Structural Testing: Path-Based Coverage

CSCE 247 - Lecture 23 - 04/17/2019

We Will Cover

- Additional structural testing strategies
  - Path-based testing strategies
  - Procedure coverage
- Challenges of structural testing
  - Infeasibility problem
  - Sensitivity to structure and oracle

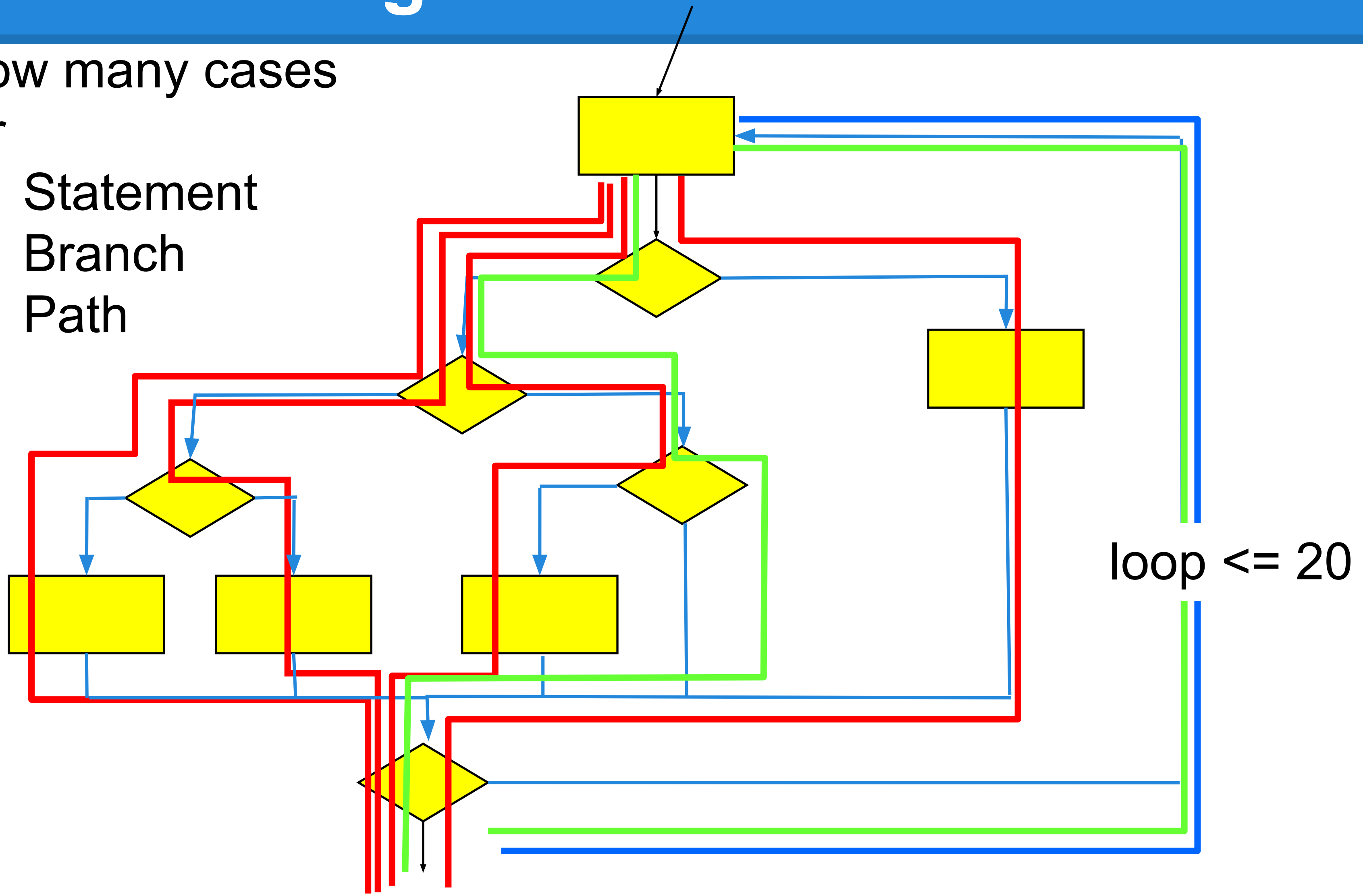
Path Coverage

- Other criteria focus on single elements.
  - However, all tests execute a sequence of elements - a path through the program.
  - Combination of elements matters - interaction sequences are the root of many faults.
- Path coverage requires that all paths through the CFG are covered.
- $$\text{Coverage} = \frac{\text{Number of Paths Covered}}{\text{Number of Total Paths}}$$

Path Testing

How many cases for

- Statement
- Branch
- Path



Number of Tests

Path coverage for that loop bound requires:
3,656,158,440,062,976 test cases

If you run 1000 tests per second, this will take **116,000 years**.

However, there are ways to get some of the benefits of path coverage without the cost...

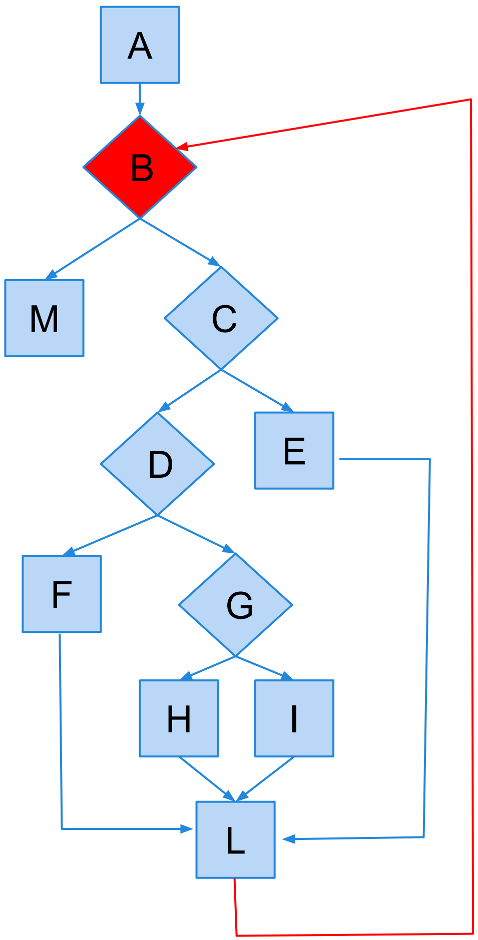
Path Coverage

- Theoretically, the strongest coverage metric.
    - Many faults emerge through sequences of interactions.
- But… Generally impossible to achieve.
    - Loops result in an infinite number of path variations.
    - Even bounding number of loop executions leaves an infeasible number of tests.

