UNIT 2

Dynamic testing: Black-Box Testing Techniques:

Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as functional testing. The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.

It is obvious that in black-box technique, test cases are designed based on functional specifications. Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software, as shown in Fig. 4.1.

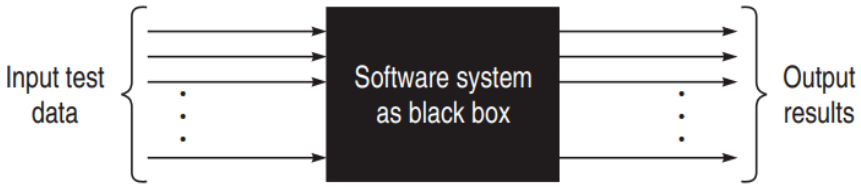


Figure 4.1 Black-box testing

Black-box testing attempts to find errors in the following categories:

- To test the modules independently.
- To test the functional validity of the software so that incorrect or missing functions can be recognized.
- To look for interface errors.
- To test the system behaviour and check its performance.
- To test the maximum load or stress on the system.
- To test the software such that the user/customer accepts the system within defined acceptable limits.

There are various methods to test a software product using black-box techniques. One method chooses the boundary values of the variables, another makes equivalence classes so that only one test case in that class is chosen and executed. Some methods use the state diagrams of the system to form the black-box test cases, while a few other methods use table structure to organize the test cases. It does not mean that one can pick any of these methods for testing the software. Sometimes a combination of methods is employed for rigorous testing. The objective of any test case is to have maximum coverage and capability to discover more and more errors.

BOUNDARY VALUE ANALYSIS (BVA)

An effective test case design requires test cases to be designed such that they maximize the probability of finding errors. BVA technique addresses this issue. With the experience of testing team, it has been observed that test cases designed with boundary input values have a high chance to find errors. It means that most of the failures crop up due to boundary values.

BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain. For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9, 10, 11) and on 255(256, 255, 254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9, 10, 11) and 100(99, 100, 101), as shown in Fig. 4.2.

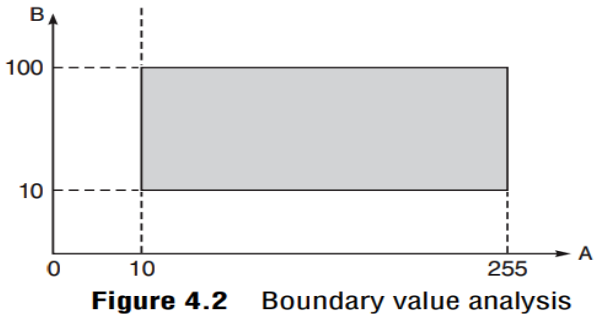


Figure 4.2 Boundary value analysis

BVA offers several methods to design test cases

1. BOUNDARY VALUE CHECKING (BVC)
2. ROBUSTNESS TESTING METHOD
3. WORST-CASE TESTING METHOD

BOUNDARY VALUE CHECKING (BVC): -

In this method, the test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain.

The variable at its extreme value can be selected at:

(a) Minimum value (Min)

(b) Value just above the minimum value (Min+)

(c) Maximum value (Max)

(d) Value just below the maximum value (Max−)

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases (see Fig. 4.3) can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

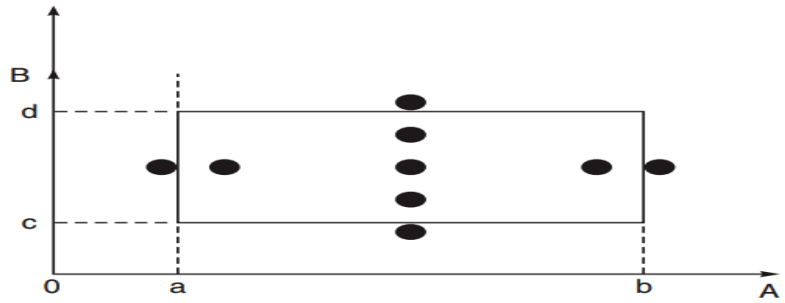


Figure 4.3 Boundary value checking

It can be generalized that for n variables in a module, 4n + 1 test cases can be designed with boundary value checking method.

ROBUSTNESS TESTING METHOD: -

The idea of BVC can be extended such that boundary values are exceeded as:

- A value just greater than the Maximum value (Max+)
- A value just less than Minimum value (Min−)

When test cases are designed considering the above points in addition to BVC, it is called robustness testing.

Let us take the previous example again. Add the following test cases to the list of 9 test cases designed in BVC:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

10. A_{max+}, B_{nom} 11. A_{min-}, B_{nom}

12. A_{nom}, B_{max+} 13. A_{nom}, B_{min-}

It can be generalized that for n input variables in a module, 6n + 1 test cases can be designed with robustness testing.

WORST-CASE TESTING METHOD: -

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called worst-case testing method.

Again, take the previous example of two variables, A and B. We can add the following test cases to the list of 9 test cases designed in BVC as:

1. A_{nom}, B_{min} 2. A_{nom}, B_{min+}
3. A_{nom}, B_{max} 4. A_{nom}, B_{max-}
5. A_{min}, B_{nom} 6. A_{min+}, B_{nom}
7. A_{max}, B_{nom} 8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}
10. A_{min}, B_{min} 11. A_{min+}, B_{min}
12. A_{min}, B_{min+} 13. A_{min+}, B_{min+}
14. A_{max}, B_{min} 15. A_{max-}, B_{min}
16. A_{max}, B_{min+} 17. A_{max-}, B_{min+}
18. A_{min}, B_{max} 19. A_{min+}, B_{max}
20. A_{min}, B_{max-} 21. A_{min+}, B_{max-}
22. A_{max}, B_{max} 23. A_{max-}, B_{max}
24. A_{max}, B_{max-} 25. A_{max-}, B_{max-}

It can be generalized that for n input variables in a module, 5^n test cases can be designed with worst-case testing.

BVA is applicable when the module to be tested is a function of several independent variables. This method becomes important for physical quantities where boundary condition checking is crucial. For example, systems having requirements of minimum and maximum temperature, pressure or speed, etc. However, it is not useful for Boolean variables.

Example 4.1

A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) Test cases using BVC: - Since there is one variable, the total number of test cases will be 4n + 1 = 5.

In our example, the set of minimum and maximum values is shown below:

Min value = 1
Min^+ value = 2
Max value = 100
Max^- value = 99
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

(b) Test cases using robust testing: - Since there is one variable, the total number of test cases will be 6n + 1 = 7. The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

Using these values, test cases can be designed as shown below:

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

(c) Test cases using worst-case testing: - Since there is one variable, the total number of test cases will be $5^n = 5$. Therefore, the number of test cases will be same as BVC.

Example 4.2

A program computes a^b where a lies in the range [1, 10] and b within [1, 5]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution: -

(a) Test cases using BVC Since there are two variables, a and b, the total number of test cases will be 4n + 1 = 9. The set of boundary values is shown below:

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

(b) Test cases using robust testing: - Since there are two variables, a and b, the total number of test cases will be 6n + 1 = 13. The set of boundary values is shown below:

	a	b
Min value	1	1
Min^- value	0	0
Min^+ value	2	2
Max value	10	5
Max^+ value	11	6
Max^- value	9	4
Nominal value	5	3

Using these values, test cases can be designed as shown below:

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

(c) Test cases using worst-case testing: - Since there are two variables, a and b, the total number of test cases will be $5^n = 25$.

The set of boundary values is shown below:

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

There may be more than one variable at extreme values in this case. Therefore, test cases can be designed as shown below:

Test Case ID	a	b	Expected Output
1	1	1	1
2	1	2	1
3	1	3	3
4	1	4	1
5	1	5	1
6	2	1	2
7	2	2	4
8	2	3	8
9	2	4	16
10	2	5	32
11	5	1	5
12	5	2	25
13	5	3	125
14	5	4	625
15	5	5	3125
16	9	1	9
17	9	2	81
18	9	3	729
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000

Example 4.3

A program reads three numbers, A, B, and C, within the range [1, 50] and prints the largest number. Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) Test cases using BVC: - Since there are three variables, A, B, and C, the total number of test cases will be 4n + 1 = 13. The set of boundary values is shown below:

Min value = 1
Min^{+} value = 2
Max value = 50
Max^{-} value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	1	25	27	C is largest
2	2	25	28	C is largest
3	49	25	25	B and C are largest
4	50	25	29	A is largest
5	25	1	30	C is largest
6	25	2	26	C is largest
7	25	49	27	B is largest
8	25	50	28	B is largest
9	25	28	1	B is largest
10	25	27	2	B is largest
11	25	26	49	C is largest
12	25	26	50	C is largest
13	25	25	25	Three are equal

(b) Test cases using robust testing: - Since there are three variables, A, B, and C, the total number of test cases will be 6n + 1 = 19.

The set of boundary values is shown below:

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 50
Max^+ value = 51
Max^- value = 49
Nominal value = 25–30

Using these values, test cases can be designed as shown below:

Test Case ID	A	B	C	Expected Output
1	0	25	27	Invalid input
2	1	25	27	C is largest
3	2	25	28	C is largest
4	49	25	25	B and C are largest
5	50	25	29	A is largest
6	51	27	25	Invalid input
7	25	0	26	Invalid input
8	25	1	30	C is largest
9	25	2	26	C is largest
10	25	49	27	B is largest
11	25	50	28	B is largest
12	26	51	25	Invalid input
13	25	25	0	Invalid input
14	25	28	1	B is largest
15	25	27	2	B is largest
16	25	26	49	C is largest
17	25	26	50	C is largest
18	25	29	51	Invalid input
19	25	25	25	Three are equal

(c) Test cases using worst-case testing: - Since there are three variables, A, B, and C, the total number of test cases will be $5^n = 125$.

