Program Testing (Continued)

(Lecture 15)

Anil Kumar Dudyala
Dept. of CSE, NIT, Patna

Source
(Rajib Mall)

Organization of this Lecture:

- Review of last lecture.
- Data flow testing
- Mutation testing
- Cause effect graphing
- Performance testing.
- Test summary report
- Summary

Data Flow-Based Testing

- Selects test paths of a program:
  - According to the locations of
    - Definitions and uses of different variables in a program.

Data Flow-Based Testing

- For a statement numbered S,
  - DEF(S) = {X/statement S contains a definition of X}
  - USES(S)= {X/statement S contains a use of X}
  - Example: 1: a=b; DEF(1)={a}, USES(1)={b}.
  - Example: 2: a=a+b; DEF(1)={a}, USES(1)={a,b}.

Data Flow-Based Testing

- A variable X is said to be live at statement S1, if
  - X is defined at a statement S:
  - There exists a path from S to S1 not containing any definition of X.

Definition-use chain (DU chain)

- [X,S,S1],
  - S and S1 are statement numbers,
  - X in DEF(S)
  - X in USES(S1), and
  - the definition of X in the statement S is live at statement S1.

DU Chain Example

```
1 X(){
2  a=5; /* Defines variable a */
3  While(C1) {
4     if (C2)
5           b=a*a;   /*Uses variable a */
6           a=a-1; /* Defines variable a */
7     }
8   print(a); } /*Uses variable a */
```

Data Flow-Based Testing

- One simple data flow testing strategy:
  - Every DU chain in a program be covered at least once.
- Data flow testing strategies:
  - Useful for selecting test paths of a program containing nested if and loop statements.

Data Flow-Based Testing

```
. 1 X(){
. 2  B1;        /* Defines variable a */
. 3  While(C1) {
. 4     if (C2)
. 5           if(C4) B4; /*Uses variable a */
. 6        else B5;
. 7         else  if (C3) B2;
. 8        else B3;     }
. 9  B6 }
```

Data Flow-Based Testing

- [a,1,5]: a DU chain.
- Assume:
  - DEF(X) = {B1, B2, B3, B4, B5}
  - USED(X) = {B2, B3, B4, B5, B6}
  - There are 25 DU chains.
- However only 5 paths are needed to cover these chains.

Mutation Testing

- The software is first tested:
  - using an initial testing method based on white-box strategies we already discussed.
- After the initial testing is complete,
  - mutation testing is taken up.
- The idea behind mutation testing:
  - make a few arbitrary small changes to a program at a time.

Mutation Testing

- Each time the program is changed,
  - it is called a **mutated program**
  - the change is called a **mutant**.

Mutation Testing

- A mutated program:
  - tested against the full test suite of the program.
- If there exists at least one test case in the test suite for which:
  - a mutant gives an incorrect result,
  - then the mutant is said to be dead.

Mutation Testing

- If a mutant remains alive:
  - even after all test cases have been exhausted,
  - the test suite is enhanced to kill the mutant.
- The process of generation and killing of mutants:
  - can be automated by predefining a set of primitive changes that can be applied to the program.

Mutation Testing

- The primitive changes can be:
  - altering an arithmetic operator,
  - changing the value of a constant,
  - changing a data type, etc.

Mutation Testing

- A major disadvantage of mutation testing:
  - computationally very expensive,
  - a large number of possible mutants can be generated.

Cause and Effect Graphs

- Testing would be a lot easier:
  - if we could automatically generate test cases from requirements.
- Work done at IBM:
  - Can requirements specifications be systematically used to design functional test cases?

Cause and Effect Graphs

- Examine the requirements:
  - restate them as logical relation between inputs and outputs.
  - The result is a Boolean graph representing the relationships
    - called a cause-effect graph.

Cause and Effect Graphs

- Convert the graph to a decision table:
  - Each column of the decision table corresponds to a test case for functional testing.

