


Here is a **complete and exam-focused explanation** of **Unit 2: Flow Graphs and Path Testing** from **Software Testing**, including **definitions, diagrams, examples, and key points for revision**.

Unit 2 – Flow Graphs and Path Testing

1. What is Path Testing?

Path Testing is a **white-box testing technique** in which test cases are designed to execute **all possible paths** in the program's **control flow graph (CFG)**.

 **Goal:** To **identify and test all independent paths** to ensure each statement and condition in the program is executed at least once.

2. What is a Flow Graph (Control Flow Graph - CFG)?

A **flow graph** is a graphical representation of all the **paths** that might be traversed through a program during its execution.

3. Flow Graph Elements

Term	Description
Node	Represents a statement or a group of statements in the program
Edge	Represents the flow of control (like arrows from one node to another)
Region	An area enclosed by edges and nodes; used to find independent paths

Entry/Exit Start and end points of the flow

4. Flow Graph Example

```
1: if (x > 0)
2:   x = x + 1;
3: else
4:   x = x - 1;
5: print(x);
```

Corresponding Flow Graph:

[1]

/ \

[2] [4]

\ /

[5]

- Nodes: 1, 2, 4, 5
- Paths:
 - $1 \rightarrow 2 \rightarrow 5$
 - $1 \rightarrow 4 \rightarrow 5$

5. Independent Path

An **independent path** is any path through the program that introduces **at least one new edge** not included in other paths.



6. Cyclomatic Complexity

A metric to calculate the **number of independent paths** in a program.

Formula:

$$V(G) = E - N + 2P$$

Where:

- $V(G)$ = Cyclomatic Complexity
 - E = Number of edges
 - N = Number of nodes
 - P = Number of connected components (usually 1)
-



Example Calculation

From the graph above:

- Nodes (N) = 4
- Edges (E) = 5
- $P = 1$

$$V(G) = E - N + 2P$$

$$= 5 - 4 + 2(1)$$

$$= 3$$

So, there are **3 independent paths** to be tested.

7. Test Case Design in Path Testing

Step	Description
1. Draw flow graph	Convert code to flow graph
2. Calculate Cyclomatic Complexity	To find number of paths
3. Identify independent paths	Use decision points
4. Design test cases	Ensure each path is tested

8. Advantages of Path Testing

- Ensures **maximum coverage** of code
 - Detects **logic errors and unreachable code**
 - Helps identify **infinite loops or dead code**
-

9. Limitations

- Complex for large programs
 - Difficult when code has **many loops or conditions**
 - Doesn't test actual **output correctness** — only path coverage
-



Summary Table

Concept	Meaning
Path Testing	White-box testing of all independent paths
Flow Graph	Graph showing control flow in the program
Cyclomatic Complexity	Metric to determine number of independent paths
Independent Path	A unique control path with at least one new edge



Sample Exam Answer (3-5 Marks)

Q: Define Flow Graph and explain Path Testing with an example.

Ans:

A **flow graph** is a diagram that shows the possible paths of control in a program.

Path testing is a white-box technique that ensures each independent path through the code is tested at least once.

For example, in the following code:

```
if (x > 0) x++; else x--;
```

```
print(x);
```

The paths are **if-true** → **print**, and **if-false** → **print**. These two paths should be tested with inputs like **x = 1** and **x = -1** to ensure both branches are covered.

Let me know if you'd like to continue with another topic like:

- Unit 2: Predicate Coverage
- Unit 2: Graph Matrices
- or any other unit or revision sheet.

Here is a complete, exam-ready explanation of **Achievable Paths** from **Unit 2: Software Testing**, with **definition**, **diagram**, **examples**, and **summary**.

Unit 2: Achievable Paths

1. Definition of Achievable Paths

Achievable paths are the **valid execution paths** that can be **actually followed** when running the program with some **input values**.

Not all paths in a control flow graph (CFG) are **logically or practically possible**. **Achievable paths** are the ones that are **feasible and executable**.

2. Why Important in Testing?

- We don't test **impossible or infeasible paths**.
 - Focus is on designing **test cases** for paths that can actually **occur** during program execution.
 - Ensures we are **not wasting effort** on untestable code.
-

3. Example + Diagram

Consider the following code:

1: if (x > 0)

2: if (x % 2 == 0)

```

3:   print("Even");
4: print("Done");

```

▶ Flow Graph

[1]

|

(x>0)

/ \

T F

| |

[2] [4]

|

(x%2==0)

/ \

T F

| |

[3] [4]

▶ Achievable Paths

1. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ($x > 0$ and $x \% 2 == 0$)
2. $1 \rightarrow 2 \rightarrow 4$ ($x > 0$ and $x \% 2 != 0$)
3. $1 \rightarrow 4$ ($x \leq 0$)

All of these are **achievable** by giving proper inputs:

- $x = 4 \rightarrow \text{path 1}$
 - $x = 3 \rightarrow \text{path 2}$
 - $x = -1 \rightarrow \text{path 3}$
-

4. Infeasible Path (Not Achievable)

An **infeasible path** is one that **cannot** be executed **regardless of input** due to logical constraints in the program.

Example: If the code has:

if ($x > 0 \ \&\& \ x < 0$)

This path is **infeasible** because no value of x can be both greater and less than 0.