The set of boundary values is shown below:

Min value = 1
Min^+ value = 2
Max value = 50
Max^- value = 49
Nominal value = 25–30

There may be more than one variable at extreme values in this case. Therefore, test cases can be design as shown below:

Test Case ID	A	B	C	Expected Output
1	1	1	1	All three are equal
2	1	1	2	C is greatest
3	1	1	25	C is greatest
4	1	1	49	C is greatest
5	1	1	50	C is greatest
6	1	2	1	B is greatest
7	1	2	2	B and C
8	1	2	25	C is greatest
9	1	2	49	C is greatest
10	1	2	50	C is greatest
11	1	25	1	B is greatest
12	1	27	2	B is greatest
13	1	26	25	B is greatest
14	1	25	49	B is greatest
15	1	27	50	C is greatest
16	1	49	1	B is greatest
17	1	49	2	B is greatest
18	1	49	25	B is greatest
19	1	49	49	B and C
20	1	49	50	C is greatest
21	1	50	1	B is greatest
22	1	50	2	B is greatest
23	1	50	25	B is greatest
24	1	50	49	B is greatest
25	1	50	50	B and C
26	2	1	1	A is largest
27	2	1	2	A and C
28	2	1	25	C is greatest
29	2	1	49	C is greatest
30	2	1	50	C is greatest
31	2	2	1	A and B
32	2	2	2	All three are equal
33	2	2	25	C is greatest
34	2	2	49	C is greatest
35	2	2	50	C is greatest
36	2	25	1	B is greatest
37	2	27	2	B is greatest
38	2	28	25	B is greatest
39	2	26	49	C is greatest
40	2	28	50	C is greatest
41	2	49	1	B is greatest
42	2	49	2	B is greatest
43	2	49	25	B is greatest
44	2	49	49	B and C
45	2	49	50	C is greatest
46	2	50	1	B is greatest
47	2	50	2	B is greatest
48	2	50	25	B is greatest
49	2	50	49	B is greatest
50	2	50	50	B and C
51	25	1	1	A is greatest
52	25	1	2	A is greatest
53	25	1	25	A and C
54	25	1	49	C is greatest
55	25	1	50	C is greatest
56	25	2	1	A is greatest
57	25	2	2	A is greatest
58	25	2	25	A and C
59	25	2	49	C is greatest
60	25	2	50	C is greatest

61	25	27	1	B is greatest
62	25	26	2	B is greatest
63	25	25	25	All three are equal
64	25	28	49	C is greatest
65	25	29	50	C is greatest
66	25	49	1	B is greatest
67	25	49	2	B is greatest
68	25	49	25	B is greatest
69	25	49	49	B is greatest
70	25	49	50	C is greatest
71	25	50	1	B is greatest
72	25	50	2	B is greatest
73	25	50	25	B is greatest
74	25	50	49	B is greatest
75	25	50	50	B is greatest
76	49	1	1	A is greatest
77	49	1	2	A is greatest
78	49	1	25	A is greatest
79	49	1	49	A and C
80	49	1	50	C is greatest
81	49	2	1	A is greatest
82	49	2	2	A is greatest
83	49	2	25	A is greatest
84	49	2	49	A and C
85	49	2	50	C is greatest
86	49	25	1	A is greatest
87	49	29	2	A is greatest
88	49	25	25	A is greatest
89	49	27	49	A and C
90	49	28	50	C is greatest
91	49	49	1	A and B
92	49	49	2	A and B
93	49	49	25	A and B
94	49	49	49	All three are equal
95	49	49	50	C is greatest
96	49	50	1	B is greatest
97	49	50	2	B is greatest
98	49	50	25	B is greatest
99	49	50	49	B is greatest
100	49	50	50	B and C
101	50	1	1	A is greatest
102	50	1	2	A is greatest
103	50	1	25	A is greatest
104	50	1	49	A is greatest
105	50	1	50	A and C
106	50	2	1	A is greatest
107	50	2	2	A is greatest
108	50	2	25	A is greatest
109	50	2	49	A is greatest
110	50	2	50	A and C
111	50	26	1	A is greatest
112	50	25	2	A is greatest
113	50	27	25	A is greatest
114	50	29	49	A is greatest
115	50	30	50	A and C
116	50	49	1	A is greatest
117	50	49	2	A is greatest
118	50	49	26	A is greatest
119	50	49	49	A is greatest
120	50	49	50	A and C
121	50	50	1	A and B
122	50	50	2	A and B
123	50	50	26	A and B
124	50	50	49	A and B
125	50	50	50	All three are equal

Example 4.4

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$$1 \le mm \le 12$$

$$1 \le dd \le 31$$

$$1900 \le yyyy \le 2025$$

Its output would be the next date or it will display 'invalid date.' Design test cases for this program using BVC, robust testing, and worst-case testing methods.

Solution

(a) Test cases using BVC Since there are three variables, month, day, and year, the total number of test cases will be $4n + 1 = 13$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min$^+$ value	2	2	1901
Max value	12	31	2025
Max$^-$ value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	15	1962	16-1-1962
2	2	15	1962	16-2-1962
3	11	15	1962	16-11-1962
4	12	15	1962	16-12-1962
5	6	1	1962	2-6-1962
6	6	2	1962	3-6-1962
7	6	30	1962	1-7-1962
8	6	31	1962	Invalid input
9	6	15	1900	16-6-1900
10	6	15	1901	16-6-1901
11	6	15	2024	16-6-2024
12	6	15	2025	16-6-2025
13	6	15	1962	16-6-1962

(b) Test cases using robust testing The total number of test cases will be $6n + 1 = 19$. The set of boundary values is shown below:

	Month	Day	Year
Min^- value	0	0	1899
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Max^+ value	13	32	2026
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	0	15	1962	Invalid date
2	1	15	1962	16-1-1962
3	2	15	1962	16-2-1962
4	11	15	1962	16-11-1962
5	12	15	1962	16-12-1962
6	13	15	1962	Invalid date
7	6	0	1962	Invalid date
8	6	1	1962	2-6-1962
9	6	2	1962	3-6-1962
10	6	30	1962	1-7-1962
11	6	31	1962	Invalid input
12	6	32	1962	Invalid date
13	6	15	1899	Invalid date
14	6	15	1900	16-6-1900
15	6	15	1901	16-6-1901
16	6	15	2024	16-6-2024
17	6	15	2025	16-6-2025
18	6	15	2026	Invalid date
19	6	15	1962	16-6-1962

(c) Test cases using worst-case testing The total number of test cases will be $5^n = 125$. The set of boundary values is shown below:

	Month	Day	Year
Min value	1	1	1900
Min^+ value	2	2	1901
Max value	12	31	2025
Max^- value	11	30	2024
Nominal value	6	15	1962

Using these values, test cases can be designed as shown below:

Test Case ID	Month	Day	Year	Expected Output
1	1	1	1900	2-1-1900
2	1	1	1901	2-1-1901
3	1	1	1962	2-1-1962
4	1	1	2024	2-1-2024
5	1	1	2025	2-1-2025
6	1	2	1900	3-1-1900
7	1	2	1901	3-1-1901
8	1	2	1962	3-1-1962
9	1	2	2024	3-1-2024
10	1	2	2025	3-1-2025
11	1	15	1900	16-1-1900
12	1	15	1901	16-1-1901
13	1	15	1962	16-1-1962
14	1	15	2024	16-1-2024
15	1	15	2025	16-1-2025
16	1	30	1900	31-1-1900
17	1	30	1901	31-1-1901
18	1	30	1962	31-1-1962
19	1	30	2024	31-1-2024
20	1	30	2025	31-1-2025
21	1	31	1900	1-2-1900
22	1	31	1901	1-2-1901
23	1	31	1962	1-2-1962
24	1	31	2024	1-2-2024
25	1	31	2025	1-2-2025
26	2	1	1900	2-2-1900
27	2	1	1901	2-2-1901
28	2	1	1962	2-2-1962
29	2	1	2024	2-1-2024
30	2	1	2025	2-2-2025
31	2	2	1900	3-2-1900
32	2	2	1901	3-2-1901
33	2	2	1962	3-2-1962
34	2	2	2024	3-2-2024
35	2	2	2025	3-2-2025
36	2	15	1900	16-2-1900
37	2	15	1901	16-2-1901
38	2	15	1962	16-2-1962
39	2	15	2024	16-2-2024
40	2	15	2025	16-2-2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	2-6-1900
52	6	1	1901	2-6-1901
53	6	1	1962	2-6-1962
54	6	1	2024	2-6-2024

55	6	1	2025	2-6-2025
56	6	2	1900	3-6-1900
57	6	2	1901	3-6-1901
58	6	2	1962	3-6-1962
59	6	2	2024	3-6-2024
60	6	2	2025	3-6-2025
61	6	15	1900	16-6-1900
62	6	15	1901	16-6-1901
63	6	15	1962	16-6-1962
64	6	15	2024	16-6-2024
65	6	15	2025	16-6-2025
66	6	30	1900	1-7-1900
67	6	30	1901	1-7-1901
68	6	30	1962	1-7-1962
69	6	30	2024	1-7-2024
70	6	30	2025	1-7-2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	2-11-1900
77	11	1	1901	2-11-1901
78	11	1	1962	2-11-1962
79	11	1	2024	2-11-2024
80	11	1	2025	2-11-2025
81	11	2	1900	3-11-1900
82	11	2	1901	3-11-1901
83	11	2	1962	3-11-1962
84	11	2	2024	3-11-2024
85	11	2	2025	3-11-2025
86	11	15	1900	16-11-1900
87	11	15	1901	16-11-1901
88	11	15	1962	16-11-1962
89	11	15	2024	16-11-2024
90	11	15	2025	16-11-2025
91	11	30	1900	1-12-1900
92	11	30	1901	1-12-1901
93	11	30	1962	1-12-1962
94	11	30	2024	1-12-2024
95	11	30	2025	1-12-2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date
101	12	1	1900	2-12-1900
102	12	1	1901	2-12-1901
103	12	1	1962	2-12-1962
104	12	1	2024	2-12-2024
105	12	1	2025	2-12-2025
106	12	2	1900	3-12-1900
107	12	2	1901	3-12-1901
108	12	2	1962	3-12-1962
109	12	2	2024	3-12-2024
110	12	2	2025	3-12-2025

