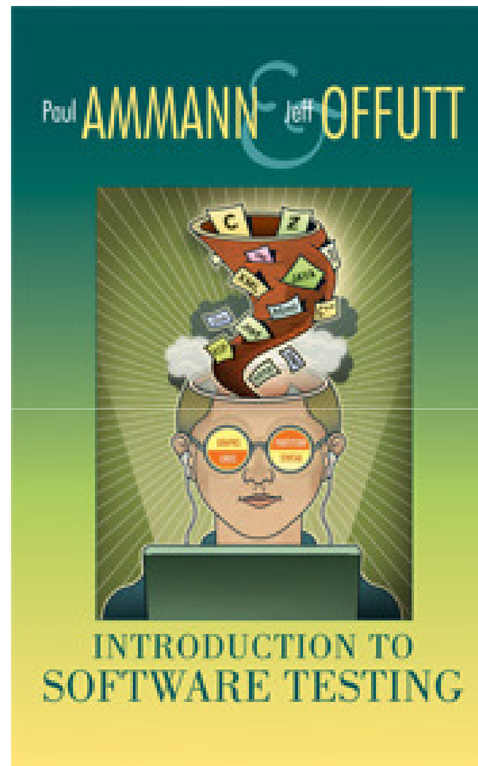
Testing Object-Oriented Systems

Test coverage criteria

Marcello Bonsangue
Frank de Boer

Leiden Institute of Advanced Computer Science
Research & Education

Introduction to Software Testing



These slides are loosely based on
Part II of this book

Constructing a Test Suite

- Unit, integration and system testing need a set of test cases
  - How can we generate them?

- Test coverage: define a model of the software, then find ways to cover it

- Test requirements : specific things that must be satisfied or covered during testing

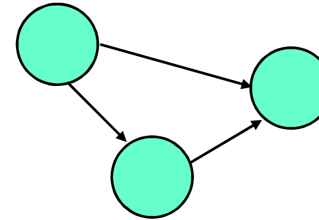
- Coverage Criterion : A collection of rules and a process that define test requirements

Coverage

- Given a set of test requirements *TR* for coverage criterion *C*, a test suite *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

Criteria Based on Structures

1. Graphs



2. Logical Expressions

$(\neg X \vee \neg Y) \wedge A \wedge B$

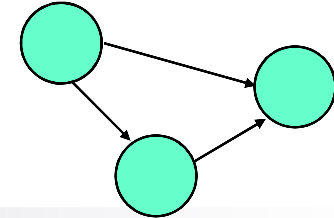
3. Input Domain Characterization

A: {0, 1, 10}
B: {600, 700, 800}
C: {liacs, cs, toos}

4. Syntactic Structures

```
if (x > y) then z = x - y;
           else z = 2 * x;
```

Covering Graphs



- Graphs are the most commonly used structure for testing

- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - Finite State Machines and statecharts
  - Use cases

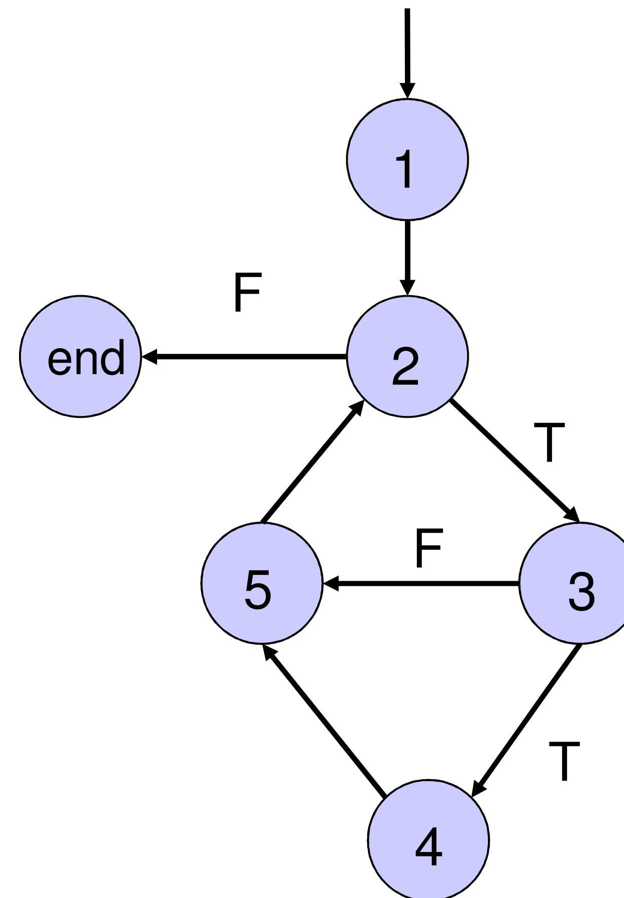
- Tests usually are intended to "cover" the graph in some way