5. Role in Path Testing

- During **path testing**, we identify **all independent paths**.
 - Then, we **eliminate infeasible paths**.
 - Finally, test cases are designed for **achievable paths only**.
-

6. Summary Table

Term	Meaning
Achievable Path	A path in a flow graph that can be executed with some input

Infeasible Path A path that cannot be executed with any input

Used for Creating valid and effective test cases

Sample Exam Answer (3 Marks)

Q: What are Achievable Paths in software testing? Explain with an example.

Ans:

Achievable paths are valid execution paths in a program's flow graph that can be followed by providing suitable input values. For example, in a nested if condition, paths where the conditions can be true or false are achievable if we can input values to satisfy them. Infeasible paths, which cannot be executed, are excluded during path testing to ensure accurate test coverage.

Let me know if you want to continue with:

- **Predicate Coverage**
- **Graph Matrices**
- Or any other topic from Unit 2 or other units.

Here is a **complete exam-ready explanation** of **Application - Transaction Flow Testing Techniques** from **Unit 2: Software Testing**, including **definition, example, diagram, and key points** for quick revision.

Unit 2: Application – Transaction Flow Testing Techniques

1. What is Transaction Flow Testing?

Transaction Flow Testing is a **white-box testing** technique used to **model and test business processes or transactions** in a system by representing them as **transaction flow graphs**.

It focuses on the **logical flow of a transaction** (such as logging in, placing an order, booking a ticket) through a system.

2. Purpose of Transaction Flow Testing

- To **test how transactions move** through the system
- To ensure **all possible transaction paths** are tested
- To detect **missing processes, broken logic, or incorrect sequencing**

3. Key Concepts

Concept	Description
Transaction	A sequence of actions triggered by a user (e.g., login, purchase, etc.)
Transaction Flow Graph	A flowchart-like graph showing the transaction steps and branches
Node	Represents a step in the transaction
Edge	Represents the flow from one step to another

4. Example: Online Order Transaction

Process:

1. Login
2. Search product
3. Add to cart
4. Checkout
5. Payment

Transaction Flow Graph

[Start]



[Login]



[Search Product]



[Add to Cart]



[Checkout]



[Payment]



[End]

5. How to Apply Transaction Flow Testing?

1. **Identify transactions** in the application (e.g., user registration, ordering, etc.)
 2. **Draw a transaction flow graph** to represent steps
 3. **Analyze possible paths** through the graph
 4. **Design test cases** to cover:
 - Normal flow
 - Alternate flow
 - Error/retry flow
-

✅ 6. Example Test Cases for Above Transaction

Test Case	Description	Input Values	Expected Result
TC1	Successful transaction	Valid login, in-stock product	Order placed successfully
TC2	Login failure	Invalid username/password	Error message shown
TC3	Payment failure	Invalid card	Payment declined

🧠 7. Advantages of Transaction Flow Testing

- Useful for **real-world business applications**
- Ensures all **paths of user interaction** are tested
- Detects **logical flaws** in transaction handling

8. Summary Table

Term	Meaning
Transaction Flow Testing	Testing logic flow of complete user transactions
Transaction Flow Graph	Visual model of steps and decisions in a transaction process
Focus	Real-world use cases and flows (login, order, payment, etc.)

Sample Exam Answer (3–5 Marks)

Q: Explain Transaction Flow Testing Techniques with an example.

Ans:

Transaction Flow Testing is a white-box technique that models the logical flow of user transactions through a system. It is especially useful for business applications where user actions follow specific steps. For example, an online shopping transaction includes login → search → add to cart → checkout → payment. Each of these steps is represented as nodes in a transaction flow graph, and test cases are designed to cover normal and alternative paths like failed login or failed payment.

Let me know if you'd like to move on to the next topic from Unit 2 or revise any unit quickly!

Here's your **exam-ready notes** for **Unit 3: Data Flow Testing Strategies** from the **Software Testing** syllabus, including **definition, diagram, examples**, and **key points**.

Unit 3 – Topic: Data Flow Testing Strategies

1. Definition

Data Flow Testing is a **white-box testing technique** that focuses on the **points at which variables receive values (definitions)** and the **points at which these values are used (uses)** in a program.

✅ It helps detect:

- Unused variables
- Incorrect variable usage
- Potential bugs due to uninitialized or overwritten data

2. Basic Concepts

Term	Meaning
Definition (def)	A point where a variable is assigned a value (e.g., <code>x = 10;</code>)
Use (use)	A point where the value of a variable is used (e.g., <code>print(x);</code> or <code>if (x > 0)</code>)
DU Pair	A pair consisting of one definition and one reachable use of a variable
Kill	A re-definition of a variable, destroying its previous value

3. Diagram (Control Flow Graph with Data Flow)

Consider the code:

1: `int x = 0;`

2: `if (a > b)`

3: x = a;

4: else

5: x = b;

6: print(x);

Control Flow Graph (CFG) with defs/uses:

(1) def(x)

|

v

(2) use(a, b)

/ \

v v

(3) def(x) (5) def(x)

\ /

v v

(6) use(x)

- **DU Pairs for x:**

- $(1 \rightarrow 6), (3 \rightarrow 6), (5 \rightarrow 6)$

4. Types of Uses

Type	Description	Example
------	-------------	---------

C-use Computation use (used in expressions) `z = x + y;`

P-use Predicate use (used in decision making) `if (x > 0)`

5. Data Flow Testing Strategies

Strategy	Description
All-Defs	Test paths from definition of a variable to at least one use
All-Uses	Test paths from every definition to all possible uses
All-DU-Paths	Test all possible paths from definition to use without killing the variable
All P-Uses / All C-Uses	Focuses on either all predicate uses or all computation uses

