An Effective Approach for Designing Suitable Test Cases for Small Laboratory Program: An Article

Madhumita Pal Mohanta
Dept. of MCA & CSE,
St Mary's Technical Campus
Kolkata,Barasat, India

Samir Malakar
Department of MCA,
MCKV Institute of Engineering,
Howrah, India

ABSTRACT

Effective but simple testing mechanisms to develop software testing skills at learning stage is an utmost required. Testing techniques can mostly be unexplored while learning any programming languages. Programs written seem to be correct and no directed testing is performed, therein the bugs present are not explored by the learners. This paper proposes simple testing approaches for small programs that can be exercised while learning any programming language.

General Terms

Software Testing, Software Engineering.

Keywords

Software testing, Software engineering, test case design, testing, test cases for small programs

1. INTRODUCTION

Testing in computer science is a process of evaluating a single program or a set of programs with the intention to improve the correctness and performance in all aspects of the system. It is an effective and real-life practice for improving the reliability of a system. Testing is being performed at every stage in software(S/W) development life cycle (SDLC).

Testing is broadly classified into three categories, namely unit testing (UT), integration testing (IT) and system testing (ST). UT is carried out by the developing team on the individual units of source code developed by the developers for validating their progression to the next level. IT is the testing of combined parts of an application to check whether the functions behave as per the requirement as individual or combined. Once all the components are integrated, the application as a whole is tested (i.e., ST) to see whether it meets requirement standards. Out of these testing mechanisms, unit testing plays a vital role for early detection of error and eventually it optimizes the testing cost as well as cost of the final product. It is better to learn unit testing mechanisms or test case design for unit testing while learning any programming language.

Test case generation for above mentioned all testing mechanisms is the most intellectual and demanding task. It is the most critical one, since it has a strong impact on the effectiveness and efficiency of the whole testing process [6]. As testing is labour-intensive and expensive, a number of works have been introduced to automate the testing process for cost reduction and improvement of software quality [6-12]. These automated test case generation tools provide good test quality in less time as compared of a manual approach, but they do not provide detailed knowledge about the techniques that are used for generation of test cases. As a result, a big gap found between the automated testing applications and practical usability of test case generation mechanisms, proposed by researchers.

Many research works for generation of test cases and test coverage has been proposed for UT. The motivation for the current work is to provide a simple but effective test case design approach, which can be adopted for testing small program(s) while learning. The approach must provide practical benefits to the learners along with simplicity so that it can be practiced before receiving advanced software engineering knowledge [13].

Testing is generally unexplored at the outset of any programming language learning. As a result, learners are totally unaware of all possible errors or bugs which are hidden in their code segment. Towards such aim this paper provides a simple test case design approach for small programs that learners should practice during their learning process. This experience of test case design at a very basic level will be very beneficial in long run.

2. PROPOSED MECHANISM

The overall sequential testing process and consequent test case design for testing laboratory code segment is presented by a flow diagram in Fig 1. This approach will give complete test coverage for testing of programs in any language. Throughout the article 'C' language code segment is considered.

2.1 Problem Analysis

The first step towards testing a code needs the problem domain knowledge for which the program was built up. Problem domain knowledge mainly consist the input domain of the program and the expected output to achieve. Without proper clarity of the problem domain, any test conducted may found to be erroneous in future. Careful examination of the problem statement will help the learners to find exact problem domain.

2.2 Static White Box (SWB)Testing

The next step for testing is the SWB testing [14]. This process includes careful and methodical reviews of the S/W design, architecture and code without executing it. The reason to perform SWB testing is to find bugs early that would be difficult to uncover or isolate later stages of testing. For small programs under this testing process, the following inspection and reviews need to be carried out to find possible bugs.

A) Check for recommended style for program writing [16]. Proper program writing style will considerably helpful to find and reduce syntactical and semantic errors in the program.

B)To verify the variable declarations, definitions and uses.

C) To ensure right logic implementation as compiler cannot detect logical error.

Syntax error checking or some other error checking (like, use of a variable without declaration, proper use of parentheses, etc.) will be checked by the compiler during compilation of the code.

