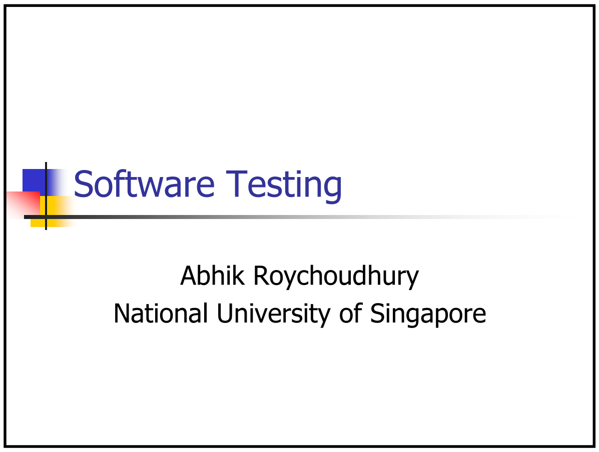
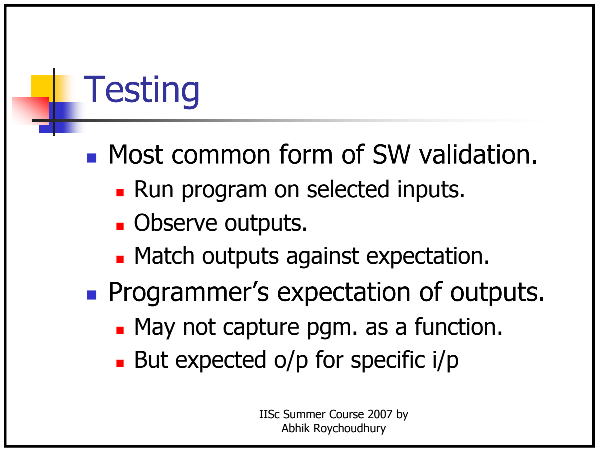


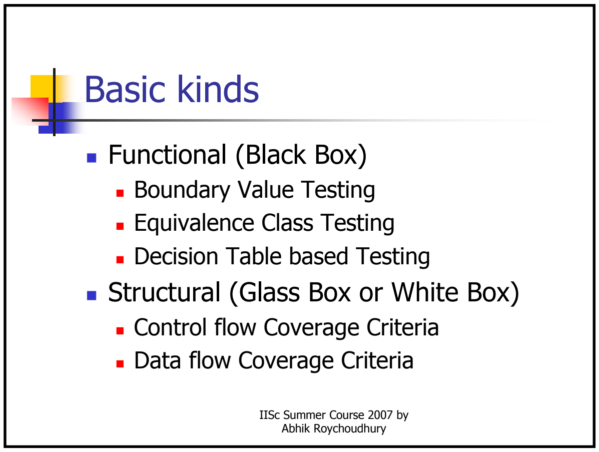
Software Testing

Abhik Roychoudhury
National University of Singapore



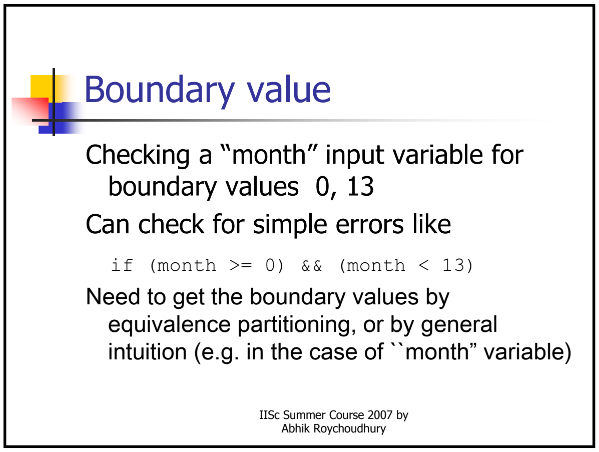
Testing

- Most common form of SW validation.
  - Run program on selected inputs.
  - Observe outputs.
  - Match outputs against expectation.
- Programmer’s expectation of outputs.
  - May not capture pgm. as a function.
  - But expected o/p for specific i/p



Basic kinds

- Functional (Black Box)
  - Boundary Value Testing
  - Equivalence Class Testing
  - Decision Table based Testing
- Structural (Glass Box or White Box)
  - Control flow Coverage Criteria
  - Data flow Coverage Criteria



