Control Flow Coverage

From S. Somé, A. Williams

Test Coverage

- Coverage can be based on:
  - source code
  - object code
  - model
    - control flow graph
    - (extended) finite state machines
    - data flow graph
  - requirements checklist
  - ...

Coverage: what to measure?

- For any coverage measure, we need:
  - A coverage unit: an element with the properties:
    - We can count the total number of units in the software.
    - We can identify which units were "hit" during a single execution run.
  - This means that we can determine the percentage of units hit during one or more execution runs.

Coverage measurement

- Types of coverage :
  - Control-flow based: based on structural elements of a code or model
  - Data-flow based: trace data in code or a model from where values are defined to where they are used
  - Checklist: ensure that all items on a list have been covered

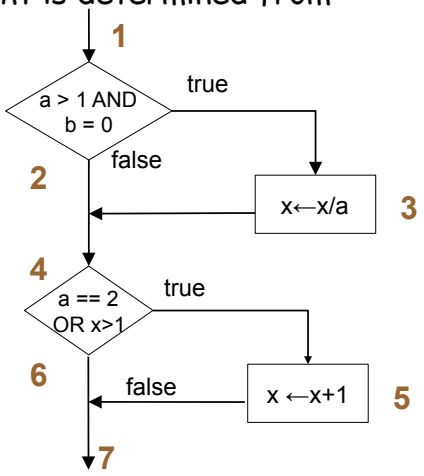
Control flow coverage

- Method coverage
- Statement coverage
- Branch coverage (also called decision coverage)
  - Minimum coverage specified by the IEEE unit test standard
- Multiple Condition coverage
  - Covers combinations of condition in decisions
- Path coverage
  - 100% path coverage impossible in practice (loops)

Flow graph

- The flow graph on the right is determined from the code on the left:

```
int proc(int a, int b, int x)
{
  if ((a>1) && (b==0)) // 1
  {
    x = x/a; // 3
  }
  if ((a==2)||(x>1)) // 4
  {
    x = x+1; // 5
  }
  return x; // 7
}
```



Statement Coverage

- Criterion: All statements must be covered during test execution.
- This is the weakest form of coverage.
  - Some branches may be missed.
- Find paths that cover all statements
- Choose input data that will result in the selected paths.

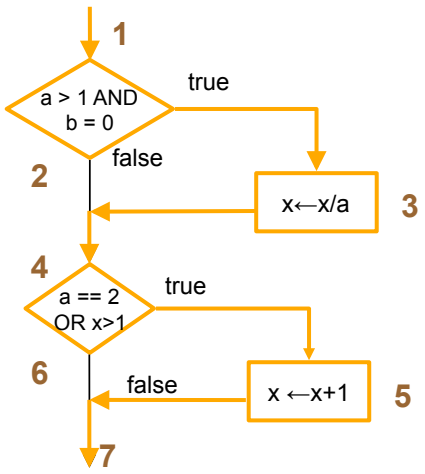
Statement Coverage

- The following path is sufficient for statement coverage:

  1 - 3 - 4 - 5 - 7

  Possible input:

  a = 2, b = 0, x = 4



Branch Coverage

- Criterion: At any branch point, each branch must be covered during test execution.
  - The true and false branch of a 2-way `if` statement.
  - Each case in a `switch` statement.
- Find paths that cover all branches
- Choose input data that will result in the selected paths.
- Branch coverage necessarily includes statement coverage.

Branch Coverage

- The following paths are sufficient for branch coverage:

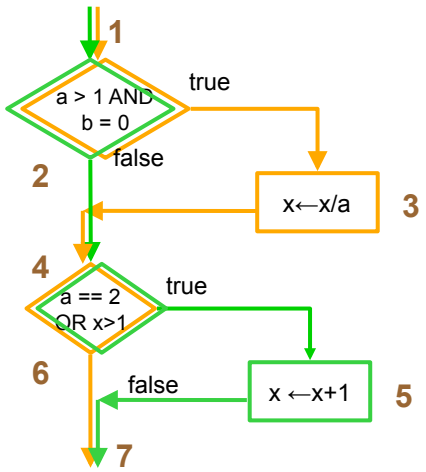
`1 - 2 - 4 - 5 - 7`

`1 - 3 - 4 - 6 - 7`

- Possible input:

