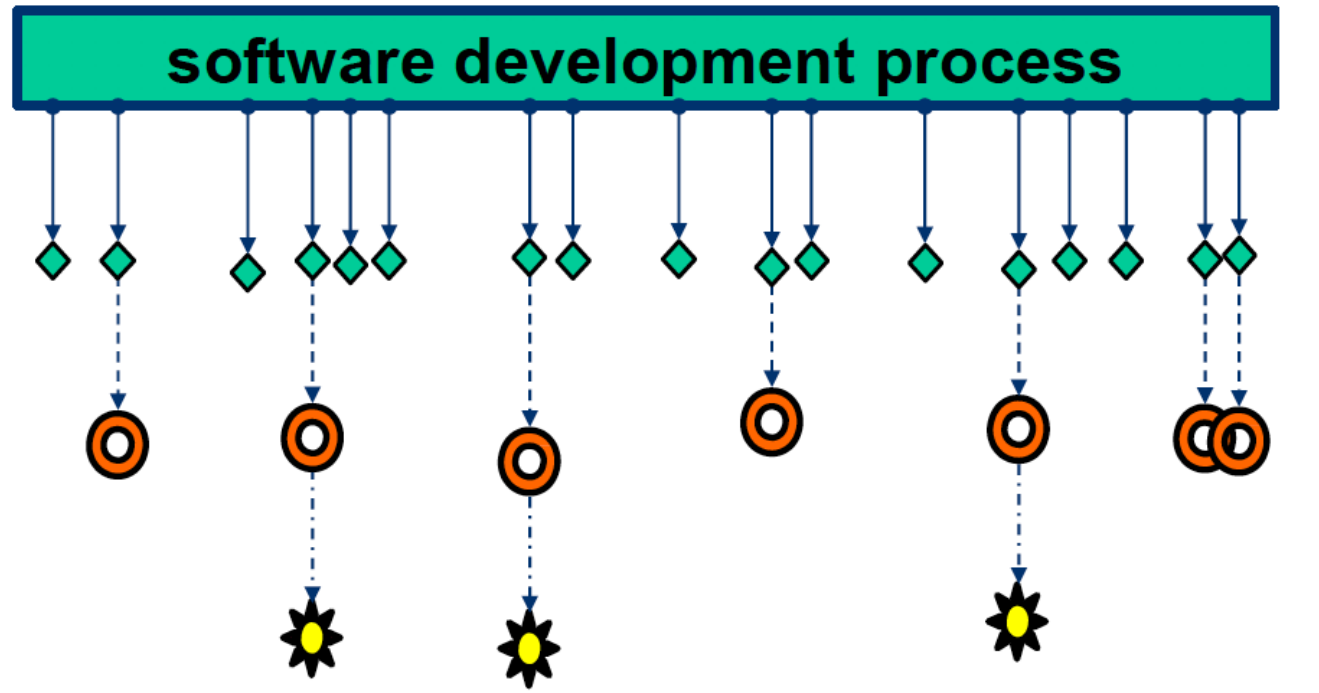
ECSE 429

Intro to Software Quality Assurance

Basic Terminology

Error, Fault, and Failure



- mistake: people commit errors
- defect: a mistake in software can lead to defect
- failure: occurs when defect executes
- incident: consequence of failures-failure occurrence may or may not be apparent to user

Defects Root Cause and Effect

- Root cause of defect: earliest action/condition that contributed to creating defect
- Root Cause Analysis: Identify root cause to reduce occurrence of similar defects in the future
- Effects of a defect: observed by user/customer, product owner

Nine Causes of Software Defects

1. Faulty requirements definitions
2. Client-developer communication
3. Deliberate deviations from software requirements

4. Logical design errors
5. Coding errors
6. non-compliance with documentation and coding instructions
7. Shortcomings of the testing process
8. Procedure errors
9. Documentation errors

Software Quality

Conformance to Requirements

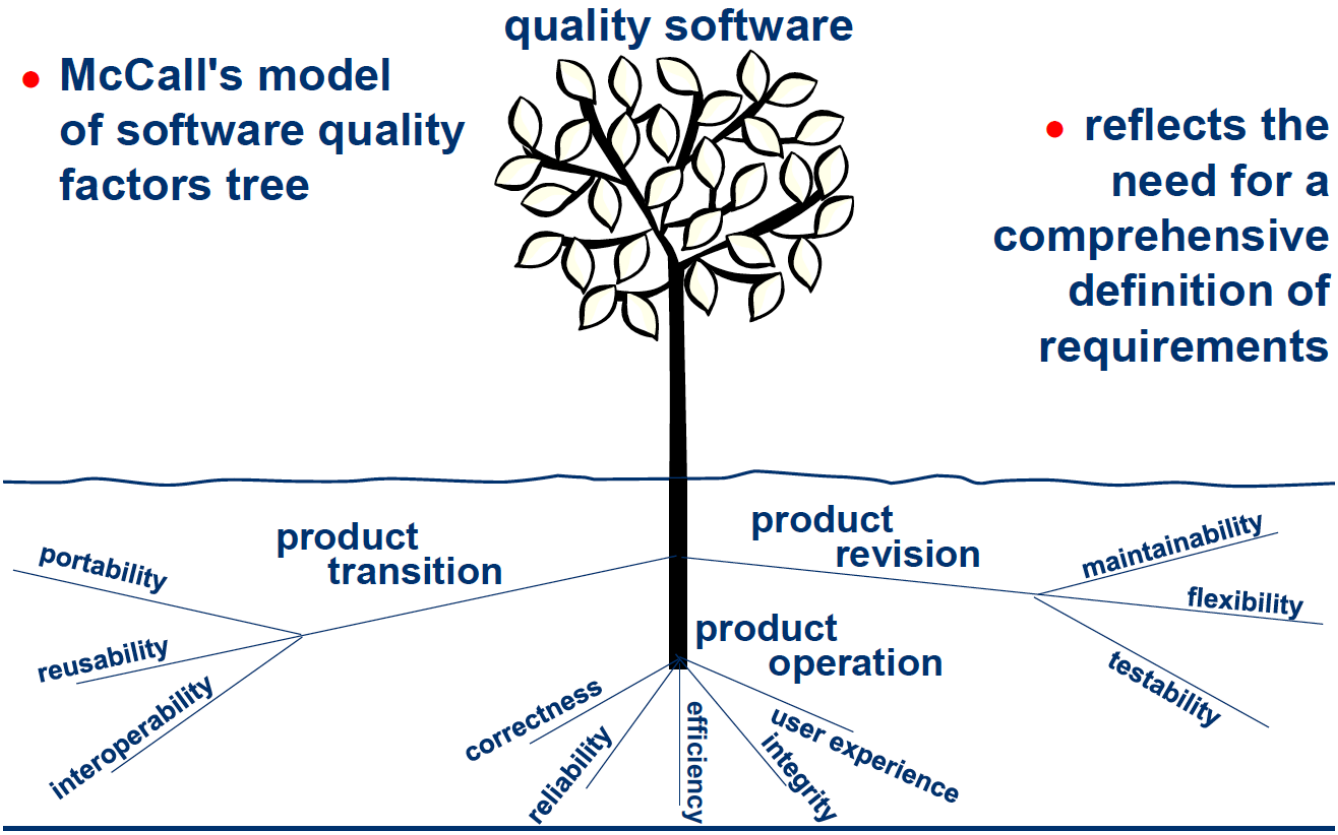
- Lack of bugs:
  - Low defect rate
  - well documented defects
- High reliability/Availability
  - **Mean time to failure** the probability of failure free operation until a specified time
  - **Mean time between failures** the probability that the system is up and running at any given point in time

Software Quality Challenge

- The uniqueness of the software product
  - High complexity: pervasive in an increasing number of industries
  - Invisibility of the product
  - Limited opportunities to detect defect compared to other industries
    - Development, not production
- The environments in which software is developed
  - Contracted
  - Subjected to customer-supplier relationship
  - Requirements for teamwork
  - etc.

Software Quality Factors

McCaul Method



- Correctness
  - Accuracy and completeness of required output
  - up to date
- Reliability
  - First failure
  - Max failure rate
- Efficiency
  - hardware resources needed to perform function
- Integrity
  - System security
- UX
- Maintainability
  - Effort to identify and fix errors
- Flexibility
- Testability
  - Support for testing, traceability
- Portability
  - Adaptation to other environments
- Reusability
  - Use of software components for other projects

- Interoperability
    - Ability to interface with other components/systems

SQA

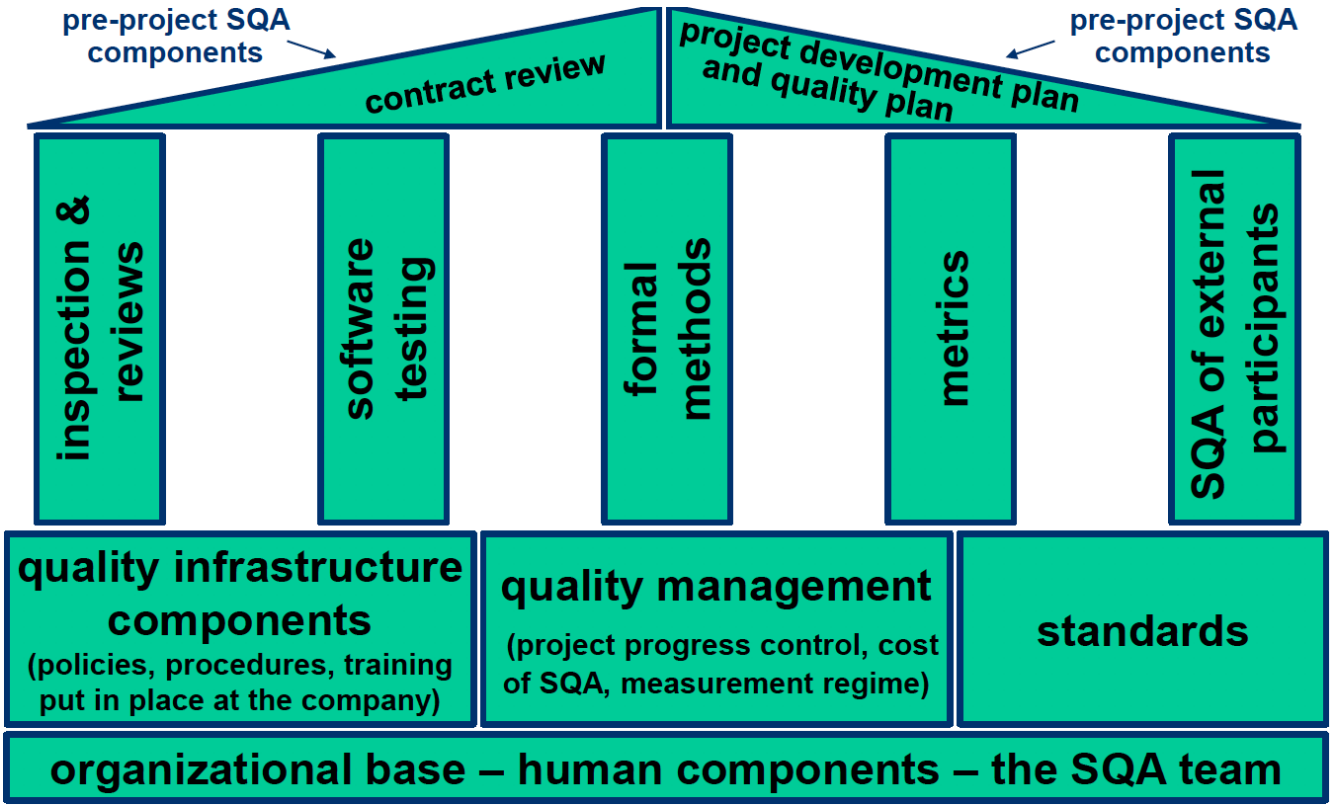
Objectives

1. Acceptable confidence that software will conform to functional technical requirements
2. Acceptable confidence that software will conform to managerial scheduling and budgetary requirements
3. Activities for the improvement/efficiency of software software development, software maintenance, and software quality assurance activities