Boundary Interior Coverage

- Need to partition the infinite set of paths into a finite number of classes.
- **Boundary Interior Coverage** groups paths that differ only in the subpath they follow when repeating the body of a loop.
    - Executing a loop 20 times is a different path than executing it twice, but the same *subsequences* of statements repeat over and over.

Boundary Interior Coverage



B -> M
B -> C -> E -> L -> B
B -> C -> D -> F -> L -> B
B -> C -> D -> G -> H -> L -> B
B -> C -> D -> G -> I -> L -> B

Number of Paths

- Boundary Interior Coverage removes the problem of infinite loop-based paths.
- However, the number of paths through this code can still be exponential.
    - N non-loop branches results in 2^N paths.
- Additional limitations may need to be imposed on the paths tested.

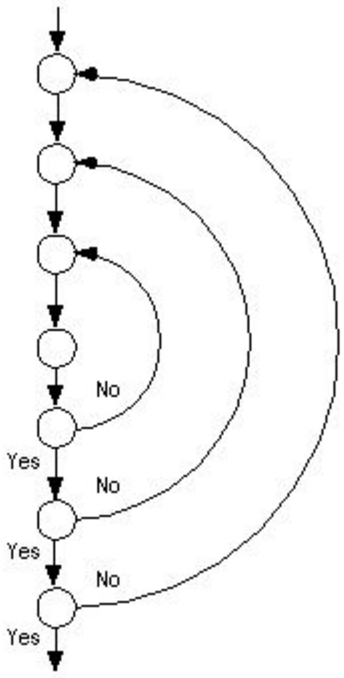
```
if (a)      S1;
if (b)      S2;
if (c)      S3;
...
if (x)      SN;
```

Loop Boundary Coverage

- Focus on problems related to loops.
- Cover *scenarios representative of how loops might be executed*.
- For simple loops, write tests that:
  - Skip the loop entirely.
  - Take exactly one pass through the loop.
  - Take two or more passes through the loop.
  - (optional) Choose an upper bound N, and:
    - M passes, where $2 < M < N$
    - (N-1), N, and (N+1) passes

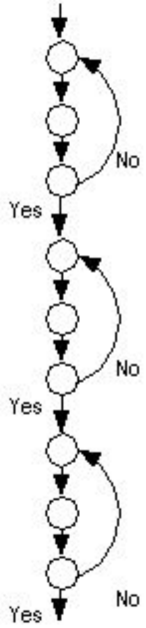
Nested Loops

- Often, loops are nested within other loops.
- For each level, you should execute similar strategies to simple loops.
- In addition:
  - Test innermost loop first with outer loops executed minimum number of times.
  - Move one loops out, keep the inner loop at “typical” iteration numbers, and test this layer as you did the previous layer.
  - Continue until the outermost loop tested.



Concatenated Loops

- One loop executes. The next line of code starts a new loop.
- These are generally independent.
  - Most of the time...
- If not, follow a similar strategy to nested loops.
  - Start with bottom loop, hold higher loops at minimal iteration numbers.
  - Work up towards the top, holding lower loops at “typical” iteration numbers.



Why These Loop Strategies?

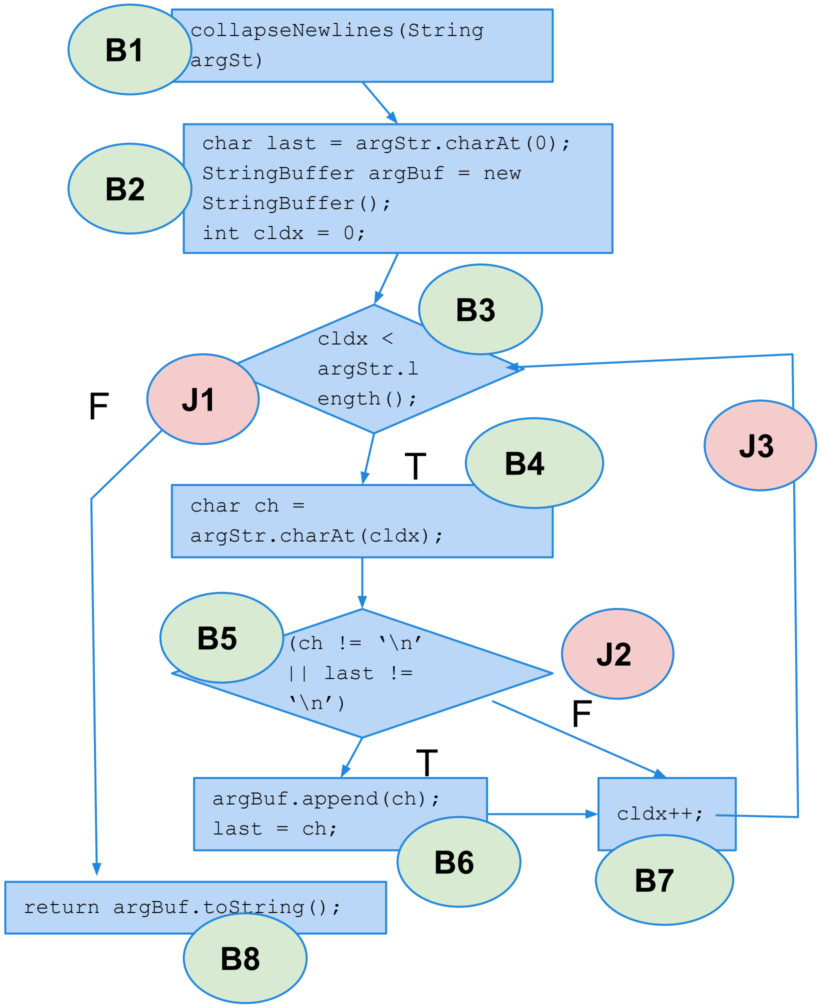
Why do these loop values make sense?

