CHAPTER – 9
FUNCTIONAL TESTING

OUTLINE OF THE CHAPTER

- Functional Testing Concepts of Howden
- Different Types of Variables
- Test Vectors
- Howden’s Functional Testing Summary
- Pairwise Testing
- Orthogonal Array
- In Parameter Order
- Equivalence Class Partitioning
- Guidelines for Equivalence Class Partitioning
- Identification of Test Cases
- Advantages of Equivalence Class Partitioning
- Boundary Value Analysis (BVA)
- Guidelines for Boundary Value Analysis
- Decision Tables
- Random Testing
- Adaptive Random Testing
- Error Guessing
- Category Partition

FUNCTIONAL TESTING CONCEPTS

The four key concepts in functional testing are:

Precisely identify the domain of each input and each output variable

Select values from the data domain of each variable having important properties

Consider combinations of special values from different input domains to design test cases

Consider input values such that the program under test produces special values from the domains of the output variables

DIFFERENT TYPES OF VARIABLES

Numeric Variables

- A set of discrete values
- A few contiguous segments of values

Arrays

- An array holds values of the same type, such as integer and real. Individual elements of an array are accessed by using one or more indices.

Substructures

- A structure means a data type that can hold multiple data elements. In the field of numerical analysis, matrix structure is commonly used.

Subroutine Arguments

- Some programs accept input variables whose values are the names of functions. Such programs are found in numerical analysis and statistical applications

TEST VECTORS

A test vector is an instance of an input to a program, a.k.a. test data

If a program has *n* input variables, each of which can take on *k* special values, then there are k^n possible combinations of test vectors

We have more than one million test vectors even for $k = 3$ and $n = 20$

There is a need to identify a method for reducing the number of test vectors

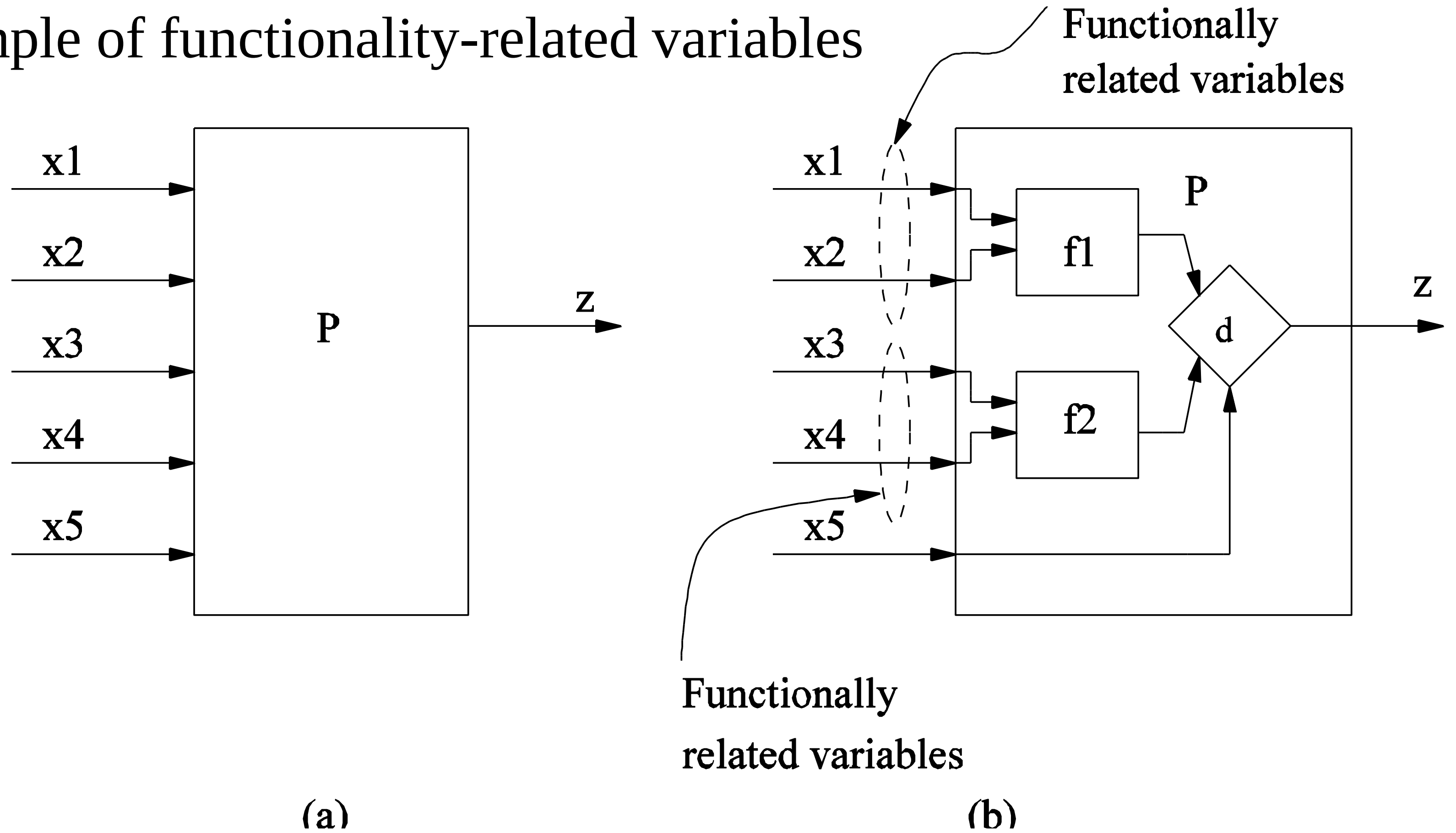
It is suggested that there is no need of combining values of all input variables to design a test vector, if the variables are not *functionally related*

It is difficult to give a formal definition of the idea of functionally related variables, but it is easy to identify them

- Variables appearing in the same assignment statement are functionally related
- Variables appearing in the same branch predicate – the condition part of an if statement, for example – are functionally related

TEST VECTORS

- Example of functionality-related variables


x1
x2
x3
x4
x5
P
z
(a)
Functionally related variables
x1
x2
x3
x4
x5
P
f1
f2
d
z
Functionally related variables
(b)