Control Flow Graphs

Program

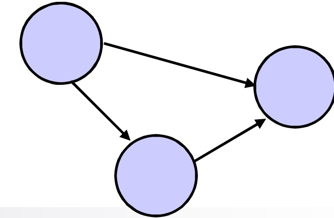
```
1 c = 0
2 while (a > 1) do
3         if (a+2 > c) then
4                   c = c + a
5         a = a - 2
  od
```

Flow graph

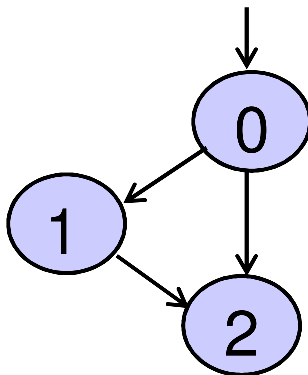


- Nodes = basic blocks
- Edges = control flow

Node and Edge Coverage



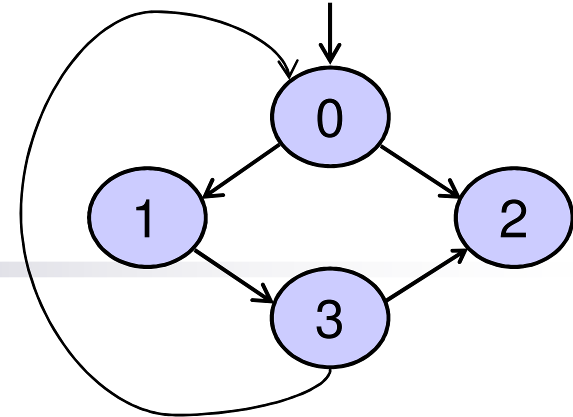
- Node coverage: TR contains each reachable node.
- Edge coverage: TR contains all reachable edges
- NC and EC are only different when there is more than one outgoing edge from a node (as in an "if-else" statement)



Node Coverage : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

Edge Coverage : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

Loops in Graphs

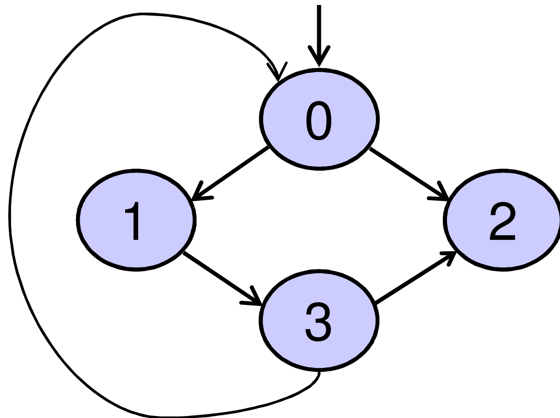


- If a graph contains a loop, it has an infinite number of paths
  - A suite containing all paths is not feasible
  - Execute each loop exactly once may be not enough
  - Execute loops a specified number of times is subjective
- Prime path coverage: TR contains all prime paths
  - A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Loops in Graphs

- Simple Path : A path with no node appearing more than once, except possibly for the first and last nodes
  - No internal loops
  - A loop is a simple path