Steps to Create Cause-Effect Graph

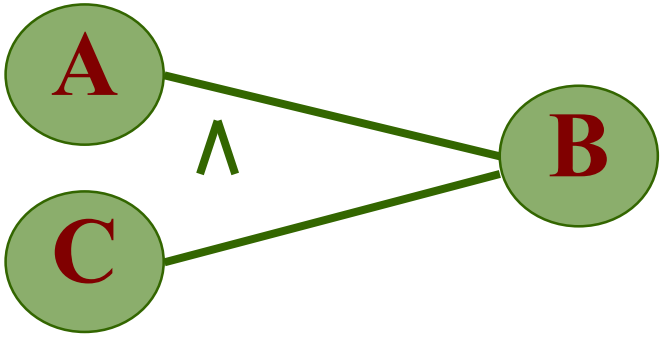
- Study the functional requirements.
- Mark and number all causes and effects.
- Numbered causes and effects:
  - become nodes of the graph.

Steps to Create Cause-Effect Graph

- Draw causes on the LHS
- Draw effects on the RHS
- Draw logical relationship between causes and effects
  - as edges in the graph.
- Extra nodes can be added
  - to simplify the graph

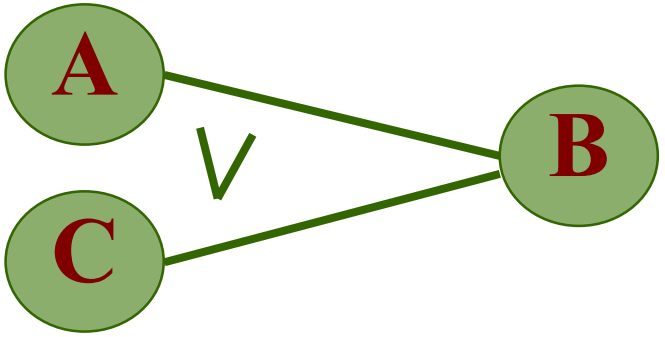
Drawing Cause-Effect Graphs

If A then B

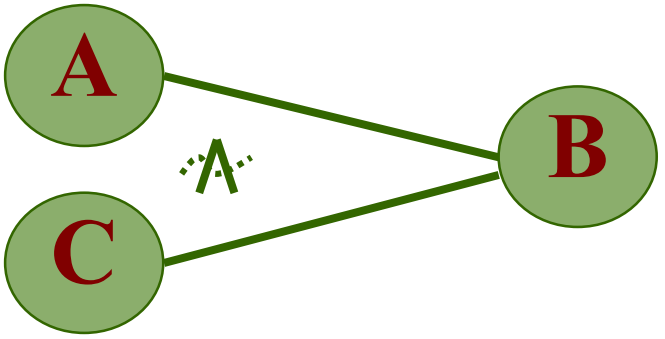


If (A and B)then C

Drawing Cause-Effect Graphs

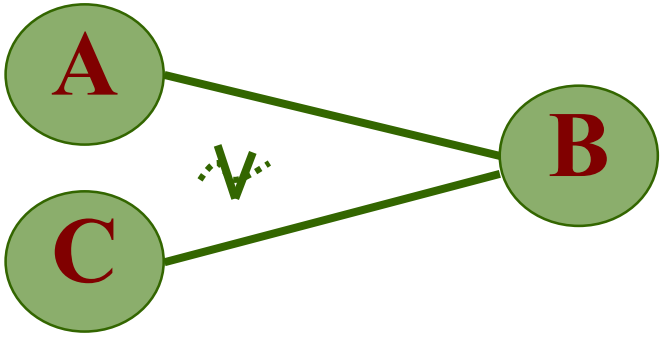


If (A or B)then C

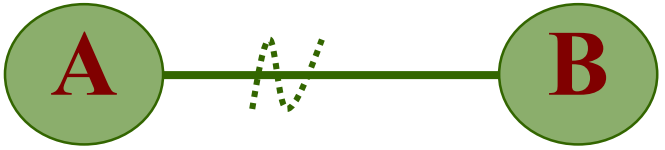


If (not(A and B))then C

Drawing Cause-Effect Graphs



If (not (A or B))then C



If (not A) then B

Cause effect graph- Example

- A water level monitoring system
  - used by an agency involved in flood control.
  - **Input:** level(a,b)
    - a is the height of water in dam in meters
    - b is the rainfall in the last 24 hours in cms

Cause effect graph- Example

- Processing
  - The function calculates whether the level is safe, too high, or too low.
- Output
  - message on screen
    - level=safe
    - level=high
    - invalid syntax

Cause effect graph- Example

- We can separate the requirements into 5 clauses:
  - first five letters of the command is "level"

1

  - command contains exactly two parameters

2

    - separated by comma and enclosed in parentheses

Cause effect graph- Example

- Parameters A and B are real numbers:

3 – such that the water level is calculated to be low

4 – or safe.