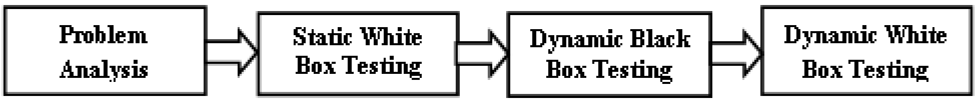

Fig 1: Flow diagram of overall testing process

2.3 Dynamic Black Box (DBB) Testing

The third step for testing is to perform DBB testing [14]. In this process, testing is done under program execution without having an insight into the details of underlying code. Proper design of test cases is highly expected for such testing which will be given as inputs to the program in execution. The corresponding outputs are checked to find bugs or errors if present. A number of techniques has been proposed [14,15] for generation of test cases, out of which two major techniques, namely, (1) Equivalence Partitioning and (2) Boundary Value Analysis, should be practiced for dynamic black box testing.

2.3.1 Test case generation by Equivalence Partitioning (EP)

EP is a method for deriving test cases from all possible classes of input domain set called Equivalence Classes. Each member of the equivalence class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned classes is considered for testing [15]. To perform EP, one needs to follow two steps, namely (1) Identifying equivalence classes and (2) Designing suitable test cases therein.

2.3.1.1 Identification of Equivalence classes

Two types of classes can always be identified. They are

a) Valid Equivalence Classes(VEC) which consider valid input set for the program and

b) Invalid Equivalence Classes (IEC) which consider invalid input set for the program that may leads to unexpected behaviour in the program.

2.3.1.2 Identification of test cases

A few guidelines are given below to identify test cases from the obtained equivalence classes. They are

(a) Assign a unique identification number to each equivalence classes.

(b) Write a new test case covering as many of the uncovered valid equivalence classes as possible.

(c) Write a test case that covers one and only one of the uncovered invalid equivalence classes, until all invalid equivalence classes have been covered by test cases [15].

EP method is clarified here with Example 1.

Example 1: Let a program that reads three input values (say A, B, C) of integer type with a range [1, 20] and prints the largest number. The partition of the input domain as valid and invalid input, i.e., the equivalence classes can be prepared as below.

Valid equivalence classes:

$C1 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge 1 <= A <= 20\}$
$C2 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge 1 <= B <= 20\}$
$C3 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge 1 <= C <= 20\}$

Invalid equivalence classes:

$C4 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge A<1\}$
$C5 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge A>20\}$
$C6 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge B<1\}$
$C7 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge B>20\}$
$C8 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge C<1\}$
$C9 = \{<A, B, C> \mid A, B, C \in \mathbb{Z} \wedge C>20\}$

The Test cases and the expected outcomes are depicted in Table 1.

Table1. Instance of a test case design using equivalence partitioning mechanism

Test Case ID	Variables			Expected Result	Classes Covered By the Test cases
	A	B	C		
1	12	19	5	B is greatest	C1,C2,C3
2	0	12	17	Invalid Input	C4
3	35	15	18	Invalid Input	C5
4	16	0	18	Invalid Input	C6
5	16	46	11	Invalid Input	C7
6	12	8	0	Invalid Input	C8
7	6	10	47	Invalid Input	C9

2.3.2 Test case generation by Boundary Value Analysis (BVA)

Test cases can be derived from the boundaries of equivalence classes. Boundary conditions are special because programming, by its nature, is vulnerable to problems at its edges [14]. For BVA, test for values on and at either sides of each boundary of equivalence classes are done. The test case design by BVA is explained using example 2.

Example 2:Let in an examination grading system, if the student scores 0 to less than 40 then assign E Grade, if the student scores between 40 to 49 then assign D Grade, if the student scores between 50 to 69 then assign C Grade, if the student scores between 70 to 89 then assign B Grade, and if the student scores between 90 to 100 then assign A Grade. Also consider that marks are awarded as a whole number.

Analyzing above problem the following equivalence classes are identified.

Valid equivalence classes:

C1: 0<=Grade<=39

C2: 40<= Grade<= 49
C3: 50<= Grade<= 69
C4: 70<= Grade <= 89
C5: 90<= Grade<= 100

Invalid equivalence classes:

C6: Grade <0
C7: Grade > 100

Now based on BVA the following input values to test boundaries of equivalence classes are shown in Table 2.