Three key principles

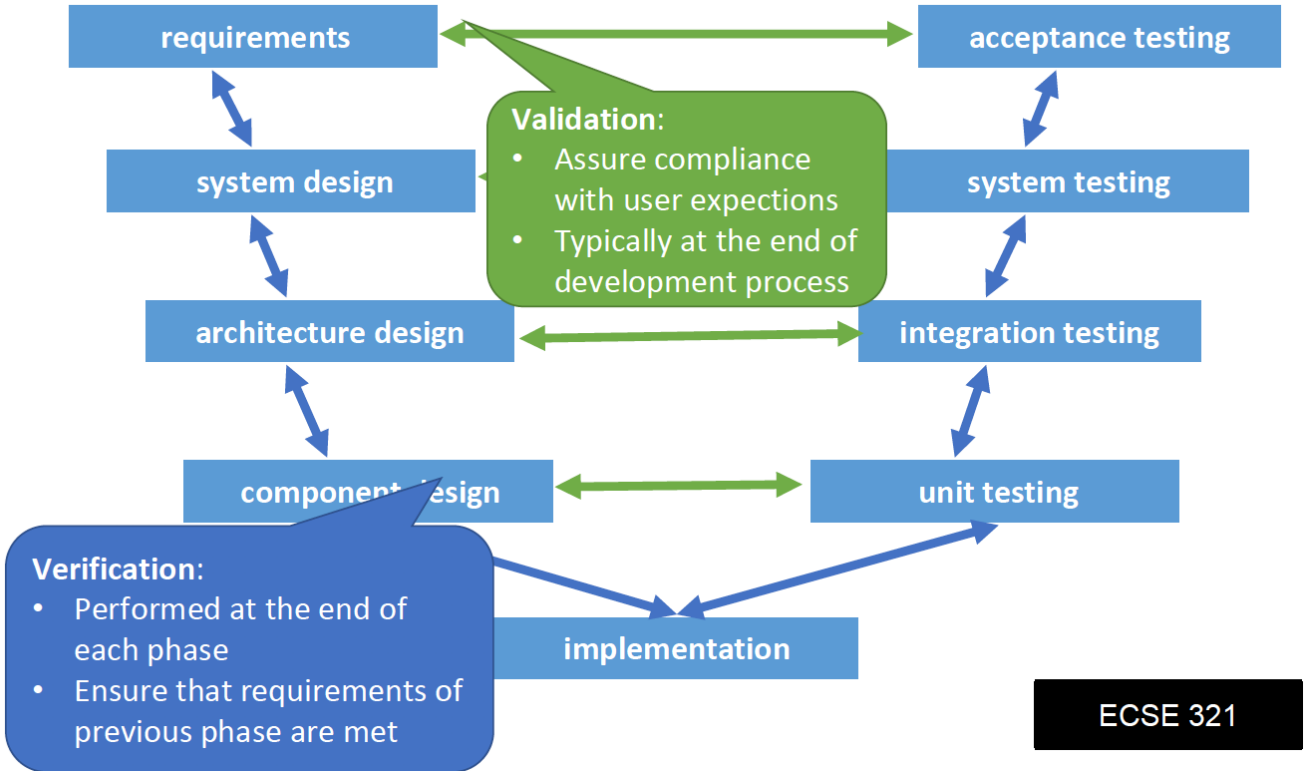
1. know what you are doing
    - what is being built, how its being built, what it does
    - Management structure
    - Reporting Policies
    - Tracking
2. know what you should be doing
    - Having explicit requirements and specifications
    - Requirements analysis
    - Acceptance tests
    - User feedback
3. know how to measure the difference
    - Measure comparing what is being done with what should be done
    - Includes:
        - Formal methods: prove mathematically
        - Testing: explicit input to exercise software and check for expected output
        - Inspections: human examination of requirements, design, code… checklists\
        - Metrics: measure a known set of properties related so quality

Software Quality Shrine



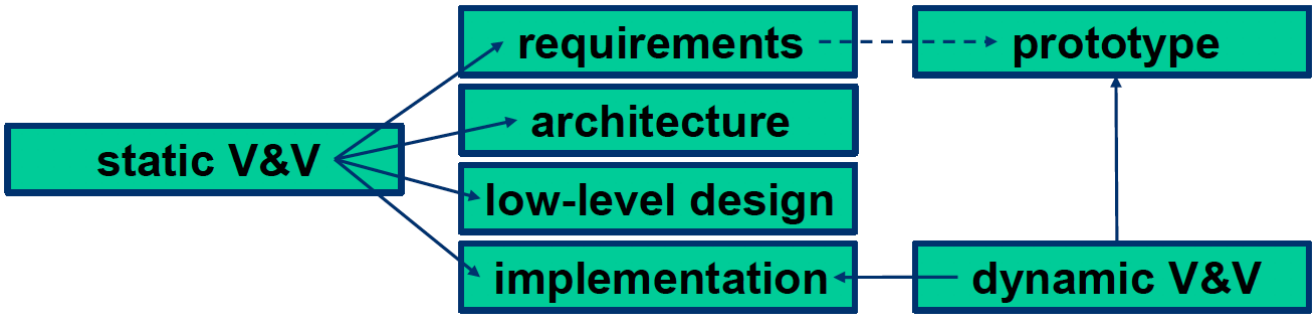
Verification vs Validation

- Verification: are we building the product right?
- Validation: are we building the right product?



SQA Includes

- Defect prevention
  - Prevents defects from occurring in the first place
  - Activities: training, planning, simulation
- Defect detection
  - finds defects in a software artifact
  - Activities: inspections, testing, or measuring
- Defect removal
  - Isolation, correction, verification of fixes
  - Activities: fault isolation, fault analysis, regression testing
- Typical Activities of an SQA Process
  - Requirements validation
  - design verification
  - Static code checking
  - dynamic testing
  - Process engineering and standards
  - Metrics and continuous improvement



Software Development Lifecycle Models

- Sequential and Iterative development processes
- Continuous Integration (CI)
  - A software development process where a continuous integration server rebuilds a branch of source code ever time code is committed to the source control system
- Continuous Deployment
  - A software production process where changes are automatically deployed to production without any manual intervention
- Continuous Delivery
  - A software production process where the software can be released to production at any time with as much automation as possible for each step

Software Testing

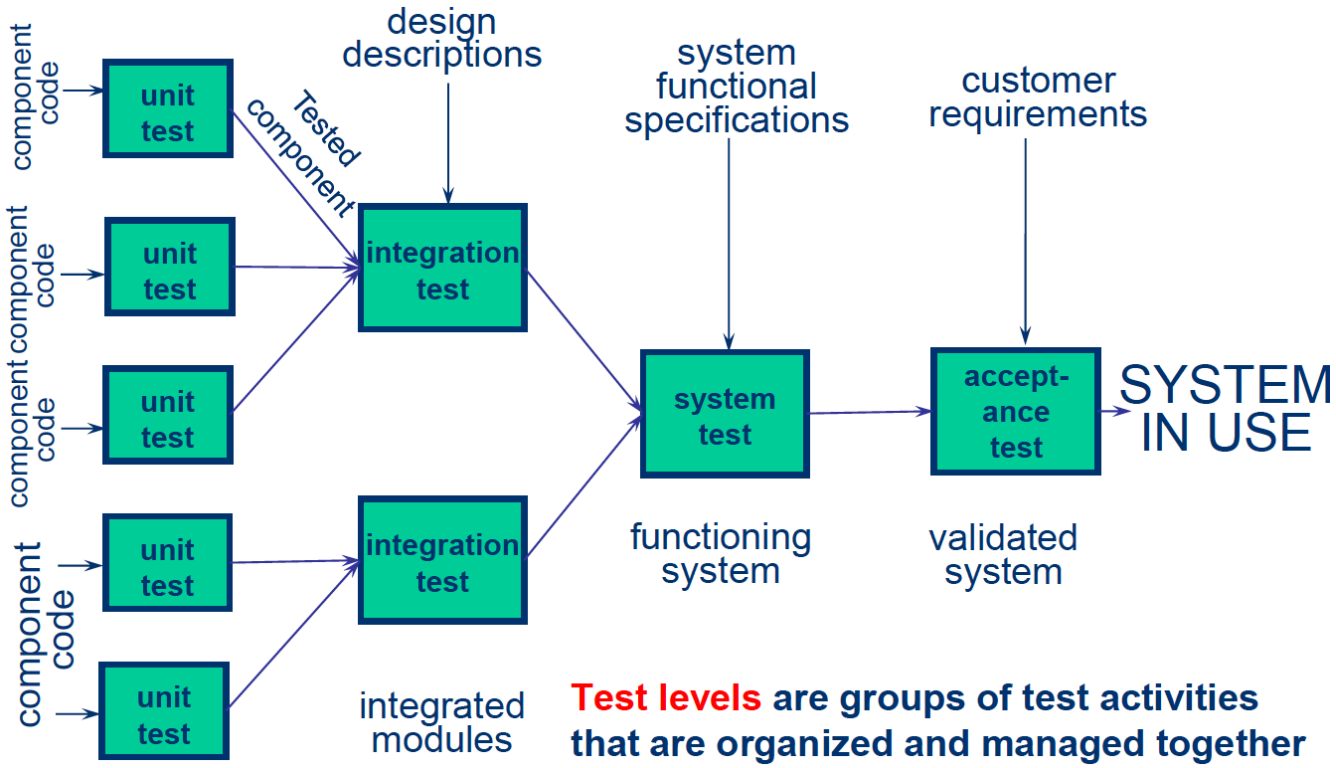
Why is Testing Difficult

- Upper limit to total number of test cases
- **continuity property** small differences in operating conditions will not result in dramatically different behavior → **not true in software!**

Seven Testing Principles

1. Program testing can be used to show the presence of bugs, but never their absence
2. Exhaustive testing is impossible
3. Early testing saves time and money
4. Defects cluster together
5. Pesticide Paradox: a system tends to build a resistance to a particular technique
6. Testing is context dependent
7. Absence of errors is a fallacy

Test Levels



Component/Unit Testing

Integration Testing

System Testing

Acceptance Testing

- User level: fitness for use by intended users
- Operational
- Contractual: check with respect to contract's acceptance criteria
- Alpha and Beta

Test Types

- Functional vs. non-functional
  - Functional: evaluate that the system performs with respects to requirements
  - non-functional: evaluate characteristics system as a whole
- Black-box vs. White-box
  - Black-box (functional)
  - White-box: aim to derive tests based on systems internal structure

Oracles and Test Coverage

Regression Testing

- Different **scopes**
  - Local: direct testing of the code changed or added
  - Surrounding: testing of function supported or directly impacted by the change
  - Confidence: predefined suite of tests routinely run after any change is made to product/system

Maintenance Testing

- Goal:
  - Carried after system is in production
  - Evaluate success + ensure lack of side effects
- Triggers for Maintenance
  - Modification
  - Migration
  - Retirement
- Impact analysis for maintenance

    - Evaluate the planned/execute changes

Test Activities and Processes

Test Planning

- Define objectives
- Define approach on how to meet test objectives within constraints

Test Monitoring and Control

- Compare actual progress against the test plan using monitoring metrics
- Evaluate exit criteria (check test results against coverage criterial)

Test Analysis

- BDD
- Determines what to test in terms of measurable coverage criteria
    - Identify testable features
    - Define and prioritize associated test conditions
    - Capture traceability
- Analyze test basis to identify testable features

Test Design

- Elaborate test condition s into high-level test cases
- Design and prioritize test cases
- Identify necessary data to support test conditions and cases
- Design test environment and identify required infrastructure and tools
- Capture traceability between test basis, test conditions, test cases, and test procedures

Test Implementation

- Do we now have everything in place to run the tests?
- Create test software for test execution
- Preparing test data and load into test environment
- Verify and update traceability between the test basis, test conditions, test cases, test procedures, and test suites

Test Execution

- Run test suites in accordance with test execution schedule

    - Execute
    - Compare results
    - Analyze anomalies
    - Report defects based on the observed failures
    - Log
    - Verify and update traceability

Test Completion

- Collect data from completed test activities to consolidate experience
- Occurs at project milestones
    - Check if all defect reports are closed
    - Create a test summary report
    - Analyze lessons learned

Importance of Traceability

- Bidirectional traceability between test basis and test work product
    - Analyze the impact of changes
    - Auditing and certification
    - Improve understandabilty of various test reports

Test Driven Development

- Listen → Test → Code → Design
- Listen to customers while gathering requirements, develop test cases, code the program, (re-)design / refactor / clean up as more code is added to the system

