Software Testing Techniques & Test Case Minimization for Boundary Value Analysis

By

Md. Samiul Hasan Khondaker
ID: 111-35-165
Department of Software Engineering
Daffodil International University, Dhaka

&

Nazmus Sakib
ID: 111-35-164
Department of Software Engineering
Daffodil International University, Dhaka

A Thesis Submitted on Software Testing Techniques & Test Case Minimization for Boundary Value Analysis for the Degree of Bachelor of Science (Engineering)
In
The Department of Software Engineering

Supervised By
Dr. Shaikh Muhammad Allayear
Associate Professor
Department of Software Engineering
Daffodil International University, Dhaka

DAFFODIL INTERNATIONAL UNIVERSITY, DHAKA
April 2016

APPROVAL

This Project titled **“Software Testing Techniques & Test Case Minimization for Boundary Value Analysis”** submitted By Md.Samiul Hasan Khondaker ID No: 111-35-165 & Nazmus Sakib ID No:111-5-164 to the Department of Software Engineering, Daffodil International University, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of B.Sc. in Software Engineering and approved as to its style and contents. The presentation has been held 9th April, 2016.

DECLARATION

We hereby declare that we have taken this thesis under the supervision of **Dr. Shaikh Muhammad Allayear, Associate Professor, Department of Software Engineering, Daffodil International University.** We also declare that neither this thesis nor any part of this has been submitted elsewhere for award of any degree.

Supervised by:

[illegible]

....

Dr. Shaikh Muhammad Allayear
Associate Professor
Department of Software Engineering
Daffodil International University

Submitted by:

[illegible]

....

Nazmus Sakib
ID: 111-35-164
Batch: 4th
Department of Software Engineering
Daffodil International University

[illegible]

....

Md. Samiul Hasan Khondokar
ID: 111-35-165
Batch: 4th
Department of Software Engineering
Daffodil International University

ACKNOWLEDGMENTS

First and foremost, we are thankful to Almighty Allah for his blessings on us to contribute to the knowledge of world. We are particularly grateful to our supervisor Dr. Shaikh Muhammad Allayear, for his invaluable expertise and advice, inspiring and challenging discussions, and endless patience that have supported us throughout this work. We would also like to thank our department head, Dr. Touhid Bhuyian, for his enthusiastic guidance and excellent comments on our work.

The research presented in this thesis was conducted in close cooperation between academia and industry. Recognition must also be given here to the coauthors of my papers and others who have helped writing and reviewing us. In particular, we would like to thank our all batch mates (4th Batch) in the Software Engineering Department for an inspiring and supporting collaborative atmosphere. We would also like to mention all other faculty members at the Department of Software Engineering and all students of this department, thanks for providing an excellent environment to work in.

Last but not the least, we would like to thank our family and friends for constantly reminding us of the most important things in life and for always supporting us in critical situation.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

List of Abbreviations

AUT	**Application Under Test**
BVA	**Boundary Value Analysis**
COTS	**Commercial Off-The-Shelf**
ECP	**Equivalence Class Partitioning**
HTML	**Hyper Text Markup Language**
IEEE	**Institute Of Electrical and Electronics Engineers**
I/O	**Input Output**
ISO	**International Organization For Standardization**
ISTQB	**International Software Testing Qualification Board**
PIN	**Personal Identity Number**
SDLC	**Software Development Life Cycle**
SRS	**Software Requirement Specification**
STLC	**Software Testing Life Cycle**
UAT	**User Acceptance Test**
XML	**Extensible Markup Language**

ABSTRACT

Software Testing is the most important phase of Software development processes. Software Testing is the process of finding error or defect in the system and its documentation and assesses about the quality and providing information for stakeholder for decision making. For system testing black box testing is used widely and boundary value analysis is one of the five test design technique in black box technique find the most crucial defect that lie on the edge of the equivalence partition . The purpose of boundary value analysis is to concentrate effort on error prone area by accurately pinpointing the boundaries of condition .BVA produces test inputs near each sub domain's to find failure cause by incorrect implementation of boundary. It is a challenge to minimize test case with appropriate boundary range or value for boundary value analysis. It takes time and thinking to write test case with appropriate values. In this paper we give a generic formula and developed a tool to generate different test cases automatically for boundary value analysis and minimize test case from as usual techniques .To show the validity of the generic formula and tool we used deferent case study and compare the result with proposed solution and tool .

Keywords

Black Box, equivalence class partitioning, boundary value analysis, test case, testing techniques

CHAPTER 1: INTRODUCTION

1.1 Background

For better product quality it is essential to do rigorous testing of software products (Source code and Documentation). Software Testing is expensive part of SDLC (Software Development Life Cycle), often consisting of approximately 40-60 % of total budgets [23]. Although Software testing is an expensive process but it happens that it fails to find many defects in system and as a result it causes failure. Developing effective and efficient technique has been a major problem when creating test cases. Software testing has always an interesting re-search area for both industry and academia. it is also true that the cost of testing increases dramatically while under testing would miss a lot of bugs . Therefore selecting and design efficient test cases are the most crucial task of software testing.

There are several well-known techniques associated with creating test cases for a system .There are different testing techniques which are applied in different phases of testing process e.g. black box and white box testing . Boundary value analysis is essential and popular Black box testing technique.

BVA, tests the behavior of a program at the boundaries. When checking a range of values, after selecting the set of data that lie in the valid partitions next is to check how the program behaves at the boundary values of the valid partitions. Boundary value analysis is most common when checking a range of numbers. For each range, there are two boundaries, the lower boundary (start of the range) and the upper boundary (end of the range) and the boundaries are the beginning and end of each valid partition. We should design test cases which exercises the program functionality at the boundaries, and with values just inside and just outside the boundaries. Here, the assumption is that if the program works correctly for these extreme cases, then it will work correctly for all values in between the valid partition. Testing has shown that defects that arise when checking a range of values the most defects are near or at the boundaries.

1.2 Research Questions

This study focus on software testing techniques specially on boundary value analysis that are used in industry. Boundary value analysis is most important task to validate the system correctness.

Question 1: Are there any Software Metric/Measurements to calculate BVA, like Cyclomatic Complexity (M=E-N+2P)?

From literature review we find some metric or tools or equation that will answer question 1. And here raises the question two.

Question 2: From existing solutions can we minimize the test cases for boundary value analysis?

We know that software testing is an expensive process. Sometimes it takes 40-60 % of overall budget [23]. In software testing most crucial part is effective test case design.

In this thesis we developed an equation and tools which will minimize very effectively the test cases for boundary value analysis for integer values.

1.2 Research Methodology:

This thesis reviews the existing software testing techniques and existing solution for boundary value analysis.

Literature review grounds the studies of the research questions. Firstly, the thesis discusses the software testing techniques. Secondly, focus on boundary value analysis and its existing solutions and propose a new equation and tools that will minimize test cases very effectively for boundary value analysis for integer values.

The resource includes books, journals, conference articles, and Internet retrieves.

The literature review was performed using Systematic review method which was adopted for searching different testing techniques from literature review. Over all 29 articles was found those are most relevant to our literature study. Among those articles 5 are grey literature articles and 4 are book related information. Over all we find 12 Journals and 8 conference papers. The database was taken in such a manner that it covers most of the journals as well as conference papers. All the articles are related to testing techniques. In 29 articles authors speak about testing techniques in one another manner. The majority of articles mainly focus on case studies, theoretical reports, literature study, experience reports and field studies.

1.3 Thesis Structure

This thesis is organized as follows:

In chapter 2, we briefly describe software testing techniques both dynamic and static. Testing types also describe briefly with definition, scope, opacity and specification.

In chapter 3, we describe Equivalence Class Partitioning (ECP) and Boundary Value Analysis (BVA) and relation between them. Current related work on boundary value analysis.

In chapter 4, we have done some case study on boundary value analysis and compare with our proposed solutions of test case minimization for boundary value analysis for integer values and floating point values and we also implement a tool that auto generates test cases for both integer and floating point values.

We discuss the summary and future work in Chapter 5.

Chapter 2: Software Testing in a Nutshell

2.1 What is Software Testing?

In recent years, software been introduced virtually everywhere. There will soon be no appliances, machines, or facilities for which control is not implemented by software or software parts. In automobiles, for example, microprocessors and their accompanying software control more and more functionality, from engine management to the transmission and brakes. Thus, software is crucial to the correct functioning of devices and industry. Likewise, the smooth operation of an enterprise or organization depends largely on the reliability of the software systems used for supporting the business processes and particular tasks. How fast an insurance company can introduce a new product, or even a new rate, most likely depends on how quickly the IT systems can be adjusted or extended. Within both embedded and commercial software systems, quality has become the most important factor in determining success [1]. If we want better quality of software products, then we have to do rigorous testing of systems and documentation. It can help to reduce the risk of problems occurring during operation and contribute to the quality of the software system, if the defects found are corrected before the system is released for operational use.

Software Testing is the process of executing a program or system with the intent of finding errors. And, it involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results. Software testing is an activity that should be done throughout the whole development process [3].

2.2 Why testing is necessary?

The economic importance of software

- The functioning the machines and equipment depends largely on software
- We cannot imagine large systems in telecommunication, finance or traffic control running without software.

Software quality

- More and more, the quality of software has become the determining factor for the success of technical or commercial systems and products

Testing for quality improvement

- Testing and reviewing insure the improvement of the quality of software products as well as the quality of software development process itself.

Failure example 1: Ariane 5 launch

Flight 501, which took place on June 4, 1996, was the first test flight of the Ariane 5 expendable launch system. It was not successful; the rocket tore itself apart 37 seconds after launch because of a malfunction in the in the control software, making the fault one of the most expensive computer bugs in history.

The Ariane 5 software reused the specifications the Ariane 4, but the Ariane 5's flight path was considerably different and beyond the range for which the reused code had been designed. Specifically, the Ariane 5's greater acceleration caused the back-up and primary inertial guidance computers to crash, after which the launcher's nozzles were directed spurious data. Pre-flight tests had never been performed on the re-alignment code under simulated Ariane 5 flight conditions, so the error was not discovered before launch [24].

Failure example 2: Lethal X-Rays

Because of a software failure a number of patients received a lethal dose dose a gamma rays:

Therac-25 was a radiation therapy machine produced by atomic Energy of Canada Limited. It was involved with at least six known accidents between 1985 and 1987, in which patients were in some cases on the order of hundreds of gray. At least five patients died of the overdoses. These accidents highlighted the dangers of software control of safety-critical system [25].

2.3 Types of Software Testing

Basically software testing is two types. Static testing (analysis) and Dynamic testing. Both dynamic testing and static testing has same objective, finding defects. But the testing process is difference. Unlike dynamic testing, which requires the execution of the software, static testing rely on manual examination and automated analysis of code and software models. Also dynamic testing can identify some types of defects that could not find static and vice versa.