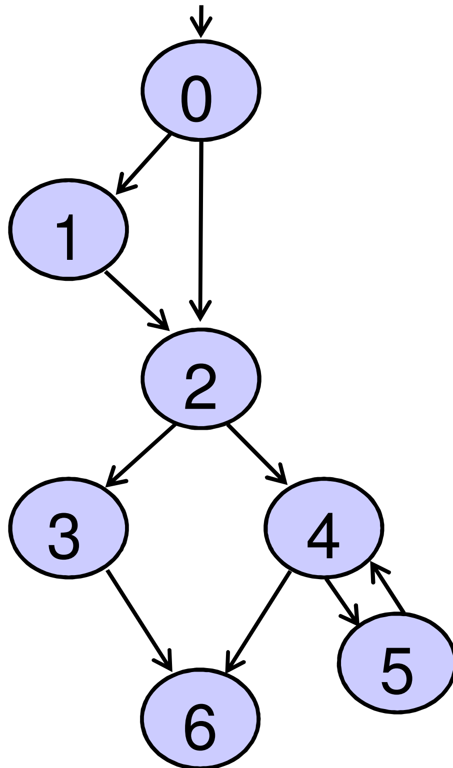
- Prime Path : A simple path that does not appear as a proper subpath of any other simple path



Simple Paths from 0: [0, 1, 3, 0], [0, 1, 3, 2], [0, 1, 3], [0, 1], [0, 2], [0]

Prime Paths from 0: [0, 1, 3, 0], [0, 1, 3, 2]

Prime Paths - Example



Prime Paths
[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Execute loop 0 times

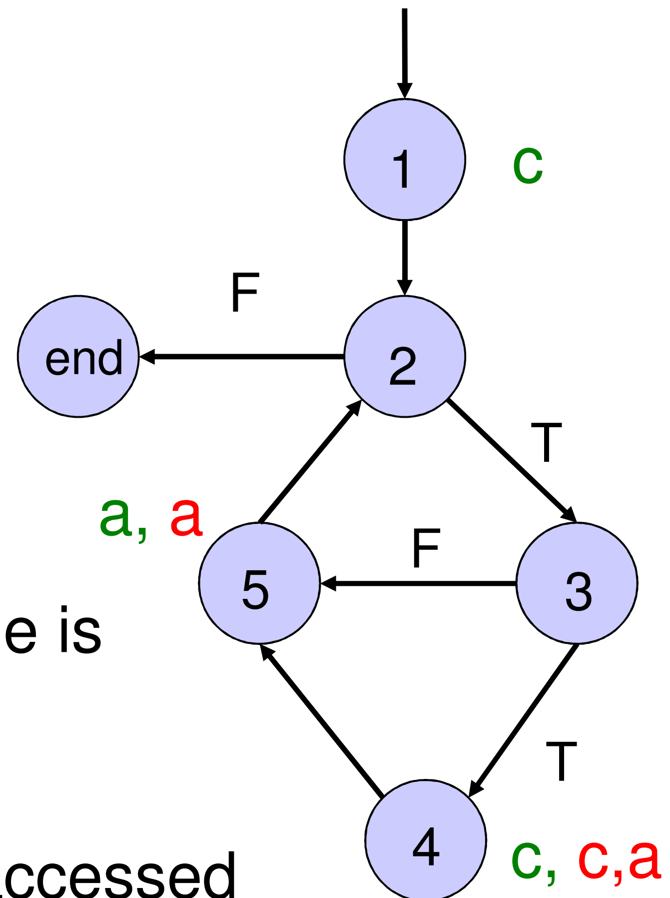
Execute loop once

Execute loop more than once

Data Flow Graphs

Program

```
1 c = 0
2 while (a > 1) do
3         if (a+2 > c) then
4                 c = c + a
5         a = a - 2
   od
```

