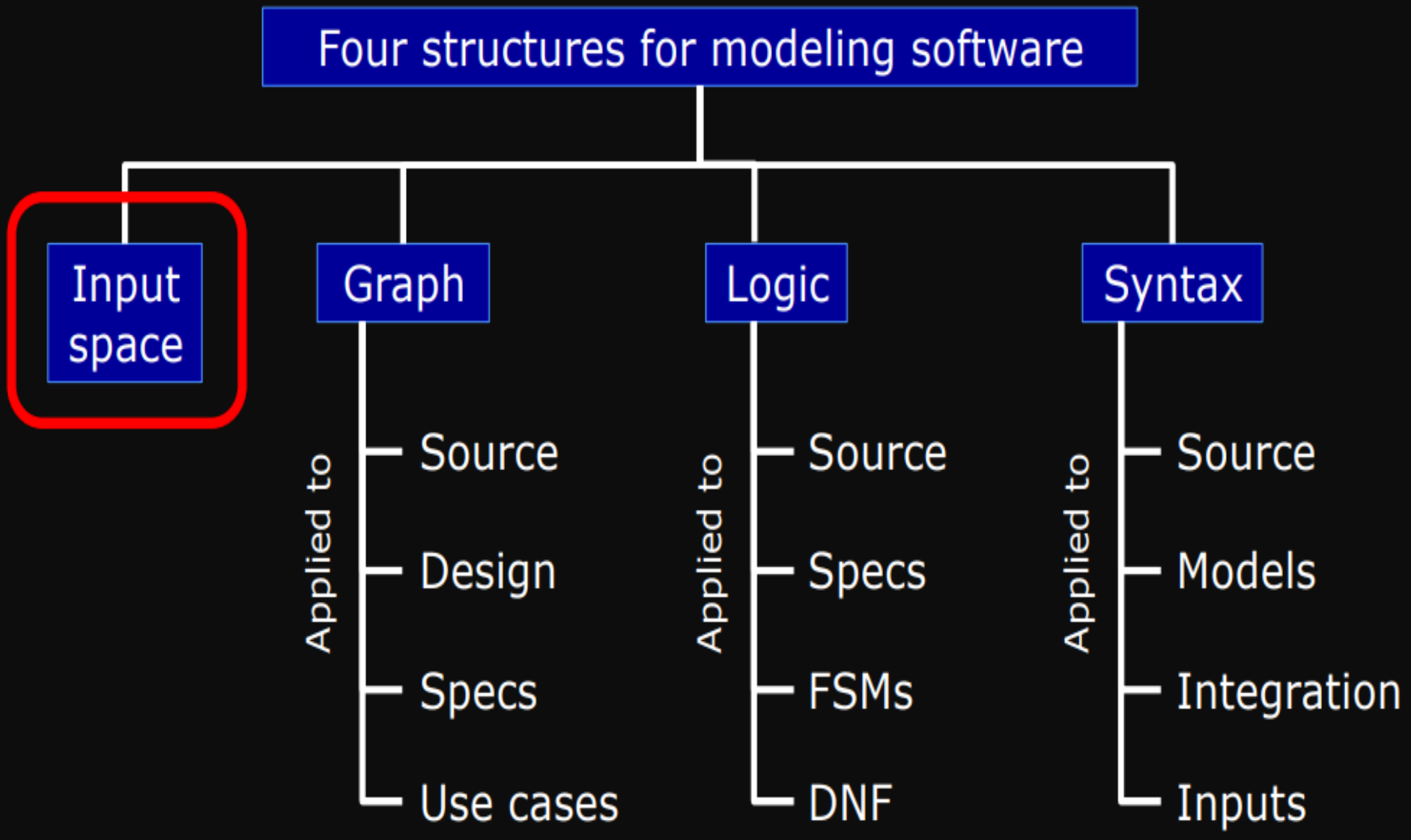
Introduction to Software Testing
Module 4

