STQM-2

Functional Testing

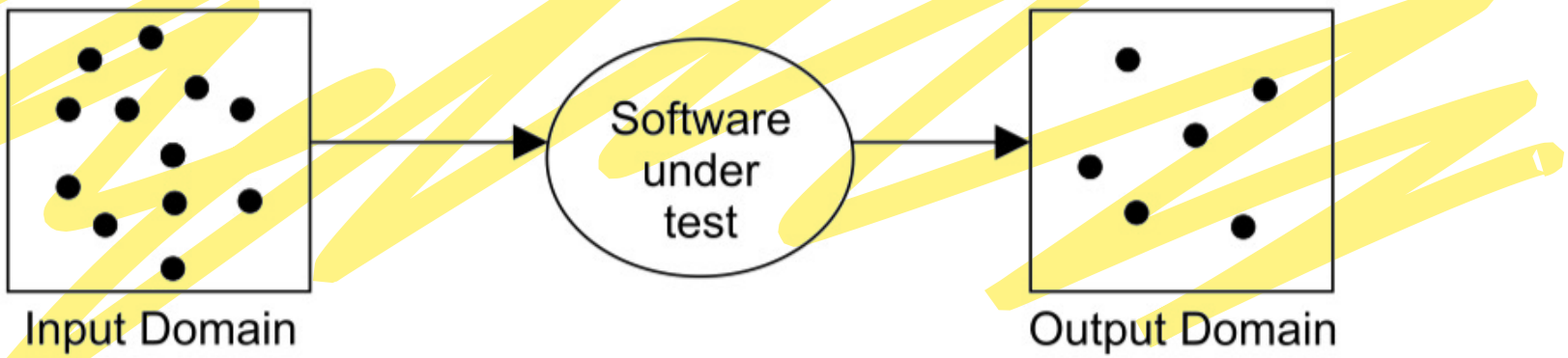
Functional Testing

- If observed behaviour of the software is different from the expected behaviour, we treat this as a failure condition.
- Failure is a dynamic condition that always occurs after the execution of the software.
- Everyone is in search of such test cases which may make the software fail and every technique attempts to find ways to design those test cases which have a higher probability of showing a failure.

Functional Testing

- Functional testing techniques attempt to design those test cases which have a higher probability of making a software fail.
- These techniques also attempt to test every possible functionality of the software.
- Test cases are designed on the basis of functionality and the internal structure of the program is completely ignored.
- Observed output(s) is (are) compared with expected output(s) for selected input(s) with preconditions, if any.

Functional Testing



Every dot in the input domain represents a set of inputs and every dot in the output domain represents a set of outputs.

Every set of input(s) will have a corresponding set of output(s).

The test cases are designed on the basis of user requirements without considering the internal structure of the program.

This black box knowledge is sufficient to design a good number of test cases.

Functional Testing

- In functional testing techniques, execution of a program is essential and hence these testing techniques come under the category of 'validation'.
- Here, both valid and invalid inputs are chosen to see the observed behaviour of the program.
- These techniques can be used at all levels of software testing like unit, integration, system and acceptance testing.
- They also help the tester to design efficient and effective test cases to find faults in the software

Functional Testing-BOUNDARY VALUE ANALYSIS

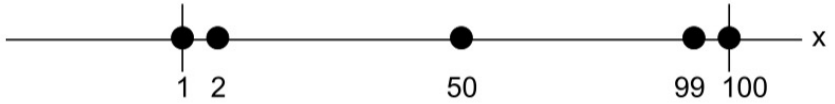
- Input values and design test cases with input values that are on or close to boundary values
- Experience has shown that such test cases have a higher probability of detecting a fault in the software.
- In boundary value analysis, we select values on or close to boundaries and all input values may have one of the following:

Functional Testing- Testing-BOUNDARY VALUE ANALYSIS

- (i) Minimum value
- (ii) Just above minimum value
- (iii) Maximum value
- (iv) Just below maximum value
- (v) Nominal (Average) value

Functional Testing

- Suppose there is a program ‘Square’ which takes ‘x’ as an input and prints the square of ‘x’as output. The range of ‘x’ is from 1 to 100
- These five values (1, 2, 50, 99 and 100) are selected on the basis of boundary value analysis and give reasonable confidence about the correctness of the program.



- There is no need to select all 100 inputs and execute the program one by one for all 100 inputs.
- The number of inputs selected by this technique is 4n + 1 where ‘n’ is the number of inputs. One nominal value is selected which may represent all values which are neither close to boundary nor on the boundary.

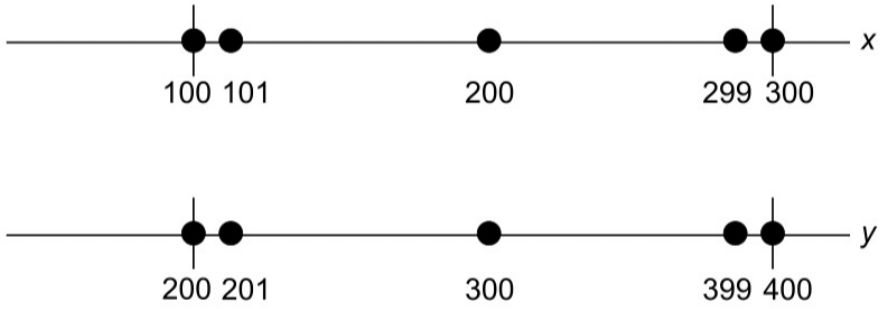
Functional Testing-BOUNDARY VALUE ANALYSIS

Table 2.1. Test cases for the 'Square' program

Test Case	Input x	Expected output
1.	1	1
2.	2	4
3.	50	2500
4.	99	9801
5.	100	10000

Functional Testing BOUNDARY VALUE ANALYSIS

- Consider a program ‘Addition’ with two input values x and y and it gives the addition of x and y as an output. The range of both input values are given as:

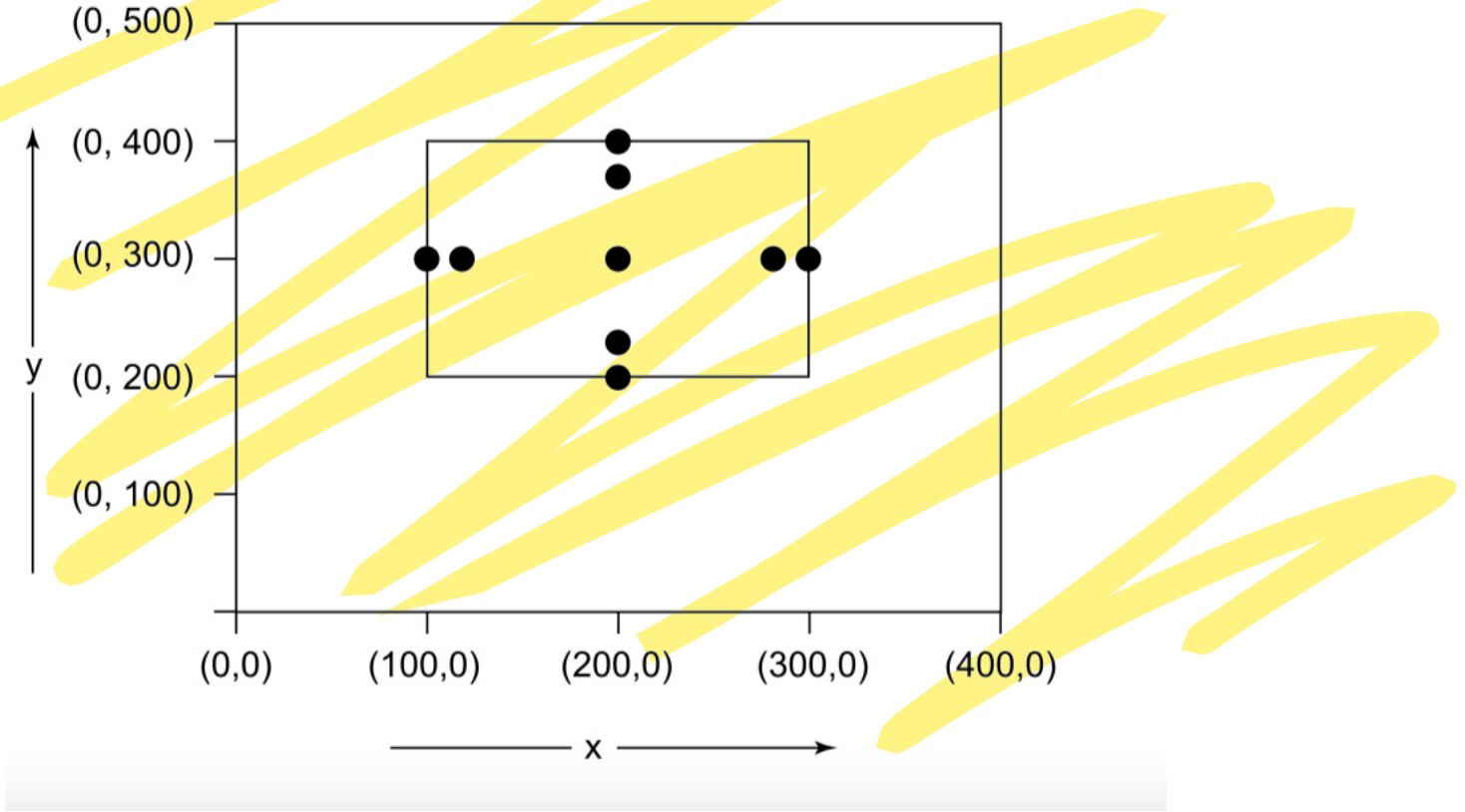


- The ‘x’ and ‘y’ inputs are required for the execution of the program.

