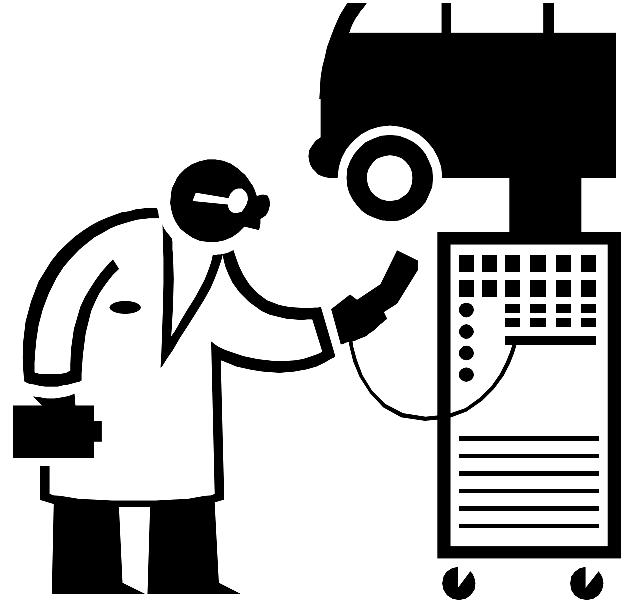
Software Testing

James M. Bieman
Computer Science Dept
Colorado State University

Outline

1. Introduction
2. Theory
3. Testing Strategies
    1. Black Box testing
    2. White Box Testing
4. Mutation testing

Reality



Even after verifying the design and code we will still need to test.

Some Terminology

- Reliability: probability that a program runs for a given time without software error.
- Validation: determination that a program is consistent with its requirements.
- Verification: determination that a program is correct with respect to its (formal) spec.
- Testing: examination of the behavior of a program by running it on selected sample data sets (inputs).

Terminology (2)

- Unit testing: testing a procedure, function, or class.
- Integration testing: testing connection between units and components.
- System testing: test entire system.
- Acceptance testing: testing to decide whether to purchase the software.

Terminology (3)

- Alpha testing: system testing by a user group within the developing organization.
- Beta testing: system testing by select customers.
- Regression testing: retesting after a software modification.

Dynamic Fault Classification

- Logic faults: omission or commission.
- Overload: data fields are too small.
- Timing: events are not synchronized.
- Performance: response is too slow.
- Environment: error caused by a change in the external environment.

Test Harnesses

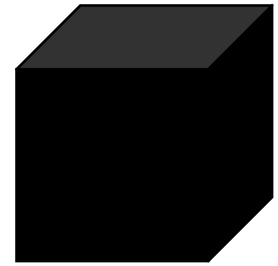
Allows us to test incomplete systems.

- Test drivers: test components.
- Stubs: test a system when some components it uses are not yet implemented.
  - Often a short, dummy program --- a method with an empty body.

Test Oracles

- Determine whether a test run completed with or without errors.
- Often a person, who monitors output.
  - Not a reliable method.
- Automatic oracles check output using another program.
  - Requires some kind of executable specification.

Testing Strategies: Black Box Testing



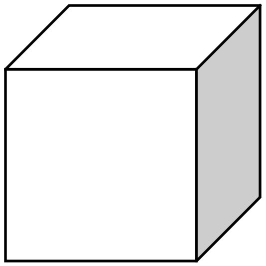
- Test data derived solely from specifications.

  Also called “functional testing”.

- Statistical testing.

  Used for reliability measurement and prediction.

Testing Strategies: White Box Testing



- Internal program structure used to derive test cases.
- Often test cases are selected to exercise particular program paths.
- Ideal white box test is to execute “all paths”.
- Also called “structural testing”.

Testing Theory: Why Is Testing So Difficult?

- Theory often tells us what we can’t do.
- Testing theory main result: *perfect testing* is impossible.

An Abstract View of Testing

- Let program *P* be a function with an input domain *D* (i.e., set of all integers).
- We seek test data *T*, which will include selected inputs of type *D*.
  - *T* is a subset of *D*.
  - *T* must be of finite size.

  Why?