Table2. Instance of a test case design using BVA

Test Case ID	Variable	Expected Result
	Grade	
1	-1	Invalid Input
2	0	E Grade
3	1	E Grade
4	38	E Grade
5	39	E Grade
6	40	D Grade
7	41	D Grade
8	48	D Grade
9	49	D Grade
10	50	C Grade
11	51	C Grade
12	68	C Grade
13	69	C Grade
14	70	B Grade
15	71	B Grade
16	88	B Grade
17	89	B Grade
18	90	A Grade
19	91	A Grade
20	99	A Grade
21	100	A Grade
22	101	Invalid Input

2.4 Dynamic White Box (DWB) testing

DWB testing [14] is designed to serve the testing process of a system by having an insight view of the code as well as executing the same. The steps for DWB test case design are shown in a flow diagram in Fig 2. It includes (1) Basis Path testing, (2) Multiple condition testing, (3) Loop testing and (4)Data flow testing. These are the major testing schemas. Though these steps are shown sequentially in Fig 2 for learner's simplicity, but any sequence can be followed for experienced developers.

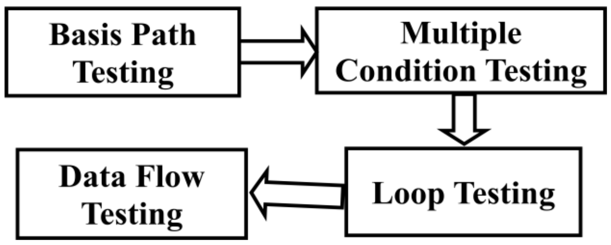

Fig. 2 Flow diagram for dynamic white box testing

2.4.1 Basis path testing

Test cases are designed based on independent paths in the flow graph of the code. A flow graph presents the flow of control in the program. To draw the flow graph, all statements of the program are numbered sequentially. The different numbered statements serve as nodes of the 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. Independent path is any path through a program that introduces at least one new set of processing statements (i.e. nodes). McCabe's Cyclomatic Complexity [2] is a S/W metric which provides the maximum number of linearly independent paths that are present in a flow graph. For a given graph G the Cyclomatic Complexity V (G) is calculated by any one of the following

a. The number of regions in the graph;

b. V (G) = e-n+2, where 'e' is the number of edges, and 'n' is the number of nodes;

c. V (G) = p+1, where 'p' is the number of decision node.

Calculation of V (G) is described using example 3.
Example 3:

```
1    while(x!=y){
2    if(x>y)
3    x=x-y;
4    else y=y-x;
5    }
6    return x;
```

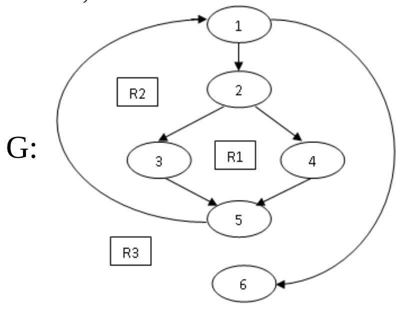

Fig. 3 Flow graph of Example 3

The Cyclomatic Complexity V(G) of the flow graph in Fig 3 can be obtained as V(G) = the number of regions in the graph = |{R1,R2,R3}| = 3. Therefore, the maximum number of linearly independent path in G is 3. One can choose the following three independent paths in G as:

Path1 (1->6);

Path2 (1->2->3->5->1->6);

Path3 (1->2->4->5->1->6).

Based on these independent paths the test cases that can be designed are:

TC1 [For the Path1]

Input: x=y   Expected Output: x

TC2 [For the Path2]

Input: x>y   Expected Output: x-y

TC3 [For the Path3]

Input: x<y   Expected Output; y-x

2.4.2 Multiple condition testing

Multiple Condition testing generates test cases for each possible combination of the conditions in a decision. For the decision (x<0 OR y<0) having two conditions, the test cases will be generated for each of the combination (False, False), (False, True), (True, False) and (True, True). Multiple condition testing is the most desirable structural coverage measure. But in some cases, for a decision with 'n' inputs it requires '2n' tests which may be impractical [3].

2.4.3 Loop testing

Error often occurs at the beginning and end of a loop in a program. Thus separate testing is needed for every loop in the program. The loops are broadly classified into four categories namely simple loops, nested loops, concatenated loops and unstructured loops [17].

For simple loop, perform the following test:

(a) Skip the loop entirely.

(b) Once pass through the loop.

(c) Two passes through the loop.

(d) m passes through the loop where m<n.

(e) n-1, n, n+1 passes through the loop, where 'n' is the maximum number of allowable passes through the loop. Choice of n depends on test case designers and complexity of the program.

