23IT5609 : STM III-II Sem

Mr. G Lachiram
Associate Professor
Department of IT

Introduction to Software Testing

Purpose of Testing

The primary purpose of software testing is to ensure that the software system is reliable, functions as expected, and meets the requirements specified by the users and stakeholders. Testing helps identify defects early, improves product quality, and ensures that the final product is dependable and user-friendly.

•Identify defects early

It refers to discovering bugs, errors, or flaws in the software during early stages of development (e.g., during requirements review, design review, unit testing) rather than after release.
Early detection can happen even before full system testing — within modules, components, or during integration.

•Ensure software quality and reliability

Ensuring software quality and reliability means verifying that the software consistently performs its intended functions correctly, without failures, under all expected conditions. Testing helps confirm that the software meets the required standards, behaves predictably, and provides a stable user experience.

Dichotomies in Testing

Static Testing
Testing without executing the code.
Focuses on reviewing documents, code, and design.
Detects errors early in the development cycle.
Examples: Code review, walkthroughs, inspections, static analysis tools.
Dynamic Testing
Testing by executing the code.
Validates the functional behavior and performance of the software.
Detects runtime errors.
Examples: Unit testing, Integration testing, System testing, Acceptance testing.

Black-Box Testing
Tester does **not** know internal code or structure.
Focuses on input-output behavior and functional requirements.
Suitable for functional testing.
Techniques: Equivalence partitioning, Boundary value analysis, Decision tables.
White-Box Testing
Tester knows the internal logic and code structure.
Focuses on paths, conditions, and code coverage.
Suitable for unit and structural testing.
Techniques: Statement coverage, Branch coverage, Path coverage.

Functional Testing

Tests *what* the system does.

Ensures each function works as per requirements.

Based on user requirements and use cases.

Examples: Unit testing, Integration testing, System testing, User acceptance testing.

Non-Functional Testing

Tests *how well* the system performs.

Evaluates quality attributes such as speed, security, reliability.

Examples: Performance testing, Load testing, Usability testing, Security testing.

Manual Testing

Test cases executed by a human tester.

Useful for usability, exploratory, and ad-hoc testing.

No programming skills required.

Slower and more prone to human errors.

Automation Testing

Automated tools/scripts execute test cases.

Suitable for regression, performance, and repetitive tests.

Requires programming/scripting skills.

Faster execution, high accuracy, and reusable test scripts.

Model for Testing

1. Requirements Phase

Testing starts by reviewing requirements.

Aim: Identify missing, incorrect, or ambiguous requirements.

Output: Test basis and initial test scenarios.

2. System Design Phase

Testers review architectural and design documents.

Identify design-level errors before coding begins.

Prepare high-level test cases and plan test strategy.

3. Coding Phase
Developers write code and testers prepare detailed test cases.
Unit tests designed to validate logic at the smallest component level.
4. Testing Levels (Layered Testing)
The model defines four main test levels:
a) Unit Testing
Tests individual components or functions.
Performed by developers.
Ensures each module works correctly.

b) Integration Testing

Tests how modules interact with each other.

Detects interface and data flow issues.

c) System Testing

Tests the complete, integrated system.

Ensures the system meets all functional and non-functional requirements.

d) Acceptance Testing

Performed by end users or clients.

Validates that the system is ready for deployment and fulfills business needs.

5. Test Planning and Execution

Planning includes scope, strategy, tools, schedule, and resources.

Execution includes running test cases, reporting defects, and retesting.

6. Feedback and Iteration

Defects detected during testing are fixed, and testing is repeated.

Ensures continuous improvement.

Consequences of Bugs

Software bugs can cause a wide range of negative effects depending on the severity, location, and timing of the defect. These consequences can impact users, developers, organizations, and even safety-critical systems.

1. System Failures

Bugs may cause crashes, hangs, incorrect outputs, or unexpected behavior.

Critical failures can stop business operations.

2. Security Vulnerabilities

Bugs may expose the system to hacking, data breaches, malware attacks, or unauthorized access.

Can lead to loss of sensitive information.

3. Financial Loss

Fixing bugs after deployment is expensive.

Organizations may lose revenue through downtime, incorrect transactions, or refunds.

Extra cost for maintenance and patching.

4. Reduced Reliability and Quality

Frequent bugs make the system unreliable.

Users lose trust and confidence in the product.

5. Poor User Experience

Bugs can cause slow performance, unexpected errors, or incorrect results.

Leads to user frustration and dissatisfaction.

6. Reputation Damage

Continuous failures harm the company’s image and market presence.

Loss of customers and reduced competitive advantage.

7. Legal and Compliance Issues

Bugs in critical systems (healthcare, finance, aviation) may violate regulations.

Can result in legal actions, penalties, or lawsuits.

8. Safety Risks

In safety-critical systems (medical devices, automobiles, aircraft, nuclear plants), bugs may cause physical harm or even loss of life.

9. Project Delays

Time spent identifying, fixing, and retesting bugs increases project duration.

May cause missed deadlines and cost overruns.

Taxonomy of Bugs

A **taxonomy of bugs** is a systematic classification of software defects based on their nature, origin, or impact. Categorizing bugs helps in understanding their causes, preventing future defects, and improving the overall software quality.