- In proving formal correctness of a loop, we would establish preconditions, postconditions, and invariants that are true on each execution of the loop, then prove that these hold.
  - The loop executes **zero** times when the postconditions are true in advance.
  - The loop invariant is true on loop entry (**one**), then each loop iteration maintains the invariant (**many**).
    - (invariant and !(loop condition) implies postconditions)
- Loop testing strategies echo these cases.

Linear Code Sequences and Jumps

- Often, we want to reason about the subpaths that execution can take.
- A subpath from one branch of control to another is called a LCSAJ.
- The LCSAJs for this example:

From	To	Sequence of Basic Blocks
entry	j1	b1, b2, b3
entry	j2	b1, b2, b3, b4, b5
entry	j3	b1, b2, b3, b4, b5, b6, b7
j1	return	b8
j2	j3	b7
j3	j2	b3, b4, b5
j3	j3	b3, b4, b5, b6, b7



LCSAJ Coverage

- We can require coverage of all sequences of LCSAJs of length *N*.
  - We can string subpaths into paths that connect *N* subpaths.
  - LCSAJ Coverage (N=1) is equivalent to statement coverage.
  - LCSAJ Coverage (N=2) is equivalent to branch coverage
- Higher values of N achieve stronger levels of path coverage.
- Can define a threshold that offers stronger tests while remaining affordable.

Procedure Call Testing

- Metrics covered to this point all look at code *within* a procedure.
- Good for testing individual units of code, but not well-suited for integration testing.
  - i.e., subsystem or system testing, where we bring together units of code and test their combination.
- Should also cover connections between procedures:
  - **calls** and **returns**.

Entry and Exit Testing

- A single procedure may have several entry and exit points.
  - In languages with goto statements, labels allow multiple entry points.
  - Multiple returns mean multiple exit points.
- Write tests to ensure these entry/exit points are entered and exited in the context they are intended to be used.

```
int status (String str){
    if(str.equals("panic"))
        return 0;
    else if(str.contains("+"))
        return 1;
    else if(str.contains("-"))
        return 2;
    else
        return 3;
}
```

- Finds interface errors that statement coverage would not find.

Call Coverage

- A procedure might be called from multiple locations.
- Call coverage requires that a test suite executes all possible method calls.
- Also finds interface errors that statement/branch coverage would not find.

```
void orderPizza (String str){
    if(str.contains("pepperoni"))
        addTopping("pepperoni");
    if(str.contains("onions"))
        addTopping("onions");
    if(str.contains("mushroom"))
        addTopping("mushroom")
}
```

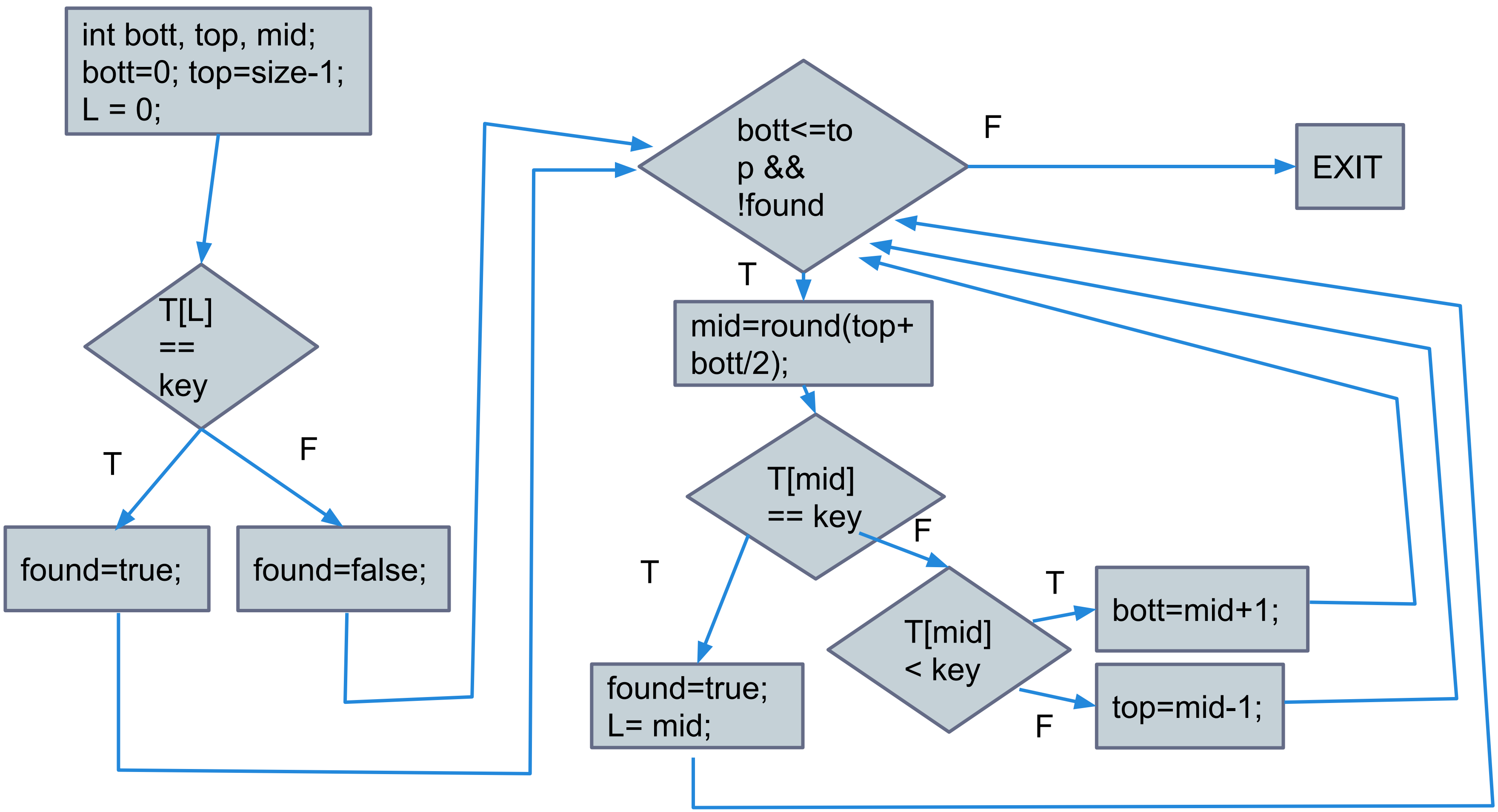
- Challenging for OO systems, where a method call might be bound to different objects at runtime.