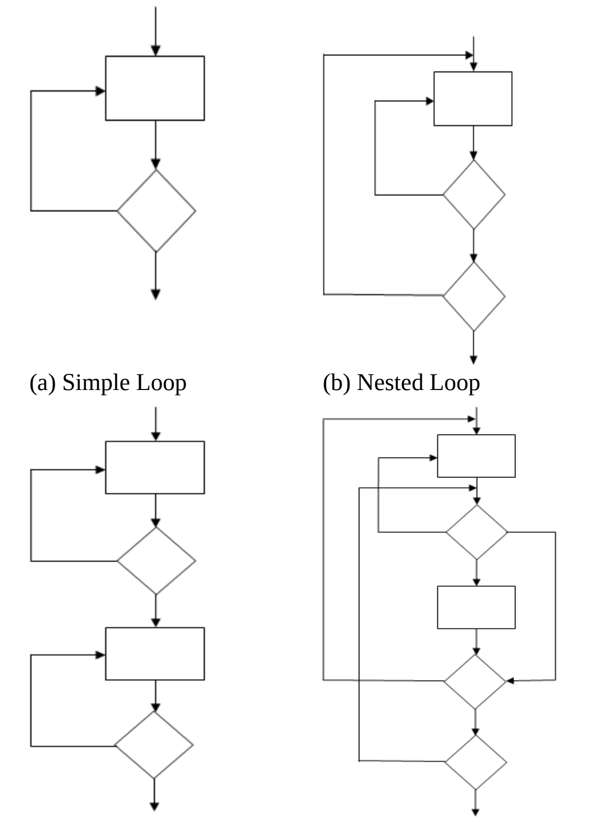

Fig 4: Schematic diagram of four categories of loop structure

Fig 4(a) shows a schematic diagram of a typical simple loop.

For Nested loop, with the increase of level of nesting, the number of probable tests also increases. This testing may seems to continue indefinitely. But to have a practical number of testing, it is recommended to practice the following approach:

(a) Start at the innermost loop, and set all other loops to minimum values.

(b) Conduct simple loop tests for the innermost loop holding the outer loop at their minimum iteration parameter value.

(c) Work outward, performing tests for the next loop.

(d) Continue until all loops have been tested.

Fig 4(b) shows a schematic diagram of a typical nested loop.

For Concatenated loop, testing are done using simple loop tests if each loop is independent from the other else nested loops tests are performed. Fig 4(c) shows a schematic diagram of a typical concatenated loop.

In case of Unstructured Loop, the loop should be redesigned into one among the above three types of loop or a combination of them [6]. Fig 4(d) shows a schematic diagram of a typical unstructured loop.

2.4.4 Data flow testing

Data flow testing is performed with the intention to uncover bugs in data usage during the execution of the code. Following steps should be considered for generating test cases in data flow testing.

(a) First, design the data flow graph. For this, find all the definition nodes and usage nodes for all the variables in the program and mark them in the flow graph.

Variables are defined by declaring and assigning values and they are used in expressions for certain operation. For example statement 'x = y+z;' defines variable x and uses variables y and z. Similarly statement 'scanf ("%d %d", &x,&y);' defines variables x and y. Statement 'printf("output : %d", x+y);' uses variables x and y. Declaration 'int x, y, A[10];' defines three variables x, y and A(where A is an array). Similarly statement 'if(x>y)' uses variables x and y.

Further, Usage node can be either a predicate usage node (p-use) or a computational usage node (c-use). Use of a variable (e.g. 'x') that occurs in an assignment statement (e.g. 'z=x+1;') or in an output statement (printf ("output : %d", x);) or as a parameter within a function call (e.g. 'f(x);') or in subscript expressions (e.g. A[x]), are classified as c-use. The occurrence of a variable in an expression used as a condition in a branch statement or a loop statement such as an 'if' or a 'while' is considered as a p-use. For example, in the statement 'while (z>x') both the variables z and x are of p-use.

(b) Trace all-use paths from the data flow graph.

All- uses states that at least one definition-clear path (dc-path) from every definition of every variable to every use of that definition be exercised under some test [1, 15]. A definition clear path (dc-path) with respect to a variable is a path between the definition node and the usage node such that no other node in that path is a defining node of that variable. Tracing all-use paths are described using example 4.