1. Logical Bugs

Occur due to incorrect logic or algorithms.

Produce wrong outputs or faulty behavior.

Example: wrong formula, incorrect condition.

2. Syntax Bugs

Errors in code syntax that violate programming language rules.

Detected by compilers/interpreters.

Example: missing semicolon, misspelled keywords.

3. Runtime Bugs

Occur during execution of the program.

Often related to memory, null references, or invalid operations.

Example: division by zero, null pointer exception.

4. Calculation / Data Bugs

Incorrect data handling, processing, or arithmetic errors.

Example: rounding errors, data type mismatches.

5. Interface Bugs

Issues in interactions between modules or external systems.

Example: wrong API usage, incompatible data formats.

6. Compatibility Bugs

Appear when software behaves differently on various platforms.

Example: browser compatibility issues, OS-specific errors.

7. Performance Bugs

Software fails to meet required speed, response time, or efficiency.

Example: slow loading pages, memory leaks.

8. Security Bugs

Vulnerabilities that expose the system to attacks.

Example: SQL injection, insecure authentication.

9. Usability Bugs

Issues that affect user experience or UI design.

Example: unclear navigation, mismatched button actions.

10. Integration Bugs

Occur when combining modules, services, or components.

11. Documentation Bugs

Errors in user manuals, help guides, or instructions.

Example: wrong steps or missing information.

12. Boundary/Edge Case Bugs

Occur at limits of input ranges or unexpected inputs.

Example: handling 0, negative values, overflow.

Flow Graphs & Path Testing

1. Flow Graphs (Control Flow Graph – CFG)

A **flow graph** is a graphical representation of the control structure of a program. It shows how the control flows from one statement to another using **nodes** and **edges**.

Key Elements

Node (Circle): Represents a statement or block of statements.

Edge (Arrow): Represents the flow of control from one node to another.

Decision Node: A branching point (e.g., IF, WHILE).

Entry/Exit Nodes: Starting and ending points of the program.

Purpose of Flow Graphs

Helps understand program logic.

Identifies independent paths for testing.

Used to calculate **cyclomatic complexity**.

Forms the foundation for **path testing**.

2. Path Testing

Path testing is a **white-box testing technique** that ensures all possible execution paths in a program are executed at least once.

Objective

To detect logic errors by executing all **independent paths** derived from the flow graph.

Key Terms

a) Path

A sequence of nodes/edges from start to end.

b) Independent Path

A path that introduces at least **one new edge** not included in any other path.

c) Cyclomatic Complexity (McCabe's Metric)

A measure of the program's logical complexity:

Cyclomatic Complexity $(V) = E - N + 2$

E = number of edges

N = number of nodes

It indicates the **minimum number of test cases** required for complete path coverage.

Basic Concepts of Path Testing

1. Control Flow Graph (CFG)

A graphical representation of the program’s control flow.

Consists of **nodes** (statements/blocks) and **edges** (control flow).

Basis for identifying paths.

2. Path

A sequence of nodes and edges from the **entry** to the **exit** of the program.
Represents one possible route of execution.

3. Independent Path

A path that introduces **at least one new edge** not included in any previously identified path.
Ensures maximum coverage with minimum test cases.

Predicates & Path Predicates

1. Predicates

A **predicate** is a logical condition that evaluates to either **true** or **false**.

In programming and testing, predicates are typically **Boolean expressions** used in **decision statements** like if, while, or for.

Purpose in testing: Predicates help determine **which path a program will take** in its execution.

2. Path Predicates

A **path predicate** is the **logical AND of all predicates along a specific path** in a program’s control flow graph (CFG).

It represents **conditions that must be true for execution to follow a particular path**.

If the path has multiple decisions, the path predicate is formed by combining the **predicates at each decision point** along the path using AND (∧) and OR (V) as required by the path's flow.

Achievable Paths

1. Definition of Achievable Paths

An **achievable path** (also called a **feasible path**) in a program's **control flow graph (CFG)** is:

A path through the program that can actually be executed for some input values.

Not all paths in a CFG are achievable because some may contain **contradictory conditions**.

Achievable paths are important because **testing should focus on them**; impossible paths cannot be executed and thus do not need test cases.

How to Identify Achievable Paths

Draw the Control Flow Graph (CFG) of the program.

List all possible paths from start to end.

Formulate the path predicate for each path.

Check the satisfiability of each path predicate:

- If there exists at least one input that makes the path predicate true → path is achievable.
- If no input can satisfy it → path is **unachievable**.

Path Sensitizing

1. Definition of Path Sensitizing

Path Sensitizing is:

The process of finding **input values** that will cause the program to execute a **specific path** in the control flow graph (CFG).

In other words, **it “activates” a particular path** by satisfying all the predicates along that path.

This is a key step in **path testing**, because knowing the path is not enough; you need **actual test inputs** to traverse it.

2. How Path Sensitizing Works

Steps:

Select a path in the CFG from start to end.

Formulate the path predicate: combine all decision predicates along the path (use AND/OR/NOT depending on branches).

Solve the path predicate to find input values that make it true.

Use these inputs as test cases to execute the path.

