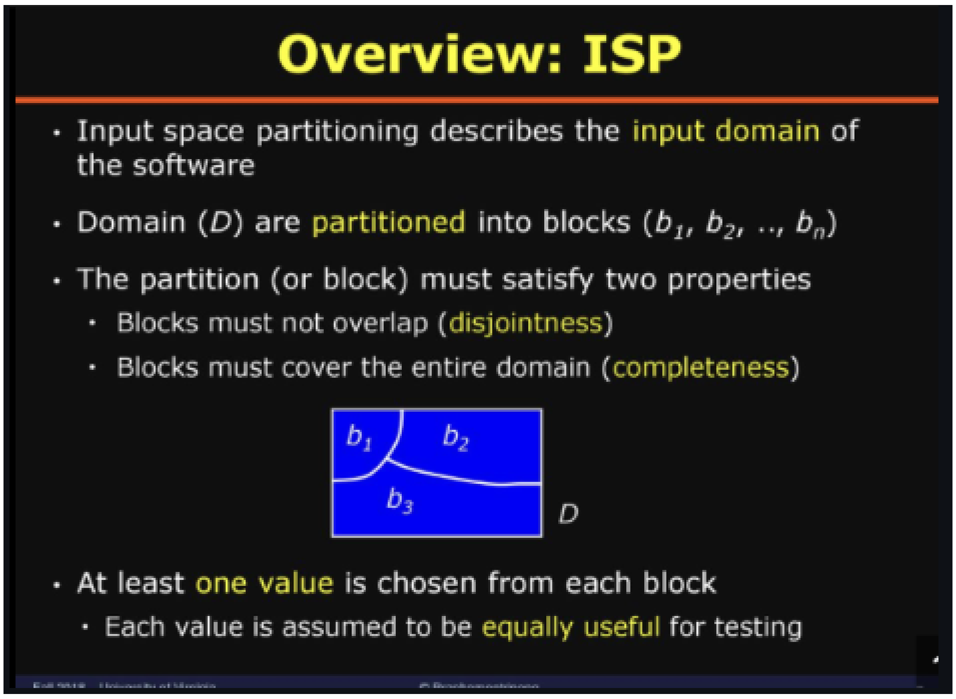
Module 4

farisbashatm@gmail.com

Input Space Partitioning

Definition: Input Space Partitioning (ISP) is a testing technique that divides the input domain of a software into distinct blocks, each assumed to contain values that are equally useful for testing.



Overview: ISP

- Input space partitioning describes the input domain of the software
- Domain (D) are partitioned into blocks (b_1, b_2, .., b_n)
- The partition (or block) must satisfy two properties
  - Blocks must not overlap (disjointness)
  - Blocks must cover the entire domain (completeness)
- At least one value is chosen from each block
  - Each value is assumed to be equally useful for testing

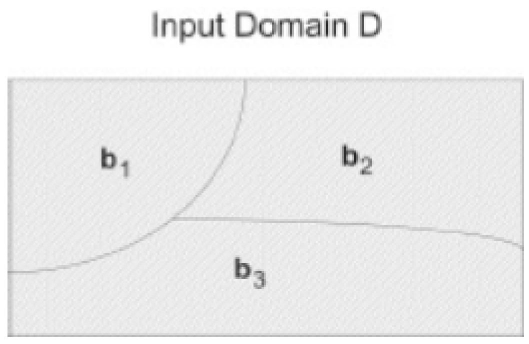
Key Concepts:

1. **Input Domain**: The set of all possible values that input parameters can take. These parameters can be method parameters, state objects, or user-level inputs, depending on the testing level (unit, integration, system).
2. **Partition**: The input domain is divided into blocks (or partitions) where each block represents a subset of possible input values.
3. **Equivalence Class**: Each block in a partition is assumed to contain values that are equivalent in terms of their testing effectiveness.

Steps:

1. **Identify Input Parameters**: Determine all possible inputs.
2. **Define Partitions**: Split the input domain into blocks (partitions) that cover all possible values without overlap.
3. **Select Test Values**: Choose representative values from each block.