`1. a = 2, b = 2, x = -1`

`2. a = 3, b = 0, x = 1`



Multiple Condition Coverage

- Criterion:
  - Every atomic (i.e. does not include AND or OR) condition must be true and false at some point during test execution.
  - In a compound logical statement (i.e. includes AND and OR), every combination of atomic conditions must be covered during test execution.
- Achieving multiple condition coverage also satisfies statement and branch coverage

Multiple Condition Coverage

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

Need cases where

1. `a > 1` is `true` and `b = 0` is `true`
2. `a > 1` is `true` and `b = 0` is `false`
3. `a > 1` is `false` and `b = 0` is `true`
4. `a > 1` is `false` and `b = 0` is `false`
5. `a = 2` is `true` and `x > 1` is `true`
6. `a = 2` is `true` and `x > 1` is `false`
7. `a = 2` is `false` and `x > 1` is `true`
8. `a = 2` is `false` and `x > 1` is `false`

Multiple Condition Coverage

1. `a > 1` is `true` and `b = 0` is `true`
2. `a > 1` is `true` and `b = 0` is `false`
3. `a > 1` is `false` and `b = 0` is `true`
4. `a > 1` is `false` and `b = 0` is `false`
5. `a = 2` is `true` and `x > 1` is `true`
6. `a = 2` is `true` and `x > 1` is `false`
7. `a = 2` is `false` and `x > 1` is `true`
8. `a = 2` is `false` and `x > 1` is `false`

Possible input:

`a = 2, b = 0, x = 2` [1][5]

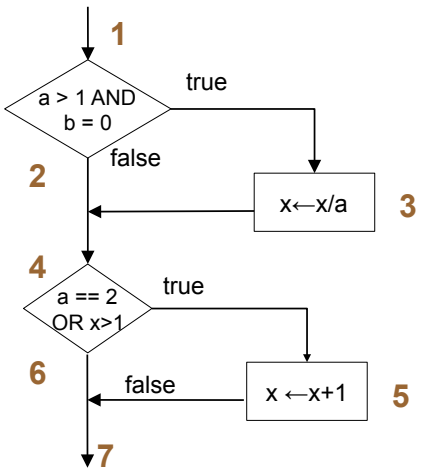
`a = 2, b = 1, x = 0` [2][6]

`a = 0, b = 0, x = 2` [3][7]

`a = 0, b = 1, x = 0` [4][8]

Multiple Condition Coverage

- Multiple condition coverage covers all branches and statements.
- Input values:
  - `a = 2, b = 0, x = 2`
  - `a = 2, b = 1, x = 0`
  - `a = 0, b = 0, x = 2`
  - `a = 0, b = 1, x = 0`
- Paths covered
  - 1 - 3 - 4 - 5 - 7
  - 1 - 2 - 4 - 5 - 7
  - 1 - 2 - 4 - 5 - 7
  - 1 - 2 - 4 - 6 - 7



All Paths Coverage

- Criterion:
  - All paths through the code must be covered.
- This is typically infeasible when loops are present.
  - A version of this coverage with loops is to treat loops as having two paths:
    1. The loop is executed (normally, once).
    2. The loop is skipped.
- Some paths may also be infeasible because there is no combination of data conditions that permit a path to be taken.

All Paths Coverage

- Set of all paths:

  1 - 2 - 4 - 6 - 7

  1 - 3 - 4 - 6 - 7

  1 - 2 - 4 - 5 - 7

  1 - 3 - 4 - 5 - 7

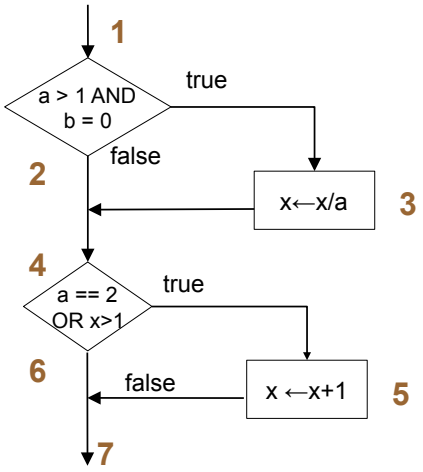
- Input values:

  a = 0, b = 1, x = 0

  a = 3, b = 0, x = 0

  a = 2, b = 1, x = 0

  a = 2, b = 0, x = 0



Comparison

- From the previous two examples, we can see that:
  - Multiple condition coverage does not imply all paths coverage.
  - All paths coverage does not imply multiple condition coverage.