Path Instrumentation

1. Definition of Path Instrumentation

Path Instrumentation is:

The technique of **modifying a program to record which paths are executed** during testing.

Essentially, it allows testers to **track path coverage**.

Used in **path testing** to verify that test cases actually traverse the intended paths.

Helps in collecting **runtime information** about program execution.

2. How Path Instrumentation Works

Identify paths or decisions in the program.

Insert instrumentation code at strategic points (e.g., before or after branches or statements) to record:

- Which **decision was taken**
- Which **path was followed**

Run the program with test inputs.

Application of Path Testing

1. Definition Recap

Path Testing is a **white-box testing technique** where:

Test cases are designed to execute **all possible paths** (or a representative set of paths) in a program’s **control flow graph (CFG)**.Focuses on **control structures** like loops, conditions, and branches.

2. Applications of Path Testing

A. Detecting Logic Errors

Ensures that **all decision outcomes** are tested.

Helps find errors in **conditional statements** like if, switch, or while.

Example: Missing else handling or wrong logical operators.

B. Verifying Complex Conditional Statements

Programs with multiple nested if or switch statements can have many execution paths.

Path testing ensures **each possible combination of conditions** is tested.

Example: if ((x>0 && y<5) || z==10)

UNIT-II
Transaction Flow Testing

A **transaction flow** represents the **ordered sequence of operations** performed during the life cycle of a database transaction—from initiation to completion. It is used to understand, analyze, and test the logical behavior of a transaction in a system.

Key Steps in a Transaction Flow

Transaction Start

The user or application initiates a transaction using commands like BEGIN TRANSACTION or automatically through system operations.

Read/Write Operations (Data Access)

The transaction performs a series of database operations:

- **Read** operations (R(x))
- **Write** operations (W(x))

These steps represent the core work of the transaction.

Processing / Computation

The system performs logical computations or business logic using the data read from the database.

Validation / Checking Constraints

Ensures data integrity rules, constraints, and conditions are satisfied.

May involve concurrency control checks (locks, timestamp validation, etc.).

Commit or Rollback Decision

If all operations succeed → **Commit**

If any failure occurs → **Rollback** (undoes the changes)

Transaction End

After commit/rollback, the transaction is considered completed, and resources (locks, buffers) are released.

Used to analyze business processes & logic flow.

Transaction Flows represent the step-by-step sequence of operations involved in completing a business transaction. They are used to **analyze business processes**, understand **logic flow**, and ensure that every operation follows the correct order.

Purpose / Why We Use Transaction Flows

To visualize the **logical sequence** of steps in a transaction

To understand and improve **business processes**

To identify **bottlenecks**, redundancies, or missing steps

To ensure the system behaves correctly under different conditions

Useful for **testing, debugging, and validating** transaction behavior

Helps detect missing, incorrect, or redundant steps.

Data cleaning is a crucial process that helps detect missing, incorrect, or redundant steps within a dataset. It involves identifying and correcting errors, inconsistencies, and inaccuracies to ensure the data is accurate, consistent, and reliable for analysis and decision-making. Here are some key techniques used in data cleaning:

Handling Missing Data: Strategies include removing records with missing values, imputing values, or using algorithms to predict and fill in missing values.

Removing Duplicates: Ensures each data point is unique and accurately represented, preventing skewing analyses and leading to inaccurate results.

Correcting Inaccuracies: Identifies and corrects data entry errors, such as typos or incorrect values, ensuring data accuracy.

Standardizing Formats: Ensures data is entered in a consistent manner, which is essential for accurate analysis.

Transaction Flow Testing Techniques

Transaction Flow Testing Techniques

Transaction Flow Testing focuses on validating the **sequence of operations** in a transaction to ensure correctness, completeness, and reliability. The following techniques are commonly used:

1. Transaction Flow Graph (TFG) Analysis

Represents the transaction as a **graph** with nodes (steps/events) and edges (flow transitions).

Used to understand the logic flow and identify potential issues.

2. Path Testing

Derives all possible **paths** through the transaction flow.

Ensures every path is executed at least once.

Helps uncover logical errors, missing conditions, and incorrect steps.

3. Use Case–Based Testing

Test cases derived from **real business scenarios** and user interactions.

Ensures the system supports intended business processes correctly.

4. Scenario Testing

Focuses on **end-to-end scenarios**, combining multiple transactions.

Helps validate business workflows and their interdependencies.

5. State Transition Testing

Used when the transaction changes system **states** (e.g., order status, payment status).

Checks valid and invalid state transitions.

Dataflow Testing – Basics

Dataflow Testing is a **white-box testing technique** that focuses on how **data is defined, used, and moved** (flows) through a program.

It helps detect problems related to variable usage such as **uninitialized variables**, **unused variables**, or **incorrect data manipulation**.

Key Concepts

1. Definition (DEF)

A statement where a variable is **assigned a value**.

Example: x = 10;

2. Use (USE)

A statement where a variable’s **value is used**.

Two types:

Computation Use (c-use) – used in calculations

Example: y = x + 2;

Predicate Use (p-use) – used in decision conditions

Example: if (x > 5)

3. Definition-Use (DU) Chain

A path between a variable’s **definition** and its **use**, without being redefined in between.