Functionality related variables

HOWDEN'S FUNCTIONAL TESTING SUMMARY

Let us summarize the main points in functional testing:

Identify the input and the output variables of the program and their data domains

Compute the expected outcomes as illustrated in Figure 9.5(a), for selected input values

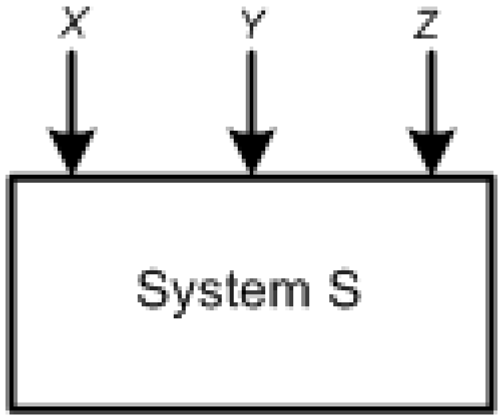
Determine the input values that will cause the program to produce selected outputs as illustrated in Figure 9.5(b).

PAIRWISE TESTING

Pairwise testing means that each possible combination of values for every pair of input variables is covered by at least one test case

Consider the system S in the figure, which has three input variables X, Y, and Z.

For the three given variables X, Y, and Z, their value sets are as follows: D(X) = {True, False}, D(Y) = {0, 5}, and D(Z) = {Q, R}

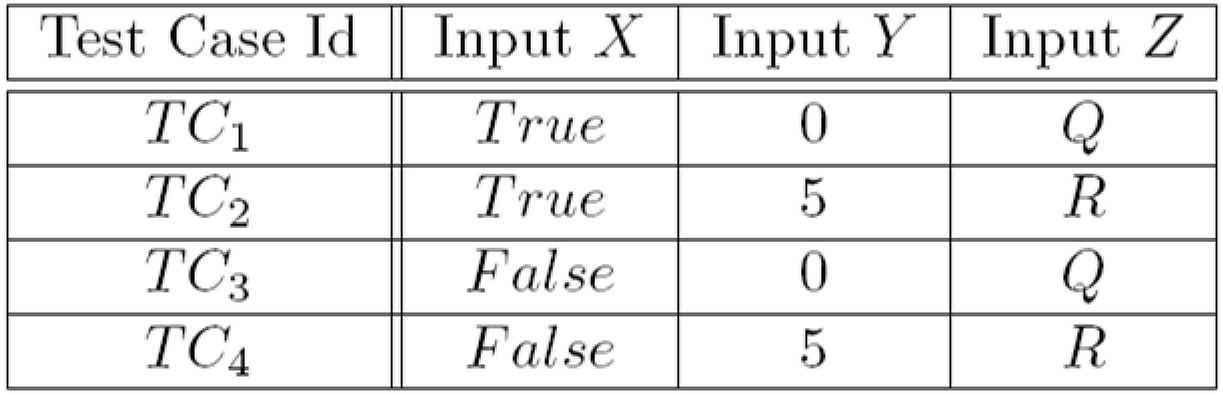


System S with three input variables.

PAIRWISE TESTING

The total number of all-combination test cases is 2 × 2 × 2 = 8

However, a subset of four test cases, as shown in Table 9.5, covers all pairwise combinations



Test Case Id	Input X	Input Y	Input Z
TC_1	$True$	0	Q
TC_2	$True$	5	R
TC_3	$False$	0	Q
TC_4	$False$	5	R

Table 9.5: Pairwise test cases for system S.

ORTHOGONAL ARRAY

Consider the two-dimensional array of integers shown in Table 9.6

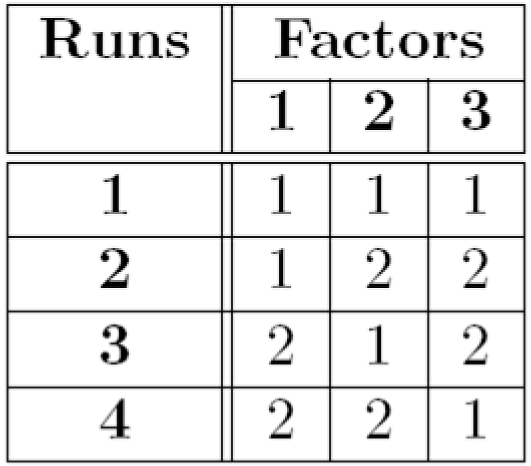
This is an example of $L_4(2^3)$ orthogonal array

The "4" indicates that the array has 4 rows, also known as runs

The "2^3" part indicates that the array has 3 columns, known as factors, and each cell in the array contains 2 different values, known as levels.

Levels mean the maximum number of values that a single factor can take on

Orthogonal arrays are generally denoted by the pattern $L_{Runs}(Levels^{Factors})$



Runs	**Factors**		
	1	**2**	**3**
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

Table 9.6: $L_4(2^3)$ orthogonal array.

ORTHOGONAL ARRAY

Let us consider our previous example of the system S.

The system S which has three input variables X, Y, and Z.

For the three given variables X, Y , and Z, their value sets are as follows: D(X) = {True, False}, D(Y) = {0, 5}, and D(Z) = {Q,R}.

Map the variables to the factors and values to the levels onto the $L_4(2^3)$ orthogonal array (Table 9.6) with the resultant in Table 9.5.

In the first column, let 1 = True, 2 = False.

In the second column, let 1 = 0, 2 = 5.

In the third column, let 1 = Q, 2 = R.

Note that, not all combinations of all variables have been selected

Instead combinations of all pairs of input variables have been covered with four test cases

ORTHOGONAL ARRAY

Orthogonal arrays provide a technique for selecting a subset of test cases with the following properties:

- It guarantees testing the pairwise combinations of all the selected variables
- It generates fewer test cases than a all-combination approach.
- It generates a test suite that has even distribution of all pairwise combinations
- It can be automated

ORTHOGONAL ARRAY

In the following, the steps of a technique to generate orthogonal arrays are presented. The steps are further explained by means of a detailed example.

Step 1: Identify the maximum number of independent input variables with which a system will be tested. This will map to the *factors* of the array—each input variable maps to a different factor.

Step 2: Identify the maximum number of values that each independent variable will take. This will map to the levels of the array.