Test Automation

In Class Quiz

1. Valid objective for testing? find as many failures as possible so that defects can be identified and corrected

2. Difference between testing and debugging? testing shows failures caused by defects; debugging finds, analyzes, and removes the cause of failures in the software
3. Failure in testing or production? product crashed when the user selected and option in a dialog box
4. Which is a key principle of software testing? it is impossible to test all input and precondition combinations of a system
5. In what way can testing be a part of Quality Assurance? It reduces the level of risk to the quality of the system
6. Which of the following is performed during the test analysis activity? evaluating test basis for testability
7. How can white-box testing be applied during acceptance testing? To check if all work process flows have been covered

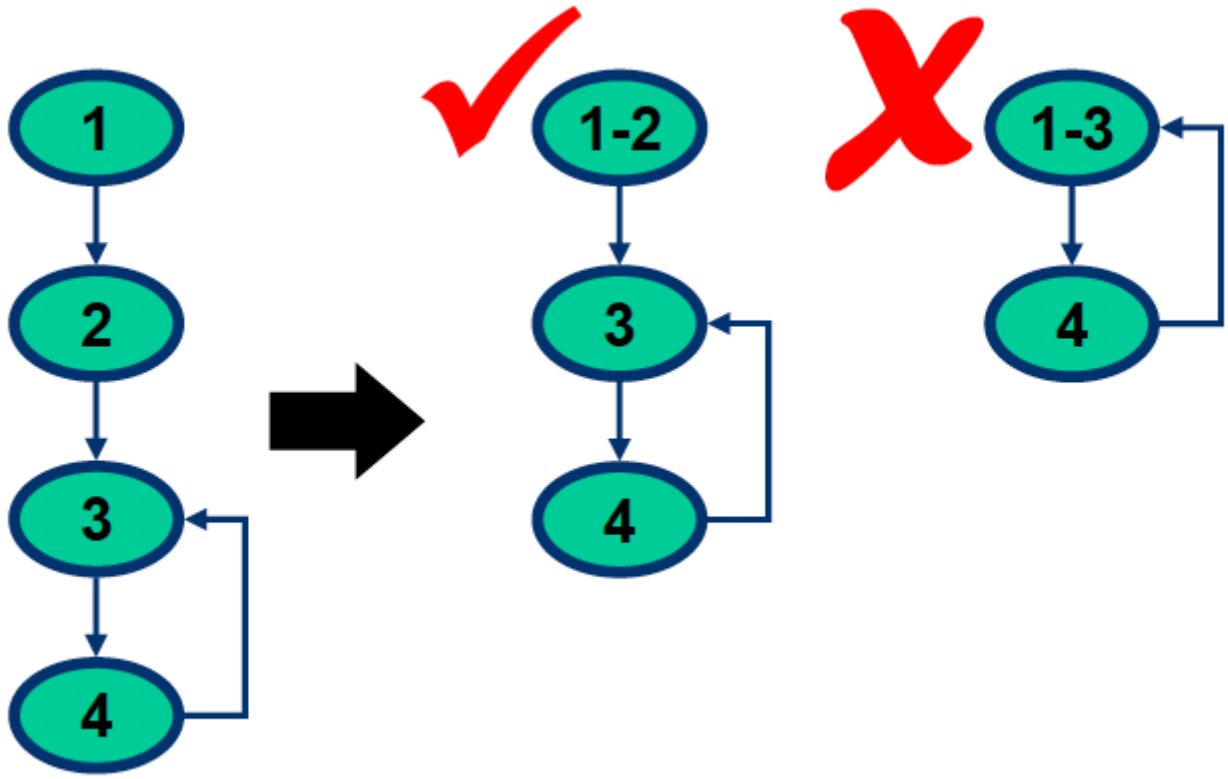
Static Validation and Verification Techniques

White Box Testing

What is White Box Testing

- Focus on system's internal logic
- Based on a system's **source code** as opposed to its specific implementation
- The notion of coverage can also be applied to structural (white-box) testing
- Advantages: Test what is actually there (source code)
- Disadvantages:
  - May miss functionality in specification that was not implemented
  - Can be cumbersome
- **Control Flow**-oriented approaches: based on the analysis of how control flows through a program
- **Data flow**-oriented approaches: based on the analysis of how data through the program
- **Mutation** testing: helps develop effective tests

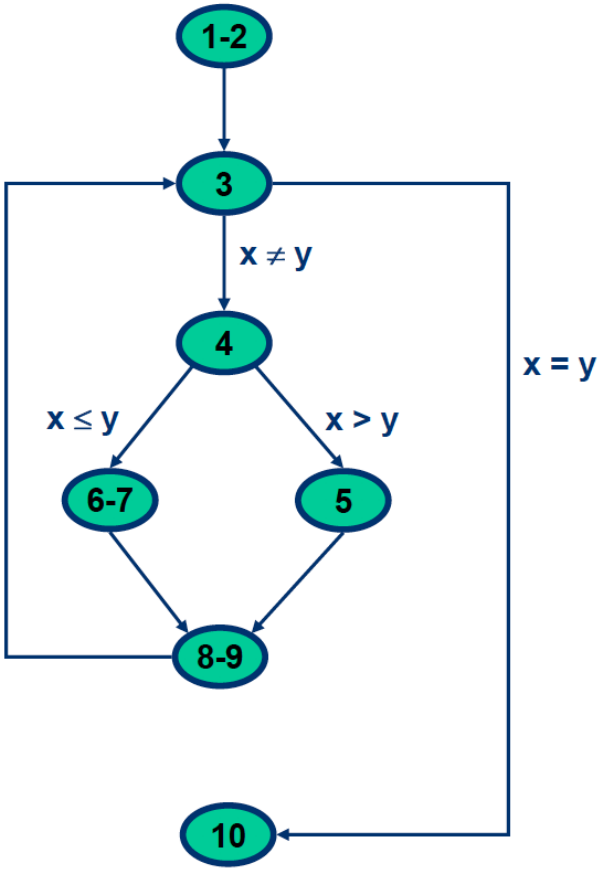
Control Flow Analysis



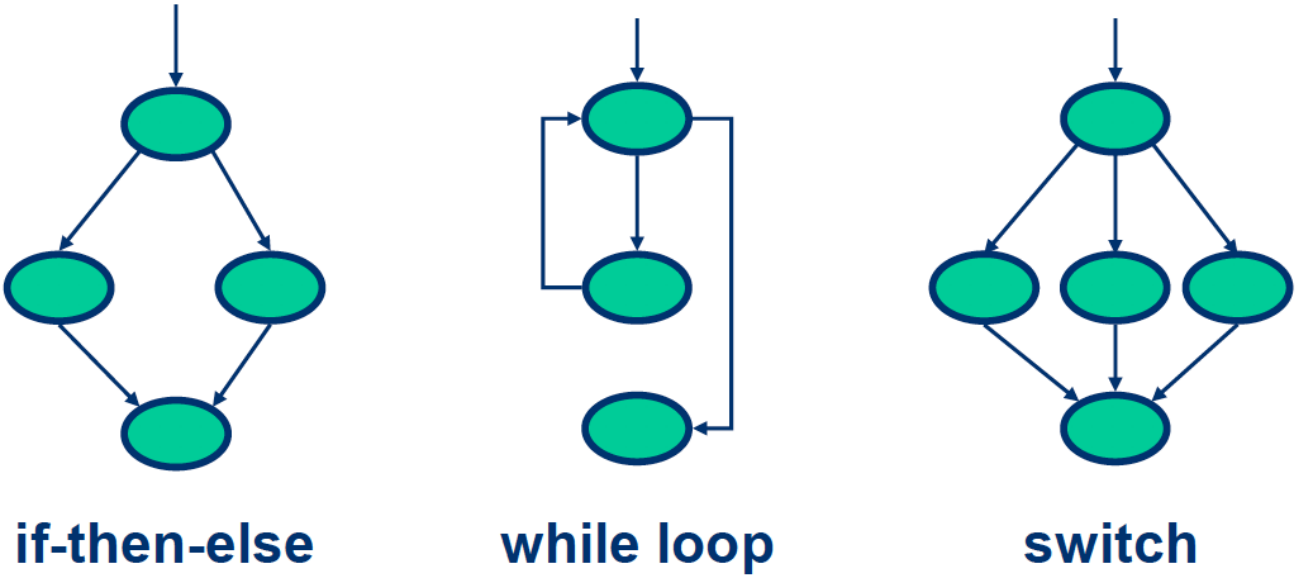
- Directed graph
  - Nodes are blocks of sequential statements
  - Edges are transfers of control
- Edges may be labeled with predicate representing the condition of control transfer
- Intermediate statements in a sequence of statements are not shown
  - As long as there is not more thank one exiting edge and one entering edge

Example Control Flow

```
 1 read(x);
 2 read(y);
 3 while x ≠ y loop
 4   if x>y then
 5         x := x - y;
 6   else
 7         y := y - x;
 8   end if;
 9 end loop;
10 gcd := x;
```



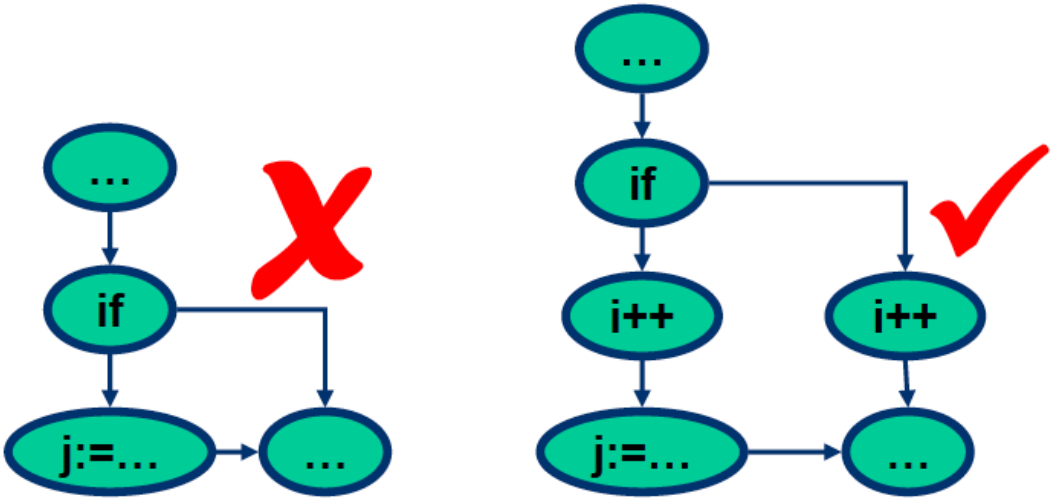
Control Flow Basics



- From source code to control flow graph—issue about branching
- In a control flow graph, nodes correspond to branching (if, while, ...) should not contain any assignment

Example

```
if (i++==1) {
  j := ...
}
```

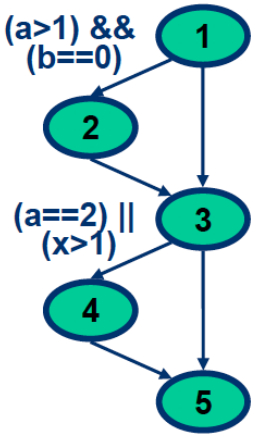


Statement/Node Coverage

- All instructions Executed
- Faults cannot be discovered if the parts containing them are not executed
- Equivalent to covering all nodes in control flow graph
- Executing a statement is a weak guarantee of correctness, but easy to achieve
- In general, several inputs execute the same statements

Example

```
int proc(int a,int b,int x) {
1   if ((a>1) && (b==0))
2      x = x/a;
3   if ((a==2) || (x>1))
4      x = x+1;
5   return x;
}
```



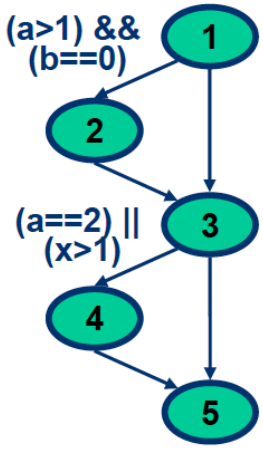
Path	a	b	x	Cond. 1	Cond. 2
1-2-3-4-5	2	0	-	T T → T	T - → T

Branch/Edge Coverage

- Use the program structure i.e. control flow graph
- Every decision branch is executed (sometimes called **decision coverage**)
- **Each edge** of control flow graph is traversed at least once
- Exercise all conditions that govern control flow programs at least once with true and once with false
- Minimum coverage - IEEE unit test standard