111	12	15	1900	16-12-1900
112	12	15	1901	16-12-1901
113	12	15	1962	16-12-1962
114	12	15	2024	16-12-2024
115	12	15	2025	16-12-2025
116	12	30	1900	31-12-1900
117	12	30	1901	31-12-1901
118	12	30	1962	31-12-1962
119	12	30	2024	31-12-2024
120	12	30	2025	31-12-2025
121	12	31	1900	1-1-1901
122	12	31	1901	1-1-1902
123	12	31	1962	1-1-1963
124	12	31	2024	1-1-2025
125	12	31	2025	1-1-2026

EQUIVALENCE CLASS TESTING

We know that the input domain for testing is too large to test every input. So we can divide or partition the input domain based on a common feature or a class of data. Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors. This test case will have a representative value of a class which is equivalent to a test case containing any other value in the same class. If one test case in an equivalence class detects a bug, all other test cases in that class have the same probability of finding bugs. Therefore, instead of taking every value in one domain, only one test case is chosen from one class. In this way, testing covers the whole input domain, thereby reduces the total number of test cases. In fact, it is an attempt to get a good hit rate to find maximum errors with the smallest number of test cases.

Equivalence partitioning method for designing test cases has the following goals:

Completeness: - Without executing all the test cases, we strive to touch the completeness of testing domain.

Non-redundancy: - When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases. Time and resources are wasted in executing these redundant test cases, as they explore the same type of bug. Thus, the goal of equivalence partitioning method is to reduce these redundant test cases.

To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes
2. Design test cases

1. IDENTIFICATION OF EQUIVALENT CLASSES: -

How do we partition the whole input domain? Different equivalence classes are formed by grouping inputs for which the behaviour pattern of the module is similar. The rationale of forming equivalence classes like this is the assumption that if the specifications require exactly the same behaviour for each element in a class of values, then the program is likely to be constructed such that it either succeeds or fails for each value in that class. For example, the specifications of a module that determines the absolute value for integers specify different behaviour patterns for positive and negative integers. In this case, we will form two classes: one consisting of positive integers and another consisting of negative integers

Two types of classes can always be identified as discussed below:

Valid equivalence classes: - These classes consider valid inputs to the program.

Invalid equivalence classes: - One must not be restricted to valid inputs only.

We should also consider invalid inputs that will generate error conditions or unexpected behaviour of the program, as shown in Fig. 4.4.

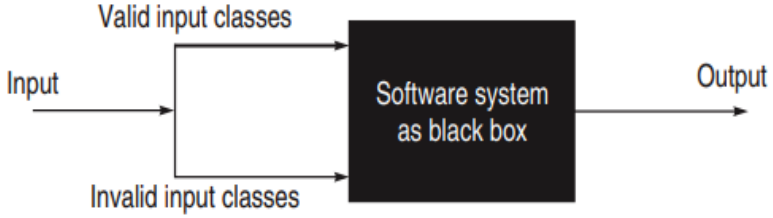


Figure 4.4 Equivalence classes

There are no well-defined rules for identifying equivalence classes, as it is a heuristic process. However, some guidelines are defined for forming equivalence classes:

- If there is no reason to believe that the entire range of an input will be treated in the same manner, then the range should be split into two or more equivalence classes.
- If a program handles each valid input differently, then define one valid equivalence class per valid input.
- Boundary value analysis can help in identifying the classes. For example, for an input condition, say 0 <= a <= 100, one valid equivalent class can be formed from the valid range of a. And with BVA, two invalid classes that cross the minimum and maximum values can be identified, i.e. a < 0 and a > 100.
- If an input variable can identify more than one category, then for each category, we can make equivalent classes. For example, if the input is a character, then it can be an alphabet, a number, or a special character. So we can make three valid classes for this input and one invalid class.
- If the requirements state that the number of items input by the system at some point must lie within a certain range, specify one valid class where the number of inputs is within the valid range, one invalid class where there are very few inputs, and one invalid class where there are too many inputs. For example, specifications state that a maximum of 4 purchase orders can be registered against a product. The equivalence classes are: the valid equivalence class (1 ≤ no. of purchase orders ≤ 4), the invalid class (no. of purchase orders > 4), and the invalid class (no. of purchase orders < 1).
- If an input condition specifies a 'must be' situation (e.g., 'first character of the identifier must be a letter'), identify a valid equivalence class (it is a letter) and an invalid equivalence class (it is not a letter).
- Equivalence classes can be of the output desired in the program. For an output equivalence class, the goal is to generate test cases such that the output for that test case lies in the output equivalence class. Determining test cases for output classes may be more difficult, but output classes have been found to reveal errors that are not revealed by just considering the input classes.
- Look for membership of an input condition in a set or group and identify valid (within the set) and invalid (outside the set) classes. For example, if the requirements state that a valid province code is ON, QU, and NB, then identify: the valid class (code is one of ON, QU, NB) and the invalid class (code is not one of ON, QU, NB).
- If the requirements state that a particular input item match a set of values and each case will be dealt with differently, identify a valid equivalence class for each element and only one invalid class for values outside the set. For example, if a discount code must be input as P for a preferred customer, R for a standard reduced rate, or N for none, and if each case is treated differently, identify: the valid class code = P, the valid class code = R, the valid class code = N, the invalid class code is not one of P, R, N.
- If an element of an equivalence class will be handled differently than the others, divide the equivalence class to create an equivalence class with only these elements and an equivalence class with none of these elements. For example, a bank account balance may be from 0 to Rs 10 lakh and balances of Rs 1,000 or more are not subject to service charges. Identify: the valid class: (0 ≤ balance < Rs 1,000), i.e. balance is between 0 and Rs 1,000 – not including Rs 1,000;

the valid class: (Rs 1,000 ≤ balance ≤ Rs 10 lakh, i.e. balance is between Rs 1,000 and Rs 10 lakh inclusive the invalid class: (balance < 0) the invalid class: (balance > Rs 10 lakh).

2. IDENTIFYING THE TEST CASES: -

A few guidelines are given below to identify test cases through generated equivalence classes:

- Assign a unique identification number to each equivalence class.
- Write a new test case covering as many of the uncovered valid equivalence classes as possible, until all valid equivalence classes have been covered by test cases.
- 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. The reason that invalid cases are covered by individual test cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states 'Enter type of toys (Automatic, Mechanical, Soft toy) and amount (1–10000)', the test case [ABC 0] expresses two error (invalid inputs) conditions (invalid toy type and invalid amount) will not demonstrate the invalid amount test case, hence the program may produce an output 'ABC is unknown toy type' and not bother to examine the remainder of the input.

Example 4.5

A program reads three numbers, A, B, and C, with a range [1, 50] and prints the largest number. Design test cases for this program using equivalence class testing technique.

Solution

1. First we partition the domain of input as valid input values and invalid values, getting the following classes:

$$I_1 = \{<A,B,C> : 1 \le A \le 50\}$$
$$I_2 = \{<A,B,C> : 1 \le B \le 50\}$$
$$I_3 = \{<A,B,C> : 1 \le C \le 50\}$$
$$I_4 = \{<A,B,C> : A < 1\}$$
$$I_5 = \{<A,B,C> : A > 50\}$$
$$I_6 = \{<A,B,C> : B < 1\}$$
$$I_7 = \{<A,B,C> : B > 50\}$$
$$I_8 = \{<A,B,C> : C < 1\}$$
$$I_9 = \{<A,B,C> : C > 50\}$$

Now the test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class.

The test cases are shown below:

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I_1, I_2, I_3
2	0	13	45	Invalid input	I_4
3	51	34	17	Invalid input	I_5
4	29	0	18	Invalid input	I_6
5	36	53	32	Invalid input	I_7
6	27	42	0	Invalid input	I_8
7	33	21	51	Invalid input	I_9

2. We can derive another set of equivalence classes based on some possibilities for three integers, *A*, *B*, and *C*. These are given below:

$$I_1 = \{<A,B,C> : A > B,\ A > C\}$$
$$I_2 = \{<A,B,C> : B > A,\ B > C\}$$
$$I_3 = \{<A,B,C> : C > A,\ C > B\}$$
$$I_4 = \{<A,B,C> : A = B,\ A \neq C\}$$
$$I_5 = \{<A,B,C> : B = C,\ A \neq B\}$$
$$I_6 = \{<A,B,C> : A = C,\ C \neq B\}$$
$$I_7 = \{<A,B,C> : A = B = C\}$$

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Example 4.6: -

A program determines the next date in the calendar. Its input is entered in the form of <ddmmyyyy> with the following range:

$1 \leq mm \leq 12$

$1 \leq dd \leq 31$

$1900 \leq yyyy \leq 2025$

Its output would be the next date or an error message 'invalid date.' Design test cases using equivalence class partitioning method.

