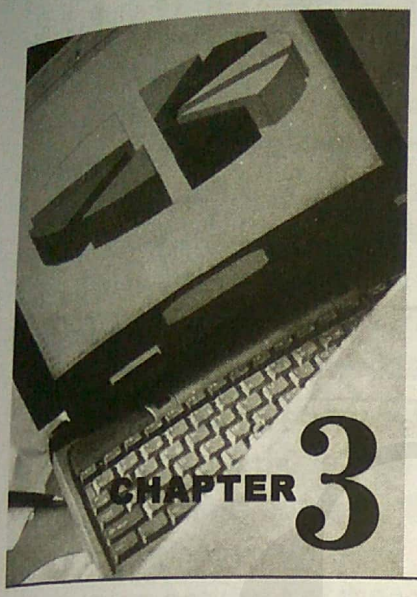


CHAPTER 3

BLACK BOX (OR FUNCTIONAL) TESTING TECHNIQUES

3.0. INTRODUCTION TO BLACK BOX (OR FUNCTIONAL TESTING)

The term **'Black Box'** refers to the software which is treated as a black box. By treating it as a black box, we mean that the system or source code is not checked at all. It is done from customer's viewpoint. The test engineer engaged in black box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software. We will now discuss various techniques of performing black box testing.

INSIDE THIS CHAPTER

3.1. BOUNDARY VALUE ANALYSIS (BVA)

It is a black box testing technique that believes and extends the concept that the density of defect is more towards the boundaries. This is done to the following reasons :

(i) Programmers usually are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.

(ii) Different terminating conditions of for-loops, while loops and repeat loops may cause defects to move around the boundary conditions.

(iii) The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

Strongly typed languages such as Ada and Pascal permit explicit definition of variable ranges. Other languages such as COBOL, FORTRAN and C are **not strongly typed,** so boundary value testing is more appropriate for programs coded in such languages.

3.1.1. What is BVA ?

The basic idea of BVA is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum and at their maximum. That is, {min, min+, nom, max–, max}. This is shown in Fig. 3.1.

BVA is based upon a critical assumption that is known as '**Single fault assumption theory**'. According to this assumption, we derive the test cases on the basis of the fact that failures are not due to simultaneous occurrence of two (or more) faults. So, we derive test cases by holding the values of all but one variable at their nominal values and letting that variable assume its extreme values.

If we have a function of n-variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max– and max values, repeating this for each variable. Thus, **for a function of n variables, BVA yields (4n + 1) test cases.**

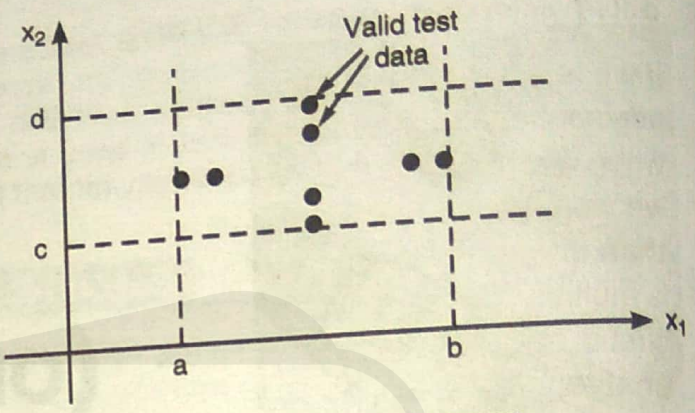


Fig. 3.1. BVA Test Cases.

3.1.2. Limitations of BVA

1. Boolean and logical variables present a problem for Boundary Value Analysis.
2. BVA assumes the variables to be truly independent which is not always possible.
3. BVA test cases have been found to be rudimentary because they are obtained with very little insight and imagination.

3.1.3. Robustness Testing

Another variant to BVA is Robustness testing. In **BVA**, we are within the legitimate boundary of our range. That is, we consider the following values for testing :

{min, min+, nom, max–, max} whereas in **Robustness testing**, we try to cross these legitimate boundaries also. So, now we consider these values for testing :

{min–, min, min+, nom, max–, max, max+}

Again, with robustness testing, we can focus on exception handling. With strongly typed languages, robustness testing may be very awkward. For example, in PASCAL, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution.

For a program with n-variables, robustness testing will yield (6n + 1) test-cases. So, we can draw a graph now. (Fig. 3.2)

Each dot represents a test value at which the program is to be tested. In Robustness testing,

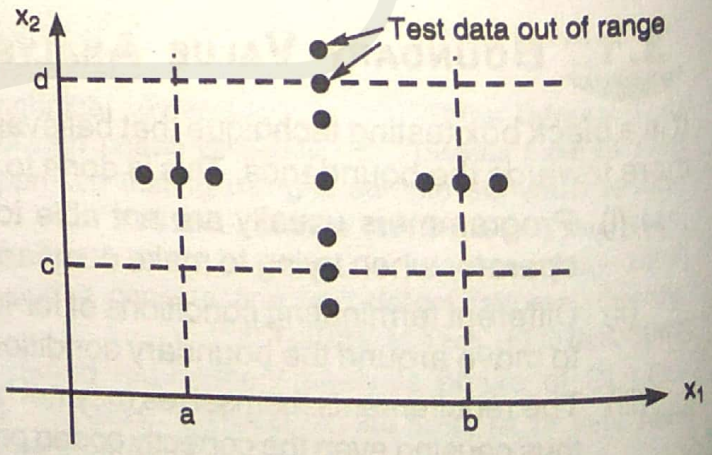


Fig. 3.2. Robustness Test Cases.

we cross the legitimate boundaries of input domain. In graph of Fig. 3.2, we show this by dots that are outside the range [a, b] of variable x_1. Similarly, for variable x_2, we have crossed its legitimate boundary of [c, d] of variable x_2.

This type of testing is quite common in electric and electronic circuits.

Furthermore, this type of testing also works on 'single fault assumption theory'.

3.1.4. Worst-Case Testing

If we reject our basic assumption of single fault assumption theory and focus on what happens when we reject this theory—it simply means that we want to see that what happens **when more than one variable has an extreme value.** This is multiple path assumption theory. In electronic circuit analysis, this is called as **"worst-case analysis".** We use this idea here to generate worst-case test cases.

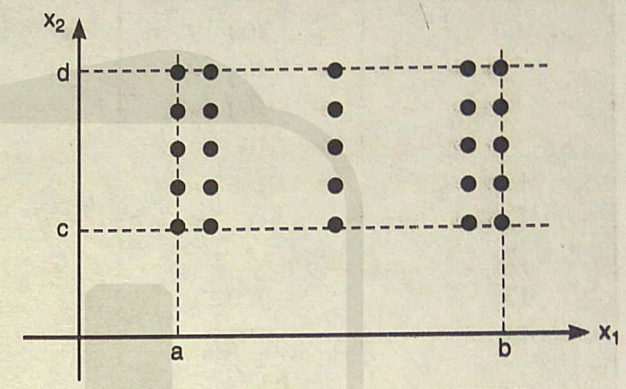


Fig. 3.3. Worst-Case Test Cases.