6. Example

```
1: int x;      // def(x)
2: x = 5;      // def(x)
3: if (x > 0)   // p-use(x)
4:  y = x + 1;  // c-use(x)
```

- **All-Defs:** Check that **x**'s value is used in at least one of (3) or (4)
- **All-Uses:** Test both (3) and (4)
- **All-DU-Paths:** Cover all non-killing paths from def → use



7. Benefits of Data Flow Testing

Advantage	Description
Detects logical errors	Like using uninitialized variables
Improves test effectiveness	Beyond simple statement/branch coverage
Locates dead code or useless defs	Code that is written but never used



8. Limitations

- Complex for large programs
- Requires accurate Control Flow Graphs
- Not all DU paths may be feasible or practical



9. Summary Table for Revision

Term	Meaning
def	Variable defined/assigned
use	Variable used (P-use / C-use)
DU Pair	Def and its corresponding Use
Kill	Re-definition before use

Sample Exam Answer (3-5 Marks)

Q: What is Data Flow Testing? Explain with example.

Ans:

Data Flow Testing is a white-box testing method that examines the flow of data through variables in a program. It focuses on the points where variables are **defined**, **used**, and **redefined**.

Example: In the code `x = 5; if (x > 0) print(x);`, `x` is defined and then used. Testing should ensure that the definition of `x` reaches its use without being redefined.

This technique helps detect bugs like using variables before assigning values or having definitions that are never used.


Let me know if you'd like the next Unit 3 topic like **Control Flow Graphs**, **Loop Testing**, or any others!

Here's a **complete, exam-ready explanation** of the topic **Domain Testing** – including **Domains and Paths**, and **Domains and Interface Testing** – from **Unit 3: Software Testing**, with **definitions**, **examples**, **diagrams**, and key points for **quick revision**.

Unit 3 – Topic: Domain Testing

1. What is Domain Testing?

Domain Testing is a **white-box and black-box testing technique** used to test **input values** by dividing them into **valid and invalid domains**, and then selecting **representative values** for each domain.

 **Goal:** To ensure that the system behaves **correctly** for all values in valid domains and **appropriately handles** invalid domains.

2. Key Terms

Term	Description
Domain	A range/set of input values for a variable or function
Boundary	The edge between valid and invalid input domains
Path	A sequence of program statements executed for a given input
Interface	The connection point between two modules or systems

◆ 3. Domains and Paths

 **Definition:**

Domains and Paths refers to using **input domains** to guide the selection of **execution paths** during testing. The input domain determines **which path** in the code will be taken.

🌟 **Example:**

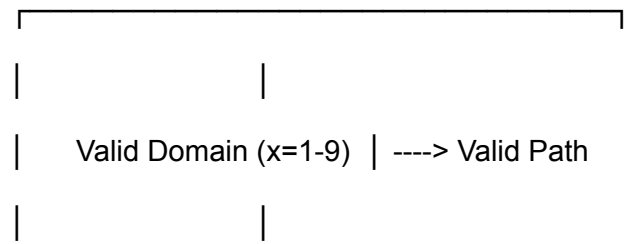
```
if (x > 0 && x < 10)
    print("Valid");
else
    print("Invalid");
```

Input (x)	Domain	Path Taken
5	Valid Domain	Goes to print("Valid")
-1, 10	Invalid Domain	Goes to print("Invalid")

📝 We test **values inside, at the boundaries**, and **outside** the domain to verify **all code paths**.

📊 **Diagram: Domains and Paths**

Input Space



|_____|

↑

↑

$x = 0$

$x = 10$

(boundary) (boundary)

$x < 1$ or $x > 9$ ----> Invalid Path

Testing Approach

- Pick test cases for:
 - Inside the domain
 - On the boundary
 - Outside the domain
- Cover **all execution paths** (Valid & Invalid)

◆ 4. Domains and Interface Testing

Definition:

In **Domain and Interface Testing**, the input domain is used to test **interfaces between software modules** to detect **data loss, miscommunication, or unexpected behavior** when data is passed across boundaries.

Why Important?

- When two systems/modules communicate, **input values must be interpreted correctly** on both sides.
- If one module expects x : $[1-10]$ but another sends $x = 11$, it can lead to bugs or crashes.

✨ Example:

Module A sends value x to Module B

Module A sends	Module B expects	Result
$x = 5$	$1-10$	✓ OK
$x = 0$	$1-10$	✗ Invalid input
$x = 12$	$1-10$	✗ Boundary failure

🖼 Diagram: Domain Interface Testing

[Module A: $1-15$] -----> [Module B: $1-10$]

$x=12$ $x=12$ ✗ Not Accepted

✓ Goals of Interface Domain Testing

- Check **mismatched expectations** between modules
- Ensure that data passed between modules remains:

- **Within expected domain**
- **Formatted correctly**
- **Handled properly on both ends**



Summary Table

Concept	Description
Domain Testing	Divides input into valid/invalid domains
Domains and Paths	Use domains to test all possible code paths
Domains & Interfaces	Use domains to test data passed across modules
Key Focus	Boundaries, input combinations, communication



Sample Exam Answer (3-5 Marks)

Q: Explain Domain Testing with Domains and Paths, and Interface Testing.