2.3.1 Static Testing

In static testing the test object is not provided with test data and executed but rather analyzed.[2] Static analysis or testing can be done in two ways. One is review and another is static analysis by tools.

2.3.1.1 Review

Reviews are a way of testing software work products (including code) and can be performed well before dynamic test execution [2]. A review could be done entirely as a manual activity, but there is also tool support. The main manual activity is to examine a work product and make comments about it. Any software work product can be reviewed, including requirements specifications, design specifications, code, test plans, test specifications, test cases, test scripts, user guides or web pages [2].

Benefits of reviews include early defect detection and correction, development productivity improvements, reduced development timescales, reduced testing cost and time, lifetime cost reductions, fewer defects and improved communication. Reviews can find omissions, for example in requirements, which are unlikely to be found in dynamic testing.

Another great benefit of review is defects detected during reviews early in the life cycle (e.g., defects found in requirements) are often much cheaper to remove than those detected by running tests on the executing code.

Typical defects that are easier to find in reviews than in dynamic testing include: deviations from standards, requirement defects, design defects, insufficient maintainability and incorrect interface specifications.

Generally there are two types of review formal and informal. But by the characteristics of review process and meeting we can classify 4 types of review.

i) Formal review (Inspection)

ii) Informal review

iii) Walkthrough

iv) Technical review

2.3.1.2 Static Analysis by tools

The goal of static analysis is like reviews, to reveal defects or defect-prone parts in a document. In static analysis, tools do the analysis. Static analysis can find defects in software source code and software models. Static analysis is performed without actually executing the software being examined by the tool.

Static analysis can locate defects that are hard to find in dynamic testing. As with reviews, static analysis finds defects rather than failures. Static analysis tools analyze program code (e.g., control flow and data flow), as well as generated output such as HTML and XML. [2]

The following defects and dangerous constructions can be detected by static analysis:

- Syntax violations
- Deviations from conventions and standards
- Control flow anomalies
- Data flow anomalies
- Typical defects discovered by static analysis tools include:
- Referencing a variable with an undefined value
- Inconsistent interfaces between modules and components
- Variables that are not used or are improperly declared
- Unreachable (dead) code
- Missing and erroneous logic (potentially infinite loops)
- Overly complicated constructs
- Programming standards violations
- Security vulnerabilities
- Syntax violations of code and software models

Static analysis tools are typically used by developers (checking against predefined rules or programming standards) before and during component and integration testing or when checking-in code to configuration management tools, and by designers during software modeling. Static analysis tools may produce a large number of warning messages, which need to be well-managed to allow the most effective use of the tool [2].

2.3.2 Dynamic Testing

Dynamic testing is also known as dynamic analysis. This technique is used to test the dynamic behavior of the code.

Dynamic testing is a method of assessing the feasibility of a software program by giving input and examining output (I/O). The dynamic method requires that the code be compiled and run.

Dynamic test design techniques are divided into three categories:

i) Black box testing

ii) White box testing

iii) Experience-based testing.

2.3.2.1 Black box testing

In black box testing, the test object is seen as a black box. Test cases are derived from the specification of the test object; information about the inner structure is not necessary or available. The test cases are derived from requirements specification and that's why it's also called specification based testing.

Black-box test design techniques) are a way to derive and select test conditions, test cases, or test data based on an analysis of the test basis documentation. This includes both functional and non-functional testing.

Black box testing, also called functional testing and behavioral testing, focuses on determining whether or not a program does what it is supposed to do based on its functional requirements. Black box testing attempts to find errors in the external behavior of the code in the following categories [5]: (1) incorrect or missing functionality; (2) interface errors; (3) errors in data structures used by interfaces; (4) behavior or performance errors; and (5) initialization and termination errors

Common characteristics of specification-based test design techniques include:

i) Models, either formal or informal, are used for the specification of the problem to be solved, the software or its components

ii) Test cases can be derived systematically from these models

Advantages:

The main advantage of black box testing is that, testers no need to have knowledge on specific programming language, not only programming language but also knowledge on implementation. In black box testing both programmers and testers are independent of each other. Another advantage is that testing is done from user's point of view. The significant advantage of black box testing is that it helps to expose any ambiguities or inconsistencies in the requirements specifications [6].

Black box testing techniques are:

I. Equivalence Class Partitioning
II. Boundary Value Analysis
III. Decision Tables
IV. State Transition Diagrams (or) State Transition Diagrams
V. Use Case Testing

2.3.2.1.1 Equivalence Class Partitioning

Equivalence class partitioning testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behavior [7]. The partitioning is done such that the program behaves in a similar way to every input value belonging to an equivalence class [6].

2.3.2.1.2 Boundary value Analysis

Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the **'boundaries'** of equivalence classes. Typically programming errors occur at the boundaries of equivalence classes are known as "**Boundary Value Analysis"** [8].

2.3.2.1.3 Decision Table

Decision tables are human readable rules used to express the test experts or design experts knowledge in a compact form [9]. Decision Tables can be used when the outcome or the logic involved in the program is based on a set of decisions and rules which need to be followed. Decision table mainly consists of four areas called the condition stub, the condition entry, the action stub and finally action entry [6] [10].

TABLE 1: Blank decision table

Conditions	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Repayment money has been mentioned				
Terms of loan has been mentioned				

Next, recognize all of the combinations in "Yes" and "No" (In Table 2). In each column of two conditions mention "Yes" or "No", user will get here four combinations (two to the power of the number of things to be combined). Note, if user has three things to combine, they will have eight combinations, with four things, there are 16, etc. Because of this, it's always good to take small sets of combinations at once. To keep track on combinations, give alternate "Yes" and "No" on the bottom row, put two "Yes" and then two "No" on the row above the bottom row, etc., so the top row will have all "Yes" and then all "No" (Apply the same principle to all such tables) [20].

TABLE 2: Decision table – Input combination

Conditions	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Repayment money has been mentioned	**Y**	**Y**	**N**	**N**
Terms of loan has been mentioned	**Y**	**N**	**Y**	**N**

In the next step, recognize the exact outcome for each combination (In Table 3). In this example, user can enter one or both of the two fields. Each combination is sometimes referred to as a step.

TABLE 3: Decision table – Combinations and outcomes

Conditions	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Repayment money has been mentioned	**Y**	**Y**	**N**	**N**
Terms of loan has been mentioned	**Y**	**N**	**Y**	**N**
Actions/Outcomes				
Process loan money	**Y**	**Y**		

Process term	**Y**		**Y**	

At this time you didn’t think that what will happen when customer don’t enter anything in either of the two fields. The table has shown a combination that was not given in the specification for this example. This combination can result as an error message, so it is necessary to add another action (In Table 4). This will flash the strength this method to find out omissions and ambiguities in specifications [20].

TABLE 4: Decision table – Additional outcomes

Conditions	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Repayment money has been mentioned	**Y**	**Y**	**N**	**N**
Terms of loan has been mentioned	**Y**	**N**	**Y**	**N**
Actions/Outcomes				
Process loan money	**Y**	**Y**		
Process term	**Y**		**Y**	
Error message				**Y**

We will provide you some other example that allows the customer to enter both repayment and term. This will change the outcome of our table, this will generate an error message if both are entered (Shown in Table 5).

TABLE 5: Decision table – Changed outcomes

Conditions	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Repayment money has been mentioned	**Y**	**Y**	**N**	**N**
Terms of loan has been mentioned	**Y**	**N**	**Y**	**N**
Actions/Outcomes				
Process loan money		**Y**		
Process term			**Y**	
Error message	**Y**			**Y**

The final process of this method is to write test cases to use each of the four steps in the table.

2.3.2.1.4 State Transition Testing

State Graph is an excellent tool to capture certain types of system requirements and document internal system design. When a system must remember what happened before or when valid and invalid orders of operation exists, and then state transition testing could be used. This state graphs are used when system moves from one state to another state. State graphs are represented with symbols, circle is used to represent state, arrows are used to represent transition, and event is represented by label on the transition [6][11].

For example, if you request to withdraw $100 from a bank ATM, you may be given cash. Later you may make exactly the same request but it may refuse to give you the money because of your insufficient balance. This later refusal is because the state of your bank account has changed from having sufficient funds to cover the withdrawal to having insufficient funds. The transaction that caused your account to change its state was probably the earlier withdrawal. A state diagram can represent a model from the point of view of the system, the account or the customer.

Let us consider another example of a word processor. If a document is open, you are able to close it. If no document is open, then 'Close' is not available. After you choose 'Close' once, you cannot choose it again for the same document unless you open that document. A document thus has two states: open and closed.

We will look first at test cases that execute valid state transitions.
Figure 1 below, shows an example of entering a Personal Identity Number (PIN) to a bank account. The states are shown as circles, the transitions as lines with arrows and the events as the text near the transitions. (We have not shown the actions explicitly on this diagram, but they would be a message to the customer saying things such as 'Please enter your PIN'.)

The state diagram shows seven states but only four possible events (Card inserted, Enter PIN, PIN OK and PIN not OK). We have not specified all of the possible transitions here – there would also be a time-out from 'wait for PIN' and from the three tries which would go back to the start state after the time had elapsed and would probably eject the card. There would also be a transition from the 'eat card' state back to the start state. We have not specified all the possible events either – there would be a 'cancel' option from 'wait for PIN' and from the three tries, which would also go back to the start state and eject the card.

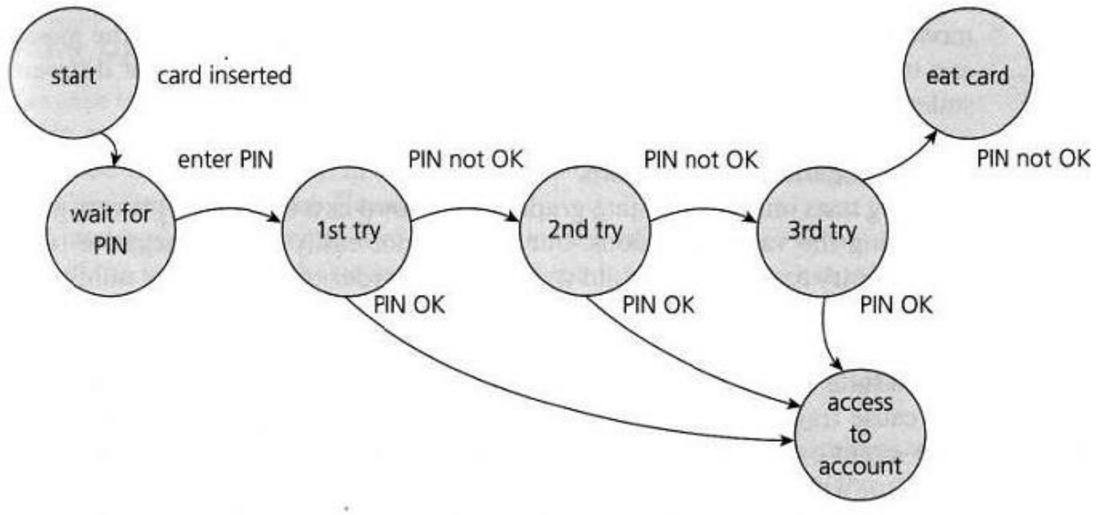


Figure 1: State Transition of Personal Identity Number (pin) to a bank account.

