Unit Testing

Sandro Morasca

Università degli Studi dell'Insubria

Dipartimento di Scienze Teoriche e Applicate

Via Ottorino Rossi 9 – Padiglione Rossi

I-21100 Varese, Italy

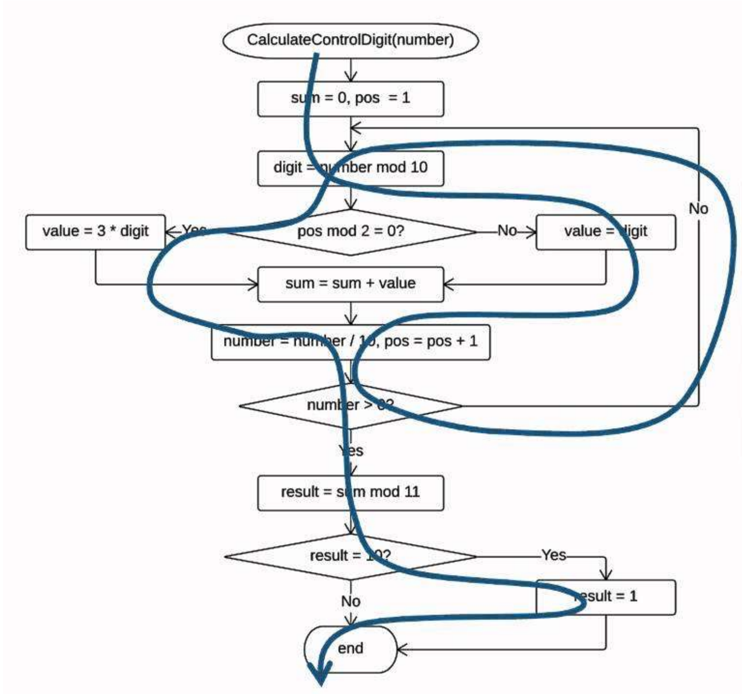
sandro.morasca@uninsubria.i

“Indirect” Testing

- **Testing the system at a high level**
  - “indirectly” from the “outside”
    - black-box testing
- Problem:
  - I don’t know how math.sum is implemented, so how do I know if testing is covering all structural elements?

```java
@Test
public void testSum() {
    MyMath math = new MyMath();
    int result = math.sum(a:3,b:4);
    assertEquals(7, result);
}
```

- **Testing the system at a lower level**
  - "directly" from the "inside"
    - white-box testing
- Benefit:
  - cover structural elements of a software system



Specification-based Testing

- **From formal specifications**
  - can be automated
  - examples: Test case generation from
    - Algebraic specifications
    - Finite state automata
    - Grammars
- **From semi-formal specifications**
  - partitions can be easily identified
  - can be partially automated

From informal specifications

- It cannot be automated
- Some structure (e.g., organization standards) can help
- Guidelines to increase confidence level and reduce discretionality

- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is certainly inadequate
- **Control flow coverage criteria**
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
  - Path coverage
  - Data flow (syntactic dependency) coverage
- Attempted compromise between the impossible and the inadequate

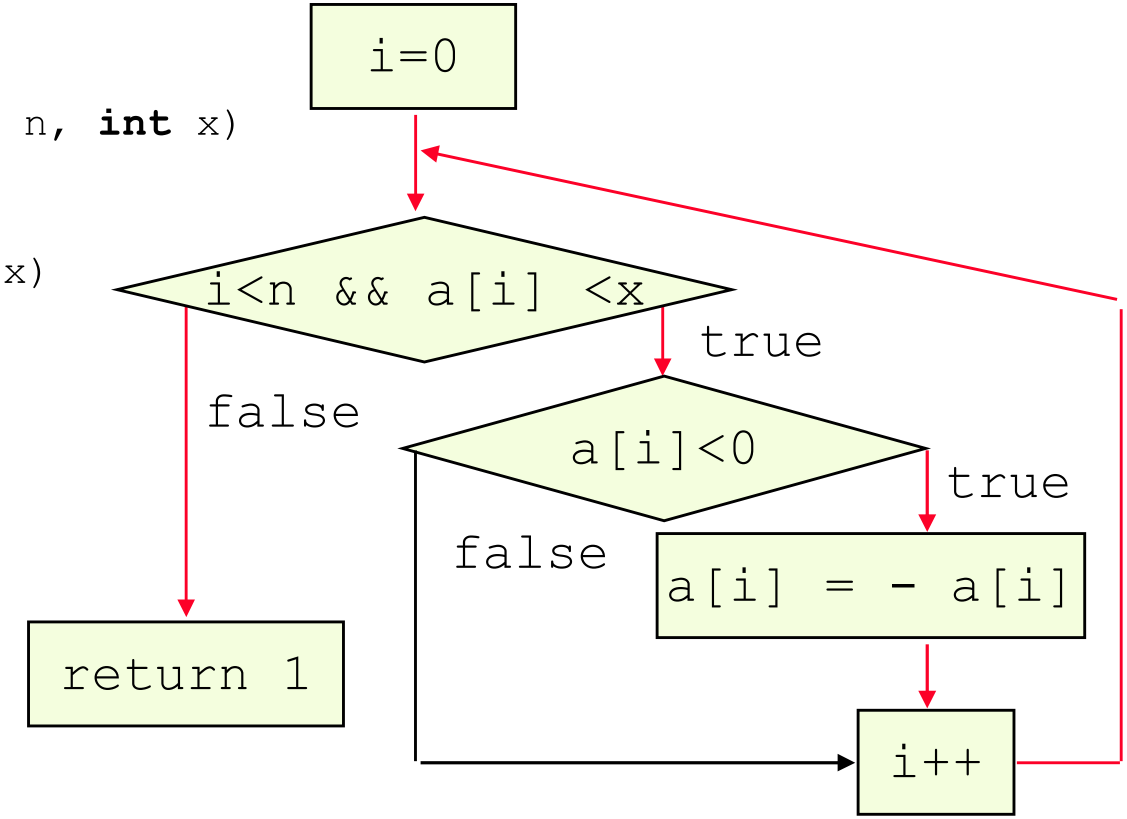
Statement Coverage

- Statement coverage requires that each executable statement be exercised at least once
- Basic idea:
  - a fault in a statement cannot be revealed if the faulty statement is not executed
- Use of control flow graphs to reach all statements
- Figure of merit for testing
  - maximize the percentage of executed statements