- The parameters A and B are real numbers:

5 – such that the water level is calculated to be high.

Cause effect graph- Example

10 – Command is syntactically valid

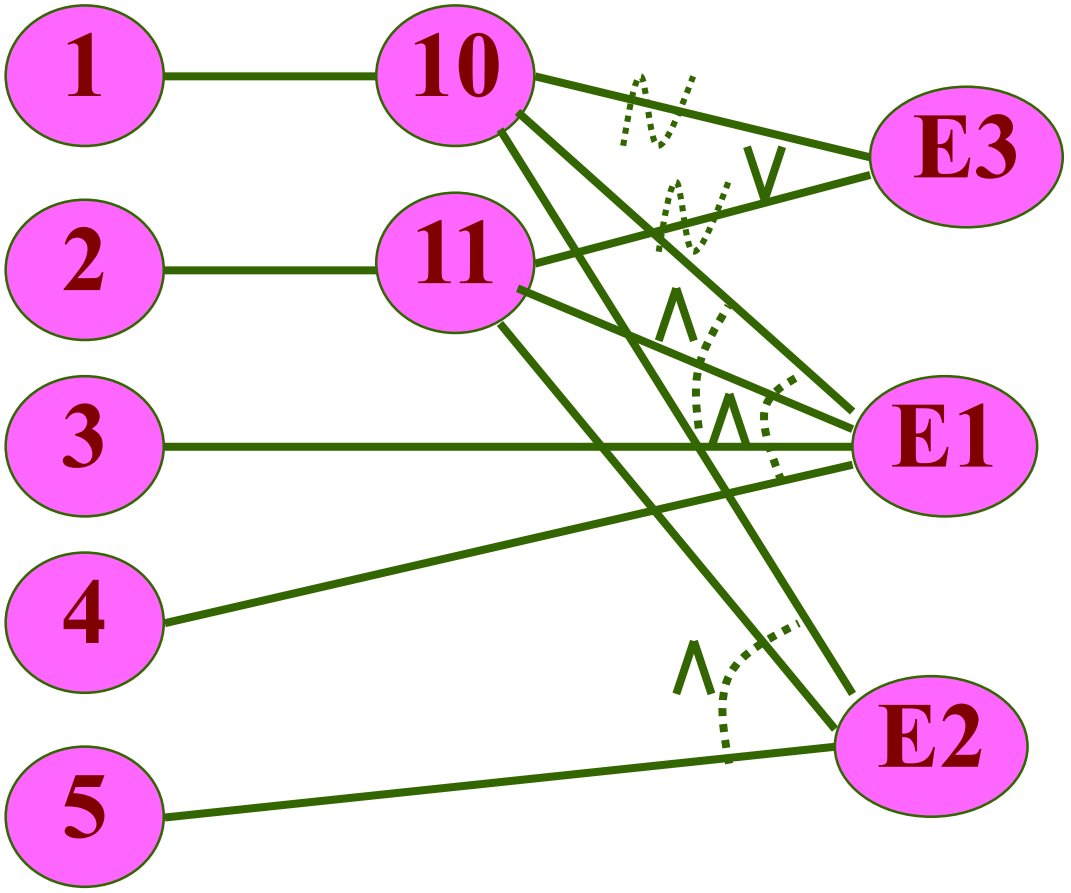
11 – Operands are syntactically valid.