In deriving test cases, we may start with a typical scenario.

- First test case here would be the normal situation, where the correct PIN is entered the first time.
- A second test (to visit every state) would be to enter an incorrect PIN each time, so that the system eats the card.
- A third test we can do where the PIN was incorrect the first time but OK the second time, and another test where the PIN was correct on the third try. These tests are probably less important than the first two.
- Note that a transition does not need to change to a different state (although all of the transitions shown above do go to a different state). So there could be a transition from ‘access account’ which just goes back to ‘access account’ for an action such as ‘request balance’.

Test conditions can be derived from the state graph in various ways. Each state can be noted as a test condition, as can each transition. However this state diagram, even though it is incomplete, still gives us information on which to design some useful tests and to explain the state transition technique.

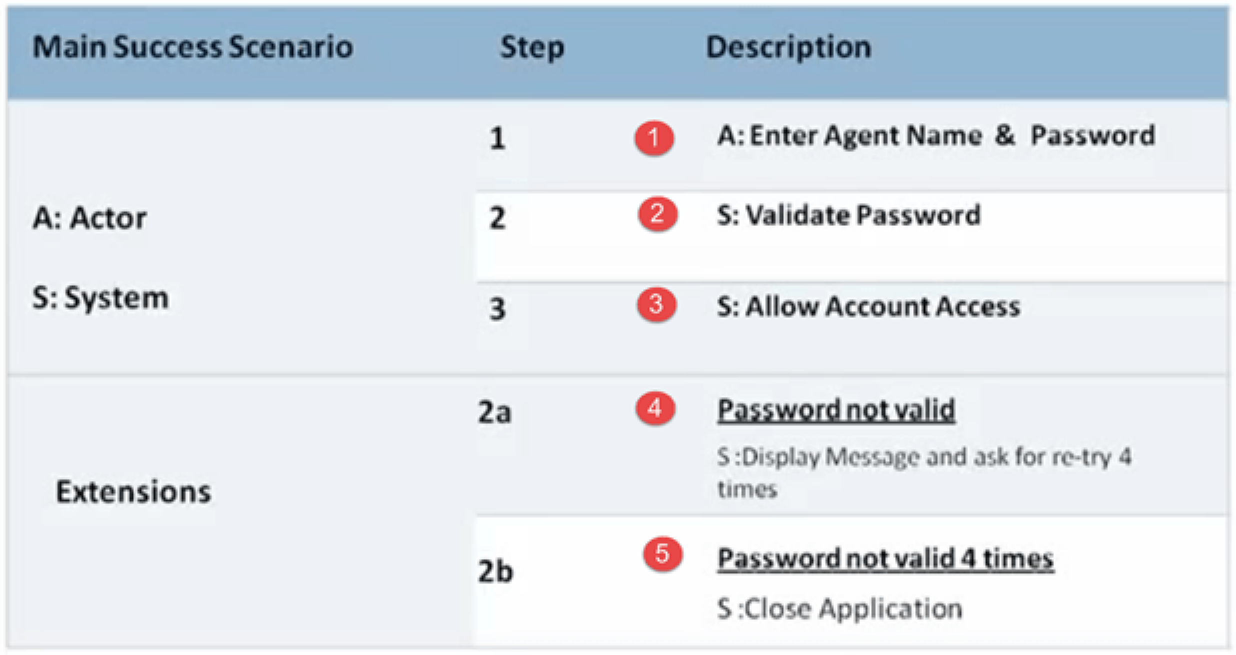
We need to be able to identify the coverage of a set of tests in terms of transitions. We can also consider transition pairs and triples and so on. Coverage of all individual transitions is also known as 0-switch coverage, coverage of transition pairs is l-switch coverage, coverage of transition triples is 2-switch coverage, etc. Deriving test cases from the state transition model is a black-box approach. Measuring how much we have tested (covered) will discuss in a white-box perspective. However, state transition testing is regarded as a black-box technique.

2.3.2.1.5 Use Case Testing

In use case testing tests can be derived from use cases. A use case describes interactions between actors (users or systems), which produce a result of value to a system user or the customer. Use cases may be described at the abstract level like business use case, technology-free, business process level or at the system level. Each use case has preconditions which need to be met for the use case to work successfully. Each use case terminates with post-conditions which are the observable results and final state of the system after the use case has been completed. A use case usually has a mainstream scenario and alternative scenarios. Use cases describe the “process flows” through a system based on its actual likely use, so the test cases derived from use cases are most useful in uncovering defects in the process flows during real-world use of the system. Use cases are very useful for designing acceptance tests with customer/user participation. They also help uncover integration defects caused by the interaction and interference of different components, which individual component testing would not see. Designing test cases from use cases may be combined with other specification-based test techniques [2].

The Use-Case Testing Technique helps identifying test cases that cover the entire system, on a transaction by transaction basis from the start to finish.

In a use-case, an actor is represented by "A" and system by "S"**.** First we list the Main Success Scenario.



Main Success Scenario	Step		Description
A: Actor S: System	1	1	A: Enter Agent Name & Password
	2	2	S: Validate Password
	3	3	S: Allow Account Access
Extensions	2a	4	Password not valid S :Display Message and ask for re-try 4 times
	2b	5	Password not valid 4 times S :Close Application

Figure 2: Use Case Testing

2.3.2.2 White box testing

White box testing tests the structure or internal mechanism of a component or a system. Tester must have good programming skill and knowledge. White-box testing is also known as structural testing, clear box testing, and glass box testing.

It involves testing of all logic of program, testing of loops, condition testing and data flow based testing. This helps in detecting errors even with unclear and incomplete software specification. The objective of white box testing is to ensure that the test cases exercise each path through a program [5].The test cases also ensure that all independent paths within the program have been executed at least once. All internal data structures are exercised to ensure validity. All loops are executed at their boundaries and within operational bounds [12]

White box techniques are

I. Statement testing
II. Decision testing or branch testing
III. Condition coverage Testing
IV. Path coverage testing
V. Loop coverage testing

2.3.2.2.1 Statement testing

Statement testing focuses on each statement of the test object. The test cases shall execute a predefined minimum quota or even all statements of the test object [1]. In component testing, statement coverage is the assessment of the percentage of executable statements that have been exercised by a test case suite. The statement testing technique derives test cases to execute specific statements, normally to increase statement coverage.[2] Statement coverage is determined by the number of executable statements covered by (designed or executed) test cases divided by the number of all executable statements in the code under test[2].

To understand the statement coverage in a better way let us take an example which is basically a pseudo-code.

Consider code sample 1:

```
READ X
READ Y
I F X>Y THEN Z = 0
ENDIF
```

To achieve 100% statement coverage of this code segment just one test case is required, one which ensures that variable A contains a value that is greater than the value of variable Y, for example, X = 12 and Y = 10. Note that here we are doing structural test *design* first; since we are choosing our input values in order ensure statement coverage.

Now, let’s take another example where we will measure the coverage first. In order to simplify the example, we will regard each line as a statement. A statement may be on a single line, or it may be spread over several lines. One line may contain more than one statement, just one statement, or only part of a statement. Some statements can contain other statements inside them. In code sample 2, we have two read statements, one assignment statement, and then one IF statement on three lines, but the IF statement contains another statement (print) as part of it.

```
1 READ X
2 READ Y
3 Z =X + 2*Y
4 IF Z> 50 THEN
5 PRINT large Z
6 ENDIF
```

Code sample 2

Although it isn't completely correct, we have numbered each line and will regard each line as a statement. Let's analyze the coverage of a set of tests on our six-statement program:

TEST SET 1
Test 1_1: X= 2, Y = 3
Test 1_2: X =0, Y = 25
Test 1_3: X =47, Y = 1

Which statements have we covered?

- In Test 1_1, the value of Z will be 8, so we will cover the statements on lines 1 to 4 and line 6.
- In Test 1_2, the value of Z will be 50, so we will cover exactly the same statements as Test 1_1.
- In Test 1_3, the value of Z will be 49, so again we will cover the same statements.

Since we have covered five out of six statements, we have 83% statement coverage (with three tests). What test would we need in order to cover statement 5, the one statement that we haven't exercised yet? How about this one:

Test 1_4: X = 20, Y = 25

This time the value of Z is 70, so we will print 'Large Z' and we will have exercised all six of the statements, so now statement coverage = 100%. Notice that we measured coverage first, and then designed a test to cover the statement that we had not yet covered.

Note that Test 1_4 on its own is more effective which helps in achieving 100% statement coverage, than the first three tests together. Just taking Test 1_4 on its own is also more efficient than the set of four tests, since it has used only one test instead of four.

2.3.2.2.2 Decision testing or branch testing

Decision coverage or branch testing is the assessment of the percentage of decision outcomes that have been exercised by a test case suite [2]. The decision testing technique derives test cases to execute specific decision outcomes. Branches originate from decision points in the code and show the transfer of control to different locations in the code [2].

For Example, whenever there are two or more possible exits from the statement like an IF statement, a DO-WHILE or a CASE statement it is known as decision because in all these statements there are two outcomes, either TRUE or FALSE.

With the loop control statement like DO-WHILE or IF statement the outcome is either TRUE or FALSE and decision coverage ensures that each outcome (i.e. TRUE and FALSE) of control statement has been executed at least once.

Alternatively you can say that control statement IF has been evaluated both to TRUE and FALSE.

The formula to calculate decision coverage is:

Decision Coverage = (Number of decision outcomes executed/Total number of decision outcomes)*100%
Research in the industries has shown that even if through functional testing has been done it only achieves 40% to 60% decision coverage.

Decision coverage is stronger that statement coverage and it requires more test cases to achieve 100% decision coverage.

Let us take one example to explain decision coverage:

```
READ X
READ Y
IF “X > Y”
PRINT X is greater that Y
ENDIF
```

To get 100% statement coverage only one test case is sufficient for this pseudo-code.

TEST CASE 1: X=10 Y=5
However this test case won’t give you 100% decision coverage as the FALSE condition of the IF statement is not exercised.

In order to achieve 100% decision coverage we need to exercise the FALSE condition of the IF statement which will be covered when X is less than Y.

So the final TEST SET for 100% decision coverage will be:

TEST CASE 1: X=10, Y=5
TEST CASE 2: X=2, Y=10

2.3.2.2.3 Condition coverage testing

In condition coverage testing each simple condition be evaluated as true and false at least once and each branch is taken at least once. That is, every statement and both alternatives for every simple condition will have been executed [13]

A criterion that is sometimes stronger than decision coverage is condition coverage. In this case, one writes enough test cases such that each condition in a decision takes on all possible outcomes at least once. Since, as with decision coverage, this does not always lead to the execution of each statement, an addition to the criterion is that each point of entry to the program or subroutine, as well as ON-units, be invoked at least once. For instance, the branching statement

```
DO K=0 TO 50 WHILE (J+K<QUEST)
```

Contains two conditions: is K less than or equal to 50, and is J + K less than QUEST? Hence test cases would be required for the situations K = 50, K > 50 (i.e., reaching the last iteration of the loop), J + K < QUEST, and J + K = QUEST .