For each variable, we start with the five-element set that contains the min, min+, nom, max–, and max values. We then take the **Cartesian product** of these sets to generate test cases. This is shown in Fig. 3.3.

For a program with n-variables, 5^n test cases are generated.

NOTE

Robust-worst case testing yields 7^n test cases.

3.1.5. Examples with Their Problem Domain

3.1.5.1. Test-Cases for the Triangle Problem

Before we generate the test cases, firstly we need to give the problem domain :

Problem Domain : "The triangle program accepts three integers, a, b and c as input. These are taken to be the sides of a triangle. The integers a, b and c must satisfy the following conditions :

$C_1 : 1 \le a \le 200$ $\quad C_4 : a < b + c$

$C_2 : 1 \le b \le 200$ $\quad C_5 : b < a + c$

$C_3 : 1 \le c \le 200$ $\quad C_6 : c < a + b$

The output of the program may be : Equilateral, Isosceles, Scalene or "NOT-A-TRIANGLE".

How to Generate BVA Test Cases ?

We know that our range is [1, 200] where 1 is the lower bound and 200 being the upper bound. Also, we find that this program has three inputs—a, b and c. So, for our case

$$n = 3$$

$\because$ BVA yields (4n + 1) test cases, so we can say that the total number of test cases will be $(4 * 3 + 1) = 12 + 1 = 13$.

We draw the Table 3.1 now which shows those 13 test-cases.

TABLE 3.1. BVA test cases for triangle problem.

Case ID	a	b	c	Expected Output
1.	100	100	1	Isosceles
2.	100	100	2	Isosceles
3.	100	100	100	Equilateral
4.	100	100	199	Isosceles
5.	100	100	200	Not a triangle
6.	100	1	100	Isosceles
7.	100	2	100	Isosceles
8.	100	100	100	Equilateral
9.	100	199	100	Isosceles
10.	100	200	100	Not a triangle
11.	1	100	100	Isosceles
12.	2	100	100	Isosceles
13.	100	100	100	Equilateral
14.	199	100	100	Isosceles
15.	200	100	100	Not a triangle

Please note that we explained above that we can have 13 test cases (4n + 1) for this problem. But instead of 13, now we have 15 test cases. Also, test case ID number 8 and 13 are redundant. So, we ignore them. However, we do not ignore test case ID number 3 as we must consider at least one test case out of these three. Obviously, it is a mechanical work !

So, we can say that these 13 test cases are sufficient to test this program using BVA technique.

Question for Practice

1. Applying Robustness testing technique, how would you generate the test-cases for the triangle problem given above.

3.1.5.2. Test-Cases for Next Date Function

Before we generate the test cases for the Next Date Function, we must know the problem domain of this program :

Problem Domain : "Next Date is a function of three variables : month, date and year. It returns the date of next day as output. It reads current date as input date. The conditions are :

$C_1 : 1 \leq \text{month} \leq 12$

$C_2 : 1 \leq \text{day} \leq 31$

$C_3 : 1900 \leq \text{year} \leq 2025$".

If any of conditions C_1, C_2 or C_3 fails, then this function produces an output "value of month not in the range 1...12".

Because many combinations of dates exist so this function just displays one message : "Invalid Input Date".

Complexities in Next Date Function

A very common and popular problem occurs if the year is a leap year. We have taken into consideration that there are 31 days in a month. But what happens if a month has 30 days or even 29 or 28 days? A year is called as a leap year if it is divisible by 4, unless it is a century year. **Century years** are leap years only if they are multiples of 400. So, **1992, 1996 and 2000 are leap years while 1900 is not a leap year.**

Furthermore, in this Next Date problem we find examples of **Zipf's law** also, which states that **"80% of the activity occurs in 20% of the space."** Here also, much of the source-code of Next Date function is devoted to the leap year considerations.

How to Generate BVA test Cases for this Problem ?

The Next Date program takes date as input and checks it for validity. If valid, it returns the next date as its output.

As we know, with single fault assumption theory, (4n + 1) test cases can be designed. Here, also n = 3. So, total number of test cases are (4 × 3 + 1) = 12 + 1 = 13.

The boundary value test cases are

TABLE 3.2.

Case ID	Month (mm)	Day (dd)	Year (yyyy)	Expected Output
1.	6	15	1900	16 June, 1900
2.	6	15	1901	16 June, 1901
3.	6	15	1962	16 June, 1962
4.	6	15	2024	16 June, 2024
5.	6	15	2025	16 June, 2025
6.	6	1	1962	2 June, 1962
7.	6	2	1962	1 June, 1962
8.	6	30	1962	1 July, 1962
9.	6	31	1962	Invalid date as June has 30 days.
10.	1	15	1962	16 January, 1962
11.	2	15	1962	16 February, 1962
12.	11	15	1962	16 November, 1962
13.	12	15	1962	16 December, 1962

So, we have applied BVA on our Next-Date Problem.

Question for Practice

1. Applying Robustness testing, how would you generate the test cases for the Next-Date function given above.

3.1.5.3. Test-Cases for the Commission Problem

Before we generate the test cases, we must formulate the problem statement or domain for commission problem.

Problem Domain : "A rifle sales person sold rifle locks, stocks and barrels that were made by a gunsmith. Locks cost $45, stocks cost $30 and barrels cost $25. This salesperson had to sell at least one complete rifle per month, and the production limits were such that the most the sales person could sell in a month was 70 locks, 80 locks and 90 barrels. The sales person used to send the details of sold items to the gunsmith. The gunsmith then computed the sales person's commission as follows :

(a) 10% on sales upto and including $1000.

(b) 15% of the next $800.

(c) And 20% on any sales in excess of $1800.

The commission program produced a monthly sales report that gave the total number of locks, stocks and barrels sold, the sales person's total dollar sales and finally, the commission."

How to Generate BVA Test Cases for this Problem ?

Since we have 3 inputs in this program. So, I need (4n + 1) = 4 * 3 + 1 = 12 + 1 = 13 test cases to test this program.

The boundary value test cases are listed below in Table 3.3. We can also find that the monthly sales are limited as follows :

$1 \leq locks \leq 70$

$1 \leq stocks \leq 80$

$1 \leq barrels \leq 90$

TABLE 3.3.

Case ID	Locks	Stocks	Barrels	Sales
1.	35	40	1	2800
2.	35	40	2	2825
3.	35	40	45	3900
4.	35	40	89	5000
5.	35	40	90	5025
6	35	1	45	2730
7.	35	2	45	2760
8.	35	40	45	3900
9.	35	79	45	5070
10.	35	80	45	5100
11.	1	40	45	2370
12.	2	40	45	2415
13.	35	40	45	3900
14.	69	40	45	5430
15.	70	40	45	5475

Out of these 15 test cases, 2 are redundant. So, 13 test cases are sufficient to test this program.

3.1.6. Guidelines for BVA

1. The normal versus robust values and the single fault versus the multiple-fault assumption theory result in better testing. These methods can be applied to both input and output domain of any program.
2. Robustness testing is a good choice for testing internal variables.
3. Keep in mind that you can create extreme boundary results from non-extreme input values.

3.2. Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations :

(a) We want exhaustive testing and

(b) We want to avoid redundancy.

This is not handled by BVA technique as we can see massive redundancy in the tables of test cases.