Statement Coverage

- One test input (n=1, a[0]=-7, x=9) is enough to guarantee statement coverage of function `select`
- Faults in handling positive values of `a[i]` would not be revealed

```
int select(int a[], int n, int x)
{
   int i=0;
   while (i<n && a[i] < x)
   {
      if(a[i] < 0)
         a[i] = - a[i];
      i++;
   }
   return 1;
}
```



Basic Block Coverage

- Nodes in a control flow graph often represent basic blocks instead of individual statements
  - all the statements of a basic block are always executed together
- Figure of merit for testing
  - maximize the percentage of basic blocks

Branch (Decision) Coverage

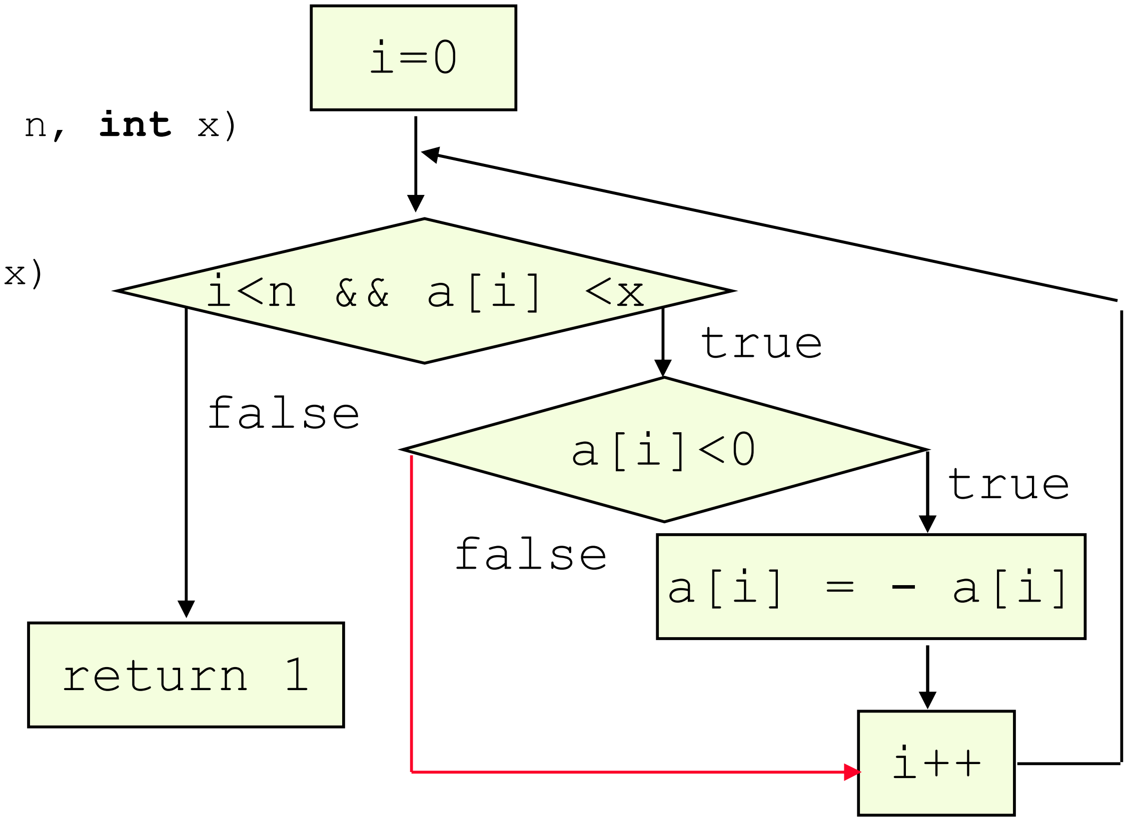
- Branch coverage requires that each branch be exercised at least once
  - not just each statement
- **A "branch" is one of the possible execution paths the code can take after a decision statement**
- Basic idea:
  - a fault in a decision cannot be revealed if the faulty decision is not executed
- Use of control flow graphs to exercise all branches
- Figure of merit for testing
  - maximize the percentage of executed branches

Branch (Decision) Coverage

Basic Concepts
Specifications
➢ Coverage
Fault-based
Infrastructure

- We must add a test input (n=1, a[0]=7, x=9) to cover branch `false` of the `if` statement
  - Faults in handling positive values of a[i] would be revealed
  - Faults in exiting the loop with condition a[i] <x would not be revealed

```
int select(int a[], int n, int x)
{
   int i=0;
   while (i<n && a[i] < x)
   {
      if(a[i] < 0)
         a[i] = - a[i];
      i++;
   }
   return 1;
}
```



Branch (Decision) Coverage

- Another disadvantage of branch coverage is that it ignores branches within boolean expressions which occur due to short-circuit operators
- `function1` may never be executed, even with 100% branch coverage

```
if( condition1 && ( condition2 || function1() ) )
{
  statement1;
}
else
{
  statement2;
}
```