Used to verify correct data flow.

What Dataflow Testing Detects

Variables used **before initialization**

Variables **defined but never used**

Variables redefined **without being used**

Incorrect or missing definitions

Data anomalies due to improper data flow

Strategies in Dataflow Testing

Strategies in Dataflow Testing

Dataflow testing strategies determine how thoroughly the **definition–use (DU) chains** of variables in a program must be exercised. The common strategies are:

1. All-Defs Strategy

Ensures that **every definition (DEF)** of every variable is tested at least once.

For each variable definition, test at least one path that leads to any **use (USE)** of that variable.

Goal: Verify that all variable definitions are reachable and correct.

2. All-Uses Strategy

Ensures that **every definition-use (DU) pair** for each variable is tested.

Covers both:

- **c-use** (computational use)
- **p-use** (predicate use)

Goal: Validate that every definition is used meaningfully in both computations and decisions.

3. All-DU-Paths Strategy

The most thorough and exhaustive strategy.

Ensures that **all possible paths** from each definition to each use are executed, without redefining the variable.

Goal: Detect subtle data anomalies caused by:

- Loops
- Unreachable paths
- Redefinitions
- Missing uses

Application of Dataflow Testing

Dataflow Testing is applied to examine how data **moves, changes, and is used** throughout a program. It is especially useful for identifying errors that are not easily detected through control-flow testing alone.

1. Detecting Data Anomalies

Dataflow testing helps detect:

Undefined variable use (using a variable before it is assigned)

Unreachable definitions (variables defined but never used)

Redundant definitions (redefined before use)

Improper variable updates

These issues lead to logical errors and unexpected outputs.

2. Validating Correct Data Usage

It ensures:

Every variable definition flows correctly to its use

No variable is used in decision-making or computation without proper value

No variable remains unused unnecessarily

This improves the **semantic correctness** of the program.

3. Improving Reliability of Complex Code

Ideal for:

Loops

Branching code

Nested conditions

Modules with multiple variable interactions

Dataflow testing ensures hidden data-related bugs are caught early.

4. Enhancing Code Maintainability

By identifying redundant or unused variables and definitions, it helps:

Clean up dead code

Reduce confusion

Simplify future modifications

This results in cleaner and more maintainable code.

Domain Testing – Domains & Paths

1. Domains in Domain Testing

A **domain** is a set of input values for which the program behaves similarly or produces similar outcomes.

What are Domains?

Input values are grouped into **regions** (domains) where the program logic behaves the same.

Each domain is separated by **boundaries**.

Testing ensures the program handles:

- **Valid domains**
- **Invalid domains**
- **Boundary values**

Purpose of Domains

To check correctness of the program for all input regions.

To detect:

- Missing conditions
- Wrong boundary conditions
- Incorrect decision logic
- Domain-specific logical errors

2. Paths in Domain Testing

Paths represent the different **execution routes** the program takes depending on the domain (input region).

What are Paths?

When input moves from one domain to another, the program may follow a **different control-flow path**.

Domain testing ensures each path is tested for:

- Correct execution
- Proper boundary handling
- No missing/incorrect logic

Types of Paths

Interior Paths

Executed within a single input domain.

Example: Only valid inputs (e.g., 10, 20, 50).

Boundary Paths

Paths executed when inputs lie **on or near domain boundaries**.

Example: Inputs at 0, 1, 99, 100.

Cross-Domain Paths

Occur when input transitions from one domain to another.

Helps detect incorrect branching or transitions.

Nice & Ugly Domains

In **Domain Testing**, input values are divided into regions (domains). The structure of these domains affects how easy or difficult the testing becomes. Based on this, domains are categorized as **Nice Domains** and **Ugly Domains**.

1. Nice Domains

Definition

A **Nice Domain** is a domain that is:

Continuous

Convex (no holes or gaps)

Well-defined with simple, straight boundaries

Easy to analyze and test

Characteristics of Nice Domains

Boundaries are simple (linear conditions such as <, >, <=, >=)

No missing regions

Input points inside the domain behave similarly

Testers can predict boundaries easily

Fewer errors and easier test case generation

2. Ugly Domains

Definition

An **Ugly Domain** is a domain that is:

Discontinuous

Has **gaps, holes, or multiple sub-domains**

Has **complex, irregular boundaries**

Hard to analyze and test

Characteristics of Ugly Domains

Boundaries may be **non-linear**, complex, or combined with multiple conditions

Domain may be split into several disjoint segments

Hard to identify valid and invalid regions

Increased chance of:

- Missing paths
- Incorrect conditions
- Boundary-related defects

Domain & Interface Testing

1. Domain Testing

Definition

Domain Testing focuses on testing the **input domain** of a program—i.e., **all possible input ranges, boundaries, and conditions**.

It validates how the program behaves for **various input regions (domains)**.

Purpose

To detect **boundary errors**, **missing conditions**, and **invalid domain handling**

To ensure proper classification of inputs into valid and invalid categories

2. Interface Testing

Definition

Interface Testing checks how different **modules, components, or systems interact** with each other. It ensures that **data is passed correctly**, and communication between components is error-free.