Cause effect graph- Example

- Three effects
  - level = safe E1
  - level = high E2
  - invalid syntax E3

Cause effect graph- Example

IMPORTANT



Cause effect graph- Decision table

	Test 1	Test 2	Test 3	Test 4	Test 5
Cause 1	I	I	I	S	I
Cause 2	I	I	I	X	S
Cause 3	I	S	S	X	X
Cause 4	S	I	S	X	X
Cause 5	S	S	I	X	X
Effect 1	P	P	A	A	A
Effect 2	A	A	P	A	A
Effect 3	A	A	A	P	P

I = Invoked

x = don't care

s = supressed

P = present

A = absent

Cause effect graph- Example

- Put a row in the decision table for each cause or effect:
  - in the example, there are five rows for causes and three for effects.

Cause effect graph- Example

- The columns of the decision table correspond to test cases.
- Define the columns by examining each effect:
  - list each combination of causes that can lead to that effect.
- We can determine the number of columns of the decision table
  - by examining the lines flowing into the effect nodes of the graph.

Cause effect graph- Example

- Theoretically we could have generated 25 to 32 test cases.
  - Using cause effect graphing technique reduces that number to 5.
- Not practical for systems which:
  - include timing aspects
  - feedback from processes is used for some other processes.

Testing

- Unit testing:
  - test the functionalities of a single module or function.
- Integration testing:
  - test the interfaces among the modules.
- System testing:
  - test the fully integrated system against its functional and non-functional requirements.

Integration testing

- After different modules of a system have been coded and unit tested:
  - modules are integrated in steps according to an integration plan
  - partially integrated system is tested at each integration step.