Example 4:

```
void test( ){

   1 intx ;
   2 int y=0;
   3 scanf("%d",x);
   4 if(x>0){
   5              y=40;
   6              if(x>20)
   7              {
   8                        if(x<=30)
   9                                 y=y+10;
   10                       else
   11                       {
   12                                y=y+x +50;
   13                                if(y>100)
   14                                         y=y+100;
15                              }
16                  }
17      }
18 printf("Final y is : %d", y);
19 return;
}
```

The data flow graph of the above example is obtained by marking all the definition nodes and usage nodes for all the variables of the program in the flow graph as shown in Fig 5.

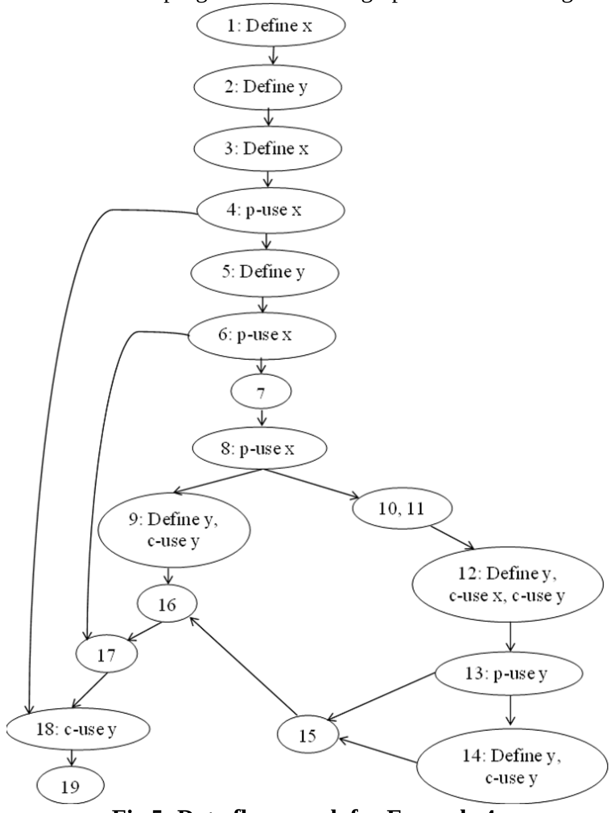

Fig 5: Data flow graph for Example 4

All the definition nodes and usage nodes for all the variables in the program are listed in Table 3.

From the data flow graph in Fig 5, the data flow testing paths for All-uses strategy (criterion) are listed in Table 4.

Test cases are designed (shown in Table 5) such that all the paths derived in Table 4 must be exercised under some test cases.

Table 3. List of definition nodes and usage nodes for all variables in Fig 5

Variables	Defined At	Used At
x	1,3	4,6,8,12
y	2,5,9,12,14	9,12,13,14,18

Table 4. Instance of All–use paths for all variables

Variable	All-uses paths
x	P1: 3->4 P2: 3->4->5->6 P3: 3->4->5->6->7->8 P4: 3->4->5->6->7->8->10->11->12
y	P5: 5->6->7->8->9 P6: 5->6->7->8->10->11->12 P7: 12->13 P8: 12->13->14 P9: 14->15->16->17->18

Table 5.Instance of a test case design using All–use paths

Test Case ID	x	Expecte d Result	Path covered
1	40(x>30)	y=130	P1, P2, P3, P4 ,P6, P7, P8, P9
2	25(20<x<=30)	y=50	P5

3. CASE STUDY

A systematic approach of the proposed mechanism is shown below that will help programmer to develop the test cases for small programs by considering example 5.

Example 5: The following program finds the remainder of a positive integer when divided by another non-zero positive integer.

```
void remainder()
   {
1           inta,b;
2           scanf("%d %d", &a, &b);
3           if(a<0 || b<=0)
4                   a=-1;  // -1 Represents invalid inputs
5           else
6                   if(a>=0 && b>0){
7                           while(a>=b)
8                                   a=a-b;
9           }
10          printf("remainder is %d", a);
     }
```

Step 1. After analysing the problem, the inputs are one positive integer and the other non-zero positive integer. The output to be shown is the remainder of the first number when divided by the second number

Step 2. SWB testing needs inspection mainly for proper logic implementation of the problem. Also visual checking for proper variable definition and the use of that variable needs to be done. Syntax checking will be done by the compiler during compilation of the code.

