t-way testing, System Under Test.

I. INTRODUCTION

Testing is an important testing step in the lifecycle of the development of software applications. To verify the correctness of applications hence quality measures we ideally test the software products in different ways by using different test techniques.

There are different types of testing methodologies. We may demarcate between black box testing and white-box testing based on the availability of source code. In addition, there are different techniques for integration tests, unit tests, system tests, and regression tests during the software development life cycle.

How to test?

- Input test data to the program.
- Observer the output
- Check if the program behaved as expected.

Examine Test Result.

- If the program does not behave as expected:
- Note the conditions under which it is failed (Test report).

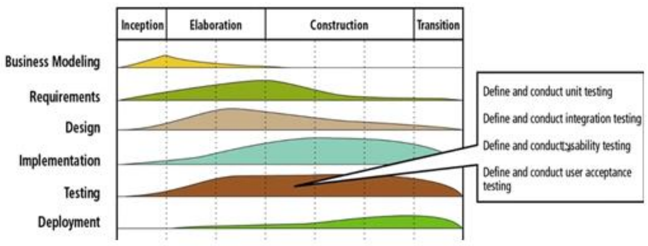
Testing Facts:

Programs used in very critical application such as Banking application, Railway Reservation system or Airline reservation system, if program suddenly crash, that will really put 10 to 1000 people in inconvenience. Therefore, programs before release, tested thoroughly so reliability is very high. Monkey testing does not work. Testing Technique has been evolved in 25-30 years.

Testing is effort intensive task. Very important in testing is Automation. Use of tool has become very important. Several tools available to help various testing activity including how much test has been done. When testing is carried out in SDLC? If we consider in Waterfall model, Testing happens in end phase of SDLC. Later development methodology like Agile or V-model, testing phase is spread all over the different phases. Any iterative development process, in every iteration testing is present all through life cycle. Bugs are explored earlier.

In the below fig1 shows effort required in Unified Process. There are 4 phases in Unified process are Inception, Elaboration, Construction, Transition. Testing effort is present all through life cycle.

Fig1: Testing Activities Now Spread Over Entire Life Cycle



Testing is getting more complex and sophisticated every year because of the below testing facts:

- Larger and more complex programs
- Newer programming paradigms
- Newer testing techniques
- Test Automation

Testing Perception

Initially testing is often viewed as not very challenging—less preferred by novices but now testing has taken a center stage in all types of software development. Large number of innovations have taken place in testing area, which requires tester to have good knowledge in test techniques.

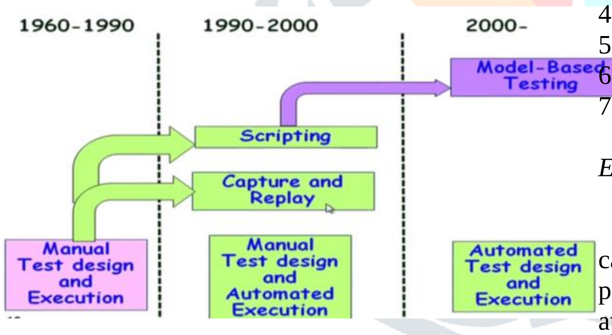
How many Latent Errors? 85% errors are removed at the end of a typical testing process. Why not more? All practical test techniques are heuristics. They help to reduce bugs but not guarantee complete bug removal. Removal of 100% of bugs not possible.

Evolution of Test Automation

As showed in diagram Test Case design and Execution is manual until 1990. More or less manual. After 1990 test tools appeared. Capture and replay tool as the tester input test cases captures test input. As the tester, input captures the result. Next time it automatically repeated. It is a big help for testing in Regression testing.

Another category of tool is called Scripting. Testers need to write Test cases as program. Advantage is Scripting type is more reusable. Initially it may take more time for writing script it is more reusable.

Fig2: Evolution of Test Automation


1960-1990
1990-2000
2000-
Model-Based Testing
Scripting
Capture and Replay
Manual Test design and Execution
Manual Test design and Automated Execution
Automated Test design and Execution

Testing Design Technique

Testing design Technique is a method to derive effect test cases from all possible test cases. Categorized into

- Black Box test Design and
- White Box Test Design

Black Box test Design: Derives test cases from functional specification of the software. Focused on the behavior on the Software on efficiency of the performance. In this not focused on the structural design. Means tester focused only on the input and output of the software and not on the how input and output is generated. It facilitates testing communication amongst modules through integration testing

As a customer, what input is giving and what is the expected output. It can be applied on both functional and non-functional mode of software testing such as performance and Scalability. It is also used in Regression testing.

Fig3: Black Box test Design

Input
Black Box
OutPut

How to Perform Black Box testing:
Generic steps followed to carry out Black Box Testing

1) Initially requirements and specification of the system are examined.
2) Tester chooses valid inputs to check whether application under test processes them correctly. In addition, some invalid inputs are chosen to verify that the application or SUT is able to detect them.
3) Tester determines expected output for all those input.
4) Software testers creates Test cases.
5) The Test cases are executed.
6) Software tester compares the actual outputs with the expected outputs.
7) Defects if any are fixed and re-tested.