Condition Coverage

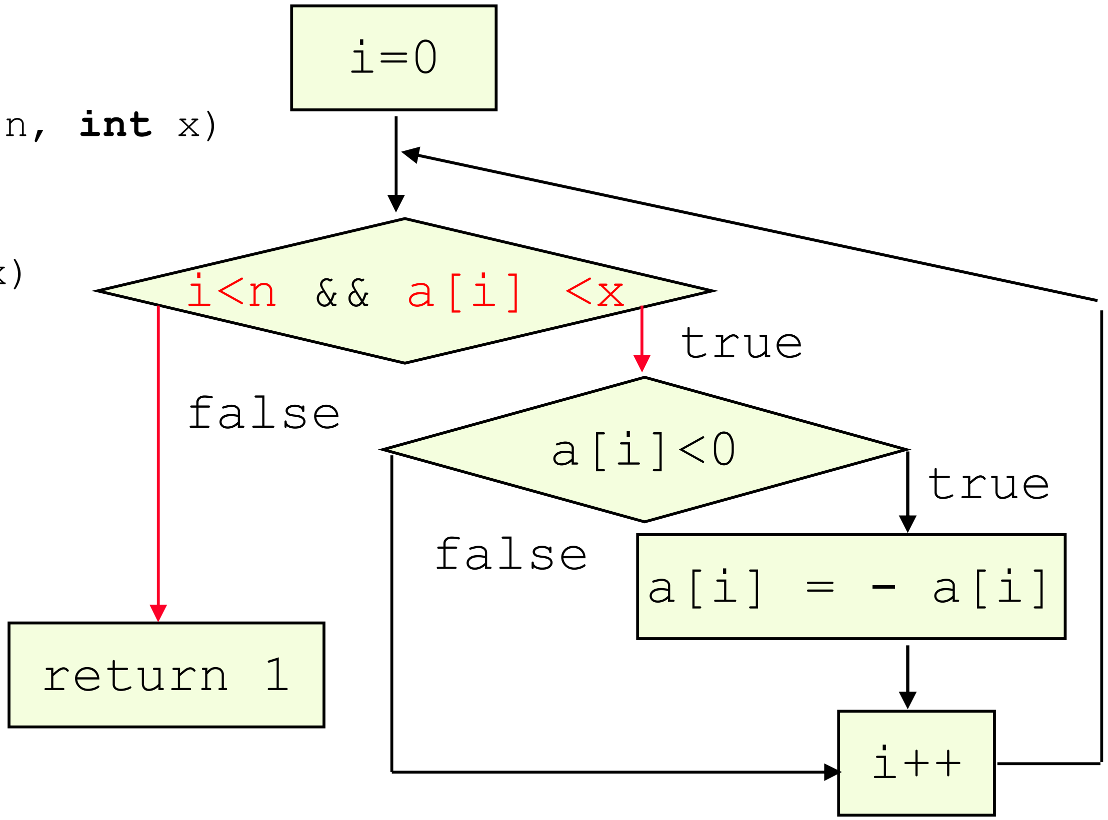
Basic Concepts
Specifications
➢ Coverage
Fault-based
Infrastructure

- Condition coverage requires that each elementary condition in a decision be exercised at least once
  - not just each branch
- Basic idea:
  - a fault in an elementary condition cannot be revealed if the faulty condition is not executed by making it true and false
- Figure of merit for testing
  - maximize the percentage of “executed” elementary conditions

Condition Coverage

- Conditions (i<n), (a[i]<x) must be made `false` and `true`
  - we must add tests that cause the while loop to exit for a value of a[i] not less than x
  - faults that arise after several iterations of the loop would not be revealed

```
int select(int a[], int n, int x)
{
   int i=0;
   while (i<n && a[i] < x)
   {
      if(a[i] < 0)
         a[i] = - a[i];
      i++;
   }
   return 1;
}
```



Full Condition Coverage

- With n elementary conditions
  - 2^n test cases
  - not all of them may be feasible
    - infeasibility problem
- This kind of coverage may be
  - tedious
  - expensive

Full Condition Coverage

- Expression: (((a || b) && c) || d) && e

	a	b	c	d	e	value
1	true	-	true	-	true	true
2	false	true	true	-	true	true
3	true	-	false	true	true	true
4	false	true	false	true	true	true
5	false	false	-	true	true	true
6	true	-	true	-	false	false
7	false	true	true	-	false	false
8	true	-	false	true	false	false
9	false	true	false	true	false	false
10	false	false	-	true	false	false
11	true	-	false	false	-	false
12	false	true	false	false	-	false
13	false	false	-	false	-	false

Modified Condition/Decision Coverage

- Modified Condition/Decision Coverage requires that each basic condition be shown to independently affect the outcome
- For each basic condition, there are two test cases
  - all other conditions are the same, and
  - the compound condition as a whole evaluates to true for one of the two test cases and false for the other
- Tradeoff between number of required test cases and thoroughness of the testing
  - required for aviation software (standard RCTA/DO-178B)
- Test cases required for a || b
  - a = true, b = false
  - a = false, b = true
  - a = false, b = false