Example

```
int proc(int a,int b,int x) {
1   if ((a>1) && (b==0))
2      x = x/a;
3   if ((a==2) || (x>1))
4      x = x+1;
5   return x;
}
```

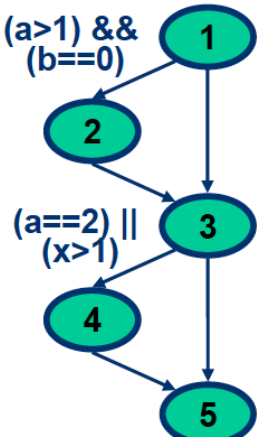


Path	a	b	x	Cond. 1	Cond. 2
1-2-3-4-5	2	0	-	T T → T	T - → T
1-3-5	0	-	1	F - → F	F F → F

Condition/Decision Coverage

- Each condition constituent evaluated true at least once and to false at least once
- **Combines** criteria for condition and branch coverage

```
int proc(int a,int b,int x) {
1   if ((a>1) && (b==0))
2      x = x/a;
3   if ((a==2) || (x>1))
4      x = x+1;
5   return x;
}
```

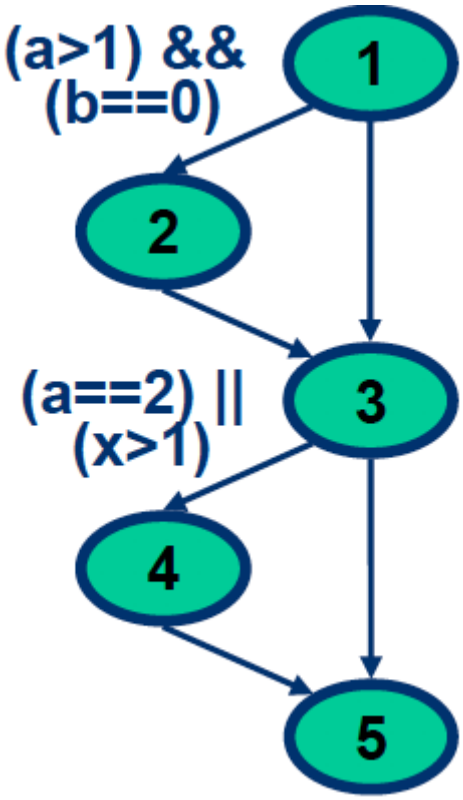


Path	a	b	x	Cond. 1	Cond. 2
1-3-4-5	1	0	2	F T → F	F T → T
1-3-4-5	2	1	1	T F → F	T F → T

Multiple Condition Coverage

- All Combinations of condition constituents in decisions

```
int proc(int a,int b,int x) {
1   if ((a>1) && (b==0))
2       x = x/a;
3   if ((a==2) || (x>1))
4       x = x+1;
5   return x;
```



Path	a	b	x	Cond. 1	Cond. 2
1-2-3-4-5	2	0	2	T T → T	T T → T
1-3-4-5	2	1	1	T F → F	T F → T
1-3-4-5	1	0	2	F T → F	F T → T
1-3-5	1	1	1	F F → F	F F → F

Modified Condition/Decision Coverage

- Full MC/DC coverage achieved if:
  - Each entry and exit point invoked

    - Each decision takes every possible outcome
    - Each condition in a decision takes every possible outcome (true/false)
- Consequences:
    - There exists a pair of test cases where only the one condition changes and the outcome changes too
    - Requires n+1 test cases for one decision with n conditions

Example

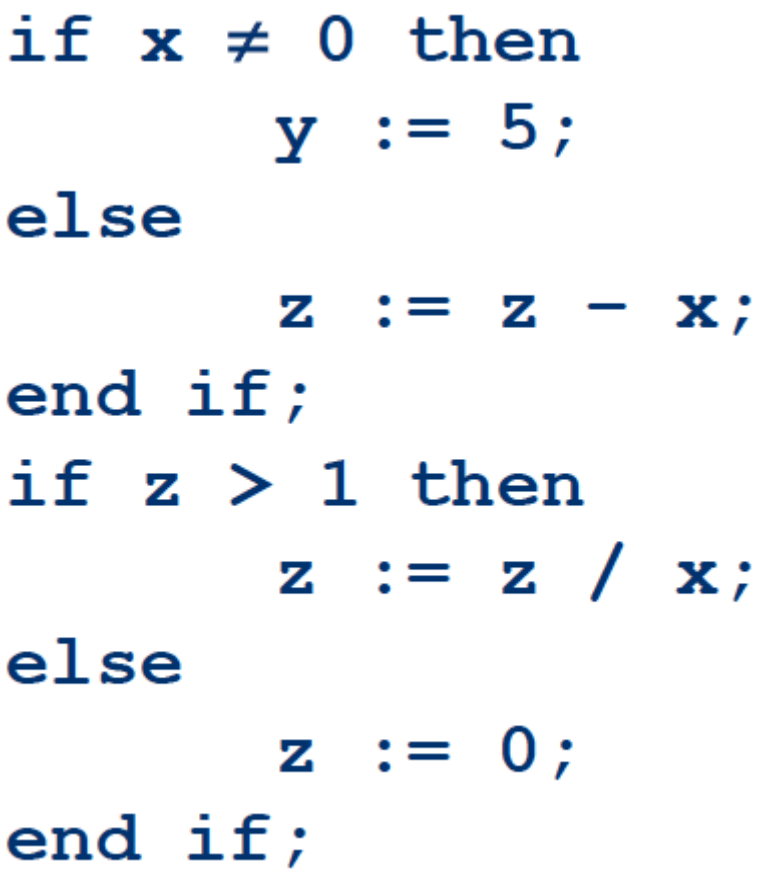
- Assume four test cases with values for conditions A, B, and C as well as the corresponding result
- Pair 1+3: only A changes its value and result changes too
- Pair 1+2 shows independence of B
- Pair 1+4 shows independence of C
- 4 test cases needed for modified condition/decision coverage for three conditions

	A	B	C	Result
Test Case 1	TRUE	TRUE	FALSE	FALSE
Test Case 2	TRUE	FALSE	FALSE	TRUE
Test Case 3	FALSE	TRUE	FALSE	TRUE
Test Case 4	TRUE	TRUE	TRUE	TRUE

Path Coverage

- Test case for each possible path
- In practice, however, the number of paths is too large if not infinite
- Some paths are infeasible
- It is key to determine "critical paths"

Example



- T1 = {<x=0, z=1>, <x=1, z=3} (executes all edges but does not show risk of division by zero)
- T2 = {<x=0, z=3>, <x=1, z=1>} (would find the problem by exercising the remaining possible flows of control through the program)
- T1UT2 → all paths covered

Path	a	b	x	Cond. 1	Cond. 2
1-2-3-4-5	2	0	-	T T → T	T - → T
1-2-3-5	3	0	1	T T → T	F F → F
1-3-4-5	1	1	2	F F → F	F T → T
1-3-5	1	1	1	F F → F	F F → F

Loop Coverage

- Minimal Coverage should when possible, execute the loop body:
  - Zero times
  - once
  - Twice or more
- Single loop → more extensive coverage, set loop control variable:
  - Minimum -1, minimum, minmum +1

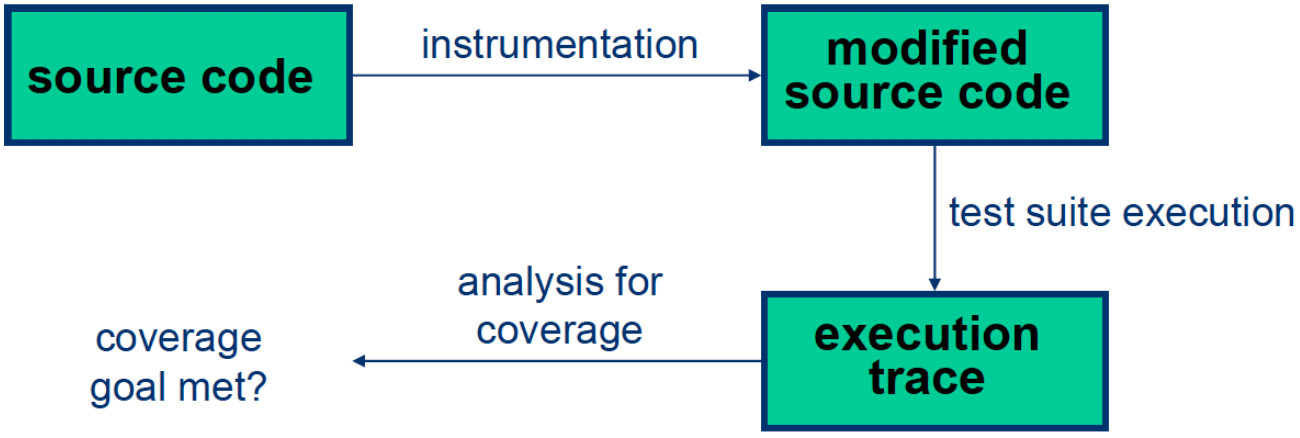
    - Typical
    - Maximum -1, maximum, maximum + 1
- Nested Loop:
    - Start at innermost loop → proceed to outermost
    - For the current focus loop → Proceed by moving outwards
        - Set the outer loops to their minimum values
        - Set all inner loops to their typical values
        - Test cases for a single focus loop
        - Move up one level in nested loop

White-Box Testing Process

1. Set Coverage goal
2. Derive control flow graph (CFG) from source code
3. Determine paths to obtain coverage goal
4. For each path
    a. Sensitize path for input values
    b. Use specification (or oracle) for expected output
    c. Watch for infeasible paths
5. Run test cases
6. Check Coverage

Path Instrumentation

- To measure code coverage:



- Approaches:
    - Link markers, link counters, symbolic debugger, code coverage tool

Path Sensitization

- To find a set of inputs to force a selected path:
  - Backward strategy (from exit to entry)
  - Forward strategy (from entry to exit)
- Problem with infeasible paths (may call for rewriting of program)
  - Not all statements are usually reachable in real world programs
  - It is not always possible to decide automatically if a statement is reachable and the percentage of reachable statements
- When one does not reach 100% coverage it is difficult to determine the reason

Path Condition

- Conjunction of branch predicates required to hold for all the branches along a path
- Used to find:
  - Values for a path (sensitizing)
  - Infeasible paths
- Determined using symbolic evaluation
  - Variables take symbolic values (e.g. x_0, x_1, ... ,)

Symbolic Values

WARNING	Do lecture exercises

Data Flow Analysis

- CFG paths that are significant for the data flow in the program
- Focusses on the assignment of values to variables and their uses i.e. where data is defined and used
- Analyze occurrences of variable
- **Definition of Occurrence** value is bound to variable
- **Use Occurrence** value of variable is referred
  - Predicate use: variable used to decide whether predicate is true
  - Computational: use compute a value for defining other variable or output value

Definitions and Uses

- A program written in a programming language such as C and Java contains variables
- Variables are defined by assigning values to them and are used in expressions

Pointers

- Consider the following sequence of statements that use pointers:

```
z = &x; //defines a pointer variable z but does not use x
y = z+1; //defines y and uses z
*z = 25; //defines x accessed through the pointer variable z
y = *z + 1; //defines y and uses x accessed through pointer variable z
```

Arrays

- Arrays are also tricky - consider the following declaration and two statements in C:

```
int A[0];
A[i] = x + y
```

- First statement defines variable A
- Second statement defines A and uses i, x and y
- Alternate view for second statement: defines A[i] not the entire array

c-use

- Uses of a variable that occur within an expression as a part of an assignment statement, in an output statement, as a parameter within a function call, and in subscript expressions, are classified as c-use, where the "c" in c-use stands for computational
- In general, a definition of A[E] is interpreted as a c-use of variables in E followed by a def of A
- In general, a reference to A[E] is interpreted as a use of variables in E followed by a use of A

NOTE	c-use example question on slide 44

p-use

- The occurrence of a variable in an expression used as a condition in a branch statement such as an if and while, is considered as a p-use, where the "p" in p-use stands for a predicate

NOTE	p-use example question on slide 45

Basics of Data Flow Analysis

- Variable definition
  - d(v,n): value is assigned to v at node n (e.g. LHS of assignment, input statement)
- Variable use
  - c-use(v,n): variable v used in a computation at node (e.g. RHS of assignment, argument of procedure call, output statement)

- p-use(v,m,n): variable v used in predicate from node M to n (e.g as part of and expression used for a decision)

