STM [UNIT-2]

Long Answers [5M]

① Explain transaction flow testing with the help of a sample flow graph.

Ans) → It is a structural testing technique that uses a Transaction flow graph to design test cases.

→ The main goal is to ensure that all possible transaction paths are tested.

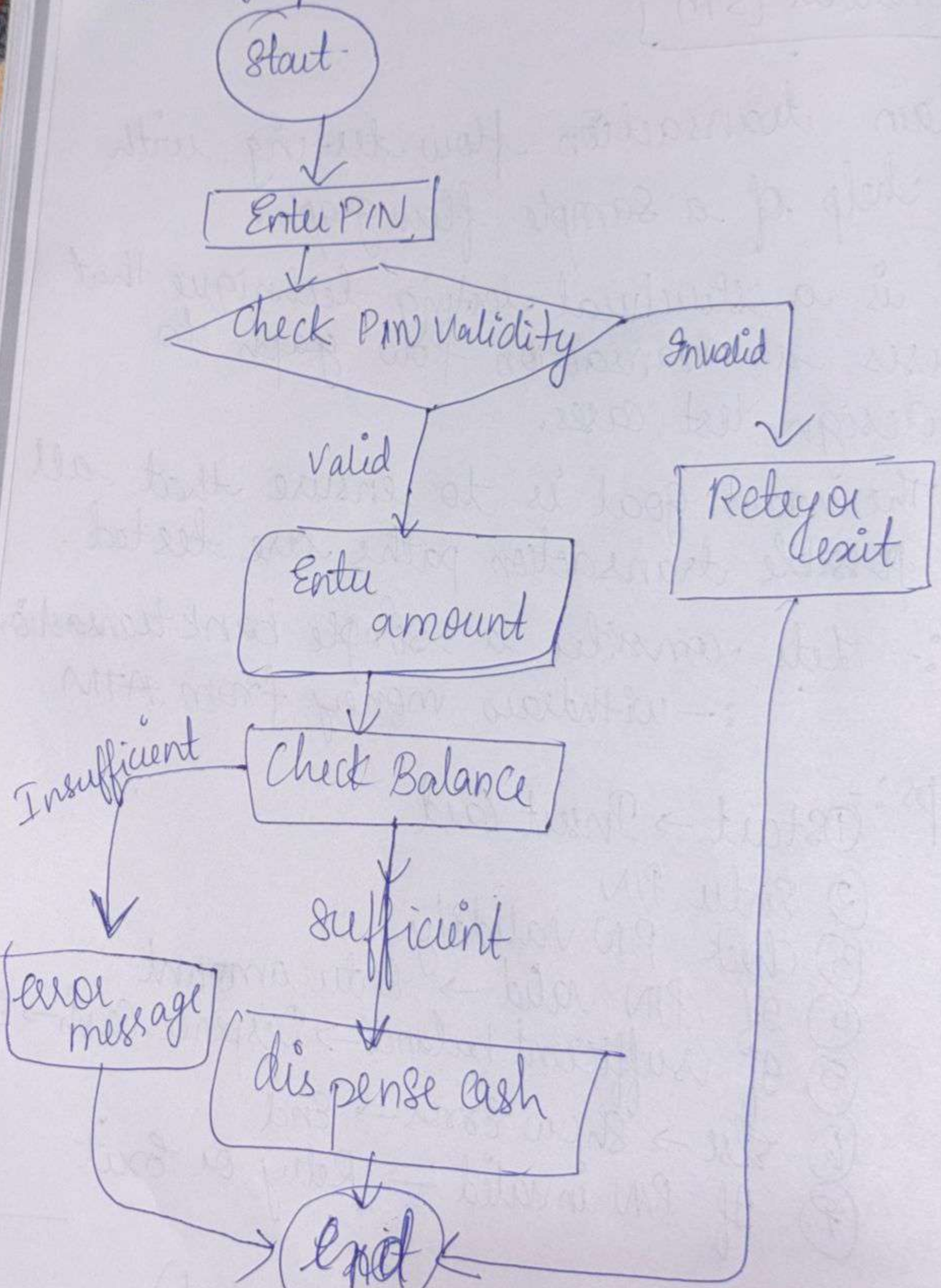
Ex:- Lets consider a simple bank transaction, :- withdraw money from ATM.