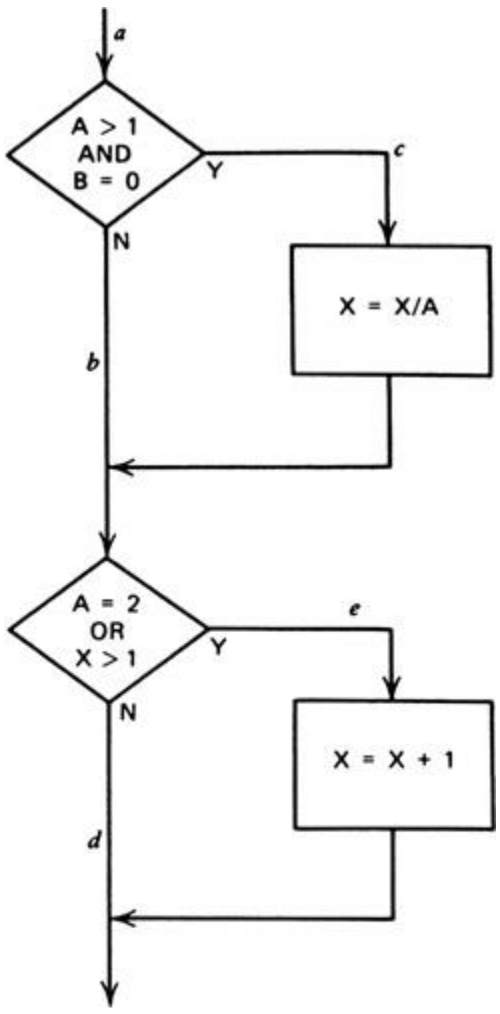


Figure 3: A small program to be tested.

Figure 3 has four conditions: A > 1, B = 0, A = 2, and X > 1. Hence enough test cases are needed to force the situations where A > 1, A=1, B = 0, and B ? 0 are present at point a and where A = 2, A ? 2, X > 1, and X = 1 at point b. A sufficient number of test cases satisfying the criterion, and the paths traversed by each, are

A=2, B=0, X=4 *ace*
A=1, B=1, X=1 *adb*

Note that although the same numbers of test cases were generated for this example, condition coverage is usually superior to decision coverage in that it may (but does not always) cause every individual condition in a decision to be executed with both outcomes, where decision coverage does not. For instance, the DO statement

DO K=0 TO 50 WHILE (J+K<QUEST)

is a two-way branch (execute the loop body or skip it). If one is using decision testing, the criterion can be satisfied by letting the loop run from K = 0 to 51, without ever exploring the circumstance where the WHILE clause becomes false. With the condition criterion, however, a test case would be needed that generated a false outcome for the condition J + K < QUEST.

Although the condition-coverage criterion appears, at first glance, to satisfy the decision-coverage criterion, it does not always do so. If the decision IF (A&B) is being tested, the condition-coverage criterion would allow one to write two test cases A is true, B is false, and A is false, B is true but this would not cause the THEN clause of the IF to execute. The condition-coverage tests for the earlier example covered all decision outcomes, but this was only by chance. For instance, two alternative test cases

A = 1, B = 0, X = 3
A = 2, B = 1, X = 1

Cover all condition outcomes, but they cover only two of the four decision outcomes (they both cover path abe and hence do not exercise the true outcome of the first decision and the false outcome of the second decision).

2.3.2.2.4 Path Coverage

In path coverage testing every distinct entry-exit path through the code is executed at least once in some test. Path coverage requires considering every combination of every branch through the code, including implicit paths through every simple condition [13]

Path coverage refers to designing test cases such that all linearly independent paths in the program are executed at least once. A linearly independent path can be defined in terms of what's called a control flow graph of an application [22].

Great. So what exactly is a control flow graph?

This is simply a visual way to describes the sequence in which the different statements of an application get executed. In other words and perhaps not too surprisingly given the name, a control flow graph describes how the control *flows* through the application [22]. The control flow graph for any program (no matter how small or large) can be easily drawn if you know how to represent various language statements or structures. To show that this is not all some academic theory, here are some examples of control flow graphs for very common constructs that you'll find in most programming languages:

IF construct

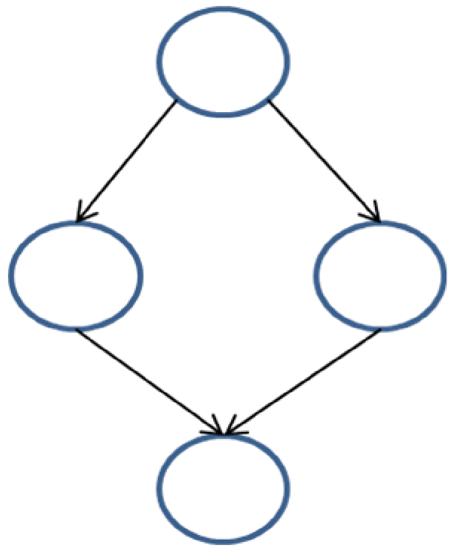


Figure 4: Path Coverage IF Construct

UNTIL construct

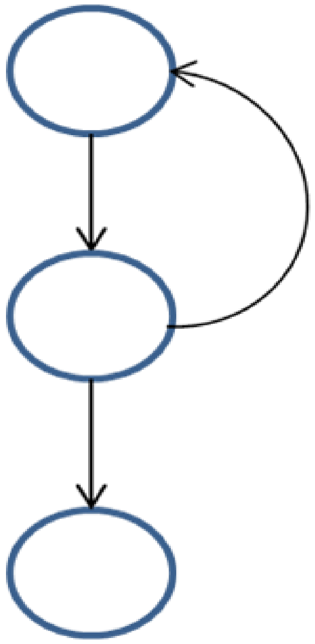


Figure 5: Path Coverage UNTIL Construct

WHILE construct

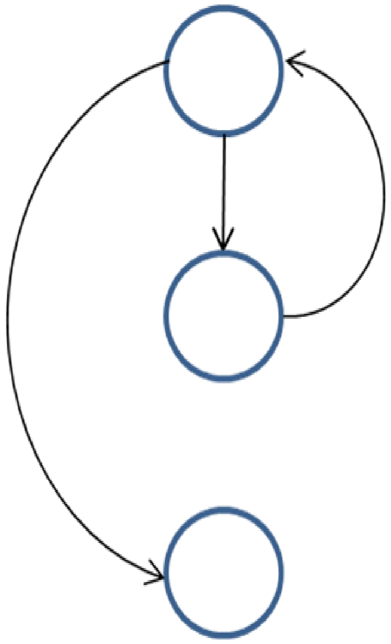


Figure 6: Path Coverage WHILE Construct

From a developer perspective, in order to draw the control flow graph, all the statements of a particular part of the application must be numbered. The different numbered statements serve as the *nodes* of the control flow graph. An *edge* from one node to another node exists if the execution of the statement representing the first node can result in the transfer of control to the other node. (Check out my previous post where I showed graphical representations of nodes and edges.) we'll give some actual examples of this in a little bit.

Combined with what I just said and simply by looking at the above (admittedly sparse) examples of graphs, you can see that there are a few ways to consider coverage from a testing standpoint.

- **Node Coverage:** Achieved when the paths identified have a test that goes to every node in the graph.
- **Link Coverage:** Achieved when the paths identified have a test that goes along every link, or line, in the graph. In many cases, node coverage will take care of this.
- **Loop Coverage:** Achieved when the numerous paths identified have tests that explores the interaction between sub-paths within a loop.

Let's consider this with a specific example. Here's a control flow graph:

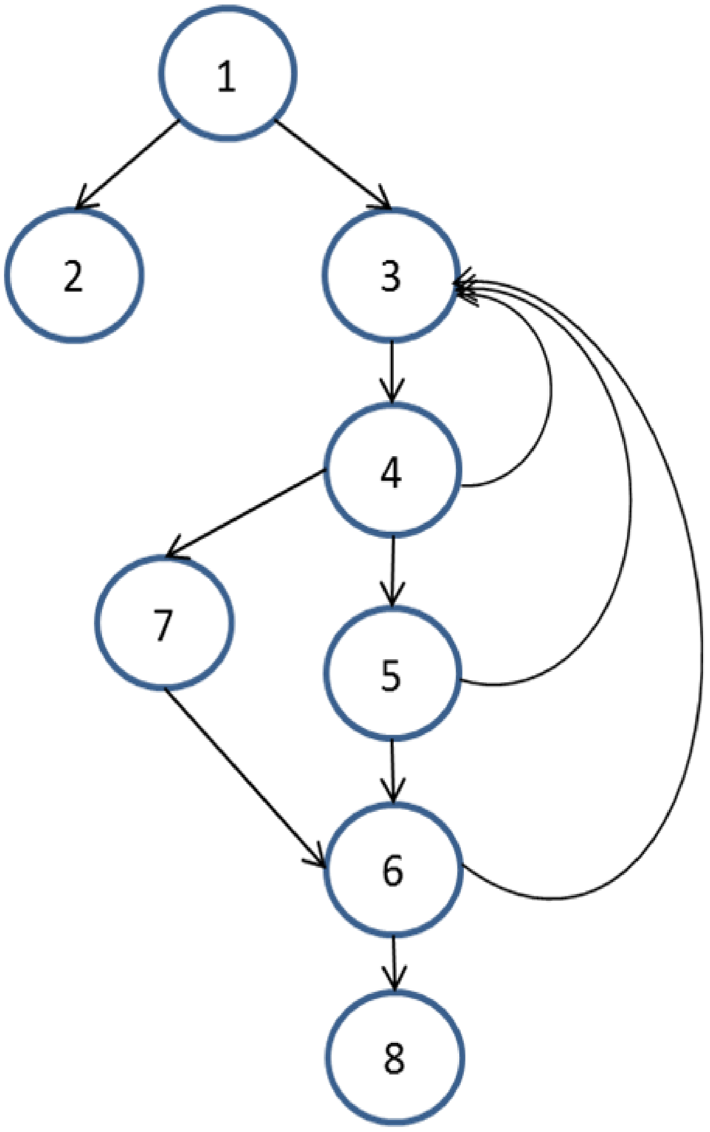


Figure 7: Path Loop Coverage

Going with the above testing coverage ideas, this table lists the test paths could consider through the program flow.