Input Space Partition Testing

Structures for Criteria-Based Testing



Input Domains

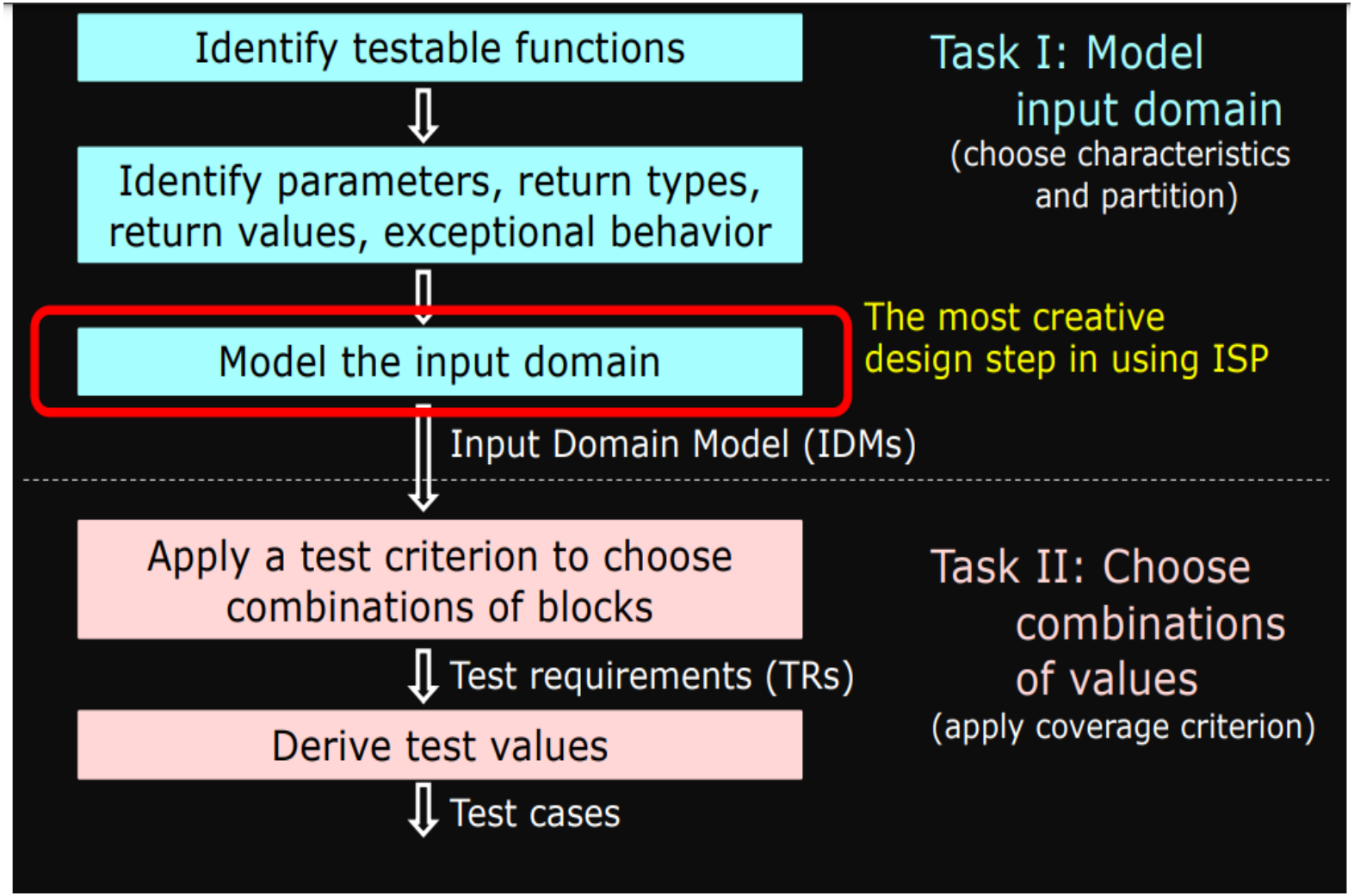
- The input domain for a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about choosing finite sets of values from the input domain
- *Input parameters* define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Benefits of ISP

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System
- Relatively easy to apply with no automation
- Easy to adjust the procedure to get more or fewer tests
- No implementation knowledge is needed
  - just the input space

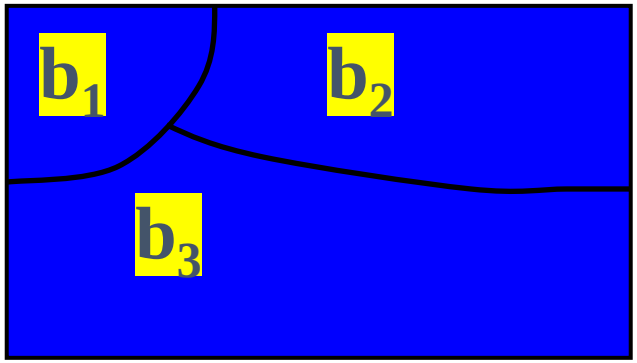
ISP

- Input space partitioning describes the input domain of the software
- Domain (D) are partitioned into blocks (b1, b2, .., bn)



Partitioning Domains

- *Domain D*
- *Partition scheme q* of *D*
- The partition *q* defines a *set of blocks*, $Bq = b_1, b_2, \ldots b_Q$
- The partition must satisfy two properties :
  1. blocks must be *pairwise disjoint* (no overlap)
  2. together the blocks *cover* the domain *D* (complete)



$$b_i \cap b_j = \Phi, \quad \forall\ i \neq j,\ b_i, b_j \in B_q$$

$$\bigcup_{b \in Bq} b = D$$

Using Partitions – Assumptions

- Choose a value from each block
- Each value is assumed to be equally useful for testing
- Application to testing
  - Find characteristics in the inputs : parameters, semantic descriptions, …
  - Partition each characteristic
  - Choose tests by combining values from characteristics
- Example Characteristics
  - Input X is null
  - Order of the input file F (sorted, inverse sorted, arbitrary, …)
  - Min separation of two aircraft
  - Input device (DVD, CD, VCR, computer, …)

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the "*order of file F*"

Solution:

Each characteristic should address just one property

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

File F sorted ascending

- b1 = true
- b2 = false

File F sorted descending

- b1 = true
- b2 = false

but … something's fishy …

What if the file is of length 1?

The file will be in all three blocks …

That is, disjointness is not satisfied

Properties of Partitions

- If the partitions are not complete or disjoint, that means the partitions have not been considered carefully enough
- They should be reviewed carefully, like any design attempt
- Different alternatives should be considered
- We model the input domain in five steps …

Modeling the Input Domain

- **Step 1 : Identify testable functions**
  - Individual methods have one testable function
  - In a class, each method often has the same characteristics
  - Programs have more complicated characteristics—modeling documents such as UML use cases can be used to design characteristics
  - Systems of integrated hardware and software components can use devices, operating systems, hardware platforms, browsers, etc

- **Step 2 : Find all the parameters**
  - Often fairly straightforward, even mechanical
  - Important to be complete
  - Methods : Parameters and state (non-local) variables used
  - Components : Parameters to methods and state variables
  - System : All inputs, including files and databases

Modeling the Input Domain (*cont*)