Step 3: Find a suitable orthogonal array with the smallest number of runs $L_{\text{Runs}}(X^Y)$, where X is the number of levels and Y is the number of factors. A suitable array is one that has at least as many factors as needed from step 1 and has at least as many levels for each of those factors as identified in step 2.

Step 4: Map the variables to the factors and values of each variable to the levels on the array.

Step 5: Check for any "left-over" levels in the array that have not been mapped. Choose arbitrary valid values for those left-over levels.

Step 6: Transcribe the runs into test cases.

ORTHOGONAL ARRAY

Web Example. Consider a website that is viewed on a number of browsers with various plug-ins and operating systems (OSs) and through different connections as shown in Table 9.6. The table shows the variables and their values that are used as elements of the orthogonal array. We need to test the system with different combinations of the input values.

TABLE 9.6 Various Values That Need to Be Tested in Combinations

Variables	Values
Browser	Netscape, Internet Explorer (IE), Mozilla
Plug-in	Realplayer, Mediaplayer
OS	Windows, Linux, Macintosh
Connection	LAN, PPP, ISDN

Note: LAN, local-area network; PPP, Point-to-Point Protocal; ISDN, Integrated Services Digital Network.

ORTHOGONAL ARRAY

Following the steps laid out previously, let us design an orthogonal array to create a set of test cases for pairwise testing:

Step 1: There are four independent variables, namely, Browser, Plug-in, OS, and Connection.

Step 2: Each variable can take at most three values.

Step 3: An orthogonal array $L_9(3^4)$ as shown in Table 9.7 is good enough for the purpose. The array has nine rows, three levels for the values, and four factors for the variables.

Step 4: Map the variables to the factors and values to the levels of the array: the factor 1 to Browser, the factor 2 to Plug-in, the factor 3 to OS, and the factor 4 to Connection. Let 1 = Netscape, 2 = IE, and 3 = Mozilla in the Browser column. In the Plug-in column, let 1 = Realplayer and 3 = Mediaplayer. Let 1 = Windows, 2 = Linux, and 3 = Macintosh in the OS column. Let 1 = LAN, 2 = PPP, and 3 = ISDN in the Connection column. The mapping of the variables and the values onto the orthogonal array is given in Table 9.8.

ORTHOGONAL ARRAY

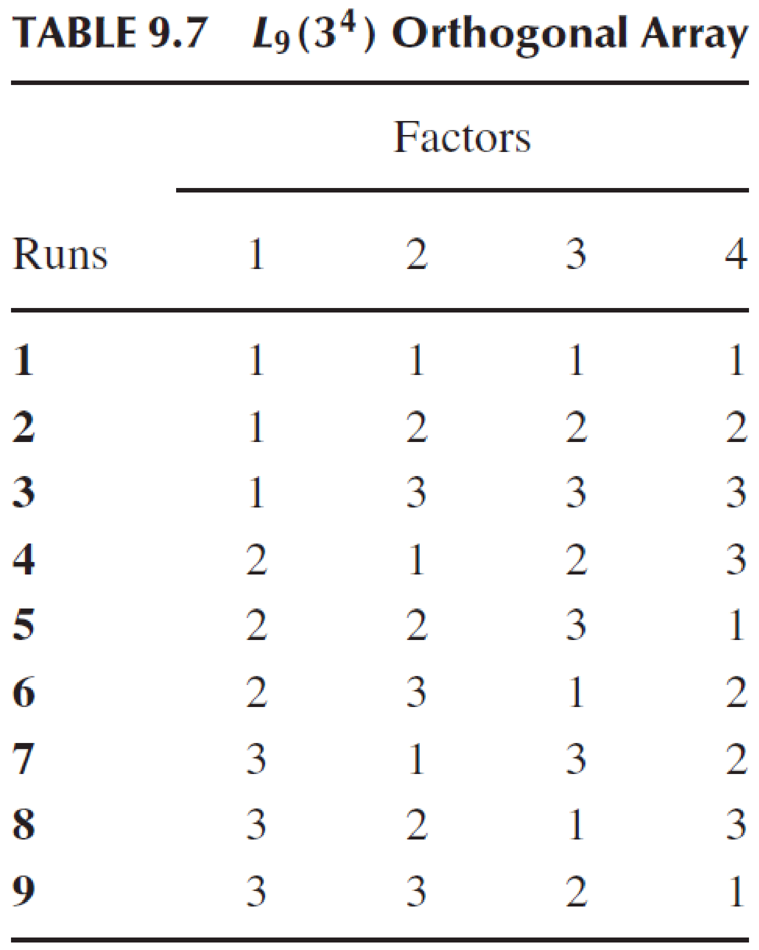


TABLE 9.7 $L_9(3^4)$ **Orthogonal Array**

	Factors			
Runs	1	2	3	4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

ORTHOGONAL ARRAY

Step 5: There are left-over levels in the array that are not being mapped. The factor 2 has three levels specified in the original array, but there are only two possible values for this variable. This has caused a level (2) to be left over for variable Plug-in after mapping the factors. One must provide a value in the cell. The choice of this value can be arbitrary, but to have a coverage, start at the top of the Plug-in column and cycle through the possible values when filling in the left-over levels. Table 9.9 shows the mapping after filling in the remaining levels using the cycling technique mentioned.

Step 6: We generate nine test cases taking the test case values from each run. Now let us examine the result:

- Each Browser is tested with every Plug-in, with every OS, and with every Connection.
- Each Plug-in is tested with every Browser, with every OS, and with every Connection.
- Each OS is tested with every Browser, with every Plug-in, and with every Connection.
- Each Connection is tested with every Browser, with every Plug-in, and with every OS.