Modified Condition/Decision Coverage

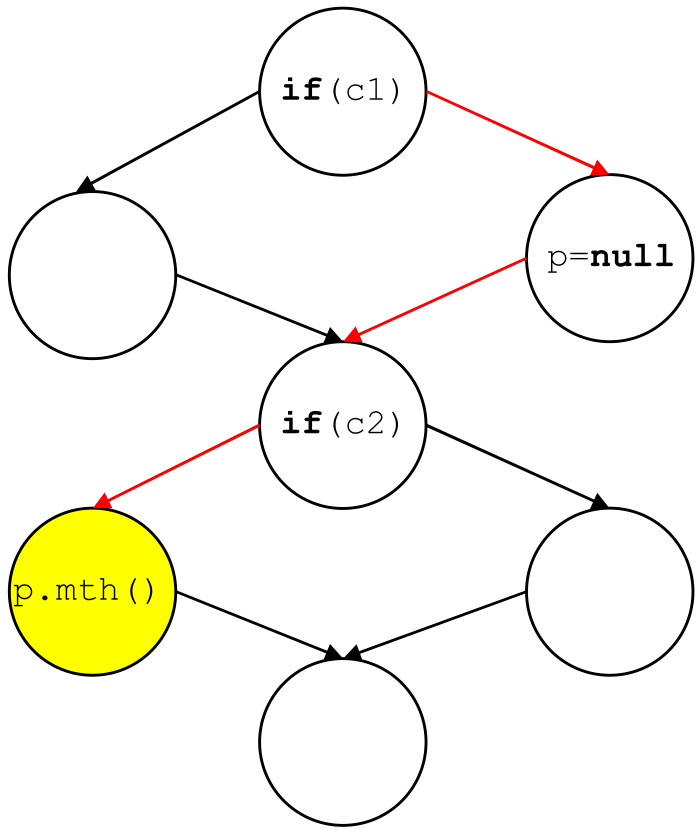
- Expression: (((a || b) && c) || d) && e

	a	b	c	d	e	value
1	**true**	-	**true**	-	**true**	true
2	false	**true**	true	-	true	true
3	true	-	false	**true**	true	true
6	true	-	true	-	**false**	false
11	true	-	**false**	**false**	-	false
13	**false**	**false**	-	false	-	false

Path Coverage

- All paths need to be exercised by at least one test case
  - sometimes a fault is revealed only by exercising some sequence of decisions

```
void m()
{
   Obj p = new …;
   if(c1)
   {
       p = null;
   }
   if(c2)
   {
       p.mth();
   }
}
```

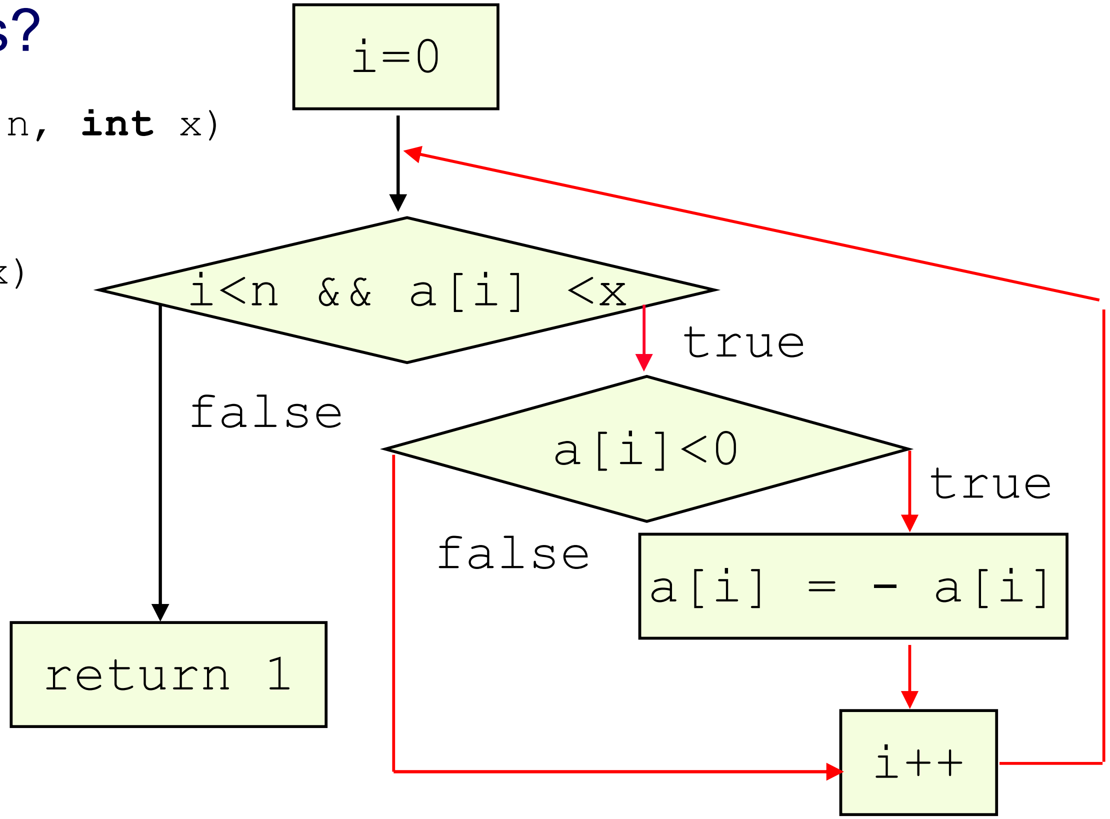


Path Coverage

Basic Concepts
Specifications
➢ Coverage
Fault-based
Infrastructure

- A path is a unique sequence of branches from the entry point to the exit point
  - how about loops?

```
int select(int a[], int n, int x)
{
   int i=0;
   while (i<n && a[i] < x)
   {
      if(a[i] < 0)
         a[i] = - a[i];
      i++;
   }
   return 1;
}
```



Path Coverage

- Problems with loops
  - unbounded number of paths
  - unbounded number of test cases
- Possibile approximated solutions
  - skip the loop (0 iterations)
  - check the loop once (1 iteration)
  - exactly n executions (n iterations)
  - 1, 2, …, n executions
  - (n-1), n, (n+1) iterations

Path Coverage

- Other problems
  - even programs with only if-then-else's may cause a combinatorial explosion
    - with n if-then-else's there are 2^n paths
- Some paths may not be feasible

```
if( bool )
{
    statement1; //does not change the value of bool
}
statement2; //does not change the value of bool
if( bool )
{
    statement3;
}
```