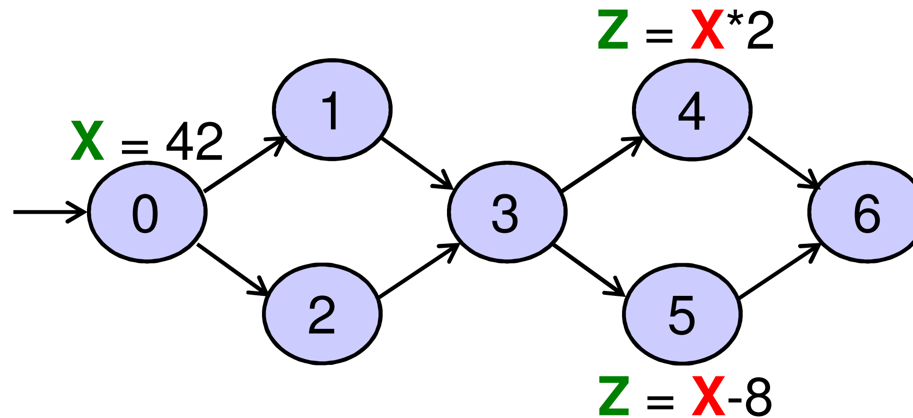


- Definition = a location where a variable is stored in memory
- Use = a location where a variable is accessed

Data Flow Test Criteria

- All-Defs Coverage (ADC): TR contains at least one path such that each definition reaches a use
- All-Uses Coverage (ADC): TR contains paths such that each definition reaches all possible uses
- All-DU Coverage (ADUC): TR contains all paths between definitions and uses

Data Flow Testing - Example



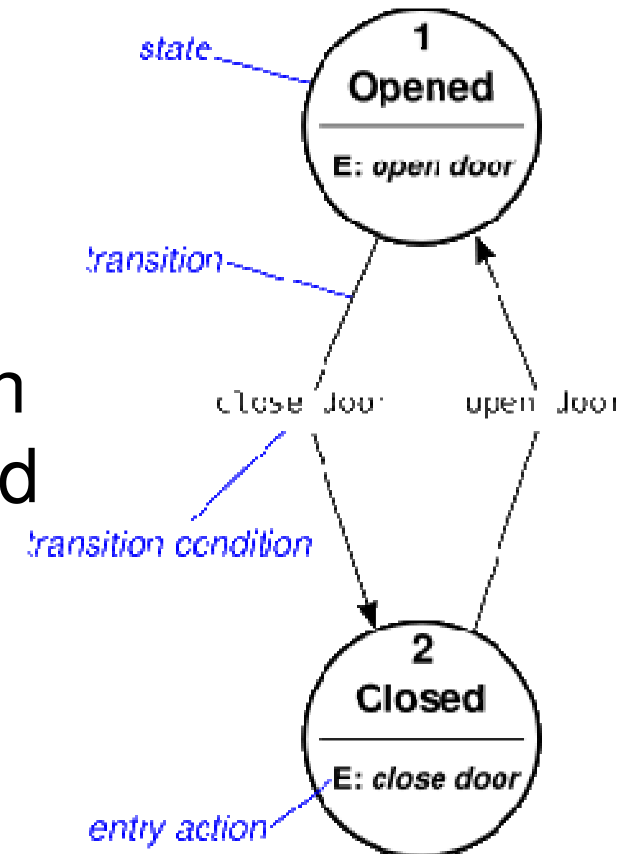
All-defs for *X*
[0, 1, 3, 4]

All-Uses for *X*
[0, 1, 3, 4]
[0, 1, 3, 5]

All-DU-paths for *X*
[0, 1, 3, 4]
[0, 2, 3, 4]
[0, 1, 3, 5]
[0, 2, 3, 5]

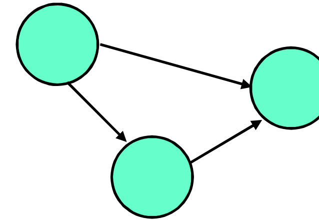
Finite State Machines

- State coverage
- Transition coverage
- Deriving test case that distinguish states using automata theory and regular expressions