Ans:

Domain Testing is a technique where input values are divided into valid and invalid **domains**, and representative values are tested.

In **Domains and Paths**, the domain determines the execution path. For example, if $x \in (1-10)$, the valid path is taken; otherwise, an error path is taken.

In **Domains and Interface Testing**, input domains are used to test the **data passed between modules**. If one module sends values outside the expected domain of the receiving module, it may cause failures.

Let me know if you'd like the next topic from Unit 3 like **Loop Testing**, **Control Flow Graphs**, or anything else!

Great! Let's now cover **Unit 4: Linguistic Testing** with complete **definition, explanation, examples, and key points** to help you prepare for your **Software Testing exam**.

Unit 4 – Topic: Linguistic Testing

1. Definition of Linguistic Testing

Linguistic Testing is a type of software testing that verifies the **language accuracy, grammar, and proper localization** of a software application. It ensures that the application is **linguistically and culturally correct** in the target language(s).

 It focuses on **correctness of the translated content** such as:

- Grammar
 - Spelling
 - Sentence structure
 - Cultural references
 - Date, time, currency formats
-

2. Purpose of Linguistic Testing

- To **validate translated content** in multi-language applications
 - To ensure the **localization is culturally and linguistically appropriate**
 - To provide a **better user experience** for global users
-

3. What Does Linguistic Testing Check?

Aspect	Example
Grammar	Correct use of verbs, tense, structure
Spelling	Regional spellings like color/colour
Punctuation	Proper commas, periods, etc.
Contextual Meaning	Translation matches intended meaning
UI Layout Issues	Text fits in buttons, no truncation
Local Formatting	Date: 16/05/2025 vs 05/16/2025
Currency and Numbers	₹100.00 vs \$100.00

4. Example of Linguistic Testing

Imagine an app translated from English to Hindi:

English Text	Hindi Translation (Incorrect)	Correct Translation
--------------	-------------------------------	---------------------

"Logout"	"लॉगआउट करें" (Correct)	✓
"Settings"	"सेटिंग" (Incorrect, half-translated)	"सेटिंग्स" or "समायोजन"
"Buy Now"	"अब खरीदें" (Correct)	✓

5. Linguistic Testing vs Localization Testing

Feature	Linguistic Testing	Localization Testing
Focus	Grammar, language accuracy	Functional and cultural adaptation
Tester Type	Native language experts	QA/localization engineers
Example Check	"Submit" is translated properly	Date format shows DD/MM/YYYY in India

6. Tools for Linguistic Testing

- Google Translate (for quick check, **not reliable for accuracy**)
- **CAT tools** (Computer-Assisted Translation tools) like:
 - SDL Trados
 - MemoQ
- Language quality tools like:

- **Xbench**
 - **QA Distiller**
-

7. Who Performs Linguistic Testing?

- **Native speakers**
 - **Language specialists**
 - **Professional translators**
 - Sometimes assisted by QA/testers for UI issues
-

8. Common Issues Detected in Linguistic Testing

- Mistranslation
 - Incorrect context
 - Cut-off text or overlapping text
 - Improper gender usage in translation
 - Wrong terminology
 - Inconsistent use of tone or formality
-

9. Summary for Exam

- **Linguistic Testing** ensures **language correctness** of the application
- Checks grammar, spelling, context, formatting, and layout
- Mainly performed in **localized (multi-language) software**

- A part of the **localization quality assurance (LQA)** process
 - Performed by **native linguists** or translation experts
-

✅ You are now 100% ready to answer any question on **Linguistic Testing** in your exam!

Would you like the **next topic in Unit 4** or another topic (like **Exploratory Testing**, **Usability Testing**, etc.)? Just tell me!

Let's now cover "**Metrics**" from **Unit 4: Software Testing** in a complete and exam-ready format, including **definitions**, **diagrams (if applicable)**, **examples**, and key points.

Unit 4 – Topic: Metrics in Software Testing

1. Definition of Metrics

In software testing, **metrics** are **quantitative measures** used to assess the **quality**, **progress**, **efficiency**, and **effectiveness** of the testing process.

✅ They help in **decision-making**, **improving testing processes**, and **ensuring software quality**.

2. Types of Software Testing Metrics

A. Product Metrics

- Measure characteristics of the **software product** itself.

Metric	Meaning
--------	---------

Defect Density	Number of defects per KLOC
----------------	----------------------------

Code % of code tested
Coverage

Test Coverage % of requirements tested

B. Process Metrics

- Measure the **efficiency of the testing process**.

Metric	Meaning
Defect Removal Efficiency (DRE)	% of defects removed before release
Test Case Effectiveness	% of defects detected by test cases
Test Execution Rate	# of test cases executed per time unit

C. Project Metrics

- Measure aspects of the **overall project performance**.