Purpose

To ensure correct integration between modules

To validate data flow, parameter passing, and message formats

To detect mismatch errors between connected components

UNIT-III

Paths, Path Products & Regular Expressions

1. Paths

Definition: A **path** is a sequence of nodes and edges followed during the execution of a program from **entry to exit** in a control flow graph (CFG).

Types of Paths

Simple Path: No repeated nodes (except possibly start/end in loops)

Independent Path: Adds at least one new edge not included in previously identified paths

Feasible Path: Can actually be executed with some input

Infeasible Path: Cannot be executed due to logical constraints (e.g., contradictory conditions)

2. Path Products

Definition

A **path product** is a method of representing paths compactly by multiplying the possible choices at each decision point.

Purpose of Path Products

Summarize multiple paths

Identify combinations of branches

Help in calculating the number of possible paths

Useful for deriving test cases

Path Products & Path Expressions

1. Path Products

Definition

A **Path Product** is an algebraic representation of all possible paths in a program.

It is formed by multiplying (combining) the choices available at each decision point.

Key Idea

At every decision node, outgoing branches represent **options**.

These options are written inside parentheses with + (choice/OR).

2. Path Expressions

Definition

A **Path Expression** is a regular-expression-like notation used to describe **the structure of paths** in a control-flow graph, including loops and repeated sequences.

Symbols Used

Concatenation (sequence): AB → A followed by B

Choice/Union: A + B → either A or B

Kleene Star *: A* → A repeated zero or more times

Parentheses for grouping

Purpose

Represent sets of paths compactly

Handle loops gracefully

Avoid writing infinite paths explicitly

Reduction Procedure

The **Reduction Procedure** is a systematic method used in **path testing** to convert a *control flow graph* into a **single regular expression** (called a *path expression*) that represents **all possible execution paths** in the program.

It works by **eliminating intermediate nodes** of the graph one by one and replacing them with equivalent path products.

Why Reduction Procedure?

To convert a **graph → mathematical expression**

To compute **all possible paths** compactly

To help generate **test paths** for white-box testing

To simplify complex graphs by eliminating nodes

Reduction Steps (Simple Guide)

1. Start with full flow graph

Nodes typically:

Entry → A → B → C → Exit

2. Remove one node at a time

When removing node **B**, compute equivalent expressions for paths going:

Incoming → B → Outgoing

3. Continue reducing

Until only:

RENTRY,EXIT

4. Final result

A **path expression**, which represents *all feasible paths* from Entry to Exit.

Applications of Path Expressions

Path Expressions are algebraic expressions that represent **all possible execution paths** in a program. They are derived from **control flow graphs** using **path products, regular expressions, and reduction procedures**.
They are widely used in **white-box testing** to analyze and generate test paths.

1. Test Case Generation

Path expressions compactly represent all feasible paths.

From the final expression, testers can extract:

Independent paths

Loop handling paths (0, 1, many iterations)

Decision-based test paths

This helps in **designing effective and minimal test cases**.

2. Identification of Feasible and Infeasible Paths

Path expressions show the exact structure of paths.

Thus they help in:

Detecting **dead code** (path not present)

Identifying **infeasible paths**

Finding **redundant paths**

This increases the **testability** of the program.

Regular Expressions & Flow Anomaly Detection

1. Regular Expressions in Path Testing

Regular Expressions are used to represent the **complete behavior** of a program's control flow.

Example

If the flow graph has:

Path A followed by B, then either C or D

The regular expression is:

A B (C + D)

Where:

Concatenation → sequence of operations

Union (+) → selection / branching

Closure (*) → loops (0 or more times)

2. Flow Anomaly Detection

Flow anomalies occur when variables are used in an unexpected or illegal order.

In Dataflow testing, we focus on operations on variables:

d → define

u → use

k → kill (or un-define)

A flow anomaly is detected when the sequence of actions **violates normal data usage rules**.

Decision Tables

Decision Tables are a structured way to represent **complex decision logic** in a tabular form. They show **conditions, actions, and rules** in a compact and systematic manner.

They are widely used in **software testing** and **business rules analysis**.

1. Components of a Decision Table

Conditions – Boolean expressions or criteria to evaluate (e.g., Age > 18)

Condition Alternatives – Possible values for each condition (e.g., True / False)

Actions – Operations to be performed based on conditions (e.g., Approve, Reject)

Rules – Combination of condition values mapping to specific actions

Types of Decision Tables

Limited Entry Table

Conditions have only **True/False** values.

Example: Yes/No, On/Off.

Extended Entry Table

Conditions can take **multiple values** beyond True/False.

Example: High/Medium/Low

Path Expressions in Logic Testing

1. Purpose of Path Expressions in Logic Testing

Represent **all possible logical paths** in a program

Identify **independent and dependent paths**

Generate **test cases** to cover all logical scenarios

Detect **missing or redundant conditions**

Ensure thorough **decision/branch coverage**

2. Representation

Symbols Used

Concatenation (AB) → Execute A then B

Union / Choice (A + B) → Either A or B

Kleene Star (A*) → Repeat A zero or more times

Parentheses () → Grouping of paths

KV Charts (Karnaugh–Veitch Charts)

Karnaugh-Veitch (KV) Charts, also known as **Karnaugh Maps (K-Maps)**, are **graphical tools** used to simplify **Boolean expressions** and **logic functions**. They provide a visual method to minimize logic circuits and reduce the number of gates.