- Even exhaustive path coverage with loops would not give the certainty of correctness

Base Path Coverage

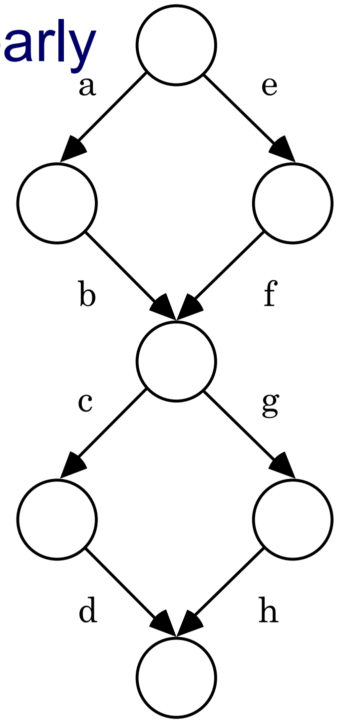
Basic Concepts
Specifications
➢ Coverage
Fault-based
Infrastructure

- Test each base path
- The cyclomatic number is a measure of the control flow complexity of a program
- The control flow of a program can be represented by its control flow graph G, with one entry node and one exit node
- v(G), the cyclomatic number of the program is the number of base paths of G, i.e., the number of linearly independent paths from the entry to the exit node
- A linearly independent path is any path through the program that introduces at least one new edge that is not included in any other linearly independent paths.

Base Path Coverage

Basic Concepts
Specifications
➢ Coverage
Fault-based
Infrastructure

- Out of the four possible paths only three are linearly independent, e.g., Z = W + X - Y



	a	b	c	d	e	f	g	h
W	1	1	1	1	0	0	0	0
X	0	0	0	0	1	1	1	1
Y	1	1	0	0	0	0	1	1
Z	0	0	1	1	1	1	0	0

Base Path Coverage

- If G is strongly connected, then $v(G) = e - n + 1$, where
  - e: number of edges of G
  - n: number of nodes of G
- A control flow graph becomes connected if an edge is inserted from the exit node to the entry node. Therefore, for control flow graphs

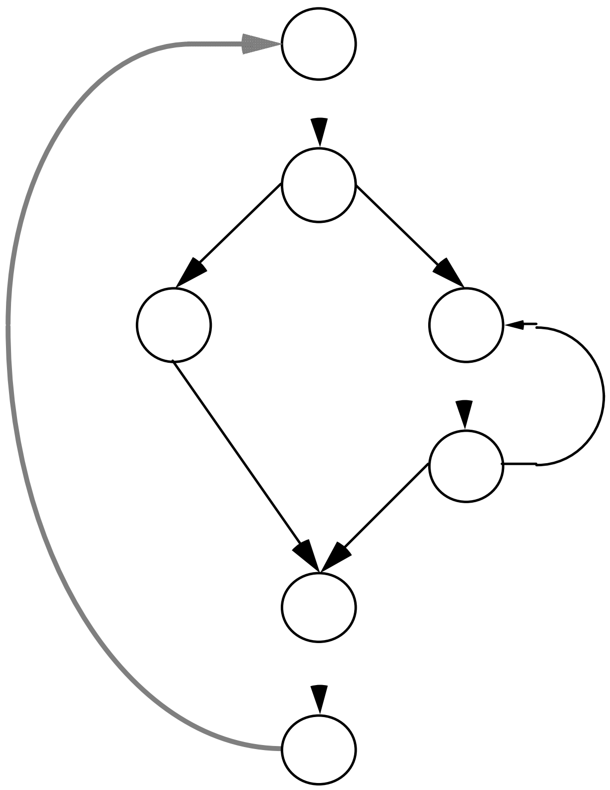
  $v(G) = e - n + 2$
- If G has p connected components (i.e., there is a main program and p-1 subprograms

  $v(G) = e - n + 2p$
- (Mills' theorem) If d is the number of decision nodes of G, then (an r-way decision node contributes r-1)