ORTHOGONAL ARRAY

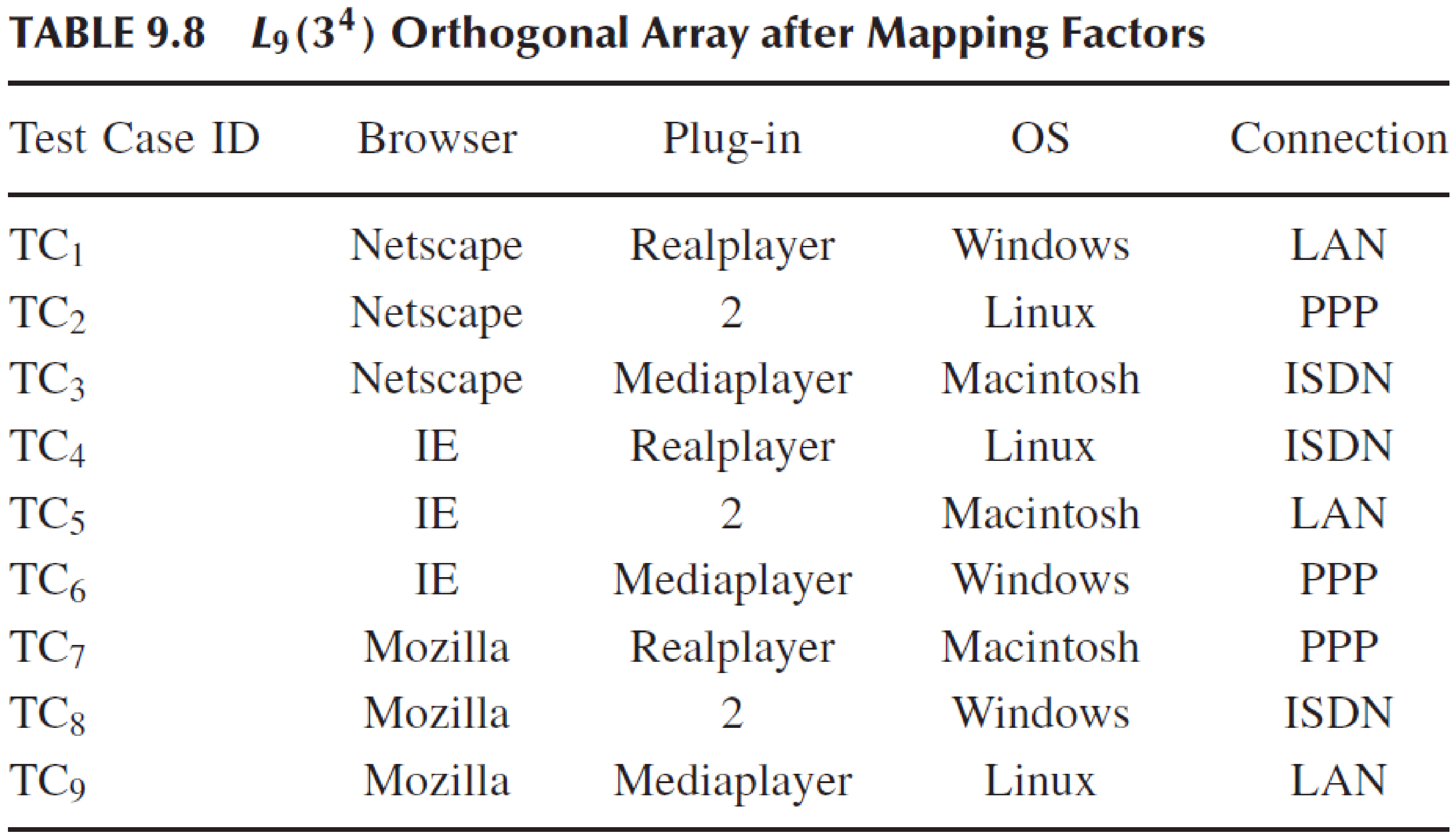


TABLE 9.8 $L_9(3^4)$ **Orthogonal Array after Mapping Factors**

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	2	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	2	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	2	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