Steps:-
① Start → Insert Card
② Enter PIN
③ Check PIN validity
④ If PIN valid → Enter amount
⑤ If sufficient balance → Dispence Cash → End
⑥ Else → Show Error → End
⑦ If PIN invalid → Retry or Exit

extra

• It focuses on:-
→ Node coverage (all steps tested)
→ Edge coverage (all flows tested)
→ Path coverage (all transaction paths tested)

Flow graph



Start
Enter PIN
Check PIN Validity
Invalid
Valid
Retry or exit
Enter amount
Check Balance
Insufficient
sufficient
error message
dispense cash
End

(2) Describe the techniques used in transaction flow testing.

Ans The following techniques are commonly used:-

(i) Path coverage :-

→ Ensures that all independent transaction paths are executed at least once.

→ Covers different ways a transaction can proceed from start to end.

→ Ex:- In ATM transactions :-
Valid PIN + suff. balance,
valid PIN + insuff. balance,
invalid PIN.

(ii) Branch coverage :-

→ Ensures that all decision outcomes (True/False, Yes/No) in the transaction are tested.

→ Helps to verify that both success and failure flows are handled correctly.

→ Ex:- "Is balance sufficient?" → must test both Yes and No outcomes.

(iii) Node coverage :-

→ Ensures that every node in the transaction flow is executed at least once.

→ Guarantees that no step is left untested.

→ Ex:- Steps like :- ENTER PIN, Enter amount etc. must be executed atleast once.

(IV) Edge Coverage :-

→ Ensures that every control flow between nodes is tested.

→ Stronger than node coverage, as it validates all possible links between steps.

→ Ex :- From Check Balance to both :-
Dispense cash and Error msg.

(V) Loop Coverage :-

→ Ensures that loop in transaction flows are tested with :-
- Zero iterations
- One iteration
- Multiple iterations

→ Ex :- Retry option for entering PIN.

3)Write short notes on path sensitizing and its importance in path testing.

- Path sensitizing is a technique in path testing where specific input values are chosen to ensure that a particular path in the program is executed.
- Not all paths in the control flow graph can be taken with arbitrary inputs hence we need to sensitize (activate) a path using suitable inputs.
- Example:
  If a program has a decision if (x > 0) → to sensitize the *True path*, input must satisfy x > 0.

◈ Importance of Path Sensitizing in Path Testing

1. Ensures Path Feasibility
   - Not all theoretical paths are feasible (some may never execute).
   - Path sensitizing helps determine whether a path is practically executable.
2. Helps in Test Case Design
   - Guides testers in choosing exact input values that drive execution through the desired path.
3. Increases Coverage
   - Makes sure that different branches and conditions are covered effectively.
4. Detects Logical Errors
   - By forcing execution along specific paths, hidden logical bugs can be discovered.
5. Improves Program Reliability
   - More reliable testing since each path that *can* be executed is verified.

4) Discuss the role of predicates and path predicates in transaction flow testing.

- A predicate is a logical condition or decision point in the program that evaluates to either True or False.
- In a Transaction Flow Graph (TFG), predicates are represented as decision nodes.
- Predicates control the flow of execution by deciding which branch or path is taken.
- Example:
  if (balance >= amount) is a predicate that decides whether to allow withdrawal or show an error.

◈ Path Predicate

- A path predicate is a set of logical conditions (predicates) that must be satisfied for a specific path in the program to be executed.
- It is obtained by combining the results of all predicates along a path.

Example:
For ATM withdrawal → Path: *Valid PIN → Sufficient Balance → Dispense Cash*

- Path Predicate = (PIN == correct) AND (balance >= amount)

◈ Role in Transaction Flow Testing

1. Guides Path Sensitizing
   - Path predicates help determine input values that will execute a particular transaction path.
2. Ensures Correct Coverage
   - Each predicate and its possible outcomes must be tested to achieve branch and path coverage.
3. Identifies Feasible vs. Infeasible Paths

    - Some paths may be infeasible (cannot occur with any input).
    - Path predicates help detect and eliminate such paths.
4. Improves Test Case Design
    - By analyzing predicates, testers can create minimal but effective test cases that cover all important transaction paths.
5. Validates Business Logic
    - In transaction-based systems, predicates often represent critical business rules (e.g., *sufficient balance*, *valid account*).
    - Testing them ensures that business logic is correctly implemented.

5) Compare control flow testing and transaction flow testing with examples.