Functional Testing BOUNDARY VALUE ANALYSIS

- We also consider 'single fault' assumption theory of reliability which says that failures are rarely the result of the simultaneous occurrence of two (or more) faults. Normally, one fault is responsible for one failure. With this theory in mind, we select one input value on
- boundary (minimum),
- just above boundary (minimum+),
- just below boundary (maximum–),
- on boundary (maximum),
- nominal (average)

Functional Testing BOUNDARY VALUE ANALYSIS



Functional Testing BOUNDARY VALUE ANALYSIS

Table 2.2. Test cases for the program 'Addition'

Test Case	x	y	Expected Output
1.	100	300	400
2.	101	300	401
3.	200	300	500
4.	299	300	599
5.	300	300	600
6.	200	200	400
7.	200	201	401
8.	200	300	500
9.	200	399	599
10.	200	400	600

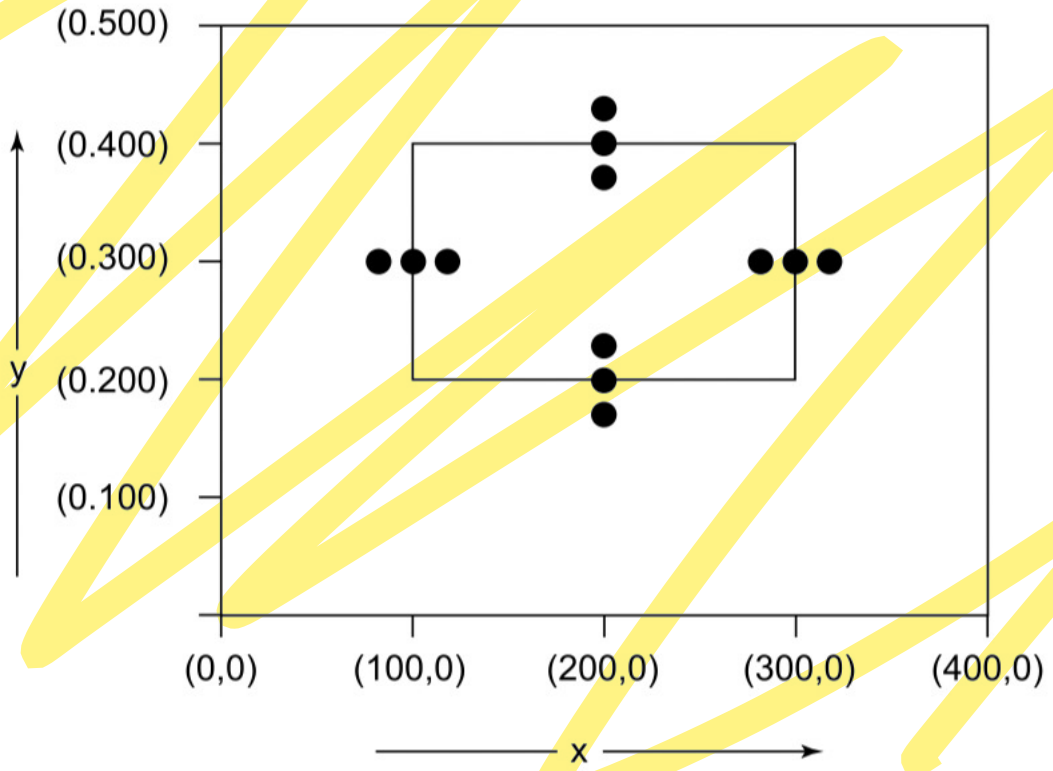
Robustness Testing

- Invalid values are also important to check the behaviour of the program
- we also select invalid value
- Hence, two additional states are added i.e.
  - just below minimum value (minimum value –)
  - and just above maximum value (maximum value +).
- This extended form of boundary value analysis is known as robustness testing.

Robustness Testing

- the total test cases in robustness testing are 6n + 1, where ‘n’ is the number of input values.
- All input values may have one of the following values:
- (i) Minimum value
- (ii) Just above minimum value
- (iii) Just below minimum value
- (iv) Just above maximum value
- (v) Just below maximum value
- (vi) Maximum value
- (vii) Nominal (Average) value

Robustness Testing



Robustness Testing

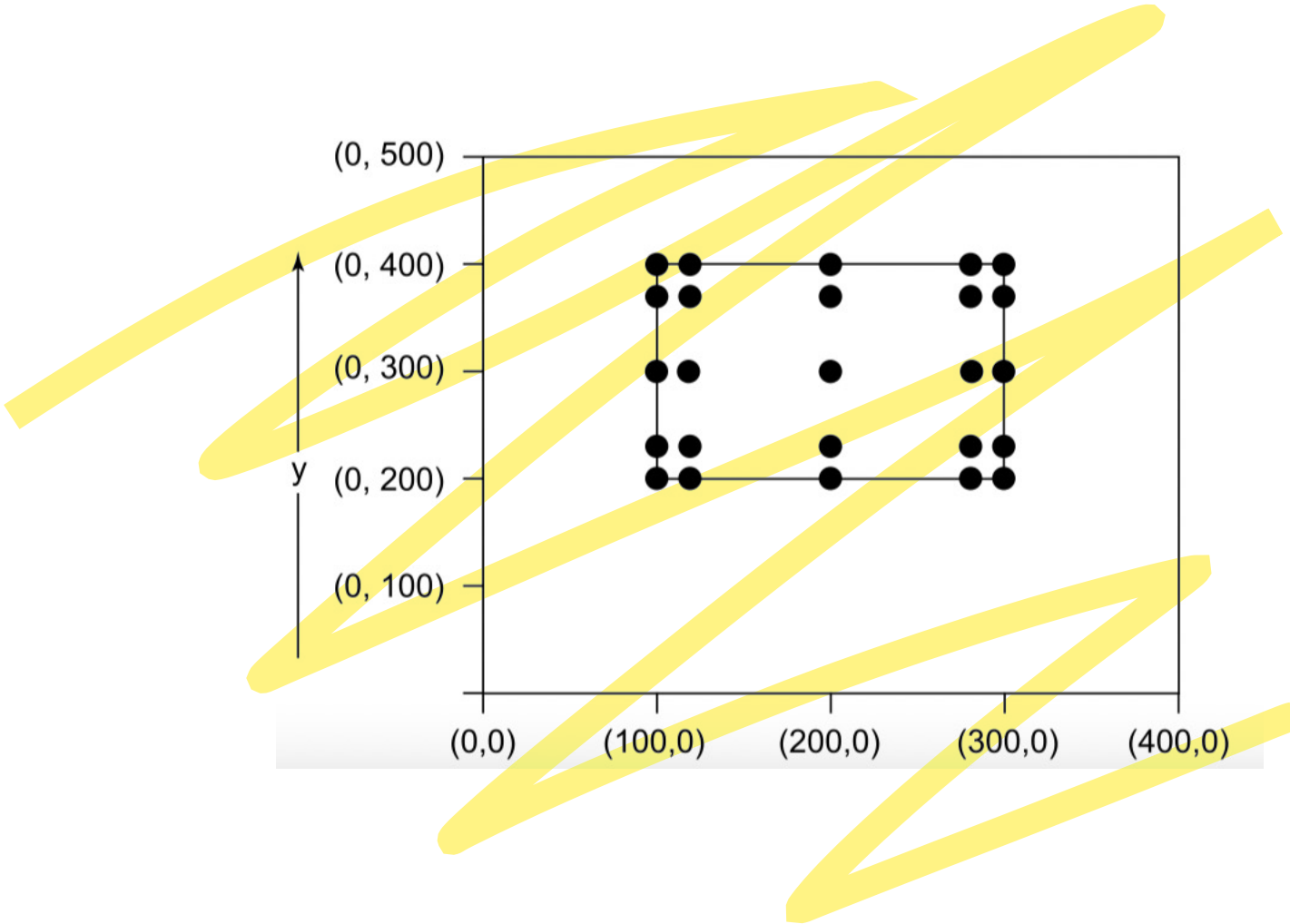
Table 2.7. Robustness test cases for two input values x and y

Test Case	x	y	Expected Output
1.	99	300	Invalid Input
2.	100	300	400
3.	101	300	401
4.	200	300	500
5.	299	300	599
6.	300	300	600
7.	301	300	Invalid Input
8.	200	199	Invalid Input
9.	200	200	400
10.	200	201	401
11.	200	399	599
12.	200	400	600
13.	200	401	Invalid Input

Worst-Case Testing

- Don’t consider the ‘single fault’ assumption theory of reliability. Now, failures are also due to occurrence of more than one fault simultaneously
- The restriction of one input value at any of the above mentioned values and other input values must be at nominal is not valid in worst-case testing.
- This will increase the number of test cases from 4n + 1 test cases to 5 n test cases, where ‘n’ is the number of input values.