Solution

First we partition the domain of input in terms of valid input values and invalid values, getting the following classes:

$$I_1 = \{<m, d, y> : 1 \leq m \leq 12\}$$
$$I_2 = \{<m, d, y> : 1 \leq d \leq 31\}$$
$$I_3 = \{<m, d, y> : 1900 \leq y \leq 2025\}$$
$$I_4 = \{<m, d, y> : m < 1\}$$
$$I_5 = \{<m, d, y> : m > 12\}$$
$$I_6 = \{<m, d, y> : d < 1\}$$
$$I_7 = \{<m, d, y> : d > 31\}$$
$$I_8 = \{<m, d, y> : y < 1900\}$$
$$I_9 = \{<m, d, y> : y > 2025\}$$

The test cases can be designed from the above derived classes, taking one test case from each class such that the test case covers maximum valid input classes, and separate test cases for each invalid class. The test cases are shown below:

Test case ID	mm	dd	yyyy	Expected result	Classes covered by the test case
1	5	20	1996	21-5-1996	I_1, I_2, I_3
2	0	13	2000	Invalid input	I_4
3	13	13	1950	Invalid input	I_5
4	12	0	2007	Invalid input	I_6
5	6	32	1956	Invalid input	I_7
6	11	15	1899	Invalid input	I_8
7	10	19	2026	Invalid input	I_9

Example 4.7: -

A program takes an angle as input within the range [0, 360] and determines in which quadrant the angle lies. Design test cases using equivalence class partitioning method.

Solution

1. First we partition the domain of input as valid and invalid values, getting the following classes:

$$I_1 = \{<\text{Angle}> : 0 \le \text{Angle} \le 360\}$$
$$I_2 = \{<A,B,C> : \text{Angle} < 0\}$$
$$I_3 = \{<A,B,C> : \text{Angle} > 0\}$$

The test cases designed from these classes are shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	I_1
2	–1	Invalid input	I_2
3	361	Invalid input	I_3

2. The classes can also be prepared based on the output criteria as shown below:

$$O_1 = \{<\text{Angle}>: \text{First Quadrant, if } 0 \le \text{Angle} \le 90\}$$
$$O_2 = \{<\text{Angle}>: \text{Second Quadrant, if } 91 \le \text{Angle} \le 180\}$$
$$O_3 = \{<\text{Angle}>: \text{Third Quadrant, if } 181 \le \text{Angle} \le 270\}$$
$$O_4 = \{<\text{Angle}>: \text{Fourth Quadrant, if } 271 \le \text{Angle} \le 360\}$$
$$O_5 = \{<\text{Angle}>: \text{Invalid Angle}\};$$

However, O_5 is not sufficient to cover all invalid conditions this way. Therefore, it must be further divided into equivalence classes as shown below:

$$O_{51} = \{<\text{Angle}>: \text{Invalid Angle, if Angle} < 0\}$$
$$O_{52} = \{<\text{Angle}>: \text{Invalid Angle, if Angle} > 360\}$$

Now the test cases can be designed from the above derived classes as shown below:

Test Case ID	Angle	Expected results	Classes covered by the test case
1	50	I Quadrant	O_1
2	135	II Quadrant	O_2
3	250	III Quadrant	O_3
4	320	IV Quadrant	O_4
5	370	Invalid angle	O_{51}
6	–1	Invalid angle	O_{52}

White-Box Testing Techniques

White-box testing is another effective testing technique in dynamic testing. It is also known as glass-box testing, as everything that is required to implement the software is visible. The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as *structural* or *development* testing.

In white-box testing, structure means the logic of the program which has been implemented in the language code. The intention is to test this logic so that required results or functionalities can

be achieved. Thus, white-box testing ensures that the internal parts of the software are adequately tested.

NEED OF WHITE-BOX TESTING: -

The supporting reasons for white-box testing are given below:

1. In fact, white-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software. Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques. Thus, white-box testing is not an alternative but an essential stage.
2. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.
3. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
4. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
5. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors.

LOGIC COVERAGE CRITERIA: -

Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered. Therefore the intention in white-box testing is to cover the whole logic. The basic forms of logic coverage are

1. Statement Coverage: -

The first kind of logic coverage can be identified in the form of statements. It is assumed that if all the statements of the module are executed once, every bug will be notified.

Consider the following code segment

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

Figure 5.1 Sample code

If we want to cover every statement in the above code, then the following test cases must be designed:

Test case 1: $x = y = n$, where n is any number

Test case 2: $x = n$, $y = n'$, where n and n' are different numbers.

Test case 1 just skips the while loop and all loop statements are not executed. Considering test case 2, the loop is also executed. However, every statement inside the loop is not executed. So two more cases are designed:

Test case 3: $x > y$

Test case 4: $x < y$

These test cases will cover every statement in the code segment, however statement coverage is a poor criteria for logic coverage. We can see that test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected. Thus, *statement coverage is a necessary but not sufficient criteria for logic coverage.*

2. Decision or Branch Coverage: -

Branch coverage states that each decision takes on all possible outcomes (True or False) at least once. In other words, each branch direction must be traversed at least once. In the previous sample code shown in Figure 5.1, while and if statements have two outcomes: True and False. So test cases must be designed such that both outcomes for while and if statements are tested. The test cases are designed as:

$$\text{Test case 1: } x == y$$
$$\text{Test case 2: } x \; != y$$
$$\text{Test case 3: } x < y$$
$$\text{Test case 4: } x > y$$

3. Condition Coverage: -

Condition coverage states that each condition in a decision takes on all possible outcomes at least once. For example, consider the following statement:

$$while\ ((I \leq 5)\ \&\&\ (J < COUNT))$$

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes. The following test cases are designed:

$$\text{Test case 1: } I \leq 5, J < \text{COUNT}$$
$$\text{Test case 2: } I < 5, J > \text{COUNT}$$

4. Decision/condition Coverage: -

Condition coverage in a decision does not mean that the decision has been covered. If the decision

$$if\ (A\ \&\&\ B)$$

is being tested, the condition coverage would allow one to write two test cases:

$$\text{Test case 1: } A \text{ is True, } B \text{ is False.}$$
$$\text{Test case 2: } A \text{ is False, } B \text{ is True.}$$

But these test cases would not cause the THEN clause of the IF to execute (i.e. execution of decision). The obvious way out of this dilemma is a criterion called decision/condition coverage. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

5. Multiple condition coverage: -

In case of multiple conditions, even decision/ condition coverage fails to exercise all outcomes of all conditions. The reason is that we have considered all possible outcomes of each condition in the decision, but we have not taken all combinations of different multiple conditions. Certain conditions mask other conditions. For example, if an AND condition is False, none of the subsequent conditions in the expression will be evaluated. Similarly, if an OR condition is True, none of the subsequent conditions will be evaluated. Thus, condition coverage and decision/condition coverage need not necessarily uncover all the errors.

Therefore, multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once. Thus, as in decision/condition coverage, all possible combinations of multiple conditions should be considered. The following test cases can be there:

Test case 1: $A = \text{True}, B = \text{True}$
Test case 2: $A = \text{True}, B = \text{False}$
Test case 3: $A = \text{False}, B = \text{True}$
Test case 4: $A = \text{False}, B = \text{False}$

BASIS PATH TESTING

Basis path testing is the oldest structural testing technique. The technique is based on the control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing. Path coverage is a more general criterion as compared to other coverage criteria and useful for detecting more errors. But the problem with path criteria is that programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths. Some criteria should be devised such that selected paths are executed for maximum coverage of logic. Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

The guidelines for effectiveness of path testing are discussed below:

1. Path testing is based on control structure of the program for which flow graph is prepared.
2. Path testing requires complete knowledge of the program's structure.
3. Path testing is closer to the developer and used by him to test his module.
4. The effectiveness of path testing gets reduced with the increase in size of software under test.
5. Choose enough paths in a program such that maximum logic coverage is achieved.

1 CONTROL FLOW GRAPH: -

The control flow graph is a graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V. Based on the concepts of directed graph, following notations are used for a flow graph:

- **Node: -** It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- **Edges or links: -** They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- **Decision node: -** A node with more than one arrow leaving it is called a decision node.
- **Junction node: -** A node with more than one arrow entering it is called a junction.
- **Regions: -** Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

2 FLOW GRAPH NOTATIONS FOR DIFFERENT PROGRAMMING CONSTRUCTS: -

Since a flow graph is prepared on the basis of control structure of a program, some fundamental graphical notations are shown here (see Fig. 5.2) for basic programming constructs.

Using the above notations, a flow graph can be constructed. Sequential statements having no conditions or loops can be merged in a single node. That is why, the flow graph is also known as ***decision-to-decision-graph*** or ***DD graph***.

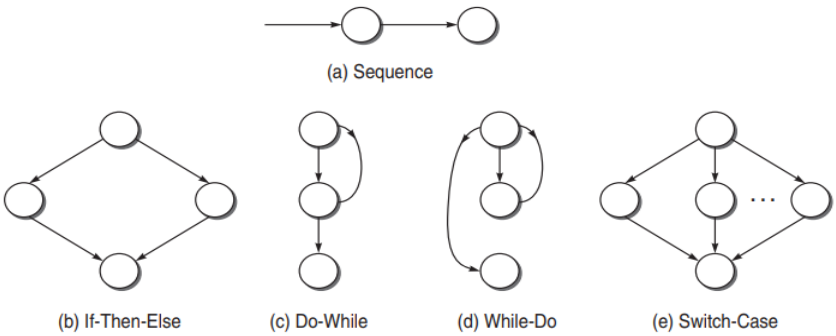


Figure 5.2

3 PATH TESTING TERMINOLOGY: -

Path: - A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times.

Segment: - Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junctionprocess-decision, decision-process-junction, and decision-process-decision). A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.

Path segment: - A path segment is a succession of consecutive links that belongs to some path.

Length of a path: - The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path. An alternative way to measure the length of a path is by the number of nodes traversed. This method has some analytical and theoretical benefits. If programs are assumed to have an entry and an exit node, then the number of links traversed is just one less than the number of nodes traversed.

Independent path: - An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined.

4 CYCLOMATIC COMPLEXITY: -

McCabe has given a measure for the logical complexity of a program by considering its control flow graph. His idea is to measure the complexity by considering the number of paths in the control graph of the program. But even for simple programs, if they contain at least one cycle, the number of paths is infinite. Therefore, he considers only independent paths.