Infeasible paths

- Set of paths:

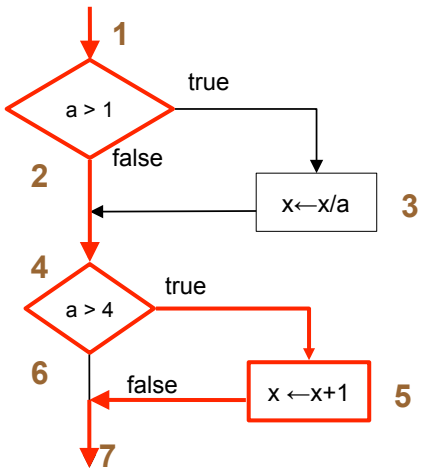
  1 – 2 – 4 – 6 – 7

  1 – 3 – 4 – 6 – 7

  1 – 2 – 4 – 5 – 7

  1 – 3 – 4 – 5 – 7

- To be able to take this path, we would have to have `a <= 1` AND `a > 4` – which is logically impossible!



Appendix A:
Another Example of Code Coverage

Detailed Test Creation Example

- The next few slides will go over the detailed steps to create test cases using a coverage-based approach.
- Situation for the example:
  - We want to create some regression tests for some code we already have.
  - The branch coverage criterion will be used for test selection.

Steps for Test Creation (1)

1. Parse the code, and generate a flow graph.
2. Analyze the flow graph to determine units of test coverage criteria.
3. Determine a set of sub-paths through each structural test unit of interest.
4. Combine sub-paths into a complete path (from entry point to exit point).
5. Trace path in reverse from exit to entry, and collect conditions at each branch point.

Steps for Test Creation (2)

6. Solve set of data conditions required to take complete path.
   - This has the effect of deriving equivalence classes of input data.
   - Any member of the equivalence class should result in executing the same path through the code.
   - Feasibility of data conditions must be checked
     - If no data satisfies input conditions, return to step 4 to search for a different complete path to cover test unit.

Steps to Generate Structural Tests (3)

7. Choose representative members from equivalence classes.
8. Trace path in forward direction, in order to predict expected output.
   - In many cases, this is extremely difficult.
9. Combine input with test environment information.
10. Generate test script.

Example: Delete rows from array (1)

```
public int[][] deleteRows( int[][] anArray,
                           int firstRow, int lastRow )
{
   int[][] result = null;
   int numRows = anArray.length;
   if ( ( firstRow >= numRows ) || ( firstRow < 0 ) )
   {
      System.out.println( "Bad first row." );
   }
   else if ( ( lastRow >= numRows ) || ( lastRow < 0 ) )
   {
      System.out.println( "Bad last row." );
   }
   else if ( lastRow < firstRow )
   {
      System.out.println( "Not a valid range." );
   }
```