1. Purpose of KV Charts

Simplify **Boolean expressions**

Identify **redundant terms**

Reduce **logic circuit complexity**

Help in **designing efficient combinational circuits**

2. Structure of a KV Chart

Grid Format: Each cell represents a unique combination of input variables.

Rows & Columns: Represent input variable combinations in **Gray code** order (only one bit changes between adjacent cells).

Cell Values: Typically 1 (true) or 0 (false) for the function output

3. Rules for Using KV Charts

Group 1s (or 0s for POS) in **rectangular blocks**

Size must be a **power of 2** (1, 2, 4, 8, …)

Groups can wrap around edges

Minimize the number of terms in the expression

Include each 1 in at least one group

Choose the largest possible groups to maximize simplification

4. Advantages of KV Charts

Quick **visual simplification** of logic expressions

Minimizes **errors** compared to algebraic simplification

Helps in **designing optimal circuits**

Effective for **up to 4–6 variables**

Specifications in Logic-Based Testing

Logic-Based Testing is a **white-box testing technique** that derives test cases from the **logical conditions and decision structures** in a program. The **specifications** define **what to test, how to test, and expected outcomes** based on the program's logic.

1. Definition of Specifications

Specifications in logic-based testing refer to the **formal description of program logic** including:

Conditions (e.g., if x > 0)

Decisions/Branches (e.g., if-else, switch)

Logical combinations (AND, OR, NOT)

Loops and iterations (for, while)

These specifications are used to **design test cases** that cover all possible logical scenarios.

2. Components of Specifications

Input Conditions – Boolean expressions that determine flow

Actions – Operations performed when conditions are true/false

Decision Rules – Mapping from conditions to actions

Expected Outcomes – Result of executing each logical path

3. Purpose of Specifications in Logic Testing

Ensure **all decisions are tested**

Cover all **true/false branches**

Detect **missing, incorrect, or redundant conditions**

Provide a **basis for systematic test-case design**

4. Types of Specifications Used

Decision Tables – Map conditions to actions for all rules

Predicate Logic Expressions – Formalize conditions (e.g., X > 0 AND Y < 10)

Control Flow Graphs (CFGs) – Represent sequences of logical decisions

Boolean Expressions – Identify independent paths and combinations

UNIT-IV
State and State Graph

State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.

Finite state machines can also be implemented as table-driven software, in which case they are a powerful design option.

A state is defined as: “A combination of circumstances or attributes belonging for the time being to a person or thing.”

For example, a moving automobile whose engine is running can have the following states with respect to its transmission.

Reverse gear

Neutral gear

First gear

Second gear

Third gear

Fourth gear State graph -

For example, a program that detects the character sequence “ZCZC” can be in the following states.

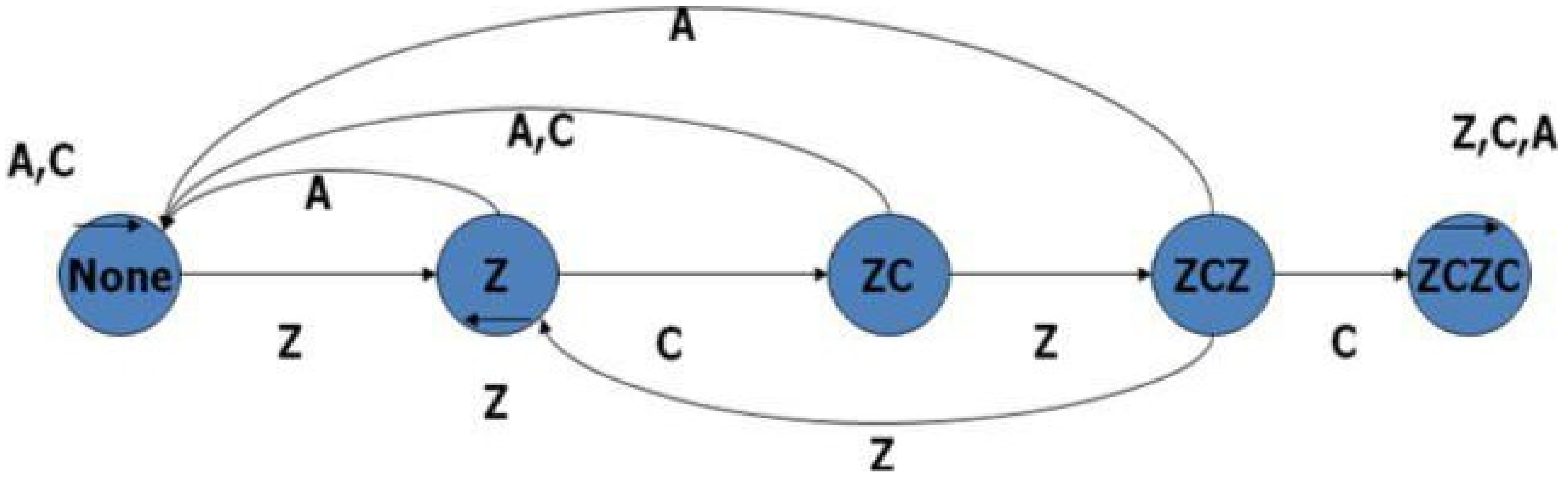
Neither ZCZC nor any part of it has been detected.