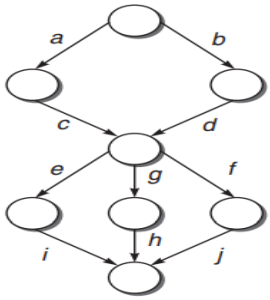


Figure 5.3 Sample graph

In the graph shown in Figure 5.3, there are six possible paths: acei, acgh, acfh, bdei, bdgh, bdfj.

In this case, we would see that, of the six possible paths, only four are independent, as the other two are always a linear combination of the other four paths. Therefore, the number of independent paths is 4. In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

where n is the number of nodes and e is the number of arcs/edges.

However, it may be possible that the graph is not strongly connected. In that case, the above formula does not fi t. Therefore, to make the graph strongly connected, we add an arc from the last node to the fi rst node of the graph. In this way, the fl ow graph becomes a strongly connected graph. But by doing this, we increase the number of arcs by 1 and therefore, the number of independent paths (as a function of the original graph) is given by

$$V(G) = e - n + 2$$

This is called the cyclomatic number of a program. We can calculate the cyclomatic number only by knowing the number of choice points (decision nodes) d in the program. It is given by

$$V(G) = d + 1$$

This is also known as *Miller's theorem*. We assume that a *k-way* decision point contributes for k−1 choice points.

The program may contain several procedures also. These procedures can be represented as separate flow graphs. These procedures can be called from any point but the connections for calling are not shown explicitly. The cyclomatic number of the whole graph is then given by the sum of the numbers of each graph. It is easy to demonstrate that, if p is the number of graphs and e and n are referred to as the whole graph, the cyclomatic number is given by

$$V(G) = e - n + 2p$$

And Miller's theorem becomes

$$V(G) = d + p$$

Formulae Based on Cyclomatic Complexity

Based on the cyclomatic complexity, the following formulae are being summarized.

Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G)$ = number of regions in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else

When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1$, where k is the number of arrows leaving the node.

Calculating the cyclomatic complexity number of the program having many connected components Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

$$V(G) = V(P) + V(X) + V(Y) + V(Z)$$

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$$V(G) = e - n + 2P$$

The complexity number derived collectively will be same as calculated above. Thus,

$$V(P \cup X \cup Y \cup Z) = V(P) + V(X) + V(Y) + V(Z)$$

Guidelines for Basis Path Testing: -

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

The following steps should be followed for designing test cases using path testing:

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
- The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

Example 1

Consider the following program segment:

```
main()
{
    int number, index;
1.  printf("Enter a number");
2.  scanf("%d, &number);
3.  index = 2;
4.  while(index <= number - 1)
5.  {
6.      if (number % index == 0)
7.      {
8.          printf("Not a prime number");
9.          break;
10.     }
11.     index++;
12.     }
13.    if(index == number)
14.        printf("Prime number");
15. } //end main
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

For a DD graph, the following actions must be done:

- Put the line numbers on the execution statements of the program, as shown in Fig. 5.4. Start numbering the statements after declaring the variables, if no variables have been initialized. Otherwise, start from the statement where a variable has been initialized.

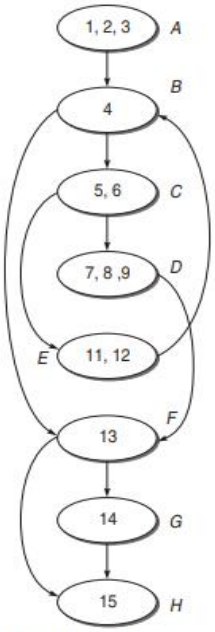


Figure 5.4 DD graph for Example 5.1

- Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.
- Put the edges between the nodes according to their flow of execution.
- Put alphabetical numbering on each node like *A*, *B*, etc.

The DD graph of the program is shown in Figure 5.4.

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * p$

$= 10 - 8 + 2$

$= 4$

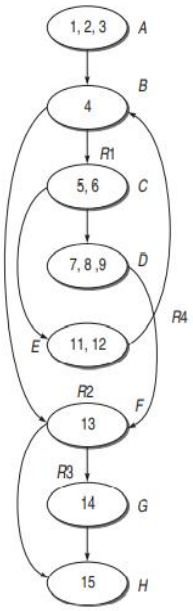


Figure 5.5 DD graph for Example 5.1 showing regions

(ii) $V(G) =$ Number of predicate nodes + 1

$= 3$ (Nodes *B*, *C*, and *F*) + 1

$= 4$

(iii) $V(G)$ = Number of regions
$= 4(R1, R2, R3, R4)$

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) A-B-F-H
(ii) A-B-F-G-H
(iii) A-B-C-E-B-F-G-H
(iv) A-B-C-D-F-H

(d) Test case design from the list of independent paths

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Example 2

Consider the following program that reads in a string and then checks the type of each character.

```
main()
{
    char string [80];
    int index;
1.  printf("Enter the string for checking its characters");
2.  scanf("%s", string);
3.  for(index = 0; string[index] != '\0'; ++index)   {
4.      if((string[index] >= '0' && (string[index] <='9'
5.                  printf("%c is a digit", string[index]);
6.      else if ((string[index] >= 'A' && string[index] <'Z'))  ||
                ((string[index] >= 'a' && (string[index] <'z')))
7.              printf("%c is an alphabet", string[index]);
8.      else
9.              printf("%c is a special character", string[index]);
10.     }
11. }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.6.

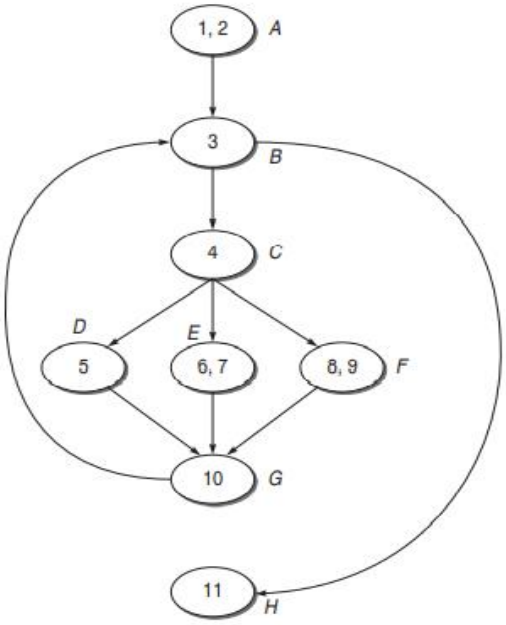


Figure 5.6 DD graph for Example 5.2

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2 * P$
$= 10 - 8 + 2$
$= 4$

(ii) $V(G)$ = Number of predicate nodes + 1
= 3 (Nodes B, C) + 1
= 4

Node C is a multiple IF-THEN-ELSE, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes
= Number of links out of main node − 1
= 3 − 1 = 2 (For node C)

(iii) $V(G)$ = Number of regions
= 4

(c) Independent paths

Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(i) *A-B-H*
(ii) *A-B-C-D-G-B-H*
(iii) *A-B-C-E-G-B-H*
(iv) *A-B-C-F-G-B-H*

(d) Test case design from the list of independent paths

Test Case ID	Input Line	Expected Output	Independent paths covered by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H

Example 3: -

Consider the following program:

```
    main()
    {
        char chr;
1.      printf ("Enter the special character\n");
2.      scanf (%c", &chr);
3.      if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
           (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
           (chr != 56) && (chr != 57)
4.      {
5.          switch(chr)

6.            {
7.            Case '*': printf("It is a special character");
8.            break;
9.            Case '#': printf("It is a special character");
10.           break;
11.           Case '@': printf("It is a special character");
12.           break;
13.           Case '!':  printf("It is a special character");
14.           break;
15.           Case '%': printf("It is a special character");
16.           break;
17.           default : printf("You have not entered a special character");
18.           break;
19.           }// end of switch
20.     } // end of If
21.     else
22.         printf("You have not entered a character");
23. } // end of main()
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all the methods.
(c) List all independent paths.
(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.7.

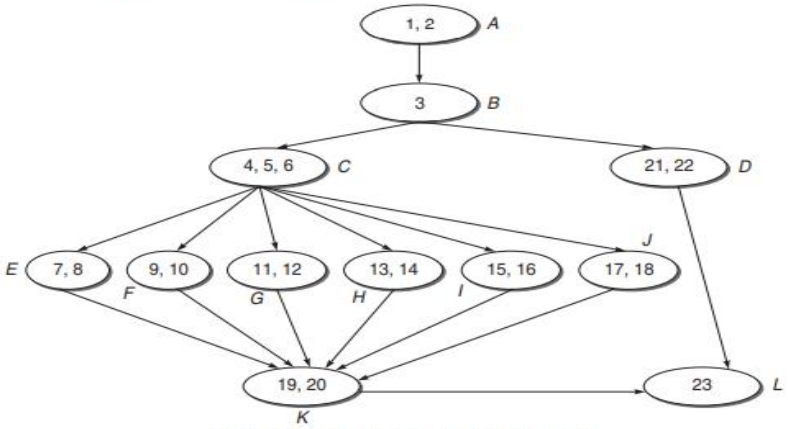


Figure 5.7 DD graph for Example 5.3

(b) Cyclomatic complexity

(i) $V(G) = e - n + 2p$
$= 17 - 12 + 2$
$= 7$

(ii) $V(G)$ = Number of predicate nodes + 1
= 2 (Nodes B, C) + 1
= 7

Node C is a switch-case, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes
= Number of links out of main node −1
= 6 − 1 = 5 (For node C)

(iii) $V(G)$ = Number of regions = 7

(c) Independent paths

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

1. *A-B-D-L*
2. *A-B-C-E-K-L*
3. *A-B-C-F-K-L*
4. *A-B-C-G-K-L*
5. *A-B-C-H-K-L*
6. *A-B-C-I-K-L*
7. *A-B-C-J-K-L*

(d) Test Case Design from the list of Independent Paths

Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

Example 4: -

Consider a program to arrange numbers in ascending order from a given list of N numbers.