Activity: Writing Loop-Covering Tests

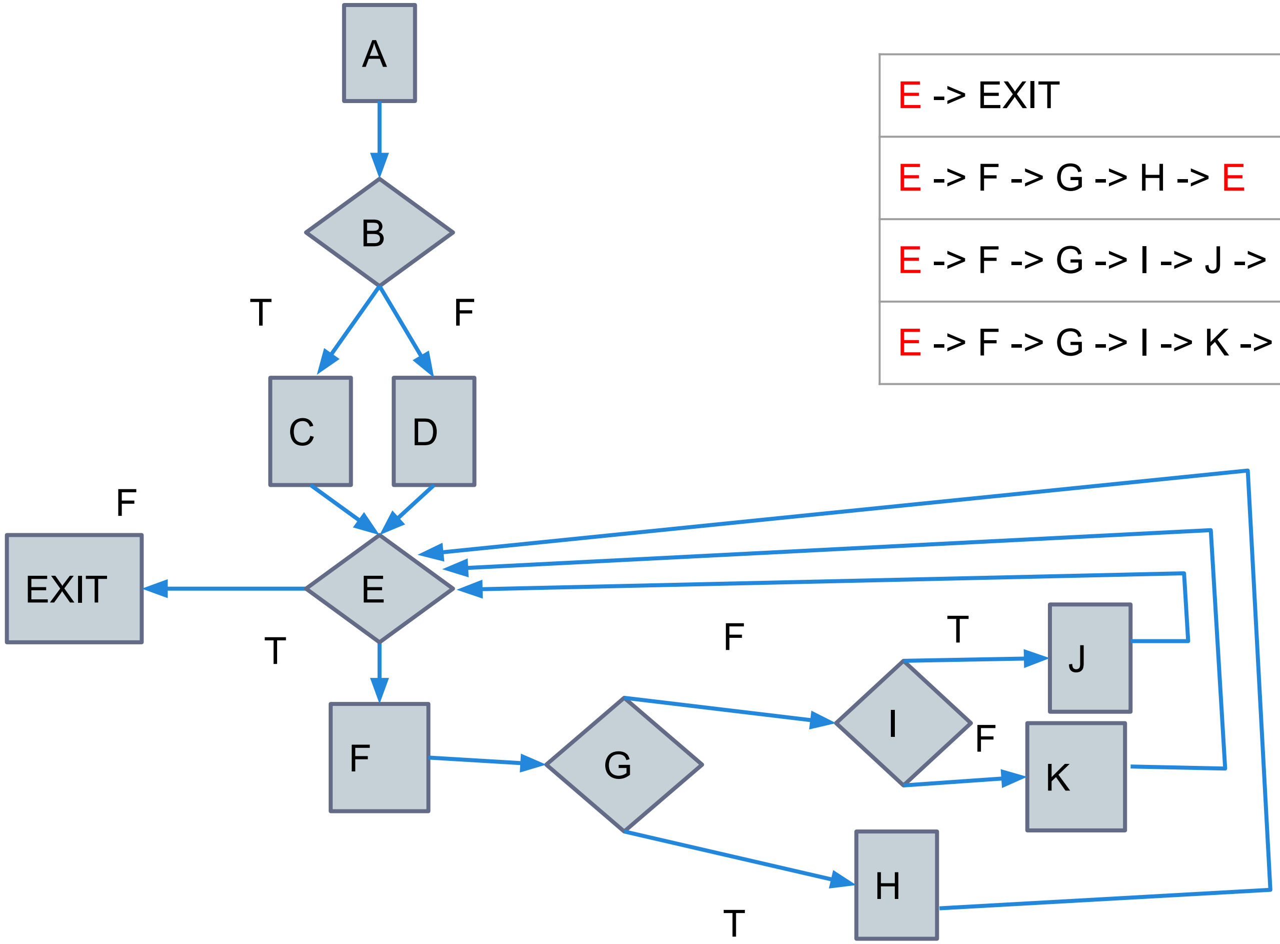
For the binary-search code:

1. Draw the control-flow graph for the method.
2. Identify the subpaths through the loop and draw the unfolded CFG for boundary interior testing.
3. Develop a test suite that achieves loop boundary coverage.