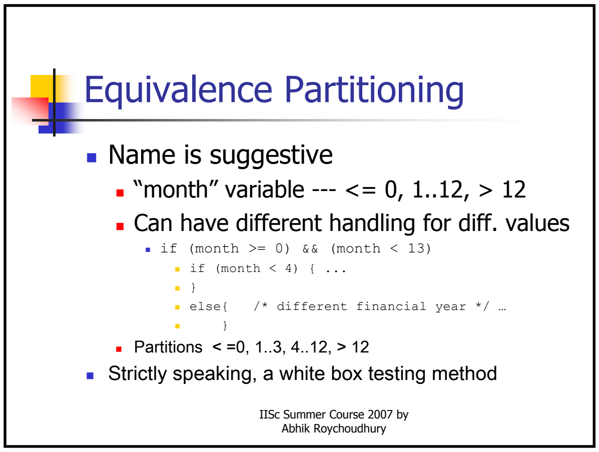
Boundary value

Checking a “month” input variable for boundary values 0, 13

Can check for simple errors like

```
if (month >= 0) && (month < 13)
```

Need to get the boundary values by equivalence partitioning, or by general intuition (e.g. in the case of ``month” variable)

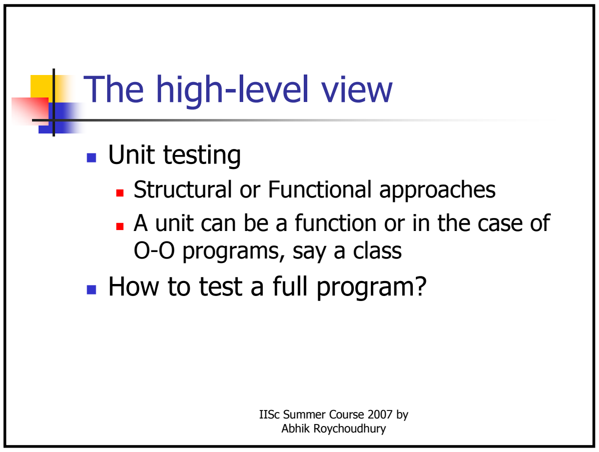


Equivalence Partitioning

- Name is suggestive
  - “month” variable --- <= 0, 1..12, > 12
  - Can have different handling for diff. values

```
if (month >= 0) && (month < 13)
    if (month < 4) { ...
    }
    else{   /* different financial year */ …
        }
```

  - Partitions < =0, 1..3, 4..12, > 12
- Strictly speaking, a white box testing method



The high-level view

- Unit testing
  - Structural or Functional approaches
  - A unit can be a function or in the case of O-O programs, say a class
- How to test a full program?

The high-level view

- Integration testing
  - Call graph based integration
  - Path based integration
- Overall system testing
  - Testing multi-threaded programs
    - Both structural & functional approaches
    - More research is necessary.

The high-level view

- Regression testing
  - Check that the program still works after a feature is added to a tested program
- Stress testing
  - Testing program under extreme conditions
    - e.g. a web service with lots of users, or
    - a database application with lots of data.

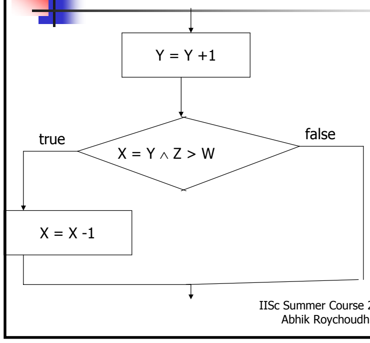
Common terminology

- Test case
  - A test input (or its execution trace)
- Test suite
  - Set of test cases
- Test purpose
  - A formal specification to guide testing
    - e.g. a regular expr. which the test case should satisfy
- Coverage criterion
  - A guide to exhaustively cover program structure.
    - e.g. Statement coverage, Branch coverage etc.