- Step 3 : **Mo**del the input domain
  - The domain is scoped by the parameters
  - The structure is defined in terms of characteristics
  - Each characteristic is partitioned into sets of blocks
  - Each block represents a set of values
  - This is the most creative design step in using ISP
- Step 4 : Apply a test criterion to choose combinations of values
  - A test input has a value for each parameter
  - One block for each characteristic
  - Choosing all combinations is usually infeasible
  - Coverage criteria allow subsets to be chosen
- Step 5 : Refine combinations of blocks into test inputs
  - Choose appropriate values from each block

Two Approaches to Input Domain Modeling

1. Interface-based approach
   - Develops characteristics directly from individual input parameters
   - Simplest application
   - Can be partially automated in some situations

2. Functionality-based approach
   - Develops characteristics from a behavioral view of the program under test
   - Harder to develop—requires more design effort
   - May result in better tests, or fewer tests that are as effective

Input Domain Model (IDM)

1. Interface-Based Approach

- Mechanically consider each parameter in isolation
- This is an easy modeling technique and relies mostly on syntax
- Some domain and semantic information won't be used
  - Could lead to an incomplete IDM
- Ignores relationships among parameters

2. Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Interface-based

- Develop characteristics directly from parameters
  - Translate parameters to characteristics
- Consider each parameter separately
- Rely mostly on syntax
- Ignore some domain and semantic information
  - Can lead to an incomplete IDM
- Ignore relationships among parameters

Functionality-based

- Develop characteristics that correspond to the intended functionality
- Can use relationships among parameters, relationships of parameters with special values (null, blank, …), preconditions, and postconditions
- Incorporate domain and semantic knowledge
  - May lead to better tests
- The same parameter may appear in multiple characteristics

Steps 1 & 2 – Identifying Functionalities, Parameters and Characteristics

- A creative engineering step
- More characteristics means more tests
- Interface-based : Translate parameters to characteristics
- Candidates for characteristics :
  - Preconditions and postconditions
  - Relationships among variables
  - Relationship of variables with special values (zero, null, blank, …)
- Should not use program source – characteristics should be based on the input domain
  - Program source should be used with graph or *logic* criteria
- Better to have more characteristics with few blocks
  - Fewer mistakes and fewer tests

Steps 1 & 2 : Interface vs Functionality-Based

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Interface-Based Approach

Two parameters : list, element

Characteristics :

- **list is null (block1 = true, block2 = false)**
- **list is empty (block1 = true, block2 = false)**

Functionality-Based Approach

Two parameters : list, element

Characteristics :

- **number of occurrences of element in list**
  - **(0, 1, >1)**
- **element occurs first in list**
  - **(true, false)**
- **element occurs last in list**
  - **(true, false)**

Step 3 : Modeling the Input Domain

- Partitioning characteristics into blocks and values is a very creative engineering step
- More blocks means more tests
- The partitioning often flows directly from the definition of characteristics and both steps are sometimes done together
  - Should evaluate them separately – sometimes fewer characteristics can be used with more blocks and vice versa
- Strategies for identifying values :
  - Include valid, invalid and special values
  - Sub-partition some blocks
  - Explore boundaries of domains
  - Include values that represent “normal use”
  - Try to balance the number of blocks in each characteristic
  - Check for completeness and disjointness

Interface-Based IDM – TriTyp

- **TriTyp, from Chapter 3, had one testable function and three integer inputs**

First Characterization of TriTyp's Inputs			
Characteristic	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of 3*3*3 = 27 tests
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

Interface-Based IDM – TriTyp (*cont*)

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Refinement of q_1"	greater than 1	equal to 1	equal to 0	less than 0
q_2 = "Refinement of q_2"	greater than 1	equal to 1	equal to 0	less than 0
q_3 = "Refinement of q_3"	greater than 1	equal to 1	equal to 0	less than 0

- A maximum of 4*4*4 = 64 tests
- This is only complete because the inputs are integers (0 . . 1)

Possible values for partition q_1

Characteristic	b_1	b_2	b_3	b_4
Side1	2	1	0	-1

Test boundary conditions

Functionality-Based IDM – TriTyp

- First two characterizations are based on syntax–parameters and their type
- A semantic level characterization could use the fact that the three integers represent a triangle

Geometric Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- **Oops … something's fishy … equilateral is also isosceles !**
- **We need to refine the example to make characteristics valid**

Correct Geometric Characterization of TriTyp's Inputs

Characteristic	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

Functionality-Based IDM – TriTyp (*cont*)

- Values for this partitioning can be chosen as

Possible values for geometric partition q_1				
Characteristic	b_1	b_2	b_3	b_4
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Functionality-Based IDM – TriTyp (*cont*)

- A different approach would be to break the geometric characterization into four separate characteristics

Four Characteristics for TriTyp

Characteristic	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

- **Use constraints to ensure that**
  - **Equilateral = True implies Isosceles = True**
  - **Valid = False implies Scalene = Isosceles = Equilateral = False**

Using More than One IDM

- Some programs may have dozens or even hundreds of parameters
- Create several small IDMs
  - A divide-and-conquer approach
- Different parts of the software can be tested with different amounts of rigor
  - For example, some IDMs may include a lot of invalid values
- It is okay if the different IDMs overlap
  - The same variable may appear in more than one IDM

Step 4 – Choosing Combinations of Values

- Once characteristics and partitions are defined, the next step is to choose test values
- We use criteria – to choose effective subsets
- The most obvious criterion is to choose all combinations …

All Combinations (ACoC) : All combinations of blocks from all characteristics must be used.