Step 3. Test cases for DBB testing are generated by using EP and BVA.

Step 3.1. Test case generation by EP

For the variables in the code (a & b), the range of the integers will depend on the c compiler. For a 16 bit Compiler like Turbo C or Turbo C++ the range is -32768 to 32767. For a 32 bit compiler like Visual Studio or gcc the range would be -2147483648 to 2147483647. The following equivalence classes are identified for the above program under test (assuming c compiler as Visual c++).

Valid equivalence classes:

$$C1 = \{<a, b> \mid a, b \in \mathbb{Z} \wedge 0<=a<=2147483647\}$$

$$C2 = \{<a, b> \mid a, b \in \mathbb{Z} \wedge 1<=b<=2147483647\}$$

Invalid equivalence classes:

$$C3 = \{<a, b> \mid a<0 \wedge 1<=b<=2147483647\}$$

$$C4 = \{<a, b> \mid a>2147483647 \wedge 1<=b<=2147483647\}$$

$$C5 = \{<a, b> \mid 0<=a<=2147483647 \wedge b<=0\}$$

$$C6 = \{<a, b> \mid 0<=a<=2147483647 \wedge b>2147483647\}$$

The test cases generated from the above equivalence classes are shown in Table 6.

Step 3.2. Test case generation by BVA

From the equivalence classes derived in Step 3.1, boundaries of all the classes are identified and the test cases generated from each boundary are listed in Table 7.

Table 6.List of test cases using Equivalence Partitioning mechanism

Test Case ID	Variables		Expected Result	Classes Covered By the Test cases
	a	b		
1	159	10	9(Valid Input)	C1,C2
2	-30	5	-1(Invalid Input)	C3
3	21474 83660	25	-1(Invalid Input)	C4
4	40	-25	-1(Invalid Input)	C5
5	35	21474 83670	-1(Invalid Input)	C6

Table 7. List of test cases using BVA

Test Case ID	Variables		Expected Result
	a	b	
6	-1	10	-1(Invalid Input)
7	0	10	0(Valid Input)
8	1	10	1(Valid Input)
9	2147483646	10	6(Valid Input)
10	2147483647	10	7(Valid Input)
11	2147483648	10	-1(Invalid Input)
12	100	0	-1(Invalid Input)
13	100	1	0(Valid Input)
14	100	2	0(Valid Input)
15	100	2147483646	100(Valid Input)
16	100	2147483647	100(Valid Input)
17	100	2147483648	-1(Invalid Input)

Step 4. Test cases for DWB testing are generated by using Basis Path testing, Multiple Condition Testing, Loop Testing and Data Flow Testing.

Step 4.1. Basis Path testing

Test cases are generated based on independent paths in the flow graph of the code segment. The flow graph of example 5 is shown in Fig 6.

The Cyclomatic complexity of the flow graph in Fig 6 is V(G)= e-n+2= 12- 10+2 =4. Thus maximum number of linearly independent path is 4.

The independent paths that can be obtained from the graph G in Fig 6 are:

Path 1(1->2->3->4->9->10)

Path 2(1->2->3->5->6->9->10)

Path 3(1->2->3->5->6->7->9->10)
Path 4(1->2->3->5->6->7->8->7->9->10)

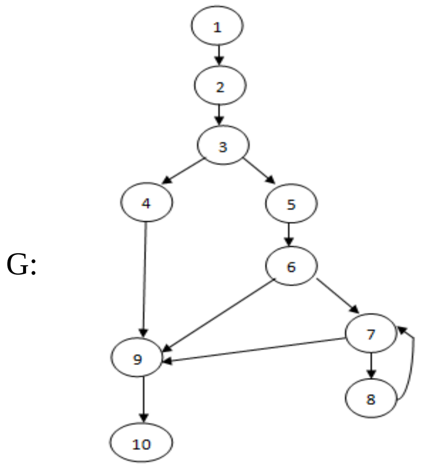

Fig 6: Flow graph for the code segment in Example 5

Based on the independent paths, the test cases that can be generated are listed in Table 8.

Here Path 2 cannot be covered since all the probable values of ‘a’ and ‘b’ has been considered already.

Step 4.2. Multiple Condition Testing

For the compound condition in statement 3 the test cases that can be generated are listed in Table 9

For the compound condition in statement 6 the test cases that can be generated are listed in Table 10.