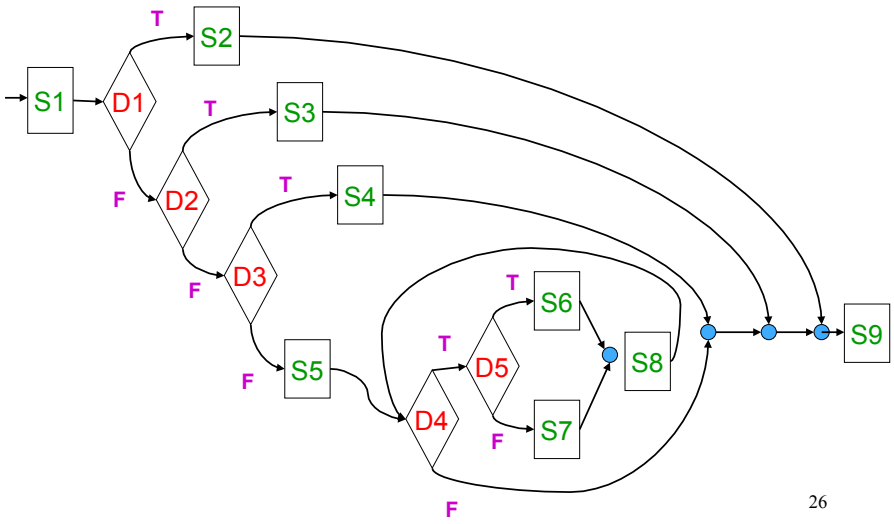
S1 S2 S3 S4 D1 D2 D3

Example: Delete rows from array (2)

```
    else
    {
       int numNewRows = numRows - ( lastRow - firstRow + 1 );
       result = new int[numNewRows][anArray[0].length];
       int offset = 0;
       for ( int row = 0; row < numRows; row++ )
       {
          if ( ( row >= firstRow ) && ( row <= lastRow ) )
          {
             offset++;
          }
          else
          {
             result[row-offset] = anArray[row];
          }
       }
    }
    return result;
}
```

S5 S8 D4 D5 S6 S7 S9

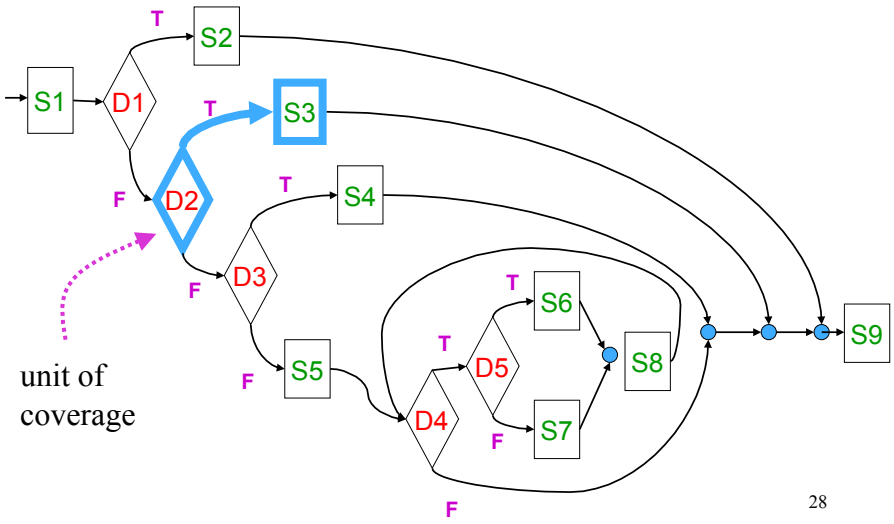
Flow graph



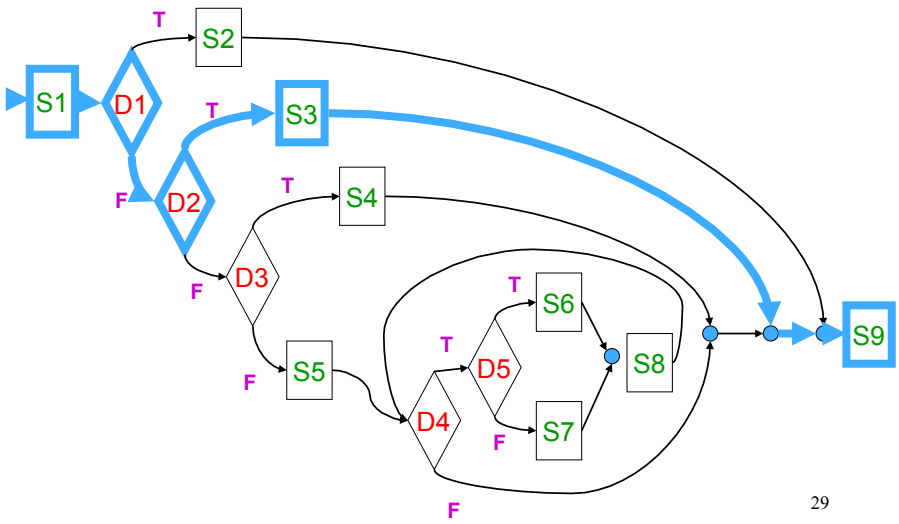
Potential Coverage Measures

- Statement coverage: Cover nodes S1 through S9, and D1 through D5
- Decision coverage: cover true and false cases for each individual condition with D1 through D5
  - Will cover all edges in flow graph.
- Example: use decision coverage