```
1.  main()
    {
    int num,small;
    int i,j,sizelist,list[10],pos,temp;
    clrscr();
    printf("\nEnter the size of list :\n ");
    scanf("%d",&sizelist);
2.  for(i=0;i<sizelist;i++)
3.  {
    printf("\nEnter the number");
    scanf ("%d",&list[i]);
    }
4.  for(i=0;i<sizelist;i++)
5.  {
    small=list[i];
    pos=i;
6.  for(j=i+1;j<sizelist;j++)
    {
7.  if(small>list[j])
8.  {
    small=list[j];
    pos=j;
    }
9.  }
10. temp=list[i];
    list[i]=list[pos];
    list[pos]=temp;
11. }
12. printf("\nList of the numbers in ascending order : ");
13. for(i=0;i<sizelist;i++)
14. printf("\n%d",list[i]);
15. getch();
    }
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

Solution

(a) DD graph

The DD graph of the program is shown in Fig. 5.8.

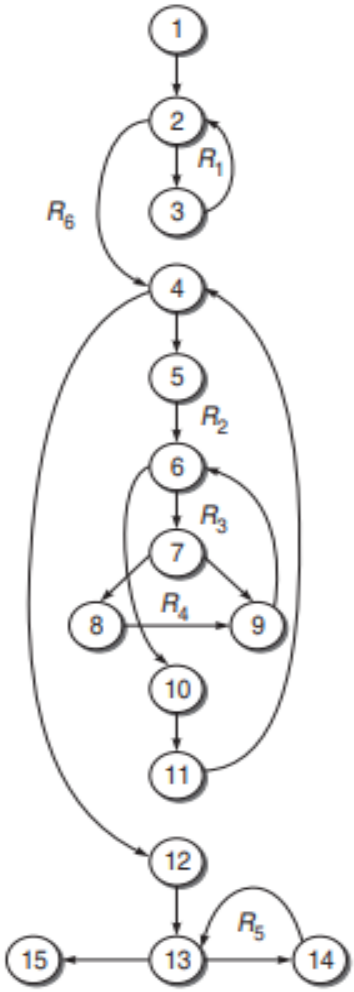


Figure 5.8 DD graph for Example 5.4

(b) Cyclomatic complexity

1. $V(G) = e - n + 2p$
   $= 19 - 15 + 2$
   $= 6$

2. $V(G)$ = Number of predicate nodes + 1
   $= 5 + 1$
   $= 6$

3. $V(G)$ = Number of regions
   $= 6$

(c) Independent paths

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

1. 1-2-3-2-4-5-6-7-8-9-6-10-11-4-12-13-14-13-15
2. 1-2-3-2-4-5-6-7-9-6-10-11-4-12-13-14-13-15
3. 1-2-3-2-4-5-6-10-11-4-12-13-14-13-15
4. 1-2-3-2-4-12-13-14-13-15 (path not feasible)
5. 1-2-4-12-13-15
6. 1-2-3-2-4-12-13-15 (path not feasible)

(d) Test case design from the list of independent paths

Test Case ID	Input	Expected Output	Independent path covered by Test Case
1	Sizelist = 5 List[] = {17,6,7,9,1}	1,6,7,9,17	1
2	Sizelist = 5 List[] = {1,3,9,10,18}	1,3,9,10,18	2
3	Sizelist = 1 List[] = {1}	1	3
4	Sizelist = 0	blank	blank

Example 5: -

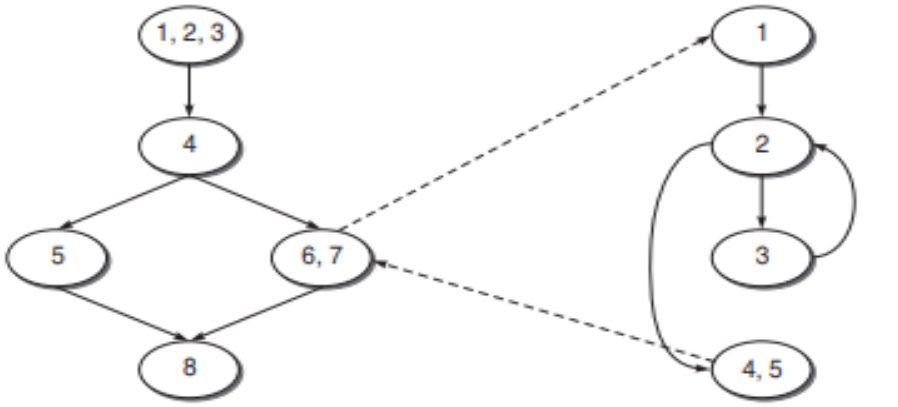
Consider the program for calculating the factorial of a number. It consists of main() program and the module fact(). Calculate the individual cyclomatic complexity number for main() and fact() and then, the cyclomatic complexity for the whole program.

```
        main()
        {
            int number;
            int fact();
1.          clrscr();
2.          printf("Enter the number whose factorial is to be found out");

3.          scanf("%d", &number);
4.          if(number <0)
5.             printf("Facorial cannot be defined for this number);
6.          else
7.             printf("Factorial is %d", fact(number));
8.      }

        int fact(int number)
        {
            int index;
1.          int product =1;
2.          for(index=1; index<=number; index++)
3.             product = product * index;
4.          return(product);
5.      }
```

Solution

DD graph

The DD graph of the program is shown in Fig. 5.9



(a) Flow graph for main () (b) Flow graph for fact ()

Figure 5.9

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p$
$= 5 - 5 + 2$
$= 2$

(b) $V(M) = d + p$
$= 1 + 1$
$= 2$

(c) $V(M) =$ Number of regions $= 2$

Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

(b) $V(R) =$ Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R) =$ Number of regions $= 2$

Cyclomatic complexity of the whole graph considering the full program

(a) $V(G) = e - n + 2p$
$= 9 - 9 + 2 \times 2$
$= 4$
$= V(M) + V(R)$

(b) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(c) $V(G) =$ Number of regions
$= 4$
$= V(M) + V(R)$

5. APPLICATIONS OF PATH TESTING: -

Path testing has been found better suitable as compared to other testing methods. Some of its applications are discussed below.

Thorough testing / More coverage: - Path testing provides us the best code coverage, leading to a thorough testing. Path coverage is considered better as compared to statement or branch coverage methods because the basis path set provides us the number of test cases to be covered which ascertains the number of test cases that must be executed for full coverage. Generally, branch coverage or other criteria gives us less number of test cases as compared to path testing. Cyclomatic complexity along with basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique.

Unit testing: - Path testing is mainly used for structural testing of a module. In unit testing, there are chances of errors due to interaction of decision outcomes or control flow problems which are hidden with branch testing. Since each decision outcome is tested independently, path testing uncovers these errors in module testing and prepares them for integration.

Integration testing: - Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules. Path testing analyses all the paths on the interface and explores all the errors.

Maintenance testing: - Path testing is also necessary with the modified version of the software. If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing. In any case, path testing is still able to detect any security threats on the interface with the called modules.

Testing effort is proportional to complexity of the software: - Cyclomatic complexity number in basis path testing provides the number of tests to be executed on the software based on the complexity of the software. It means the number of tests derived in this way is directly proportional to the complexity of the software. Thus, path testing takes care of the complexity of the software and then derives the number of tests to be carried out.

Basis path testing effort is concentrated on error-prone software: - Since basis path testing provides us the number of tests to be executed as a measure of software cyclomatic complexity, the cyclomatic number signifies that the testing effort is only on the error-prone part of the software, thus minimizing the testing effort.

LOOP TESTING

Loop testing can be viewed as an extension to branch coverage. Loops are important in the software from the testing viewpoint. If loops are not tested properly, bugs can go undetected. Loop testing can be done effectively while performing development testing (unit testing by the developer) on a module. Sufficient test cases should be designed to test every loop thoroughly.

There are four different kinds of loops. How each kind of loop is tested, is discussed below.

1. **Simple loops: -** Simple loops mean, we have a single loop in the flow, as shown in Fig. 5.9.

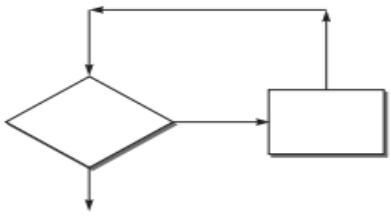


Figure 5.9 (a)

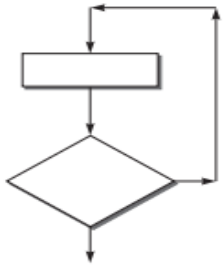


Figure 5.9 (b)

The following test cases should be considered for simple loops while testing them:

- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min−1, max−1, max, and max+1 number of iterations through the loop.

2. **Nested loops: -** When two or more loops are embedded, it is called a nested loop, as shown in Fig. 5.10. If we have nested loops in the program, it becomes difficult to test. If we adopt the approach of simple tests to test the nested loops, then the number of possible test cases grows geometrically. Thus, the strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered.

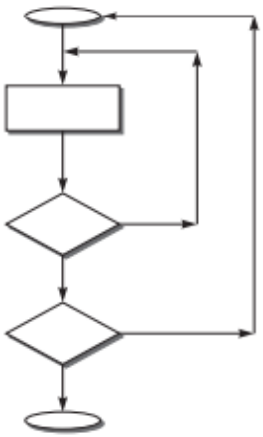


Figure 5.10 Nested loops

3. **Concatenated loops: -** The loops in a program may be concatenated (Fig. 5.11). Two loops are concatenated if it is possible to reach one after exiting the other, while still on a path from entry to exit. If the two loops are not on the same path, then they are not concatenated. The two loops on the same path may or may not be independent. If the loop control variable for one loop is used for another loop, then they are concatenated, but nested loops should be treated like nested only.