Metric	Meaning
Cost of Quality	Total cost to ensure quality

Schedule
Variance

Difference from planned vs actual time



3. Key Software Testing Metrics with Formulas

Metric Name	Formula	Usefulness
Defect Density	$\text{Defects} / \text{KL0C (thousand lines of code)}$	Measures quality of code
Test Case Effectiveness	$\text{Defects Detected} / \text{Total Defects}$	Checks how good the test cases are
Test Case Execution Rate	$\text{Executed Test Cases} / \text{Total Test Cases}$	Measures progress in test execution
Defect Removal Efficiency (DRE)	$(\text{Defects Found During Testing}) / (\text{Total Defects})$	Evaluates test team performance
Mean Time to Detect (MTTD)	$\text{Time defects found} / \text{Total number of defects}$	Measures defect detection speed
Mean Time to Repair (MTTR)	$\text{Time taken to fix defects} / \text{Number of defects fixed}$	Measures fixing speed



4. Example: Defect Density

Assume:

- 5,000 lines of code (5 KLOC)
- 20 defects found

Defect Density = $20 / 5 = 4$ defects/KLOC

✓ Lower value = better code quality

5. Why Are Metrics Important?

Reason	Benefit
Track progress	Know how much testing is done
Improve quality	Identify weak areas in testing/code
Make decisions	Release readiness, resource allocation
Optimize effort	Focus on high-defect modules or test cases

6. Summary for Exam

- **Metrics** = measurable indicators of software quality and testing performance
- Types: **Product**, **Process**, **Project** metrics
- Help in **monitoring, evaluating, and improving** the testing process
- Common metrics: **Defect Density**, **DRE**, **Test Coverage**, **MTTR**, etc.

✓ You're now ready to write **short notes or detailed answers** on **Metrics** in the exam.

Would you like to continue with the **next Unit 4 topic** (like **Exploratory Testing, Usability Testing, or Types of Reviews**) or jump to another unit?

Here's a complete, exam-ready explanation of **Path Products and Path Expressions**, from **Unit 5: Logic-Based Testing** — including **definitions, diagrams, examples**, and important points for your **Software Testing** exam.

Unit 5 – Topic: Path Products and Path Expressions

1. What is Path Testing? (Quick Recap)

Path Testing is a **white-box testing technique** where we test **all possible paths** (independent and dependent) in a control flow graph (CFG) of the program to ensure **maximum code coverage**.

2. Control Flow Graph (CFG)

A **Control Flow Graph** is a directed graph that shows:

- **Nodes** = Statements or blocks in the program
- **Edges** = Flow of control (i.e., direction of execution)

Example:

[Start]

|

v

[A]

/ \

v v
[B] [C]
\
v v
[D]
|
[End]

3. What is a Path Product?

A **Path Product** is a mathematical way to **represent the execution paths** in a program using symbols for each node and combining them using operations.

Definition:

A **path product** is an algebraic representation of all possible execution paths in a program using sequence (.), choice (+), and iteration (*) operators.

Symbols Used in Path Products:

Symbol	Meaning	Example
.	Sequence (AND)	A.B means A then B
+	Choice (OR)	A + B means A or B

* Loop / Iteration A* means repeat A 0 or more times

Example of Path Product

Using the previous CFG:

A

/ \

B C

\ /

D

Let's write the **Path Product**:

A.(B + C).D

■ Meaning:

- Start at A
 - Then either go to **B** or **C**
 - Then always go to **D**
-

4. What is a Path Expression?

A **Path Expression** is a **regular expression-like format** that describes **all paths** through a program module, based on its **control flow graph**.

✓ Path Expressions are written using:

- **Sequence (.)**
- **Selection (+)**
- **Repetition (*)**

📌 Essentially, a path product **is** a type of path expression.

Example – If-Else Statement

if ($x > 0$)

A;

else

B;

C;

CFG:

```

[Start]
  |
  v
[Condition]
 /   \
[A]   [B]
 \   /
  v   v
    [C]
  
```

Path Product / Path Expression:

(Condition).(A + B).C

5. Why Use Path Products/Expressions?

Purpose	Benefit
Identify paths to test	Helps design effective test cases
Maximize code coverage	Ensures all logic branches tested
Detect redundant paths	Improves testing efficiency
Useful in automation	Helps in path-based test generation

6. Summary for Exam

- **Path Products/Expressions** describe **all possible execution paths** through a program.
 - Use **CFG** as the base, then convert it using:
 - **.** (sequence), **+** (choice), ***** (loop)
 - Helps in designing test cases for **maximum path coverage**
 - Similar to **regular expressions**
-

Short Exam Answer Example

Q: Define Path Product and Path Expression with example.

Ans:

A **path product** or **path expression** is a regular-expression-like representation of all possible execution paths in a program's control flow graph.

Example:

For a CFG with a condition branching to either A or B and then C, the path product is:

`Condition.(A + B).C`

It helps in white-box testing by guiding the design of test cases to cover all logical paths.

✓ Let me know if you want a **diagram-based handwritten-style summary** or help with the next topic like **Cause-Effect Graphing**, **Exploratory Testing**, or **Mutation Testing**.

Here's a complete, exam-ready note on **Syntax Testing** from **Unit 4: Software Testing**, including **definition**, **explanation**, **example**, and **diagram**, tailored for your exam tomorrow.

Unit 4 – Topic: Syntax Testing

1. Definition of Syntax Testing

Syntax Testing is a **black-box testing technique** used to test **input combinations** to a software system to determine whether it **correctly accepts valid inputs** and **rejects invalid ones** based on a specified **input grammar or syntax rules**.

✓ It is especially useful when the input data must follow a **specific format or grammar**, such as in programming languages, command-line inputs, data file formats, or forms.