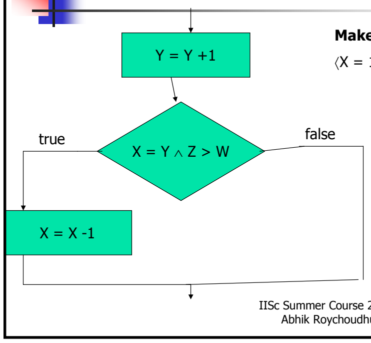
Coverage Criteria

- Control flow based
  - Statement, Edge, Condition, Path
- Data flow based
  - All defs, All uses etc

Example



Statement Coverage



Make the branch condition true

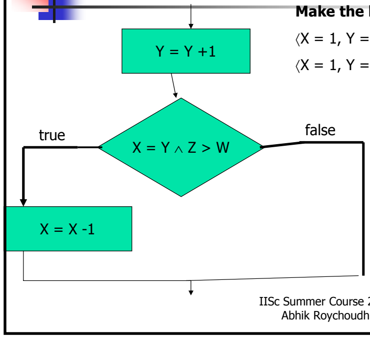
⟨X = 1, Y = 1, Z = 2, W = 1⟩

Edge Coverage

Make the branch condition true/false

⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩



Condition Coverage

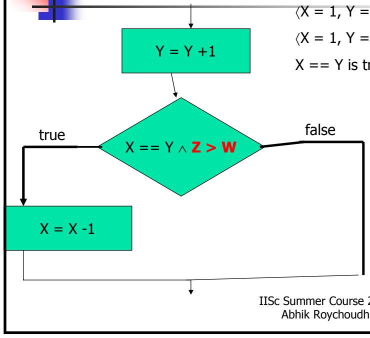
- For each executable condition c
  - Check whether it can be both true or false
    - c could be unsatisfiable or valid in all pgm. executions
  - For all such conditions c, c should be true in at least one test in the test suite, and c should be false in at least one test in the test suite.

Condition Coverage

⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

X == Y is true in both the test cases

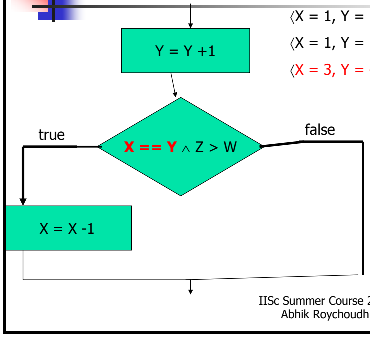


Condition Coverage

⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

⟨X = 3, Y = 4, Z = 7, W = 5⟩



Multiple condition coverage

- Branch condition
  - A ∧ B
- Edge coverage
  - A = B = true
  - A = true, B = false
- Condition coverage
  - A = B = true
  - A = true, B = false
  - A = false, B = true
- Still we do not explore all combinations !