Worst-Case Testing



Worst-Case Testing

Table 2.8. Worst test cases for the program 'Addition'

Test Case	x	y	Expected Output
1.	100	200	300
2.	100	201	301
3.	100	300	400
4.	100	399	499
5.	100	400	500
6.	101	200	301
7.	101	201	302
8.	101	300	401
9.	101	399	500
10.	101	400	501
11.	200	200	400
12.	200	201	401
13.	200	300	500
14.	200	399	599

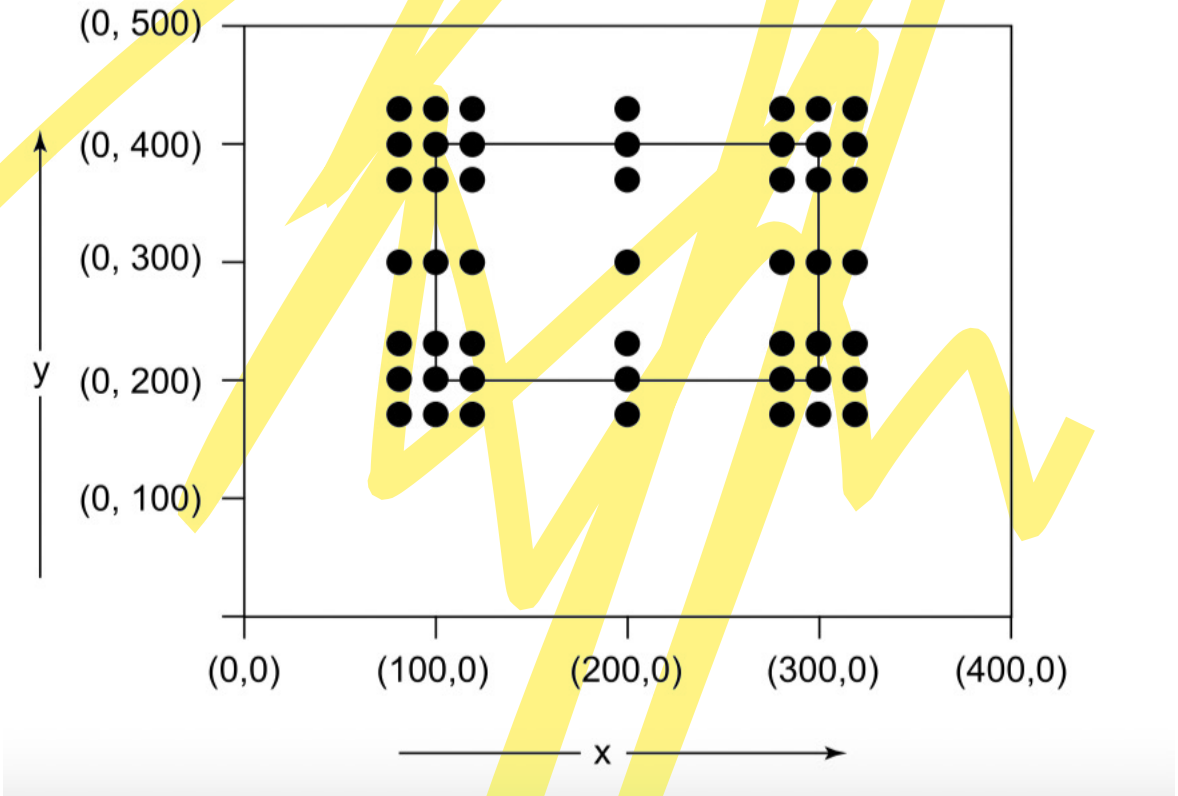
Worst-Case Testing

Test Case	x	y	Expected Output
15.	200	400	600
16.	299	200	499
17.	299	201	500
18.	299	300	599
19.	299	399	698
20.	299	400	699
21.	300	200	500
22.	300	201	501
23.	300	300	600
24.	300	399	699
25.	300	400	700

Robust Worst-Case Testing

- In robustness testing, we add two more states i.e. just below minimum value (minimum value–) and just above maximum value (maximum value+).
- We also give invalid inputs and observe the behaviour of the program. A program should be able to handle invalid input values, otherwise it may fail and give unexpected output values. ‘There are seven states (minimum - , minimum, minimum + , nominal, maximum – , maximum, maximum +) and a total of 7 n test cases will be generated.

Robust Worst-Case Testing



Applicability-Boundary Value Analysis

- Boundary value analysis is a simple technique and may prove to be effective when used correctly.
- Here, input values should be independent which restricts its applicability in many programs.
- This technique does not make sense for Boolean variables where input values are TRUE and FALSE only, and no choice is available for nominal values, just above boundary values, just below boundary values, etc.
- This technique can significantly reduce the number of test cases and is suited to programs in which input values are within ranges or within sets. This is equally applicable at the unit, integration, system and acceptance test levels.

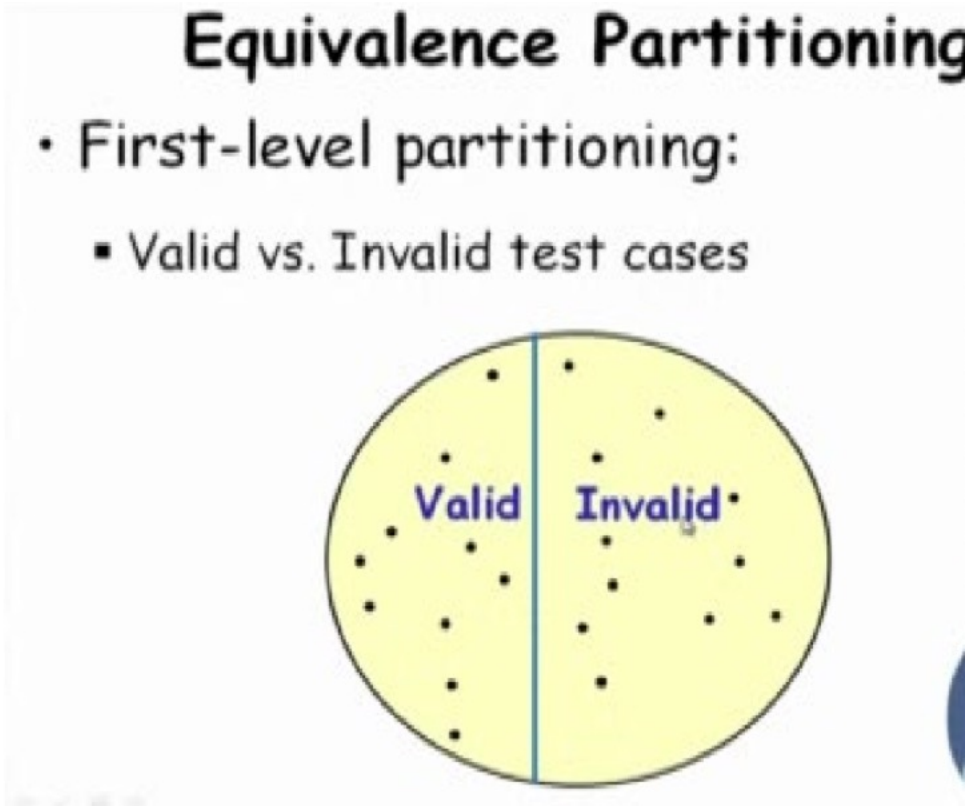
EQUIVALENCE CLASS TESTING

- We may divide input domain into various categories with some relationship and expect that every test case from a category exhibits the same behaviour.
- If categories are well selected, we may assume that if one representative test case works correctly, others may also give the same results.
- This assumption allows us to select exactly one test case from each category and if there are four categories, four test cases may be selected.
- Each category is called an equivalence class and this type of testing is known as equivalence class testing.

EQUIVALENCE CLASS TESTING

- Main problem here in equivalence partitioning is to design the equivalence classes, and
- Choose one value from each equivalence class, and also to consider if the equivalence classes have any boundary.
- we may assume that if one representative test case works correctly, others may also give the same results.
- This assumption allows us to select exactly one test case from each category and if there are four categories, four test cases may be selected.
- Each category is called an equivalence class and this type of testing is known as equivalence class testing.

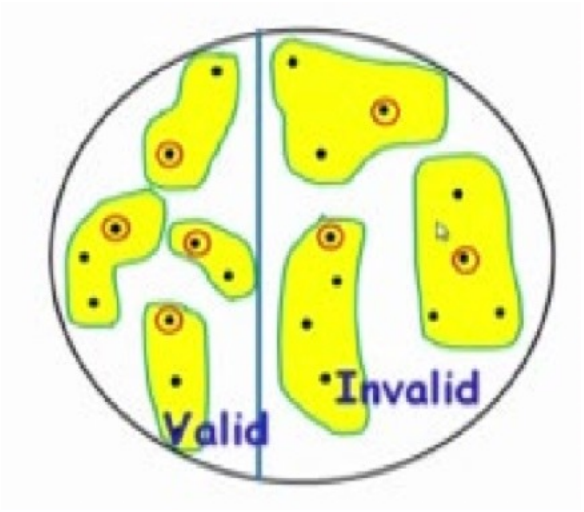
EQUIVALENCE CLASS TESTING



EQUIVALENCE CLASS TESTING

Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



EQUIVALENCE CLASS TESTING

- given any unit for which you have to design equivalence class, there

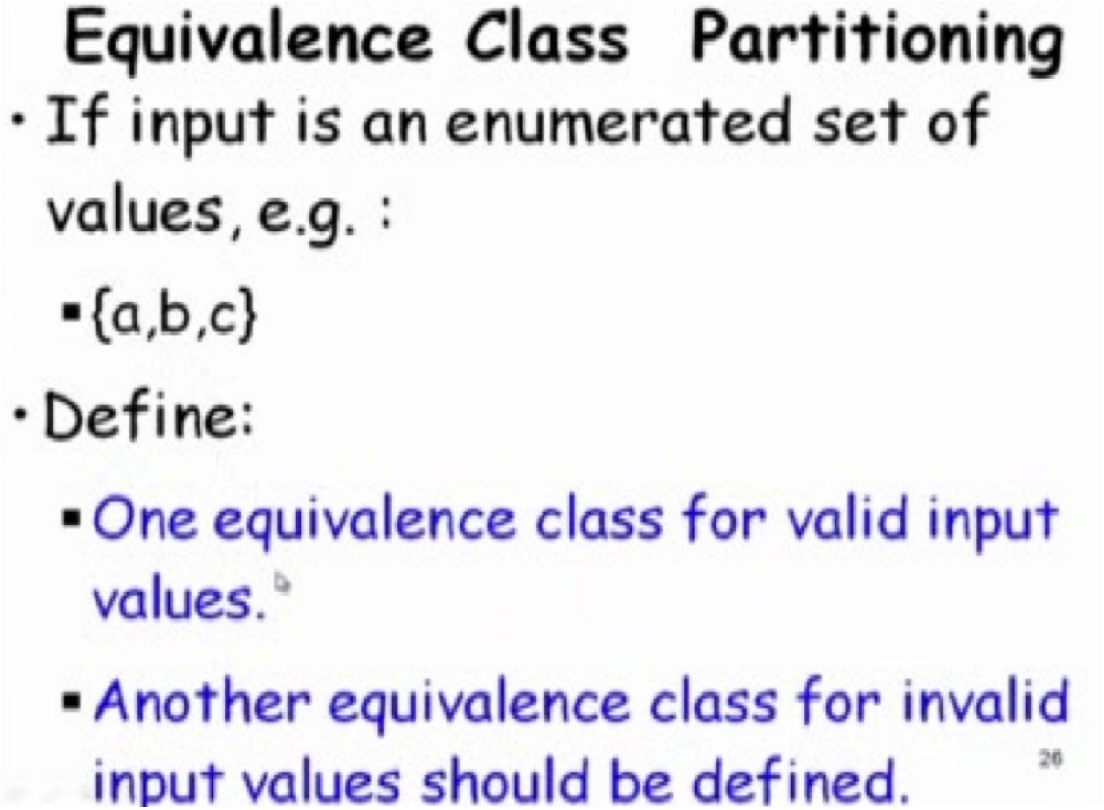
are at least two - one are the invalid set of values and the other are the valid set of values, and each of them will have typically different sets of equivalence classes.