Example:



For a function with an integer input:

- **Input Domain**: All integers.
- **Partitions**:
  - $b1$: Negative integers
  - $b2$: Zero
  - $b3$: Positive integers

Advantages:

- **Easy to Use**: Simple and quick to implement.
- **Independent of Implementation**: Based on input descriptions, not on how the software works.
- **Flexible**: Easy to adjust the number of tests.

Properties:

1. **Completeness**: Partitions cover the entire input domain.
2. **Disjointness**: No overlap between partitions.

Input Domain Modeling with Triang Example

Input Domain Modeling: The process of defining the input space for testing a function.

Steps in Input Domain Modeling:

1. **Identify Testable Functions**: Determine the methods or functions that need testing. For example, the `triang()` method, which classifies triangles based on side lengths.
2. **Identify Parameters**: Recognize all parameters affecting the function. For `triang()`, the parameters are the lengths of the three sides of the triangle.
3. **Model Input Domain**: Create an input domain model (IDM) to abstractly represent the input space. The IDM includes characteristics and partitions for each parameter.

Example: `triang()` Method

Parameters:

- `a`: Length of the first side
- `b`: Length of the second side
- `c`: Length of the third side

Interface-Based Approach:

Focuses on individual parameters.

- **Characteristics**:
  - Relation of `a` to zero
  - Relation of `b` to zero
  - Relation of `c` to zero

Functionality-Based Approach:

Focuses on the functionality or behavior.

- **Characteristics**:
  - Type of triangle: Equilateral, Isosceles, Scalene
  - Validity of triangle: Valid, Invalid (e.g., the sum of any two sides must be greater than the third)

Designing Characteristics

1. **Interface-Based**:
   - Simple translation from parameters to characteristics.
   - Example for `a`:
     - Is `a` zero? (True/False)
2. **Functionality-Based**:
   - Based on behavior or intended functionality.
   - Example for triangle type:
     - Type of triangle based on sides: Equilateral (all sides equal), Isosceles (two sides equal), Scalene (no sides equal).

TriTyp Classification Problem

The Triangle Classification Problem, also known as the TriTyp Problem, involves determining the type of triangle based on the lengths of its sides
The triangle problem is the most widely used example in software testing literature

- **Problem Statement**: Given three integers a, b, and c representing the sides of a triangle, determine the type of triangle or if it's not a triangle based on specific conditions.

- **Inputs**:
  - Integers a, b, and c representing the sides of the triangle.
  - Conditions:
    - $1 \leq a \leq 200$
    - $1 \leq b \leq 200$
    - $1 \leq c \leq 200$
    - $a < b + c$
    - $b < a + c$
    - $c < a + b$
- **Outputs**:
  - If any input value fails conditions $c1$, $c2$, or $c3$, output a message indicating the failure.
  - If all conditions are met:
    - If all sides are equal, output "Equilateral".
    - If exactly one pair of sides is equal, output "Isosceles".
    - If no pair of sides is equal, output "Scalene".
    - If any of conditions $c4$, $c5$, or $c6$ is not met, output "NotATriangle".
- **Example Fix**:
  - For the input set (2, 2, 5):
    - The condition $c6$ is not met, as c is greater than $a + b$.
    - Therefore, the output should be "NotATriangle" because the values do not form a valid triangle.

> Code is not necessary !!

Here's a simple example of the `triang()` method and how it might be tested using these approaches:

Conditions for `triang` Function

1. **Invalid Triangle Conditions**:
   - Any side is less than or equal to zero.
   - The sum of any two sides is less than or equal to the third side.
2. **Triangle Type Conditions**:
   - **Equilateral**: All three sides are equal.
   - **Isosceles**: Exactly two sides are equal.
   - **Scalene**: No sides are equal.

```python
def triang(a, b, c):
    if a <= 0 or b <= 0 or c <= 0:
        return "Not a triangle"
    if a + b <= c or a + c <= b or b + c <= a:
        return "Not a triangle"
    if a == b == c:
        return "Equilateral"
    if a == b or a == c or b == c:
        return "Isosceles"
    return "Scalene"

# Interface-Based Testing
print(triang(0, 1, 2))  # Not a triangle
print(triang(3, 4, 5))  # Scalene

# Functionality-Based Testing
print(triang(1, 1, 1))  # Equilateral
print(triang(2, 2, 3))  # Isosceles
print(triang(1, 2, 3))  # Not a triangle
```

Input domain modeling is crucial for systematic testing. It involves identifying testable functions, determining parameters, and creating an input domain model. The model can be developed using interface-based or functionality-based approaches, each with its strengths and weaknesses. The `triang()` example demonstrates how different characteristics and partitions are defined for effective testing.