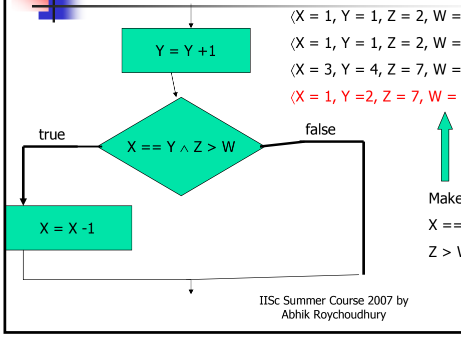
Multiple Condition Coverage

⟨X = 1, Y = 1, Z = 2, W = 1⟩

⟨X = 1, Y = 1, Z = 2, W = 2⟩

⟨X = 3, Y = 4, Z = 7, W = 5⟩

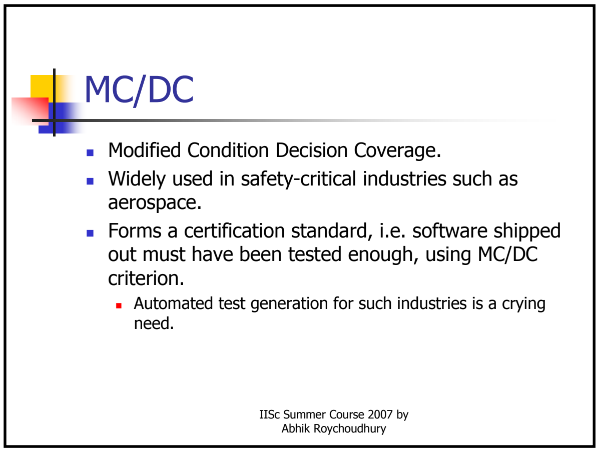
⟨X = 1, Y =2, Z = 7, W = 5⟩



Make both

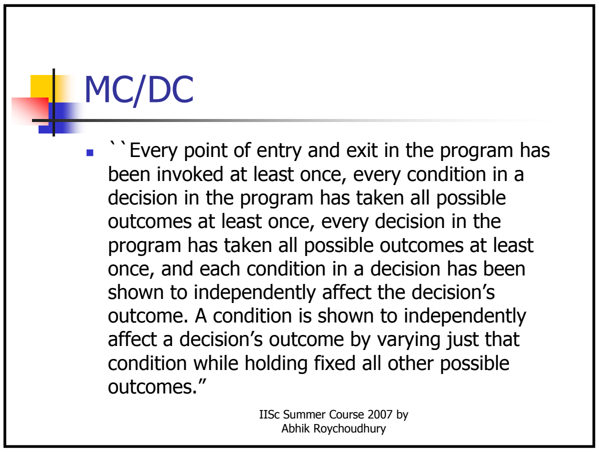
X == Y and

Z > W to be false



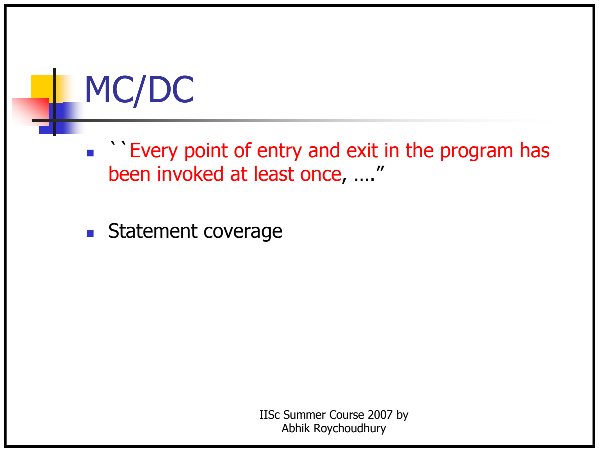
MC/DC

- Modified Condition Decision Coverage.
- Widely used in safety-critical industries such as aerospace.
- Forms a certification standard, i.e. software shipped out must have been tested enough, using MC/DC criterion.
  - Automated test generation for such industries is a crying need.



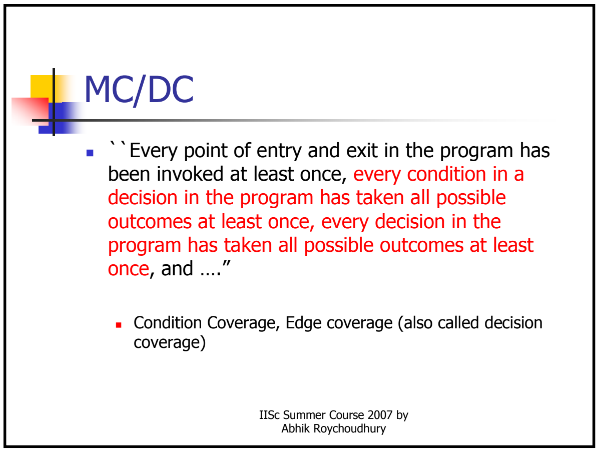
MC/DC

- ``Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible outcomes.”



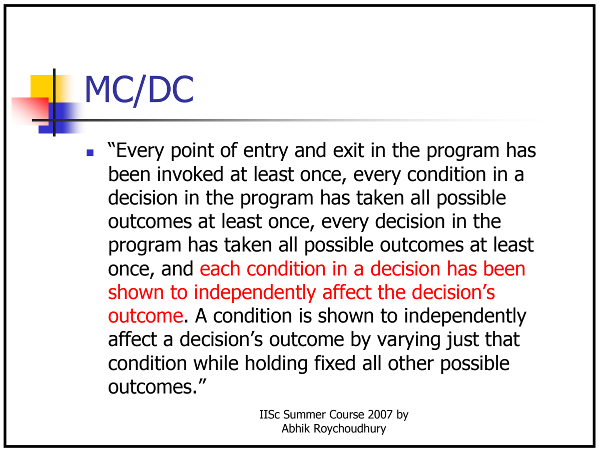
MC/DC

- ``Every point of entry and exit in the program has been invoked at least once, ….”
- Statement coverage