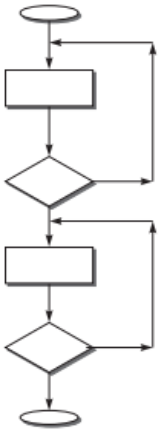


Figure 5.11 Concatenated loops

4. **Unstructured loops: -** This type of loops is really impractical to test and they must be redesigned or at least converted into simple or concatenated loops.

DATA FLOW TESTING

In path coverage, the stress was to cover a path using statement or branch coverage. However, data and data integrity is as important as code and code integrity of a module. We have checked every possibility of the control flow of a module. But what about the data flow in the module? Has every data object been initialized prior to use? Have all defined data objects been used for something? These questions can be answered if we consider data objects in the control flow of a module.

Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors. Errors may be unintentionally introduced in a program by programmers. For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate. For example

```
int a;
if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used. For example, if an out-of-scope data is being used in a computation, then it is a bug. There may be several patterns like this which indicate data anomalies.

To examine the patterns, the control flow graph of a program is used. This test strategy selects the paths in the module's control flow such that various sequences of data objects can be chosen. The major focus is on the points at which the data receives values and the places at which the data initialized has been referenced. Thus, we have to choose enough paths in the control flow to ensure that every data is initialized before use and all the defined data have been used somewhere. Data flow testing closely examines the state of the data in the control flow graph, resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

1. STATE OF A DATA OBJECT: -

A data object can be in the following states:

i. **Defined (d): -** A data object is called defined when it is initialized, i.e. when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, and something is pushed onto the stack, a record written, and so on.

ii. **Killed/Undefined/Released (k): -** When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.

iii. **Usage (u): -** When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either *computational use* (c-use) or *predicate use* (p-use).

2. DATA-FLOW ANOMALIES: -

Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code. An anomaly is denoted by a two-character sequence of actions. For example, 'dk' means a variable is defined and killed without any use, which is a potential bug. There are nine possible two-character combinations out of which only four are data anomalies, as shown in Table 5.1.

Table 5.1 Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur. In addition to the above two-character data anomalies, there may be single-character data anomalies also. To represent these types of anomalies, we take the following conventions:

~x : indicates all prior actions are not of interest to x.

x~ : indicates all post actions are not of interest to x.

All single-character data anomalies are listed in Table 5.2.

Table 5.2 Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	First Use	Data is used without defining it. Potential bug.
~k	First Kill	Data is killed before defining it. Potential bug.
D~	Define last	Potential bug.
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

3. TERMINOLOGY USED IN DATA FLOW TESTING: -

Some terminology, which will help in understanding all the concepts related to data-flow testing, is being discussed. Suppose P is a program that has a graph G(P) and a set of variables V. The graph has a single entry and exit node.

a) Definition node: - Defining a variable means assigning value to a variable for the very first time in a program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.

b) Usage node: - It means the variable has been used in some statement of the program. Node n that belongs to G(P) is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n. For example, output statements, assignment statements (right), conditional statements, loop control statements, etc.

A usage node can be of the following two types:

i. **Predicate Usage Node:** If usage node n is a predicate node, then n is a predicate usage node.

ii. **Computation Usage Node:** If usage node n corresponds to a computation statement in a program other than predicate, then it is called a computation usage node.

c) Loop-free path segment: - It is a path segment for which every node is visited once at most.

d) Simple path segment: - It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.

e) Definition-use path (du-path): - A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node.

f) Definition-clear path (dc-path): - A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v.

The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons. Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths. The application of data flow testing can be extended to debugging where a testing person finds the problematic areas in code to trace the bug. So the du-paths which are not dc-paths need more attention.

4. STATIC DATA FLOW TESTING: -

With static analysis, the source code is analysed without executing it. Let us consider an example of an application given below.

Example 1: -

Consider the program given below for calculating the gross salary of an employee in an organization. If his basic salary is less than Rs 1500, then HRA = 10% of basic salary and DA = 90% of the basic. If his salary is either equal to or above Rs 1500, then HRA = Rs 500 and DA = 98% of the basic salary. Calculate his gross salary.

```
    main()
    {
1.      float bs, gs, da, hra = 0;
2.      printf("Enter basic salary");
3.      scanf("%f", &bs);
4.      if(bs < 1500)
5.      {
6.          hra = bs * 10/100;
7.          da = bs * 90/100;
8.      }
9.      else
10.     {
11.         hra = 500;
12.         da = bs * 98/100;
13.     }
14.     gs = bs + hra + da;
15.     printf("Gross Salary = Rs. %f", gs);
16. }
```

Find out the define-use-kill patterns for all the variables in the source code of this application.

Solution

For variable 'bs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'gs', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

From the above static analysis, it was observed that static data flow testing for the variable 'hra' discovered one bug of double definition in line number 1.

Static Analysis is not enough: -

It is not always possible to determine the state of a data variable by just static analysis of the code. For example, if the data variable in an array is used as an index for a collection of data elements, we cannot determine its state by static analysis. Or it may be the case that the index is generated dynamically during execution, therefore we cannot guarantee what the state of the array element is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence, not completely anomalous. Thus, all anomalies using static analysis cannot be determined and this problem is provably unsolvable.

5. DYNAMIC DATA FLOW TESTING: -

Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code. The test cases are designed in such a way that every definition of data variable to each of its use is traced and every use is traced to each of its definition. Various strategies are employed for the creation of test cases. All these strategies are defined below.

All-du Paths (ADUP): - It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy, since it is a superset of all other data flow testing strategies. Moreover, this strategy requires the maximum number of paths for testing.

All-uses (AU): - This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

All-p-uses/Some-c-uses (APU + C): - This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use. If there are definitions of the variable with no p-use following it, then add computational use (c-use) test cases as required to cover every definition.

All-c-uses/Some-p-uses (ACU + P): - This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every

computational use. If there are definitions of the variable with no c-use following it, then add predicate use (c-use) test cases as required to cover every definition.

All-Predicate-Uses (APU): - It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-Computational-Uses (ACU): - It is derived from the strategy ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-Definition (AD): - It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

Example 1: -

Consider the program given below. Draw its control flow graph and data flow graph for each variable used in the program, and derive data flow testing paths with all the strategies discussed above.