Criteria Based on Structures

1. Graphs



2. Logical Expressions

$(\neg X \vee \neg Y) \wedge A \wedge B$

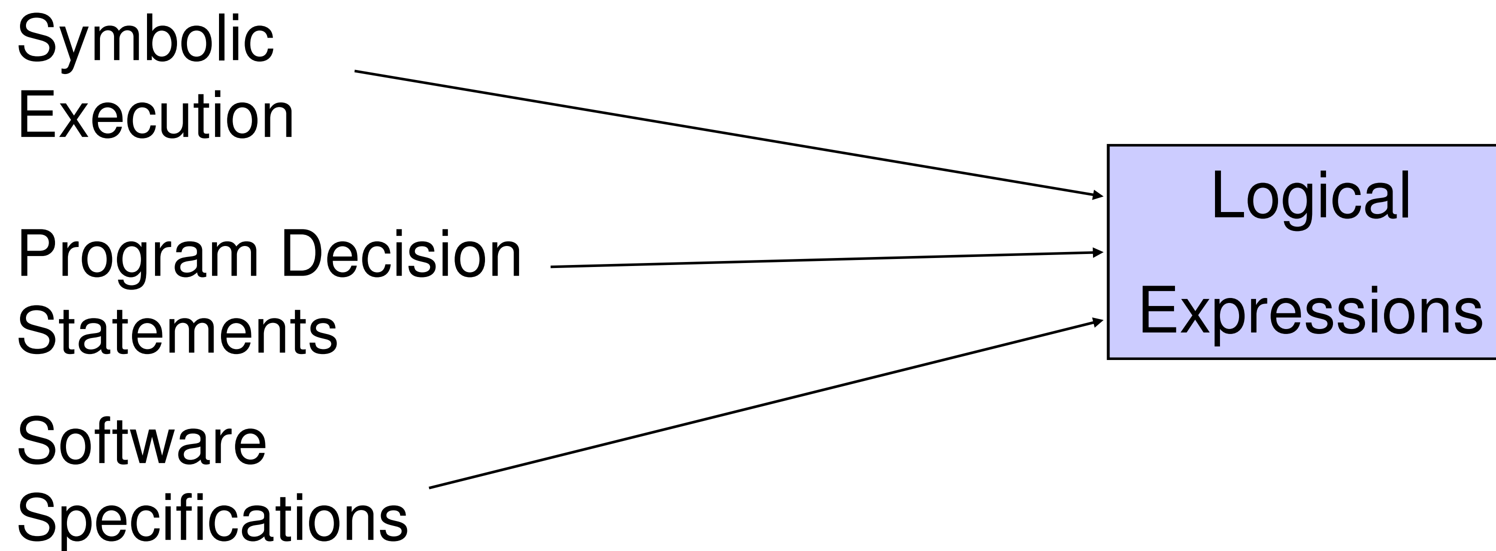
3. Input Domain Characterization

A: {0, 1, 10}
B: {600, 700, 800}
C: {liacs, cs, toos}

4. Syntactic Structures

```
if (x > y) then z = x - y;
           else z = 2 * x;
```

Logical Expressions



Symbolic Execution

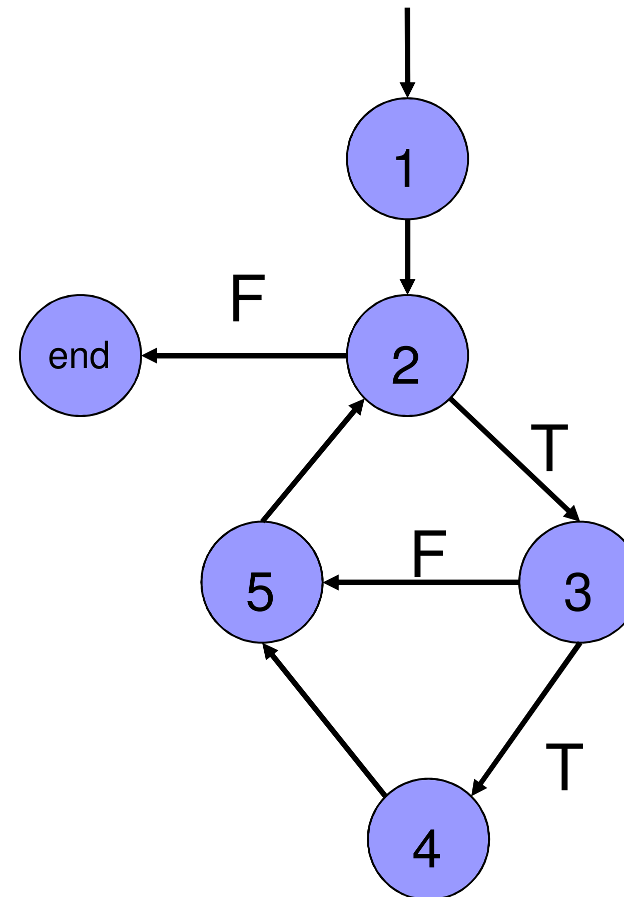
- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Basic technique: execute using symbols
  - by calculating pre or post conditions