ORTHOGONAL ARRAY

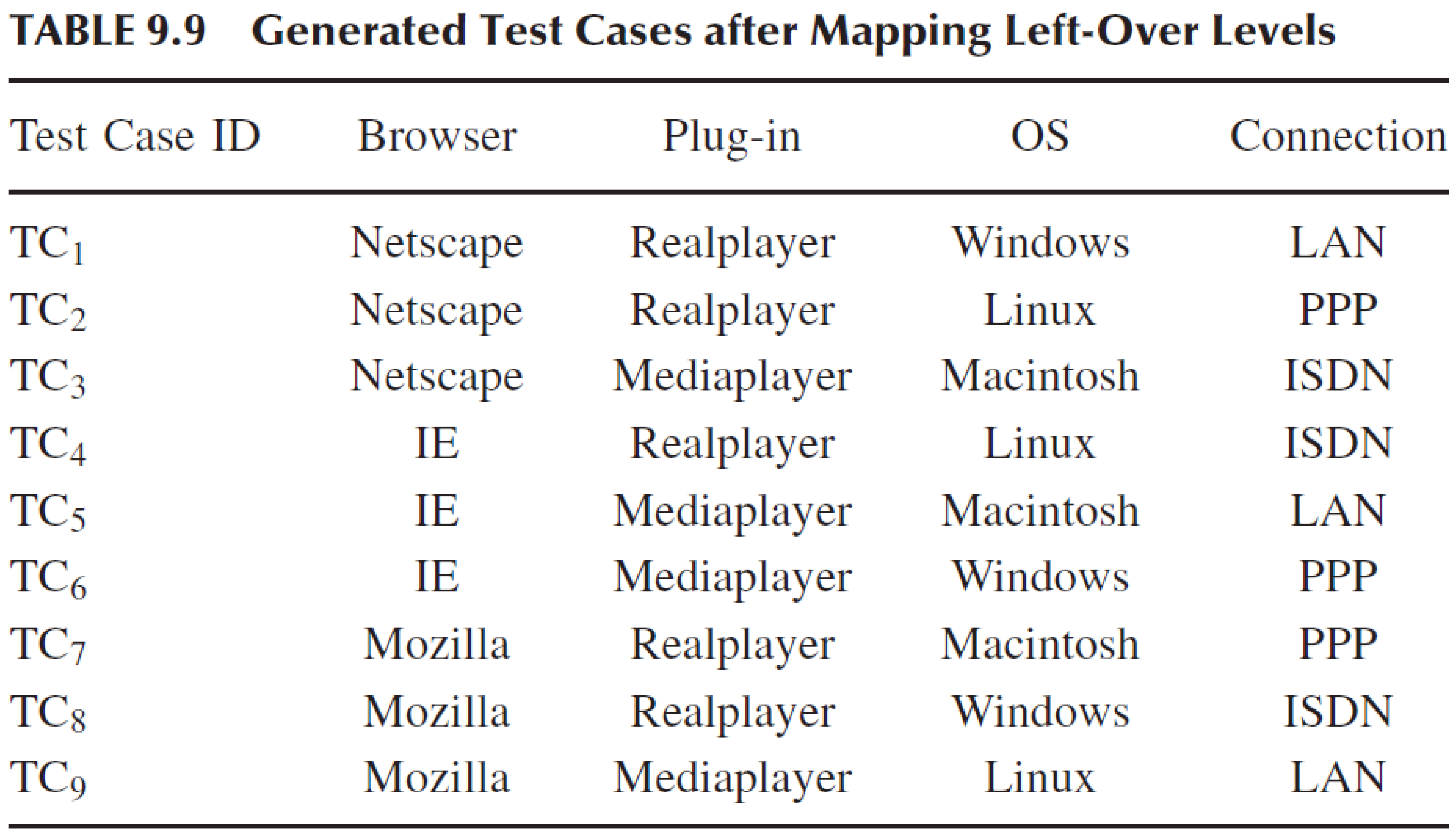


TABLE 9.9 Generated Test Cases after Mapping Left-Over Levels

Test Case ID	Browser	Plug-in	OS	Connection
TC_1	Netscape	Realplayer	Windows	LAN
TC_2	Netscape	Realplayer	Linux	PPP
TC_3	Netscape	Mediaplayer	Macintosh	ISDN
TC_4	IE	Realplayer	Linux	ISDN
TC_5	IE	Mediaplayer	Macintosh	LAN
TC_6	IE	Mediaplayer	Windows	PPP
TC_7	Mozilla	Realplayer	Macintosh	PPP
TC_8	Mozilla	Realplayer	Windows	ISDN
TC_9	Mozilla	Mediaplayer	Linux	LAN

IN PARAMETER ORDER

Tai and Lei have given an algorithm called In Parameter Order (IPO) to generate a test suite for pairwise coverage of input variables

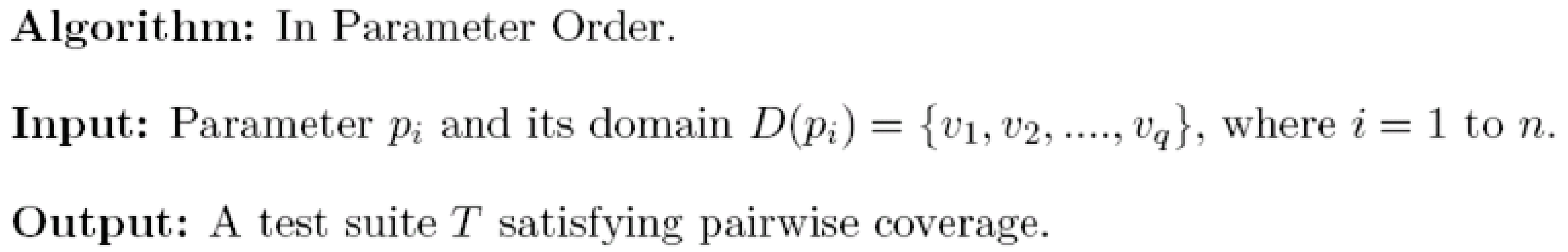
The algorithm runs in three phases, namely, *initialization*, *horizontal growth*, and *vertical growth*, in that order

In the *initialization phase*, test cases are generated to cover two input variables

In the *horizontal growth* phase, the existing test cases are extended with the values of the other input variables.

In the *vertical growth phase*, additional test cases are created such that the test suite satisfies pairwise coverage for the values of the new variables.

IN PARAMETER ORDER

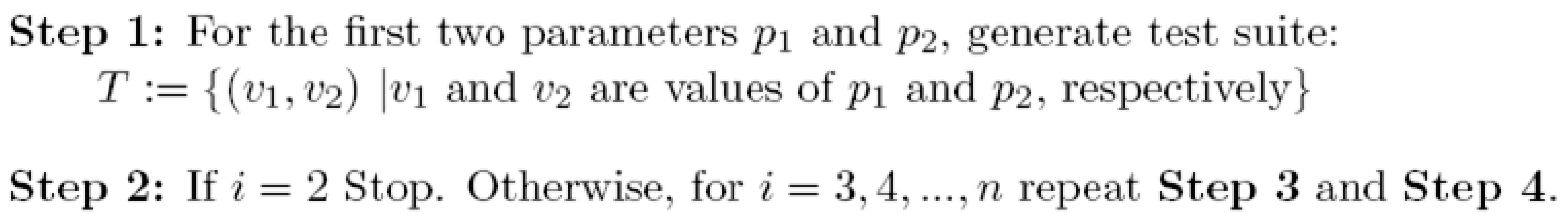


Algorithm: In Parameter Order.

Input: Parameter p_i and its domain $D(p_i) = \{v_1, v_2, ..., v_q\}$, where $i = 1$ to n.

Output: A test suite T satisfying pairwise coverage.