2. Purpose of Syntax Testing

- Ensure that the software **accepts only valid input formats**
- Catch errors related to **parsing or input handling**

- Verify that invalid or malformed input is **properly rejected**
-

3. When is Syntax Testing Used?

- When inputs must follow **formal syntax rules** (grammar)
 - In **compilers, interpreters, parsers**
 - Input forms like:
 - Email: `example@domain.com`
 - Dates: `DD-MM-YYYY`
 - Phone numbers: `+91-9876543210`
-

4. Basic Components

- **Input Grammar:** A formal set of rules describing allowed input
 - **Test Cases:** Derived from grammar, include both **valid and invalid** inputs
 - **Parser:** Part of the software that checks input syntax
-

5. Example of Syntax Testing

Imagine a form that accepts a date in the format **DD-MM-YYYY**:

Grammar Rules:

- DD: 01 to 31
- MM: 01 to 12

- YYYY: 1900 to 2099

✓ **Valid Input:**

- 15-08-2025

✗ **Invalid Inputs:**

- 32-12-2025 (invalid day)
- 10-13-2023 (invalid month)
- 15/08/2025 (wrong separator)

In syntax testing, you write test cases to **verify whether the software accepts the valid one and rejects the invalid ones.**

6. Syntax Testing Diagram (Conceptual)

[Input Specification]

|

v

[Generate Grammar]

|

v

[Create Test Inputs (Valid/Invalid)]

|

v

[Run Through System Under Test]

|

v

[Check Pass/Fail for Each Input]

7. Test Case Design in Syntax Testing

Test Case	Input	Expected Output
TC1	12-09-2025	Accepted (Valid)
TC2	99-12-2025	Rejected (Invalid)
TC3	12/09/2025	Rejected (Wrong sep)
TC4	ab-cd-efgh	Rejected (Not digits)
TC5	01-01-1899	Rejected (Year out of range)

8. Advantages of Syntax Testing

- Ensures system only accepts **well-formed input**
- Catches **early input validation issues**
- Improves **robustness** and **security**
- Prevents crashes from malformed input

⚠️ 9. Limitations of Syntax Testing

- Only checks **structure**, not **meaning** (semantic errors)
 - Can be complex for large grammars
 - Needs detailed knowledge of input syntax rules
-

🧠 10. Summary for Exam

Key Point	Description
Testing Type	Black-box testing
Focus	Input grammar and structure
Tests	Valid + Invalid input formats
Goal	Accept valid input, reject invalid ones
Used In	Forms, compilers, command-line inputs

📝 Sample Exam Answer (3-5 Marks)

Q: Define Syntax Testing with an example.

Ans:

Syntax Testing is a black-box testing technique used to check whether the software correctly accepts inputs that follow a specified grammar or format and rejects those that do not.

Example:

For a date field that accepts input in the format DD-MM-YYYY, a valid input is `15-08-2025`, and an invalid input is `99-13-2023`.

Syntax testing ensures that such inputs are properly handled.

✅ Let me know if you'd like to continue with the next topic — like **Exploratory Testing**, **Usability Testing**, or any other pending topic from **Unit 4 or Unit 5**. Great! Let's start with **Unit 5: Logic-Based Testing** — I'll give you **definitions, diagrams, examples, and explanations** in a simple exam-ready format.

Unit 5: Logic-Based Testing

1. Definition of Logic-Based Testing

Logic-Based Testing is a **white-box testing technique** that focuses on **logical conditions, decision outcomes, and control flow** within the code. It helps identify bugs in the decision-making logic of the program.

Example:

Testing if-else conditions, loops, or logical expressions like `if (age > 18 && age < 60)`.

2. Types of Logic-Based Testing Techniques

A. Predicate Coverage (PC)

Definition:

Ensures that **every decision (predicate)** in the program is tested with **both True and False** outcomes.

Example Code:

```
if (x > 0)
    print("Positive");
```

Test Cases:

- $x = 5 \rightarrow \text{True}$
- $x = -1 \rightarrow \text{False}$

✓ Predicate is tested for both outcomes



B. Clause Coverage (CC)

Definition:

Ensures that **each clause** (condition) in a compound predicate is tested for **True and False**.

Example Predicate:

$(x > 0) \ \&\& \ (y < 10)$

Clauses:

- $x > 0$
- $y < 10$

Test Cases:

- $x = 5, y = 5 \rightarrow \text{Both True}$
- $x = -1, y = 5 \rightarrow \text{First False}$
- $x = 5, y = 15 \rightarrow \text{Second False}$

✓ Each clause gets both T/F values



C. Combinatorial Clause Coverage (CoC)

Definition:

All **possible combinations** of truth values of clauses are tested.

Example Predicate:

$(A \ || \ B)$

Combinations:

- A = T, B = T
- A = T, B = F
- A = F, B = T
- A = F, B = F

✓ All combinations are tested



D. Modified Condition/Decision Coverage (MC/DC)

Definition:

Each **clause** should independently affect the **decision outcome** while keeping other clauses constant.

Example Predicate:

$(x > 0) \ \&\& \ (y < 10)$

We must show that **changing x** can change the result of the predicate independently, and same for y.