Aspect	Control Flow Testing	Transaction Flow Testing
Definition	Structural testing technique that focuses on the **control flow of a program** (loops, branches, decisions).	Structural testing technique that focuses on the **transaction flow** in a system (sequence of operations that form a transaction).
Basis	Based on the **Control Flow Graph (CFG)** of the program.	Based on the **Transaction Flow Graph (TFG)** of the system.
Focus	Tests **individual program logic paths** (conditions, loops, branches).	Tests **end-to-end transaction scenarios** (user/business-level flows).
Level	Works at the **code level**.	Works at the **system/business logic level**.
Coverage Types	- Statement coverage - Branch coverage - Path coverage - Loop coverage	- Node coverage - Edge coverage - Path coverage (transaction-based)
Example	Test cases ensure both **True and False** branches are executed.	ATM Transaction: - Insert card → Enter PIN → Valid → Withdraw cash - Insert card → Invalid PIN → Exit → Test cases ensure all **transaction outcomes** are tested.
Use Case	Good for **unit testing** and verifying program logic correctness.	Good for **integration/system testing** of business applications with multiple transactions.
Objective	Ensure all logical paths in **code execution** are tested.	Ensure all possible **transaction scenarios** are tested.

6) Explain the basic concept of dataflow testing

Data Flow Testing is a structural testing technique that focuses on how data (variables) are defined, used, and destroyed in a program.

Instead of just testing control flow, it tracks the lifecycle of variables to detect errors in variable usage.

◈ Variable Lifecycle in Data Flow Testing

Every variable passes through three main stages:

1. Definition (d) → Variable is assigned a value.
    - Example: x = 10
2. Use (u) → Variable is referenced.
    - c-use (computational use): Used in expressions (e.g., y = x + 1).
    - p-use (predicate use): Used in conditions (e.g., if (x > 0)).
3. Kill (k) → Variable goes out of scope or is redefined.

◈ Main Idea

- Test cases are designed to cover definition–use pairs (du-pairs).
- Ensures that each variable's value is properly assigned before use, and no unintended overwriting or omission happens.

Purpose & Benefits

- Detects anomalies like:
    - Using a variable **before definition**.
    - Defining a variable **but never using it**.
    - Using a variable **after it is killed**.
- Improves **reliability, efficiency, and correctness** of the software.

7) Describe the main dataflow testing strategies with examples.

- Data flow testing focuses on how variables are defined (d), used (u), and killed (k) in a program.
- A definition-use pair (du-pair) is a connection between the point where a variable is defined and where it is used.
- Test cases are designed to cover these du-pairs.

There are three main strategies:

1□All-defs Strategy

- Definition: For every variable definition in the program, at least one test case must ensure that the definition reaches some use of that variable.
- Focus: Ensures that every variable definition is tested with at least one use.

Ex:-

```
int x;
x = 10;
y = x + 5;
```

Variable x is defined at x=10 and then used in y = x + 5.

At least one test case must cover this du-pair.

2□All-uses Strategy

- Definition: For every variable definition, test cases must ensure that all possible uses of that variable (both c-use and p-use) are exercised.
- Focus: More thorough than All-defs, as it covers all uses.

Ex:-

```
int x;
x = 5;
if (x > 0)
  y = x + 1;
```

- x is defined at x = 5.
- Two uses:
  - p-use → if (x > 0)

- c-use → y = x + 1

- Test cases must ensure both uses are covered.

3️⃣ All-du-paths Strategy

- Definition: For every variable definition, all possible definition–use paths (du-paths) must be tested.
- Focus: Strongest strategy – ensures every path from definition to use is executed.

Example:

```
int x;
x = 7;
if (x > 0)
  y = x + 1;
else
  z = x - 1;   // c-use
```

- x = 7 defines x.
- du-paths from definition:
  - Path 1: x=7 → if(x>0) → y=x+1
  - Path 2: x=7 → if(x<=0) → z=x-1
- Test cases must ensure both du-paths are executed.

8) Identify different types of anomalies detected using dataflow testing.

Data flow testing tracks how variables are **defined (d)**, **used (u)**, and **killed (k)**.

- Sometimes variables are misused, leading to **data flow anomalies** (bugs related to variable usage).
- These anomalies are identified by analyzing the sequence of operations on variables.

🔹 Common Data Flow Anomalies

1️⃣ dd Anomaly (Define–Define)

- A variable is **defined again without being used** after the previous definition.
- Meaning → The first definition is wasted / redundant.