EQUIVALENCE CLASS TESTING

Equivalence Class Partitioning

- If the input data to the program is specified by a range of values:
  - e.g. numbers between 1 to 5000.
  - One valid and two invalid equivalence classes are defined.

EQUIVALENCE CLASS TESTING



if the input value is enumerated
set of equivalence classes then we have 1 valid equivalence class and 1 invalid equivalence class, which is either a, b, c or which is neither of a, b, c.

Equivalence Partitioning: Example 1

- Example: **Image Fetch-image(URL)**
  - **Equivalence Definition 1:** Partition based on URL protocol ("http", "https", "ftp", "file", etc.)
  - **Equivalence Definition 2:** Partition based on type of file being retrieved (HTML, GIF, JPEG, Plain Text, etc.)

Let us say, we have a function, name of the function is Fetch-
image it takes a URL and returns an image. So what will be the equivalence classes for this function? Fetch-image, which takes URL and returns an image - in this case, we can have multiple definition of equivalence and therefore we have to consider all of them. For example, one equivalence class can be, the URL can be based on “http”, “https’, “ftp”, “file”, etcetera all these are valid URL categories. Then we have to consider each of them as an equivalence class for the input.

Similarly, the value that is stored in the URL can be different types of images. For example,
HTML, GIF, JPEG, etcetera, Plain Text, this can be our images.

- The input domain equivalence classes for the program ‘Square’ which takes ‘x’ as an input (range 1-100) and prints the square of ‘x’ (seen in Figure 2.2) are given as:
- (i) I 1 = { 1 x 100}(Valid input range from 1 to 100)
- (ii) I 2 = { x < 1 } (Any invalid input where x is less than 1)
- (iii) I 3 = { x > 100 } (Any invalid input where x is greater than 100)

Table 2.18. Test cases for program ‘Square’ based on input domain

Test Case	Input x	Expected Output
I_1	0	Invalid Input
I_2	50	2500
I_3	101	Invalid Input

EQUIVALENCE CLASS TESTING

For Addition of two numbers:

(i) $I_1 = \{ 100 \le x \le 300 \text{ and } 200 \le y \le 400 \}$ (Both x and y are valid values)
(ii) $I_2 = \{ 100 \le x \le 300 \text{ and } y < 200 \}$ (x is valid and y is invalid)
(iii) $I_3 = \{ 100 \le x \le 300 \text{ and } y > 400 \}$ (x is valid and y is invalid)
(iv) $I_4 = \{ x < 100 \text{ and } 200 \le y \le 400 \}$ (x is invalid and y is valid)
(v) $I_5 = \{ x > 300 \text{ and } 200 \le y \le 400 \}$ (x is invalid and y is valid)
(vi) $I_6 = \{ x < 100 \text{ and } y < 200 \}$ (Both inputs are invalid)
(vii) $I_7 = \{ x < 100 \text{ and } y > 400 \}$ (Both inputs are invalid)
(viii) $I_8 = \{ x > 300 \text{ and } y < 200 \}$ (Both inputs are invalid)
(ix) $I_9 = \{ x > 300 \text{ and } y > 400 \}$ (Both inputs are invalid)

EQUIVALENCE CLASS TESTING

Graphical representation of inputs

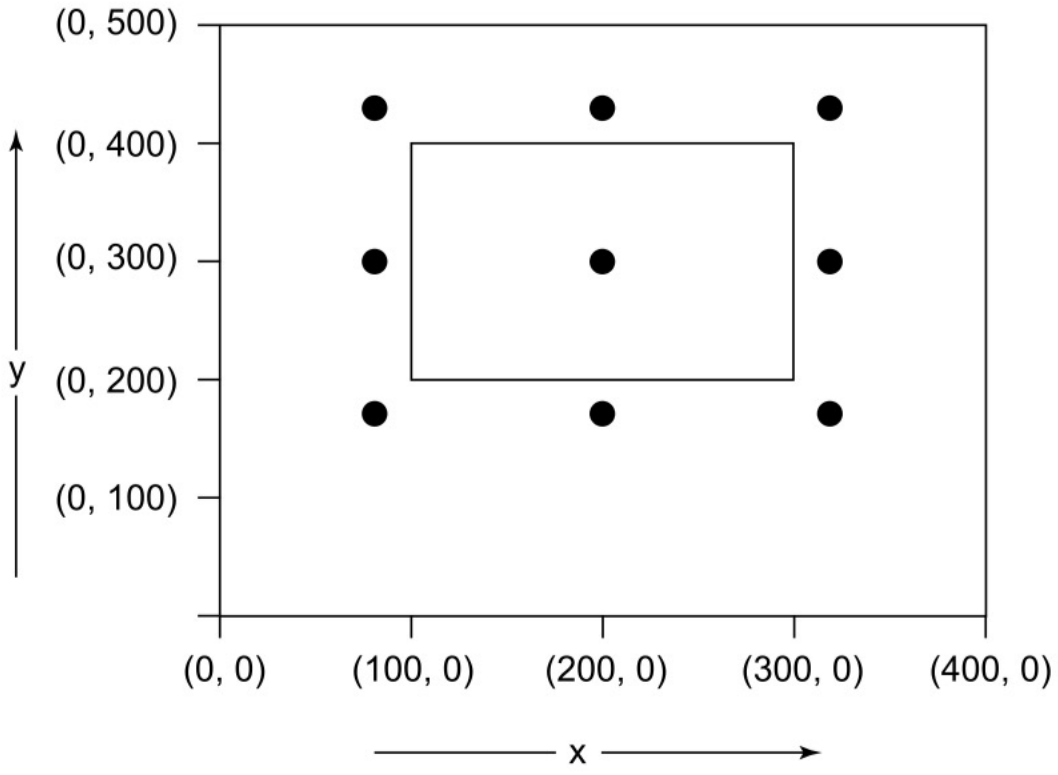


Table 2.19. Test cases for the program 'Addition'

Test Case	x	y	Expected Output
I_1	200	300	500
I_2	200	199	Invalid input
I^3	200	401	Invalid input
I_4	99	300	Invalid input
I_5	301	300	Invalid input
I_6	99	199	Invalid input
I_7	99	401	Invalid input
I_8	301	199	Invalid input
I_9	301	401	Invalid input

EQUIVALENCE CLASS TESTING

- Thus, for ‘Square’ program, the output domain equivalence classes are given as:

- O 1 = {square of the input number ‘x’}
- O 2 = {Invalid input)

Table 2.20. Test cases for program ‘Square’ based on output domain

Test Case	Input x	Expected Output
O_1	50	2500
O_2	0	Invalid Input

EQUIVALENCE CLASS TESTING

O_1 = { Addition of two input numbers x and y }

O_2 = {Invalid Input}

The test cases are given in Table 2.21.

Table 2.21. Test cases for program 'Addition' based on output domain

Test Case	x	y	Expected Output
O_1	200	300	500
O_2	99	300	Invalid Input

EQUIVALENCE CLASS TESTING

- Three different types of invalid equivalence testing

EQUIVALENCE CLASS TESTING

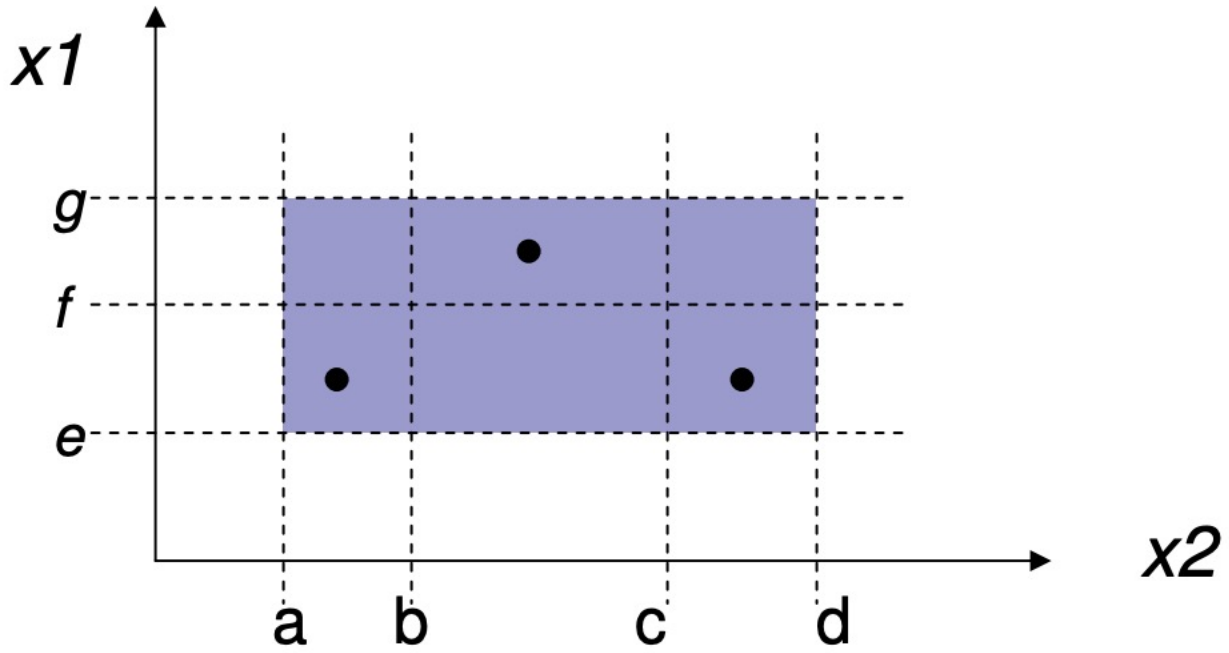
- Weak Normal Equivalence Class Testing
  - Weak equivalence class testing is based on the single fault assumption, stating that rarely is an error caused as a result of two or more faults occurring simultaneously. Therefore weak equivalence class testing only takes one variable from each equivalence class