  $v(G) = d + p$

  i.e., we do not need to build graph G

Base Path Coverage

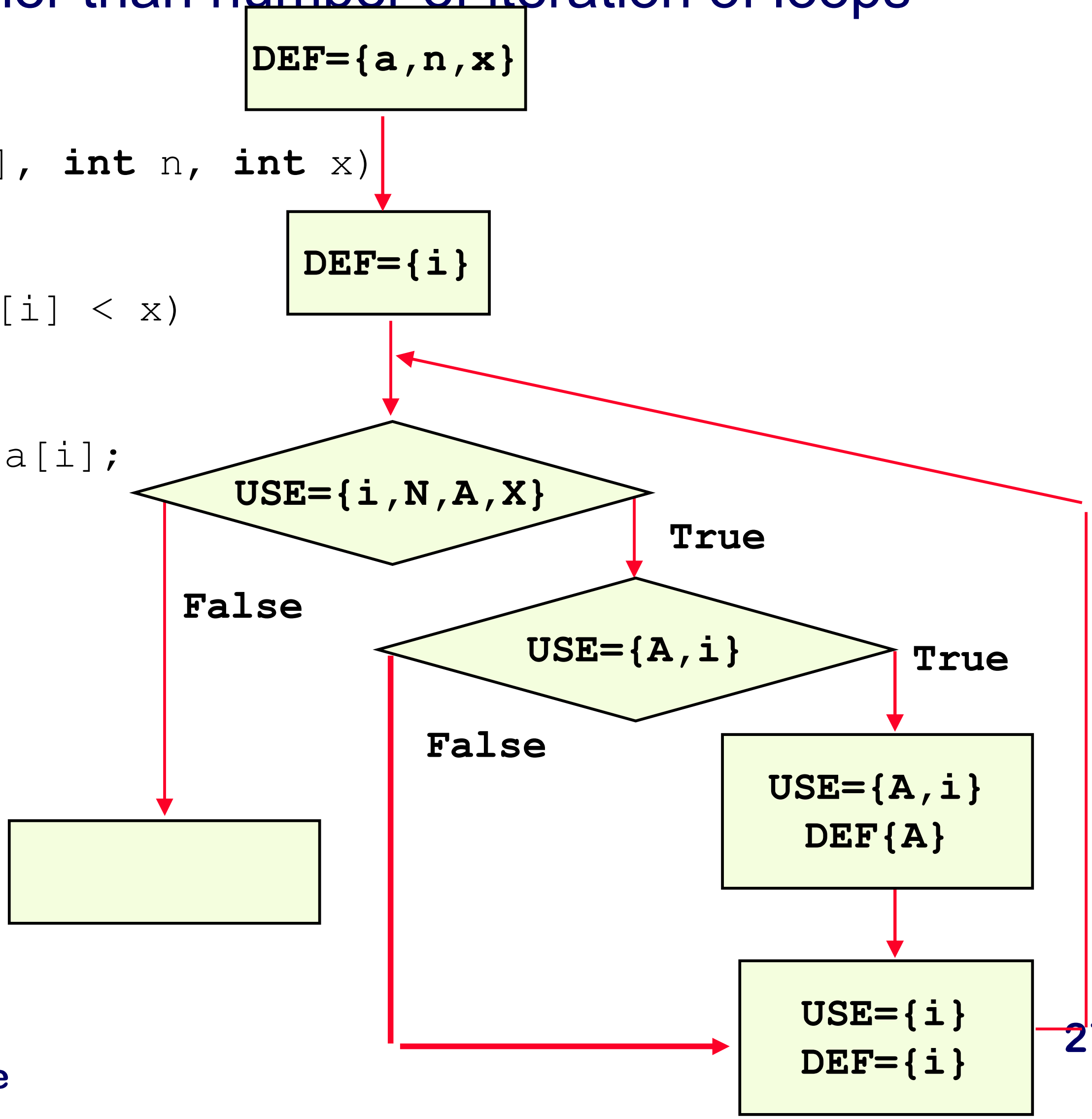


```
S1;
if C
   then S2
   else
     repeat
       S3
     until D;
S4;
```

$$v(G) = e - n + 2 = 8 - 7 + 2 = 3$$

- Exercise Def-Use paths: selects paths based on effects on the variables, rather than number of iteration of loops

```
int select(int a[], int n, int x)
{
   int i=0;
   while (i<n && a[i] < x)
   {
      if(a[i] < 0)
         a[i] = - a[i];
      i++;
   }
   return 1;
}
```



Method Coverage

- In this case, a unit is a class
  - exercise all methods in a class
  - useful during integration testing or system testing

Coverage Goals

- From the literature: 80% - 90%
  - having 100% coverage makes you feel better
  - but it may be very expensive
- Avoid setting a goal of less than 80%
  - typical industrial goal: 85%
  - however, many successful industry projects only reach 50% - 70%
- What type of coverage?
  - start with statement coverage …
  - … then move to the other ones

The Budget Coverage Criterion

- Industry’s answer to “when is testing done”
  - when the money is used up
  - when the deadline is reached
- This is sometimes a rational approach!
  - Implication 1
    - adequacy criteria answer the wrong question
    - selection is more important.
  - Implication 2
    - practical comparison of approaches must consider the cost of test case selection

Challenges in Structural Coverage

- Interprocedural and cross-level coverage
  - e.g., interprocedural data flow, call-graph coverage
- Regression testing
- Late binding (OO programming languages)
  - coverage of actual and apparent polymorphism
- Fundamental challenge: Infeasible behaviors
  - underlies problems in inter-procedural and polymorphic coverage, as well as obstacles to adoption of more sophisticated coverage criteria and dependence analysis

The Infeasibility Problem

- Syntactically indicated behaviors (paths, data flows, etc.) are often impossible
  - infeasible control flow, data flow, and data states
- Adequacy criteria are typically impossible to satisfy
- approaches
  - manual justification for omitting each impossible test case (especially for more demanding criteria)
  - adequacy “scores” based on coverage
    - example: 95% statement coverage, 80% def-use coverage

Fault-based Testing

- Identify a set of program locations (related to specific faults)
- Generate alternate programs by seeding faults in the original program in the identified locations
- Generate test cases to estimate adequacy in detecting real faults from adequacy in detecting seeded faults

Effective Fault-based testing

- Depend on a fault model (set of possible deviations from the correct program)
- Extremely successful for hardware testing
  - very good fault models
  - excellent for testing silicon (implementations)
  - BUT useless for catching design problems, e.g., the Pentium bug
- Less effective for software testing:
  - lack of good fault models
  - many faults derive from design errors