- **Number of tests is the product of the number of blocks in each characteristic :** $\prod_{i=1}^{Q}(B_i)$
- **The second characterization of TriTyp results in 4*4*4 = 64 tests – too many ?**

ISP Criteria – Each Choice

- 64 tests for TriTyp is almost certainly way too many
- One criterion comes from the idea that we should try at least one value from each block

Each Choice (EC) : One value from each block for each characteristic must be used in at least one test case.

- **Number of tests is the number of blocks in the largest characteristic**

$$\mathbf{Max}_{i=1}^{Q}(B_i)$$

For TriTyp: 2, 2, 2
1, 1, 1
0, 0, 0
-1, -1, -1

ISP Criteria – Pair-Wise

- Each choice yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

Pair-Wise (PW) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

- **Number of tests is at least the product of two largest characteristics**

$$(\text{Max}_{i=1}^{Q}(B_i)) * (\text{Max}_{j=1,\, j!=i}^{Q}(B_j))$$

For TriTyp: 2, 2, 2	2, 1, 1	2, 0, 0	2, -1, -1
1, 2, 1	1, 1, 0	1, 0, -1	1, -1, 2
0, 2, 0	0, 1, -1	0, 0, 2	0, -1, 1
-1, 2, -1	-1, 1, 2	-1, 0, 1	-1, -1, 0

ISP Criteria –T-Wise

- A natural extension is to require combinations of *t* values instead of *2*

t-Wise (TW) : A value from each block for each group of t characteristics must be combined.

- **Number of tests is at least the product of *t* largest characteristics**
- **If all characteristics are the same size, the formula is**

$$(\text{Max}\ {}^{Q}_{i=1}(B_i))^t$$

- **If *t* is the number of characteristics *Q*, then all combinations**
- **That is … *Q-wise = AC***
- ***t*-wise is expensive and benefits are not clear**

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

Base Choice (BC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

- **Number of tests is one base test + one test for each other block**

$$1 + \sum_{i=1}^{Q} (B_i - 1)$$

For TriTyp: Base	**2, 2, 2**	**2, 2, 1**	**2, 1, 2**	**1, 2, 2**
		2, 2, 0	**2, 0, 2**	**0, 2, 2**
		2, 2, -1	**2, -1, 2**	**-1, 2, 2**

Base Choice Notes

- The base test must be feasible
  - That is, all base choices must be compatible
- Base choices can be
  - Most likely from an end-use point of view
  - Simplest
  - Smallest
  - First in some ordering
- The base choice is a crucial design decision
  - Test designers should document why the choices were made

ISP Criteria – Multiple Base Choice

- Testers sometimes have more than one logical base choice

Multiple Base Choice (MBC) : One or more base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choices in each other characteristic.

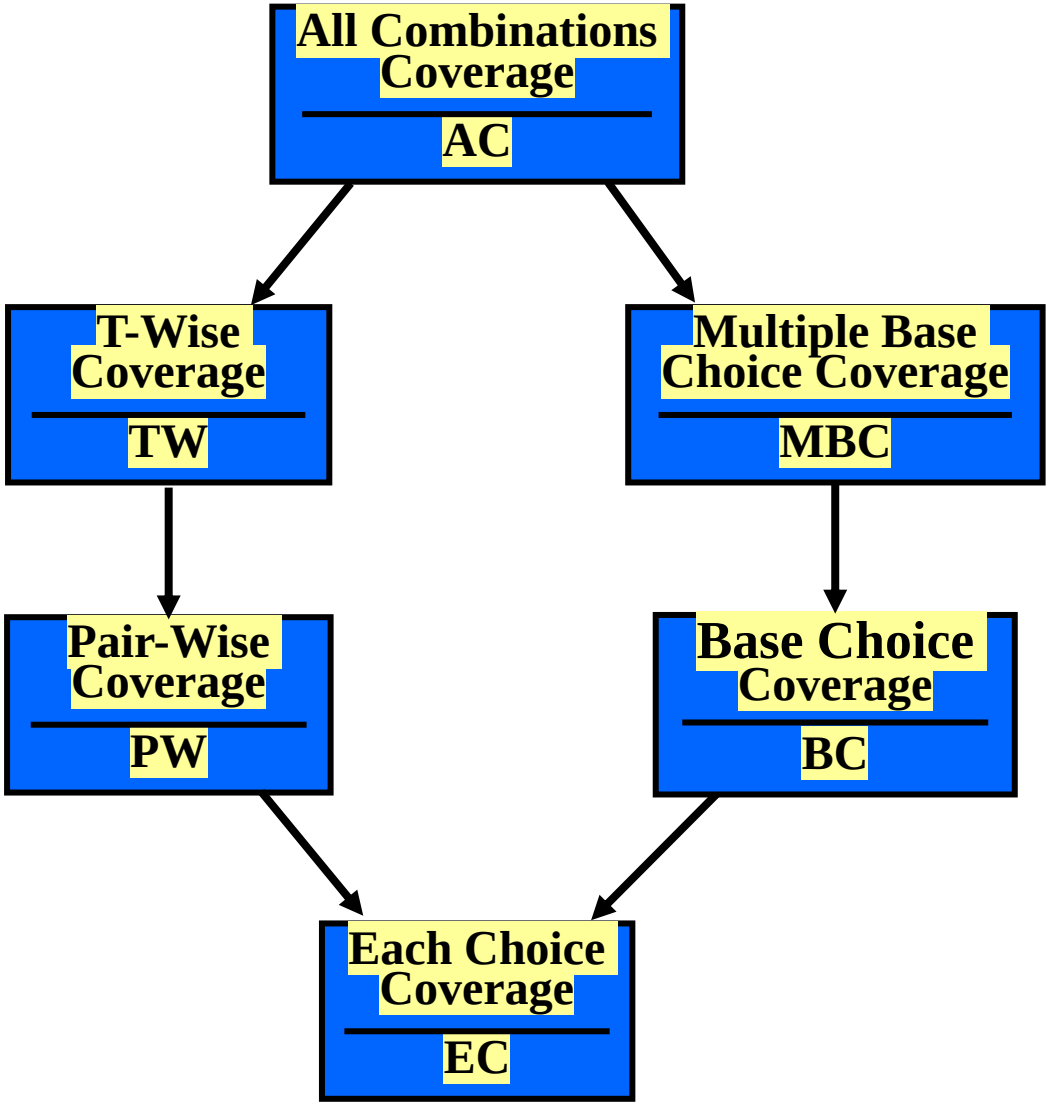
- **If there are M base tests and m_i base choices for each characteristic:**