- Strong Normal Equivalence Class Testing
  - Conversely Strong Equivalence Class testing is based on the multiple assumption which states that errors will result in a combination of faults. Therefore strong equivalence class testing tests every combination of elements formed as a result of the Cartesian product of the Equivalence relation

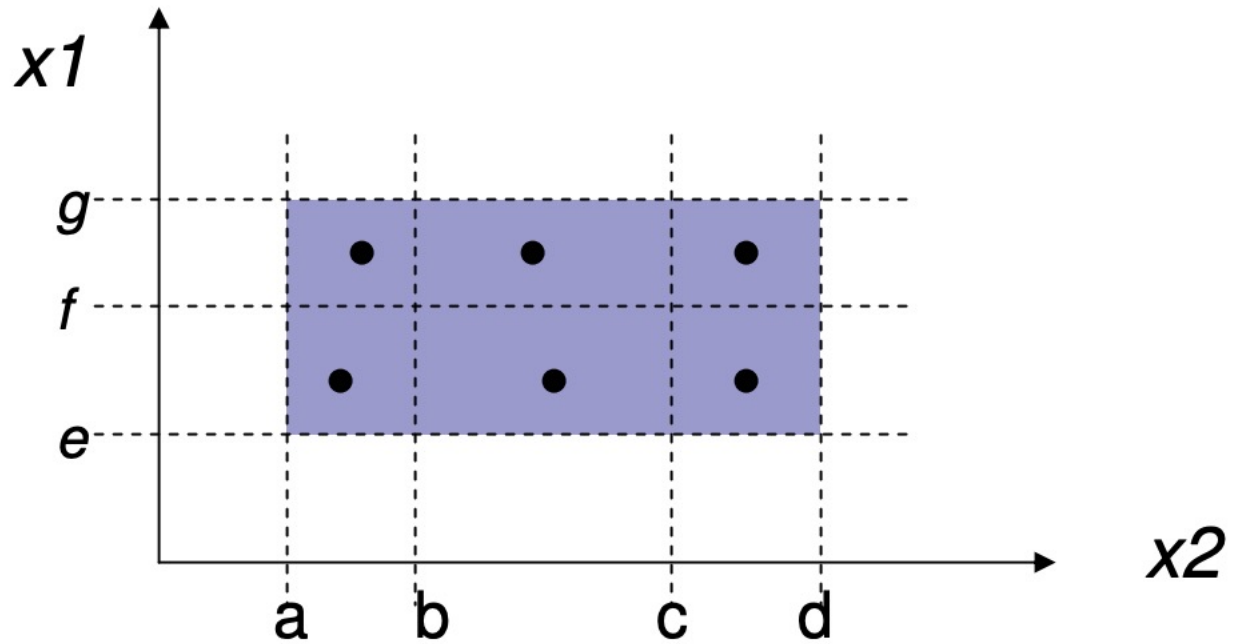
EQUIVALENCE CLASS TESTING

- Weak Robust Equivalence Class Testing
  - As with weak normal Equivalence Class testing we only test for one variable from each Equivalence Class. However we now also test for invalid values as well. Since weak Equivalence Class Testing is based on the single fault assumption a test case will have one invalid value and the remaining values will all be valid.
- Strong Robust Equivalence Class Testing
  - This form of Equivalence Class testing produces test cases for all valid and invalid elements of the Cartesian product of all the equivalence classes.

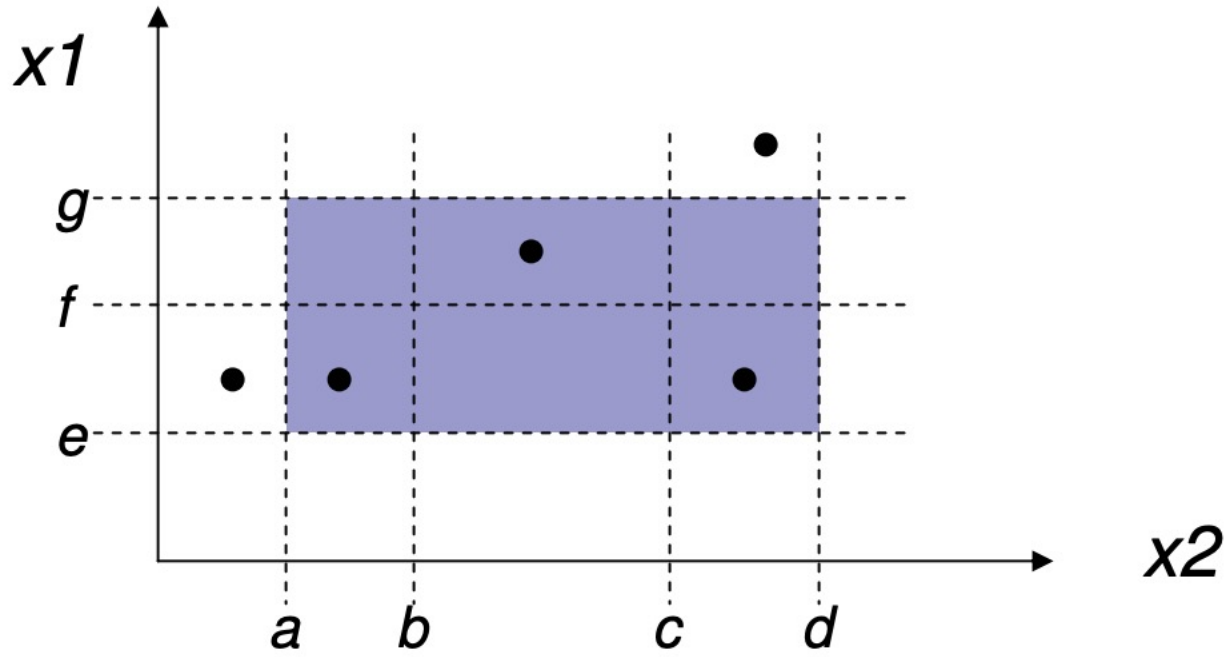
Weak Normal Equivalence Class Testing



Strong Normal Equivalence Class Testing



Weak Robust Equivalence Class Testing



Strong Robust Equivalence Class Testing

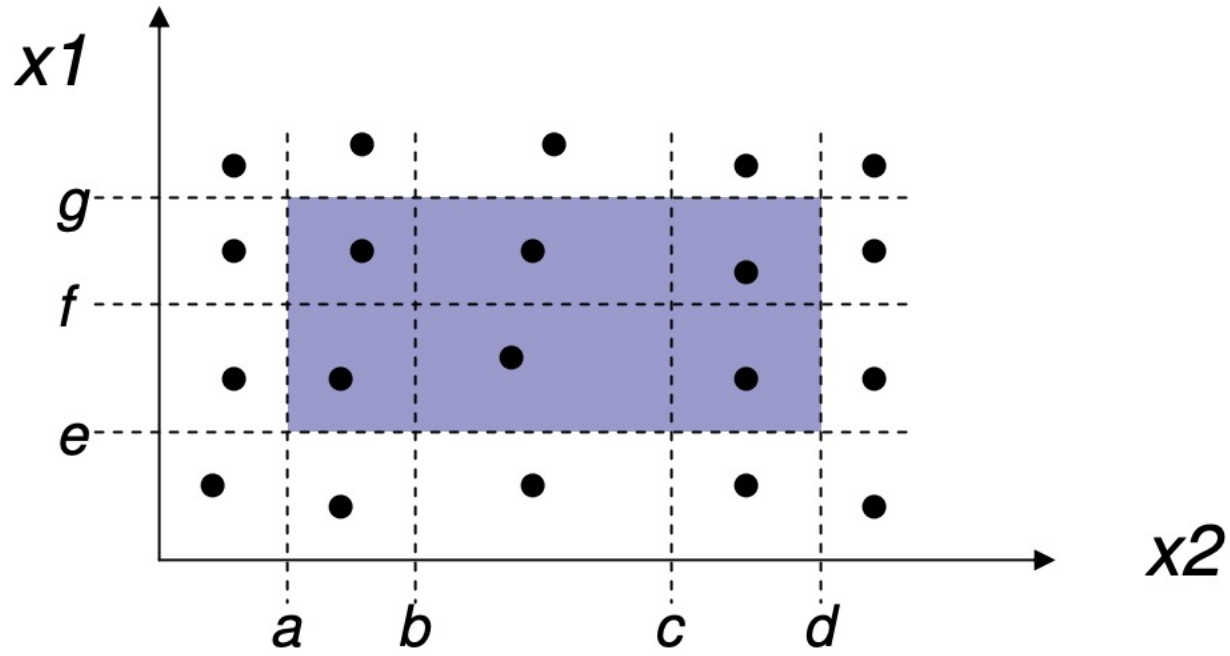


Figure 3.4

Applicability

- It is applicable at unit, integration, system and acceptance test levels. The basic requirement is that inputs or outputs must be partitioned based on the requirements and every partition will give a test case.
- The selected test case may test the same thing, as would have been tested by another test case of the same equivalence class, and if one test case catches a bug, the probably will too.
- If one test case does not find a bug, the other test cases of the same equivalence class may also not find any bug. We do not consider dependencies among different variables while designing equivalence classes.
- The design of equivalence classes is subjective and two testing persons may design two different sets of partitions of input and output domains.
- This is understandable and correct as long as the partitions are reviewed and all agree that they acceptably cover the program under test.