Test Cases:

- $x = 5, y = 5 \rightarrow \text{True}$
- $x = -1, y = 5 \rightarrow \text{False} \rightarrow \text{Change in } x \text{ changed result}$
- $x = 5, y = 15 \rightarrow \text{False} \rightarrow \text{Change in } y \text{ changed result}$

✓ MC/DC satisfied

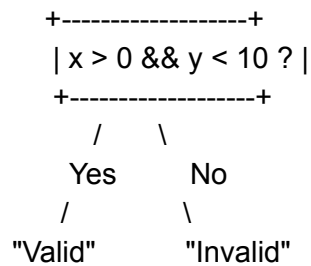


3. Diagram – Logic Flow

Example Code:

```
if (x > 0 && y < 10)
    print("Valid");
else
    print("Invalid");
```

Flowchart Diagram:



This is a **decision node** where **predicate logic** is used.

4. Truth Table for Logical Expression

Example Predicate: `A && B`

A	B	A && B
T	T	T
T	F	F
F	T	F
F	F	F

Use this table for **Combinatorial Clause Coverage** and **MC/DC testing**

5. Key Differences

Technique	Goal
Predicate Coverage	Test each decision for T and F
Clause Coverage	Test each condition in T and F
Combinatorial Clause	All combinations of clause values
Modified Condition (MC/DC)	Each clause independently affects outcome

6. Why Logic-Based Testing?

- Helps find **hidden logical errors**
- Ensures **decision-making logic** is reliable
- Improves **code quality and reliability**

7. Summary

- Logic-Based Testing = **white-box** technique
- Tests **conditions, decisions, and combinations**
- Focuses on **internal logic**, not just output

Would you like to continue to **Unit 5 – Next Topic (like Path Testing or Decision Tables)** or jump to **Unit 4**? Let me know your next topic or unit and I'll prepare it just like this one.

Perfect! Let's now cover **Decision Tables** — another important topic from **Unit 5: Logic-Based Testing** — with **definitions, diagrams, examples, and clear explanations** so you're 100% ready for your exam.

Unit 5 – Topic: Decision Tables

1. Definition of Decision Table

A **Decision Table** is a **tabular method** for representing and analyzing complex business rules and logical conditions. It helps identify all possible combinations of inputs and the corresponding actions to be taken.



2. Structure of a Decision Table

A Decision Table is divided into **four parts**:

1. **Conditions (Inputs)** – Logical rules or input values.
2. **Actions (Outputs)** – What to do when certain conditions are met.
3. **Rules/Columns** – Each column shows a unique combination of inputs and their resulting actions.
4. **Condition Entries** – "Y" for Yes, "N" for No.



3. Diagram: Format of a Decision Table

CONDITIONS		Rule 1	Rule 2
Is user logged in?	Y	N	
Is user admin?	Y	-	
ACTIONS			
Allow access	X		
Show login screen		X	

- **Y** = Yes, **N** = No, **-** = Don't care, **X** = Action to be taken



4. Example: Login Access Control

Conditions:

- C1: Is username correct?
- C2: Is password correct?

Actions:

- A1: Allow login
- A2: Show error

Decision Table:

Conditions	Rule 1	Rule 2	Rule 3	Rule 4
C1: Username correct	Y	Y	N	N
C2: Password correct	Y	N	Y	N
-----	-----	-----	-----	-----
A1: Allow login	X			
A2: Show error		X	X	X

5. Use of Decision Tables in Testing

- To test all combinations of inputs and their expected outputs.
 - Helps in identifying **missing conditions** or **ambiguous rules**.
 - Used in **black-box testing**.
-

6. Advantages of Decision Tables





Advantage	Explanation
Clear representation	Shows all possible input combinations
Easy to convert to test cases	Each column = one test case
Handles complex logic well	Suitable for systems with many business rules

7. Steps to Create a Decision Table

1. **Identify conditions (inputs)**
 2. **List possible actions (outputs)**
 3. **Create all possible combinations of conditions**
 4. **Determine actions for each rule**
 5. **Simplify table (if needed)**
-

8. Test Cases from Decision Table (Example)

For the login example:

- **Test Case 1:** Username = correct, Password = correct →  Login success
 - **Test Case 2:** Username = correct, Password = wrong →  Show error
 - **Test Case 3:** Username = wrong, Password = correct →  Show error
 - **Test Case 4:** Username = wrong, Password = wrong →  Show error
-

9. Summary Points for Exam

- **Decision Table** is a logical representation of input-output relationships.
 - Divides into **conditions**, **actions**, and **rules**.
 - Helps generate **test cases** efficiently for complex logic.
 - Mostly used in **black-box testing**.
-

 That's all you need to confidently answer **Decision Table questions** in your exam!

Would you like to continue with the **next topic in Unit 5** (like **Path Testing** or **Cause-Effect Graphing**) or move to another unit (e.g., Unit 4)? Just tell me the topic or unit name!

Great! Let's now cover **Transition Testing**, another key topic from **Unit 5: Logic-Based Testing**, with clear **definitions**, **diagrams**, **examples**, and **exam-focused points**.

Unit 5 – Topic: Transition Testing

1. Definition of Transition Testing

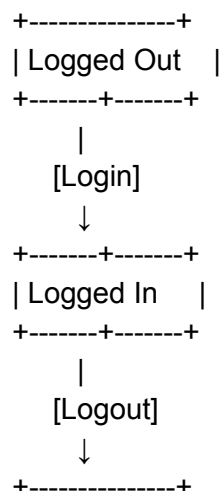
Transition Testing (also called **State Transition Testing**) is a **black-box testing technique** used to test **valid and invalid state changes** in a system. It's useful when software behavior depends on **different states** and **events** that trigger transitions between those states.