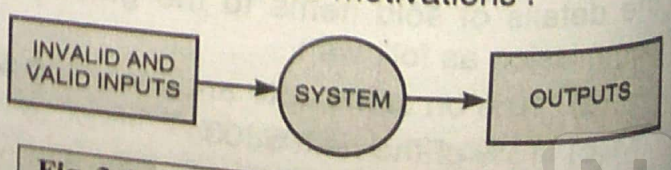


Fig. 3.4. Equivalence Class Partitioning.

In this technique, **the input and the output domain** is divided into a finite number of equivalence classes. Then, **we select one representative of each class and test our program against it.** It is assumed by the tester that if one representative from a class is able to detect error then why should he consider other cases. Furthermore, if this single representative test case did not detect any error then we assume that no other test case of this class can detect error. In this method we consider both valid and invalid input domains. The system is still treated as a black-box meaning that we are not bothered about its internal logic.

The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, the potential redundancy among test cases can be reduced.

For example, in our triangle problem, we would certainly have a test case for an equilateral triangle and we might pick the triple (10, 10, 10) as inputs for a test case. If this is so then it is obvious that there is no sense in testing for inputs like (8, 8, 8) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case. Thus, they would be redundant. The key and the crafsmanship lies in the choice of the equivalence relation that determines the classes.

Four types of equivalence class testing are discussed below :

1. Weak Normal Equivalence Class Testing.
2. Strong Normal Equivalence Class Testing.
3. Weak Robust Equivalence Class Testing.
4. Strong Robust Equivalence Class Testing.

We will discuss these one by one.

3.2.1. Weak Normal Equivalence Class Testing

The word **'weak'** means **'single fault assumption'.** This type of testing is accomplished by using one variable from each equivalence class in a test case. We would, thus, end up with the weak equivalence class test cases as shown in Fig. 3.5.

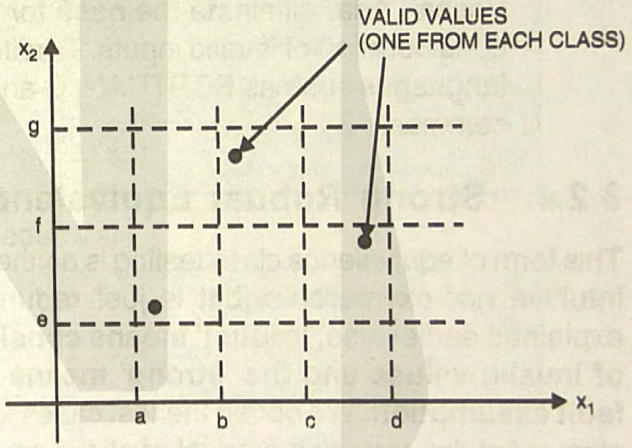


Fig. 3.5. Weak Normal Equivalence Class Test Cases.

Each dot in Fig. 3.5 indicates a test data. From each class we have one dot meaning that there is one representative element of each test case. In fact, we will have, always, the same number of weak equivalence class test cases as the classes in the partition.

3.2.2. Strong Normal Equivalence Class Testing

This type of testing is based on the multiple fault assumption theory. So, now we need test cases from each element of the cartesian product of the equivalence classes, as shown in Fig. 3.6.

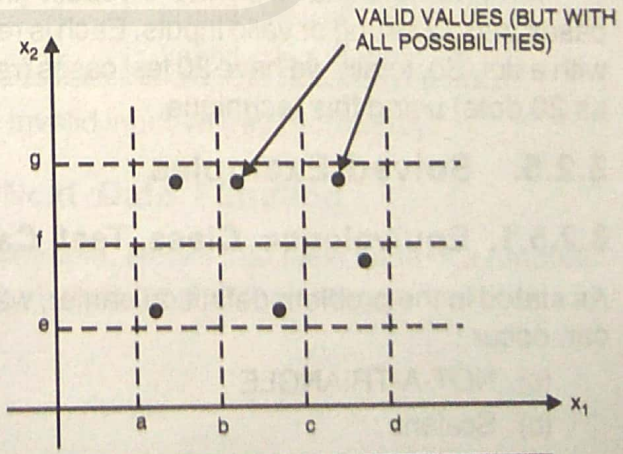


Fig. 3.6. Strong Normal Equivalence Class Test Cases.

Just like we have truth tables in digital logic, we have similarities between these truth tables and our pattern of test cases. The cartesian product guarantees that we have a notion of "completeness" in two ways :

(a) We cover all equivalence classes.
(b) We have one of each possible combination of inputs.

3.2.3. Weak Robust Equivalence Class Testing

The name for this form of testing is counter intuitive and oxymoronic. **The word 'weak' means single fault assumption theory and the word 'Robust' refers to invalid values.** The test cases resulting from this strategy are shown in Fig. 3.7.

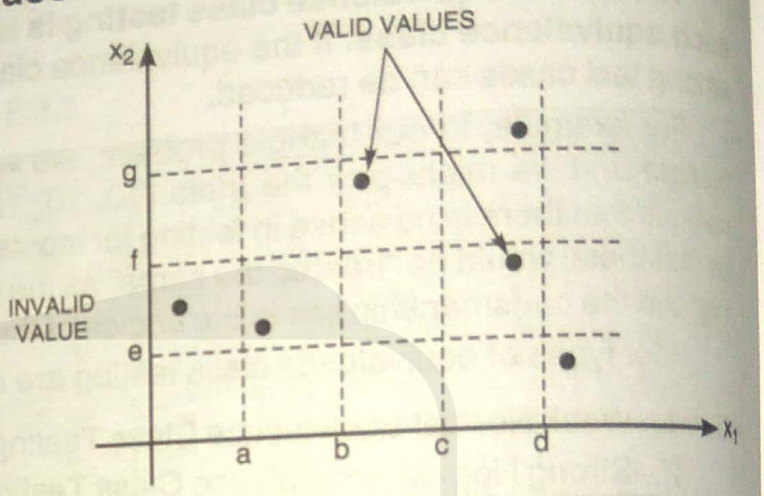


Fig. 3.7. Weak Robust Equivalence Class Test Cases.

Two problems occur with robust equivalence testing. They are listed below :

1. Very often the specification does not define what the expected output for an invalid test case should be. Thus, testers spend a lot of time defining expected outputs for these cases.
2. Also, strongly typed languages like Pascal, Ada, eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN, C and COBOL were dominant. Thus this type of error was common.

3.2.4. Strong Robust Equivalence Class Testing

This form of equivalence class testing is neither counter intuitive nor oxymoronic, but is just redundant. As explained earlier also, **'robust' means consideration of invalid values and the 'strong' means multiple fault assumption.** We obtain the test cases from each element of the cartesian product of all the equivalence classes. This is shown in Fig. 3.8.