Decision Table Testing

- An output may be dependent on many input conditions and decision tables give a pictorial view of various combinations of input conditions. There are four portions of the decision table:
  - **Condition Stubs**: All the conditions are represented in this upper left section of the decision table. These conditions are used to determine a particular action or set of actions.
  - **Action Stubs:** All possible actions are listed in this lower left portion of the decision table.
  - **Condition Entries:** In the condition entries portion of the decision table, we have a number of columns and each column represents a rule. Values entered in this upper right portion of the table are known as inputs.

Decision Table Testing

- **Action Entries:** Each entry in the action entries portion has some associated action or set of actions in this lower right portion of the table. These values are known as outputs and are dependent upon the functionality of the program.

- Decision table testing technique is used to design a complete set of test cases without using the internal structure of the program. Every column is associated with a rule and generates a test case.

Table 2.30. Decision table

	Stubs	Entries
Condition	c_1 c_2 c_3	
Action	a_1 a_2 a_3 a_4	

Four Portions

1. Condition Stubs
2. Condition Entries
3. Action Stubs
4. Action Entries

Decision Table Testing

Table 2.31. Typical structure of a decision table

Stubs	R_1	R_2	R_3	R_4
c_1	F	T	T	T
c_2	-	F	T	T
c_3	-	-	F	T
a_1	X	X		X
a_2			X	
a_3	X			

Decision Table Testing

- input values are only True (T) or False (F), which are binary conditions. The decision tables which use only binary conditions are known as limited entry decision tables.
- The decision tables which use multiple conditions where a condition may have many possibilities instead of only ‘true’ and ‘false’ are known as extended entry decision tables

Decision Table Testing

- However, this is applicable only for limited entry decision tables where only ‘true’ and ‘false’ conditions are considered.
- Hence, the actual number of columns in any decision table is the sum of the rule counts of every column shown in the decision table.

Decision Table Testing

- Consider a program for the determination of the largest amongst three numbers. Its input is a triple of positive integers (say x,y and z) and values are from interval [1, 300].

Table 2.33. Decision table

c_1: x > = 1?	F	T	T	T	T	T	T	T	T	T	T	T	T	T
c_2: x <= 300?	-	F	T	T	T	T	T	T	T	T	T	T	T	T
c_3: y > = 1?	-	-	F	T	T	T	T	T	T	T	T	T	T	T
c_4: y <= 300?	-	-	-	F	T	T	T	T	T	T	T	T	T	T
c_5: z > = 1?	-	-	-	-	F	T	T	T	T	T	T	T	T	T
c_6: z <= 300?	-	-	-	-	-	F	T	T	T	T	T	T	T	T
c_7: x>y?	-	-	-	-	-	-	T	T	T	T	F	F	F	F
c_8: y>z?	-	-	-	-	-	-	T	T	F	F	T	T	F	F
c_9: z>x?	-	-	-	-	-	-	T	F	T	F	T	F	T	F
Rule Count	256	128	64	32	16	8	1	1	1	1	1	1	1	1
a_1 : Invalid input	X	X	X	X	X	X								
a_2 : x is largest								X		X				
a_3 : y is largest											X	X		
a_4 : z is largest									X				X	
a_5 : Impossible							X							X

Decision Table Testing

Example 2.3: Consider a program for classification of a triangle. Its input is a triple of positive integers (say a, b, c) and the input parameters are greater than zero and less than or equal to 100.
The triangle is classified according to the following rules:

Right angled triangle: $c^2 = a^2 + b^2$ or $a^2 = b^2 + c^2$ or $b^2 = c^2 + a^2$
Obtuse angled triangle: $c^2 > a^2 + b^2$ or $a^2 > b^2 + c^2$ or $b^2 > c^2 + a^2$
Acute angled triangle: $c^2 < a^2 + b^2$ and $a^2 < b^2 + c^2$ and $b^2 < c^2 + a^2$

The program output may have one of the following words:
[Acute angled triangle, Obtuse angled triangle, Right angled triangle, Invalid triangle]

Decision Table Testing

Table 2.32. Decision table for triangle problem

Condition	c_1: $a < b + c$?	F	T	T	T	T	T	T	T	T	T	T
	c_2: $b < c + a$?	-	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: $a^2 = b^2 + c^2$?	-	-	-	T	T	T	T	F	F	F	F
	c_5: $a^2 > b^2 + c^2$?	-	-	-	T	T	F	F	T	T	F	F
	c_6: $a_2 < b_2 + c_2$?	-	-	-	T	F	T	F	T	F	T	F
	Rule Count	32	16	8	1	1	1	1	1	1	1	1
Action	a_1 : Invalid triangle	X	X	X								
	a_2 : Right angled triangle							X				
	a_3 : Obtuse angled triangle									X		
	a_4 : Acute angled triangle										X	
	a_5 : Impossible				X	X	X		X			X

Decision Table Testing

- The ‘do not care’ conditions are represented by the ‘-‘sign. A ‘do not care’ condition has no effect on the output.
- The term ‘rule count’ is used with ‘do not care’ entries in the decision table and has a value 1, if ‘do not care’ conditions are not there, but it doubles for every ‘do not care’ entry. Hence each ‘do not care’ condition counts for two rules. Rule count can be calculated as:
  - Rule count = 2 ^ number of do not care conditions

Applicablity-Decision Table

- Decision tables are popular in circumstances where an output is dependent on many conditions and a large number of decisions are required to be taken.
- They may also incorporate complex business rules and use them to design test cases.
- Every column of the decision table generates a test case. As the size of the program increases, handling of decision tables becomes difficult and cumbersome.
- In practice, they can be applied easily at unit level only. System testing and integration testing may not find its effective applications.

Example

Example 2.2: Consider a program for the determination of division of a student based on the marks in three subjects. Its input is a triple of positive integers (say mark1, mark2, and mark3) and values are from interval [0, 100].
The division is calculated according to the following rules:

Marks Obtained (Average)	Division
75 – 100	First Division with distinction
60 – 74	First division
50 – 59	Second division
40 – 49	Third division
0 – 39	Fail

Total marks obtained are the average of marks obtained in the three subjects i.e.

$$\text{Average} = (\text{mark1} + \text{mark 2} + \text{mark3}) / 3$$

The program output may have one of the following words:
[Fail, Third Division, Second Division, First Division, First Division with Distinction]

O_1 = { <mark1, mark2, mark3> : First Division with distinction if average > = 75 }

O_2 = { <mark1, mark2, mark3> : First Division if 60 ≤ average ≤ 74}

O_3 = { <mark1, mark2, mark3> : Second Division if 50 ≤ average ≤ 59 }

O_4 = { <mark1, mark2, mark3> : Third Division if 40 ≤ average ≤ 49 }

O_5 = { <mark1, mark2, mark3> : Fail if average <40 }

O_6 = { <mark1, mark2, mark3> : Invalid marks if marks are not between 0 to 100 }

The test cases generated by output domain are given in Table 2.24.

Table 2.24. Output domain test cases

Test Case	mark1	mark2	mark3	Expected Output
O_1	75	80	85	First division with distinction
O_2	68	68	68	First division
O_3	55	55	55	Second division
O_4	45	45	45	Third division
O_5	25	25	25	Fail
O_6	-1	50	50	Invalid marks

I_3 = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ } (mark1 is valid, mark2 is invalid and mark3 is valid)

I_4 = { $0 \leq mark1 \leq 100$ and $0 \leq mark2 \leq 100$ and $mark3 < 0$ } (mark1 is valid, mark2 is valid and mark3 is invalid)

I_5 = { $mark1 > 100$ and $0 \leq mark2 \leq 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is valid and mark3 is valid)

I_6 = ($0 \leq mark1 \leq 100$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is valid, mark2 is invalid and mark3 is valid)

I_7 = { $0 \leq mark1 \leq 100$ and $0 \leq mark2 \leq 100$ and $mark3 > 100$ } (mark 1 is valid, mark2 is valid and mark3 is invalid)

I_8 = { $mark1 < 0$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_9 = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $mark3 < 0$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{10} = { $mark1 < 0$ and $0 \leq mark2 \leq 100$ and $mark3 < 0$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{11} = { $mark1 > 100$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_{12} = { $0 \leq mark1 \leq 100$ and $mark2 > 100$ and $mark3 > 100$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

I_{13} = { $mark1 > 100$ and $0 \leq mark2 \leq 100$ and $mark3 > 100$ } (mark1 is invalid, mark2 is valid and mark3 is invalid)

I_{14} = { $mark1 < 0$ and $mark2 > 100$ and $0 \leq mark3 \leq 100$ } (mark1 is invalid, mark2 is invalid and mark 3 is valid)

I_{15} = { $mark1 > 100$ and $mark2 < 0$ and $0 \leq mark3 \leq 100$ }{ (mark1 is invalid, mark2 is invalid and mark3 is valid)

I_{16} = { $0 \leq mark1 \leq 100$ and $mark2 < 0$ and $mark3 > 100$ } (mark1 is valid, mark2 is invalid and mark3 is invalid)

Test Case	mark1	mark2	mark3	Expected Output
I_1	50	50	50	Second division
I_2	–1	50	50	Invalid marks
I_3	50	–1	50	Invalid marks
I_4	50	50	–1	Invalid marks
I_5	101	50	50	Invalid marks
I_6	50	101	50	Invalid marks
I_7	50	50	101	Invalid marks
I_8	–1	–1	50	Invalid marks
I_9	50	–1	–1	Invalid marks
I_{10}	–1	50	–1	Invalid marks
I_{11}	101	101	50	Invalid marks
I_{12}	50	101	101	Invalid marks
I_{13}	101	50	101	Invalid marks
I_{14}	–1	101	50	Invalid marks
I_{15}	101	–1	50	Invalid marks
I_{16}	50	–1	101	Invalid marks
I_{17}	50	101	–1	Invalid marks
I_{18}	–1	50	101	Invalid marks
I_{19}	101	50	–1	Invalid marks
I_{20}	–1	–1	–1	Invalid marks
I_{21}	101	101	101	Invalid marks
I_{22}	–1	–1	101	Invalid marks
I_{23}	–1	101	–1	Invalid marks
I_{24}	101	–1	–1	Invalid marks