Combination Strategies Criteria (Multiple partitions)

When dealing with multiple partitions simultaneously, the challenge is to decide how to combine blocks from different characteristics into test cases. Here are several strategies:

1. All Combinations Coverage (ACoC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Test cases for ACoC
(A, 1, x)
(A, 1, y)
(A, 2, x)
(A, 2, y)
(A, 3, x)
(A, 3, y)
(B, 1, x)
(B, 1, y)
(B, 2, x)
(B, 2, y)
```

```
(B, 3, x)
(B, 3, y)
```

- **Definition:** Use all possible combinations of blocks from all characteristics.
- **Example:** For three partitions with blocks [A, B], [1, 2, 3], and [x, y], ACoC requires 12 tests: (A, 1, x), (B, 1, x), (A, 1, y), etc.
- **Drawback:** Impractical for many partitions due to a high number of tests.

2. Each Choice Coverage (ECC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Test cases for ECC
(A, 1, x)
(B, 2, y)
(A, 3, x)
```

- **Definition:** Ensure that each block from every characteristic is used in at least one test.
- **Example:** For three partitions with blocks [A, B], [1, 2, 3], and [x, y], ECC can be satisfied with tests like (A, 1, x), (B, 2, y), and (A, 3, x).
- **Drawback:** May result in weak test cases as it doesn't combine all blocks.

3. Pair-Wise Coverage (PWC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Test cases for PWC
(A, 1, x)
(A, 2, y)
(A, 3, y)
(B, 1, y)
(B, 2, x)
(B, 3, x)
```

- **Definition:** Ensure each block from each characteristic is paired with every block from every other characteristic.

- **Example:** For three partitions, PWC needs to cover combinations like (A, 1), (B, 1), (1, x), (A, 2), etc., which can be combined efficiently into test cases.
- **Benefit:** Reduces the number of tests compared to ACoC while providing stronger coverage than ECC.

4. T-Wise Coverage (TWC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Test cases for TWC (t=2)
(A, 1)
(A, 2)
(A, 3)
(B, 1)
(B, 2)
(B, 3)
(1, x)
(1, y)
(2, x)
(2, y)
(3, x)
(3, y)
```

- **Definition:** Ensure that each group of t characteristics has all combinations of blocks tested.
- **Example:** For t = 3 and three characteristics each with four blocks, TWC ensures comprehensive testing.
- **Drawback:** Expensive in terms of the number of test cases; going beyond pair-wise (t=2) is often not very beneficial.

5. Base Choice Coverage (BCC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Base test for BCC
(A, 1, x)

# Additional tests
(B, 1, x)
(A, 2, x)
```

```
(A, 3, x)
(A, 1, y)
```

- **Definition:** Choose a "base" block for each characteristic and create a base test. Additional tests are created by varying one characteristic at a time from the base test.
- **Example:** For three partitions with base choices (A, 1, x), the base test is (A, 1, x). Additional tests include (B, 1, x), (A, 2, x), etc.
- **Benefit:** Allows focusing on important blocks, useful for stress testing.

6. Multiple Base Choice Coverage (MBCC)

```
# Partitions with blocks
Partition 1: [A, B]
Partition 2: [1, 2, 3]
Partition 3: [x, y]

