NEC
Empowered by Innovation

University of Waterloo

Software Testing and Quality Assurance

Theory and Practice

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 of Howden

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
- Howden 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**

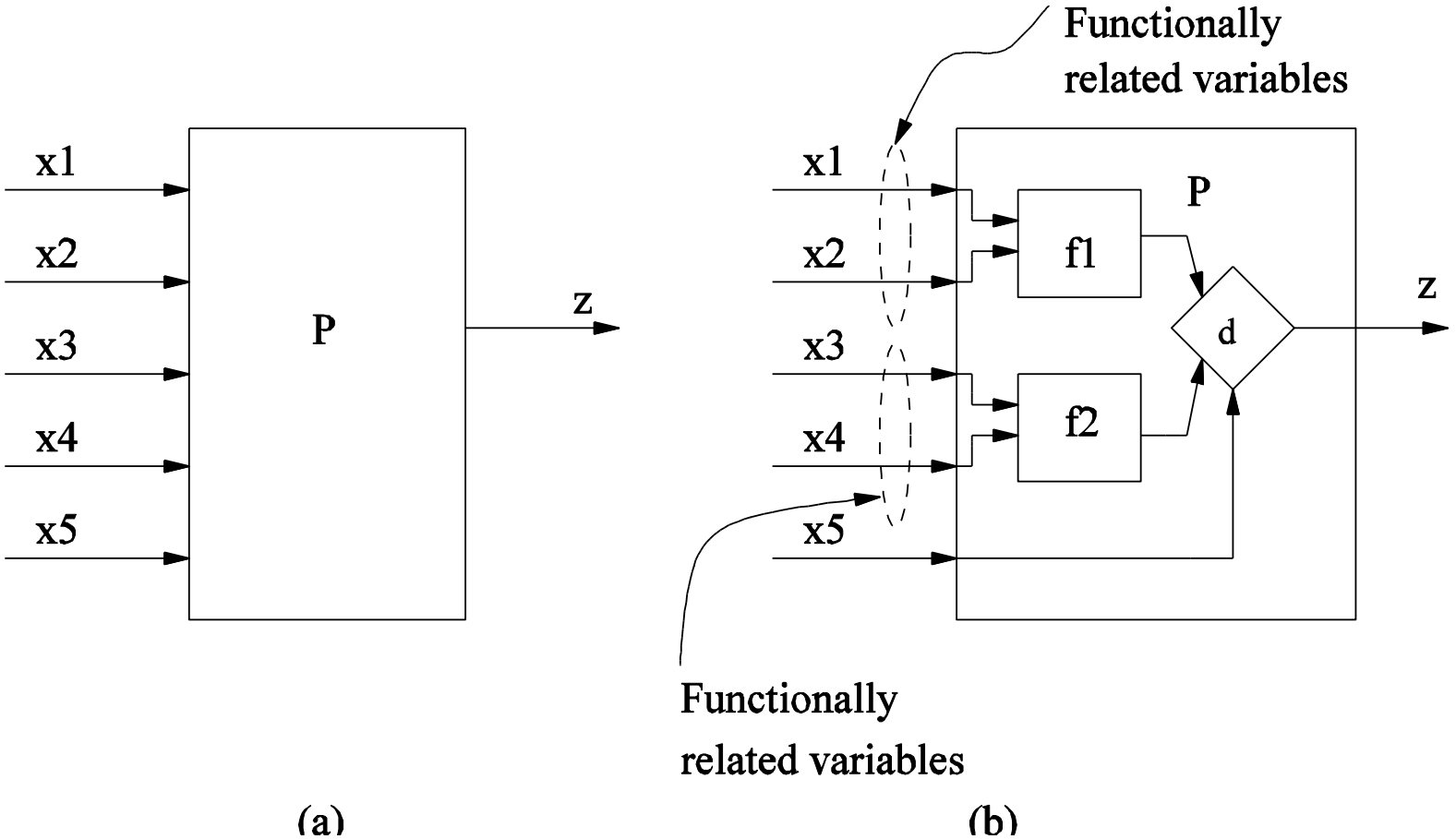


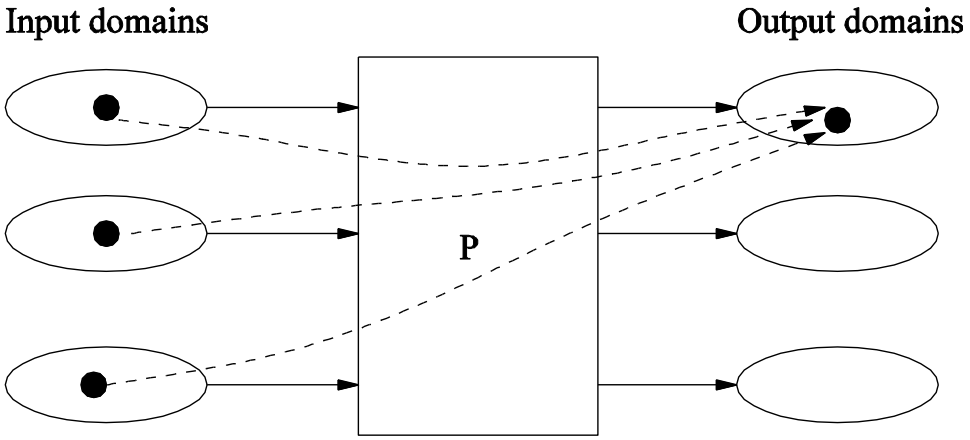
Figure 9.3: Functionality related variables

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).