Extended Decision Table

- I 1 = { A1 : 0 mark1 100 }
- I 2 = { A2 : mark1 < 0 }
- I 3 = { A3 : mark1 > 100 }
- I 4 = { B1 : 0 mark2 100 }
- I 5 = {B2 : mark2 < 0 }
- I 6 = { B3 : mark2 > 100 }
- I 7 = { C1 : 0 mark3 100 }
- I 8 = { C2 : mark3 < 0 }
- I 9 = { C3 : mark3 > 100 }

Extended Decision Table

$I_{10} = \{ D1 : 0 \leq avg \leq 39 \}$

$I_{11} = \{ D2 : 40 \leq avg \leq 49 \}$

$I_{12} = \{ D3 : 50 \leq avg \leq 59 \}$

$I_{13} = \{ D4 : 60 \leq avg \leq 74 \}$

$I_{14} = \{ D5 : avg \geq 75 \}$

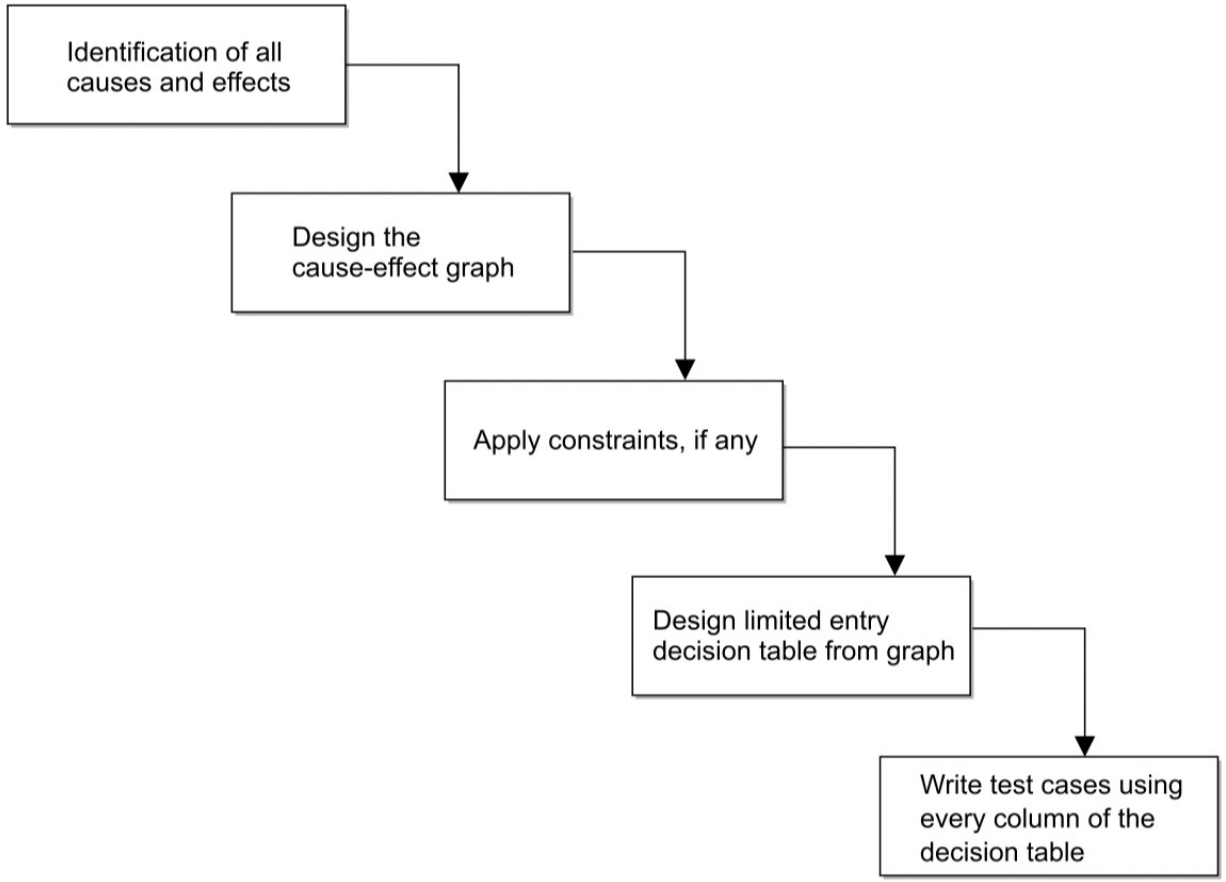
Table 2.37. Extended entry decision table

Conditions	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
c_1 : mark1 in	A1	A1	A1	A1	A1	A1	A1	A1	A1	A2	A3
c_2 : mark 2 in	B1	B1	B1	B1	B1	B1	B1	B2	B3	-	-
c_3 : mark3 in	C1	C1	C1	C1	C1	C2	C3	-	-	-	-
c_4 : avg in	D1	D2	D3	D4	D5	-	-	-	-	-	-
Rule Count	1	1	1	1	1	5	5	15	15	45	45
a_1: Invalid Marks						X	X	X	X	X	X
a_2: First Division with Distinction					X						
a_3: First Division				X							
a_4: Second Division			X								
a_5: Third Division		X									
a_6: Fail	X										

CAUSE-EFFECT GRAPHING TECHNIQUE

- This technique is a popular technique for small programs and considers the combinations of various inputs
- Two new terms are used here and these are causes and effects, which are nothing but inputs and outputs respectively.

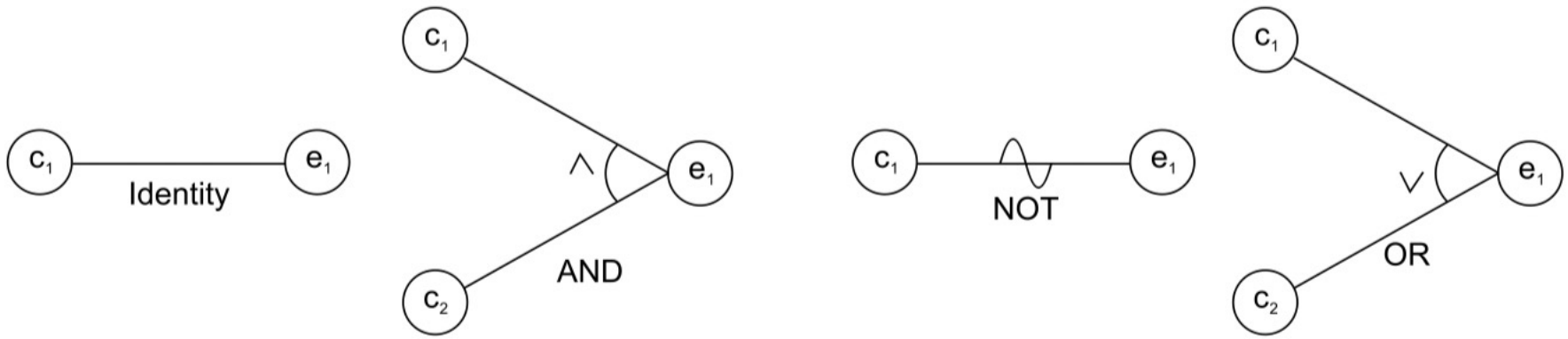
CAUSE-EFFECT GRAPHING TECHNIQUE



CAUSE-EFFECT GRAPHING TECHNIQUE

- Identification of Causes and Effects:
  - The SRS document is used for the identification of causes and effects. Causes which are inputs to the program and effects which are outputs of the program can easily be identified after reading the SRS document. A list is prepared for all causes and effects.

Design of Cause-Effect Graph The relationship amongst causes and effects are established using cause-effect graph. The basic notations of the graph



(a) Identity: This function states that if c_1 is 1, then e_1 is 1; else e_1 is 0.
(b) NOT: This function states that if c_1 is 1, then e_1 is 0; else e_1 is 1.
(c) AND: This function states that if both c_1 and c_2 are 1, then e_1 is 1; else e_1 is 0.
(d) OR: This function states that if either c_1 or c_2 is 1, then e_1 is 1; else e_1 is 0.

The AND and OR functions are allowed to have any number of inputs.