Black Box Test Design Techniques are:

1) Equivalence Partitioning
2) Boundary Value Analysis
3) Decision Table
4) State Transition
5) Exploratory Testing
6) Error Guessing
7) Combinatorial testing

Equivalence Partitioning

Test cases are divided into set of logical groups called partition, which exhibits similar behavior when processed. Each Partition covers specific aspect of the application. No need to create Test cases covering all the condition. Instead, one Test case from each condition is tested. It saves lot of time. For example, Consider User name attribute allow numeric value dividing into logical groups

1) Alphabet—one logical group
2) Numerical—One logical group

One Test case need to add from Alphabet and another from Numerical logical group. No need to create to cover all test cases. One Test case is derived from each group.

Need to test only one condition from each partition. This is because we are assuming that all the condition works in the same manner. If one condition from partition works then all the condition in the partition works.

Boundary Value Analysis

To access the input at the boundary of each equivalence partition we use this technique. It helps to find more number of defects. For example: If we want to test a field which should accept only currency more than 100 and less than 200 then we take the boundaries as 100 minus 1,100,100 Plus 1, 200 minus 1, 200, 200 plus 1. While testing design the test case we just use 99,100,101,199,200 and 201.

Decision Table

This technique is also called as Cause- effect Table. It is good for functionality, which logically has relationship between inputs that is if-else logic. This helps to find combinations of input. Test cases are identified by considering Conditions as input and Actions are output.

State Transition

Here Test cases are selected from an application where different system transition needs to check. It is applied when an application gives a different output for the same input, depending on earlier stage. For e.g. Traffic Light will change sequence when cars are moving or waiting.

Exploratory Testing

Domain Experts do this type of testing. Testing is done by exploring application without knowledge of requirements. This is a good technique for testers to explore and learn application of the system. High severity bugs are found in this type of testing.

Error Guessing

Bugs are detected based on prior experience of testers. No specific rules are followed. It is unplanned testing. Some of the examples are submitting a form without entering values in the mandatory field.

Combinatorial Testing

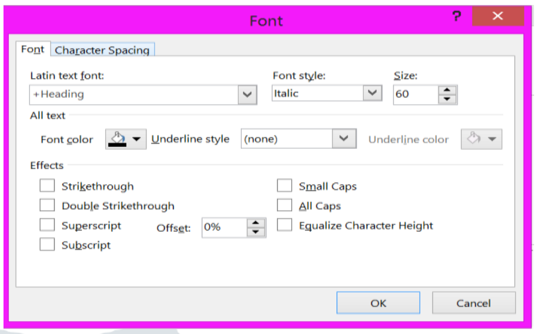
It is another Black box testing. The behavior of the program may be affected by many factors such as input parameter, Environment configuration and State variables. Equivalence partitioning and special value testing is difficult to design test cases, when number of parameters are more. Also sometimes, we have environmental configuration effect the test result. For e.g. program, set in expert mode or novis mode program behaves differently. We might have other state variables, which may affect different components of the program.

Equivalence partitioning of input variable identifies the possible types of input values requiring different processing. If the factors are more than two or three, it is impractical to test all the possible combinations of values of all factors.

In the above case, it is difficult to design test case by using Equivalence partitioning. Sometimes there are many Boolean variables in User interface and controller application. For e.g.: Font setting in PowerPoint software, there are number of values such as depending on the options we select such as Small Caps, All Caps , Super Script, Subscript etc., Font size, Font style and color etc. Therefore, font looks different.

For eg: Font Style is Italic, Size is 34, colour is Red, All caps is ON, and Superscript is ON. For this situation how to get Equivalence, class partitioning? It becomes difficult.
Fig: User Interface of Font setting in PowerPoint

Fig4: Font setting in PowerPoint software


Font
Font | Character Spacing
Latin text font: +Heading
Font style: Italic
Size: 60
All text
Font color
Underline style: (none)
Underline color
Effects
☐ Strikethrough
☐ Double Strikethrough
☐ Superscript Offset: 0%
☐ Subscript
☐ Small Caps
☐ All Caps
☐ Equalize Character Height

Let us look into CT. In the above example some parameters are directly input, some parameters are state parameter variables which we need to test.
Several type of combinatorial testing:

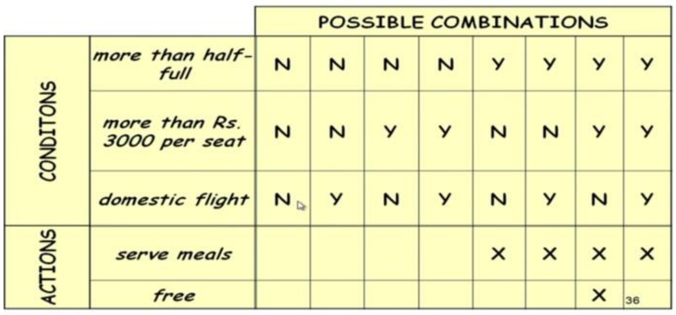
- Decision table based testing
- Cause effect graphing
- Pairwise testing

Decision table Based Testing:

It is applicable to requirements involving Conditional actions. It can be automatically translated into code. In the below decision table, Conditions are input parameters. Actions are output and Rules are test cases. Each column represents Test case.