CFG

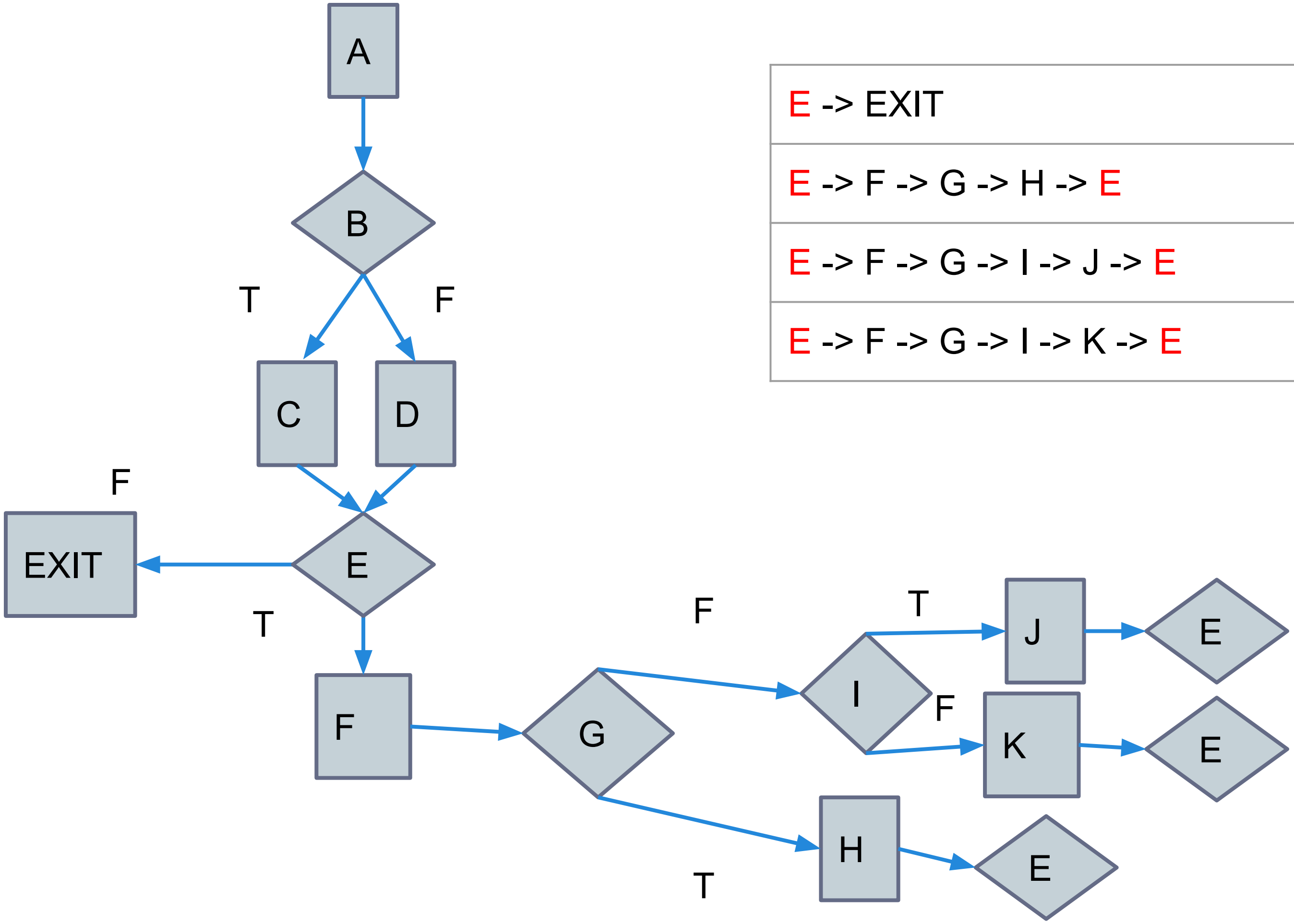

int bott, top, mid;
bott=0; top=size-1;
L = 0;
T[L]
==
key
T
F
found=true;
found=false;
bott<=to
p &&
!found
F
EXIT
T
mid=round(top+
bott/2);
T[mid]
== key
F
T
T[mid]
< key
T
bott=mid+1;
found=true;
L= mid;
top=mid-1;
F

CFG



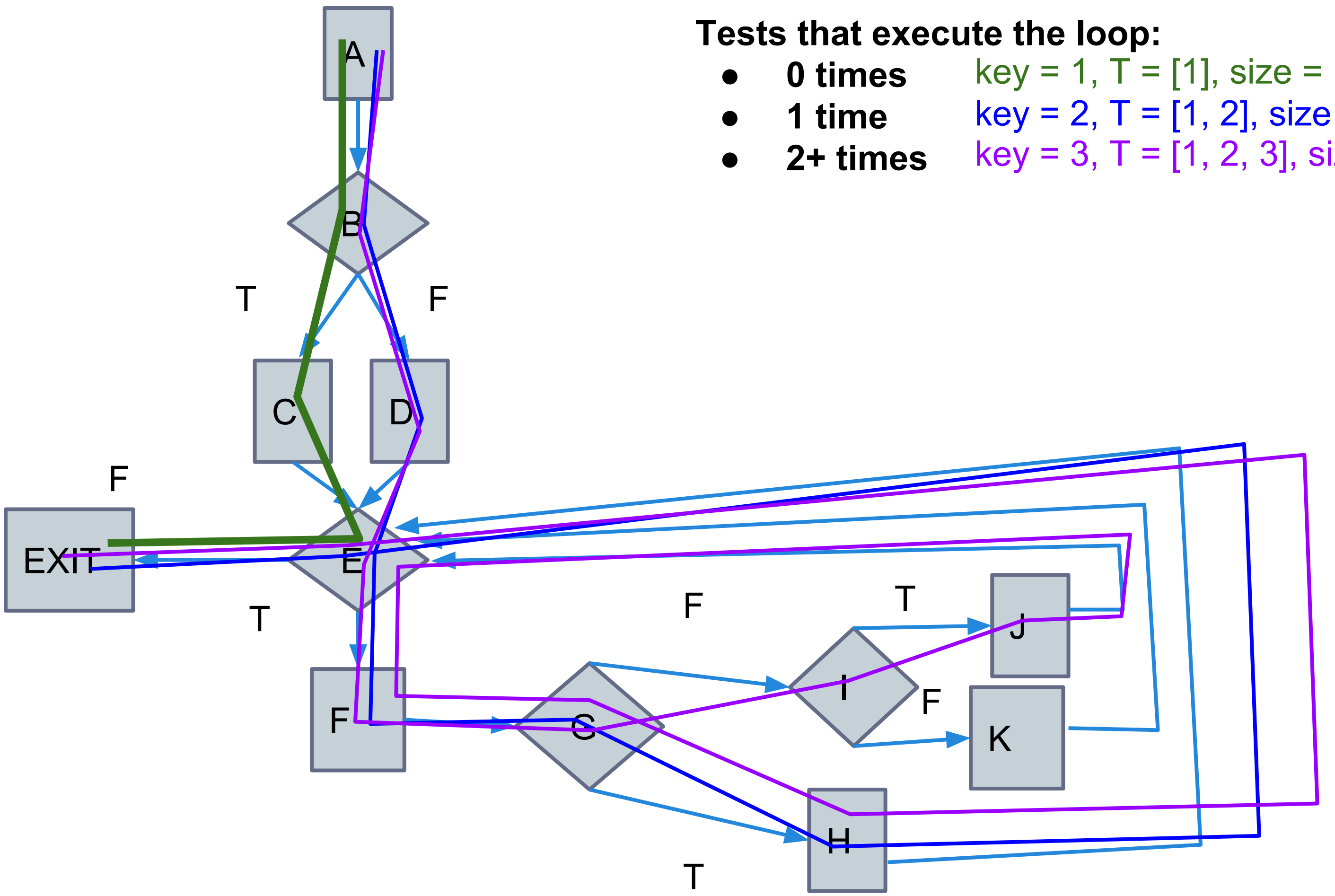
E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