Table 8.List of test cases using Basis Path testing

Test Case ID	Variables		Expected Result	Path Covered By the Test cases	Remarks
	a	b			
18	-30	5	-1(Invalid Input)	Path 1	Same as Test Case ID 2
19	1	10	1(Valid Input)	Path 3	Same as Test Case ID 8
20	5	4	1(Valid Input)	Path 4	

Step 4.3. Loop Testing

In example 5, only statement 7 contains a simple loop. The test cases for the simple loop in the statement 7 are listed in Table 11.

Step 4.4. Data Flow Testing

Analyzing the definition nodes and usage nodes for all the variables in example 5, the data flow graph is shown in Fig. 7.

All the definition nodes and usage nodes for all the variables in the program are listed Table 12.

Data flow testing paths for All-uses strategy (criterion) for the data flow diagram in Fig 7 are listed in Table 13.

Test cases that cover all the paths listed in Table 13 are shown in Table 14.

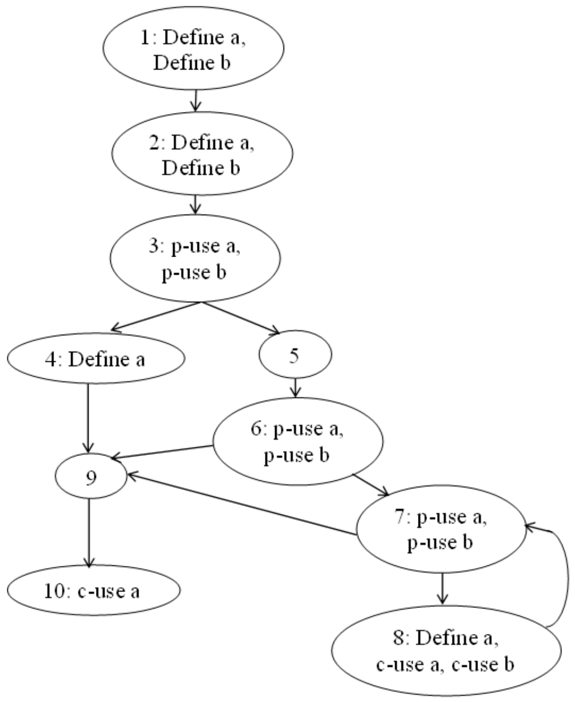

Fig 7: Data flow Graph for the code in Example 5

Table 9.List of test cases for the compound condition in statement 3 of example 5

Test Case ID	Variables		Expected Result	Combination Covered By the Test cases	Remarks
	a	b			
21	159	10	9(Valid Input)	(False, False)	Same as Test Case ID 1
22	40	-25	-1(Invalid Input)	(False, True)	Same as Test Case ID 4
23	-30	5	-1(Invalid Input)	(True, False)	Same as Test Case ID 2
24	-100	-100	-1(Invalid Input)	(True, True)	

Table 10.List of test cases for the compound condition in statement 6 of example 5

Test Case ID	Variables		Expected Result	Combination Covered By the Test cases	Remarks
	a	b			
25	-100	-100	-1(Invalid Input)	(False, False)	Same as Test Case ID 24
26	-30	5	-1(Invalid Input)	(False, True)	Same as Test Case ID 2
27	40	-25	-1(Invalid Input)	(True, False)	Same as Test Case ID 4
28	159	10	9(Valid Input)	(True, True)	Same as Test Case ID 1

Table 11.List of test cases for the simple loop in statement 7 of example 5

Test Case ID	Variables		Expected Result	Test Covered By the Test cases	Remarks
	a	b			
29	1	10	1(Valid Input)	Skip the loop entirely	Same as Test Case ID 8
30	5	4	1(Valid Input)	Only one pass through the loop	Same as Test Case ID 20
31	10	4	2(Valid Input)	Two passes through the loop	
32	159	10	9(Valid Input)	m passes through the loop where m<n, n = maximum number allowable passes through the loop	Same as Test Case ID 1
33	2147483646	1	0(Valid Input)	n-1 passes through the loop	-
34	2147483647	1	0(Valid Input)	n passes through the loop	-
35	2147483648	1	-1(Invalid Input)	For n+1 passes through the loop	-

Table 12. List of definition nodes and usage nodes for all the variables of example 5

Variables	Defined At	Used At
a	1,2,4,8	3,6,7,8,10
b	1,2	3,6,7,8

4. CONCLUSION

In this work, systematic and complete test coverage for testing of small program(s) is presented. As one proceeds on generating the test cases using various techniques, some repetition of the same test cases may occur, which are eliminated. The final test case set, for any program, is the collection of all the generated test cases using the proposed mechanism, without any repetition. The approach, as discussed here, is basically learner oriented and can easily be acquired by practicing them without advance knowledge of S/W engineering and testing.