```
    main()
    {
    int work;
 0. double payment =0;
 1. scanf("%d", work);
 2. if (work > 0) {
 3.     payment = 40;
 4. if (work > 20)
 5. {
 6.     if(work <= 30)
 7.         payment = payment + (work - 25) * 0.5;
 8.     else
 9.     {
10.         payment = payment + 50 + (work -30) * 0.1;
11.             if (payment >= 3000)
12.                 payment = payment * 0.9;
13.     }
14. }
15. }
16. printf("Final payment", payment);
```

Solution

Figure 5.12 shows the control flow graph for the given program.

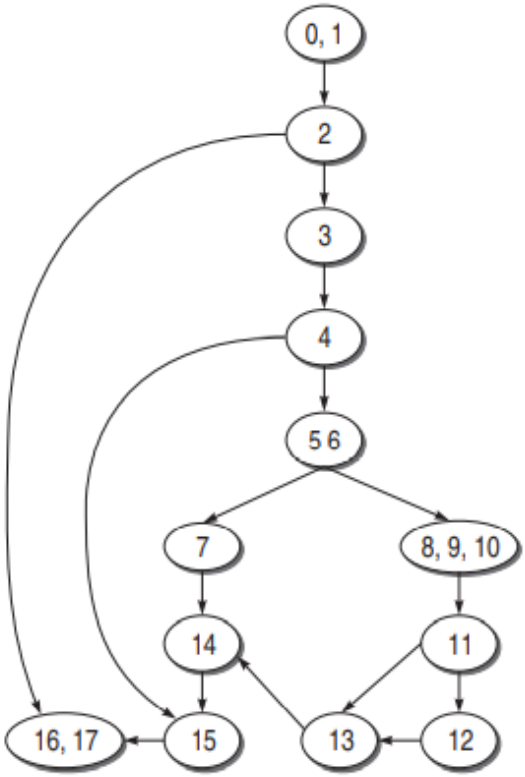


Figure 5.12 DD graph for Example 1

Figure 5.13 shows the data flow graph for the variable 'payment'.

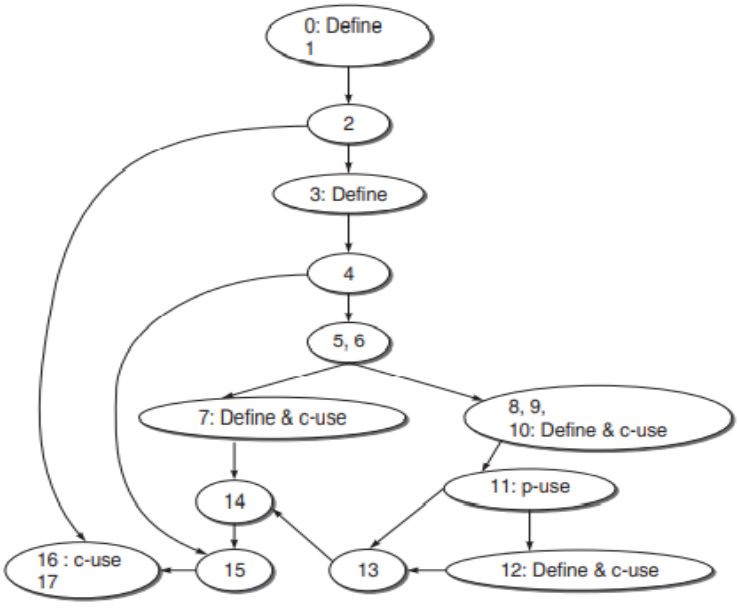


Figure 5.13 Data flow graph for 'payment'

Figure 5.14 shows the data flow graph for the variable 'work'.

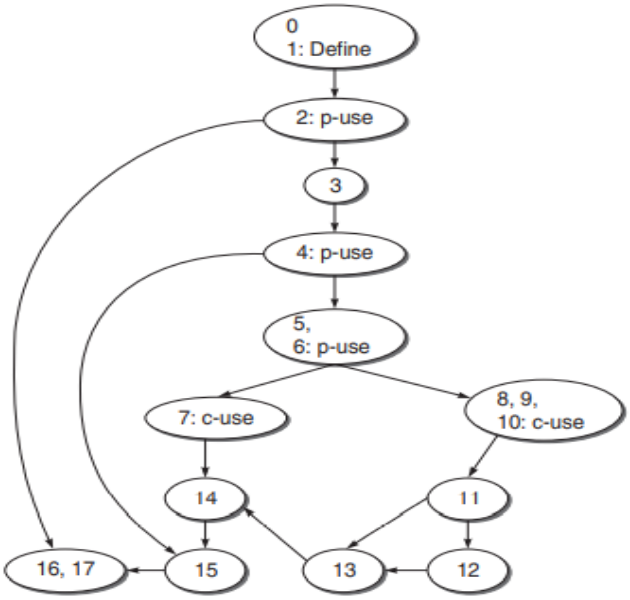


Figure 5.14 Data flow graph for variable 'work'

Prepare a list of all the definition nodes and usage nodes for all the variables in the program.

Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

Data flow testing paths for each variable are shown in Table 5.3.

Table 5.3 Data flow testing paths

Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses (APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6
All c-uses (ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All-p-uses / Some-c-uses (APU + C)	0-1-2-3-4-5-6-8-9-10-11 10-11-12 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-8-9-10
All-c-uses / Some-p-uses (ACU + P)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16 0-1-2-3-4-5-6-8-9-10-11	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10 1-2-3-4-5-6

All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11 0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All Definitions (AD)	0-1-2-16 3-4-5-6-7 7-14-15-16 10-11 12-13-14-15-16	1-2

6. ORDERING OF DATA FLOW TESTING STRATEGIES: -

While selecting a test case, we need to analyse the relative strengths of various data flow testing strategies. Figure 5.15 depicts the relative strength of the data flow strategies. In this figure, the relative strength of testing strategies reduces along the direction of the arrow. It means that all-du-paths (ADPU) is the strongest criterion for selecting the test cases.

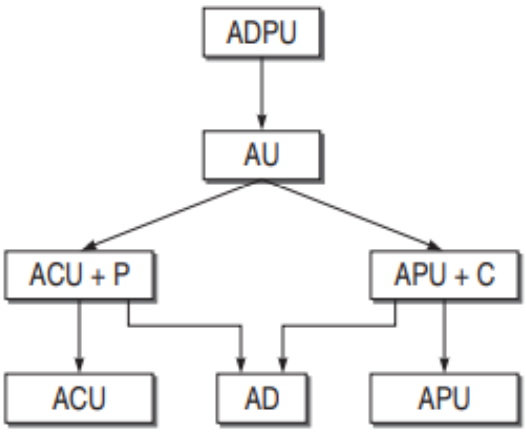


Figure 5.15 Data-flow testing strategies

REVIEW QUESTIONS: -

1. What are the types of errors detected by black-box testing?
2. Which type of testing is possible with BVA?
3. Which type of testing is possible with equivalence class partitioning?
4. A program calculates the GCD of three numbers in the range [1, 50]. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
5. A program takes as input a string (5–20 characters) and a single character and checks whether that single character is present in the string or not. Design test cases for this program using BVC, robust testing, and worst-case testing methods.
6. A program reads the data of employees in a company by taking the following inputs and prints them:

   Name of Employee (Max. 15 valid characters A–Z, a–z, space)
   Employee ID (10 characters)
   Designation (up to 20 characters)

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.

7. A mobile phone service provider uses a program that computes the monthly bill of customers as follows:

   Minimum Rs 300 for up to 120 calls
   Plus Re 1 per call for the next 70 calls
   Plus Rs 0.80 per call for the next 50 calls
   Plus Rs 0.40 per call for any call beyond 220 calls.

   Design test cases for this program using equivalence class testing technique.

8. A program reads players' records with the following detail and prints a team-wise list containing player name with their batting average:

   Player name (max. 30 characters)
   Team name (max. 20 characters)
   Batting average

   Design test cases for this program using BVC, robust testing, and worst-case testing methods.

9. A program takes as input three angles and determines the type of triangle. If all the three angles are less than 90, it is an acute angled triangle. If one angle is greater than 90, it is an obtuse angled triangle. If one angle is equal to 90, it is a right angled triangle.
   Design test cases for this program using equivalence class testing technique.
10. What is the need of white-box testing?
11. What are the different criteria for logic coverage?
12. What is basis path testing?
13. Distinguish between decision node and junction node?
14. What is an independent path?
15. What is the significance of cyclomatic complexity?
16. How do you calculate the number of decision nodes for switch-case?
17. How do you calculate the cyclomatic complexity number of the program having many connected components?
18. Consider the program.

```
#include <stdio.h>
main()
{
        int a,b, c,d;
        clrscr();
        printf("enter the two variables a,b");
        scanf("%d %d",&a,&b);
        printf("enter the option 1:Addition,
               2:subtraction,3:multiplication,4:division");
        scanf("%d",&c);
        switch(c)
        {
                case 1:d = a+b;
                printf("Addition of two no.=%d", d);
                break;
                case 2:d = a-b;
                printf("Subtraction of two no.=%d", d);
                break;
                case 3:d = a*b;
                printf("Multiplication of two no.=%d", d);
                break;
                case 4:d = a/b;
                printf("division of two no.=%d",d);
                break;
        }
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

19. Consider the program to find the greatest number:

```
#include <stdio.h>
main()
{
        float x,y,z;
        clrscr();
        printf("enter the three variables x,y,z");
        scanf("%f %f %f",&x,&y,&z);
        if(x > y)
        {
            if(x > z)
                printf("x is greatest");
            else
                printf("z is greatest");
        }
        else
        {
            if(y > z)
                printf("y is greatest");
            else
                printf("z is greatest");
        }
    getch();
}
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

20. Consider the following program which multiplies two matrices:

```
#include <stdio.h>
main()
{
int a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE], i, j, k, row1,
colm1, row2, colm2;

printf("Enter the order of first matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row1, colm1);
printf("Enter the order of second matrix <= %d %d \n", SIZE, SIZE);
scanf("%d%d",&row2, colm2);
if(colm1==row2)
```

```
        {
        printf("Enter first matrix");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colml; j++)
                scanf("%d", &a[i][j]);
                }
        printf("Enter second matrix");
        for(i=0; i<row2; i++)
                {
                for(j=0; j<colm2; j++)
                scanf("%d", &b[i][j]);
                }
        printf("Multiplication of two matrices is");
        for(i=0; i<row1; i++)
                {
                for(j=0; j<colm1; j++)
                        {
                        c[i] [j] = 0;
                        for(k=0; k<row2; k++)
                        c[i][j]+ = a[i][k] + b[k][j];
                        printf("%6d", c[i][j]);
                        }
                }
        }
        else
        {
                printf("Matrix multiplication is not possible");
        }
  }
```

(a) Draw the DD graph for the program.

(b) Calculate the cyclomatic complexity of the program using all four methods.

(c) List all independent paths.

(d) Design all test cases from the independent paths.

(e) Derive all du-paths and dc-paths using data flow testing.

21.

Consider the following program for finding the prime numbers, their sum, and count:

```
main()
{
        int num, flag, sum, count;
        int CheckPrime(int n);
        sum = count = 0;
        printf("Prime number between 1 and 100 are");
        for(num=1; num<=50; num++)
```

```
        {
                flag = CheckPrime(num);
                if(flag)
                {
                        printf("%d", num);
                        sum+ = num;
                        count++;
                }
        }
        printf("Sum of primes %d", count);
}
int CheckPrime(int n)
{
        int srt, d;
        srt = sqrt(n);
        d = 2;
        while(d <= srt)
        {
                If(n%d == 0)
                break;
                d++;
        }
        if(d > srt)
                return(1);
        else
                return(0);
}
```

(a) Draw the DD graph for the program.
(b) This program consists of main() and one module. Calculate the individual cyclomatic complexity number for both and collectively for the whole program. Show that individual cyclomatic complexity of main() and CheckPrime() and cyclomatic complexity of whole program is equal.
(c) List all independent paths.
(d) Design all test cases from independent paths.
(e) Derive all du-paths and dc-paths using data flow testing.

22.

Consider the following program for calculating the grade of a student:

```
main()
{
      char grade;
      int s1, s2, s3, s4, total;
      float average;
      printf("Enter the marks of 4 subjects");
      scanf("%d  %d  %d  %d", &s1,&s2,&s3,&s4);

            if(s1 < 40 || s2 < 40 || s3 < 40 || s4 < 40)
                  printf("Student has failed");
            else
            {
                  total = s1 + s2 + s3 + s4;
                  average = total/4.0;
                  if(average > 80.0)
                        grade = 'A';
                  else if(average >= 70.0)
                        grade = 'B';
                  else if(average >= 60.0)
                        grade = 'C';
                  else if(average >= 50.0)
                        grade = 'D';
                  else if(average >= 45.0)
                        grade = 'E';
                  else
                        grade = 'F';
                  switch(grade)
                  {
                        case 'A':   printf("Student has got A grade");
                        case 'B':   printf("Student has got B grade");
                        case 'C':   printf("Student has got C grade");
                        case 'D':   printf("Student has got D grade");
                        case 'E':   printf("Student has got E grade");
                        case 'F':   printf("Student has got F grade");
                  }
            }
      }
```

(a) Draw the DD graph for the program.
(b) Calculate the cyclomatic complexity of the program using all four methods.
(c) List all independent paths.
(d) Design all test cases from independent paths.