Initialization



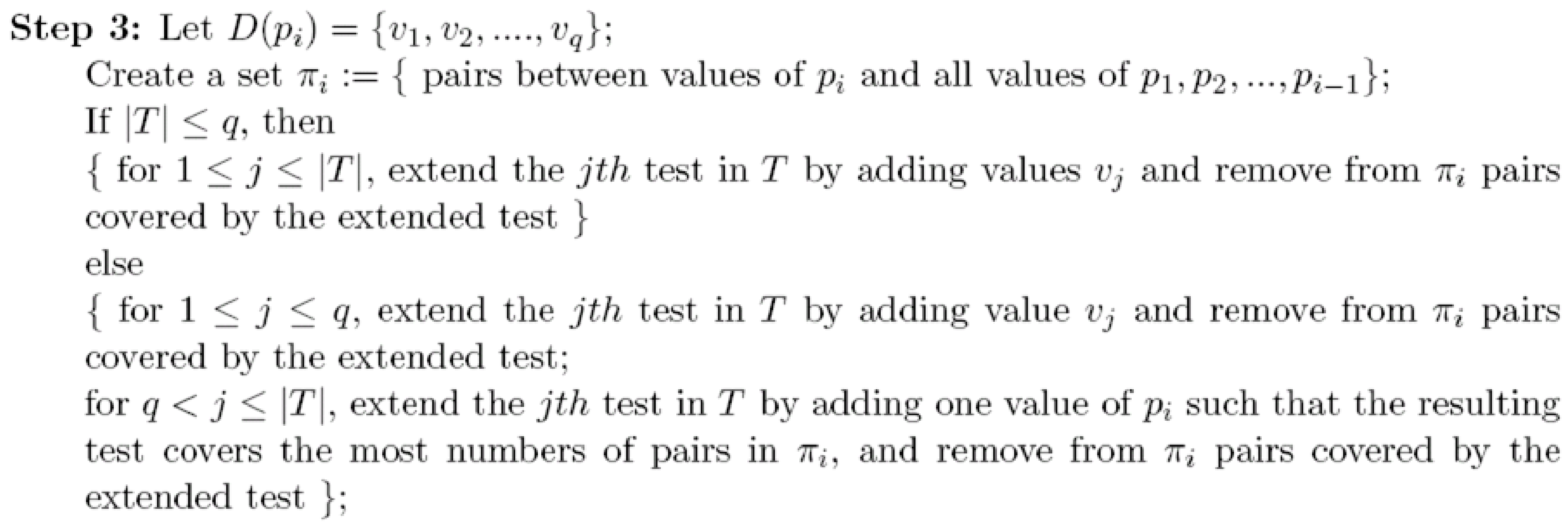
Step 1: For the first two parameters p_1 and p_2, generate test suite:

$$T := \{(v_1, v_2) \; |v_1 \text{ and } v_2 \text{ are values of } p_1 \text{ and } p_2\text{, respectively}\}$$

Step 2: If $i = 2$ Stop. Otherwise, for $i = 3, 4, ..., n$ repeat **Step 3** and **Step 4**.

IN PARAMETER ORDER

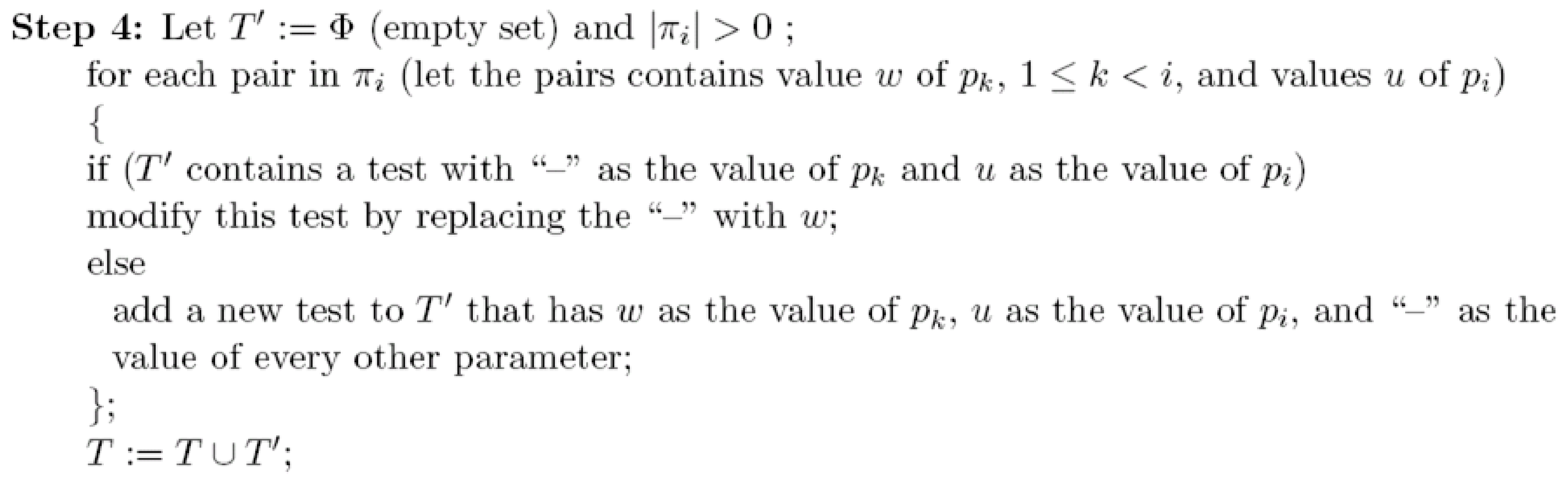
<u>Horizontal Growth Phase</u>



```
Step 3: Let $D(p_i) = \{v_1, v_2, ...., v_q\}$;
    Create a set $\pi_i :=$ { pairs between values of $p_i$ and all values of $p_1, p_2, ..., p_{i-1}$};
    If $|T| \leq q$, then
    { for $1 \leq j \leq |T|$, extend the $jth$ test in $T$ by adding values $v_j$ and remove from $\pi_i$ pairs
    covered by the extended test }
    else
    { for $1 \leq j \leq q$, extend the $jth$ test in $T$ by adding value $v_j$ and remove from $\pi_i$ pairs
    covered by the extended test;
    for $q < j \leq |T|$, extend the $jth$ test in $T$ by adding one value of $p_i$ such that the resulting
    test covers the most numbers of pairs in $\pi_i$, and remove from $\pi_i$ pairs covered by the
    extended test };
```

IN PARAMETER ORDER

Vertical Growth Phase



```
**Step 4:** Let $T' := \Phi$ (empty set) and $|\pi_i| > 0$ ;
    for each pair in $\pi_i$ (let the pairs contains value $w$ of $p_k$, $1 \leq k < i$, and values $u$ of $p_i$)
    {
    if ($T'$ contains a test with "–" as the value of $p_k$ and $u$ as the value of $p_i$)
    modify this test by replacing the "–" with $w$;
    else
      add a new test to $T'$ that has $w$ as the value of $p_k$, $u$ as the value of $p_i$, and "–" as the
      value of every other parameter;
    };
    $T := T \cup T'$;
```

IN PARAMETER ORDER

Example:

- Consider the system S which has three input parameters X, Y, and Z. Assume that a set D, a set of input test data values, has been selected for each input variable such that $D(X)$ = {True, False}, $D(Y)$ = {0, 5}, and $D(Z)$ = {P,Q,R}. The total number of possible test cases is 2 × 2 × 3 = 12, but the IPO algorithm generates six test cases. Let us apply step 1 of the algorithm.