Z has been detected.

ZC has been detected.

ZCZ has been detected.

ZCZC has been detected.


A
A,C
A,C
A
Z,C,A
None
Z
ZC
ZCZ
ZCZC
Z
C
Z
C
Z
Z

Good State Graphs and Bad

What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context.

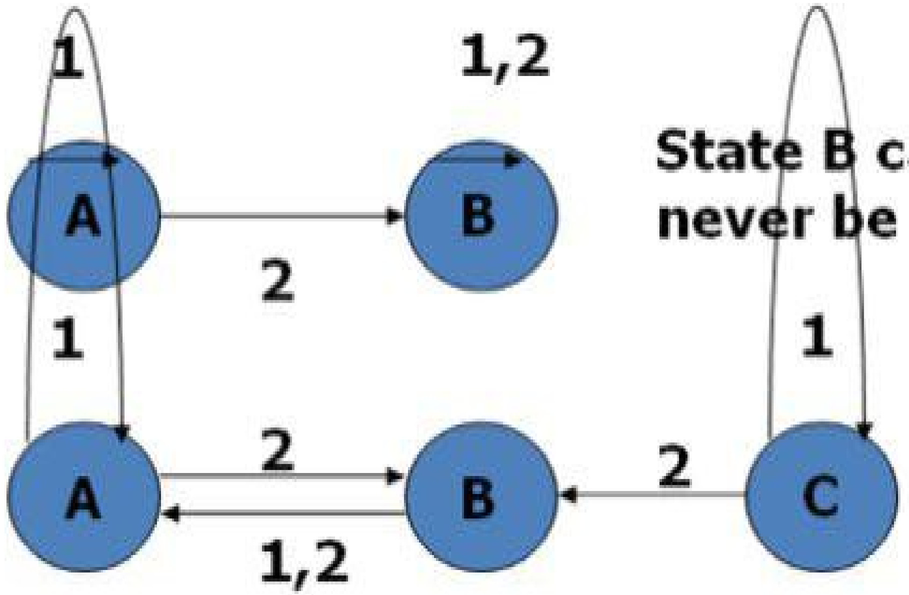
Here are some principles for judging.

The total number of states is equal to the product of the possibilities of factors that make up the state.

For every state and input there is exactly one transition specified to exactly one, possibly the same, state.

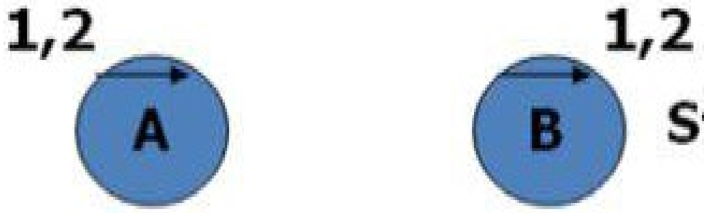
For every transition there is one output action specified. The output could be trivial, but at least one output does something sensible.

For every state there is a sequence of inputs that will drive the system back to the same state.

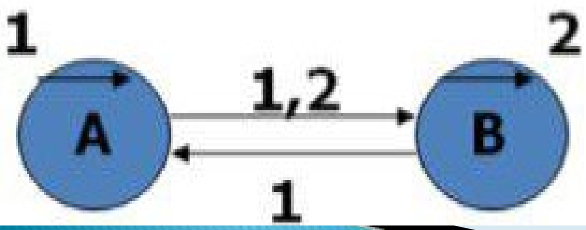


State B can never be left, the initial state can never be entered again.

State C cannot be entered.



States A and B are not reachable



Two transitions are specified for an input of 1 in state A

State Testing

1. Definition of State Testing

State Testing (also called **State-based Testing**) is a **white-box/black-box testing technique** where the **behavior of the system is tested based on its states and transitions**. The goal is to ensure that the system behaves correctly when moving from one state to another due to events or inputs.

It is particularly useful for systems where **outputs depend not only on the current input but also on the current state** (e.g., vending machines, login systems, communication protocols).

2. Key Concepts

State: A condition of the system at a particular point in time.

Event/Trigger: An input or action that causes a state change.

Transition: Movement from one state to another caused by an event.

Initial State: The state of the system before any event occurs.

Final State: The state where the system completes or terminates.

State Graph / State Transition Diagram: Visual representation of all states and transitions.

3. Steps in State Testing

Identify all states: List all possible states of the system.

Identify events or inputs: Determine what triggers cause state changes.

Draw state graph/diagram: Represent all states as nodes and transitions as arrows labeled with events.

Define test cases:

- **State coverage:** Test each state at least once.
- **Transition coverage:** Test each transition between states.
- **Event coverage:** Ensure all events causing transitions are tested.

Execute tests: Simulate events and check whether the system moves to the correct states.