# Base tests for MBCC
Base 1: (A, 1, x)
Base 2: (B, 2, y)

# Additional tests from Base 1
(A, 2, x)
(A, 3, x)
(A, 1, y)
(B, 1, x)

# Additional tests from Base 2
(B, 3, y)
(B, 2, x)
(A, 2, y)
(B, 2, y)
```

- **Definition:** Use multiple base choices for each characteristic, forming multiple base tests. Additional tests vary one characteristic at a time from each base test.
- **Example:** For two base choices for side 1 in `triang()`, tests are created from (2, 2, 2) and (1, 2, 2), with additional tests created by varying other characteristics.
- **Benefit:** Provides flexibility and more thorough testing, but can result in duplicates.

By applying these combination strategies, testers can balance between comprehensive testing and practical constraints, ensuring efficient and effective test coverage.

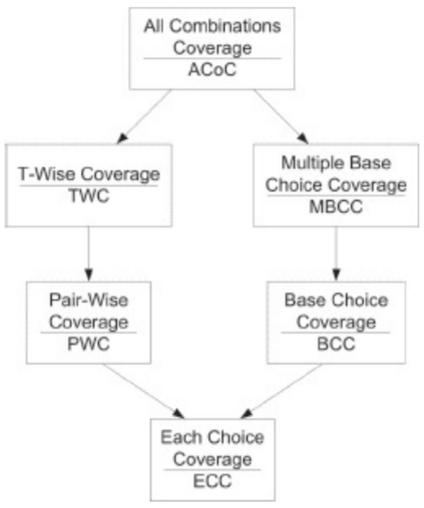


Figure 6.2. Subsumption relations among input space partitioning criteria.

==

Functional Testing Concepts of Howden

William E. Howden developed the idea of functional testing during his time at the International Mathematics and Statistics Libraries (IMSL) in Houston (1977-1978). IMSL, now known as Visual Numerics, provides a comprehensive set of mathematical and statistical functions embedded in software applications. Howden applied functional testing to programs in the IMSL package, discovering subtle errors even in later editions.

In functional testing, a program P is viewed as a function that transforms an input vector X_i into an output vector Y_i, represented as $Y_i = P(X_i)$. Here are four examples to illustrate this:

1. **Square Root Function:** $Y_1 = \sqrt{X_1}$ computes the square root of a nonnegative integer X_1.
2. **C Compiler:** $Y_2 = C\_compiler(X_2)$ transforms a C program X_2 into object code Y_2.
3. **Telephone Switch:** $Y_3 = TelephoneSwitch(X_3)$ produces various tones and voice signals Y_3 based on input data X_3.
4. **Sorting Algorithm:** $Y_4 = sort(X_4)$ sorts an array A of N elements.

Functional testing focuses on the input, output, and expected transformation without delving into the details of the transformation mechanism.

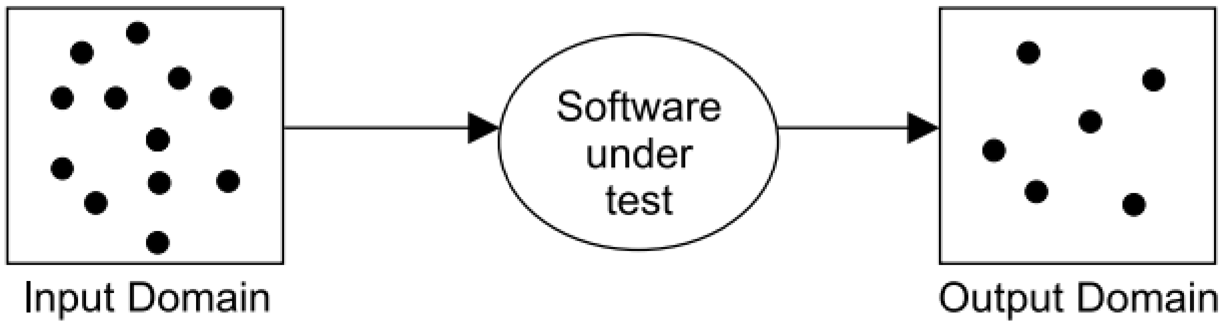
The key 4 concepts in functional testing are:

- **Identify Input and Output Domains:** Precisely determine the domains for each input and output variable.
- **Select Values with Important Properties:** Choose values from the data domain that have significant properties.
- **Combine Special Values:** Design test cases by considering combinations of special values from different input domains.

- **Produce Special Output Values:** Choose input values that result in special values in the output domains.

These principles help in generating effective test data by analyzing the domains of input and output variables, often derived from requirements and design documents.

Functional Testing



Functional testing is crucial but effort-intensive, aiming to identify test cases that likely cause software failures. Failures occur when observed behavior deviates from expected behavior, revealing software flaws during execution. Functional testing focuses on ensuring every software functionality is tested by designing test cases based on functionality, ignoring the internal structure. This approach is known as black box testing, where observed outputs are compared with expected outputs for given inputs.

Key points:

- **Objective:** Design test cases likely to cause failures and ensure all functionalities are tested.