We Need a Test Oracle

- Assume the best possible oracle --- the specification *S*, which is function with input domain *D.*
- On a single test input *i*, our program passes the test when

  $P(i) = S(i)$

  Or if we think of a spec as a Boolean function that compares the input to the output: $S(i, P(i))$

Requirement For Perfect Testing

[Howden 76]

1. If all of our tests pass, then the program is correct.

$$\forall x[x \in T \Rightarrow P(x) = S(x)]$$

$$\Rightarrow \forall y[y \in D \Rightarrow P(y) = S(y)]$$

- If for all tests t in test set T, P(t) = S(t), then we are sure that the program will work correctly for all elements in D.
- If any tests fail we look for a program fault.

Requirement For Perfect Testing

2. We can tell whether the program will eventually halt and give a result for any t in our test set T.

$\forall x[x \in T \Rightarrow$ "$\exists$ a computable procedure for determining if P halts on input x"]

But, Both Requirements Are Impossible to Satisfy.

- 1st requirement can be satisfied only if *T= D.*

  We test all elements of the input domain.
- 2nd requirement depends on a solution to the ***halting problem***, which has no solution.

We can demonstrate the problem with Requirement 1 [Howden 78].

Comments

- Key here is the need for <u>finite</u> sized test sets.
- Program domains are not usually finite.
- We seek to determine the behavior of programs with (effectively) infinite domains, using finite sets.
- We try to do the impossible.

Other Undecidable Testing Problems

- Is a control path feasible?
  - Can I find data to execute a program control path?
- Is some specified code reachable by any input data?

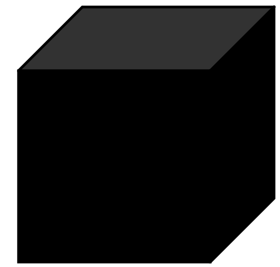
These questions cannot, in general, be answered.

Software Testing Limitations

- There is no perfect software testing.
- Testing can show defects, but can never show correctness.

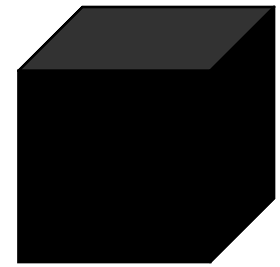
We may never find all of the program errors during testing.

Black Box Testing: Equivalence Partitioning



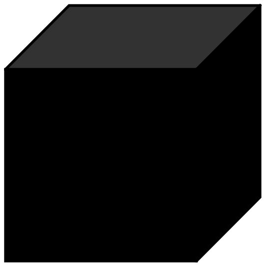
- Partition input domain of a program into a finite number of equivalence classes.
  - With respect to formal specifications.
  - Requires heuristics.
- Select at least one input from each class.

One Approach to Partitioning



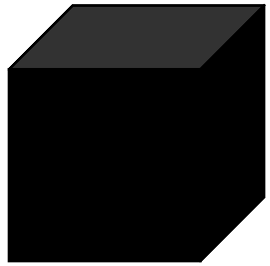
- Valid equivalence classes: valid inputs to the program.
- Invalid equivalence classes: invalid inputs to the program.

Black Box Testing: Boundary Value Analysis



- Select test cases of inputs just above and below boundaries.
  - Edges of equivalence classes.
- Select the ends of ranges of values.
  - Both just above and below legal values.

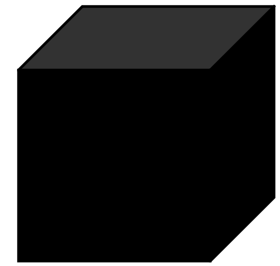
Black Box Testing: Cause-Effect Analysis



- Rely on pre-conditions and post-conditions and dream up cases.
- Identify impossible combinations.
- Construct decision table between input and output conditions.

  Each column corresponds to a test case.

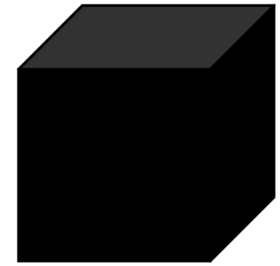
Black Box Testing: Error Guessing



- "Some people have a knack for 'smelling out' errors" [Meyers].
- Enumerate a list of possible errors or error prone situations.
- Write test cases based on the list.