Testability Tips

1. What is Testability?

Testability is the degree to which a software system or component **facilitates testing** to ensure it behaves as expected. High testability allows defects to be detected quickly and reliably.

2. General Testability Tips

A. Design-Level Tips

Modular Design:

Break the system into independent modules or components.

Easier to isolate and test each part separately.

Low Coupling & High Cohesion:

Modules should depend minimally on others (low coupling).

Each module should focus on a single task (high cohesion).

Clear State Transitions:

Ensure the system states are well-defined.

Avoid ambiguous or hidden state changes.

Use Standard Interfaces:

Standard APIs or interfaces make it easier to test and mock components.

Logging & Monitoring:

Add logs for important actions and state changes.

Helps trace errors during testing.

UNIT–V

Matrix of Graph

The Matrix of a Graph

A graph matrix is a square array with one row and one column for every node in the graph.

Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.

The relation for example, could be as simple as the link name, if there is a link between the nodes.

Some of the things to be observed:

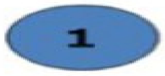
The size of the matrix equals the number of nodes.

There is a place to put every possible direct connection or link between any and any other node. The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.

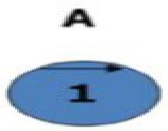
A connection from node i to j does not imply a connection from node j to node i.

If there are several links between two nodes, then the entry is a sum; the “+” sign denotes parallel links as usual.

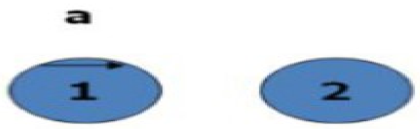
Some Graphs and their Matrices



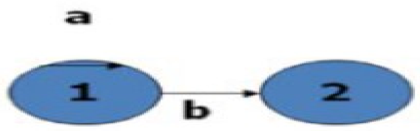
0



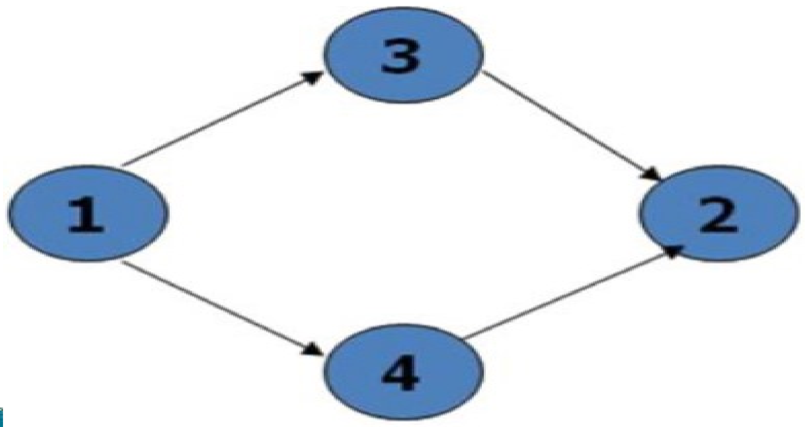
A



a	



a	b



		a	c
	b		
	d		

Relations

A relation is a property that exists between two objects of interest.

For example,

“Node a is connected to node b” or aRb where “R” means “is connected to”.

“a>=b” or aRb where “R” means greater than or equal”.

A graph consists of set of abstract objects called nodes and a relation R between the nodes.

If aRb, which is to say that a has the relation R to b, it is denoted by a link from a to b.

For some relations we can associate properties called as link weights.

Transitive Relations

A relation is transitive if aRb and bRc implies aRc.

Most relations used in testing are transitive.

Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of.

Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

Reflexive Relations

A relation R is reflexive if, for every a, aRa.

A reflexive relation is equivalent to a self loop at every node.

Examples of reflexive relations include: equals, is acquainted with, is a relative of.
Examples of irreflexive relations include: not equals, is a friend of, is on top of, is under.

Symmetric Relations

A relation R is symmetric if for every a and b, aRb implies bRa.

A symmetric relation mean that if there is a link from a to b then there is also a link from b to a.
A graph whose relations are not symmetric are called directed graph.

A graph over a symmetric relation is called an undirected graph.
The matrix of an undirected graph is symmetric (aij=aji) for all i,j)

Antisymmetric Relations

A relation R is antisymmetric if for every a and b, if aRb and bRa, then a=b, or they are the same elements.

Examples of antisymmetric relations: is greater than or equal to, is a subset of, time.

Examples of nonantisymmetric relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of

Equivalence Relations

An equivalence relation is a relation that satisfies the reflexive, transitive, and symmetric properties.

Equality is the most familiar example of an equivalence relation.

If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation.

The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class.

The idea behind partition testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes.

Testing any member of the equivalence class is as effective as testing them all.

The Powers of a Matrix

Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry. Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive. The square of the matrix represents all path segments two links long.

The third power represents all path segments three links long.

Node Reduction Algorithm (General)

The matrix powers usually tell us more than we want to know about most graphs.
In the context of testing, we usually interested in establishing a relation between two nodes-typically the entry and exit nodes.
In a debugging context it is unlikely that we would want to know the path expression between every node and every other node.
The advantage of matrix reduction method is that it is more methodical than the graphical method called as node by node removal algorithm.
Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.

NRCM
your roots to success...

THANK YOU