Mutation Analysis

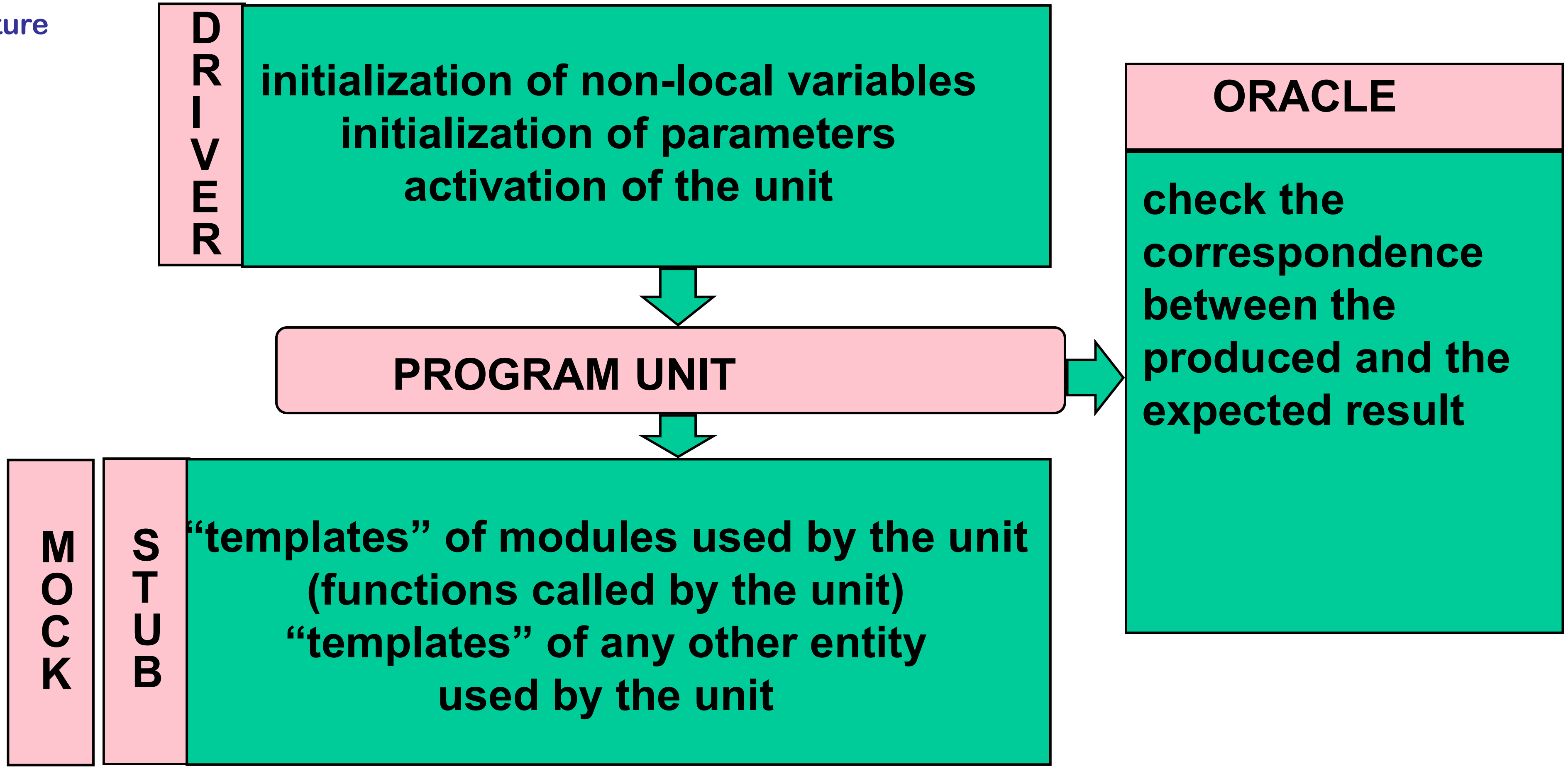
- An example of software fault-based testing
- Alternate programs (mutants) are (automatically) generated by syntactically modifying the original program (e.g., substituting operators, variables, ...)
- Can be used for assessing the *quality of a test set* as the number of undistinguishable mutants)

Mutation Analysis

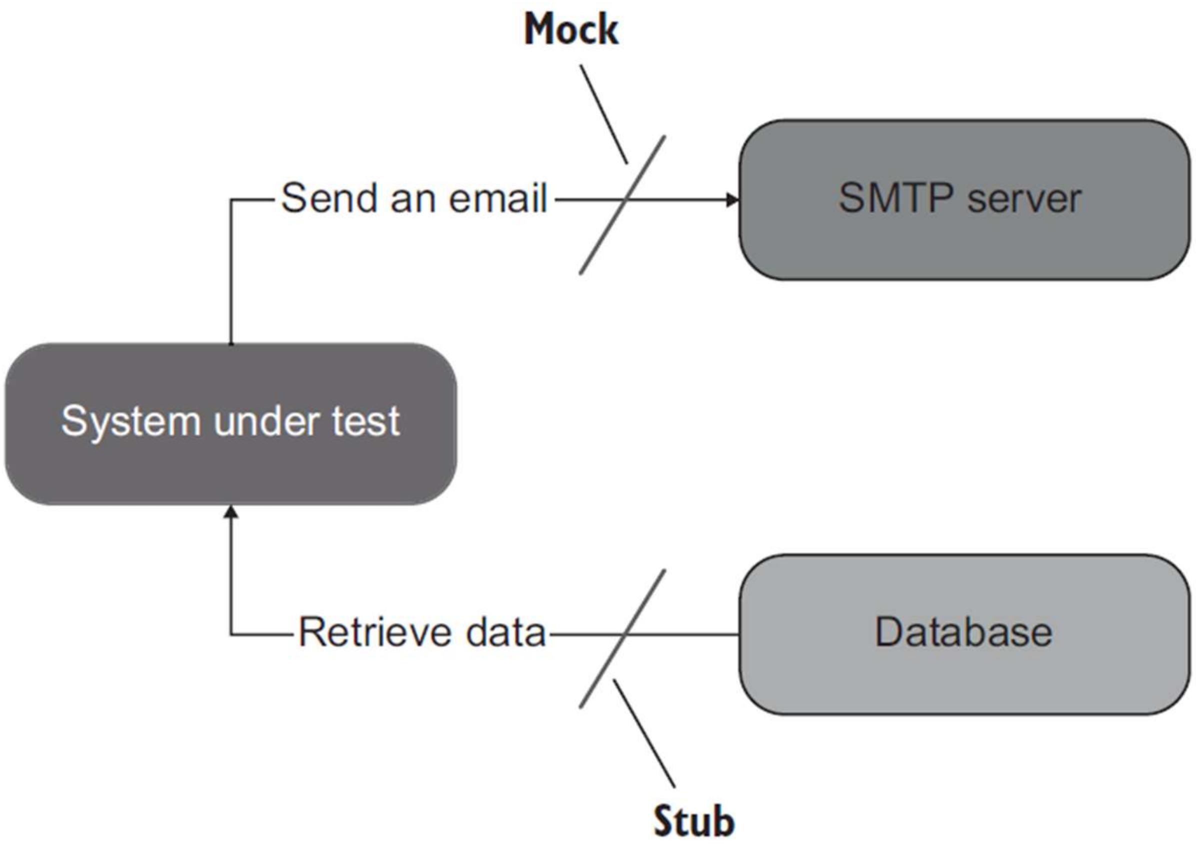
- Mutation examples
  - changing literal numbers: e.g., 0 is changed to 1
  - changing true to false and vice versa
  - changing if(to if(true ||
  - changing if(to if(false &&
- If the test suite does not kill the mutant, then it is not very good