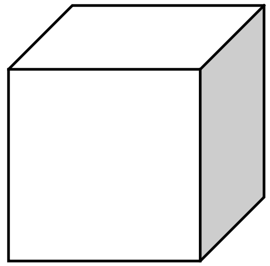
Depends upon having *fault models*, theories on the causes and effects of program faults.

Black Box Testing: Random Testing



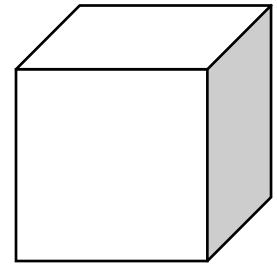
- Generate tests randomly.
- “Poorest methodology of all” [Meyers].
- Promoted by others.
- Statistical testing:
  - Test inputs from an operational profile.
  - Based on reliability theory.
  - Adds discipline to random testing.

Structural
White Box Testing



- Assume that a "path" through a program represents one relevant partition of the input domain.
- Choose one input from each path.
- Alas, a program with a loop has potentially an infinite number of paths.

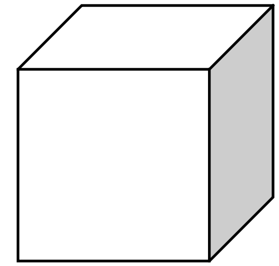
We Need to Find



1. Finite subsets *FS(P)* of the set of all paths through program *P*.
2. Choose Data to test each $p \in FS(P)$.

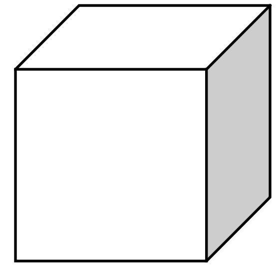
Does this guarantee that every time path *p* executes, it gives the correct output?

Choosing Data To Exercise Paths



- Symbolic execution.
  - Solve Boolean equations to find input data.
  - Some paths are infeasible.
  - Satisfiability problem: NP-complete for simple Boolean expressions. Undecidable for expressions containing ints or reals.
- Generate tests, then see how many paths are covered.

One Abstraction for Program Analysis



- Flowgraph directed graph:

$$G = (N, E, s, t)$$

N: set of nodes.

E: set of edges $E \subseteq N \times N$

s: start node $s \in N$

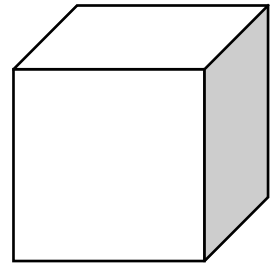
t: terminal node $t \in N$

Invariants: $N \neq \phi$, $E \neq \phi$, every $n \in N$ lies on a path from *s* to *t.*

Mapping Programs To Flowgraphs

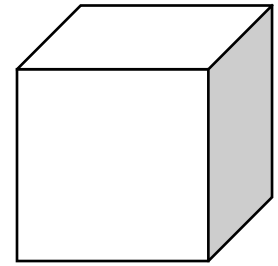
- Basic Block:
  – Maximal length sequential block of command-level code.
  – All code in a block is always executed sequentially.
- Map each basic block to one $n \in N$.
- *Each edge (nx,ny)* $\in$ *E* represents a possible transfer of control from the end of the *nx* block to the start of the *ny* block.

Mapping (2)



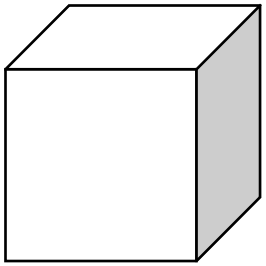
- *s, t* do not necessarily represent a basic block in code.
- Any path from *s* to *t* in a flowgraph *G* represents a potential execution path.
- *EP(G):* set of all paths from *s* to *t.*
- Some paths from *s* to *t* are <u>infeasible</u>.