$$M + \sum_{i=1}^{Q} (M * (B_i - m_i))$$

For TriTyp: Base

2, 2, 2	2, 2, 0	2, 0, 2	0, 2, 2
	2, 2, -1	2, -1, 2	-1, 2, 2
1, 1, 1	1, 1, 0	1, 0, 1	0, 1, 1
	1, 1, -1	1, -1, 1	-1, 1, 1

ISP Coverage Criteria Subsumption

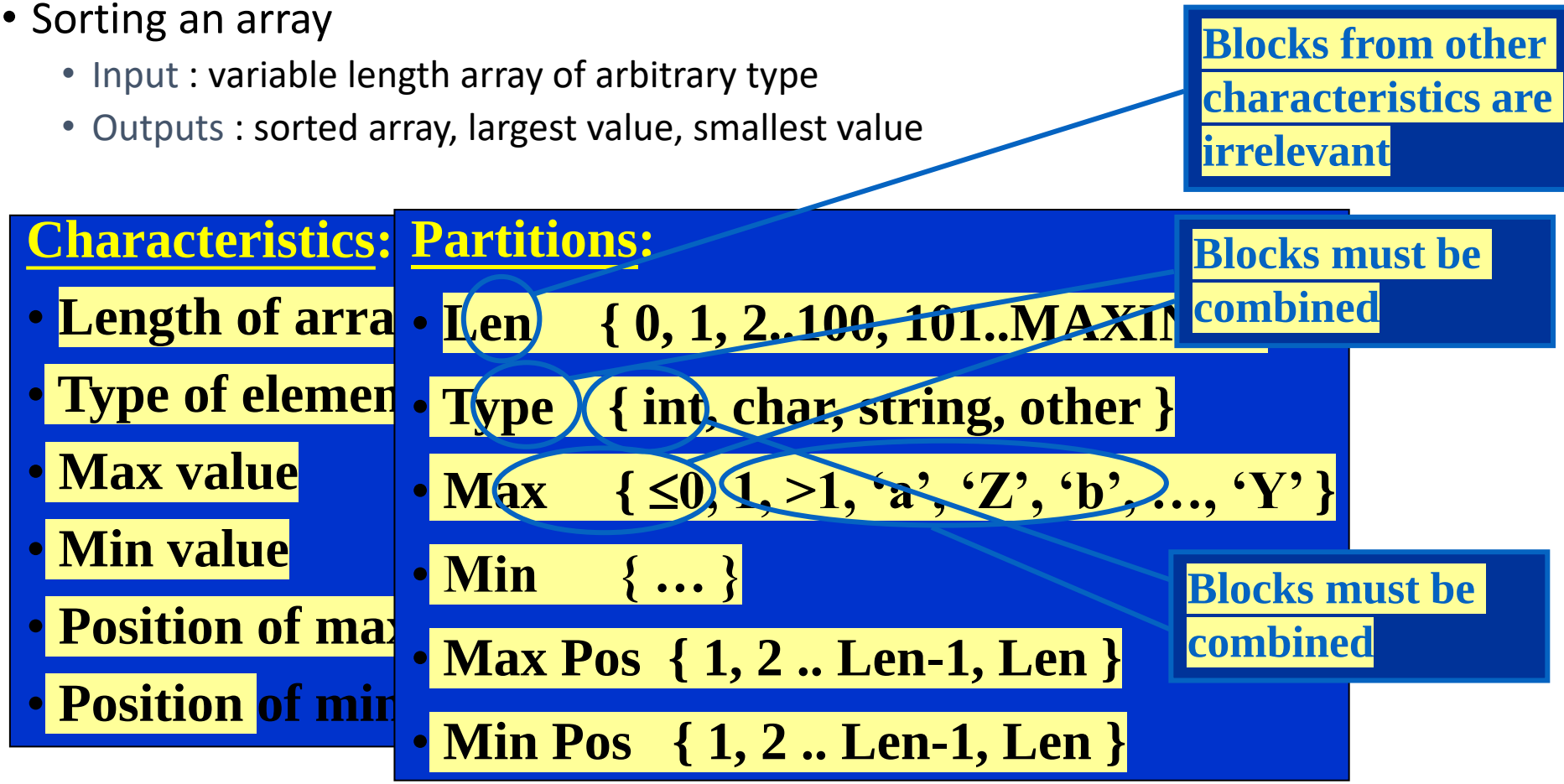


Constraints Among Characteristics

- Some combinations of blocks are infeasible
  - "less than zero" and "scalene" … not possible at the same time
- These are represented as constraints among blocks
- Two general types of constraints
  - A block from one characteristic cannot be combined with a specific block from another
  - A block from one characteristic can ONLY BE combined with a specific block form another characteristic
- Handling constraints depends on the criterion used
  - AC, PW, TW : Drop the infeasible pairs
  - BC, MBC : Change a value to another non-base choice to find a feasible combination