Table 6: **Path Loop Coverage**

Node Coverage	**Link Coverage**	**Loop Coverage**
1 2	1 3 4 3 4 5 3 4 5 6 8	1 3 |4 3| 5 7 8
1 3 4 7 6 8		1 3 |4 3| 7 6 8
1 3 4 5 6 8		1 3 4 |5 3| 4 5 6 8
		1 3 4 |5 3| 7 6 8
		1 3 4 5 |6 3| 4 5 6 8
		1 3 4 5 |6 3| 4 7 6 8
		1 3 4 7 |6 3| 4 5 6 8

In the Loop Coverage column, I used pipe characters to show where a specific loop was being entered. As an exercise for you to think about, consider that you could replace the third node test with the link test. That would mean you only needed two node coverage paths.

Realistically, it’s important to note that just using this graph, you really don’t know the data conditions that would differentiate the 4 –> 3 link from the 4 –> 5 link or the 4 –> 7 link. Those may require more tests based on the specific conditions. This is an important caveat and it’s one that plaques unit testers everywhere: just because you have coverage, doesn’t mean you have effective coverage [22].

2.3.2.3 Experience-based Techniques

In experience-based testing tests are derived from the tester's skill and intuition and their experience with similar applications and technologies. When used to augment systematic techniques, these techniques can be useful in identifying special tests not easily captured by formal techniques, especially when applied after more formal approaches [2].

Commonly used tow experienced-based techniques are error guessing and exploratory testing.

2.3.2.3.1 Error Guessing

Generally testers anticipate defects based on experience [2] A structured approach to the error guessing technique is to enumerate a list of possible defects and to design tests that attack these defects. This systematic approach is called fault attack. These defect and failure lists can be built based on experience, available defect and failure data, and from common knowledge about why software fails [2].

2.3.2.3.2 Exploratory Testing

Exploratory testing is concurrent test design, test execution, test logging and learning, based on a test charter containing test objectives, and carried out within time-boxes [2]. It is an approach that is most useful where there are few or inadequate specifications and severe time pressure, or in order to augment or complement other, more formal testing [14]. It can serve as a check on the test process, to help ensure that the most serious defects are found [2].

2.4 The Testing Spectrum

Software testing is involved in each stage of software life cycle, but the way of testing conducted at each stage of software development is different in nature and it has different objectives.

Unit testing: is a code based testing which is performed by developers, this testing is mainly done to test each and individual units separately. This unit testing can be done for small units of code or generally no larger than a class. [13].

Integration testing: validates that two or more units or other integrations work together properly, and inclines to focus on the interfaces specified in low-level design [13].

System testing: reveals that the system works end-to-end in a production-like location to provide the business functions specified in the high-level design [13].

Acceptance testing: is conducted by business owners, the purpose of acceptance testing is to test whether the system does in fact, meet their business requirements [13].

Regression Testing: is the testing of software after changes has been made; this testing is done to make sure that the reliability of each software release, testing after changes has been made to ensure that changes did not introduce any new errors into the system [13].

Alpha Testing: usually in the existence of the developer at the developer's site will be done.

Beta Testing: done at the customer's site with no developer in site.

Functional Testing: done for a finished application; this testing is to verify that it provides all of the behaviors required of it [13].

Table 7: Six levels of testing

Test type	**Opacity**	**Who will do the testing**	**Specification**	**Scope**
Unit	White box	Programmer who write the code	Low level Code structure	Separately testable component, no larger than a class
Integration	Black box White box	Programmer who write the code	Low-Level Design High-Level Design	Multiple classes
System	Black box	Independent Testers will Test	Requirements Analysis phase	For Entire product in representative environments
Acceptance(Alpha)	Black box	Customers Side (Developer's site)	Requirements Analysis Phase	Entire product at developer's site.
Beta	Black box	Customers Side	Client Adhoc Request	Entire product in customer's environment
Regression	Black & White Box Testing	Generally Programmers or independent Testers	Modification Documentation High-Level Design	This can be for any of the above

CHAPTER 3: EQUIVALENCE CLASS PARTITIONING AND BOUNDARY VALUE ANALYSIS

3.1 Equivalence Class Partitioning

Equivalence class testing is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behavior. Test cases are designed to test the input or output domain partitions. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage **[4].** The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed [l]. Jorgensen [5] identified one problem with equivalence partitioning. Often a specification does not define the output for an invalid equivalence class. Tucker [6] also noted that problems occur when the test data chosen for an equivalence class does not represent that partition in terms of the behavior of the program function that is being tested. Hamlet and Taylor [7] state that ‘‘Partition testing can be no better than the information that defines its subdomains.” If one input in an invalid equivalence class causes a failure in the program then all other inputs in that class must also cause a failure. If this is not the case then the equivalence class is not a good representative of that part of the program and thus the identification of additional partitions may be required. Due to the nature of this approach such problems may not be identified.

3.2 Boundary Value Analysis

Boundary value analysis is performed by designing tests case that exercise the edges of the input and output classes identified in the specification [6]. Test cases can be derived from the ‘boundaries’ of equivalence classes. Usually, when testers write test case on boundary values they include above, below and on the boundary of the class. One disadvantage with boundary value analysis is that it is not as systematic as other prescriptive testing techniques [13]. This is due to the fact that it requires the tester to identify the most extreme values inputs can take.

Jorgensen noted that it is this type of abstract thinking that may allow a tester to improve the quality of the test sets used [5].

3.3 Relation between ECP and BVA

In industry or practically due to time and budget consideration, it is not possible or feasible to test everything, combination of all input and precondition (Exhaustive Testing) for each set of test data, especially when there is a large pool test cases. For efficiency we need an easy way or special techniques that can select test cases intelligent from the pool of test-case, such that all test scenarios are covered. For that's why in black box technique we use **-**Equivalence Partitioning & Boundary Value Analysis testing techniques [20]**.** Equivalence Partitioning and Boundary value analysis are linked to each other and can be used together at all levels of testing. In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way. Equivalence partitions (or classes) can be found for both valid and invalid data and boundary value analysis test the boundary of equivalence partitions.

3.4 Equivalence Partitioning with Example:

It is a black box technique which can be applied to all levels of testing like unit, integration, system etc. In this technique, we divide set of test condition into partition that can be considered the same.

For example: Let's consider the behavior of tickets in the Flight reservation application, while booking a new flight.

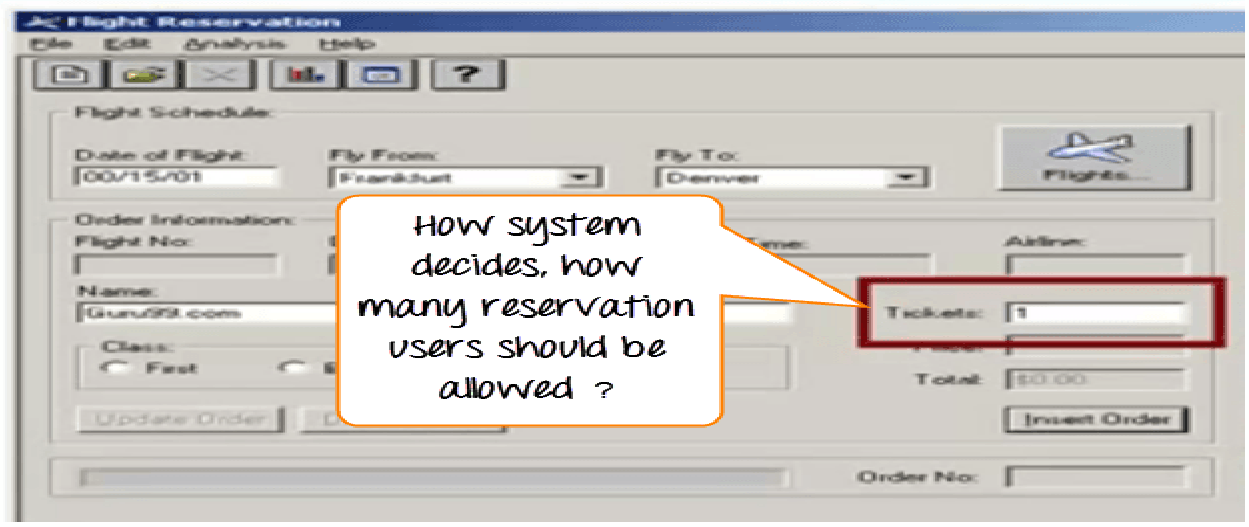


Figure 8: Equivalence Class Partitioning [21]

Ticket values 1 to 10 are considered valid & ticket is booked. While value 11 to 99 is considered invalid for reservation and error message will appear, **"Only ten tickets may be ordered at one time".**

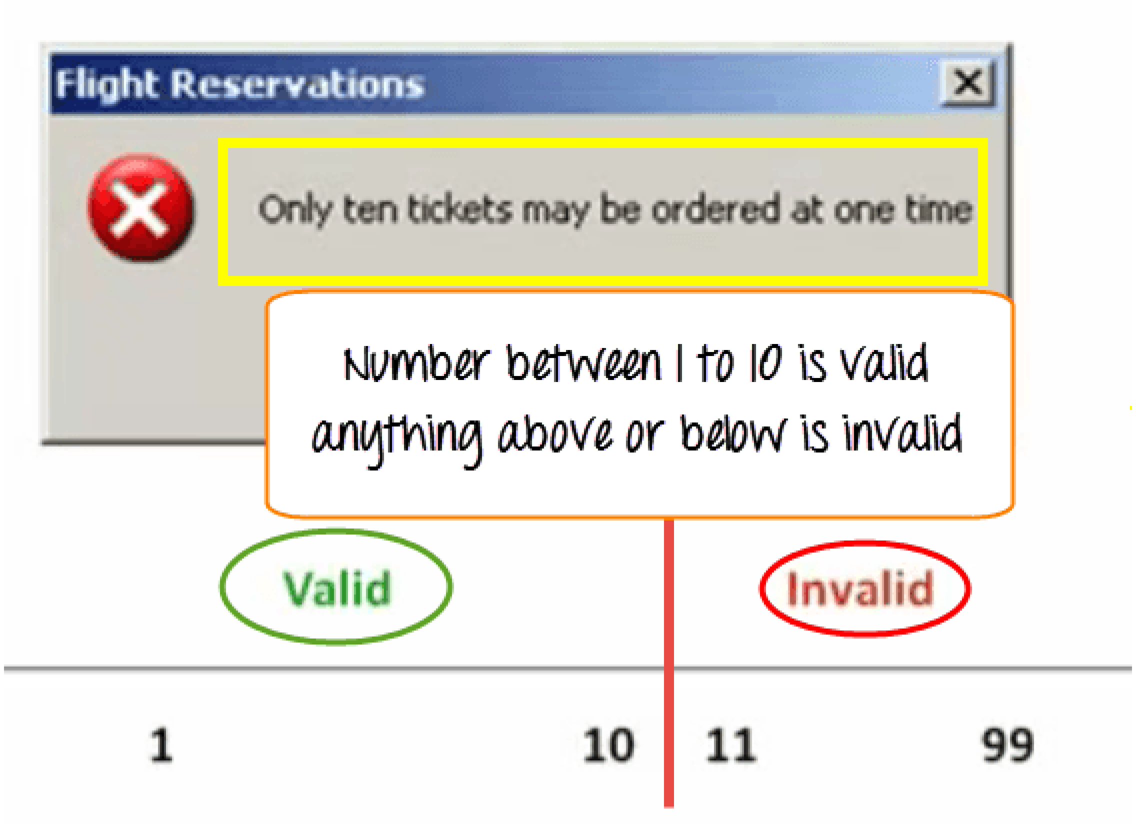

Flight Reservations
Only ten tickets may be ordered at one time
Number between 1 to 10 is valid anything above or below is invalid
Valid
Invalid
1
10
11
99

Figure 9: Equivalence Class Partitioning [21]

Here is the test condition

1. Any Number greater than 10 entered in the reservation column (let say 11) is considered invalid.
2. Any Number less than 1 that is 0 or below, then it is considered invalid.
3. Numbers 1 to 10 are considered valid
4. Any 3 Digit Number say -100 is invalid.