- Variable kill:
  - k(v,n): variable v deallocated at node n

Example

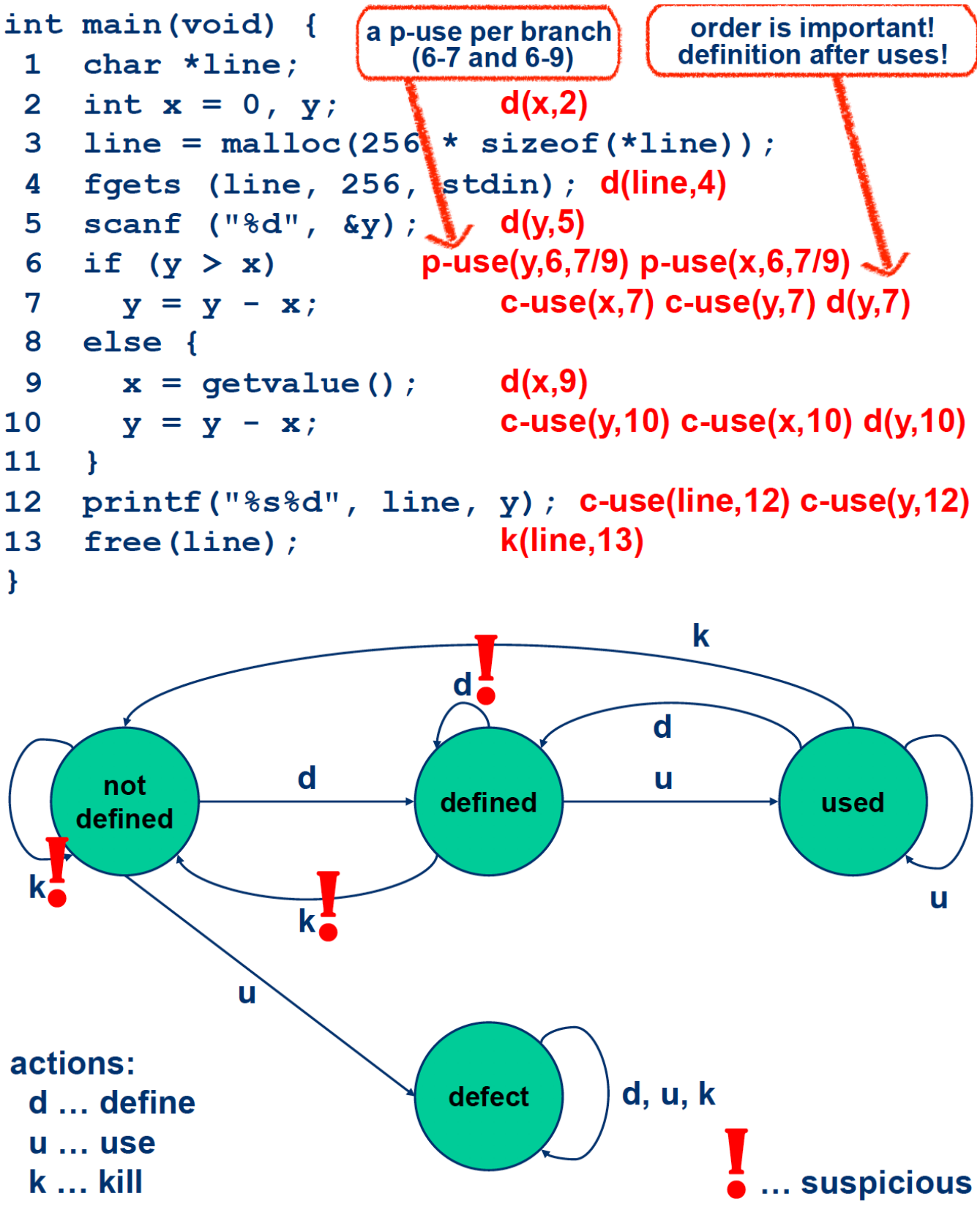


Table 1. Data Flow Actions Checklist: Pairs of Actions

Successive Actions	Result
dd	suspicious
dk	probably defect
du	normal case
kd	okay
kk	probably defect
ku	a defect
ud	okay
uk	okay
uu	okay

Table 2. Data Flow Actions Checklist: First Occurrence

First Action	Result
k	suspicious
d	okay
u	suspicious

Table 3. Data Flow Actions Checklist: First Occurrence

Last Action	Result
k	okay
d	suspicious
u	okay

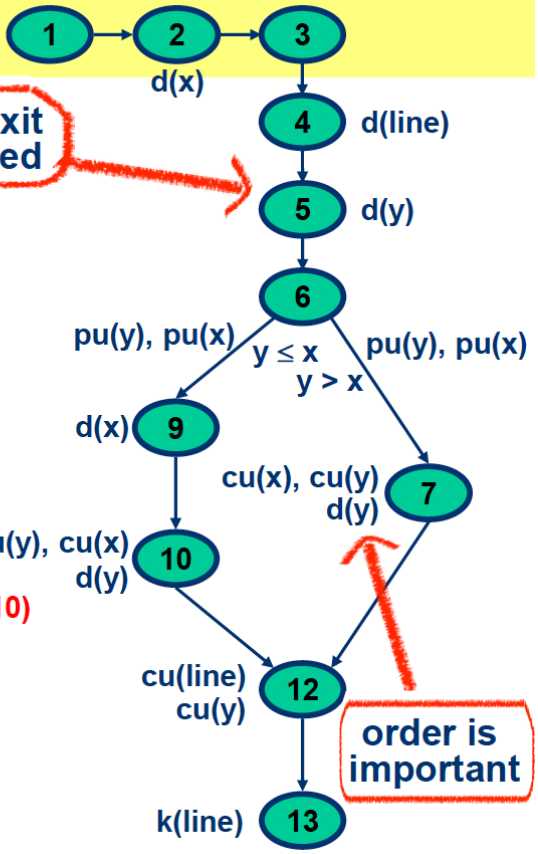
Data Flow Graph

- A data flow graph of a program captures the flow of definitions and uses across basic blocks in a program
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths in control flow graph are preserved in the data flow graph
- Annotate each node with def and c-use as needed and each edge with p-use as needed
- Label each edge with the condition which when true causes the edge to be taken

Data Flow Graph

cannot collapse a series of single input/exit nodes into one node if a node is annotated

```
int main(void) {
 1  char *line;
 2  int x = 0, y;          d(x,2)
 3  line = malloc(256 * sizeof(*line));
 4  fgets (line, 256, stdin);  d(line,4)
 5  scanf ("%d", &y);      d(y,5)
 6  if (y > x)             p-use(y,6,7/9) p-use(x,6,7/9)
 7    y = y - x;           c-use(x,7) c-use(y,7) d(y,7)
 8  else {
 9    x = getvalue();      d(x,9)
10    y = y - x;           c-use(y,10) c-use(x,10) d(y,10)
11  }
12  printf("%s%d", line, y);   c-use(line,12)
                               c-use(y,12)
13  free(line);            k(line,13)
}
```



node i	def(i)	c-use(i)	edge(i,j)	p-use(i,j)
2	x		(2,3)	
4	line		(4,5)	
5	y		(5,6)	
6			(6,7)	y,x
6			(6,9)	y,x
7	y	x,y	(7,12)	
9	x		(9,10)	
10	y	y,x	(10,12)	
12		line,y	(12,13)	

Variable	Define	Use	Comments
x	2		
		6-7, 6-9	p-use
		7	c-use
	9		
		10	c-use
line	4		
		12	c-use
y	5		
		6-7, 6-9	p-use
		7	c-use
	7		
		10	c-use
	10		
		12	c-use

Data Flow Graph: Paths and Pairs

- Complete path: initial node is start node, final node is exit node
- Simple path: all nodes except possibly first and last node are distinct
- Loop free path: all nodes are distinct
- def-clear path with respect to v: any path
  - starting from a node at which variable v is defined and
  - ending at a node at which v is used
  - without redefining v anywhere else along the path
- du-pair with respect to v: (d, u)

    - d … node where v is defined
    - u … node where v is used
    - def-clear path with respect to v from d to u
- Definition-use path (du-path) with respect to v: a path P=<n_1, n_2, …, n_j, n_k> such that d(v, n_1) and either one of the following two cases:
    - c-use of v at node n_k, and P is a def-clear simple path with respect to v (i.w. at most a single loop traversal)
    - p-use of v on edge n_j to n_k, and <n_1,n_2, …,n_j> is def-clear loop-free path (i.e. cycle free)

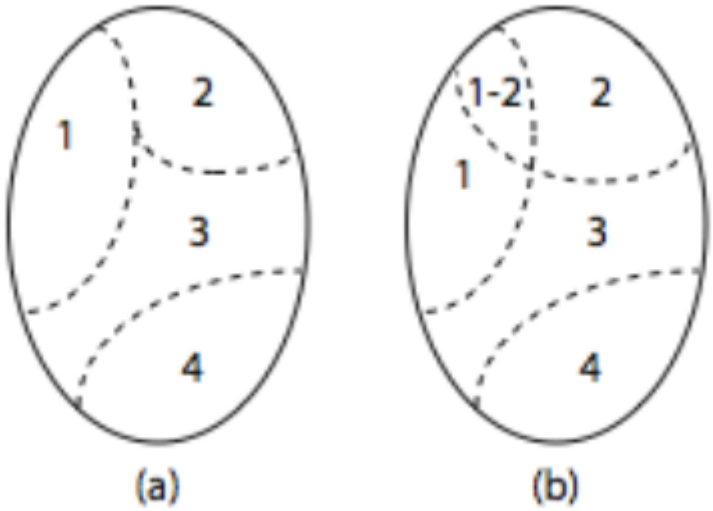
WARNING	A shit load of examples on all this stuff in lectures slide 57-66

Black-Box Component Testing

- Based on system's specification as opposed to its structure

Equivalence Partitioning

- Equivalence Classes (ECs): partition of the input set according to specification
- Entire input set is covered: completeness (a)
- Disjoint Classes: avoid redundancy (b)



Weak/Strong EC Testing

- Weak equivalence class testing (WECT): choose one variable from each equivalence class
- Strong equivalence testing (SECT): based on the Cartesian product of the partition subsets i.e. test all class interactions

EC: Discussion

- If error conditions are a high priority we should extend strong equivalence testing to include both valid and invalid inputs
- Equivalence partitioning is appropriate when input data is defined in terms of ranges and sets of discrete values
- SECT makes the assumption that values are independent

Heuristic for Identifying EC

- For Each external input
  1. If input is a range of valid values
     a. One valid EC within range
     b. Two invalid EC (one outside each end of the range)
  2. If input is a number (N) of valid values, define:
     a. One valid EC
     b. Two invalid ECs
  3. If input is an element from a set of valid values, define:
     a. One valid EC (withing set)
     b. One invalid EC (outside set)
  4. If input is a "must be" situation, define:
     a. One valid EC (satisfies)
     b. One invalid EC (does not satisfy)
  5. If there is a reason to tbelieve that elements in an EC are not handled in an identical manner by the program"
     a. subdivide EC into smaller ECs
  6. Consider creating equivalence partitions that handle the default, empty, blank, null, or none conditions

Myer’s Test Selection Approach

1. Until all valid ECs have been covered by test cases:
   a. Write da new test case that covers as many of the uncovered valid ECs as possible
2. Until all invalid ECs have been covered by test cases:
   a. Write a test case that covers one, and only one, of the uncovered invalid ECs

Boundary Value Analysis

- Testing focussed on the edge of the planned operational limits of the software

BVA: Guidelines

- Use input variables within a class at: min, min+m nom, max-, max
- Hold the values of all but one variable at their nominal value, letting one assume its extreme values
- For each boundary condition:
  - Include boundary value in at least one valid test case