Example Handling Constraints

- Sorting an array
  - Input : variable length array of arbitrary type
  - Outputs : sorted array, largest value, smallest value



Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used in practice

Problem

- Consider a simple program to classify a triangle. Its input consists of three positive integers (say x, y, z) and the data types for input parameters ensures that these will be integers greater than zero and less than or equal to 100. The three values are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled

- Solution: Following possible boundary conditions are formed:
- 1. Given sides (A; B; C) for a scalene triangle, the sum of any two sides is greater than the third and so, we have boundary conditions $A + B > C$, $B + C > A$ and $A + C > B$.
- 2. Given sides (A; B; C) for an isosceles triangle two sides must be equal and so we have boundary conditions $A = B$, $B = C$ or $A = C$.
- 3. Continuing in the same way for an equilateral triangle the sides must all be of equal length and we have only one boundary where $A = B = C$.
- 4. For right-angled triangles, we must have $A^2 + B^2 = C^2$

- On the basis of the above boundary conditions, test cases are designed as follows

Test case	x	y	z	Expected Output
1	100	100	100	Equilatera/ triangle
2	50	3	50	Isosceles triangle
3	40	50	40	Equilatera/ triangle
4	3	4	5	Right-angled triangles
5	10	10	10	Equilatera/ triangle
6	2	2	5	Isosceles triangle
7	100	50	100	Scalene triangle
8	1	2	3	Non-triangles
9	2	3	4	Scalene triangle
10	1	3	1	Isosceles triangle

Functional Testing

- FUNCTIONAL TESTING is a type of software testing that validates the software system against the functional requirements/specifications.
- The purpose of Functional tests is to test each function of the software application, by providing appropriate input, verifying the output against the Functional requirements.
- Functional testing mainly involves black box testing and it is not concerned about the source code of the application.
- This testing checks User Interface, APIs, Database, Security, Client/Server communication and other functionality of the Application Under Test.
- The testing can be done either manually or using automation.

What do you test in Functional Testing?

- The prime objective of Functional testing is checking the functionalities of the software system. It mainly concentrates on –
- **Mainline functions**: Testing the main functions of an application
- **Basic Usability**: It involves basic usability testing of the system. It checks whether a user can freely navigate through the screens without any difficulties.
- **Accessibility**: Checks the accessibility of the system for the user
- **Error Conditions**: Usage of testing techniques to check for error conditions. It checks whether suitable error messages are displayed.

How to do Functional Testing

- Following is a step by step process on **How to do Functional Testing** :

➢ Understand the Functional Requirements

➢ Identify test input or test data based on requirements

➢ Compute the expected outcomes with selected test input values

➢ Execute test cases

➢ Compare actual and computed expected results

Identify test input (test data)

Compute the expected outcomes with the selected test input values

Execute test cases

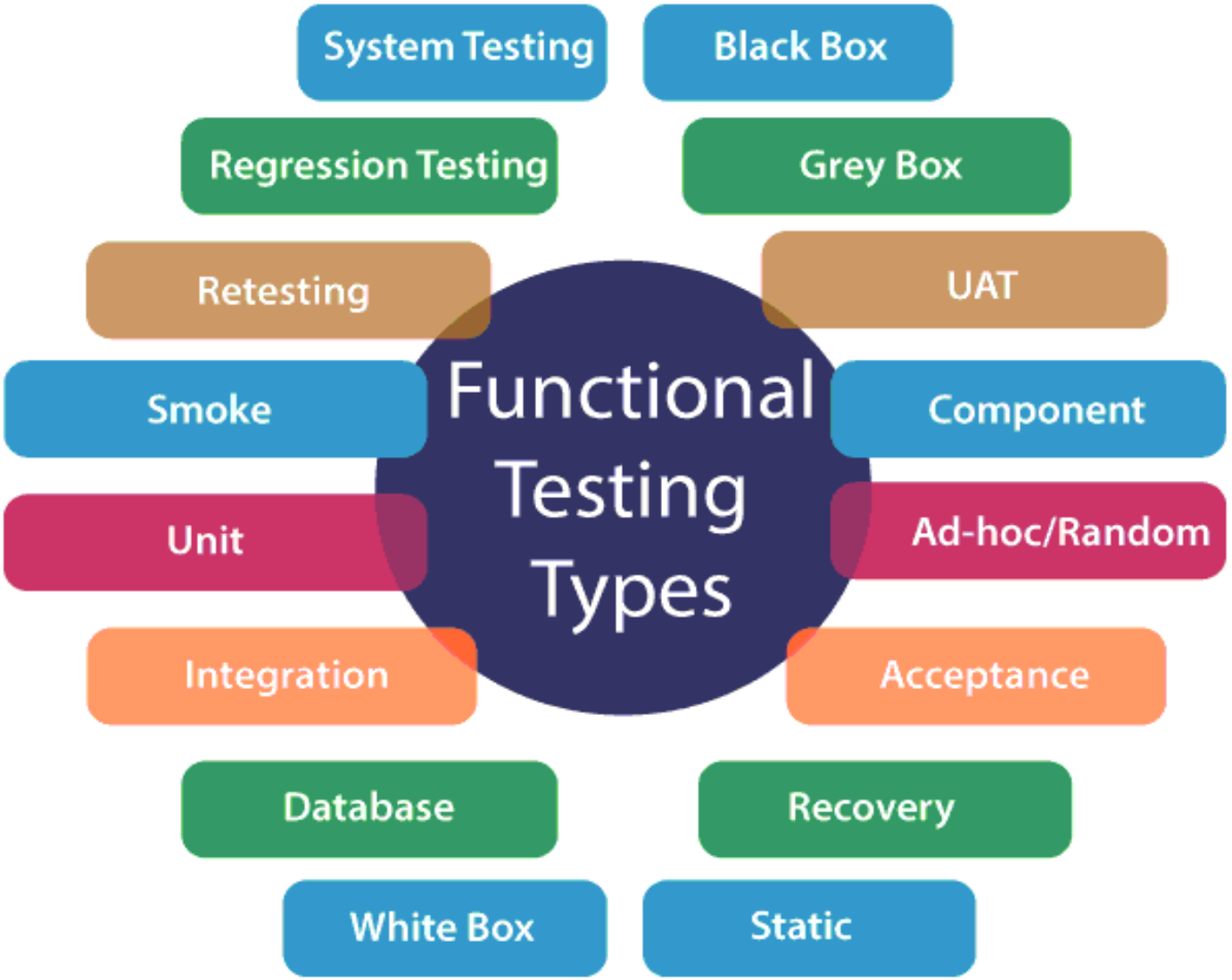
Comparison of actual and computed expected result