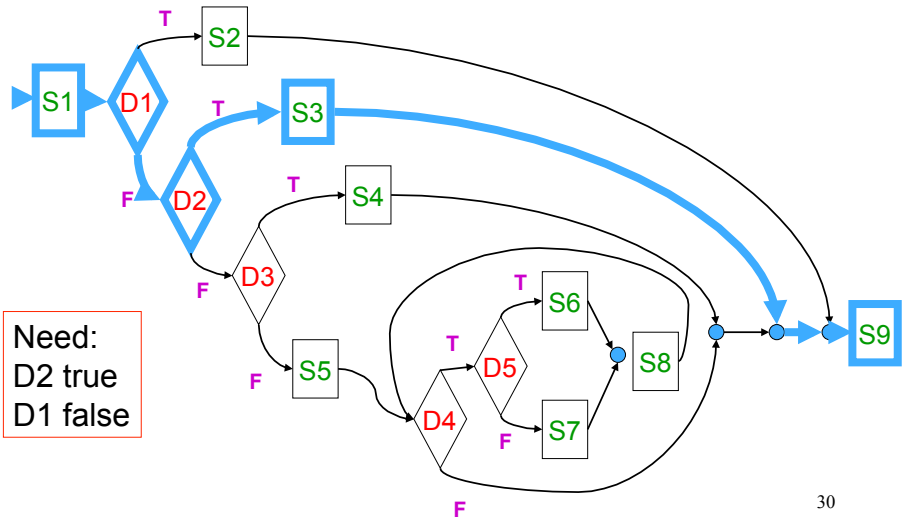
Choose a sub-path for unit of test coverage



Extend sub-path to complete path



Solve conditions



Need:
D2 true
D1 false

Solve conditions

- Decision D1 (false)

```
( firstRow >= numRows ) || ( firstRow < 0 )
```

- Decision D2 (true)

```
( lastRow >= numRows ) || ( lastRow < 0 )
```

- Determine input that affects **numRows**:

```
int numRows = anArray.length;
```

- Therefore, for this path, we need:

```
! ((firstRow >= anArray.length) || (firstRow < 0))
&& ((lastRow >= anArray.length) || (lastRow < 0 ))
```


Equivalence classes (not complete)

- `firstRow`: `(-∞,-1]` `[0,anArray.length-1]` `[anArray.length,+∞)`

S2 S2

- `lastRow`:

`(-∞,-1]` `[0,firstRow-1]` `[firstRow,anArray.length-1]` `[anArray.length,+∞)`

S3 S4 S3

- Need `firstRow` in equivalence class `[0,anArray.length-1]` and `lastRow` from equivalence class `(-∞,-1]` or `[anArray.length,+∞)`

Input Data Selection

- Need to find specific values for each input:
- **anArray**: **anArray.length >= 1**, number of columns unspecified, contents unspecified
  - Choose **anArray.length = 3**, number of columns 2, fill array with ones.
- **firstRow**: must be between 0 and 2 inclusive
  - Choose 1
- **lastRow**: must be -1 or less, or 3 or greater
  - Choose 5

Predict output (1)

```
public int[][] deleteRows( int[][] anArray,
                           int firstRow, int lastRow )
{
   int[][] result = null;
   int numRows = anArray.length;
   if ( ( firstRow >= numRows ) || ( firstRow < 0 ) )
   {
      // not executed
   }
   else if ( ( lastRow >= numRows ) || ( lastRow < 0 ) )
   {
      System.out.println( "Bad last row." );
   }
   else if ( lastRow < firstRow )
   {
      // not executed
   }
```

S1 S2 S3 S4

Predict output (2)

```
    else
    {
       [...] // not executed
    }
S9  return result;
}
```


We have:

1. Output:

   `Bad last row.`

2. Return value:

   `result = null`