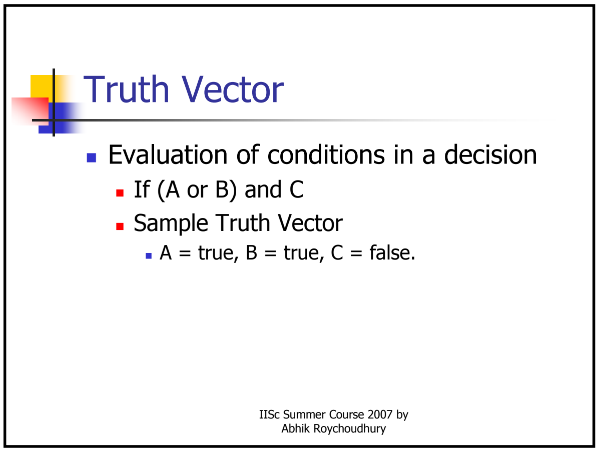
MC/DC

- ``Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and ….”
  - Condition Coverage, Edge coverage (also called decision coverage)



MC/DC

- “Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible outcomes at least once, and each condition in a decision has been shown to independently affect the decision’s outcome. A condition is shown to independently affect a decision’s outcome by varying just that condition while holding fixed all other possible outcomes.”



Truth Vector

- Evaluation of conditions in a decision
  - If (A or B) and C
  - Sample Truth Vector
    - A = true, B = true, C = false.

MC/DC pair

- Given a decision statement S,
  - A pair of truth vectors v1, v2 s.t.
    - v1, v2 evaluate S differently
    - v1, v2 differ only in the evaluation of one condition in S.

Example of MC/DC pair

- if (A or B) and C
  - v1: A = false, B = true, C = true
  - v2: A = false, B = true, C = false

  - v1 evaluates decision to true
  - v2 evaluates decision to false

MC/DC pair is not unique

- if (A or B) and C
  - MC/DC pairs for C
    - FTT FTF
    - TFT TFF
    - TTT TTF

Test suite

- Coverage of at least one MC/DC pair for each condition.

- There exist many methods for test suite reduction once we collect the MC/DC pairs
  - Test suite reduction and prioritization for MC/DC
  - J.A Jones and M. J Harrold, ICSM 2001,
  - http://www.cc.gatech.edu/aristotle/Publications/Papers/icsm01.pdf

MC/DC pairs

- **if (A or B) and C**
- Pair for A
  - TFT FFT
- Pair for B
  - FTT FFT
- Pair for C
  - FTT FTF
  - TFT TFF
  - TTT TTF

- If (not(D and E)) and F
- Pair for D
  - FTT TTT
- Pair for E
  - TFT TTT
- Pair for F
  - FTT FTF
  - TFT TFF
  - FFT FFF

Choosing tests

- Choosing tests for one branch statement is straightforward
- How to choose tests for the entire program?
  - Finding test suite for individual statements and summing up does not work!
    - Associate with each MC/DC pair, pairs of test cases which cover the pair.
    - Define the suite as sets of pairs
    - Reduce this suite using conventional techniques
    - Take all tests from the reduced set of pairs.

Path coverage

- Cover all paths in the program
  - Unboundedly many, unless loops can be bounded.
  - Lot of infeasible paths i.e. paths which do not form execution trace for any input.
    - Infeasible path detection will help test-suite construction
  - Can try for coverage of all acyclic paths in the program.

Comparison

- Compare path coverage and multiple condition coverage
  - We can execute all paths without exercising all conditions in all decisions
    - Some valuations are infeasible.
  - All conditions in all decisions may be exercised but all paths may not be covered.

Infeasible path detection

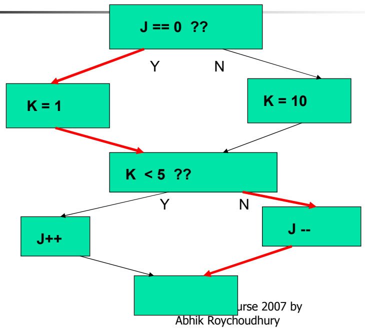
- Important problem for reducing test suite size.
- Useful to find out smallest infeasible path patterns.
- But, first how do we even test that a given path is infeasible.

Infeasible paths

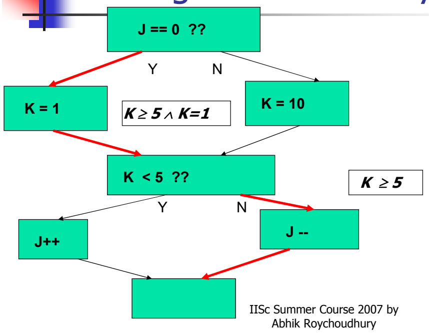
- J = 1;
- If (J == 0){
- K++; // this branch will never be taken
- } else{
- K--;
- }

Only possible to know via data flow analysis.