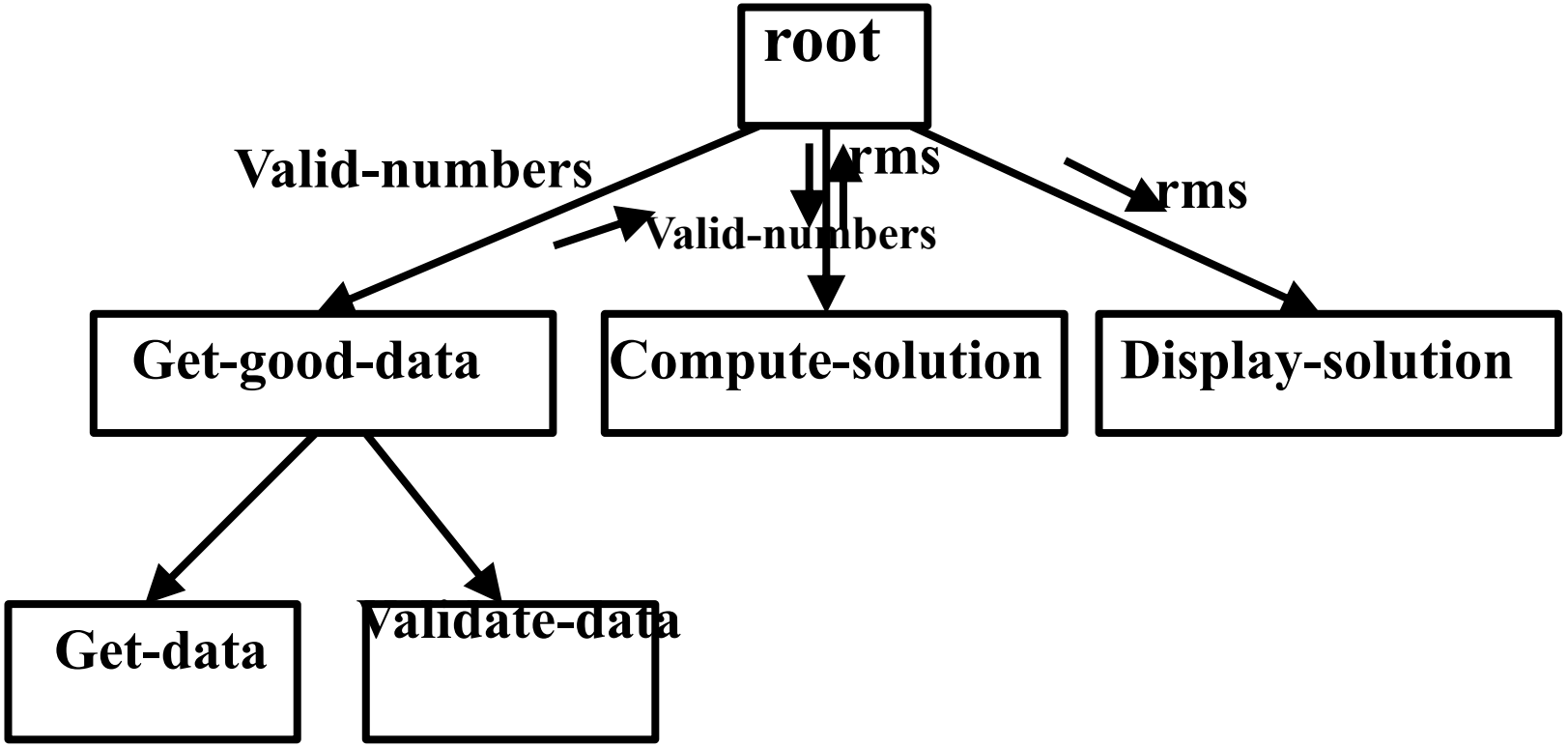
System Testing

- System testing:
  - Validate a fully developed system against its requirements.

Integration Testing

- Develop the integration plan by examining the structure chart :
  - big bang approach
  - top-down approach
  - bottom-up approach
  - mixed approach

Example Structured Design



Big bang Integration Testing

- Big bang approach is the simplest integration testing approach:
  - all the modules are simply put together and tested.
  - this technique is used only for very small systems.

Big bang Integration Testing

- Main problems with this approach:
  - If an error is found:
    - It is very difficult to localize the error
    - The error may potentially belong to any of the modules being integrated.
  - Debugging errors found during big bang integration testing are very expensive to fix.

Bottom-up Integration Testing

dummy module - driver

- Integrate and test the bottom level modules first.
- A disadvantage of bottom-up testing:
  - When the system is made up of a large number of small subsystems.
    - This extreme case corresponds to the big bang approach.

Top-down integration testing

dummy module - stubs

- Top-down integration testing starts with the main routine:
  - and one or two subordinate routines in the system.
- After the top-level 'skeleton' has been tested:
  - Immediate subordinate modules of the 'skeleton' are combined with it and tested.

Mixed integration testing

- Mixed (or sandwiched) integration testing:
  - Uses both top-down and bottom-up testing approaches.
  - Most common approach

Integration Testing

- In top-down approach:
  - testing waits till all top-level modules are coded and unit tested.
- In bottom-up approach:
  - testing can start only after bottom level modules are ready.

Phased versus Incremental Integration Testing

- Integration can be incremental or phased.
- In incremental integration testing,
  - only one new module is added to the partial system each time.

Phased versus Incremental Integration Testing

- In phased integration,
  - A group of related modules are added to the partially integrated system each time.
- Big-bang testing:
  - A degenerate case of the phased integration testing.

Phased versus Incremental Integration Testing

- Phased integration requires less number of integration steps:
  - compared to the incremental integration approach.
- However, when failures are detected,
  - it is easier to debug if using incremental testing
    - since errors are very likely to be in the newly integrated module.

System Testing

- System tests are designed to validate a fully developed system:
  - To assure that it meets its requirements.

System Testing

- There are three main kinds of system testing:
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Alpha testing

- System testing is carried out
  - by the test team within the developing organization.

Beta Testing

- Beta testing is the system testing:
  - performed by a select group of friendly customers.

Acceptance Testing

- Acceptance testing is the system testing performed by the customer
  - to determine whether he should accept the delivery of the system.

System Testing

- During system testing:
  - Functional requirements are validated through functional tests.
  - Non-functional requirements validated through performance tests.

Performance Testing

- Addresses non-functional requirements.
  - May sometimes involve testing hardware and software together.
  - There are several categories of performance testing.

Stress testing

- Evaluates system performance
  - when stressed for short periods of time.
- Stress testing
  - also known as endurance testing.

