Software Testing Methodologies CIE I

1. What is Software Testing? Is Complete Testing Possible?

Software testing is the process of evaluating and verifying that a software application or system works as intended. It aims to detect errors, gaps, or missing requirements.

Complete testing (i.e., testing all possible inputs and paths) is **not possible** for most real-world applications due to the infinite number of input combinations and execution paths. Hence, testing is done using strategies like equivalence partitioning, path testing, etc., to gain confidence in correctness.

2. What is meant by Transaction Flow Testing?

Transaction Flow Testing is a **white-box testing technique** that focuses on **logical transactions** rather than individual program paths. It uses **Transaction Flow Graphs** to represent sequences of operations (or transactions) and tests them for correct processing.

Example: A user login followed by a dashboard view can be modeled and tested as a transaction flow.

3. Define Path Predicate with an example?

A **path predicate** is a **Boolean expression** formed by the **conditions along a specific path** in the program's control flow graph. It determines whether a path is **executable** under certain input values.

Example:

```
if (x > 0) {
  if (y < 10) {
    // path A
  }
}
```

Path predicate for path A: `(x > 0) AND (y < 10)`

4. How does Interface testing relate to domain testing?

Interface Testing in domain testing checks how **different input domains or components interact** with each other at their boundaries. It ensures that the software behaves correctly when inputs from one domain are passed to another module or interface.

It's important in **boundary value analysis** and helps detect mismatches or data interpretation errors between domains.

5. How are regular expression used in path testing?

Regular expressions are used to **represent sets of paths** in the control flow graph **compactly**.

They help in modeling **all possible execution paths** using symbols and operators (like `*`, `|`, `()`) and are particularly useful in **identifying redundant or invalid paths**.

Example:

For a loop, paths like `AB*C` can be expressed to denote paths from A to C with 0 or more B's.

6. Explain the reduction procedure for simplifying flow graphs

The **reduction procedure** simplifies complex flow graphs by **replacing subgraphs** with **simpler symbols** without changing their logical behavior.

Steps:

- Identify regions like **sequences**, **loops**, and **decisions**.
- Replace these with **single nodes or edges**.
- Repeat until the graph reduces to a single node or simpler form.

This helps in **easier analysis** and finding **independent paths** for testing.

---

1. Differentiate Achievable Non Achievable Paths in path testing and examples

Aspect	**Achievable Path**	**Non-Achievable Path**
Definition	A path that can be traversed with valid input	A path that cannot be traversed under any condition
Input Condition	Exists and makes the path executable	No input condition makes the path executable
Path Predicate	Satisfiable (evaluates to true for some input)	Unsatisfiable (always false)
Testing Purpose	Included in test cases for coverage	Eliminated from test case design
Code Example	`if (x > 5) { /* path A */ }` with x = 6	`if (x > 5 && x < 3) { /* path B */ }` (always false)
Tool Support	Detected as executable by static/dynamic tools	Detected as unreachable code

Example for Achievable:

```
if (x > 0) {
    printf("Positive");
}
```

- Path: Entry → Condition True → Print → Exit
- Achievable when `x = 5`

Example for Non Achievable:

```
if (x > 0 && x < 0) {
    printf("Impossible");
```

```
}
```

- Condition `x > 0 && x < 0` is **always false**
- No value of `x` satisfies it → Path is **non-achievable**

Impact on Path Testing:

- Identifying **achievable paths** ensures **realistic test cases**.
- Removing **non-achievable paths** avoids **wasted effort** in test design

2. Discuss applications of path products in test coverage analysis

A **path product** is an **algebraic representation** of all execution paths through a program or control flow graph (CFG). It uses **concatenation, choice (|), and iteration (*)** operators similar to regular expressions

Syntax Overview:

- **Concatenation (AB)**: Sequential execution of paths A and B
- **Choice (A | B)**: Either path A or path B
- **Iteration (A*)**: Zero or more repetitions of path A

Application	Explanation
1. **Compact Path Representation**	Path products represent multiple possible execution paths concisely, helping testers see the big picture.
2. **Independent Path Identification**	By analyzing path products, testers can isolate **linearly independent paths** for **basis path testing**.
3. **Loop Analysis**	Loops are represented with the `*` operator, allowing testers to design test cases for **zero, one, or many iterations**.
4. **Test Case Derivation**	By converting the path product into specific paths, one can directly derive test cases that ensure **branch or path coverage**.
5. **Dead Code Detection**	If certain code segments never appear in any path product, they may indicate **unreachable code**.
6. **Regression Testing Support**	When software is updated, testers can compare new path products to old ones to check **coverage changes**.

Consider a flow graph with this structure:

```
text
CopyEdit
Start → A → [B or C] → D → End
```

Path Product: `A(B | C)D`

- Represents two paths:

1. A → B → D
2. A → C → D

- Enables identifying minimum test cases for **decision/branch coverage**

Benefits in Coverage Analysis:

- Enhances **test completeness** by explicitly covering all possible control paths.
- Facilitates **structured test design** using mathematical principles.
- Avoids **redundant or infeasible path testing** by simplifying flow