- **Method:** Treat software as a black box, focusing on user requirements and not the internal structure.
- **Validation:** Both valid and invalid inputs are used to observe software behavior.
- **Levels:** Applicable at unit, integration, system, and acceptance testing levels.
- **Goal:** Create efficient and effective test cases to identify software faults.

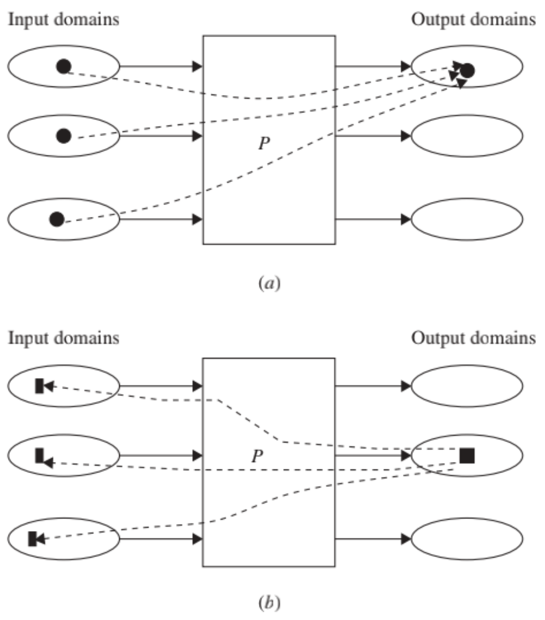


Figure 9.5 (*a*) Obtaining output values from an input vector and (*b*) obtaining an input vector from an output value in functional testing.

Important Steps in Functional Testing

1. **Identify Test Input**:
   - Determine the functionality to be tested, including usability, main functions, and error conditions.
2. **Compute Expected Outcomes**:
   - Create input data according to the function's specifications and establish the expected output.
3. **Execute Test Cases**:
   - Run the designed test cases as per the defined input and record the resulting output.
4. **Compare Actual and Expected Output**:
   - Analyze the actual output against the expected output to detect any deviations or discrepancies.

Some types of functional testing:

1. EQUIVALENCE CLASS PARTITIONING
2. Boundary Value Analysis,
3. Decision Tables,
4. Random Testing.

1.Equivalence Class Partitioning

Equivalence Class (EC) Partitioning simplifies testing by dividing a large input domain into smaller subdomains, each known as an equivalence class. This method ensures thorough testing by selecting representative inputs from each class (Figure 9.8).

Steps for Equivalence Class Partitioning:

1. **Identify Input Conditions**: Determine the conditions that define each equivalence class.
2. **Define Equivalence Classes**: Create ECs based on:
   - **Range**: For $a \leq X \leq b$, $X < a$, and $X > b$.
   - **Set of Values**: For a set of N values, create $N + 1$ ECs (each value and one invalid).
   - **Individual Values**: Create ECs for each valid input.
   - **Number of Values**: For N valid inputs, create ECs for N, zero, and more than N inputs.
   - **Must-be Value**: Create ECs for the required value and its negation.
   - **Splitting ECs**: Further divide ECs if necessary.

Identification of Test Cases:

1. **Assign Unique Numbers**: Number each EC.
2. **Valid ECs**: Write test cases covering as many uncovered valid ECs as possible.
3. **Invalid ECs**: Write test cases covering one uncovered invalid EC at a time.

Advantages:

- Reduces the number of test cases.
- Clarifies the input domain.
- Increases defect detection probability.
- Applicable to both input and output domains.

Figure 9.8:

- (a) Too many test inputs.
- (b) Input domain partitioned into subdomains.

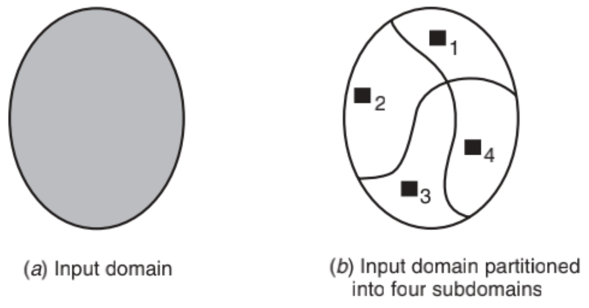


Figure 9.8 (a) Too many test inputs; (b) one input selected from each subdomain.

Equivalence Class Partitioning Example: Triangle Classification

Given three integers a, b, and c representing the sides of a triangle, determine the type of triangle or if it is not a triangle.

Inputs

- Integers a, b, and c (1 to 200)
- Conditions:
  - $a < b + c$
  - $b < a + c$
  - $c < a + b$

Outputs

- "Equilateral" if $a = b = c$
- "Isosceles" if exactly one pair of sides is equal
- "Scalene" if no pair of sides is equal
- "NotATriangle" if conditions are not met

Equivalence Classes

EC #	Description	Example Inputs	Validity
EC1	$1 \leq a, b, c \leq 200$	(3, 4, 5)	Valid