E -> EXIT
E -> F -> G -> H -> E
E -> F -> G -> I -> J -> E
E -> F -> G -> I -> K -> E

CFG



Tests that execute the loop:

- **0 times** key = 1, T = [1], size = 1
- **1 time** key = 2, T = [1, 2], size = 2
- **2+ times** key = 3, T = [1, 2, 3], size = 3

The Infeasibility Problem

Sometimes, **no** test can satisfy an obligation.

- Impossible combinations of conditions.
- Unreachable statements as part of defensive programming.
  - Error-handling code for conditions that can’t actually occur in practice.
- Dead code in legacy applications.
- Inaccessible portions of off-the-shelf systems.

The Infeasibility Problem

Stronger criteria call for potentially infeasible combinations of elements.

```
(a > 0 && a < 10)
```

It is not possible for both conditions to be false.

Problem compounded for path-based coverage criteria.

Not possible to traverse the path where both if-statements evaluate to true.

```
if (a < 0)  a = 0;
if (a > 10) a = 10;
```

The Infeasibility Problem

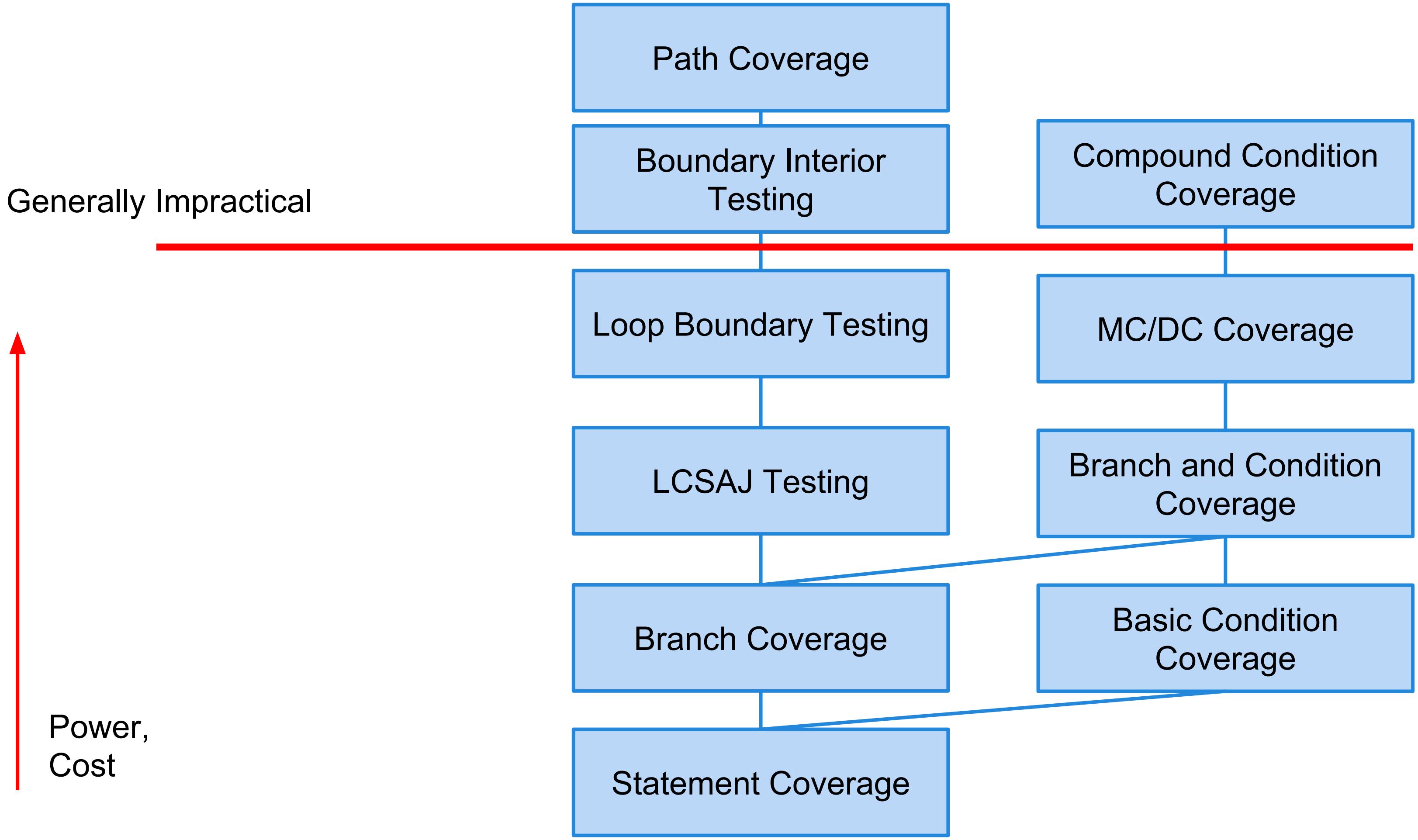
How this is usually addressed:

- Adequacy “scores” based on coverage.
  - 95% branch coverage, 80% MC/DC coverage, etc.
  - Decide to stop once a threshold is reached.
  - Unsatisfactory solution - elements are not equally important for fault-finding.
- Manual justification for omitting each impossible test obligation.
  - Required for safety certification in avionic systems.
  - Helps refine code and testing efforts.
  - … but **very** time-consuming.