- **Example:**
- int x;
- x = 5;
- x = 10;   value 5 was never used

2️⃣ du Anomaly (Define–Use without Kill)

- This is **valid usage** and forms the basis of testing (not really an anomaly).
- A variable is **defined and then used properly** before redefinition or kill.
- **Example:**
- int x;
- x = 5;
- y = x + 2;

3️⃣ dk Anomaly (Define–Kill without Use)

- A variable is **defined and then killed** (goes out of scope or overwritten) **without being used**.
- Meaning → Wasteful definition.
- **Example:**
- int x;
- x = 20;
- // function ends → x killed → dk anomaly

4️⃣ ud Anomaly (Use–Define)

- A variable is **used before it is defined**.
- Very dangerous – leads to **undefined behavior**.
- **Example:**
- int x;
- y = x + 2;   // use before definition → ud anomaly

- x = 10;   // definition happens later

<u>5️⃣ uk Anomaly (Use–Kill)</u>

- A variable is **used and then immediately killed** without redefinition.
- May indicate unnecessary operations.
- **Example:**
- int x = 5;
- printf("%d", x); // use
- // function ends → x killed (uk anomaly)

<u>6️⃣ ku Anomaly (Kill–Use)</u>

- A variable is **killed (out of scope)** but later **used without redefinition**.
- Leads to **invalid memory reference**.
- **Example:**
- { int x = 5; }   // x killed (goes out of scope)

  y = x + 1;   // using killed variable → ku anomaly

- **9) Explain how path instrumentation is used in path testing.** Path instrumentation is the process of adding extra instructions (probes) into the program code so that the execution of paths can be monitored during testing.
- It helps testers verify whether a particular path in the program has actually been executed by a test case.

  💠 Why Path Instrumentation is Needed?

- In path testing, we want to ensure that independent paths are covered.
- But simply running test cases does not show which exact path was executed.
- Path instrumentation provides a way to trace path execution.

◈ How Path Instrumentation Works

1. Identify Paths
   - First, independent paths are identified using a Control Flow Graph (CFG).
2. Insert Instrumentation (Probes)
   - Small monitoring code/statements are inserted at nodes and edges of the program.
   - These probes record which nodes/edges are visited.
3. Execute Test Cases
   - When the program is run with a test case, the probes generate a trace of the executed path.
4. Verify Path Coverage
   - The recorded trace is compared with the intended path to confirm coverage.

Importance of Path Instrumentation

- Confirms **actual path execution** during testing.
- Helps measure **coverage** (which nodes/edges/paths have been tested).
- Detects **uncovered or infeasible paths**.
- Provides a **trace log** useful for debugging and analysis.

10) Analyze the effectiveness of transaction flow testing in detecting logical errors.

- Transaction Flow Testing (TFT) is a **white-box testing technique** based on a **transaction flow graph**, where each transaction is a sequence of operations from start to end.
- It focuses on testing **end-to-end business processes** (e.g., ATM withdrawal, online order).

◈ Logical Errors in Software

Logical errors occur when:

- The **intended business rule** is not followed.

- Transactions take **incorrect or unintended paths**.
- Steps in a transaction are **skipped, repeated, or wrongly ordered**.

🔹 **How TFT Detects Logical Errors**

1. **Path Coverage**
    - TFT ensures all important **transaction paths** are tested.
    - If a transaction logic is missing/incorrect (e.g., skipping balance check in ATM withdrawal), TFT can detect it.
2. **Predicate & Path Predicate Analysis**
    - TFT checks the **conditions** (predicates) controlling the flow.
    - Incorrect conditions (e.g., wrong eligibility check) will be exposed when invalid paths are taken.
3. **Error Trapping in Transactions**
    - TFT can reveal errors such as:
        - Missing transaction steps (e.g., skipping authentication).
        - Wrong branching logic (e.g., rejecting a valid transaction).
        - Incorrect end states (e.g., money debited but not updated in balance).
4. **System-Level Testing**
    - Unlike simple control flow, TFT tests **end-to-end scenarios**.
    - This makes it very effective in detecting **business logic flaws**.

🔹 **Example**

- **ATM Withdrawal Transaction**:
    1. Insert card → 2. Enter PIN → 3. Check balance → 4. Dispense cash → 5. Update balance.
- If due to a logical error, the program **dispenses cash before balance check**, TFT test cases will expose it by forcing execution along that faulty path.

🔷 Effectiveness Analysis

✅ Strengths

- Excellent for detecting **logic-related errors** in business processes.
- Ensures **critical transaction flows** are validated.
- Covers **end-to-end functionality**, not just individual units.

⚠️ Limitations

- May not detect **data-related errors** (e.g., wrong variable usage → better caught by dataflow testing).
- Requires accurate **transaction flow graph**, which may be difficult for complex systems.
- Cannot guarantee detection of **all logical errors**, especially if some paths are infeasible.