Table 13. List of all-use paths for all the variables of example 5

Variables	All-uses paths
a	P1 : 2->3 P2 : 2->3->5->6 P3 : 2->3->5->6->7 P4 : 2->3->5->6->7->8 P5 : 4->9->10 P6 : 8->7->9->10
b	P7 : 2->3 P8 : 2->3->5->6 P9 : 2->3->5->6->7 P10 : 2->3->5->6->7->8

Table 14. List of a test cases based on All–use paths

Test Case ID	a	b	Expecte d Result	Path Covered By the Test cases	Remarks
36	159	10	9(Valid Input)	P1, P2, P3, P4, P6, P7, P8, P9, P10	Same as Test Case ID 1
37	40	-25	-1 (Invalid Input)	P5	Same as Test Case ID 4

5. REFERENCES

[1] H. Zhu, P. A. V. Hall, and J. H. R. May, Software unit test coverage and adequacy, ACM Computing Surveys, vol. 29, no. 4, pp. 366-427, 1997.

[2] T. J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2.4, pp. 308-320, 1976.

[3] J. H. Kelly, S. V. Dan, J. C. John, and K. R. Leanna, A Practical tutorial on modified condition/decision coverage, NASA/TM-2001-210876, NASA, (May), pp. 7-14, 2001.

[4] S. Rapps and E. J. Weyuker, Data flow analysis techniques for test data selection. IEEE, pp. 272-278.

[5] Bieman, J. M. and Schultz, J. L. 1992. An Empirical Evaluation (and Specication) of the All-du-paths Testing Criterion (Extended Version), IEE/BCS Software. Eng. J., 7(1) (Jan.), pp. 43-51, 1982.

[6] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, An orchestrated survey on automated software test case generation. Journal of Systems and Software, vol. 86, no. 8, 1978 – 2001, 2013.

[7] A. J. H. Simons and C. D. Thomson, Benchmarking Effectiveness for Object-Oriented Unit Testing, in 'ICST Workshops', IEEE Computer Society,pp. 375-379, 2008.

[8] B. Garcia, J. C. Duenas, and H. A. Parada G., Automatic functional and structural test case generation for web applications based on agile frameworks, Proc. of the 5th International Workshop on Automated Specification and Verification of Web Systems (WWV09). Hagenberg, Austria, July. 2009.

[9] Y. Cheon, and G. T. Leavens, A Simple and Practical Approach to Unit Testing: The JML and JUnit Way, ECOOP 2002, LNCS 2374, pp. 231-255, 2002.

[10] L. Williams, G. Kudrjavets, and N. Nagappan, On the Effectiveness of Unit Test Automation at Microsoft, IEEE International Symposium on Software Reliability Engineering (ISSRE), pp. 81- 89, 2009.

[11] K. Beck, TheJUnit Pocket Guide, 1st ed., O'Reilly, Beijing, 2004.

[12] A. J. H. Simons, JWalk: a tool for lazy, systematic testing of Java classes by design introspection and user interaction. Automated Software Engineering, 14 (4), December, ed. B. Nuseibeh, (Springer, USA), pp. 369-418, 2007.

[13] S. H. Edwards, Using Test-Driven Development in the Classroom: Providing Students with Automatic, Concrete Feedback on Performance. EISTA, International Institute of Informatics and Systemics, pp. 421–426, 2003.

[14] R. Patton, Software Testing, 2nd ed., Pearson India, 2005.

[15] N. Chauhan, Software Testing: Principles and Practices, Oxford University Press, 2010.

[16] (1994) C Style Guide, Nasa. [Online]. Available: http://homepages.inf.ed.ac.uk/dts/pm/Papers/nasa-c-style.pdf

[17] I. M. Jovanovic, Software Testing Methods and Techniques. The IPSI BgD Transactions on Internet Research, vol. 30, 2008.