EC2	a, b, c out of bounds	(0, 4, 5)	Invalid
EC3	$a = b = c$	(5, 5, 5)	Valid
EC4	Exactly one pair of sides equal	(5, 5, 7)	Valid
EC5	All sides different	(3, 4, 5)	Valid
EC6	Not satisfying triangle inequality	(1, 2, 3)	Invalid

Test Cases from ECs

Test Case #	ECs Covered	a	b	c	Expected Output
TC1	EC1, EC5	3	4	5	Scalene
TC2	EC1, EC3	5	5	5	Equilateral
TC3	EC1, EC4	5	5	7	Isosceles
TC4	EC2	0	4	5	Invalid input
TC5	EC1, EC6	1	2	3	NotATriangle

Equivalence Class Partitioning helps reduce the number of test cases while ensuring comprehensive coverage.

2.Boundary Value Analysis (BVA)

Boundary Value Analysis (BVA) is a software testing technique that focuses on selecting test data near the boundaries of a data domain to detect defects. These boundaries are where errors are likely to occur due to the incorrect implementation of conditions.

Key Points of BVA

1. **Boundary Conditions**: Predicates applied directly around the boundaries of input and output Equivalence Classes (ECs) to find potential defects.
2. **Test Data Selection**: Test cases are chosen on or near the boundary to ensure edge-case coverage, revealing defects often overlooked by designers.

Guidelines for BVA

1. **Range Specifications**: Test at the boundary points and just beyond. For example, for a range $-10.0 \leq X \leq 10.0$, use $-10.1, -10.0, -9.9$ and $10.1, 10.0, 9.9$.
2. **Number Specifications**: Test the minimum, maximum, and values just outside the range. For instance, if a dormitory houses 1 to 4 students, test with 0, 1, 4, and 5 students.
3. **Ordered Sets**: Focus on the first and last elements of ordered sets, such as lists or tables.

BVA helps create effective test cases that reveal defects, ensuring comprehensive software testing.

Boundary Value Analysis: Triangle Classification Problem

Given three integers a, b, and c representing the sides of a triangle, classify the triangle type or determine if it is not a triangle.

Inputs

- Integers a, b, and c (range: 1 to 200)
- Conditions for a valid triangle:
  - $a < b + c$
  - $b < a + c$
  - $c < a + b$

Outputs

- "Equilateral" if $a = b = c$
- "Isosceles" if exactly one pair of sides is equal
- "Scalene" if no sides are equal
- "NotATriangle" if the triangle conditions are not met

Boundaries:

- Minimum value: 1
- Maximum value: 200

Test Cases

Test Case #	a	b	c	Expected Output	Boundary Condition Covered
TC1	1	1	1	Equilateral	Minimum boundary values
TC2	200	200	200	Equilateral	Maximum boundary values
TC3	1	1	200	NotATriangle	Minimum + maximum boundary
TC4	0	5	5	Invalid input	Below minimum boundary
TC5	201	5	5	Invalid input	Above maximum boundary

3.Decision Tables

Decision tables are a powerful tool for modeling complex software requirements and design decisions by considering multiple input combinations. They are particularly useful for situations where different combinations of conditions need to be tested to ensure correct software behavior.

Structure of a Decision Table

A decision table consists of:

1. **Conditions (or Causes)**: Inputs or input conditions.
2. **Values**: Possible values for each condition (e.g., Yes (Y), No (N), Don’t care (—)).
3. **Rules**: Columns representing different combinations of conditions.
4. **Effects (or Results)**: Outputs or actions corresponding to each rule.

Steps to Develop Decision Tables

1. **Identify Conditions and Effects**: Determine the input conditions and corresponding outputs.
2. **List Conditions and Effects**: Arrange them in a table format.