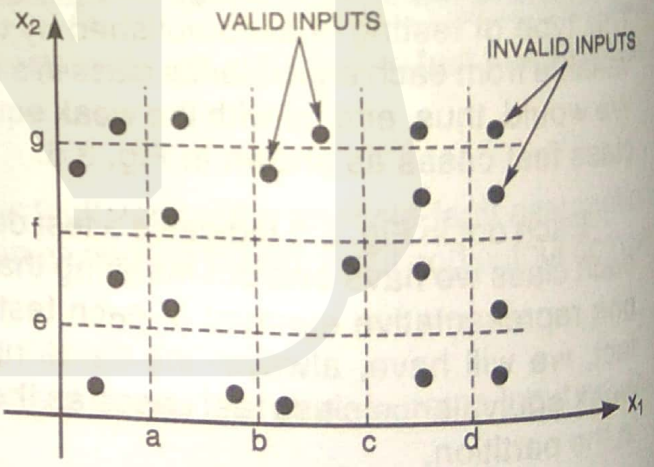


Fig. 3.8. Strong Robust Equivalence Class Test Cases.

We find here that we have 8 robust (invalid) test cases and 12 strong or valid inputs. Each is represented with a dot. So, totally we have 20 test cases (represented as 20 dots) using this technique.

3.2.5. Solved Examples

3.2.5.1. Equivalence Class Test Cases for the Triangle Problem

As stated in the problem definition earlier, we note that in our triangle problem four possible outputs can occur :

(a) NOT-A-TRIANGLE
(b) Scalene
(c) Isosceles, and
(d) Equilateral.

We can use these to identify output (range) equivalence classes as follows :

01 = {<a, b, c> : the triangle is equilateral}
02 = {<a, b, c> : the triangle is isosceles}
03 = {<a, b, c> : the triangle is scalene}
04 = {<a, b, c> : sides a, b and c do not form a triangle}

Now, we apply these four techniques of equivalence class partitioning one by one to this problem.

(a) The four **weak normal equivalence class** test cases are :

Case ID	a	b	c	Expected output
WN_1	5	5	5	Equilateral
WN_2	2	2	3	Isosceles
WN_3	3	4	5	Scalene
WN_4	4	1	2	Not a triangle

(b) Because no valid subintervals of variables a, b and c exist, so the strong normal equivalence class test cases **are identical** to the weak normal equivalence class test cases.

(c) Considering the invalid values for a, b and c yields the following additional **weak robust equivalence class test cases :**

Case ID	a	b	c	Expected output
WR_1	–1	5	5	Invalid value of a
WR_2	5	–1	5	Invalid value of b
WR_3	5	5	–1	Invalid value of c
WR_4	201	5	5	Out of range value of a
WR_5	5	201	5	Out of range value of b
WR_6	5	5	201	Out of range value of c

(d) **While strong robust equivalance class test cases** are

Case ID	a	b	c	Expected output
SR_1	–1	5	5	Invalid value of a
SR_2	5	–1	5	Invalid value of b
SR_3	5	5	–1	Invalid value of c
SR_4	–1	–1	5	Invalid values of a and b
SR_5	5	–1	–1	Invalid values of b and c
SR_6	–1	5	–1	Invalid values of a and c
SR_7	–1	–1	–1	Invalid values of a, b and c

Please note that the expected outputs describe the invalid input values thoroughly.

3.2.5.2. Equivalence Class Test Cases for Next Date Function

The actual craft of choosing the test cases lies in this example. Recall that Next Date is a function of three variables— month (mm), day (dd) and year (yyyy). Assume that their ranges are

$1 \le \text{month} \le 12$
$1 \le \text{day} \le 31$
$1812 \le \text{year} \le 2012$

So, based on valid values, the equivalence classes are :

M_1 = {month : $1 \le$ month ≤ 12}

D_1 = {day : $1 \le$ day ≤ 31}

Y_1 = {year : $1812 \le$ year ≤ 2012}

And the invalid equivalence classes are :

M_2 = {month : month < 1}

M_3 = {month : month > 12}

D_2 = {day : day < 1}

D_3 = {day : day > 31}

Y_2 = {year : year < 1812}

Y_3 = {year : year > 2012}

(a and b) : Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs and it is identical to the strong normal equivalence class test case :

Case ID	Month	Day	Year	Expected output
WN_1, SN_1	6	15	1912	6/16/1912

So, we get this test case on the basis of valid classes – M_1, D_1 and Y_1 above.

(c) Weak robust test cases are given below :

Case ID	Month	Day	Year	Expected output
WR_1	6	15	1912	6/16/1912
WR_2	–1	15	1912	Invalid value of month as month cannot be –ve
WR_3	13	15	1912	Invalid value of month as month <12, always.
WR_4	6	–1	1912	Invalid value of day as day cannot be –ve.
WR_5	6	32	1912	Invalid value of day as we cannot have 32 days in any month.
WR_6	6	15	1811	Invalid value of year as its range is 1812 to 2012 only.
WR_7	6	15	2013	Invalid value of year.

So, we get 7 test cases based on the valid and invalid classes of the input domain.

(d) Strong robust equivalence class test cases are also given below :

Case ID	Month	Day	Year	Expected output
SR_1	–1	15	1912	Invalid value of month as month cannot be –ve.
SR_2	6	–1	1912	Invalid value of day as day is –ve.
SR_3	6	15	1811	Invalid value of year.
SR_4	–1	–1	1912	Invalid month and day.
SR_5	6	–1	1811	Invalid day and year.
SR_6	–1	15	1811	Invalid month and year.
SR_7	–1	–1	1811	Invalid month, day and year.

Modified Equivalence Class for this Problem

We need the modified classes as we know that at the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1 and the year is also incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we postulate the following equivalence classes :

M_1 = {month : month has 30 days}
M_2 = {month : month has 31 days}
M_3 = {month : month is February}
D_1 = {day : $1 \leq$ day ≤ 28}
D_2 = {day : day = 29}
D_3 = {day : day = 30}
D_4 = {day : day = 31}
Y_1 = {year : year = 2000}
Y_2 = {year : year is a leap year}
Y_3 = {year : year is a common year}

So, now what will be the weak equivalence class test cases ?

As done earlier also, the inputs are mechanically selected from the approximate middle of the corresponding class :

Case ID	Month	Day	Year	Expected output
WN_1	6	14	2000	6/15/2000
WN_2	7	29	1996	7/30/1996
WN_3	2	30	2002	2/31/2002 (Impossible)
WN_4	6	31	2000	7/1/2000 (Impossible)

This **mechanical selection** of input values makes no consideration of our domain knowledge and thus we have two impossible dates. This will always be a problem with 'automatic' test case generation because all of our domain knowledge is not captured in the choice of equivalence classes.