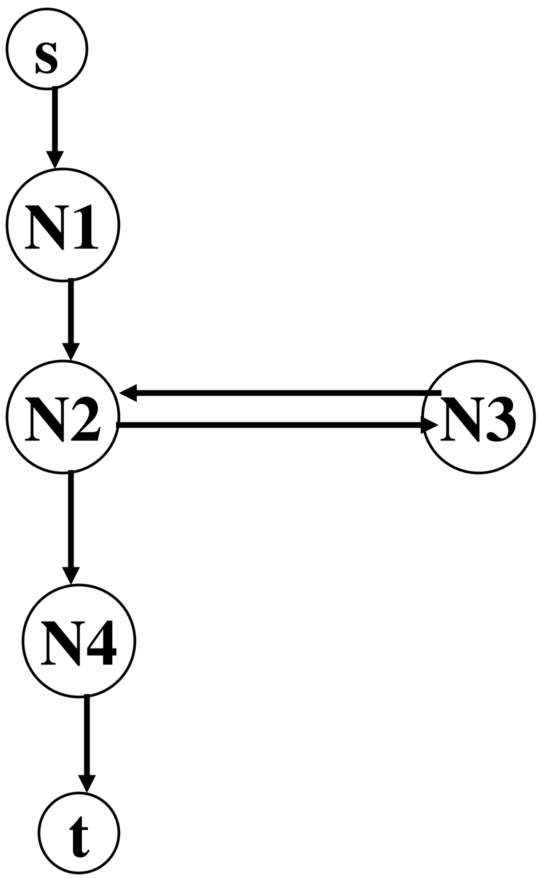
Consider Method sum



```
int sum(int a[]) {
    int i=0; int total = 0;  /* N1   */
    while (i < a.length)   /* N2  * /
      {
       total = total + a[i];  i++   /* N3 */
       }
    return total;    /* N4 */
    }
```

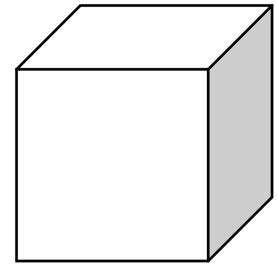
Flowgraph for sum





- **|*Paths(P)*| =** ∞
- **<s,N1,N2,N4,t> is infeasible.**
- **But there are an infinite number of feasible paths.**

Structural Testing Strategies



- To specify a structural testing strategy (flowgraph based) we specify criteria for finite sets of execution paths:

$$FS(G) \subseteq EP(G)$$

- | *FS(G)* / is finite.

Some Common Criteria

- Control flow based criteria:
  - Node or statement coverage.
  - Edge or branch coverage.

  $|FS(G)_{NodeCov}| \leq |FS(G)_{EdgeCov}|$
- Paths to test can be determined from the control flow graph alone.

Example: Edge Coverage is Stronger than Node Coverage

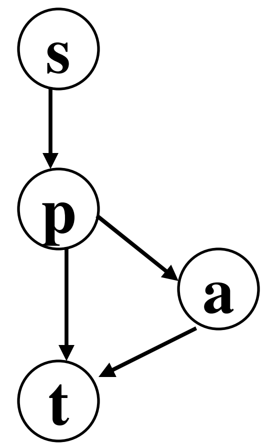
- Code: if (p) a;
- Node coverage:

  Pnodes = {<s,p,a,t>}

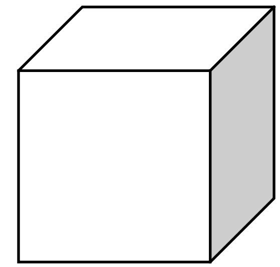
- Edge coverage:

  Pedges = {<s,p,a,t>, <s,p,t>}

Flowgraph



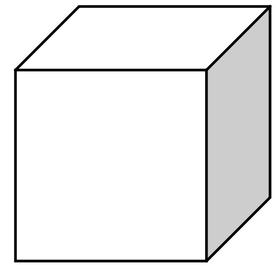
Other Control Flow Criteria



- All acyclic paths.
- Go through each “loop” 0 & 1 time.

  Acyclic paths + each “loop” (or rather cycle) exercised once.
- McCabe’s criteria: test each “linearly independent path”.