2. When to Use Transition Testing

- When the system behaves **differently based on its current state**
 - When **events cause transitions** from one state to another
 - Example: Login systems, online shopping carts, ATMs, etc.
-

3. Diagram: State Transition Diagram

Example: **Simple Login System**



| Logged Out |
+-----+

- **States:** Logged Out, Logged In
- **Events:** Login, Logout
- **Transitions:** Change from one state to another due to an event

4. State Transition Table Example

Current State	Event	Next State	Action
Logged Out	Login	Logged In	Show dashboard
Logged In	Logout	Logged Out	Show login screen
Logged Out	Logout	Logged Out	Show error message
Logged In	Login	Logged In	Show already logged in

5. Test Case Design from State Transition Table

From the table above, test cases can be:

Test Case	Current State	Input	Expected Output
TC01	Logged Out	Login	Move to Logged In
TC02	Logged In	Logout	Move to Logged Out
TC03	Logged Out	Logout	Error message
TC04	Logged In	Login	Already logged in msg

6. Types of Transitions

Transition Type	Description
-----------------	-------------

Valid Transition	Legal/expected state change
Invalid Transition	Illegal state change, should return an error

7. Where is Transition Testing Used?

- Login/logout systems
 - Banking systems (ATM states)
 - Online order processing
 - Elevator control systems
-

8. Advantages of Transition Testing

Benefit	Explanation
Detects unexpected behavior	Validates how system reacts to events
Easy visual representation	State diagrams help in better understanding
Covers user interaction flow	Tests system from user's perspective

9. Key Terms in Transition Testing

Term	Meaning
State	A condition or situation of the system
Event	Input or action that causes a change
Transition	Change from one state to another
Action	Output or result after a transition



10. Summary for Exam

- **Transition Testing** tests state changes of a system.
 - Uses **State Transition Diagrams** or **Tables**.
 - Identifies **valid/invalid transitions**.
 - Useful for systems with **finite states** (FSM – Finite State Machine).
-



That completes **Transition Testing** — you're fully ready for this topic!

Would you like the next topic in **Unit 5** (e.g., **Path Testing**, **Cause-Effect Graphing**) or shall we start **Unit 4** or another unit? Let me know!

Great! Let's now fully cover "**States, State Graphs & State Testing**" from **Unit 5: Logic-Based Testing** — with **definition, diagrams, examples, and key points** for exam preparation.



Unit 5 – Topic: States, State Graphs & State Testing



1. Definition of State

A **state** is a **specific condition or situation** of a system at a given time. A system moves from one state to another based on **inputs/events**.



In state-based testing, we test how a system behaves in different **states** and how it **transitions** between them.



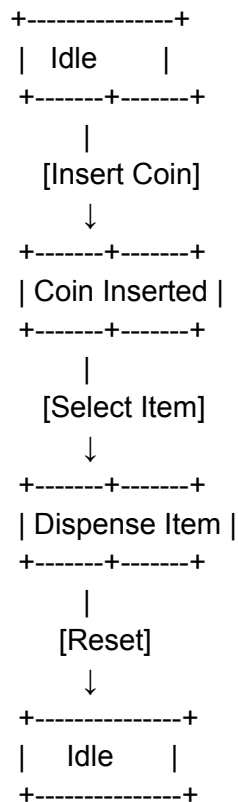
2. State Graph (State Transition Diagram)

A **State Graph** is a **visual representation** of the system's behavior using:

- **States** (circles)
- **Events or inputs** (labels on arrows)

- **Transitions** (arrows between states)
- **Actions or outputs** (optional, shown on transitions)

3. State Graph Example: Vending Machine



4. State Transition Table Format

Current State	Event/Input	Next State	Action/Output
Idle	Insert Coin	Coin Inserted	Accept Coin
Coin Inserted	Select Item	Dispense Item	Deliver Product
Dispense Item	Reset	Idle	Ready for next user

5. What is State Testing?

State Testing is a technique where test cases are designed to test the **system's behavior in various states**, and how it transitions between them.

✓ Focuses on **valid and invalid state transitions**.

6. Test Case Example from Vending Machine

Test Case	Input	Expected State	Output
TC01	Insert Coin	Coin Inserted	Coin Accepted
TC02	Select Item	Dispense Item	Product Delivered
TC03	Reset	Idle	Ready State

7. Invalid State Transitions

These are transitions that **should not happen**. For example:

- Press "Select Item" **without inserting a coin**
→ ❌ Invalid → Should show "Insert coin first"

Why test these?

To check how the system **handles errors or unexpected inputs**.

8. Types of State Testing

Type	Description
Positive Testing	Testing valid transitions
Negative Testing	Testing invalid transitions or unexpected inputs
Sequence Testing	Testing if transitions occur in the correct order

9. Advantages of State Testing

Benefit	Why It's Important
Detects logic errors	Ensures system behaves correctly in all states
Clear modeling	Easy to understand with diagrams and tables
Useful for real-time systems	Best for systems with events and status changes

10. Summary for Exam

- **State** = condition of a system at a moment
 - **State Graph** = diagram showing transitions between states
 - **State Table** = tabular form of all transitions
 - **State Testing** = tests how system behaves across various states
 - Focuses on **valid and invalid transitions**
-

 That's a complete and exam-ready explanation of **States, State Graphs, and State Testing**.

Would you like to continue with **Path Testing, Cause-Effect Graphing**, or start **another unit**?
Let me know your next topic!