Infeasible Path



Testing for infeasibility



Constraint Propagation

- Over Control Flow Graph
  - Start from an outgoing edge of a branch
  - This gives an initial constraint.
  - Traverse the CFG backwards by calculating a weakest pre-condition at each step.
  - Stop when constraint store is unsatisfiable.
- Many issues –
  - Constraint solvers ?
  - Full-fledged loop unrolling ?
    - Heuristics to stop after few iterations
    - Limited detection – infeasible paths within a loop/ loop-iteration.

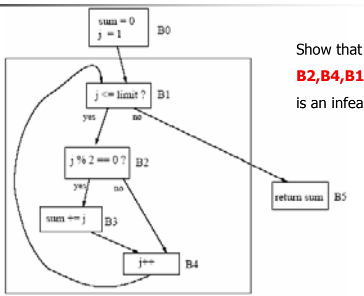
One step of propagation

- Constraint accumulated $\varphi(X_1,\ldots,X_k)$
- One step weakest pre-condition computation w.r.t. statement s
  - Effect constraint of s is
    - $\psi_s(X_1,\ldots,X_k,X_1',\ldots,X_k')$
    - Effect constraint of X = X+1 over vars. {X,Y,Z} is
      - $\psi_s(X, Y, Z, X', Y', Z') == (X'=X+1 \wedge Y' = Y \wedge Z' = Z)$
  - $WP(X_1,\ldots,X_k) =$
  - $\forall X_1',\ldots,X_k' \;\; \psi_s(X_1,\ldots,X_k,X_1',\ldots,X_k') \Rightarrow \varphi(X_1',\ldots, X_k')$

Existing Constraint Solvers

- Simplify Theorem Prover – Compaq SRC
  - Integrates automatic decision procedures.
    - Equality
    - Arithmetic
    - Arrays
  - Sound, incomplete
    - Unsatisfiable constraint may not be detected.
    - Incomplete detection of infeasible path patterns – OK !

How good is the detective?



Show that

B2,B4,B1,B2,B4

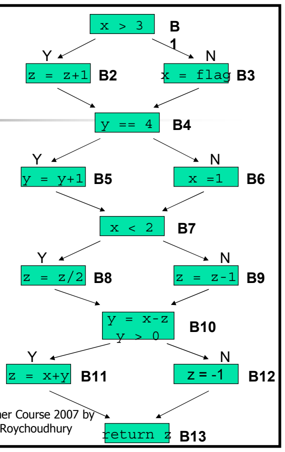
is an infeasible path

Coverage Criteria

- Control flow based
  - Statement, Edge, Condition, Path
- Data flow based
  - All defs, All uses etc

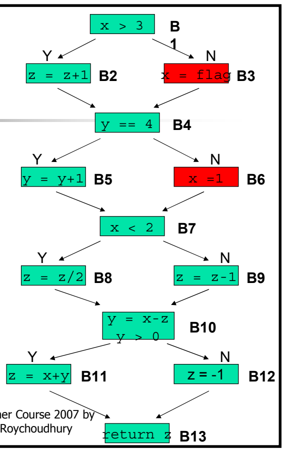
```
int P1( int flag ) {
        int x, y, z;

        if( x > 3 )
              z = z + 1;
        else
              x = flag;
        if( y == 4 )
              y = y + 1;
        else
              x = 1;
        if( x < 2 )
              z = z / 2;
        else
              z = z - 1;
        y = x – z;
        if( y > 0 )
              z = x + y;
        else
              z = –1;
        return z;
}
```



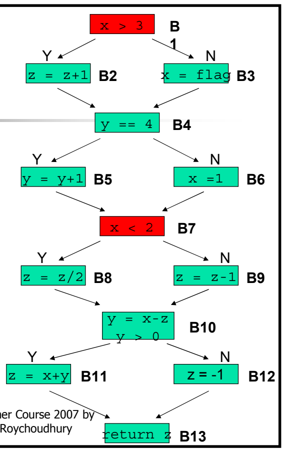
def(x)

Nodes where variable x is assigned



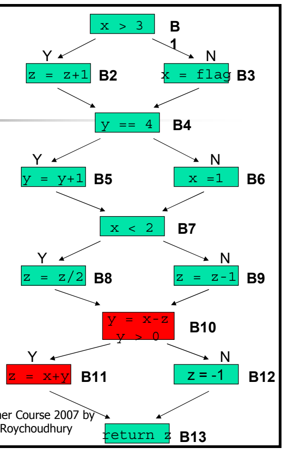
p-use(x)

Nodes where variable x is used in a predicate.