IN PARAMETER ORDER

Initialization:

- **Step 1:** Generate a test suite consisting of four test cases with pairwise coverage for the first two parameters *X* and *Y*:

$$T = \begin{bmatrix} (\text{True}, & 0) \\ (\text{True}, & 5) \\ (\text{False}, & 0) \\ (\text{False}, & 5) \end{bmatrix}$$

- **Step 2:** $i = 3 > 2$; therefore, steps 2 and 3 must be executed.

IN PARAMETER ORDER

Horizontal Growth:

- **Step 1:** Generate a test suite consisting of four test cases with pairwise coverage for the first two parameters *X* and *Y*:

$$T = \begin{bmatrix} (\text{True}, & 0) \\ (\text{True}, & 5) \\ (\text{False}, & 0) \\ (\text{False}, & 5) \end{bmatrix}$$

- **Step 2:** $i = 3 > 2$; therefore, steps 2 and 3 must be executed.

EQUIVALENCE CLASS PARTITIONING

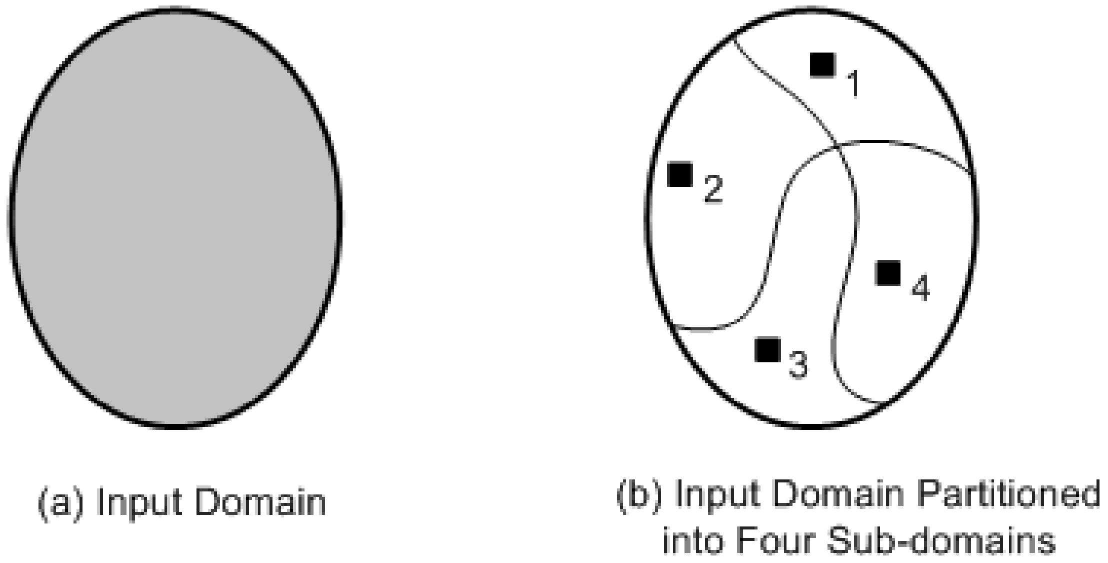
An input domain may be too large for all its elements to be used as test input Figure (a)

The input domain is partitioned into a finite number of subdomains

Each subdomain is known as an equivalence class, and it serves as a source of at least one test input Figure (b).

A valid input to a system is an element of the input domain that is expected to return a non error value

An invalid input is an input that is expected to return an error value.



(a) Too many test input;

(b) One input is selected from each of the subdomain

GUIDELINES FOR EQUIVALENCE CLASS PARTITIONING

An input condition specifies a range [a, b]

- one equivalence class for $a < X < b$, and
- two other classes for $X < a$ and $X > b$ to test the system with invalid inputs

An input condition specifies a set of values

- one equivalence class for each element of the set $\{M_1\}$, $\{M_2\}$, ..., $\{M_N\}$, and
- one equivalence class for elements outside the set $\{M_1, M_2, ..., M_N\}$

Input condition specifies for each individual value

- If the system handles each valid input differently then create one equivalence class for each valid input

An input condition specifies the number of valid values (Say N)

- Create one equivalence class for the correct number of inputs
- two equivalence classes for invalid inputs – one for zero values and one for more than N values

An input condition specifies a "must be" value

- Create one equivalence class for a "must be" value, and
- one equivalence class for something that is not a "must be" value

IDENTIFICATION OF TEST CASES

Test cases for each equivalence class can be identified by:

- Assign a unique number to each equivalence class
- For each equivalence class with valid input that has not been covered by test cases yet, write a new test case covering as many uncovered equivalence classes as possible
- For each equivalence class with invalid input that has not been covered by test cases, write a new test case that covers one and only one of the uncovered equivalence classes

ADVANTAGES OF EQUIVALENCE CLASS PARTITIONING

- A small number of test cases are needed to adequately cover a large input domain
- One gets a better idea about the input domain being covered with the selected test cases
- The probability of uncovering defects with the selected test cases based on equivalence class partitioning is higher than that with a randomly chosen test suite of the same size
- The equivalence class partitioning approach is not restricted to input conditions alone – the technique may also be used for output domains

BOUNDARY VALUE ANALYSIS (BVA)

- The central idea in Boundary Value Analysis (BVA) is to select test data near the boundary of a data domain so that data both within and outside an equivalence class are selected
- The BVA technique is an extension and refinement of the equivalence class partitioning technique
- In the BVA technique, the boundary conditions for each of the equivalence class are analyzed in order generate test cases

GUIDELINES FOR BOUNDARY VALUE ANALYSIS

The equivalence class specifies a range

- If an equivalence class specifies a range of values, then construct test cases by considering the boundary points of the range and points just beyond the boundaries of the range

The equivalence class specifies a number of values

- If an equivalence class specifies a number of values, then construct test cases for the minimum and the maximum value of the number
- In addition, select a value smaller than the minimum and a value larger than the maximum value.