  - Using invariant for loops and procedures

暑中お見舞い申し上げます

Symbolic Execution - Example

- Path Condition: A condition that should be fulfilled to walk along a path

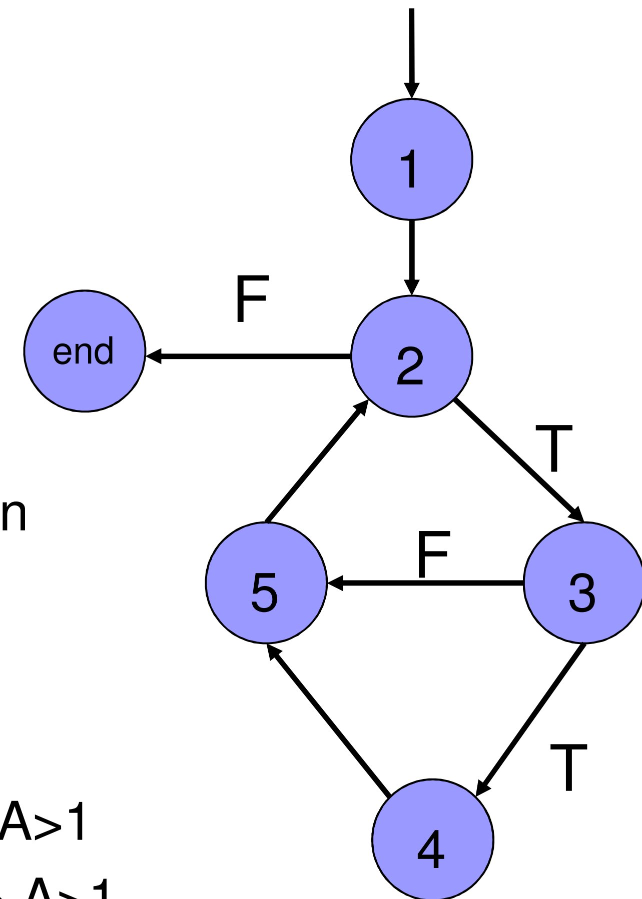
```
1 c = 0
2 while (a > 1) do
3         if (a^2 > c) then
4                 c = c + a
5         a = a - 2
  od
```



- Path = 1 : 2 : end
- Condition = $a \leq 1$

Symbolic Execution - Example

```
1 c = 0
2 while (a > 1) do
3         if (a^2 > c) then
4                 c = c + a
5         a = a - 2
  od
```