General Case and Limitations

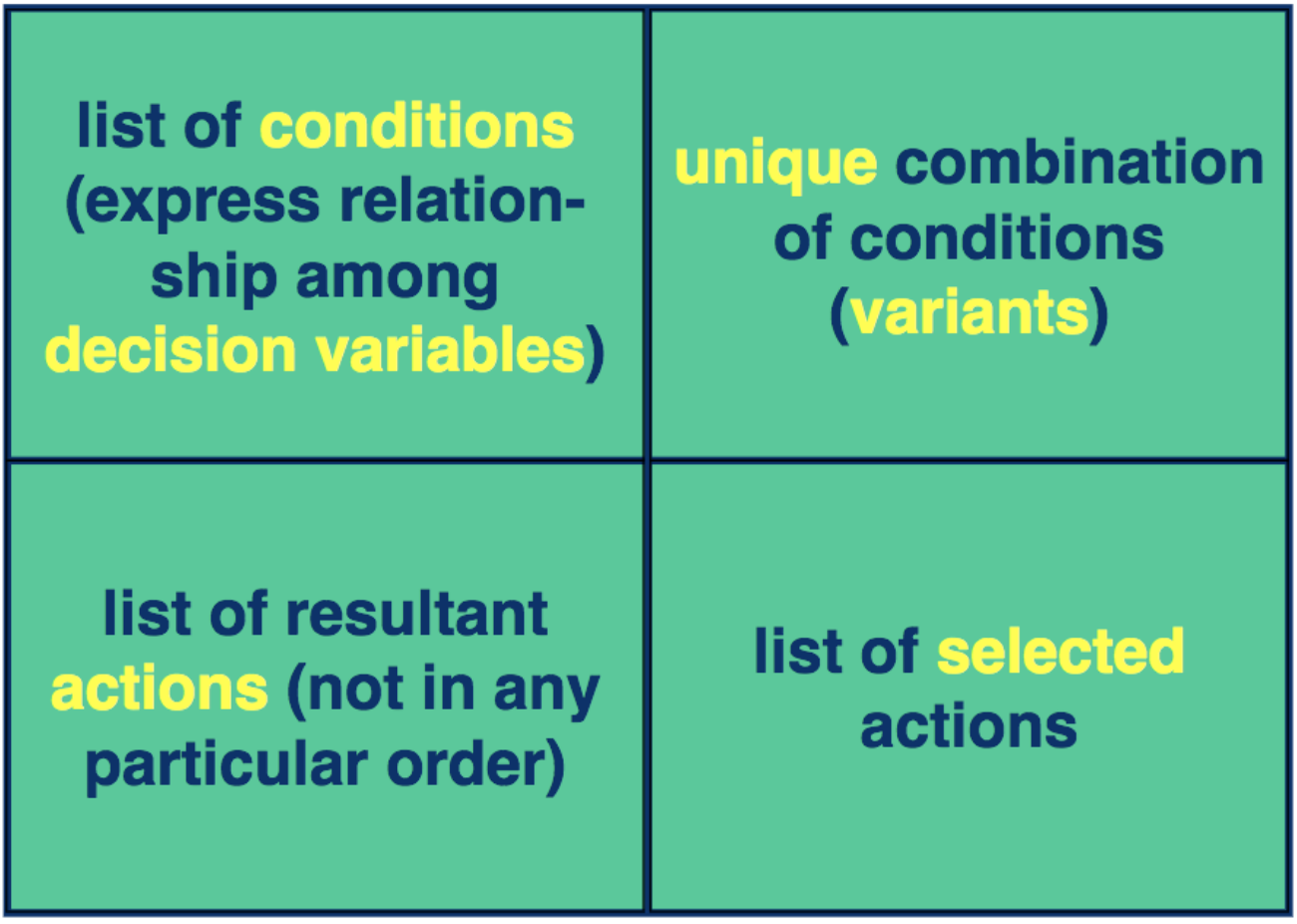
- A function with n variables will require 4n+1 test cases
- Technique to increase Robustness Testing (6n + 1 test cases)
  - For each boundary condition additionally include the value just beyond the boundary in at least one invalid test case

Worst Case Testing

- Test for when more than one variable has an extreme value (5^n cases)

Decision Tables

Decision Table: Structure



Decision Table: Format

	1	2	3	4	5	6
condition c1	T			F		
c2	T		F	T		F
c3	T	F	-	T	F	-
action a1	X	X		X		
a2	X				X	
a3		X		X	X	
a4			X			X

Decision Table: Don't Care

- Don't care decision variables reduce the number of variants
- Don't care can correspond to:
  - The inputs are necessary but have no effect
  - The inputs may be omitted
  - Mutually exclusive cases

Test Generation Strategies for Decision Tables

- All-Explicit Variants: a test case for each explicit don't care taken into account
- All-variants: a test case for each implicit variant 2^n for n conditions
- All-true: a test case for each variant with an outcome
- All-false: a test case for each variant without an outcome

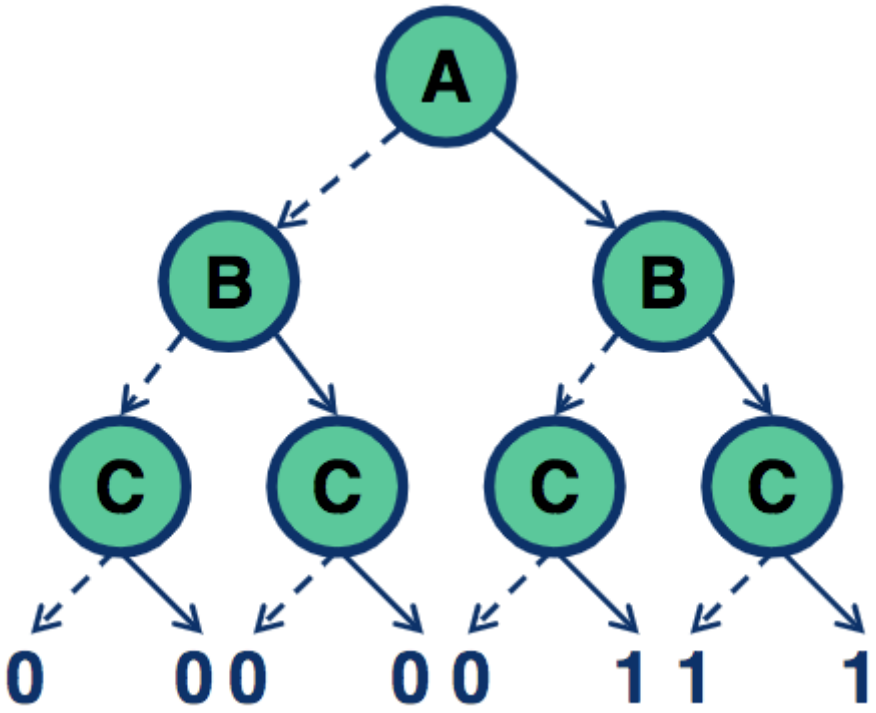
Binary Decision Diagrams

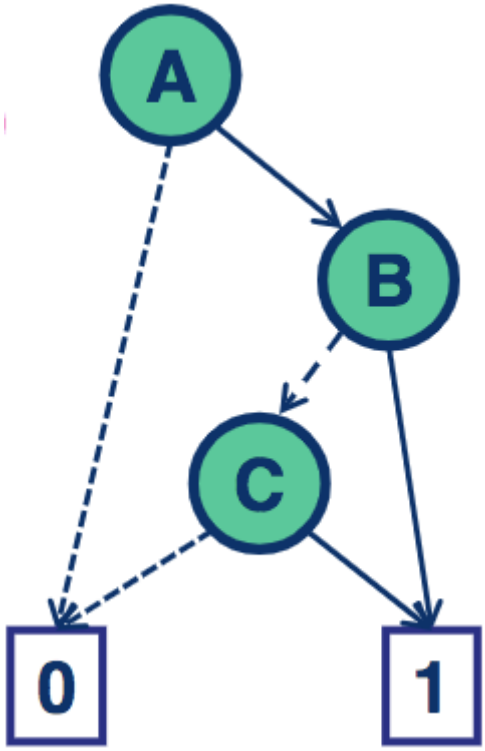
Binary Decision Trees

- Each level of the tree corresponds to a decision with respect to a boolean variable
  - False-branch: dashed
  - True-branch: solid
  - Each leaf represents the resultant value for condition on the path from the root to the leaf

Reduced Ordered Binary Decision Diagrams (ROBDD)

- Compact representation of a decision table
  - Irredundancy: F and T successors of every node are distinct
  - Uniqueness: There are no two nodes testing the same variable with the same successors



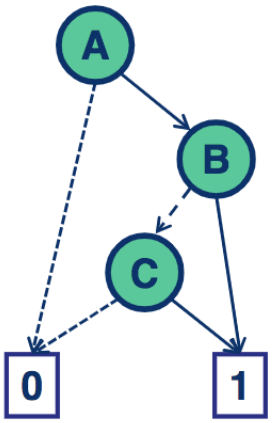


ROBDD Checklist

- Define and ordering of variables
- A ROBDD is a directed acyclic graph with one root and two leaf nodes
- Make decisions in that order along all paths
- Remove Redundant nodes
- Merge isomorphic subtrees

Test Cases from ROBDDs

1. Map ROBDD into ROBDD determinant table
2. Generate ROBDD test suite for each determinant



BDD Variant	A	B	C	Z (action)
1	F	-	-	
2	T	F	T	X
3	T	F	F	
4	T	T	-	X

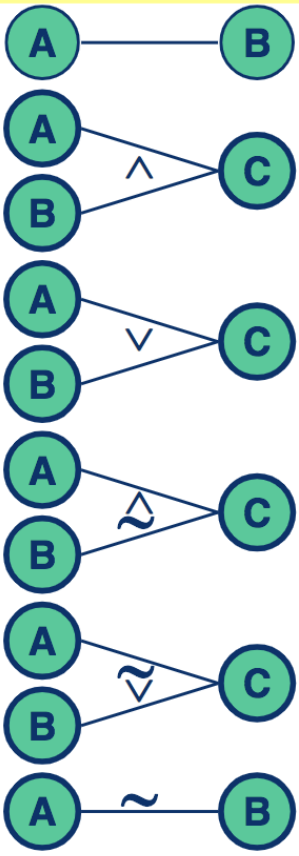
BDD test suite

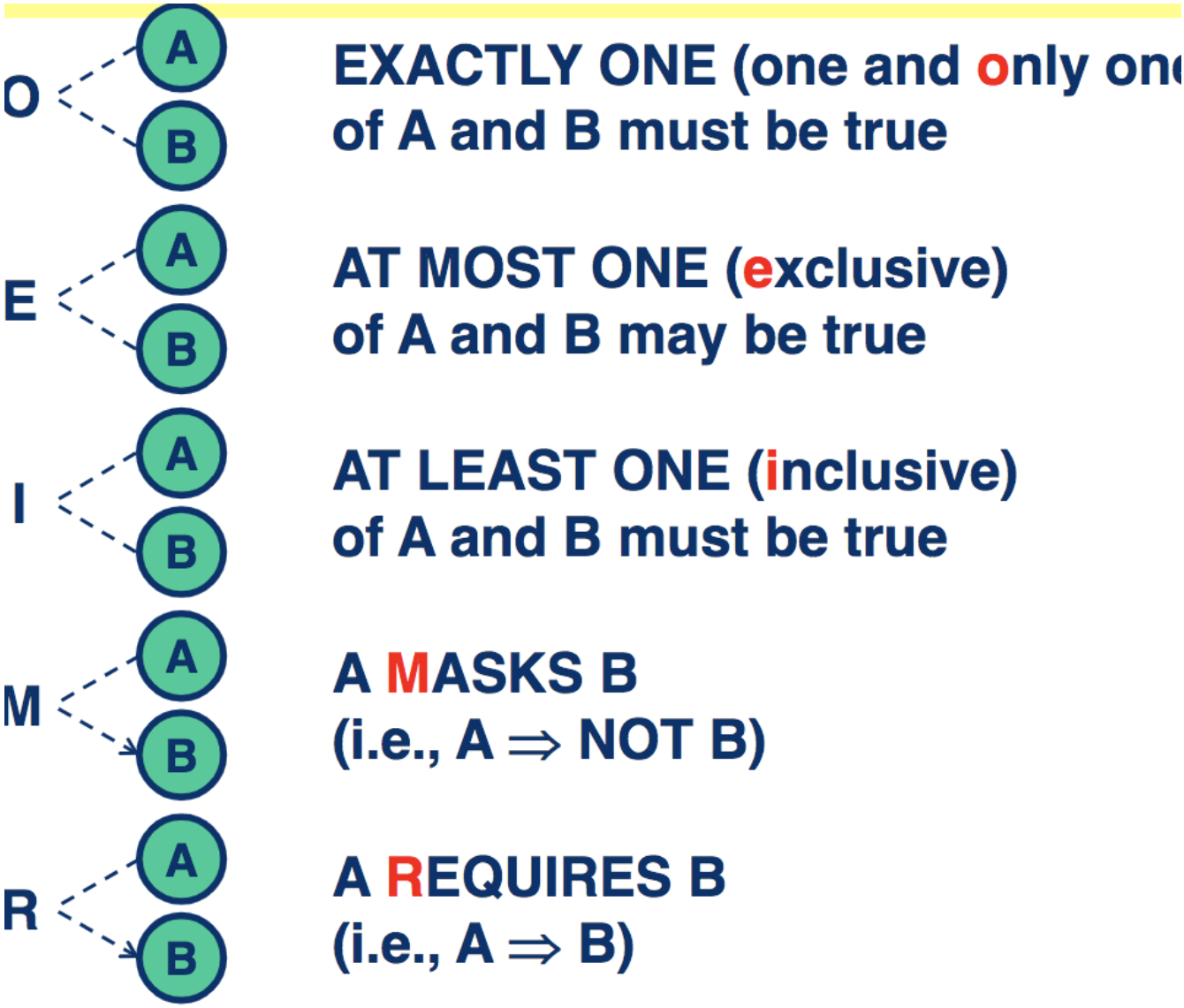
Cause-Effect Modeling

- Technique that helps derive decision tables and generate boolean formulas to yield test cases
- Causes must be boolean expressions