CAUSE-EFFECT GRAPHING TECHNIQUE

- Use of Constraints in Cause-Effect Graph
  - There may be a number of causes (inputs) in any program. We may like to explore the relationships amongst the causes and this process may lead to some impossible combinations of causes. Such impossible combinations or situations are represented by constraint symbols

(a) Exclusive

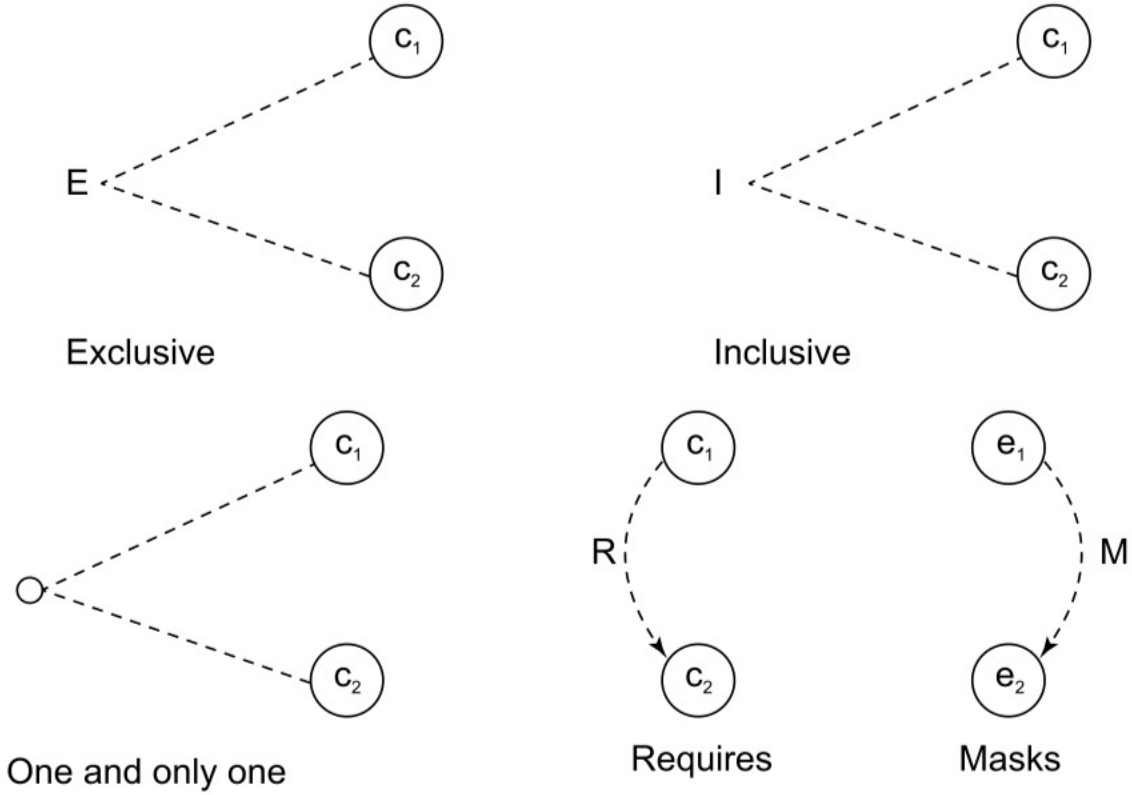
The Exclusive (E) constraint states that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). However, both c_1 and c_2 can be 0 simultaneously.

(b) Inclusive

The Inclusive (I) constraints states that at least one of c_1 or c_2 must always be 1. Hence, both cannot be 0 simultaneously. However, both can be 1.

(c) One and Only One

The one and only one (O) constraint states that one and only one of c_1 and c_2 must be 1.



(d) Requires

The requires (R) constraint states that for c_1 to be 1, c_2 must be 1; it is impossible for c_1 to be 1 if c_2 is 0.

(e) Mask

This constraint is applicable at the effect side of the cause-effect graph. This states that if effect e_1 is 1, effect e_2 is forced to be 0.

Example

Consider the example of keeping the record of marital status and number of children of a citizen. The value of marital status must be 'U' or 'M'. The value of the number of children must be digit or null in case a citizen is unmarried. If the information entered by the user is correct then an update is made. If the value of marital status of the citizen is incorrect, then the error message 1 is issued. Similarly, if the value of number of children is incorrect, then the error message 2 is issued.

The causes are:

c_1: marital status is 'U'

c_2: marital status is 'M'

c_3: number of children is a digit

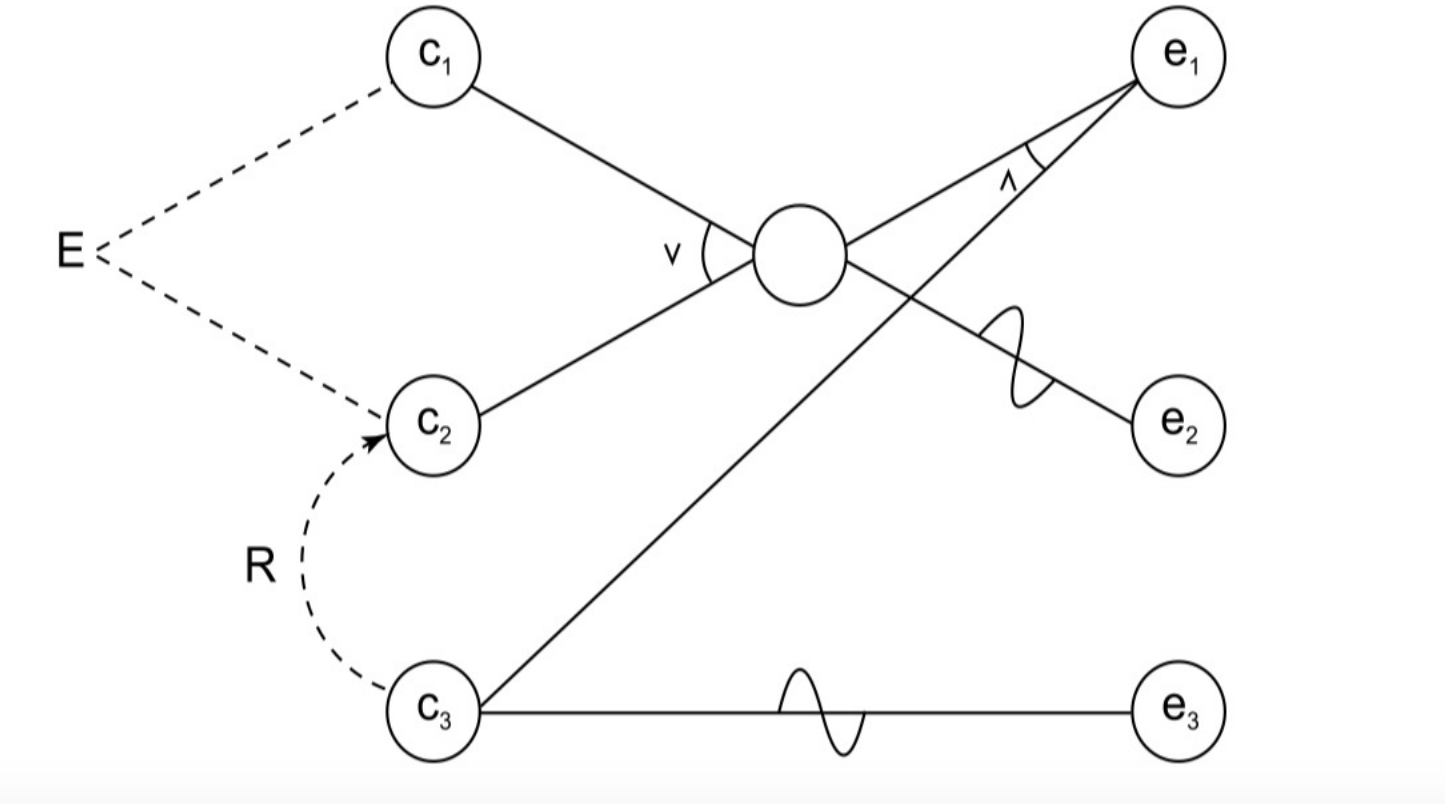
and the effects are:

e_1: updation made

e_2: error message 1 is issued

e_3: error message 2 is issued

Example-Solution



Applicability

- Cause-effect graphing is a systematic method for generating test cases. It considers dependency of inputs using some constraints.
- This technique is effective only for small programs because, as the size of the program increases, the number of causes and effects also increases and thus complexity of the cause effect graph increases. For large-sized programs, a tool may help us to design the cause-effect graph with the minimum possible complexity.
- It has very limited applications in unit testing and hardly any application in integration testing and system testing.

Example 2

- A tourist of age greater than 21 years and having a clean driving record is supplied a rental car. A premium amount is also charged if the tourist is on business, otherwise it is not charged.
- If the tourist is less than 21 year old, or does not have a clean driving record, the system will display the following message:
- “Car cannot be supplied”

Solution: The causes are

c_1: Age is over 21
c_2: Driving record is clean
c_3: Tourist is on business

and effects are

e_1: Supply a rental car without premium charge.
e_2: Supply a rental car with premium charge
e_3: Car cannot be supplied

The cause-effect graph is shown in Figure 2.15 and decision table is shown in Table 2.45. The test cases for the problem are given in Table 2.46.

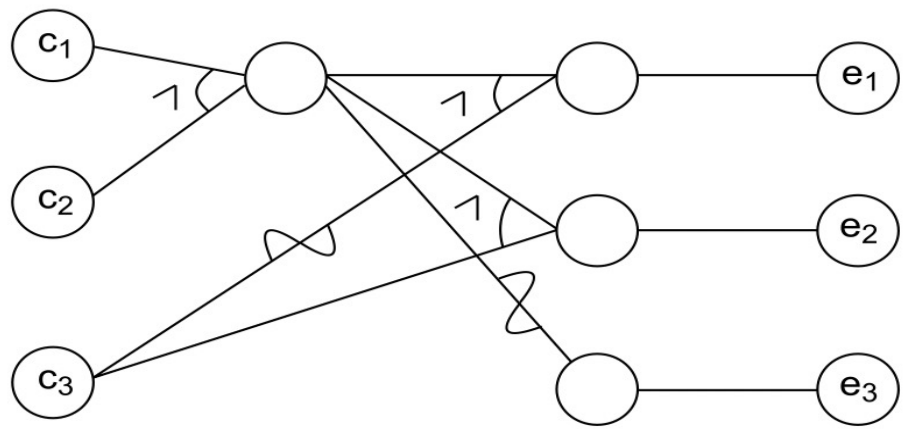


Table 2.45. Decision table of rental car problem

Conditions	1	2	3	4
c_1 : Over 21 ?	F	T	T	T
c_2 : Driving record clean ?	-	F	T	T
c_3 : On Business ?	-	-	F	T
e_1 : Supply a rental car without premium charge			X	
e_2 : Supply a rental car with premium charge				X
e_3 : Car cannot be supplied	X	X		

Table 2.46. Test cases of the given decision table

Test Case	Age	Driving_record_clean	On_business	Expected Output
1.	20	Yes	Yes	Car cannot be supplied
2.	26	No	Yes	Car cannot be supplied
3.	62	Yes	No	Supply a rental car without premium charge
4.	62	Yes	Yes	Supply a rental car with premium charge.

Reference