For example: Policy for charging customer for certain inflight services: If flight is more than half-full and ticket cost is more than Rs.3000 free meals are served unless it is a domestic flight. The meals are charged on all domestic flights.

Fig5: Decision table for charging customer for certain inflight services



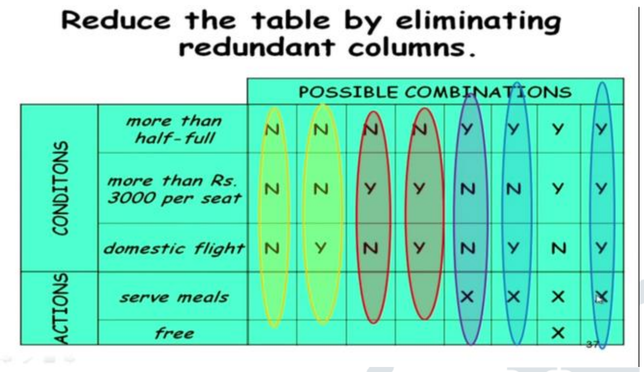
		POSSIBLE COMBINATIONS							
CONDITONS	more than half-full	N	N	N	N	Y	Y	Y	Y
	more than Rs. 3000 per seat	N	N	Y	Y	N	N	Y	Y
	domestic flight	N	Y	N	Y	N	Y	N	Y
ACTIONS	serve meals					X	X	X	X
	free							X	36

A free Meal is served free only when it is more than half-full, more than Rs.3000 per seat and it should not be a domestic flight. Serve meals are served when more than half-full and ticket cost is less /more than Rs.3000 and irrespective of domestic flight or not. However, not free meal. Hence, each column becomes test case. For N parameter and if the parameter is Boolean then number of Test Cases 2N

Causes	Effects
c1:Deposit<1year	e1:Rate 6%
c2:1year<deposit<3year	e2:Rate 7%
c3:Deposit>3years	e3:Rate 8%
c4:Deposit<1lakh	e4:Rate 9%
c5:Deposit>=1lakh	

Some of the parameter are do not care. We can remove redundant test cases.

Fig 6: Eliminating Redundant Test cases



Reduce the table by eliminating redundant columns.

		POSSIBLE COMBINATIONS							
CONDITONS	more than half-full	N	N	N	N	Y	Y	Y	Y
	more than Rs. 3000 per seat	N	N	Y	Y	N	N	Y	Y
	domestic flight	N	Y	N	Y	N	Y	N	Y
ACTIONS	serve meals					X	X	X	X
	free							X	

Hence, the final solution is as given below:

- Guidelines and Observation: Decision table testing is most appropriate for programs :
- There is lot of decision-making.
- There are important logical relationship among input variables.
- There are calculations involving subsets of input variables.
- There are cause and effect relationship between input and output.
- There is complex computation logic.

Limitation of Decision Table method: Decision table cannot scale up very well. If the number of parameter are, less we can use this method. If number of parameter is around 30 and each parameter has three values then it is difficult to design Decision Table method. It creates combinatorial explosion problem. To overcome from these Cause-Effects graphing technique is introduced.

Cause-Effects graph

It explores combination of possible inputs. Specific combination of inputs are called as causes and output are called as affects. Let us see how it avoids combinatorial explosion problem?

Here combination are represented as nodes of a cause effect graph. The graph also include constraints and a number of intermediate nodes linking causes and effects.

Cause-Effect graph example

1) If depositing less than rupees 1lakh rate of interest:
   6% for deposit up to 1 year
   7% for deposit over 1 year but less than 3 years
   8% for deposit 3 years and above

2) If depositing more than rupees 1lakh rate of interest:
   7% for deposit up to 1 year
   8% for deposit over 1 year but less than 3 years
   9% for deposit 3 years and above

Here first we need to identify the cause and effects

Fig 7: To identify the cause and effects

Cause-Effect Graphing:

Here causes are the different inputs, we have five inputs in the above example and can be represented in the below diagram. c1 to c5 are the different inputs. e10, e20, e30, e40, e50, e60 are the intermediate nodes. e1, e2, e3, e4 are outputs.