- Path = 1 : 2 : 3 : 4 : 5 : 2 : 3 : 4 : 5 : 2 : end

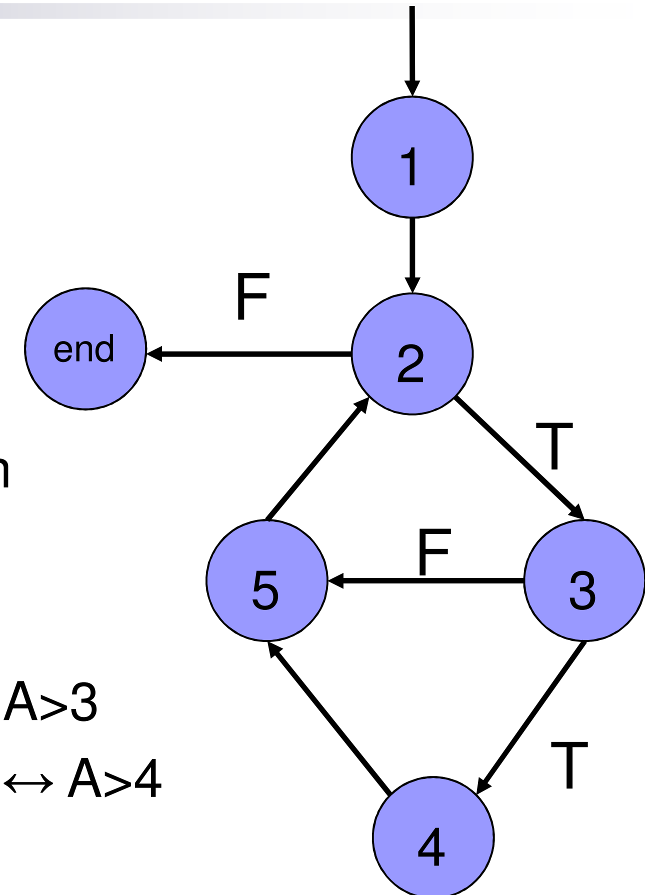
node	symbolic values	path condition
	(A, C)	true
1	(A,0)	true
2	(A, 0)	true
3	(A, 0)	$(true \land A>1) \leftrightarrow A>1$
4	(A, 0)	$(A>1 \land A^2>0) \leftrightarrow A>1$

Symbolic Execution - Example

```
1 c = 0
2 while (a > 1) do
3         if (a^2 > c) then
4                 c = c + a
5         a = a - 2
  od
```

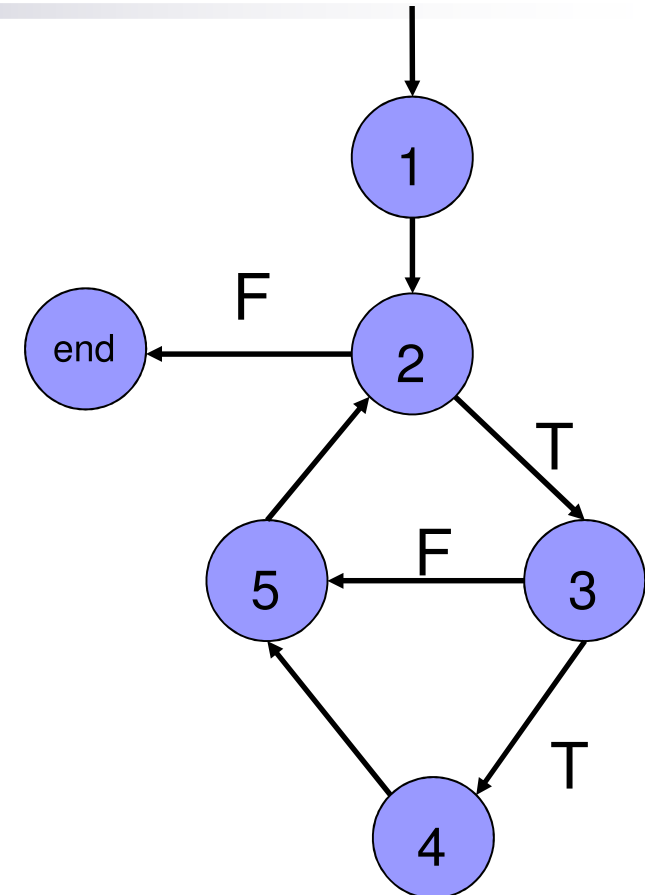
- Path = 1 : 2 : 3 : 4 : 5 : 2 : 3 : 4 : 5 : 2 : end