The **strong normal equivalence class** test cases for the **revised classes** are :

Case ID	Month	Day	Year	Expected output
SN_1	6	14	2000	6/15/2000
SN_2	6	14	1996	6/15/1996
SN_3	6	14	2002	6/15/2002
SN_4	6	29	2000	6/30/2000
SN_5	6	29	1996	6/30/1996
SN_6	6	29	2002	6/30/2002
SN_7	6	30	2000	6/31/200 (Impossible date)
SN_8	6	30	1996	6/31/1996 (Impossible date)
SN_9	6	30	2002	6/31/2002 (Impossible date)
SN_{10}	6	31	2000	7/1/2000 (Invalid input)
SN_{11}	6	31	1996	7/1/1996 (Invalid input)
SN_{12}	6	31	2002	7/1/2002 (Invalid input)
SN_{13}	7	14	2000	7/15/2000
SN_{14}	7	14	1996	7/15/1996

SN_{15}	7	14	2002	7/15/2002
SN_{16}	7	29	2000	7/30/2000
SN_{17}	7	29	1990	7/30/1996
SN_{18}	7	29	2002	7/30/2002
SN_{19}	7	30	2000	7/31/2000
SN_{20}	7	30	1996	7/31/1996
SN_{21}	7	30	2002	7/31/2002
SN_{22}	7	31	2000	8/1/1996
SN_{23}	7	31	1996	8/1/2000
SN_{24}	7	31	2002	8/1/2002
SN_{25}	2	14	2000	2/15/2000
SN_{26}	2	14	1996	2/15/1996
SN_{27}	2	14	2002	2/15/2002
SN_{28}	2	29	2000	3/1/2000 (Invalid input)
SN_{29}	2	29	1996	3/1/1996
SN_{30}	2	29	2002	3/1/2002 (Impossible date)
SN_{31}	2	30	2000	3/1/2000 (Impossible date)
SN_{32}	2	30	1996	3/1/1996 (Impossible date)
SN_{33}	2	30	2002	3/1/2002 (Impossible date)
SN_{34}	6	31	2000	7/1/2000 (Impossible date)
SN_{35}	6	31	1996	7/1/1996 (Impossible date)
SN_{36}	6	31	2002	7/1/2002 (Impossible date)

So, three month classes, four day classes and three year classes results in 3 * 4 * 3 = 36 strong normal equivalence class test cases. Furthermore, adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases.

It is quite difficult to show these 150 classes here.

3.2.5.3. Equivalence Class Test Cases for the Commission Problem

The valid classes of the input variables are as follows :

L_1 = {locks : $1 \leq$ locks ≤ 70}
L_2 = {locks = -1}
S_1 = {stock : $1 \leq$ stocks ≤ 80}
B_1 = {barrels : $1 \leq$ barrels ≤ 90}

The corresponding invalid classes of the input variables are as follows :

L_3 = {locks : locks = 0 or locks < -1}
L_4 = {locks : locks > 70}
S_2 = {stocks : stocks < 1}
S_3 = {stocks : stocks > 80}
B_2 = {barrels : barrels < 1}
B_3 = {barrels : barrels > 90}

(a & b) : As the number of valid classes is equal to the number of independent variables, so we have exactly one weak normal equivalence class test case and again, it is identical to the strong normal equivalence class test case. It is given on the next page.

Case ID	Locks	Stocks	Barrels	Expected output
WN_1, SN_1	35	40	45	3900

(c) Also, we have seven weak robust test cases as given below :

Case ID	Locks	Stocks	Barrels	Expected output
WR_1	35	40	45	3900
WR_2	0	40	45	Invalid input
WR_3	71	40	45	Invalid input
WR_4	35	0	45	Invalid input
WR_5	35	81	45	Invalid input
WR_6	35	40	0	Invalid input
WR_7	35	40	91	Invalid input

(d) And finally, the strong robust equivalence class test cases are as follows :

Case ID	Locks	Stocks	Barrels	Expected output
SR_1	–1	40	45	Value of locks not in the range 1---70
SR_2	35	–1	45	Value of stocks not in the range 1---80
SR_3	35	40	–1	Value of barrels not in the range 1---90
SR_4	–1	–1	45	Values of locks and stocks are not in their ranges.
SR_5	–1	40	–1	Values of locks and barrels are not in their ranges.
SR_6	35	–1	–1	Values of stocks and barrels are not in their ranges.
SR_7	–1	–1	–1	Values of locks, stocks and barrels are not in their ranges.

3.2.6. Guidelines for Equivalence Class Testing

The following guidelines have been found out after studying above examples :

1. The weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed and invalid values cause run-time errors then it makes no sense to use the robust form.
3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is approximate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing (BVA).
6. Equivalence class-testing is used when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the next date problem.

7. Strong equivalence class testing makes a presumption that the variables are independent and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases, as done in the next date function.
8. Several tries may be needed before the "right" equivalence relation is established.
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

3.3. Decision Table Based Testing

Of all the functional testing methods, those based on decision tables are the most rigorous because decision tables enforce logical rigour.

3.3.1. What are Decision Tables ?

Decision tables are a precise and compact way to model complicated logic. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions.

It is another popular black-box testing technique. A decision table has **four** portions :

(a) Stub portion
(b) Entry portion
(c) Condition portion, and
(d) Action portion

Rules ⟶

Condition Stub	Condition Entry
Action Stub	Action Entries

Fig. 3.9. Structure of Decision Table.

A column in the entry portion is a **rule**. Rules indicate which actions are taken for the conditional circumstances indicated in the condition portion of the rule. Decision tables in which all conditions are binary are called as **limited entry decision tables**. If conditions are allowed to have several values, the resulting tables are called **extended entry decision tables.**

To identify test cases with decision tables, we follow certain steps :

Step 1. For a module identify input conditions (causes) and action (effect).

Step 2. Develop a cause-effect graph.

Step 3. Transform this cause-effect graph, so obtained in step 2 to a decision table.

Step 4. Convert decision table rules to test cases.

Each column of the decision table represents a test case. That is,

Number of Test Cases = Number of Rules

For a limited entry decision tables, if n conditions exist, there must be 2^n rules.

3.3.2. Advantages, Disadvantages and Applications of Decision Tables

Advantages of Decision Tables

1. This type of testing also works **iteratively**. The table that is drawn in the first iteration, acts as a stepping stone to derive new decision table(s), if the initial table is unsatisfactory.
2. These tables guranatee that we consider every possible combination of condition values. This is known as its "**completeness property**". This property promises a form of complete testing as compared to other techniques.

3. Decision tables are **declarative**. There is no particular order for conditions and actions to occur.

Disadvantages of Decision Tables

Decision tables do not scale up well. We need to "factor" large tables into smaller ones to remove redundancy.

Applications of Decision Tables

This technique is useful for applications characterized by any of the following :

(a) Prominent if-then-else logic.
(b) Logical relationships among input variables.
(c) Calculations involving subsets of the input variables.
(d) Cause-and-effect relationships between inputs and outputs.
(e) High cyclomatic complexity.

Technique : To identify test cases with decision tables, we interpret conditions as inputs and actions as output. The rules are interpreted as test cases. Because the decision table can mechanically be forced to be complete, we know we have a comprehensive set of test cases.

Example of a Decision Table : The triangle problem

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
C_1 : a < b + c ?	F	T	T	T	T	T	T	T	T	T	T
C_2 : b < a + c ?	—	F	T	T	T	T	T	T	T	T	T
C_3 : c < a + b ?	—	—	F	T	T	T	T	T	T	T	T
C_4 : c = b ?	—	—	—	T	T	T	T	F	F	F	F
C_5 : a = c ?	—	—	—	T	T	F	F	T	T	F	F
C_6 : b = c ?	—	—	—	T	F	T	F	T	F	T	F
a_1 : Not a triangle	×	×	×								
a_2 : Scalene											×
a_3 : Isosceles							×		×	×	
a_4 : Equilateral				×							
a_5 : Impossible					×	×		×			

Fig. 3.10. Example of Decision Table.