Create Scaffolding

- Goal
  - setup the environment for executing the test



Mock VS Stub


Mock
Send an email
SMTP server
System under test
Retrieve data
Database
Stub

Generate Drivers and Stubs

generic (for all tests)

specific (for subsets of tests)

Brute force coding

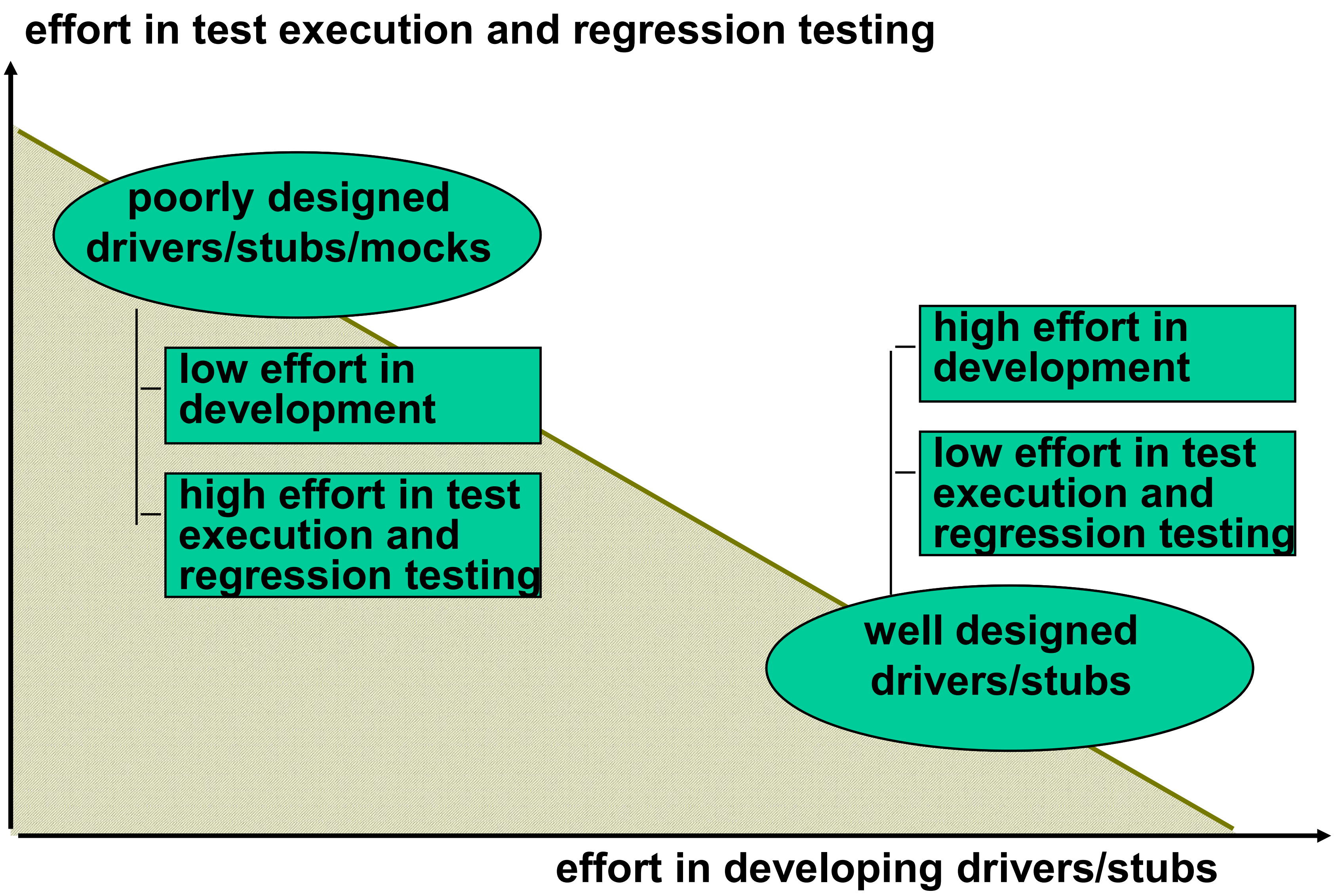
interactive: ask user for values

automatic: (approximately) compute required values

From driver/stub/mock specs

- parsing the unit to partially generate the framework
- add scripts to fill in the framework

Problems and Tradeoffs



Issues

- Objects cannot be set up in the state to be tested
  - or only with a lot of effort
- Components may have side effects
  - tests may not be repeatable
- Failures may not be caused
  - or only with a lot of effort
- Long-running services
- Classes with many dependencies
- Components that are changed often
- Classes that have not yet been implemented