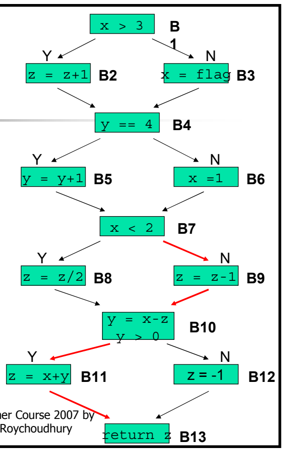
c-use(x)

Nodes where variable x is used in any expression other than a predicate (say rhs of assignment)



def-clear(x)

Set of paths which do not contain any node in def(x)



dpu(s,x)

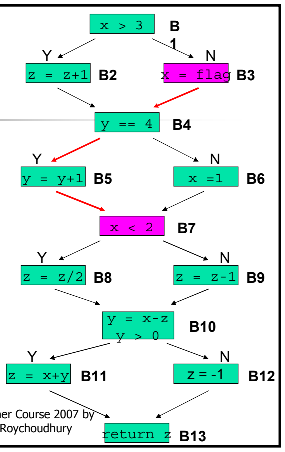
Given variable x, and s ∈ def(x)

dpu(s,x) =

{ s' | ∃ def-clear(x) path from s to s'

and s' ∈ p-use(x)

}



dcu(s,x)

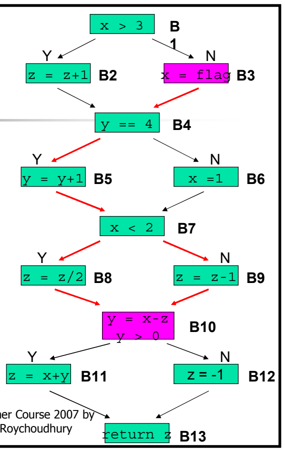
Given variable x, and s ∈ def(x)

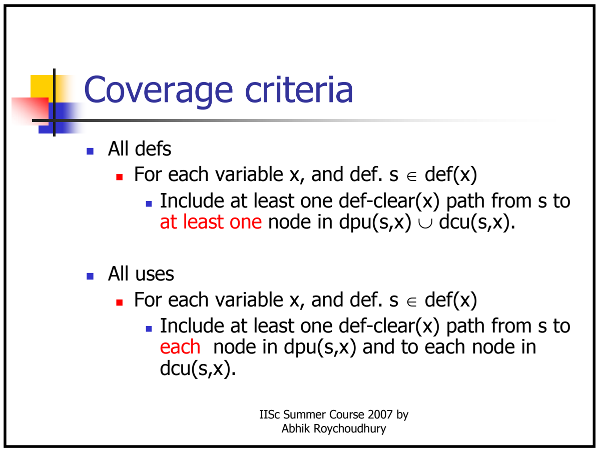
dcu(s,x) =

{ s' | ∃ def-clear(x) path from s to s'

and s' ∈ c-use(x)