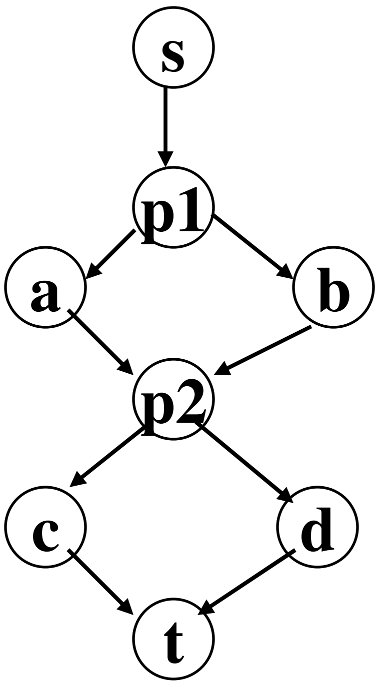
McCabe's Criteria Basis Path Testing



- Each path in the finite set contains one edge not on any other path & the set is an edge covering.
- Cyclomatic number of paths to test.

  Number of decisions + 1 paths.
- Might not include all acyclic paths.

McCabe's Criteria Misses Acyclic Paths



BPaths = {<*s, p1, a, p2, c, t*>,
<*s, p1, b, p2, c, t*>,
<*s, p1, a, p2, d, t*>}
meets McCabe's criteria.

All acyclic paths =
BPaths ∪ {< *s, p1, b, p2, d, t*>}

McCabe's Anomoly

- Number of acyclic paths in 2 or more adjacent sequential flowgraph components is computed by multiplying the number of paths in each component.

  |AcyclicP(F)| = |AcyclicP(I)| x |AcyclicP(II)|

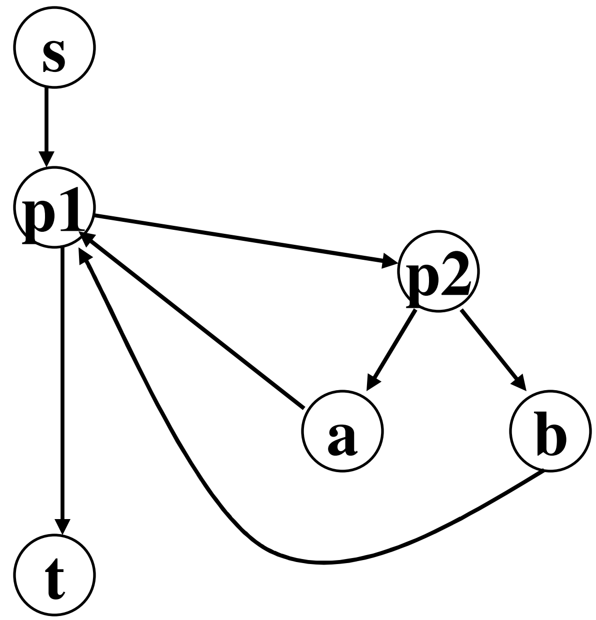
- Number of basis paths is a linear function on the number of decisions:

  |Bpaths(F)| = # decisions + 1

All Acyclic Paths + Once Through Each Cycle

- if-then-else in a while loop.
- FS(G) = {*<s,p1,t >*,
  <s, p1, p2, a, p1, t>,
  <s, p1, p2, b, p1, t>}
- Alternative FS(G) =
  {*<s,p1,t >*,
  <s,p1,p2,a,p1,p2,b, p1,t>}

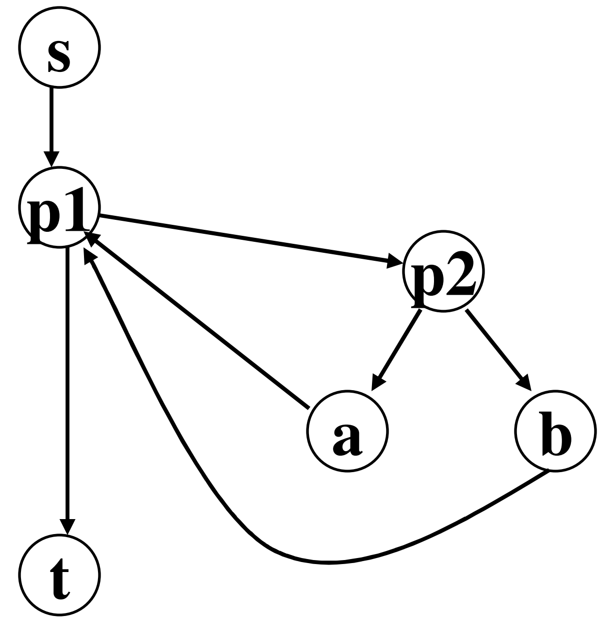
Flowgraph G:



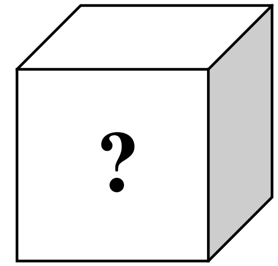
Note that

- $b \in succ(a) \wedge$
  $a \in succ(b) \wedge$
  $a \in succ(a) \wedge$
  $b \in succ(b)$
- Errors can lurk in these cases.
- Should we include all such possibilities?

Flowgraph G:



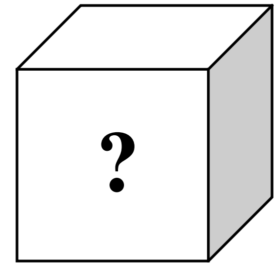
Mutation Testing



- Not a testing technique.
- Used to evaluate test data adequacy.
- Let *P* be a program, *S* is a specification, & *T = {t1,…,tn}* be a set of test input data.
- Assume $\forall t\left[t \in T \Rightarrow (P(t) = S(t))\right]$

  so *P* appears correct according to *T.*

Is *T* a Good Test Set?



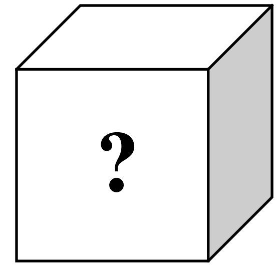
- We can create a set of mutants *M* of *P.*
- $m \in M \Rightarrow m$ *"differs from P".*

  We inject a fault in *P* to create a mutant.
- We run <u>each</u> mutant $m \in M$ on test data T.
- A mutant *m* may live or die:

$$Lives(m) \equiv \forall t\left[t \in T \Rightarrow S(t) = m(t)\right]$$

$$Died(m) \equiv \neg Lives(m)$$

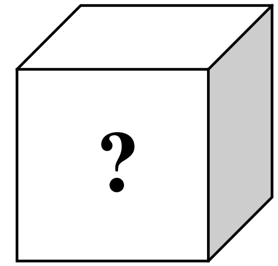
Mutants Live or Die



- The more mutants to live, the lower our confidence that *P* is actually correct.
- Why? Many mutants appear as good as *P.*

  A live mutant survived the tests as well as P.
- We assume that mutation is harmful.

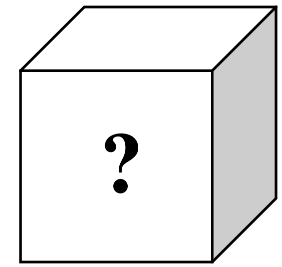
What if Mutants Survive?



- *T* can be expanded to kill the mutants.
- The key to mutation analysis:

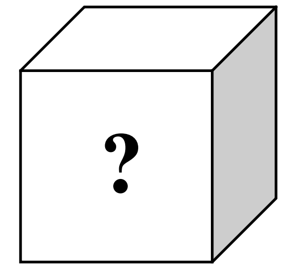
  Generating some *plausible mutants* ---
  mutants with some possibility of survival.

Mutate: program $\rightarrow$ set of programs



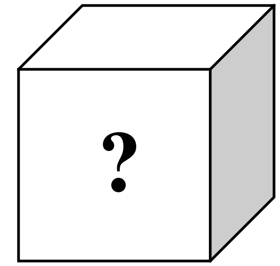
- Each mutant $m \in$ *Mutate(m)* must be syntactically correct or it is not a program.
- Syntax preserving transformations must be used.

Mutant Operators --- Mutate Functions



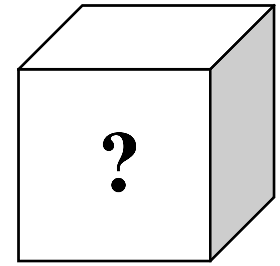
- Data objects --- constants, scalar variables, array references:
  - At each reference, generate mutants referencing all other accessible data objects.
  - Change values of a constant (a small change).
  - Change array referenced to another array.