Howden's Functional Testing Summary



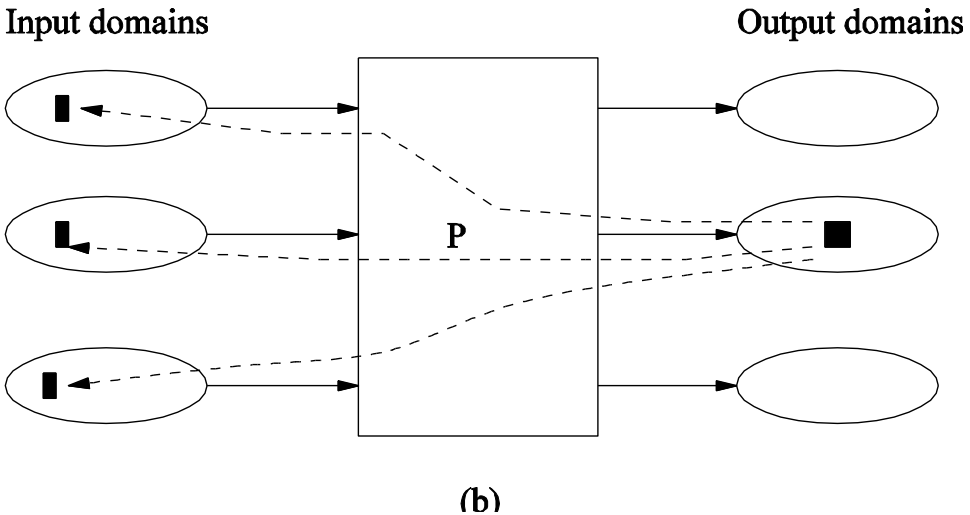


Figure 9.5: Obtaining output values from an input vector (a), and obtaining an input vector from an output value (b) in functional testing

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 Figure 9.7, 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}

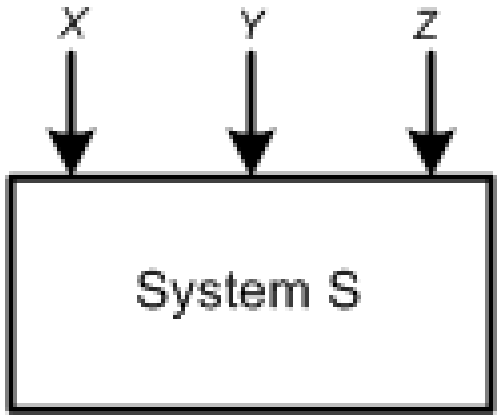


Figure 9.7: System S with three input variables.