Each "-" (hyphen) in the decision table represents a don't care entry. Use of such entries has a subtle effect on the way in which complete decision tables are recognized. For limited entry decision tables, if n conditions exist, there must be 2^n rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows :

Rule 1. Rules in which no don't care entries occur count as one rule.

Note that each column of a decision table represents a rule and number of rules is equal to the number of test cases.

Rule 2. Each don't care entry in a rule doubles the count of that rule.

Note that in this decision table we have 6-conditions (C_1---C_6). Therefore,

$$n = 6$$

Also, we can have 2^n entries i.e., $2^6 = 64$ entries. Now, we establish the rule and the rule count for the above decision table.

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	
C_1 : a < b + c ?	F	T	T	T	T	T	T	T	T	T	T	
C_2 : b < a + c ?	—	F	T	T	T	T	T	T	T	T	T	
C_3 : c < a + b ?	—	—	F	T	T	T	T	T	T	T	T	
C_4 : a = b ?	—	—	—	T	T	T	T	F	F	F	F	
C_5 : a = c ?	—	—	—	T	T	F	F	T	T	F	F	
C_6 : b = c ?	—	—	—	T	F	T	F	T	F	T	F	
Rule Count	32	16	8	1	1	1	1	1	1	1	1	= 64
a_1 : Not a triangle	×	×	×									
a_2 : Scalene											×	
a_3 : Isosceles						×		×	×			
a_4 : Equilateral				×								
a_5 : Impossible					×	×		×				

Fig. 3.11. Decision Table With Rule Counts.

From the above table we find that the rule count is 64. And we have already said that $2^n = 2^6 = 64$. So, both are 64.

The question, however, is to find out why the rule-count is 32 for the Rule-1 (or column-1) ?

We find that there are 5 don't cares in Rule-1 (or column-1) and hence $2^n = 2^5 = 32$. Hence, the rule-count for Rule-1 is 32. Similarly, for Rule-2, it is $2^4 = 16$ and $2^3 = 8$ for Rule-3. However, from Rule-4 till Rule-11, number of don't care entries is 0 (zero). So rule-count is $2^0 = 1$ for all these columns. Summing the rule-count of all columns (or R_1-R_{11}) we get a total of 64 Rule-count.

Many a times some problems also arise with these decision tables. Let us see how.

Consider the following example of a **Redundant decision table**

Conditions	1–4	5	6	7	8	9
C_1	T	F	F	F	F	T
C_2	—	T	T	F	F	F
C_3	—	T	F	T	F	F
a_1	×	×	×	—	—	—
a_2	—	×	×	×	—	×
a_3	×	—	×	×	×	×

Fig. 3.12. Redundant Decision Table.

Please note that the action entries in Rule-9 and Rules 1–4 are NOT identical. It means that if the decision table were to process a transaction in which C_1 is true and both C_2 and C_3 are false, both rules 4 and 9 apply. We observe two things :

1. Rules 4 and 9 are **in-consistent** because the action sets are different.
2. The whole table is **non-deterministic** because there is no way to decide whether to apply Rule-4 or Rule-9.

Also note carefully that there is a bottom line for testers now. **They should take care when don't care entries are being used in a decision table.**

3.3.3. Examples

3.3.3.1. Test Cases for the Triangle Problem Using Decision Table Based Testing Technique

We have already studied the problem domain for the famous triangle problem in previous chapters. Next we apply decision table based technique on the triangle problem. The following are the test cases :

Case ID	a	b	c	Expected output
D_1	4	1	2	Not a triangle
D_2	1	4	2	Not a triangle
D_3	1	2	4	Not a triangle
D_4	5	5	5	Equilateral
D_5	?	?	?	Impossible
D_6	?	?	?	Impossible
D_7	2	2	3	Isosceles
D_8	?	?	?	Impossible
D_9	2	3	2	Isosceles
D_{10}	3	2	2	Isosceles
D_{11}	3	4	5	Scalene

Fig. 3.13. Test Cases For Triangle Problem.

So, we get a total of 11 functional test cases out of which **three** are impossible cases, **three** fail to satisfy the triangle property, **one** satisfies equilateral triangle property, **one** satisfies scalene triangle property and **three** ways to get an isoceles triangle.

3.3.3.2. Test Cases for Next Date Function

In previous technique of equivalence partitioning we found that certain logical dependencies exist among variables in the input domain. These dependencies are lost in a cartesian product. The decision table format allows us to use the notion of "impossible action" to denote impossible combinations of conditions. Thus, such dependencies are easily shown in decision tables.

From the problem domain for the next date function, we know that there are some critical situations also, like treatment of leap years, special treatment to year 2000 (Y2K) and special treatment to December month and a 28 day month is also to be given. So, we go for 3 tries before we derive its test cases.

First try : Special treatment to leap years.

Second try : Special treatment to year = 2000.

Third try : Special treatment to December month and days = 28.

First try : Special treatment to leap years.

The art of testing and **the actual craftsmanship** lies in identifying appropriate conditions and actions. Considering the following equivalence classes again.

M_1 = {Month : Month has 30 days}

M_2 = {Month : Month has 31 days }

M_3 = {Month : Month is February}

D_1 = {day : $1 \leq$ day ≤ 28}
D_2 = {day : day = 29}
D_3 = {day : day = 30}
D_4 = {day : day = 31}
Y_1 = {year : year is a leap year}
Y_2 = {year : year is not a leap year}

Based on these classes, we draw the decision table :

Conditions								
C_1 : month in M_1	T							
C_2 : month in M_2		T						
C_3 : month in M_3			T					
C_4 : day in D_1				:				
C_5 : day in D_2					:			
C_6 : day in D_3						:		
C_7 : day in D_4							:	
C_8 : year in y_1								:
a_1 : Impossible	:	:	:	:	:	:	:	:
a_2 : Next date								

Fig. 3.14. First Try Decision Table.

Herein, we have $3 \times 4 \times 2 = 24$ elements and $2^n = 2^8 = 256$ entries. Many of the rules would be impossible. (note that the classes Y_1 and Y_2 collapse into one condition C_8). Some of these rules are impossible either because

(a) There are too many days in a month.
(b) Cannot happen in a non-leap year.
(c) Compute the next date.

Second try : Special treatment given to year, 2000.

As we know, the year 2000 is a leap year. Hence, we must give special treatment to this year by creating a new class-Y_1 as follows :

M_1 = {month : month has 30 days}
M_2 = {month : month has 31 days}
M_3 = {month : month is February}
D_1 = {day : $1 \leq$ day ≤ 28}
D_2 = {day : day = 29}
D_3 = {day : day = 30}
D_4 = {day : day = 31}
Y_1 = {year : year = 2000}
Y_2 = {year : year is a leap year}
Y_3 = {year : year is a common year}

This results in $3 \times 4 \times 3 = 36$ rules that represents the cartesian product of 3 month classes (M_1.... M_3) above, 4 day classes (D_1....D_4) above and 3 year classes.