node	symbolic values	path condition
5	(A, C)	A>1
2	(A-2, A)	A>1
3	(A-2, A)	$(A>1 \wedge A-2>1) \leftrightarrow A>3$
4	(A-2, A)	$(A>3 \wedge (A-2)^2>A) \leftrightarrow A>4$
5	(A-2, 2A-2)	A>4
2	(A-4, 2A-2)	A>4
end	**(A-4, 2A-2)**	$\mathbf{(A>4 \wedge A-4\leq 1) \leftrightarrow A=5}$



Symbolic Execution - Example

```
1 c = 0
2 while (a > 1) do
3         if (a^2 > c) then
4                   c = c + a
5         a = a - 2
  od
```



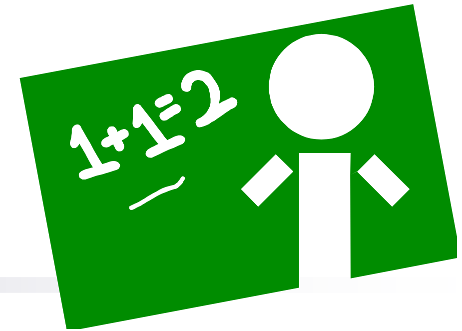
- Path = 1 : 2 : 3 : 4 : 5 : 2 : 3 : 4 : 5 : 2 : end

node	symbolic values	path condition
end	**(A-4, 2A-2)**	**A=5**

Test case: a = 5

Expected result : (A-4, 2A-2) i.e. a =1, c = 8

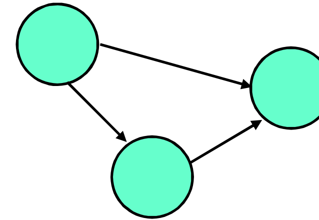
Logic Coverage



- Predicate Coverage : Each predicate must be true and false
  - When is $\neg P(x) \wedge (A < B)$ true (false)?
- Clause Coverage : Each clause must be true and false
  - When is $P(x)$ true (false)?
  - When is $(A < B)$ true (false)?
- Combinatorial Coverage : Various combinations of clauses

Criteria Based on Structures

1. Graphs



2. Logical Expressions

$(\neg X \vee \neg Y) \wedge A \wedge B$

3. Input Domain Characterization

A: {0, 1, 10}
B: {600, 700, 800}
C: {liacs, cs, toos}

4. Syntactic Structures

```
if (x > y) then z = x - y;
            else z = 2 * x;
```

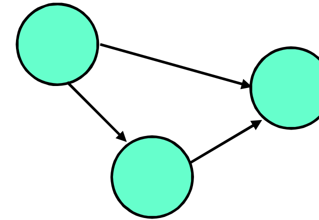
Input Domain Characterization

- Describe the input domain of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values

- Unit level - Example
  - Parameters F (int X, int Y)
  - Possible values X: { <0, 0, 1, 2, >2 }, Y : { 10, 20, 30 }
  - Tests F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)

Criteria Based on Structures

1. Graphs



2. Logical Expressions

$(\neg X \vee \neg Y) \wedge A \wedge B$

3. Input Domain Characterization

A: {0, 1, 10}
B: {600, 700, 800}
C: {liacs, cs, toos}

4. Syntactic Structures

```
if (x > y) then z = x - y;
           else z = 2 * x;
```

Syntactic Structures

- Based on grammars, or other syntactic definitions like regular expressions

- Primary example is mutation testing
  - Induce small changes to the program: mutants
  - Find tests that cause the mutant programs to fail: killing mutants
  - Failure is defined as different output from the original program
  - Check the output of useful tests on the original program

Syntactic Structures - Example

program and mutant

```
if (x > y) then
    z = x - y;
  else
   z = 2 * x;
```

```
 if (x > y) then
Δif (x >= y) then
    z = x - y;
  Δ z = x + y;
  Δ z = x – m;
    else
    z = 2 * x;
```