We cannot test all the possible values because if done, number of test cases will be more than 100.To address this problem, we use equivalence partitioning hypothesis where we divide the possible values of tickets into groups or sets as shown below where the system behavior can be considered the same.

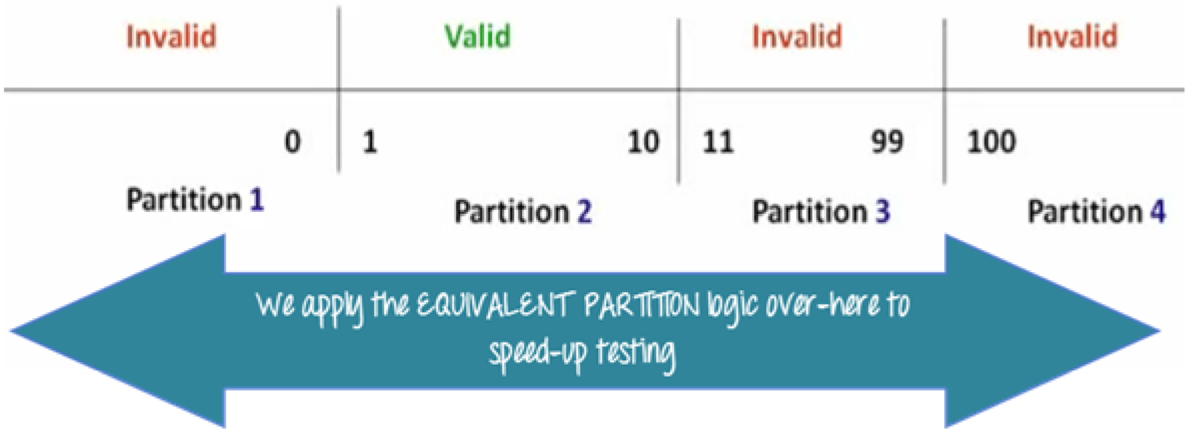


Figure 10: Equivalence Class Partitioning [21]

The divided sets are called Equivalence Partitions or Equivalence Classes. Then we pick only one value from each partition for testing. The hypothesis behind this technique is that if one condition/value in a partition passes all others will also pass. Likewise, if one condition in a partition fails, all other conditions in that partition will fail.

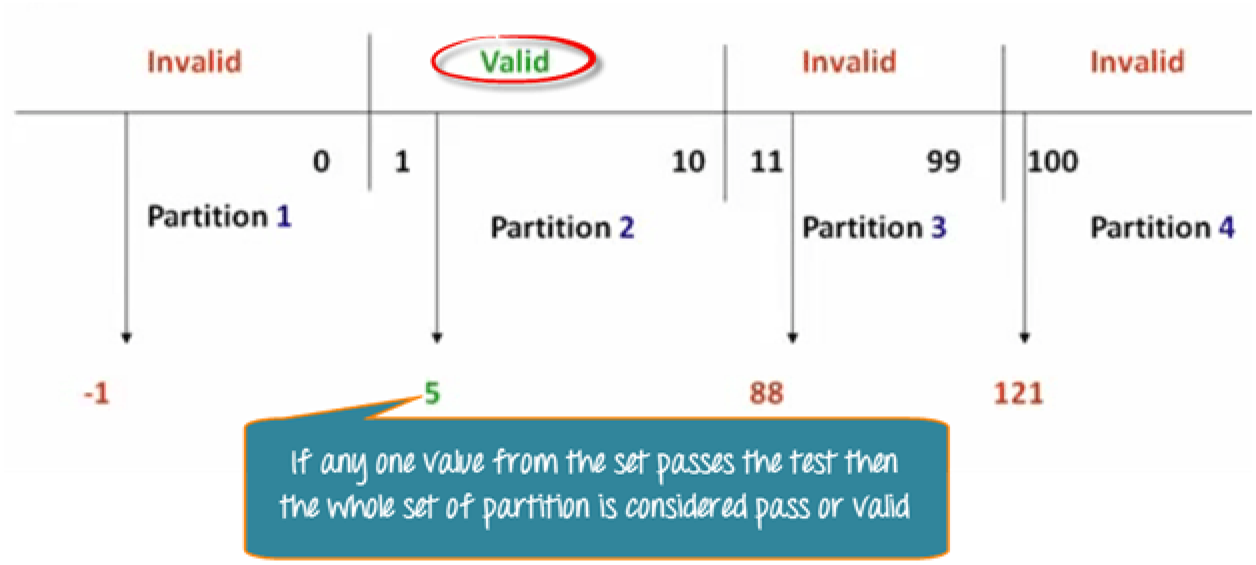

Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
-1
5
88
121
If any one value from the set passes the test then the whole set of partition is considered pass or valid

Figure 11: Equivalence Class Partitioning [21]

3.5 Boundary Value Analysis with Example

In Boundary Value Analysis, you test boundaries between equivalence partitions. Boundary value analysis delivers a very reasonable addition to the test cases that have been identified by equivalence class partitioning [21]. Faults often appear at the boundaries of equivalence classes. This happens because boundaries are often not defined clearly or are misunderstood. A test with boundary values usually discovers failures. The technique can be applied only if the set of data in one equivalence class is ordered and has identifiable boundaries [21].

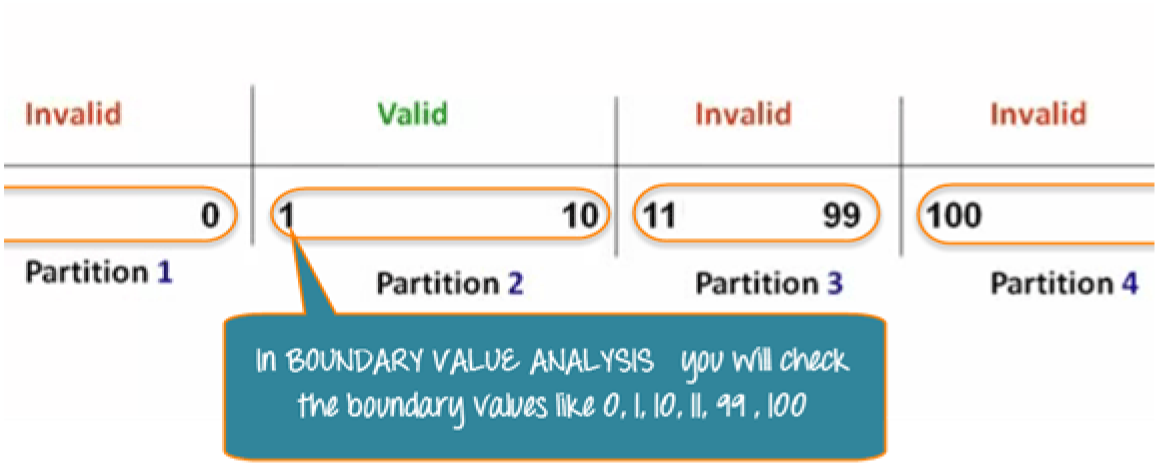

Invalid
Valid
Invalid
Invalid
0
1
10
11
99
100
Partition 1
Partition 2
Partition 3
Partition 4
In BOUNDARY VALUE ANALYSIS you will check the boundary values like 0, 1, 10, 11, 99, 100

Figure 12: Boundary Value Analysis [21]

In our earlier example instead of checking, one value for each partitions you will check the values at the partitions like 0, 1, 10, 11 and so on. As you may observe, you test values at **both valid and invalid boundaries**. Boundary Value Analysis is also called **range checking**.

Equivalence partitioning and boundary value analysis are closely related and can be used together at all levels of testing.

3.5.1 Limitation of Boundary Value Analysis:

Boundary Value Analysis- Limitations [19]**:**

a. One of the limitations of boundary value analysis is it cannot be used for Boolean and logical variables.

b. Cannot estimate boundary analysis for some cases such as countries.

c. Not that useful for strongly-typed languages.

CHAPTER 4: CASE STUDY AND TEST CASE MINIMIZATION FOR BVA (FOR INTEGER VALUE)

4.1 Background

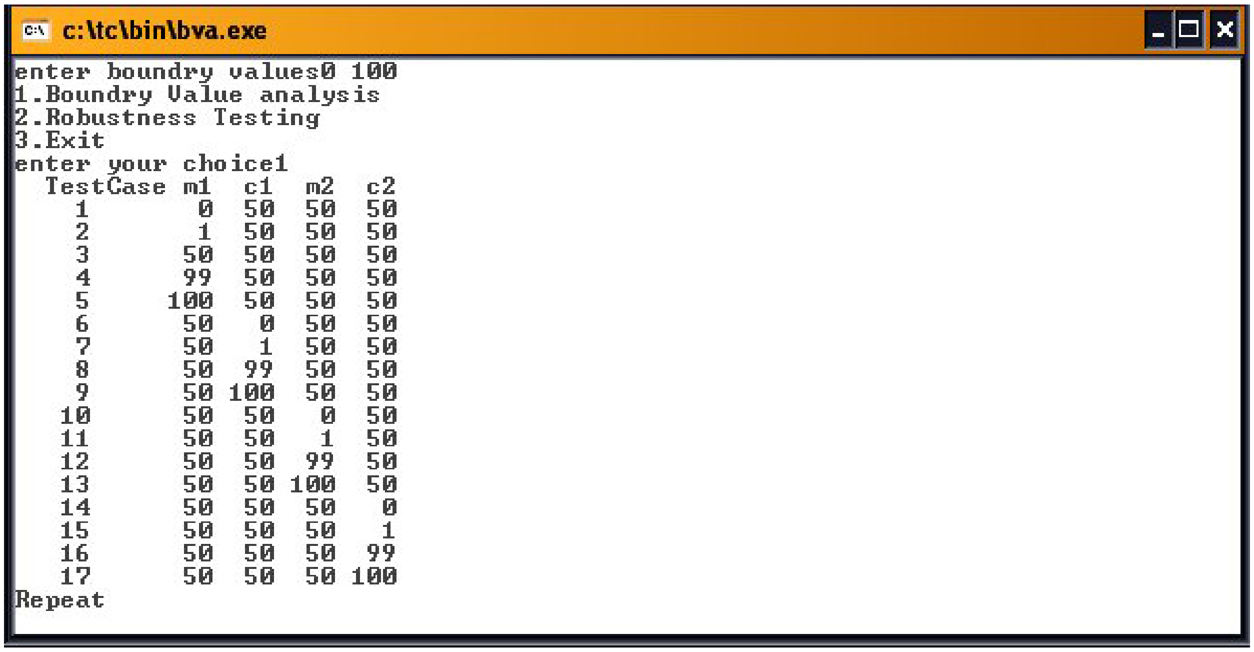
In black box testing, test cases are derived on the basis of values that lie on the edge of the equivalence partition [15]. It is found that most of the errors occur at the boundary rather than the middle of the domain. Boris Beizer, well-known author of testing book advises, "Bugs lurk in corners and congregate at boundaries." [15] Programmers often make mistakes on the boundaries of the equivalence classes/input domain. As a result, we need to focus testing at these boundaries. Boundary value is defined as a data value that corresponds to a minimum or maximum input, internal, or output value specified for a system or component [16]. For better product quality and identify the bugs on the boundaries of the equivalence class partition tester doing boundary value analysis and robustness testing. Usually in both boundary value analysis and robustness testing tester design test case randomly using maximum and minimum edge value of equivalence classes boundary. For designing test case randomly, the number of test cases increases. It is a big challenge for tester to minimize the number of test case with appropriate test value or test data and ensuring the coverage.