So, after second try, our decision table is

Rules → Conditions ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
C1: month in	M1	M1	M1	M1	M2	M2	M2	M2	M3	M3	M3	M3	M3	M3	M3	M3	
C2: day in	D1	D2	D3	D4	D1	D2	D3	D4	D1	D1	D1	D2	D2	D2	D3	D4	
C3: year in	–	–	–	–	–	–	–	–	y1	y2	y3	y1	y2	y3	–	–	
Rule count	3	3	3	3	3	3	3	3	1	1	1	1	1	1	3	3	36 rule-count
Actions																	
a1: impossible				×								×		×	×	×	
a2: increment day	×	×			×	×	×			×							
a3: reset day			×					×	×		×		×				
a4: increment month			×					?	×		×		×				
a5: reset month								?									
a6: increment year								?									

Fig. 3.15. Second Try Decision Table With 36 Rule-Count.

Why is the rule-count value 3 in column 1 ?

In column 1 or Rule 1 of this decision table, we have 3 possibilities with don't care :

$\{M_1, D_1, Y_1\}$
$\{M_1, D_1, Y_2\}$
$\{M_1, D_1, Y_3\}$

i.e., with '–' or don't care we have either Y_1 or Y_2 or Y_3.

Also note that in Rule 8, we have three impossible conditions, shown as '?' (question mark). Also, we find that this table has certain problems with December month. We fix these next in the third try.

Third try : Special treatment to December month and to number of days = 28.

Since we know that we have serious problems with the last day of last month i.e., December. We have to change month from 12 to 1. So, we modify our classes as follows :

M_1 = {month : month has 30 days}
M_2 = {month : month has 31 days except December}
M_3 = {month : month is December}
D_1 = {day : $1 \le$ day ≤ 27}
D_2 = {day : day = 28}
D_3 = {day : day = 29}

D_4 = {day : day = 30}
D_5 = {day : day = 31}
Y_1 = {year : year is a leap year}
Y_2 = {year is a common year}

The cartesian product of these contain 40 elements. Here, we have a 22-rule decision table. This table gives a clearer picture of the Next Date function than does the 36-rule decision table and is given below :

Rules → Conditions ↓	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
C1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
C2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
C3: year in	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	y1	y2	y1	y2	–	–
Actions																						
a1:impossible					×															×	×	×
a2: increment day	×	×	×		×	×	×	×		×	×	×	×	×		×	×					
a3: reset day				×						×					×			×	×			
a4: increment month				×						×								×	×			
a5: reset month															×							
a6: increment year															×							

Fig. 3.16. Decision Table for the Next Date Function.

In this table, the first five rules deal with 30-day months. Notice that the leap year considerations are irrelevant. Rules (6 – 10) and (11 – 15) deal with 31-day months where the first five with months other than December and the second five deal with December. No impossible rules are listed in this portion of the decision table.

Still there is some redundancy in this table. Eight of the ten rules simply increment the day. Do we really require eight separate test cases for this sub-function ? No certainly not but note the type of observation we get from the decision table.

Finally, the last seven rules focus on February and leap year. This decision table analysis could have been done during the detailed design of the Next Date function.

Further simplification of this decision table can also be done. If the **action sets of two rules in a decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry**. In a sense, we are identifying equivalence classes of these rules. For example, rules 1, 2 and 3 involve day classes as D_1, D_2 and D_3 (30 day classes). These can be combined together as the action taken by them is same. Similarly, for other rules also combinations can be done. The corresponding test cases are shown in the table as in Fig. 3.17.

Case ID	Month	Day	Year	Expected Output
1-3	April	15	2001	April 16, 2001
4	April	30	2001	May 1, 2001
5	April	31	2001	Impossible
6-9	January	15	2001	January 16, 2001
10	January	31	2001	February 1, 2001
11-14	December	15	2001	December 16, 2001
15	December	31	2001	January 1, 2002
16	February	15	2001	February 16, 2001
17	February	28	2004	February 29, 2004
18	February	28	2001	March 1, 2001
19	February	29	2004	March 1, 2004
20	February	29	2001	Impossible
21-22	February	30	2001	Impossible

Fig. 3.17. Test Cases for Next Date Problem Using Decision Table Based Testing.

Since, we have 22 Rules, so there are 22 test cases which are listed above.

3.3.3.3. Test Cases for the Commission Problem

The commission problem is not suitable for it to be solved using this technique of decision table analysis. This is because very little decisional logic is used in the problem. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

3.3.4. Guidelines for Decision Table Based Testing

The following guidelines have been found out after studying above examples :

(1) This technique works well where lot of decision making takes place such as the triangle problem and Next date problem.

(2) The decision table technique is indicated for applications characterized by any of the following :
- ➤ Prominant if-then-else logic.
- ➤ Logical relationships among input variables.
- ➤ Calculations involving subsets of the input variables.
- ➤ Cause-and-effect relationships between inputs and outputs.
- ➤ High cyclomatic complexity.

(3) Decision tables do not scale up well. We need to "factor" large tables into smaller ones to remove redundancy.

(4) It works iteratively meaning that the table drawn in the first iteration, acts as a stepping stone to design new decision tables, if the initial table is unsatisfactory.

3.4. Cause-Effect Graphing Technique

Cause-Effect graphing is basically a hardware testing technique adapted to software testing. It is a black-box method. It considers only the desired external behaviour of a system. This is a testing technique that aids in selecting test cases that logically relate causes (inputs) to effects (outputs) to produce test cases.

3.4.1. Causes and Effects

A 'cause' represents a distinct input condition that brings about an internal change in the system. A 'effect' represents an output condition, a system transformation or a state resulting from a combinatio of causes.

Myer suggests the following steps to derive test cases :

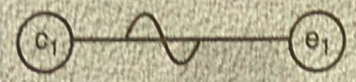
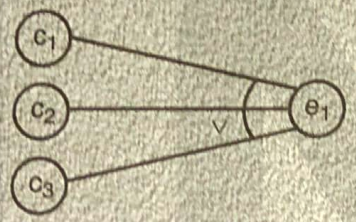
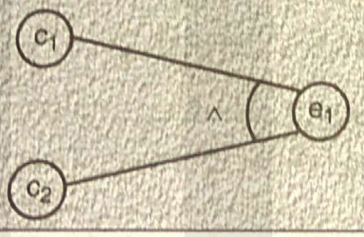
Step 1. For a module, identify the input conditions (causes) and actions (effect).

Step 2. Develop a cause-effect graph.

Step 3. Transform cause-effect graph into a decision table.

Step 4. Convert decision table rules to test cases. Each column of the decision table represen a test case.

Basic cause-effect graph symbols used are given below :

	Notation	Meaning
1.		Identity
2.		NOT
3.		OR
4.		AND

Consider each node as having the value 0 or 1 where 0 represents the 'absent state' and represents the 'present state'. Then the identity function states that if c_1 is 1, e_1 is 1 or we can sa if c_1 is 0, e_1 is 0.

The NOT function states that if C_1 is 1, e_1 is 0 and vice-versa. Similarly, OR function states th if C_1 or C_2 or C_3 is 1, e_1 is 1 else e_1 is 0. The AND function states that if both C_1 and C_2 are 1, e_1 1 ; else e_1 is 0. The AND and OR functions are allowed to have any number of inputs.