3. **Calculate Possible Combinations**: Determine all possible combinations of conditions.
4. **Fill Combinations**: Populate the table with all possible condition combinations.
5. **Reduce Combinations**: Merge columns where conditions lead to the same effect.
6. **Verify Coverage**: Ensure all possible combinations are represented.
7. **Add Effects**: Specify the effects for each combination of conditions.
8. **Generate Test Cases**: Each rule in the table becomes a test case.

Example: Senior Citizen Discount Eligibility Decision Table

Determine if a person is eligible for a senior citizen discount based on their age.

Inputs

- Age of the person

Outputs

- "Discount" if the person is eligible for a discount
- "No Discount" if the person is not eligible for a discount

Conditions

1. Age < 65
2. Age >= 65

Decision Table

Conditions	Values	Rule 1	Rule 2
Age < 65	Y/N	Y	N
Age >= 65	Y/N	N	Y
Effects			
Discount		-	1
No Discount		1	-

Explanation

- **Rule 1**: If Age < 65, the person gets No Discount.
- **Rule 2**: If Age >= 65, the person gets a Discount.

This table provides a straightforward way to determine senior citizen discount eligibility based solely on age.

4.Random Testing

Random testing is a software testing approach where test inputs are randomly selected from the input domain of the system. Here's an overview, using a simple example:

Steps in Random Testing:

1. **Identify Input Domain:** Determine the range of possible inputs.

2. **Select Test Inputs:** Randomly choose inputs from the domain.
3. **Execute System:** Run the program with these inputs.
4. **Compare Results:** Validate outputs against expected results.

Advantages:

- **Reliability Estimation:** If test inputs mimic real-world usage, statistical estimates for program reliability can be derived from test outcomes.
- **Automation-Friendly:** Large input sets can be generated automatically.

Test Oracle Requirement:
Effective verification of results requires a test oracle, a mechanism that determines the correctness of outputs. Types of oracles include:

- **Perfect Oracle:** Uses a defect-free version of the system.
- **Gold Standard Oracle:** Compares with previous versions of the system.
- **Parametric Oracle:** Compares parameters derived from outputs.
- **Statistical Oracle:** Compares statistical characteristics of outputs.

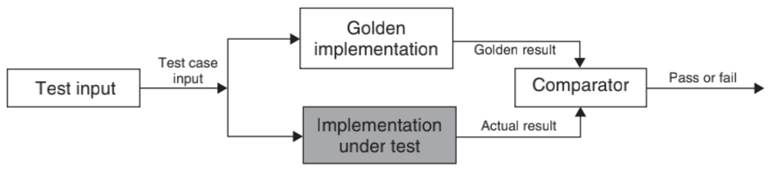


Figure 9.9 Gold standard oracle.

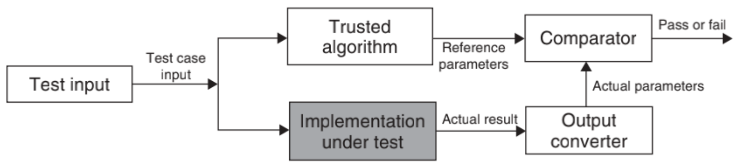


Figure 9.10 Parametric oracle.

Adaptive Random Testing:
An enhanced version, adaptive random testing, ensures inputs are evenly spread across the input domain. This method typically outperforms ordinary random testing by reducing the number of tests needed to find the first failure.

Example of Random Testing for Triangle Classification Problem

Given three integers a, b, and c representing the sides of a triangle, classify the type of triangle or determine if it is not a triangle.

Steps for Random Testing:

1. **Identify Input Domain:**
   - Integers a, b, and c with values ranging from 1 to 200.
2. **Select Test Inputs:**
   - Generate random values for a, b, and c within the specified range.

3. **Execute System:**
    - Run the triangle classification program on each set of generated inputs.
4. **Compare Results:**
    - Manually verify the classification of each triangle and compare it with the program's output.

Example Random Test Cases:

1. **Inputs:** $a = 3$, $b = 4$, $c = 5$
    - **Expected Output:** "Scalene" (all sides are different)
    - **Program Output:** Verify against expected result.
2. **Inputs:** $a = 1$, $b = 1$, $c = 1$
    - **Expected Output:** "Equilateral" (all sides are equal)
    - **Program Output:** Verify against expected result.
3. **Inputs:** $a = 1$, $b = 2$, $c = 3$
    - **Expected Output:** "NotATriangle" (fails triangle inequality $1 + 2 \leq 3$)
    - **Program Output:** Verify against expected result.

===
=

Questions & Answer

Approach	Interface-Based	Functionality-Based
Definition	Derive characteristics directly from input parameters.	Derive characteristics from the behavior of the system.
Focus	Individual input parameters.	Overall behavior and functionality of the system.
Characteristics	Based on properties of input domains.	Based on expected behavior of the system under test.
Example	Partitioning input parameter regions (e.g., null, empty).	Analyzing system behavior (e.g., occurrences of elements in a list).
Usage	Suitable for understanding input parameter properties.	Ideal for comprehending and testing system behavior.

Aspect	Computation Error	Domain Error
Definition	A computation error occurs when a specific input data causes the correct path to execute, but the output value is wrong.	A domain error occurs when a specific input data causes the program to execute a wrong (i.e. undesired) path
Example	Adding 0.1 and 0.2 in a programming language results in	Attempting to calculate the square root of a negative number.

Aspect	Computation Error	Domain Error
	0.30000000000000004 due to floating-point precision issues.	
Cause	Faulty implementation, algorithmic mistakes, or rounding errors.	Input values violating constraints or assumptions.
Impact	Results in incorrect output or unexpected behavior.	Causes program crashes or undefined behavior.
Detection	Detected during testing or validation of computations.	Detected through boundary checking or input validation.
Resolution	Requires debugging, code review, or algorithm refinement.	Handling through input validation or error handling mechanisms.
Additional Types	Overflow Error , Underflow Error	Closed boundary, Open boundary, Closed domain, Open domain

Aspect	Pairwise Coverage	T-wise Coverage
Definition	Covers all combinations of any two parameters.	Covers all combinations of any t parameters.
Example	For any two parameters, all value combinations are covered.	For any t parameters, all value combinations are covered.
Specificity	Focuses on pairs of parameters.	Can be tailored to cover any number of parameters together.
Generalization	A special case of T-wise coverage where T equals 2.	Encompasses all lower-order combinations, including pairwise coverage.
Application	Suitable for reducing the number of test cases while maintaining adequate coverage.	Offers flexibility to balance coverage and test suite size based on specific testing requirements.

Aspect	ACoC Testing	ECC Testing
Methodology	Systematic, covering all combinations	Ad hoc, focusing on likely error-prone areas
Coverage	Exhaustive	Non-exhaustive
Suitability	Critical systems with complex interactions	Quick identification of potential faults
Efficiency	Resource-intensive	Generally more efficient
Risk Mitigation	Mitigates risks associated with complex interactions	Identifies and addresses risks early
Complexity	Can be complex for systems with numerous input categories	Generally less complex