In Practice.. 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

Which Coverage Metric Should I Use?



Where Coverage Goes Wrong...

- Testing can only reveal a fault when execution of the faulty element causes a failure, but…
- Execution of a line containing a fault does not guarantee a failure.
  - (a <= b) accidentally written as (a >= b) - the fault will not manifest as a failure if a==b in the test case.
- Merely executing code does not guarantee that we will find all faults.

Don’t Rely on Metrics

- There is a *small* benefit from using coverage as a stopping criterion.
- But, auto-generating tests with coverage as the goal produces poor tests.
- Two key problems - sensitivity to how code is written, and whether infected program state is noticed by oracle.

Sensitivity to Structure

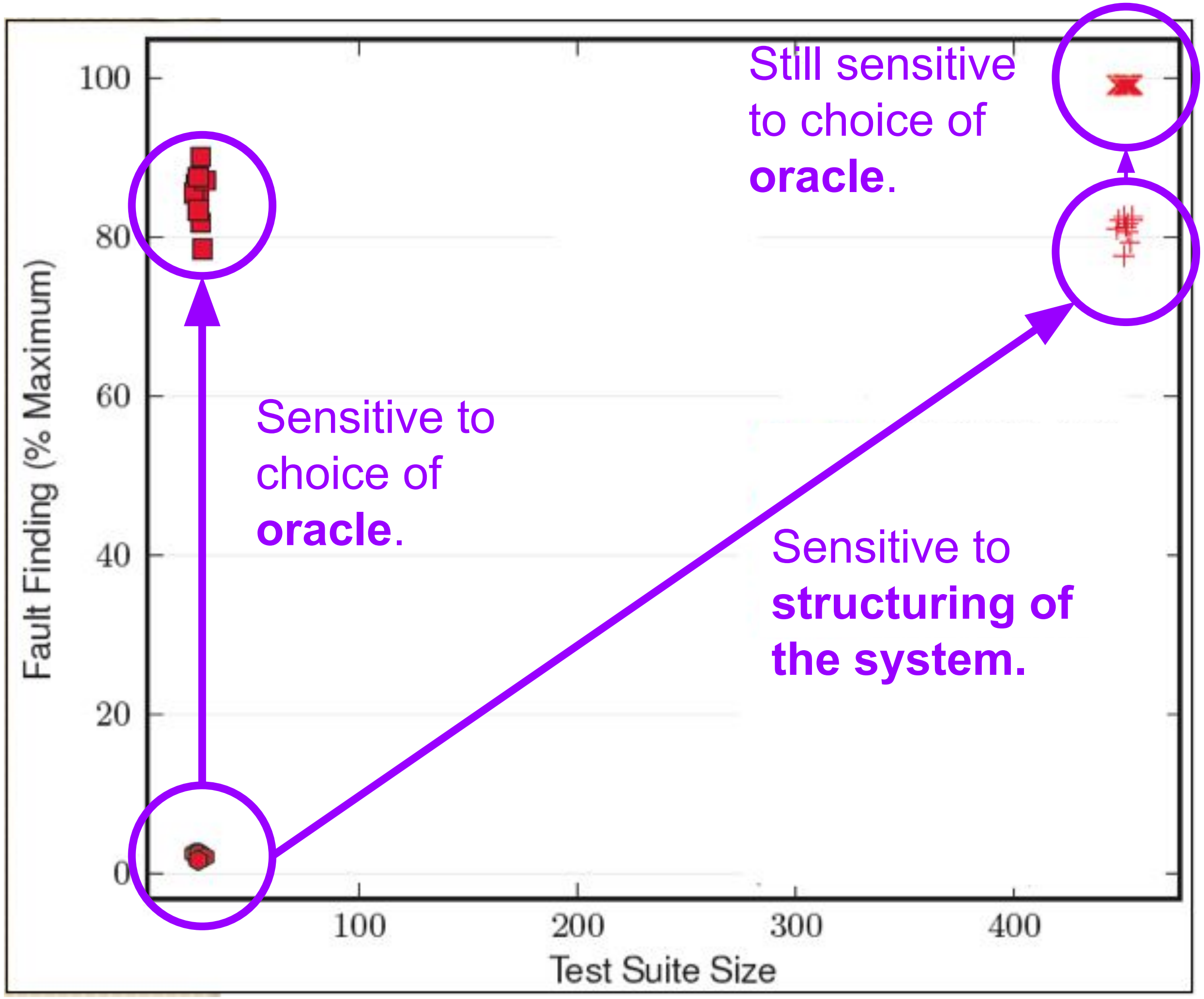
```
expr_1 = in_1 || in_2;
out_1 = expr_1 && in_3;

out_1 = (in_1 || in_2) && in_3;
```

- Both pieces of code do the same thing.
- How code is written impacts the number and type of tests needed.
- Simpler statements result in simpler tests.

Sensitivity to Oracle

- The oracle judges test correctness.
  - We need to choose what results we check when writing an oracle.
- Typically, we check certain output variables.
  - However, masking can prevent us from noticing a fault if we do not check the right variables.
  - We can't monitor and check all variables.
  - But, we can carefully choose a small number of bottleneck points and check those.
    - Some techniques for choosing these, but still more research to be done.

Coverage Effectiveness



Masking

Why do we care about faults in masked expressions?