Mutant Operators (2)



- Operators
  - Change operators to different ones of the same type (arithmetic, relational, logical).
  - Replace logical expression with true or false.
  - Delete subexpressions.
  - Delete unary operators.

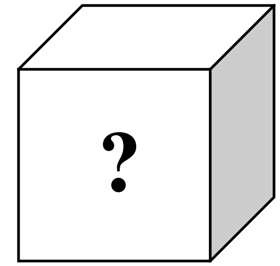
Mutant Operators (3)



- Statements
  - Delete statements.
  - Change to return statements.
  - Change order of assignments.
- Scramble labels.

Many mutants can be generated.

Mutations that Focus on Possible Errors



- Consider statement:

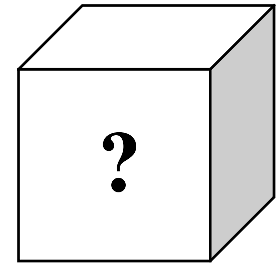
  if (i * j + 3 > k) blah;

- Mutated predicates:

  ((i +1) * j + 3 > k)
  ((i - 1) * j + 3 > k)
  (i * (j +1) + 3 > k)
  (i * j + 2 > k)
  (i * j + 4 > k)

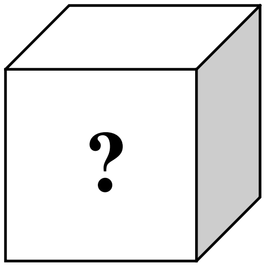
  Move predicate boundary slightly --- test correctness of predicate boundary.

More Mutations



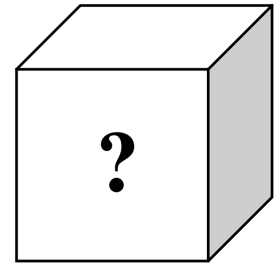
- Consider statement:
  a = b + c;
- Mutants:
  a = abs(b) + c;
  a = b + abs(c);
  a = abs(b + c);
- To kill these mutants we must include tests where:
  - b is negative
  - c is negative
  - b + c is negative.

Comments



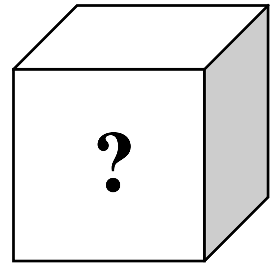
- Mutation testing is not infallible.
- MOTHRA: a current mutation system in experimental use at Purdue [DeMillo et al] used to assist in performing structural & functional testing.
- TDS for testing distributed CORBA systems [Ghosh].

Bebugging [Mills]



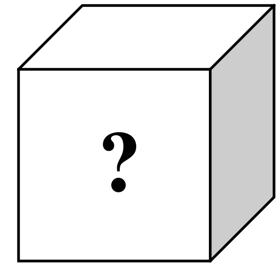
- Error seeding to generate error statistics.
- Analogy from ecology:
  - 1000 fish caught in lake, marked & released.
  - Later catch a new batch of 1000 fish.
  - We find 100 fish are marked.
  - We are justified in saying that the true number of fish in the lake is between 8,500 and 12,000.

Error Seeding



- Marked fish ← inserted errors.
- Insert errors and use a similar analysis to discover the actual number of errors.

Problems



- Need a large number of errors in the project for the statistics to work.

  Need on the order of 10,000 errors.

- Errors are not uniformly distributed like fish are.
- Tests may discover artificial errors easier than natural ones.
- Seeded errors may differ from natural ones.

Competent Programmer Hypothesis

- Programmers are not adversaries.
- Most errors are:
  - Simple in form.
  - Well understood.
  - Classifiable.
- Hypothesis is necessary for any effective testing strategy.

Summary

- Testing is very difficult.
  - Theory: testing for correctness is impossible.
  - All methods fail to find all errors.
- Random testing for evaluating reliability.
- Black box testing: use the specification to define tests.
- White box testing: use the internal structure to define tests.
- Fault injection for test evaluation.