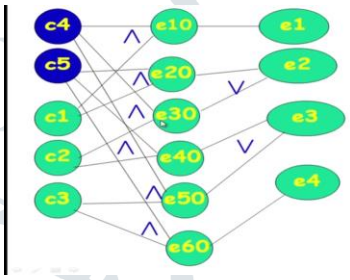
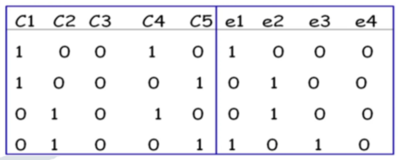


Fig 8: Cause-Effect Graphing

From this chart it is easy to develop decision table, the below fig represents the decision table. Here each column represents a test case. It is very simple technique to come up with decision table avoid exponential combinations of test cases. Decision table derived from the Cause-Effect graph:

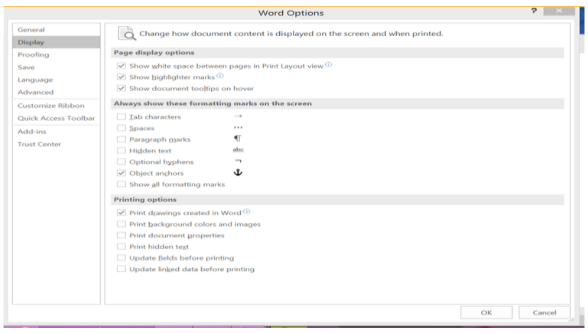


C1	C2	C3	C4	C5	e1	e2	e3	e4
1	0	0	1	0	1	0	0	0
1	0	0	0	1	0	1	0	0
0	1	0	1	0	0	1	0	0
0	1	0	0	1	1	0	1	0

Fig 9: Pair-wise Testing

It is one more Black box testing when the number of input is large, also called as all pair testing. It is that fault is caused by interactions among a few factors. When many parameters has Boolean values for ex, as given in the below font

Fig10: Display setting in Word software



Change how document content is displayed on the screen and when printed.

Page display options
- [✓] Show white space between pages in Print Layout view
- [✓] Show highlighter marks
- [✓] Show document tooltips on hover

Always show these formatting marks on the screen
- [ ] Tab characters →
- [ ] Spaces ···
- [ ] Paragraph marks ¶
- [ ] Hidden text abc
- [ ] Optional hyphens ¬
- [✓] Object anchors ⚓
- [ ] Show all formatting marks

Printing options
- [✓] Print drawings created in Word
- [ ] Print background colors and images
- [ ] Print document properties
- [ ] Print hidden text
- [ ] Update fields before printing
- [ ] Update linked data before printing

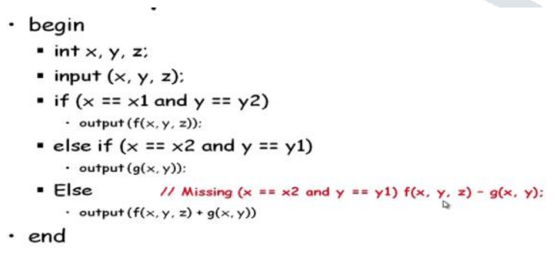
In the fig10, we could see many binary combinations. In combinatorial testing all possible combinations are generated. It is often, the fault is caused by interactions among a few factors. Combinatorial testing can dramatically reduce the number of test cases, but remains effective in terms of fault detection. Researchers are experimented with large number of software's and found that all bugs that found out if we consider two to six combinations. So if we have 40 input variables. For ex p1…p40 and each parameter takes two values, for exhaustive testing we need to execute 240 test cases. It is found experimentally all the bugs can be detected using all-pair testing, i.e. two-way testing to six-way testing. It is proved that in two-way testing 80% of the bugs are detected, in the three-way testing 90% of the bugs are detected by 4 or 5 way testing all the bugs have been detected. Hence, no need to consider all 240 test cases. Why pair-wise testing works is that if fault is caused by interactions among few factors. One test case covers many pair values. Number of Test cases required much less generating pairwise combination, which drastically reduces. If we consider all pair wise values then number of test cases may 10 or 12. Tools are available.

Fault Model: A T-way interaction fault is triggered by a certain combination of t input vales. A simple fault is 1 way fault. Pairwise fault is a t-way fault where t=2. In practice, a majority of software faults consist of simple and pairwise.

Single-mode Bugs: Simplest bugs are single-mode faults. It occurs when one option causes a problem regardless of the other settings. For e.g. A printout is always is smeared when you choose the duplex option in the print dialog box.

Double-mode faults: It occurs when two options are combined. For example, the printout is smeared only when duplex is selected and the printer selected is model 394.

Multi-Mode faults: It occurs when three or more settings produce the bug. This is the type of problems that make complete coverage seem necessary. Example of Pairwise Fault: Analysis of Program for Pairwise Technique: Below is the program, which has 3 values. It checks if Programmer missed to write a statement x==x2 and y==y2.



```
• begin
    ▪ int x, y, z;
    ▪ input (x, y, z);
    ▪ if (x == x1 and y == y2)
        · output(f(x, y, z));
    ▪ else if (x == x2 and y == y1)
        · output(g(x, y));
    ▪ Else          // Missing (x == x2 and y == y1) f(x, y, z) - g(x, y);
        · output(f(x, y, z) + g(x, y))
• end
```

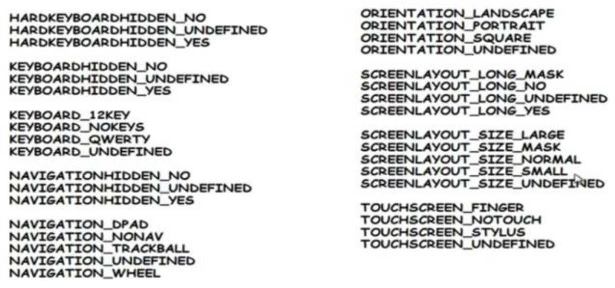
Expected Result:

- When x=x1andy=y1==>f(X,Y,Z)- g(X,Y)
- When x=x2,y=y2==>f(X,Y,Z)+g(X,Y)

It is another example where Android Smart phone testing is done for different environmental variables. If you consider these variables, there are 172,800 combinations of Test cases.