The equivalence class specifies an ordered set

- If the equivalence class specifies an ordered set, such as a linear list, table, or a sequential file, then focus attention on the first and last elements of the set.

DECISION TABLES

The structure of a decision table has been shown in Table 9.13

It comprises a set of conditions (or, causes) and a set of effects (or, results) arranged in the form of a column on the left of the table

In the second column, next to each condition, we have its possible values: Yes (Y), No (N), and Don’t Care (“-”)

To the right of the “Values” column, we have a set of rules. For each combination of the three conditions {C1,C2,C3}, there exists a rule from the set {R1,R2, ..,R8}

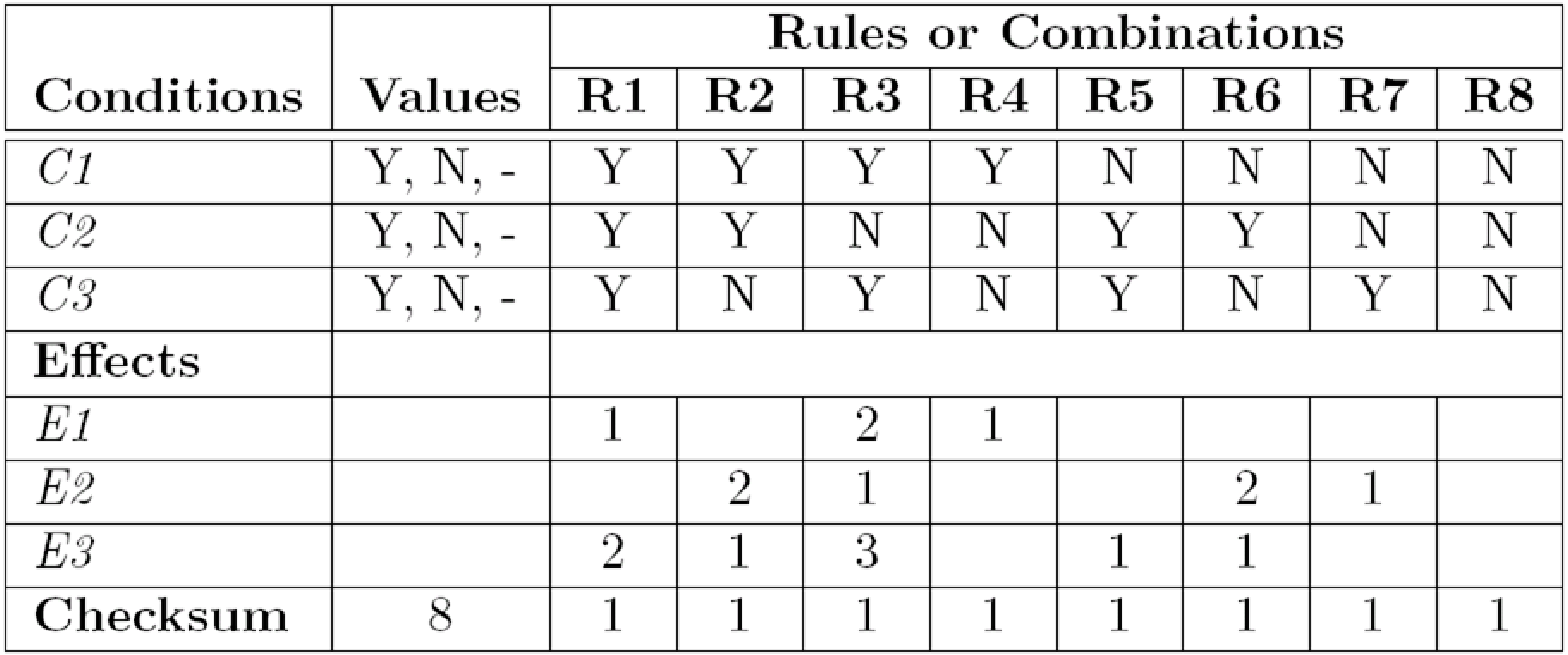
Each rule comprises a Yes (Y), No (N), or Don’t Care (“-”) response, and contains an associated list of effects {E1,E2,E3}

For each relevant effect, an effect sequence number specifies the order in which the effect should be carried out, if the associated set of conditions are satisfied

The “Checksum” is used for verification of the combinations, the decision table represent

Each rule of a decision table represents a test case

DECISION TABLES



Conditions	Values	Rules or Combinations							
		R1	R2	R3	R4	R5	R6	R7	R8
C1	Y, N, -	Y	Y	Y	Y	N	N	N	N
C2	Y, N, -	Y	Y	N	N	Y	Y	N	N
C3	Y, N, -	Y	N	Y	N	Y	N	Y	N
Effects									
E1		1		2	1				
E2			2	1			2	1	
E3		2	1	3		1	1		
Checksum	8	1	1	1	1	1	1	1	1

Table 9.13: A decision table comprising a set of conditions and effects.

DECISION TABLES

The steps in developing test cases using decision table technique:

Step 1: The test designer needs to identify the conditions and the effects for each specification unit.

- A condition is a distinct input condition or an equivalence class of input conditions
- An effect is an output condition. Determine the logical relationship between the conditions and the effects

Step 2: List all the conditions and effects in the form of a decision table. Write down the values the condition can take

Step 3: Calculate the number of possible combinations. It is equal to the number of different values raised to the power of the number of conditions

DECISION TABLES

Step 4: Fill the columns with all possible combinations – each column corresponds to one combination of values. For each row (condition) do the following:

- Determine the Repeating Factor (RF): divide the remaining number of combinations by the number of possible values for that condition
- Write RF times the first value, then RF times the next and so forth, until row is full

Step 5: Reduce combinations (rules). Find indifferent combinations - place a "-" and join column where columns are identical. While doing this, ensure that effects are the same

Step 6: Check covered combinations (rules). For each column calculate the combinations it represents. A "-" represents as many combinations as the condition has. Multiply for each "-" down the column. Add up total and compare with step 3. It should be the same

Step 7: Add effects to the column of the decision table. Read column by column and determine the effects. If more than one effect can occur in a single combinations, then assign a sequence number to the effects, thereby specifying the order in which the effects should be performed. Check the consistency of the decision table

Step 8: The columns in the decision table are transformed into test cases

THANK YOU!!