Functional Vs Non-Functional Testing:

Functional Testing	Non-Functional Testing
Functional testing is performed using the functional specification provided by the client and verifies the system against the functional requirements.	Non-Functional testing checks the Performance, reliability, scalability and other non-functional aspects of the software system.
Functional testing is executed first	Non-functional testing should be performed after functional testing
Manual Testing or automation tools can be used for functional testing	Using tools will be effective for this testing
Business requirements are the inputs to functional testing	Performance parameters like speed, scalability are inputs to non-functional testing.
Functional testing describes what the product does	Nonfunctional testing describes how good the product works
Easy to do Manual Testing	Tough to do Manual Testing

Functional Testing

Explain the types of functional testing.



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

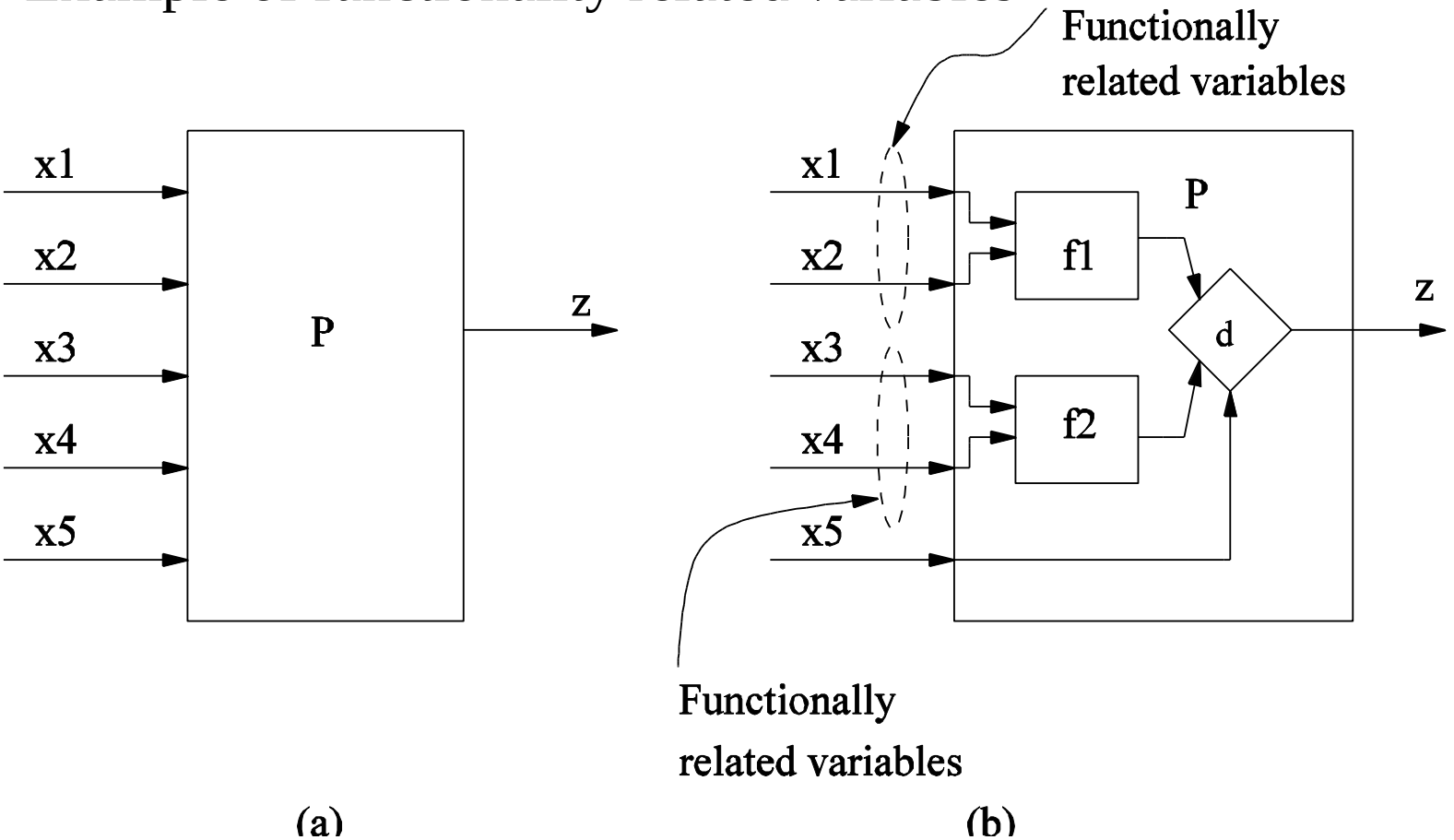


Figure 9.3: Functionality related variables

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

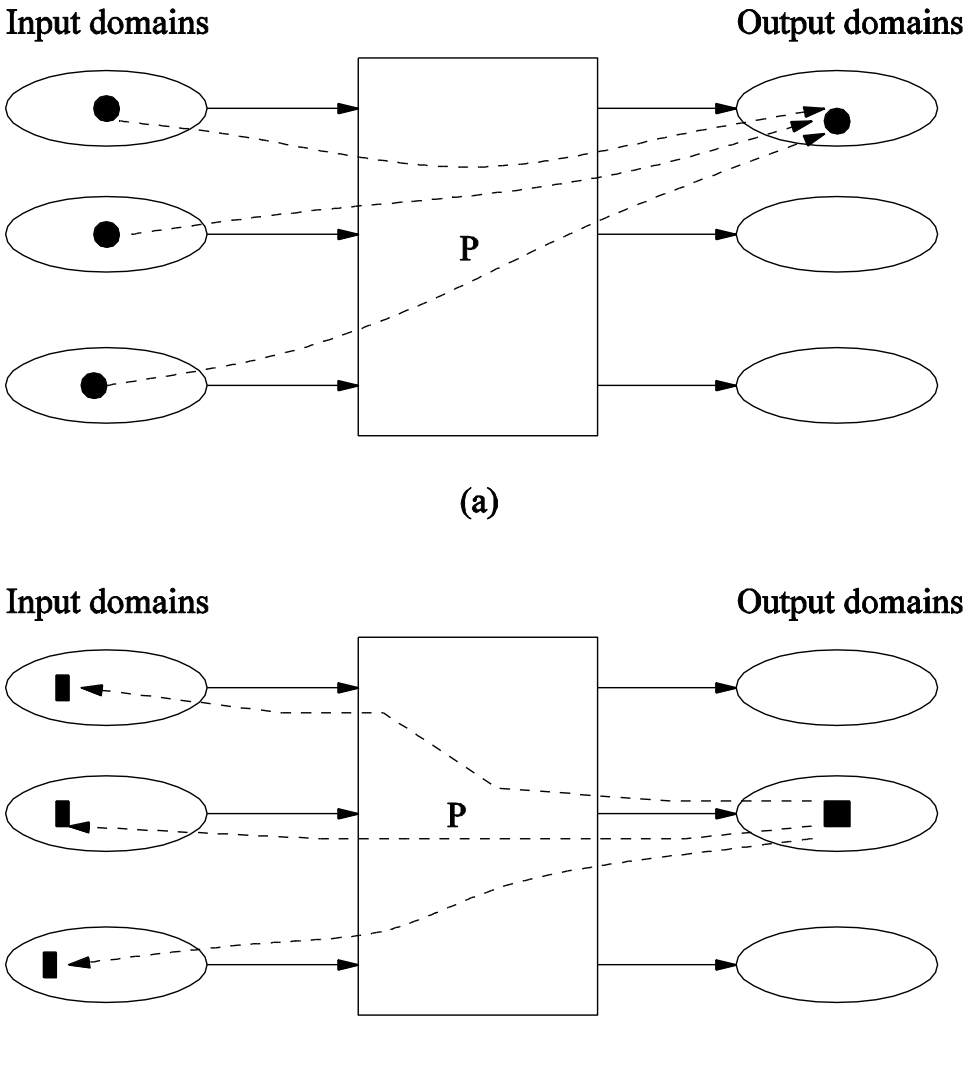


Figure 9.5: Obtaining output values from an input vector (a), and obtaining an input vector from an output value (b) in functional testing

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.

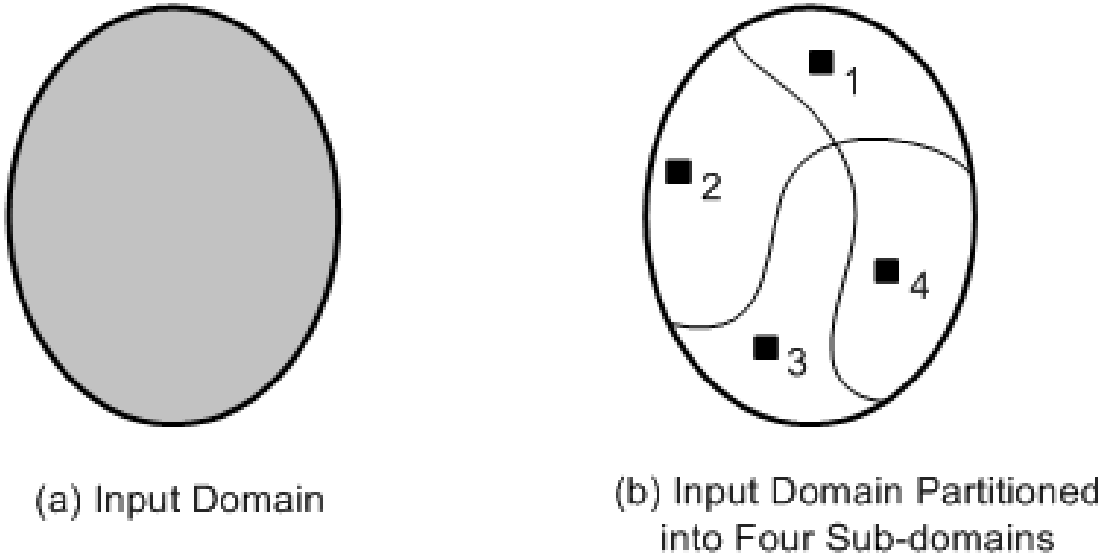


Figure 9.8: (a) Too many test input; (b) One input is selected from each of the subdomain

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