Can we have simple algorithm? To generate pair wise test cases. Generating optimal Test cases is hard problem. Few algorithm genetically algorithm and evolutionary algorithm can be used.

Android Smart Phone Testing: Exhaustive Combination =3*3*4*3*5*4*4*5*4=172 800 Combinations



HARDKEYBOARDHIDDEN_NO
HARDKEYBOARDHIDDEN_UNDEFINED
HARDKEYBOARDHIDDEN_YES

KEYBOARDHIDDEN_NO
KEYBOARDHIDDEN_UNDEFINED
KEYBOARDHIDDEN_YES

KEYBOARD_12KEY
KEYBOARD_NOKEYS
KEYBOARD_QWERTY
KEYBOARD_UNDEFINED

NAVIGATIONHIDDEN_NO
NAVIGATIONHIDDEN_UNDEFINED
NAVIGATIONHIDDEN_YES

NAVIGATION_DPAD
NAVIGATION_NONAV
NAVIGATION_TRACKBALL
NAVIGATION_UNDEFINED
NAVIGATION_WHEEL

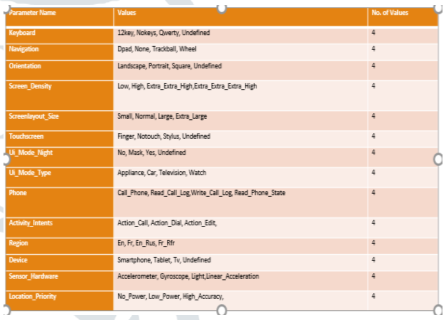
ORIENTATION_LANDSCAPE
ORIENTATION_PORTRAIT
ORIENTATION_SQUARE
ORIENTATION_UNDEFINED

SCREENLAYOUT_LONG_MASK
SCREENLAYOUT_LONG_NO
SCREENLAYOUT_LONG_UNDEFINED
SCREENLAYOUT_LONG_YES

SCREENLAYOUT_SIZE_LARGE
SCREENLAYOUT_SIZE_MASK
SCREENLAYOUT_SIZE_NORMAL
SCREENLAYOUT_SIZE_SMALL
SCREENLAYOUT_SIZE_UNDEFINED

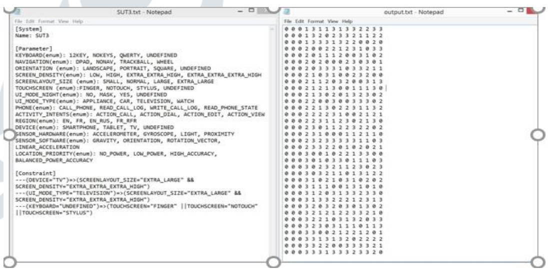
TOUCHSCREEN_FINGER
TOUCHSCREEN_NOTOUCH
TOUCHSCREEN_STYLUS
TOUCHSCREEN_UNDEFINED

Fig11: Android Smart Phone Testing



Parameter Name	Values	No. of Values
Keyboard	12key, Nokeys, Qwerty, Undefined	4
Navigation	Dpad, None, Trackball, Wheel	4
Orientation	Landscape, Portrait, Square, Undefined	4
Screen_Density	Low, High, Extra_Extra_High,Extra_Extra_Extra_High	4
Screenlayout_Size	Small, Normal, Large, Extra_Large	4
Touchscreen	Finger, Notouch, Stylus, Undefined	4
Ui_Mode_Night	No, Mask, Yes, Undefined	4
Ui_Mode_Type	Appliance, Car, Television, Watch	4
Phone	Call_Phone, Read_Call_Log,Write_Call_Log, Read_Phone_State	4
Activity_Intents	Action_Call, Action_Dial, Action_Edit,	4
Region	En, Fr, En_Rus, Fr_Rfr	4
Device	Smartphone, Tablet, Tv, Undefined	4
Sensor_Hardware	Accelerometer, Gyroscope, Light,Linear_Acceleration	4
Location_Priority	No_Power, Low_Power, High_Accuracy,	4