Pairwise Testing

- The total number of all-combination test cases is $2 \times 2 \times 2 = 8$
- 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

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 pairwiae 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:
$T := \{(v_1, v_2) \mid v_1$ and v_2 are values of p_1 and p_2, respectively$\}$

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

In Parameter Order

Horizontal Growth Phase

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 π_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 π_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 π_i, and remove from π_i pairs covered by the extended test };

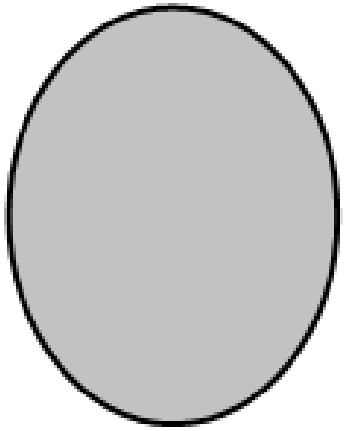
In Parameter Order

Vertical Growth Phase

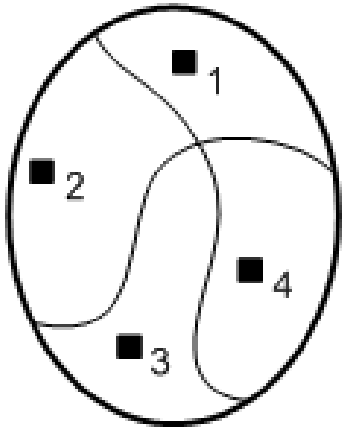
Step 4: Let $T' := \Phi$ (empty set) and $|\pi_i| > 0$;
for each pair in π_i (let the pairs contains value w of p_k, $1 \leq k < i$, and values u of p_i)
{
if (T' contains a test with "–" as the value of p_k and u as the value of p_i)
modify this test by replacing the "–" with w;
else
add a new test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

Equivalence Class Partitioning

- An input domain may be too large for all its elements to be used as test input (Figure 9.8(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 9.8(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) Input Domain



(b) Input Domain Partitioned into Four Sub-domains

Figure 9.8: (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

Random Testing

- In the random testing approach, test inputs are selected randomly from the input domain of the system
- Random testing can be summarized as:
  - Step 1 : The input domain is identified
  - Step 2 : Test inputs are selected independently from the domain
  - Step 3 : The system under test is executed on these inputs
    - The inputs constitute a random test set
  - Step 4 : The results are compared to the system specification.
    - The test is a failure if any input leads to incorrect results
    - Otherwise it is a success.
- Random testing gives us an advantage of easily estimating software reliability from test outcomes
- Test inputs are randomly generated according to an operational profile, and failure times are recorded
- The data obtained from random testing can then be used to estimate reliability

Random Testing

- Computing expected outcomes becomes difficult, if the inputs are randomly chosen
- Therefore, the technique requires good test oracles to ensure the adequate evaluation of test results
- A *test oracle* is a mechanism that verifies the correctness of program outputs
- An *oracle* provides a method to
  - generate expected results for the test inputs, and
  - compare the expected results with the actual results of execution of the Implementation Under Test (IUT)
- Four common types of oracles are as follows:
  - Perfect oracle
  - Gold standard oracle
  - Parametric oracle
  - Statistical oracle

Adaptive Random Testing

- In adaptive random testing the test inputs are selected from the randomly generated set in such a way that these are evenly spread over the entire input domain
- The goal is to select a small number of test inputs to detect the first failure
- A number of random test inputs are generated, then the "best" one among them is selected
- We need to make sure the selected new test input should not be too close to any of the previously selected ones
- That is, try to distribute the selected test inputs as spaced out as possible

Error Guessing

- It is a test case design technique where a test engineer uses his experience to
  - guess the types and probable locations of defects, and
  - design tests specifically to reveal the defects
- Though experience is of much use in guessing errors, it is useful to add some structure to the technique
- It is good to prepare a list of types of errors that can be uncovered
- The error list can aid us in guessing where errors may occur. Such a list should be maintained from experience gained from earlier test projects

Category Partition

- The Category Partition Method (CPM) is a systematic, specification based methodology that uses an informal functional specification to produce formal test specification

- The test designer's key job is to develop *categories*, which are defined to be the major characteristics of the input domain of the function under test

- Each *category* is partitioned into equivalence classes of inputs called *choices*
- The *choices* in each category must be disjoint, and together the choices in each *category* must cover the input domain

Category Partition

The method comprise of the following steps:

- **Step 1.** Analyze the Specification
- **Step 2.** Identify Categories
- **Step 3.** Partition the Categories into Choices
- **Step 4.** Determine Constraints among Choices
- **Step 5.** Formalize and Evaluate the Test Specification
- **Step 6.** Generate and Validate the Test Cases