4.1.1 Case Study 01

K Aggarwal and Yogesh Singh in their book " *Software Engineering*, Revised 2nd Edition 2006." provided an equation for test cases number for boundary value analysis [17] as follows: In a program, two input variables X and Y that may have any value from 100 to 300 (ECP edge range) and the test cases for boundary value analysis are (200,100),(200,101),(200,200), (200,299),(200,300),(100,200),(101,200),(299,200) and (300,200). Nine test data set and eighteen test data. Thus, for a program of n variables boundary value analysis yields $4n+1$ test cases [17].

4.1.2 Case Study 02

Analysis of Black Box Software Testing Techniques: A Case Study by Mumtaz Ahmad Khan and Mohd. Sadiq [12] they provided a solution, a tool that automate the boundary value analysis test case. According to their solution for a range 0 to 100 they provided 17 test cases for boundary value analysis.



```
enter boundry values0 100
1.Boundry Value analysis
2.Robustness Testing
3.Exit
enter your choice1
  TestCase m1   c1   m2   c2
    1        0   50   50   50
    2        1   50   50   50
    3       50   50   50   50
    4       99   50   50   50
    5      100   50   50   50
    6       50    0   50   50
    7       50    1   50   50
    8       50   99   50   50
    9       50  100   50   50
   10       50   50    0   50
   11       50   50    1   50
   12       50   50   99   50
   13       50   50  100   50
   14       50   50   50    0
   15       50   50   50    1
   16       50   50   50   99
   17       50   50   50  100
Repeat
```

Figure 13: Boundary Value Analysis test case values from Mumtaz Ahmad Khan and Mohd. Sadiq [12]

Test cases of 0 to 100 range for boundary value analysis:

Test Cases	**M1**	**C1**	**M2**	**C2**	**Expected Output**
1	0	50	50	50	Intersecting lines
2	1	50	50	50	Intersecting lines
3	50	50	50	50	Coincident lines
4	99	50	50	50	Intersecting lines
5	100	50	50	50	Intersecting lines
6	50	0	50	50	parallel lines
7	50	1	50	50	parallel lines
8	50	99	50	50	parallel lines
9	50	100	50	50	parallel lines
10	50	50	0	50	Intersecting lines
11	50	50	1	50	Intersecting lines
12	50	50	99	50	Intersecting lines
13	50	50	100	50	Intersecting lines
14	50	50	50	0	parallel lines
15	50	50	50	1	parallel lines
16	50	50	50	99	parallel lines
17	50	50	50	100	parallel lines

Table 8: Test cases using Boundary Value Analysis by Mumtaz Ahmad Khan and Mohd. Sadiq [12]

4.2 Proposed Solution for Integer Boundary values

As we know in software development, testing costs a big percentage of total budgets. Often 40-60% of total budgets [23]. If we can minimize test cases without compromise the coverage, it will minimize time, budget of testing and overall improve the testing process as well as development process.

Let’s consider a text field that input a value 1 to 1000. So the valid Equivalence Class is 1-1000. The boundary value analysis must contain both valid data and invalid data.

The proposed solution for BVA is following:

N = Original Range,

N-1 = Original Range – 1,

N+1 = Original Range +1,

ECP = [1-1000]

So, N= 1 & 1000, (Valid TC)

N-1,

1-1 = 0 (Invalid TC)

1000-1= 999 (Valid TC)

N+1

1+1=2 (Valid TC)

1000+1 = 1001 (Invalid TC)

So the Test Cases for 1-1000 boundary value analysis are

Valid Test Case (Valid Test Data) = 1, 1000, 2, 999

Invalid Test Case (Invalid Test Data) = 0, 1001

The total numbers of test case for 1-1000 are only six and test data are 0, 1, 2, 999, 1000,1001. Therefore, for integer type variable it’s not matter that how big the ECP range is, only 6 test cases are enough for 100% boundary value analysis.

4.3 Proposed Solution for Floating point Boundary values

Let’s consider a text field that input a value 5.43 to 10.55. So the valid Equivalence Class is 1-1000. The boundary value analysis must contain both valid data and invalid data.

The proposed solution for floating point BVA is following:

N = Original Range,

a = Number of digit after decimal point (e.g. For 5.43, a=2)

N-1 = Original Range – 1,

N+1 = Original Range +1,

N+ $\frac{1}{10^a}$ = Original Range + $\frac{1}{10^a}$ (a=number of digit after decimal point)

N- $\frac{1}{10^a}$ = Original Range - $\frac{1}{10^a}$

ECP = [5.43-10.55]

So, N= 5.43 & 10.55, (Valid TC)

a= 2

N-1,

5.43-1= 4.43 (Invalid TC)

10.55-1= 9.55 (Valid TC)

N+1

5.43+1=6.43 (Valid TC)

10.55+1 = 11.55 (Invalid TC)

N+ $\frac{1}{10^a}$,

5.43+ $\frac{1}{10^2}$ = (5.43+0.01) =5.44 (Valid TC)

10.55+$\frac{1}{10^2}$ = (10.55+0.01)= 10.56 (Invalid TC)

N- $\frac{1}{10^a}$,

5.43- $\frac{1}{10^2}$ = (5.43-0.01) =5.42 (Invalid TC)

10.55-$\frac{1}{10^2}$ = (10.55-0.01)= 10.54 (Valid TC)

So the Test Cases for 5.43& 10.55 boundary value analysis are

Valid Test Case (Valid Test Data) = 5.43, 5.44,6.43,9.55 10.54, 10.55

Invalid Test Case (Invalid Test Data) = 4.43,5.42,10.56,11.55

The total numbers of test case for 5.43 to 10.55 are ten and test data are 5.43, 5.44,6.43,9.55 10.54, 10.55, 4.43,5.42,10.56,11.55

Therefore, not only integer type but float type variable is not a matter how big the ECP range is, only 10 test cases are enough for 100% boundary value analysis.

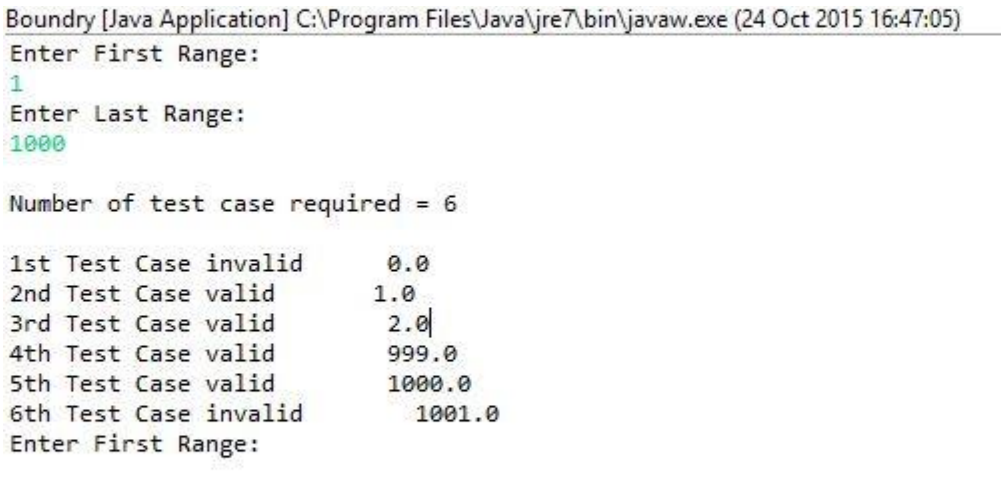
4.4 Proposed Tool for BVA

In this thesis we proposed a tool generated the test cases. Figure-14 contains the expected output for integer values of the proposed tool. Figure-15 contains the expected output for floating point values of the proposed tool. Following program shows the implementation of one of the module of the proposed tool.

```
import java.text.DecimalFormat;
import java.util.Scanner;


public class Boundry {

public static Scanner scan = new Scanner(System.in);

public static void main(String[] args) {

int v = 1;

while (v>0) {

System.out.println("Enter First Range: ");
double a2 = scan.nextDouble();
System.out.println("Enter Last Range: ");
double b2 = scan.nextDouble();

System.out.println("inputed values :"+a2+" - "+b2);


if(a2 % 1 == 0 && b2%1 == 0) {
System.out.println();
System.out.println("Number of test case required = 6");
System.out.println();

System.out.println("1st Test Case    Invalid     " + (int)(a2 - 1) );
System.out.println("2nd Test Case     Valid      " + (int)a2);
System.out.println("3rd Test Case     Valid      " + (int)(a2 + 1));
System.out.println("4th Test Case     Valid      " + (int)(b2 - 1));
System.out.println("5th Test Case     Valid      " + (int)b2);
System.out.println("6th Test Case     Invalid    " + (int)(b2 + 1));

} else {

String text1stDoube = Double.toString(Math.abs(a2));
int integerPlaces1 = text1stDoube.indexOf('.');
int decimalPlaces1 = text1stDoube.length() - integerPlaces1 - 1;



String text2ndDoube = Double.toString(Math.abs(b2));
int integerPlaces2 = text2ndDoube.indexOf('.');
int decimalPlaces2 = text2ndDoube.length() - integerPlaces2 - 1;

System.out.println();
System.out.println("Number of test case required = 10");
System.out.println();

System.out.println("1st Test Case    Invalid     " + (a2 - 1.00));
System.out.println("2nd Test Case    Invalid     " + (a2 - (Math.pow(1 / 10.0, decimalPlaces1))) );
System.out.println("3rd Test Case     Valid      " + a2);
System.out.println("4th Test Case     Valid      " + new DecimalFormat("##.##").format((a2 +
(Math.pow(1 / 10.0, decimalPlaces1)))));
System.out.println("5th Test Case     Valid      " + new DecimalFormat("##.##").format(a2+1));

System.out.println("6th Test Case     Valid      " + (b2-1));
System.out.println("7th Test Case     Valid      " + new DecimalFormat("##.##").format((b2 -
(Math.pow(1 / 10.0, decimalPlaces2)))));
System.out.println("8th Test Case     Valid      " + b2);
System.out.println("9th Test Case     Invalid    " + (b2 + (Math.pow(1 / 10.0, decimalPlaces1))));
System.out.println("10th Test Case     Invalid    " + (b2 + 1.00));

}

}

}

}
```