Fig12: Output of Pair wise testing data when interaction level t=2



SUT3.txt - Notepad

```
[System]
Name: SUT3

[Parameter]
KEYBOARD(enum): 12KEY, NOKEYS, QWERTY, UNDEFINED
NAVIGATION(enum): DPAD, NONAV, TRACKBALL, WHEEL
ORIENTATION (enum): LANDSCAPE, PORTRAIT, SQUARE, UNDEFINED
SCREEN_DENSITY(enum): LOW, HIGH, EXTRA_EXTRA_HIGH, EXTRA_EXTRA_EXTRA_HIGH
SCREENLAYOUT_SIZE (enum): SMALL, NORMAL, LARGE, EXTRA_LARGE
TOUCHSCREEN (enum):FINGER, NOTOUCH, STYLUS, UNDEFINED
UI_MODE_NIGHT(enum): NO, MASK, YES, UNDEFINED
UI_MODE_TYPE(enum): APPLIANCE, CAR, TELEVISION, WATCH
PHONE(enum): CALL_PHONE, READ_CALL_LOG, WRITE_CALL_LOG, READ_PHONE_STATE
ACTIVITY_INTENTS(enum): ACTION_CALL, ACTION_DIAL, ACTION_EDIT, ACTION_VIEW
REGION(enum): EN, FR, EN_RUS, FR_RFR
DEVICE(enum): SMARTPHONE, TABLET, TV, UNDEFINED
SENSOR_HARDWARE(enum): ACCELEROMETER, GYROSCOPE, LIGHT, PROXIMITY
SENSOR_SOFTWARE(enum): GRAVITY, ORIENTATION, ROTATION_VECTOR,
LINEAR_ACCELERATION
LOCATION_PRIORITY(enum): NO_POWER, LOW_POWER, HIGH_ACCURACY,
BALANCED_POWER_ACCURACY

[Constraint]
---(DEVICE="TV")=>(SCREENLAYOUT_SIZE="EXTRA_LARGE" &&
SCREEN_DENSITY="EXTRA_EXTRA_EXTRA_HIGH")
---(UI_MODE_TYPE="TELEVISION")=>(SCREENLAYOUT_SIZE="EXTRA_LARGE" &&
SCREEN_DENSITY="EXTRA_EXTRA_EXTRA_HIGH")
---(KEYBOARD="UNDEFINED")=>(TOUCHSCREEN="FINGER" ||TOUCHSCREEN="NOTOUCH"
||TOUCHSCREEN="STYLUS")
```

output.txt - Notepad

```
0 0 0 1 3 1 1 3 1 3 3 2 2 3 3
0 0 0 1 3 2 0 2 3 3 2 1 1 2 2
0 0 0 1 3 3 3 1 3 2 2 0 0 2 0
0 0 0 2 0 0 2 2 1 2 3 1 0 3 3
0 0 0 2 0 1 1 1 2 0 0 3 1 0 2
0 0 0 2 0 2 0 0 0 2 3 0 3 0 1
0 0 0 2 0 3 3 3 1 0 3 3 2 1 1
0 0 0 2 1 0 3 1 0 0 2 3 2 0 0
0 0 0 2 1 1 2 0 3 2 0 0 3 1 3
0 0 0 2 1 2 1 3 0 0 1 1 1 3 0
0 0 0 2 1 3 0 2 0 1 3 2 3 0 2
0 0 0 2 2 0 0 3 0 0 3 3 3 0 2
0 0 0 2 2 1 3 0 2 2 3 1 1 3 2
0 0 0 2 2 2 2 3 1 0 0 2 1 2 1
0 0 0 2 2 3 1 1 2 3 0 2 1 3 0
0 0 0 2 3 0 1 1 2 2 3 2 2 0 2
0 0 0 2 3 1 0 0 0 1 1 2 1 1 0
0 0 0 2 3 2 3 3 3 3 3 1 1 0 3
0 0 0 2 3 3 2 2 0 1 0 2 0 2 1
0 0 0 3 0 0 1 0 2 2 1 3 3 0 0
0 0 0 3 0 1 0 3 3 0 1 1 1 0 3
0 0 0 3 0 2 3 2 1 1 2 3 0 2 3
0 0 0 3 0 3 2 1 1 0 1 3 1 2 2
0 0 0 3 1 0 2 1 0 3 1 0 2 0 2
0 0 0 3 1 1 1 0 0 1 3 1 0 1 0
0 0 0 3 1 2 0 3 1 3 3 2 3 3 0
0 0 0 3 1 3 3 2 2 2 1 2 3 1 3
0 0 0 3 2 0 3 2 0 3 0 1 3 0 2
0 0 0 3 2 1 2 1 2 2 3 3 2 1 0
0 0 0 3 2 2 1 0 3 1 3 2 0 3 3
0 0 0 3 2 3 0 3 1 1 1 0 1 1 3
0 0 0 3 3 0 0 2 1 2 2 1 2 0 1
0 0 0 3 3 1 3 1 3 2 0 2 2 2 2
0 0 0 3 3 2 2 0 0 3 3 3 3 2 1
0 0 0 3 3 3 1 3 3 3 2 3 3 2 0
```

Application of Combinatorial Testing:

Combinatorial Testing research is widely popular because it can be applied on various types of different applications. In 1926, Fisher pioneered interaction tests in agricultural experiments, assessing the contributions of different fertilizers to crop yield in the context of soil heterogeneity and environmental factors such as erosion, Sun coverage, rainfall and pollen. Exhaustive testing is not possible with the limited resources for testing. Fisher applied interaction testing so that every pair of factors affecting the yield was included exactly once.