3.4.2. Test Cases for the Triangle Problem

We follow the steps listed in Section 3.4.1 to design the test cases for our triangle problem :

Step 1. Firstly, we must identify the causes and its effects. The causes are :

C_1 : Side x is less than sum of y and z

C_2 : Side y is less than sum of x and z

C_3 : Side z is less than sum of x and y

C_4 : Side x is equal to side y

C_5 : Side x is equal to side z

C_6 : Side y is equal to side z

The effects are :

e_1 : Not a triangle

e_2 : Scalene triangle

e_3 : Isosceles triangle.

e_4 : Equilateral triangle

e_5 : Impossible

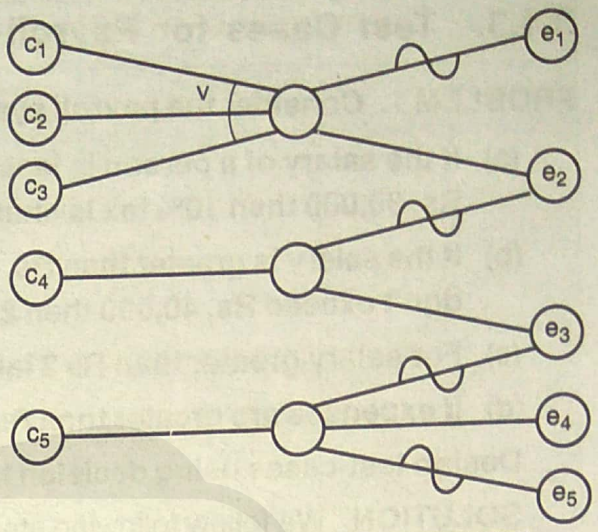


Fig. 3.18. Cause Effect Graph for Triangle Problem.

Step 2. It's cause-effect graph is shown in Fig. 3.18.

Step 3. We transform it into a decision table :

Conditions	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}
C_1 : x < y + z ?	0	1	1	1	1	1	1	1	1	1	1
C_2 : y < x + z ?	×	0	1	1	1	1	1	1	1	1	1
C_3 : z < x + y ?	×	×	0	1	1	1	1	1	1	1	1
C_3 : x = y ?	×	×	×	1	1	1	1	0	0	0	0
C_4 : x = z ?	×	×	×	1	1	0	0	1	1	0	0
C_5 : x = z ?	×	×	×	1	1	0	0	1	1	0	0
C_6 : y = z ?	×	×	×	1	0	1	0	1	0	1	0
e_1 : Not a triangle	1	1	1								
e_2 : Scalene											1
e_3 : Isosceles							1		1	1	
e_4 : Equilateral				1							
e_5 : Impossible					1	1		1			

Step 4. Since there are 11 rules, so we get 11 test cases and they are :

Test Case	x	y	z	Expected Output
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

3.4.3. Test Cases for Payroll Problem

PROBLEM 1. Consider the payroll system of a person :

(a) If the salary of a person is less than equal to Rs. 70,000 and expenses do not exceed Rs. 30,000 then 10% tax is charged by IT department.

(b) If the salary is greater than Rs. 60,000 and less than equal to Rs 2 lakhs and expenses don't exceed Rs. 40,000 then 20% tax is charged by IT department.

(c) For salary greater than Rs 2 lakhs, 5% additional surcharge is also charged.

(d) If expenses are greater than Rs. 40,000 surcharge is 9%.

Design test-cases using decision table based testing technique.

SOLUTION. We follow following steps :

Step 1. All 'cause' and their 'effects' are identified, i.e.,

Causes	Effects
C_1 : Salary < = 70,000 C_2 : Salary > 60,000 and Salary < = 2 lacs C_3 : Salary > 2 lacs C_4 : Expenses < = 30,000 C_5 : Expenses < = 40,000 C_6 : Expenses > 40,000	E_1 : 10% tax is charged. E_2 : 20% tax is charged. E_3 : (20% tax) + (5% surcharge) is charged. E_4 : (20% tax) + (9% surcharge) is charged.

Step 2. It's cause-effect graph is drawn.

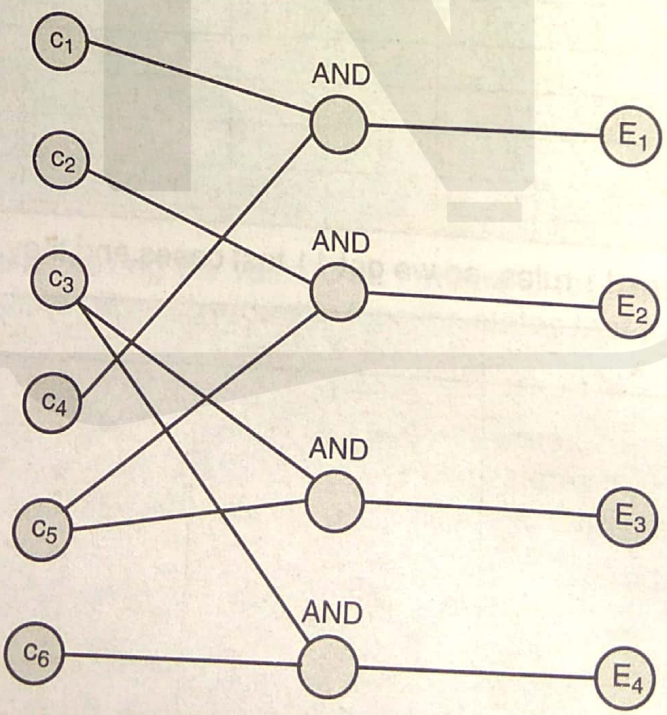


Fig. 3.19. Cause Effects Graph.

Step 3. We *transform* this cause-effect graph into a decision table. **Please note that** these '*causes*' and '*effects*' are nothing else but '*conditions*' and '*actions*' of our decision table. So, we get :

		1	2	3	4
Conditions (or causes)	C_1	1	0	0	0
	C_2	0	1	0	0
	C_3	0	0	1	1
	C_4	1	0	0	0
	C_5	0	1	1	0
	C_6	0	0	0	1
Actions (or Effects)	E_1	×	—	—	—
	E_2	—	×	—	—
	E_3	—	—	×	—
	E_4	—	—	—	×

Fig. 3.20. Decision Table.

That is, if C_1 and C_4 are 1 (or true) then the effect (or action) is E_1. Similarly, if C_2 and C_5 is 1 (or true), action to be taken is E_2 and so on.

Step 4. Since there are 4 rules in our decision table above, so we must have at least 4 test cases to test this system using this technique.

These test cases can be :

1. Salary = 20,000, Expenses = 2000
2. Salary = 1,00,000 , Expenses = 10,000
3. Salary = 3,00,000, Expenses = 20,000
4. Salary = 3,00,000, Expenses = 50,000.

So, we can say that a decision table is used to derive the test cases which can also take into account the boundary values.

3.4.4. Guidelines for Cause-Effect Functional Testing Technique

1. If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.
2. If the variables are independent, domain testing and equivalence class testing are indicated.
3. If the variables are dependent, decision table testing is indicated.
4. If the single-fault assumption is warranted, boundary value analysis (BVA) and robustness testing are indicated.
5. If the multiple-fault assumption is warranted, worst-case testing, robust worst-case testing and decision table testing are identical.
6. If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7. If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.