4.4.1 Output of the proposed tool for Integer values

Here we insert an Equivalence Class 1-1000 of integer values

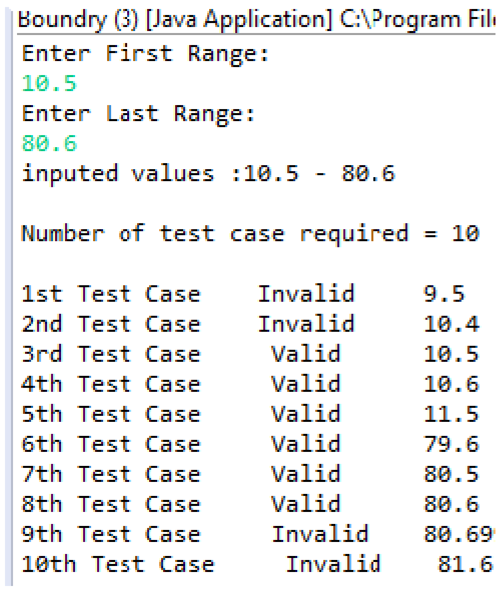


```
Boundry [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (24 Oct 2015 16:47:05)
Enter First Range:
1
Enter Last Range:
1000

Number of test case required = 6

1st Test Case invalid      0.0
2nd Test Case valid      1.0
3rd Test Case valid       2.0
4th Test Case valid       999.0
5th Test Case valid       1000.0
6th Test Case invalid       1001.0
Enter First Range:
```

Figure 14: Output of proposed tool for Integer values

4.4.2 Output of the proposed tool for Floating point values

Here we insert an Equivalence Class 10.5 – 80.6 of floating point values



```
Boundry (3) [Java Application] C:\Program Fil
Enter First Range:
10.5
Enter Last Range:
80.6
inputed values :10.5 - 80.6

Number of test case required = 10

1st Test Case    Invalid     9.5
2nd Test Case    Invalid     10.4
3rd Test Case     Valid      10.5
4th Test Case     Valid      10.6
5th Test Case     Valid      11.5
6th Test Case     Valid      79.6
7th Test Case     Valid      80.5
8th Test Case     Valid      80.6
9th Test Case     Invalid    80.69
10th Test Case     Invalid    81.6
```

Figure 15: Output of proposed tool for floating point values

4.6 Result and Analysis

For Integer: N, N+1,N-1

If we apply this formula in any integer type Equivalence Class of two variable (a range) the number of test case will be only six.

As we know the well establish formula for boundary value analysis test case is 4n+1.

If n= two variable,

So, the test cases number will be

4*2+1= 9

And 9*2=18   [2= valid & invalid data set]

Our proposed formula and tool remarkably minimize test case for boundary value analysis from 18 to only 6 with 100% coverage. Also if we are not testing any mission critical or safety critical product (Exhaustive Testing) robustness testing is not necessary for boundary value analysis based on our proposed solution for boundary value analysis as we give 100% coverage of equivalence class.

For Float: N, N+1,N-1, N+ $\frac{1}{10^a}$,N- $\frac{1}{10^a}$

If we apply this formula in any Floating Point Equivalence class of two variables (a range) the number of test case will be only ten.

As we know there is no establish formula or proposition for floating point boundary value analysis.

Our proposed formula and tool might be a new method for floating point boundary value analysis with 100% coverage. Also if we are not testing any mission critical or safety critical product (Exhaustive Testing) robustness testing is not necessary for boundary value analysis based on our proposed solution for boundary value analysis as we give 100% coverage of equivalence class.

4.7 Comparison for Integer values

Table 9 shows comparison with K Aggarwal and Yogesh Singh's solution and our proposed solution for Boundary Value Analysis.

ECP 100-300

BVA 100 & 300

	Existing Solution	**Proposed Solution**
Number of Test Cases	**19**	**6**

Test Case Values form *KK Aggarwal and Yogesh Singh [17]*	(200,100),(200,101),(200,200),(200,299),(200,300),(100,200),(101,200),(299,200) and (300,200)	Valid=100,101,299,300
		Invalid= 99, 301

Time Comparison	100%	33%(around)
Cost Comparison	100%	33%(around)

Table 9: Comparison with *K Aggarwal and Yogesh Singh's* solutions and our proposed solutions.

Table 10 shows comparison with *Mumtaz Ahmad Khan and Mohd. Sadiq's [12]* solution and our proposed solution for Boundary Value Analysis.

ECP 0-100

BVA 0 & 100

	Existing Solution	**Proposed Solution**
Number of Test Case	17	6

Test Case Values for *(Mumtaz Ahmad Khan and Mohd. Sadiq [12]*	17 test cases **x** 4 set data	Valid=0,100,1,99
		Invalid= -1, 101

Time Comparison	100%	33%(around)
Cost Comparison	100%	33%(around)

Table 10: Comparison with *Mumtaz Ahmad Khan and Mohd. Sadiq's* solutions and our proposed solutions.

4.7.1 Comparison for Floating Point value

We haven't found any equation for floating point values. So we proposed a generic formula for floating point boundary value analysis that generates 10 test cases with 100% coverage.

N = Original Range,

a = Number of digit after decimal point (e.g. For 5.43, a=2)

N-1 = Original Range – 1,

N+1 = Original Range +1,

N+ $\frac{1}{10^a}$ = Original Range + $\frac{1}{10^a}$ (a=number of digit after decimal point)

N- $\frac{1}{10^a}$ = Original Range - $\frac{1}{10^a}$

CHAPTER 5: CONCLUSION

5.1 Summary

As Glenford J. Myers [18] summaries, we can find that Boundary Value Analysis “if practised correctly, is one of the most useful test-case-design methods”. But he goes on to say that it is often used ineffectively as the testers often see it as so simple they misuse it, or don’t use it to its full potential. This is a very true interpretation of the use of Boundary Value Analysis. BVA can provide a relatively simple and formal testing technique that can be very powerful when used correctly.

The study is carried out by examining two questions: *Question 1 : Are there any Software Metric/Measurements to calculate BVA, like Cyclomatic Complexity (M=E-N+2P)?* and second is *Question 2: From existing solutions can we minimize the test cases for boundary value analysis ?*

For first question we got a general formula for boundary value analysis as follows 4n+1 , n addressed as variables number , if an ECP range contain two variable then test case number will be nine test data set and total (with valid and invalid) test input will be eighteen.

And for second question we are able to minimize test case for boundary value analysis from existing solutions. We proposed a generic formula for integer and floating point values and tool that automatically generate test case for boundary value analysis with 100% coverage. Our proposed solution deducts test cases from 18 to only 6 test cases for integer type boundary values. For floating point values we give an equation that generates 10 test cases with 100% coverage.

5.2 Future Work

Boundary value analysis has two type variables numeric (integer and floating point) and string. In our future work we will be work on a tool that automatically generate test cases for string variables.

References

[1] Software testing fundamentals by Andreas Spillner, Tilo Linz, Hans Schaefer

[2] ISTQB Foundation level syllabus

[3] A. Bertolino, "Chapter 5: Software Testing," in *IEEE SWEBOK Trial Version 1.00*, May 2001.

[4] P. Jorgensen, Software testing: a craftman's approach, CRC Press, 2002. p. 359.

[5] R. Pressman, Software Engineering: A Practitioner's Approach. Boston: McGraw Hill, 2001

[6] BLACK BOX AND WHITE BOX TESTING TECHNIQUES –A LITERATURE REVIEW Srinivas Nidhra and Jagruthi Dondeti . International Journal of Embedded Systems and Applications (IJESA) Vol.2, No.2, June 2012

[7] T. Murnane, K. Reed, and R. Hall, "On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis," in Software Engineering Conference, 2007. ASWEC 2007. 18th Australian, 2007, pp. 274 –283.

[8] T. Murnane and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in Software Engineering Conference, 2001. Proceedings. 2001 Australian, 2001, pp. 12 – 20.

[9] B. Beizer, Software Testing Techniques. Dreamtech Press, 2002

[10] M. Sharma and B. S. Chandra, "Automatic Generation of Test Suites from Decision Table – Theory and Implementation," in Software Engineering Advances (ICSEA), 2010 Fifth International Conference on, 2010, pp. 459 –464.

[11] L. Ran, C. Dyreson, A. Andrews, R. Bryce, and C. Mallery, "Building test cases and oracles to automate the testing of web database applications," Information and Software Technology, vol. 51, no. 2, pp. 460–477, Feb. 2009.

[12] Mumtaz Ahmad Khan, Mohd. Sadiq 2011 IEEE “Analysis of Black Box Software Testing Techniques: A Case Study “

[13] http://users.csc.calpoly.edu/~jdalbey/309/Lectures/coverage_defined.html

[14] [Copeland,2004] Copeland, L.(2004) *A Practitioner's Guide to Software Test Design*, Artech House: Norwood , MA

[15] B. Beizer, Black Box Testing. New York: John Wiley & Sons, Inc., 1995.

[16] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[17] KK Aggarwal and Yogesh Singh, Software Engineering, Revised 2nd Edition 2006.

[18] Glenford J. Myers, The Art of Software Testing, John Wiley and Sons, Inc. 2004

[19] T. Murnane, K. Reed, and R. Hall, “On the Learnability of Two Representations of Equivalence Partitioning and Boundary Value Analysis,” in Software Engineering Conference, 2007. ASWEC 2007. 18th Australian, 2007, pp. 274 –283.

[20] http://www.softwaretestingclass.com/what-is-decision-table-in-software-testing-with-example/

[21] http://www.guru99.com/software-testing-techniques-3.html

[22] http://testerstories.com/2014/06/path-testing-the-coverage/

[23] SOFTWARE ENGINEERING. Ninth Edition. Ian Sommerville. Addison-Wesley

[24] https://around.com/ariane.html

[25] http://www.cs.umd.edu/class/spring2003/cmsc838p/Misc/therac