- Effect of fault is only masked out for *this* test. It is still a fault. In another execution scenario, it might not be masked.
- We just haven't noticed it yet.
  - The fault isn't gone, we just have bad tests.
- One solution - ensure that there is a path from assignment to output where we will **notice the fault**.

One Solution - Observability

- Measure how well internal program state can be inferred from the output.
- The execution of an expression can be observed if we can modify its value and observe a change in the program output.
- Adds path constraints to existing coverage obligations requiring a path from expression to output free of masking.

Observable MC/DC

MC/DC + **observability** = Observable MC/DC

- MC/DC requires that conditions impact the outcome of a decision.
- OMC/DC requires that conditions impact the outcome of the **program**.

Idea: Lift observability from decision level to program level.

Tagging Semantics

Assign each condition a **tag set**:

`(ID, Boolean Outcome)`

Evaluation determines tag propagation:

`exp1=c1 && c2;` [~~(c1,true)~~, (c2,false)]

`exp2=c3 || c4;` [(c3,true), ~~(c4,false)~~]

`out=if (c5) then` [(c5,true),<exp1>,<exp2>]

[(c5,true),(c2,false),

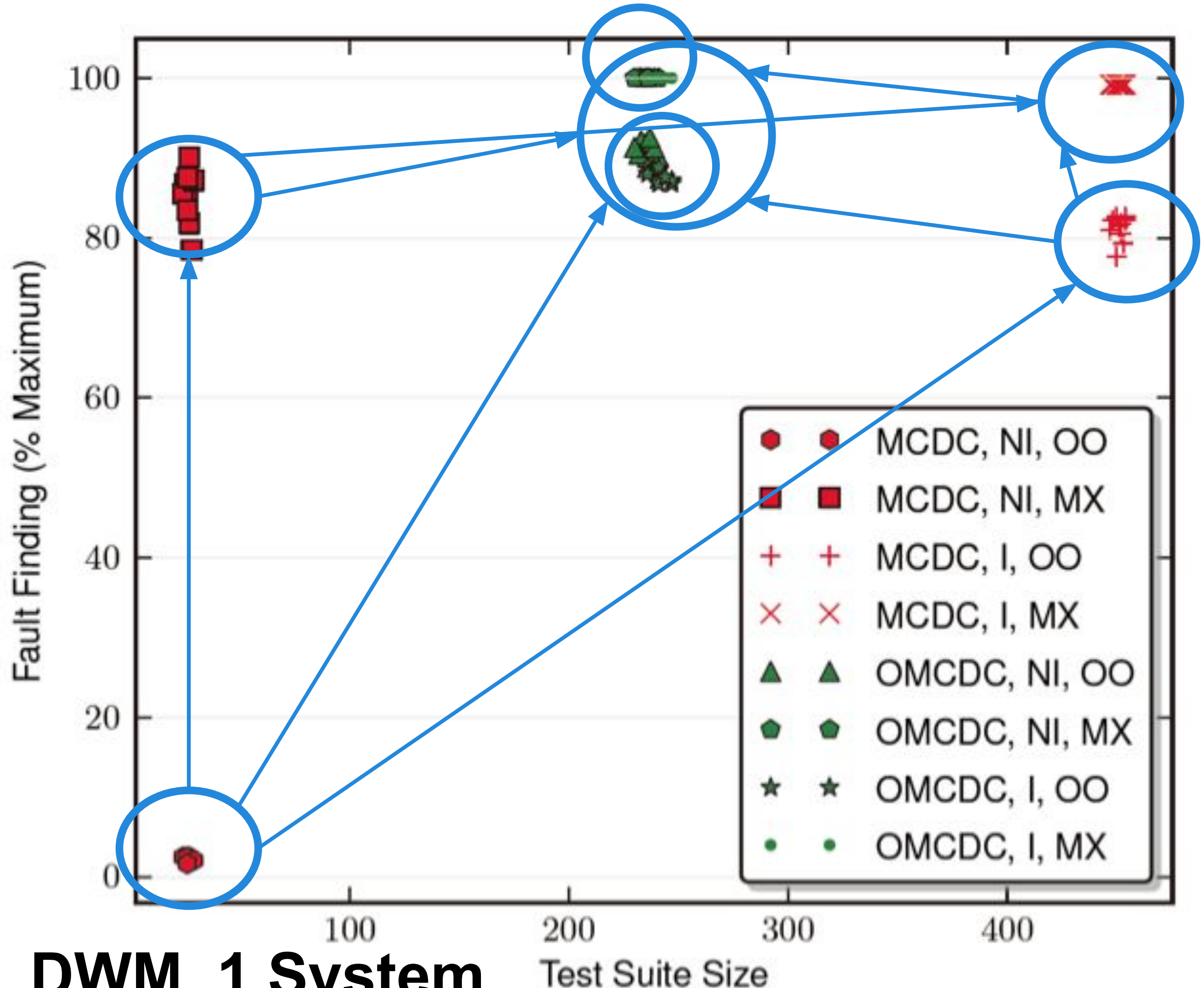
`exp1 else exp2;` ~~<exp2>~~]

Benefits of Observability

OMC/DC should improve test effectiveness by accounting for **program structure** and **oracle composition**:

- We select what points the oracle monitors, OMC/DC requires propagation path to those points.
- No sensitivity to structure because impact must be propagated at monitoring points.
  - i.e., we place conditions on the path taken.

Evaluation - Results



DWM_1 System

Still Not a Solved Problem

- OMC/DC often prescribes a large number of infeasible obligations.
- Tests can be difficult to derive.
- Often results in better fault-finding, but not 100% fault-finding (especially in complex systems).
- New coverage metrics and structural coverage methods are being formulated.

We Have Learned

- Strategies to get the benefits of path coverage without the cost.
- Procedure coverage metrics.
- How coverage criteria relate in terms of cost and power.
- Weaknesses of structural testing.

Next Time

- Dependability and When to Stop Testing
  - Statistical testing and reliability measurement.
  - Reading: Sommerville, ch. 11
- Homework 4.
  - Due April 21
  - Questions?