Cause-Effect Graph

- A node is drawn for each cause and effect: nodes are places on opposite sides of the sheet
- A line from a cause tot an effect indicates that the cause is a necessary condition for the effect
- If a single effect has two or more causes, the logical relationship of the causes is annotated by symbols for logical and and logical or
- A cause whose negation is necessary is shown by a logical not
- A single cause may be necessary for many effects
- Intermediate nodes may be used to simplify the graph and its construction


A
B
if A then B
A
B
∧
C
AND: if (A and B) then C
A
B
∨
C
OR: if (A or B) then C
(symbol sometimes omitted)
A
B
C
A
B
≈
C
NAND: if not (A and B) then C
A
B
≈
∨
C
NOR: if (neither A nor B) then C
A
~
B
NOT: if (not A) then B



Test Generation from Cause and Effect Graphs

- Divide specification into workable pieces
- Identify causes and effects from the specification
- Annotate model with constraints describing impossible combinations of causes and/or effects
- Link causes and effects
  1. Use model to Generate a limited entry decision table and convert each column of table into a test case
  2. Use model to derive Boolean formulas and generate test cases from logic formulas

Deriving a Decision Table

- A row for each cause or effect
- A column corresponds to a test variant
- Work with a single effect at a time
  - Set effect to true

    - look backward for all combinations of inputs which will force effect to be true subject to constraints
- Create a column for each possible combination of causes
- For each combination, determine state of other effects

Deriving Logic Formulas

- One predicate exists for each effect (output variable)
- If several effects are present then the resulting test is a composite of several predicates that happen to share decision/input variables and effects/actions
    1. Generate initial function from cause-effect graph
        a. Start from effect node then backtrack through graph
        b. Substitute higher level clauses with lower level clauses and Boolean expressions
    2. Transform into minimal, DNF form
        a. Use boolean algebra laws to reduce Boolean expressions
        b. re-express in sum of products form

Test Generation Strategies for Logic Formulas

Each-condition/all-condition

- Derive the set of variants following these rules
    - For each variable, include a variant such that the variable is made true with all other variables being false
    - One variant such that all variables are true, or one variant such that all variables are false

Variable Negations

- Strategy designed to reveal faults resulting from dont care implementation
- Unique true points: one variant for each product term such that this variant makes the product term true but no other product term is true
- Near false points: one variant for each literal such that this variant makes the whole function false and, if the value for the literal is negated, the whole function is true

Full Predicate Coverage

- Clause: boolean expressions that contains no boolean operators
- Predicate: boolean expression that is composed of a clauses and zero or more boolean operators
- Full predicate coverage: tries to determine whether each clause in a predicate is necessary and formulated correctly

08 System Testing

Overview of System Testing

- Performed after the software has been assembled
- Check if the system satisfies requirements
- High-order testing criteria should be expressed in the specification in a measurable way
- Acceptance Tests
  - System tests carried out by customers or under customers' responsibility
  - Verifies if the system works according to customers' expectations
- Common types of user acceptance tests:
  - Alpha testing - end user testing performed on a system that may have incompatible features
    - Within the development environment
    - Performed by an in house testing panel including end users
  - Beta testing - an end user testing performed within the user environment

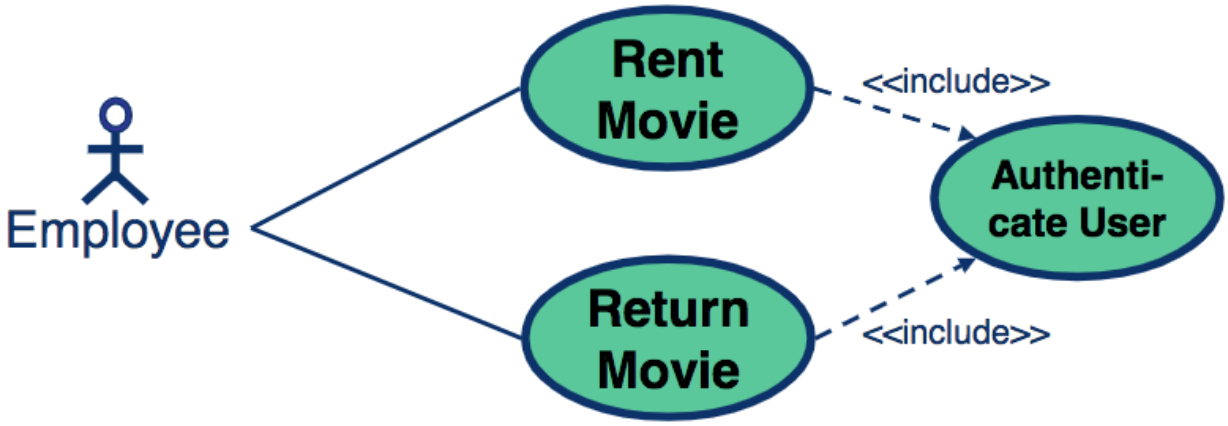
System Level Functional (Black-box) Testing

Functional Testing

- Ensure the system meets functional requirements
- Test cases derived from statement of functional requirements:
  - Natural language requirements
  - Use cases
  - Models

Test Case Derivations from Use Cases

- For each use case
  1. Develop a scenario graph
  2. Determine all possible scenarios
  3. Analyze and rank scenarios
  4. Generate test cases from scenarios to meet coverage goal
  5. Execute test cases



Scenario Graph

- Generated from a use case
- A node corresponds to a point where we are waiting for the next event to occur
- There is a single starting node
- End of use case is finish node
- An edge corresponds to a an event occurrence (may include conditions, special looping edge)
- Scenario = path from starting node to a finishing node

Grey-Box Testing

- Testing strategy based on limited knowledge of system internal
- Coverage is based on models

Testing vs. Formal Verification

Testing

- Scope
  - A test case/simulation run analyses one execution trace of the system
- Characteristics
  - Can detect errors
  - Cannot guarantee/prove absence of errors
- Cost
  - Expensive to design
  - Cheap to execute

Formal Verification

- Scope

    - Exhaustively analysis all possible execution traces of the system
- Characteristics
    - Can detect errors
    - Can guarantee/prove absence of errors
- Cost
    - Expensive to design
    - Expensive to execute

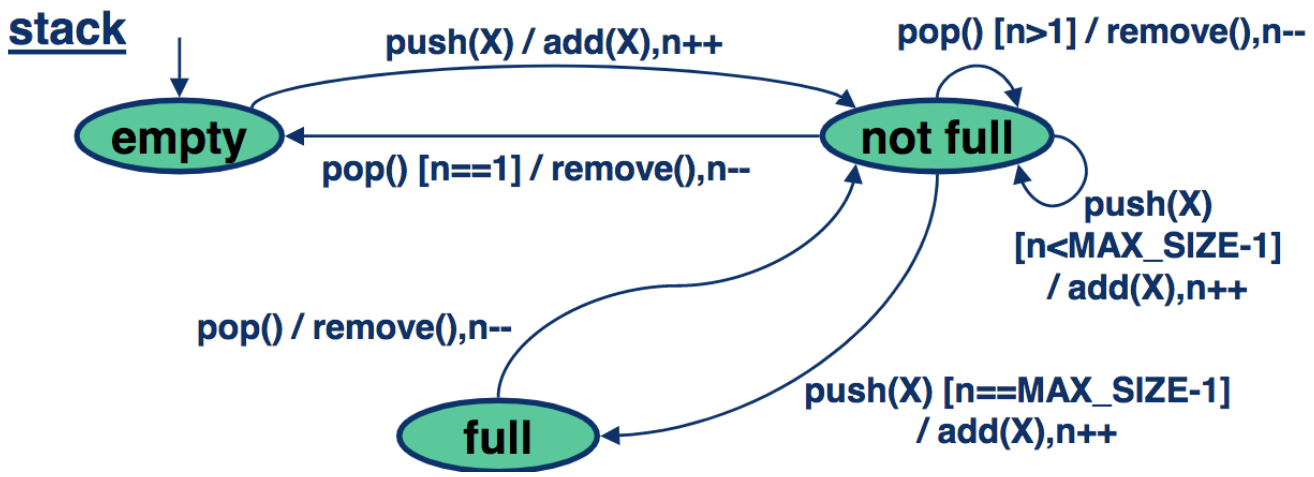
Concrete vs Abstract States

- Concrete
    - Combination of possible values of attributes
    - Can be infinite
- Abstract state
    - Predicates over concrete states
    - One abstract state ← many concrete states
    - Hierarchal status

State Machines

image::images/ECSE429LectureNotes-098f3.png[align=center

State-based Testing: Overview



- Objective: check conformance of implementation with design model

Equivalent States

- Two states are equivalent if for every input sequence, the output sequences from the two states are the same

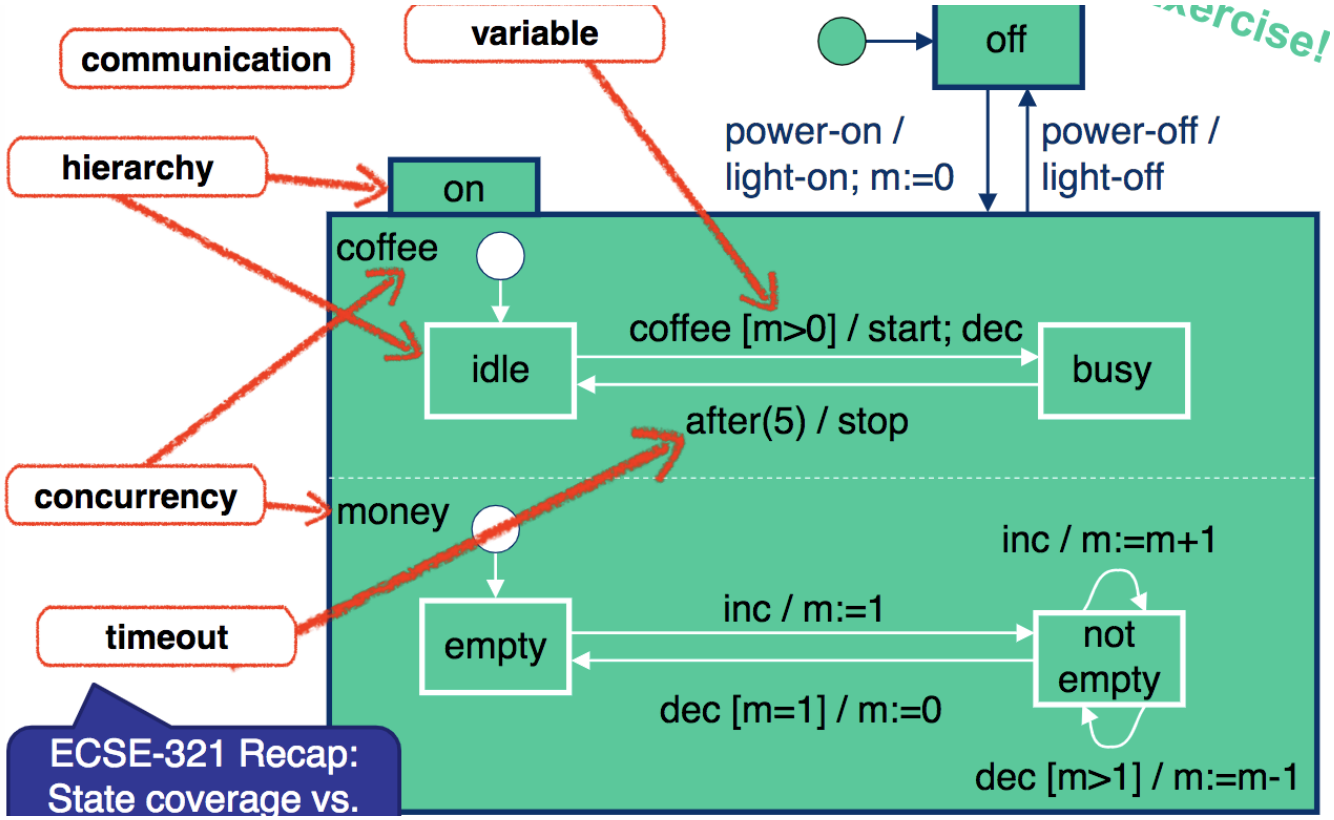
Tests for State-Based models

- State machine execution depends on input events and data
- Data flow testing methods → for data dependencies
- A test consists of a test sequence and test data