In Software testing, Mandl proposed to test Ada compiler by using pairwise combinatorial testing in 1985. Generated Test set using Orthogonal Latin Squares. The test cases generated by using CT can detect more errors than never detected previously.

Combinatorial testing has been applied to other applications as well. Almezen proposed a method, which focused on user sequences of GUI objects, and selections, which collaborate, called Complete Interaction Sequences. It is also applied for many types of system.

Conclusion and Future Work

In last 20 years, combinatorial testing is widely studied and applied. Now, it is a well acceptance testing method and proved ability of detecting interaction failures.

In this paper thorough study of Black Box, testing is done and advantages of Combinatorial testing strategy focusing mainly on test case generation. The following areas of focus in CT research in future.

Identification of good model of the test parameters is critical to combinatorial testing. We need effective ways to identify the parameters of SUT, determine the values of each Parameter and explore the interactions and constraints existing among the parameters.

Although many methods have been proposed to generate test suite for Combinatorial Testing, as the problem of test suite generation is NP-hard, there is room for further improvement of these test generation methods. In particular, a good method should support the use of seeding and make full use of constraints in generating a set of feasible test cases.

We may combine CT with other testing technique such as prioritization, to ensure most important test cases are executed early. It also can combine with metamorphic testing to solve the oracle problem of CT by automating the process to determine whether a test passes or fails.

It is one of the most effective software testing technique as it test a software with multiple configurable parameters. Moreover, with the assistance of combinatorial testing one can easily detect interactions faults caused by the combination of parameters. Another advantage of this type of testing is that it produces high quality testing at a very cost effective rate, which not only helps software developers and testers, but also benefits the organization for which the product is being developed. Therefore, other benefits of this approach are handles coverage concerns when defining the test plan. Allows systematic planning of test. Can be virtually applied to any software and at different levels of abstractions. Higher test coverage with better quality assurance. Requires no access to internal source code SUT. It maximizes the value of each tested scenario. Significant reduction in the number of tests. It can control risks and is easy to review.

Software systems are complex and can incur exponential numbers of possible tests. Any product that is released without proper testing can be a significant danger to the organization as well as the user. Therefore, to ensure that no such situation or problem occurs after the software is released, software testers perform rigorous testing. Moreover, they frequently use combinatorial testing in various testing levels, as it can easily test software with multiple configurable parameters. In short, combinatorial testing is used to detect interaction faults caused by the combination of parameters. The key insight underlying the effectiveness of combinatorial testing resulted from a series of studies and research done by NIST from 1999-2004. It an immensely useful approach that can systematically examine system setting in a manageable number of test. An approach produces and executes high quality testing at a very cost effective rate. Furthermore, it is an effective test planning technique, which can handle coverage concerns as early as possible. Hence, if a software engineer wants to get best testing results, they should for sure execute combinatorial testing at an early stage of Software Development Life Cycle (SDLC).

ACKNOWLEDGMENTS

I am grateful to Professor Girijamma .H.A for her comments. I appreciate valuable review comments given by Roshitha Shenoy, which helped me to revise this article.

REFERENCES

[1]. B.S. Ahmed, Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing, Eng. Sci. Technol. Int. J. (2015), http://dx.doi.org/10.1016/ j.jestch.2015.11.006.

[2]. D.R. Chicago Kuhn, R.N. Kacker, Y. Lei, Practical Combinatorial Testing, NIST Special Publication, 2010. 800-142.

[3]. Test case minimization approach using fault detection and combinatorial optimization techniques for configuration-aware structural testing Bestoun S. Ahmed a

[4]. K.C. Tai, Y. Lie, In-parameter-order: a test generation strategy for pairwise testing,in: 3rd IEEE International Symposium on High-Assurance Systems Engineering,Washington, DC, USA, 1998, pp. 254–261.

[5]. V.V. Kuliamin, A. Petoukhov, A survey of methods for constructing covering arrays, Programming and Computer Software 37 (2011) 121–146. [34] K. C. Tai

[6]. C. Nie, H. Leung, A survey of combinatorial testing, ACMComput. Surv. 43 (2011) 1–29.

[7]. S. Dejam, M. Sadeghzadeh, S.J. Mirabedini, Combining cuckoo and tabu algorithms for solving quadratic assignment problems, Journal of Academic and Applied Studies 2 (2012) 1–8

[8]. Williams AW. Determination of test configurations for pair-wise interaction coverage. Proceedings of 13th International Conference on the Testing of Communicating Systems, Ottawa, Canada, 2000; 59–74.

[9]. A Survey of Combinatorial Testing CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University. HARETON LEUNG, Hong Kong Polytechnic University

[10]. B.S. Ahmed, M.A. Sahib, M.Y. Potrus, Generating combinatorial test cases using simplified swarm optimization (SSO) algorithm for automated GUI functional testing, Eng. Sci. Technol. Int. J. 17 (4) (2014) 218–226.