Stress testing

- Stress tests are black box tests:
  - Designed to impose a range of abnormal and even illegal input conditions
  - So as to stress the capabilities of the software.

Stress Testing

- If the requirements is to handle a specified number of users, or devices:
  - Stress testing evaluates system performance when all users or devices are busy simultaneously.

Stress Testing

- If an operating system is supposed to support 15 multiprogrammed jobs,
  - The system is stressed by attempting to run 15 or more jobs simultaneously.
- A real-time system might be tested
  - To determine the effect of simultaneous arrival of several high-priority interrupts.

Stress Testing

- Stress testing usually involves an element of time or size,
  - Such as the number of records transferred per unit time,
  - The maximum number of users active at any time, input data size, etc.
- Therefore stress testing may not be applicable to many types of systems.

Volume Testing

- Addresses handling large amounts of data in the system:
  - Whether data structures (e.g. queues, stacks, arrays, etc.) are large enough to handle all possible situations.
  - Fields, records, and files are stressed to check if their size can accommodate all possible data volumes.

Configuration Testing

- Analyze system behavior:
  - in various hardware and software configurations specified in the requirements
  - sometimes systems are built in various configurations for different users
  - for instance, a minimal system may serve a single user,
    - other configurations for additional users.

Compatibility Testing

- These tests are needed when the system interfaces with other systems:
  - Check whether the interface functions as required.

Compatibility testing Example

- If a system is to communicate with a large database system to retrieve information:
  - A compatibility test examines speed and accuracy of retrieval.

Recovery Testing

- These tests check response to:
  - Presence of faults or to the loss of data, power, devices, or services
  - Subject system to loss of resources
    - Check if the system recovers properly.

Maintenance Testing

- Diagnostic tools and procedures:
  - help find source of problems.
  - It may be required to supply
    - memory maps
    - diagnostic programs
    - traces of transactions,
    - circuit diagrams, etc.

Maintenance Testing

- Verify that:
  - all required artifacts for maintenance exist
  - they function properly

Documentation tests

- Check that required documents exist and are consistent:
  - user guides,
  - maintenance guides,
  - technical documents

Documentation tests

- Sometimes requirements specify:
  - Format and audience of specific documents
  - Documents are evaluated for compliance

Usability tests

- All aspects of user interfaces are tested:
  - Display screens
  - messages
  - report formats
  - navigation and selection problems

Environmental test

- These tests check the system's ability to perform at the installation site.
- Requirements might include tolerance for
  - heat
  - humidity
  - chemical presence
  - portability
  - electrical or magnetic fields
  - disruption of power, etc.

Test Summary Report

- Generated towards the end of testing phase.
- Covers each subsystem:
  - A summary of tests which have been applied to the subsystem.

Test Summary Report

- Specifies:
  - how many tests have been applied to a subsystem,
  - how many tests have been successful,
  - how many have been unsuccessful, and the degree to which they have been unsuccessful,
    - e.g. whether a test was an outright failure
    - or whether some expected results of the test were actually observed.

Regression Testing

- Does not belong to either unit test, integration test, or system test.
  - In stead, it is a separate dimension to these three forms of testing.

Regression testing

- Regression testing is the running of test suite:
  - after each change to the system after each bug fix.
  - Ensures that no new bug has been introduced due to the change or the bug fix.

Regression testing

- Regression tests assure:
  - the new system's performance is at least as good as the old system.
  - Always used during incremental system development.

Summary

- We discussed two additional white box testing methodologies:
  - data flow testing
  - mutation testing

Summary

- Data flow testing:
  - derive test cases based on definition and use of data
- Mutation testing:
  - make arbitrary small changes
  - see if the existing test suite detect these
  - if not, augment test suite

Summary

- Cause-effect graphing:
  - can be used to automatically derive test cases from the SRS document.
  - Decision table derived from cause-effect graph
  - each column of the decision table forms a test case

Summary

- Integration testing:
  - Develop integration plan by examining the structure chart:
    - big bang approach
    - top-down approach
    - bottom-up approach
    - mixed approach

Summary: System testing

- Functional test
- Performance test
  - stress
  - volume
  - configuration
  - compatibility
  - maintenance