Challenges for State-based Tests

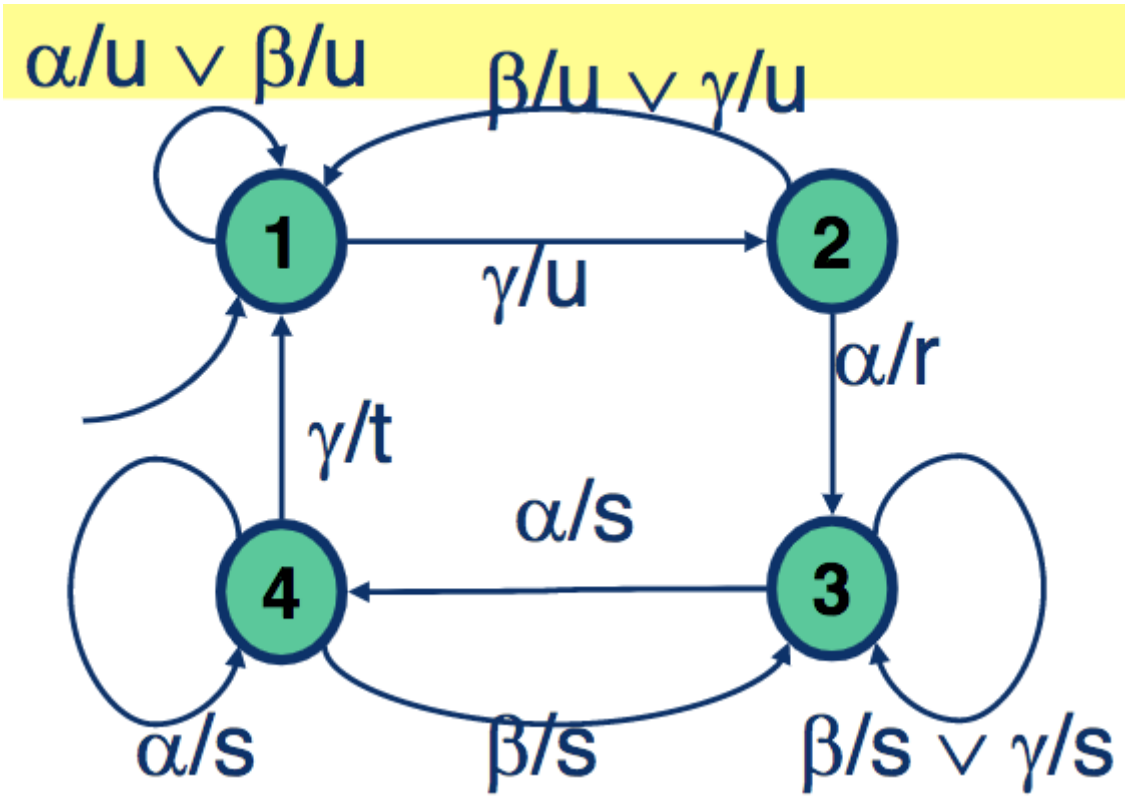
- Executability problem: find data to execute the test sequence
- Scalability problem: large number of concrete states
- Missing state model: need to revers-engineer model

Example



Fault Model for State Machines

Fault Model



- Missing or incorrect transition based on a correct input (**transfer fault**)
  - If transition from 2 to 1 on input *B* is missing
  - If transition from 1 to 2 (on input *y*) is from 1 to 3 instead
- Missing or incorrect output (action), based on a correct input and transition (**output fault**)
  - If transition from 1 to 2 (on input *y*) outputs r instead of u
- Corrupt state: based on a correct input, the implementation computes a state that is not valid (additional state)
  - If transition is from 1 to 5 (on input *y*) instead of 2
- Sneak path (extra transition): the implementation accepts a valid input that is illegal or unspecified for a state
  - Example state machine is fully specified → sneak path not possible
- Illegal input failure: the implementation fails to handle an illegal mesage correctly (incorrect output, state corrupted)
- Trap door: implementation accepts undefined messages
  - Transition from 1 to 4 on input

State-Based Testing Strategies

- All States: Testing passes through all states
- All Events: all events consumed at least once
- All Actions: all actions produced at least once
- All Transitions: All (explicit) transitions taken at least once

    - Implies all states/events/actions
    - All n-transitions sequences: sequences of length n

N+ Test Strategy

- Reveals:
    - All state control faults
    - All sneak paths
    - Many corrupt states
- Procedure:
    1. Derive round-trip path tree from state model
    2. Generate the round-trip path test cases
    3. Generate sneak path test cases
    4. Sensitize transitions in each test case

Round-Trip Path Tree

- Prerequisite
    - Flatten state model (remove concurrency and hierarchy)
- Algorithm
    - Initial state is the root node of the tree
    - An edge is drawn for every transition out of the initial node, with new leaf nodes representing resultant states
    - A leaf node is marked as terminal, if the state it represents has already been drawn or is a final state
    - No more transition are traced out of a terminal node (only one iteration of a loop is allowed
    - Repeat until all leaf nodes are terminal
- Tree structure depends on the order in which transitions are traced (breadth first or depth first)
- Depth first search yields fewer, longer test sequences
- The order in which sates are investigated is supposed to be irrelevant
- Is used to:
    - Check conformance to explicit behavior model
    - Find sneak paths

Conformance Test Cases

- Each test sequence begins at the root node and ends at a leaf node → each path through the round-trip path three produces a test case
- The expected result (oracle) is the sequence of states and actions (outputs) - assuming states can

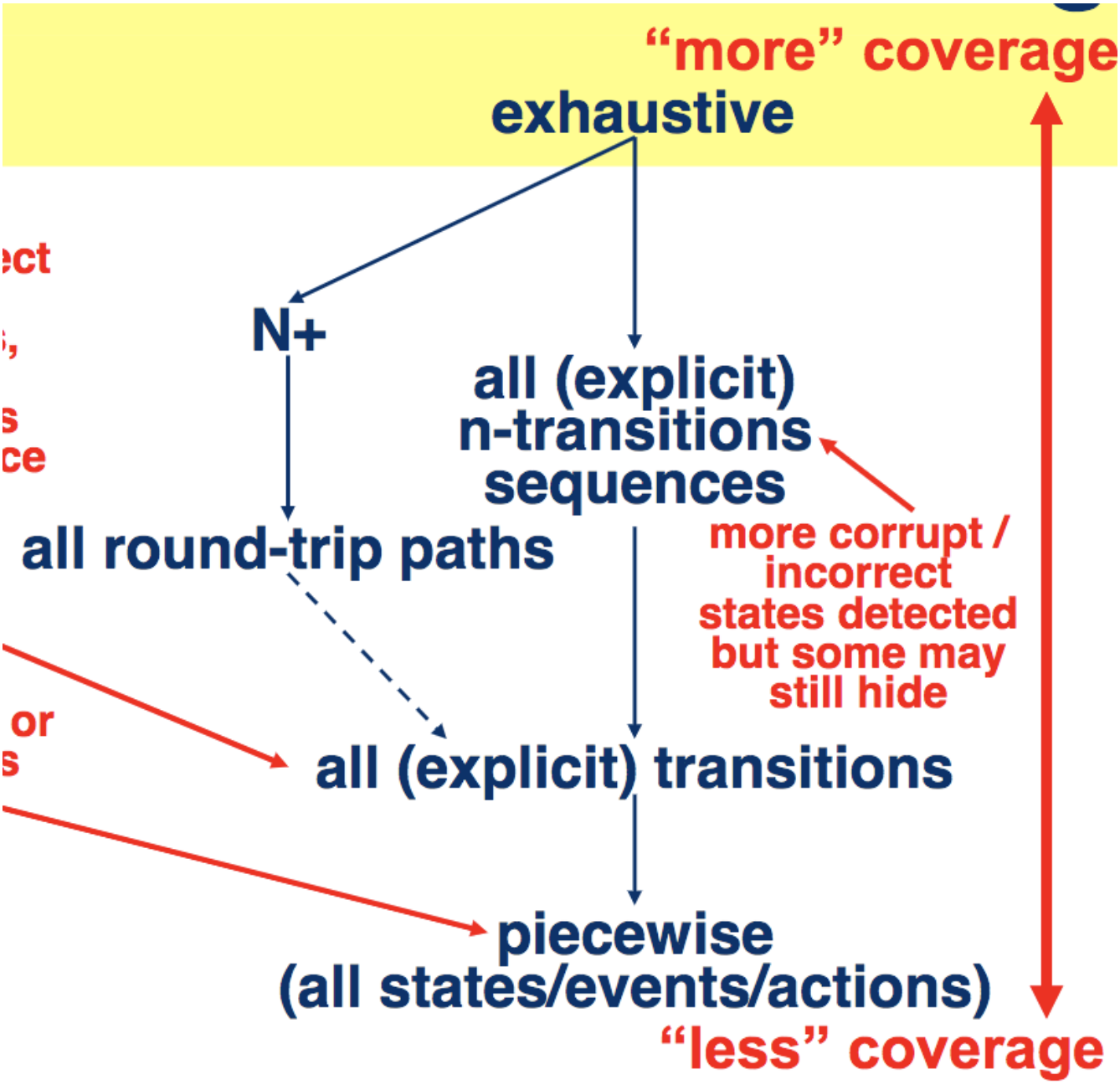
be "observed"

- Test cases are completed by identifying method parameter values and required conditions to traverse a path
- Run the test cases by setting the object under test to the initial state, applying the sequence, and then checking the intermediary states, final state, and outputs

Sneak Path Test Cases

- Covering all round-trip paths shows conformance to the explicitly modeled behavior
- When state machines are incompletely specified, need to test for sneak paths
- A sneak path is possible for each unspecified transition and for guarded transitions that evaluate false → show if implicitly excluded behavior is correctly handled
- Need to test all states' illegal events → guarantees to reveal all sneak paths
- Place the object in the corresponding state
- Apply illegal event by sending message or forcing the virtual machine to generate the desired event
- Check the actual response matches the specified response
- Check that the resultant state is unchanged

Hierarchy of State-based Testing Strategies



Support for State-Based Testability

- How to compare the actual result state from a test exec. sequence with the expected result?
- Possible approaches:
  - Built-in test support
  - Test Repetition (heuristic to reveal corrupted states when impossible to alter code)
    1. Run test case saving all output actions
    2. Repeat test and compare output with saved output - it is unlikely to get the same result starting from a corrupt state

Built-In Test Support

- Get state methods: methods that evaluate the state invariants and return boolean indicating whether an object is in a specific state

- During OO analysis and design, state is defined by a state invariant
- Each state invariant is associated with an executable assertions in the code
- Set state methods: built in methods to set objects in certain states that are difficult to reach but are the starting state for a test sequence
- Only test drivers should be allowed to use these methods

Exam Review

- 10 conceptual classes
  - IE. Compare A to B
  - Everything covered in lectures
  - Every concept covered in tutorials
  - No technology specific questions
  - No multiple choice/true or false questions
- 7 practical exercises
  - Similar to project deliverables, or assignments, quizzes, lecture slides
  - **Basically everything covered in assignments**

LTL

- Use link on mycourses to help study with ltl portion of the exam
- Here is the automaton, here is the property, see that the property holds *