[11]. M. A. Chateauneuf, C. J. Colbourn, and D. L. Kreher, "Covering Arrays of Strength 3," Designs, Codes, and Cryptography, vol. 16, pp. 235-242, 1999.

[12]. M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Constructing Strength Three Covering Arrays with Augmented Annealing," Discrete Mathematics, vol. 308, pp. 2709-2722, 2008.

[13]. NIST, "Web Site, Automated Combinatorial Testing for Software, available from http://csrc.nist.gov/groups/SNS/acts, last accessed on September, 2010."

[14]. C. Nie, B. Xu1, L. Shi1, and G. Dong, "Automatic Test Generation for N-Way Combinatorial Testing," LNCS, vol. 3712, pp. 203-211, Friday, September 09, 2005 2005.

[15]. A. W. Williams, J. H. Ho, and A. Lareau, "TConfig Test Tool Version 2.1,"Ottawa, Ontario, Canada, available from http://www.site.uottawa.ca/~awilliam, last access on March 2010:School of Information Technology and Engineering (SITE), University of Ottawa, 2003.

[16]. A. W. Williams and R. L. Probert, "A Practical Strategy for Testing Pair-Wise Coverage of Network Interfaces," in Proceedings of the 7th International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, New York, 1996, pp. 246-254.

[17]. A. W. Williams and R. L. Probert, "A Measure for Component Interaction Test Coverage," in Proceedings of the ACSI/IEEE International Conference on Computer Systems and Applications (AICCSA 2001), Beirut, Lebanon, 2001, pp. 304-311.

[18]. MIPOG - An Efficient t-Way Minimization Strategy forCombinatorial Testing Mohammed I. Younis and Kamal Z. Zamli

[19]. ADVANCED COMBINATORIAL TESTING ALGORITHMS AND APPLICATIONS by LINBIN YU

[20]. Bush KA. Orthogonal arrays of index unity. Annals of Mathematical Statistics 1952; 23:426–434

[21]. S. Maity, A. Nayak, M. Zaman, N. Bansal, and A. Srivastav, "An Improved Test Generation Algorithm for Pair-Wise Testing," in

Proceedings of the 14th International Symposium on Software Reliability Engineering (Fast Abstract ISSRE 2003) Denver, Colorado: Chillarege Press, 2003

[22]. B. Garvin, M. Cohen and M. Dwyer, "Evaluating Improvements to a Meta-Heuristic Search for Constrained Interaction Testing," Empirical Software Engineering (EMSE), vol. 16, no. 1, pp. 61-102, 2011. .

[23]. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-way Combinatorial Testing," Software Testing, Verification, and Reliability, vol. 18, pp. 125-148, 2008.

[24].

[25]. Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "IPOG: A General Strategy for T-Way Software Testing," in Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS2007), Tucson, AZ, 2007, pp. 549-556.

[26]. M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the In-Parameter-Order Strategy for Constructing Covering Arrays," Journal of Research of the National Institute of Standards and Technology, vol. 113, pp. 287-297., October 2008 2008.

[27]. M. I. Younis, K. Z. Zamli, M. F. J. Klaib, Z. C. Soh, S. C. Abdullah, and N. A. M. Isa, "Assessing IRPS as an Efficient Pairwise Test Data Generation Strategy," International Journal of Advanced Intelligence Paradigms (IJAIP), vol. 2, pp. 90-104, 2010.

[28]. D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The Combinatorial Design Approach to Automatic Test Generation," IEEE Software, vol. 13, pp. 83-88, 1996.

[29]. D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing based on Combinatorial Design," IEEE Transactions on Software Engineering, vol. 23, pp. 437–443, 1997.

[30]. K. Burr and W. Young, "Combinatorial Test Techniques: Table Based Automation, Test Generation and Code Coverage," in Proceedings of the International Conference on Software Testing Analysis & Review (STAR), San Diego, CA, 1998, pp. 503-513.

[31]. [Grindal M, Offutt J, Andler SF. Combination testing strategies—A survey. Journal of Software Testing, Verification and Reliability 2004; 5(3):167–199.

[32]. Cohen MB, Colbourn CJ, Gibbons PB, Mugridge WB. Constructing test suites for interaction testing. Proceedings of 25th IEEE International Conference on Software Engineering, Portland, Oregon, 2003; 38–48.

[33]. Kuhn DR, Reilly MJ. An investigation of the applicability of design of experiments to software testing. Proceedings of 27th NASA/IEEE Software Engineering Workshop, Greenbelt, Maryland, 2002; 91–95.

[34]. Kuhn DR, Wallace D, Gallo A. Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 2004; 30(6):418–421.

[35]. IPOG/IPOG-D: efficient test generation for multi-way combinatorial testing Yu Lei1,∗, †, Raghu Kacker2, D. Richard Kuhn2, Vadim Okun2 and James Lawrence3