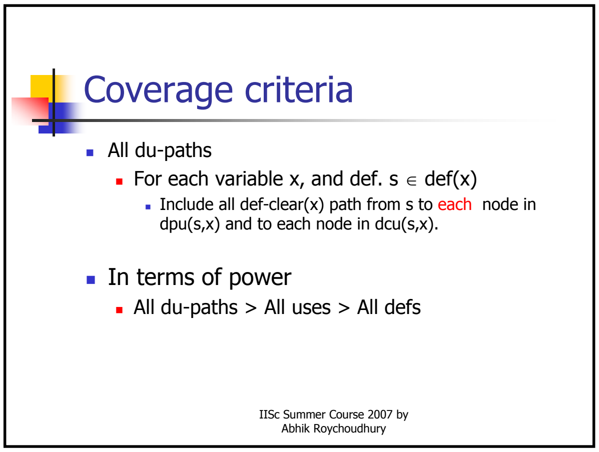
}





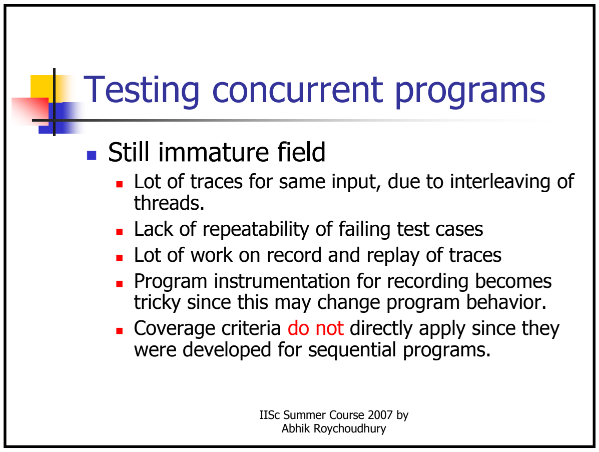
Coverage criteria

- All defs
  - For each variable x, and def. s ∈ def(x)
    - Include at least one def-clear(x) path from s to at least one node in dpu(s,x) ∪ dcu(s,x).
- All uses
  - For each variable x, and def. s ∈ def(x)
    - Include at least one def-clear(x) path from s to each node in dpu(s,x) and to each node in dcu(s,x).



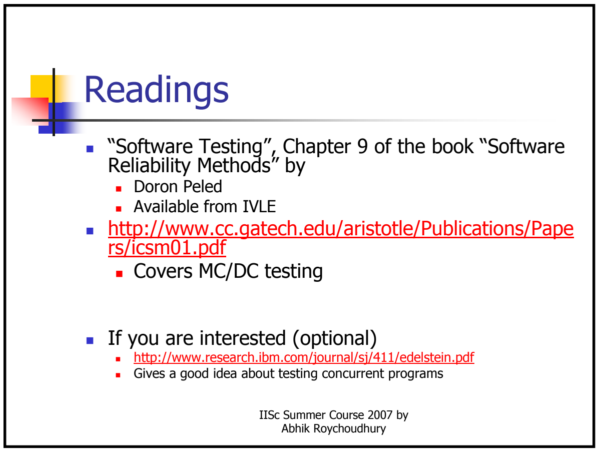
Coverage criteria

- All du-paths
  - For each variable x, and def. s ∈ def(x)
    - Include all def-clear(x) path from s to each node in dpu(s,x) and to each node in dcu(s,x).
- In terms of power
  - All du-paths > All uses > All defs



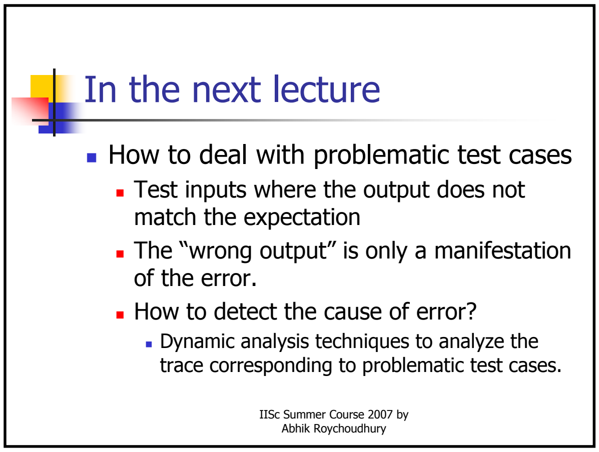
Testing concurrent programs

- Still immature field
  - Lot of traces for same input, due to interleaving of threads.
  - Lack of repeatability of failing test cases
  - Lot of work on record and replay of traces
  - Program instrumentation for recording becomes tricky since this may change program behavior.
  - Coverage criteria do not directly apply since they were developed for sequential programs.



Readings

- "Software Testing", Chapter 9 of the book "Software Reliability Methods" by
  - Doron Peled
  - Available from IVLE
- http://www.cc.gatech.edu/aristotle/Publications/Papers/icsm01.pdf
  - Covers MC/DC testing
- If you are interested (optional)
  - http://www.research.ibm.com/journal/sj/411/edelstein.pdf
  - Gives a good idea about testing concurrent programs



In the next lecture

- How to deal with problematic test cases
  - Test inputs where the output does not match the expectation
  - The "wrong output" is only a manifestation of the error.
  - How to detect the cause of error?
    - Dynamic analysis techniques to analyze the trace corresponding to problematic test cases.