Fundamentals of Dynamic Testing

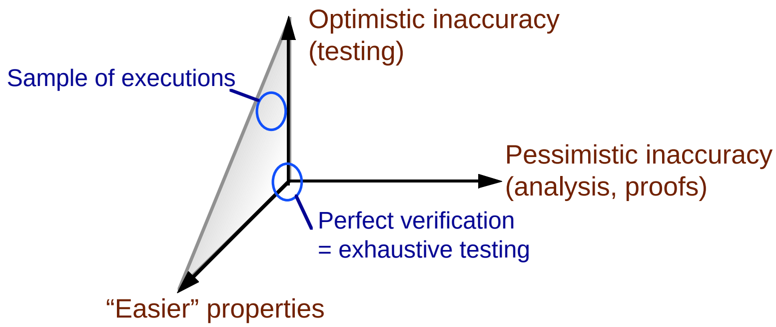
- *Three questions:*
  - How shall we tell if a test has succeeded or failed?

    [More next week on this, as well as process issues]
  - How shall we select test cases?
  - How do we know when we're done?

    [Most of today's lecture]

Historically approached in the opposite order

Inaccuracy in Dynamic Testing



- Since exhaustive testing is impossible, we must choose a sample of executions

Possible Goals of Testing

- Find faults
  - Glenford Myers, *The Art of Software Testing*
- Provide confidence
  - of reliability
  - of (probable) correctness
  - of detection (therefore absense) of particular faults

Testing for Reliability

- Reliability is statistical, and requires a statistically valid sampling scheme
- Programs are complex human artifacts with few useful statistical properties
- In some cases the environment (usage) of the program has useful statistical properties
  - Usage profiles can be obtained for relatively stable, pre-existing systems (telephones), or systems with thoroughly modeled environments (avionics)

Certifying Ultra-High reliability

- Problem: How can I show that system *X* has an expected failure rate of 10^{-9}/hour?
  - example: probability that software will ever bring down an Airbus A320
- Butler & Finelli estimate
  - for 10^{-9} per 10 hour mission
  - requires: 10^{10} hours testing with 1 computer
  - or: 10^{6} hours (114 years) testing with 10,000 computers

[ACM Sigsoft 91, Conf. on SW for Critical Systems]

Arbitrary ≠ Random

- A common error in attempting to obtain statistical confidence measures
  - Arbitrary distributions may be modeled by adversary functions, not by uniform distributions
- Example:
  - If failures were distributed randomly through the execution space of a database program, it would fail at a uniform rate over time.
  - In reality, it may never fail until a critical table overflows, and then always fail thereafter.

Glimmers of Hope

for Measuring High Reliability

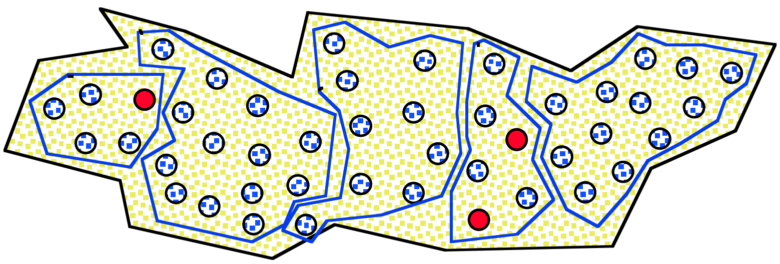
- Random distribution of faults or failures would enable statistical reasoning and classic redundancy techniques
  - A whole more reliable than its parts
- Randomization approaches
  - Blum: Self-checking programs
  - Lipton: Redundant computations
  - Podgurski: Kolmogorov complexity
- Grail or illusion?
  - Difficult to generalize beyond simple functions

Process-Based Reliability Testing

- Rather than relying only on properties of the program, we may use historical characteristics of the development process
- Reliability growth models (Musa, Littlewood, et al) project reliability based on experience with the current system and previous similar systems

Partition Testing

- Basic idea: Divide program input space into (quasi-) equivalence classes
  - Underlying idea of specification-based, structural, and fault-based testing



"Adequate" partition testing

- Ideally: adequate testing ensures some property (proof by cases)
  - Origins in Goodenough & Gerhart, Weyuker and Ostrand
  - In reality: as impractical as other program proofs

Systematic Partition Testing

- Systematic (non-random) testing is aimed at program improvement, not measurement
  - Obtaining valid samples and maximizing fault detection require different approaches; it is unlikely that one kind of testing will be satisfactory for both
- Practical “adequacy” criteria are negative: indications of important omissions

Specification-Based Partition Testing

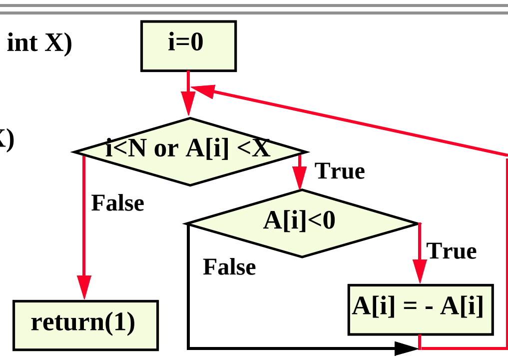
- Divide the program input space according to identifiable cases in the specification
  - May include boundary cases
  - May include combinations of features or values
    - If all combinations are considered, the space is usually too large
- Systematically “cover” the categories
  - May be driven by scripting tools or input generators

Structural Coverage Testing

- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely 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

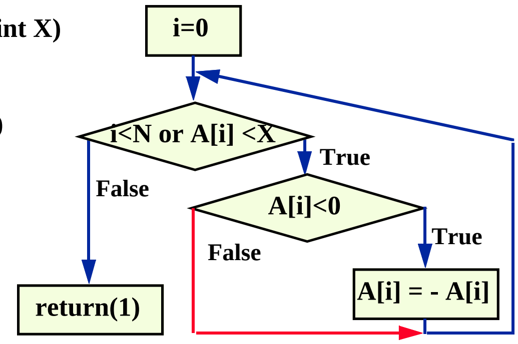
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
       if (A[i]<0)
           A[i] = - A[i];
   }
   return(1);
}
```



One test datum (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

Branch Coverage

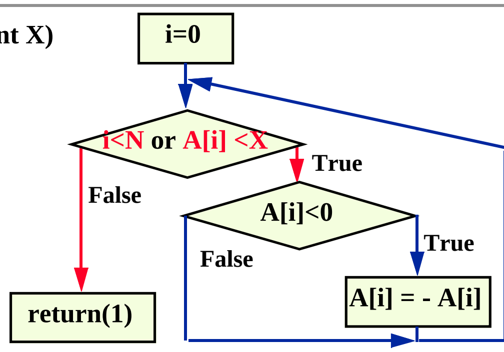
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
      if (A[i]<0)
         A[i] = - A[i];
   }
   return(1);
}
```



We must add a test datum (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

Condition Coverage

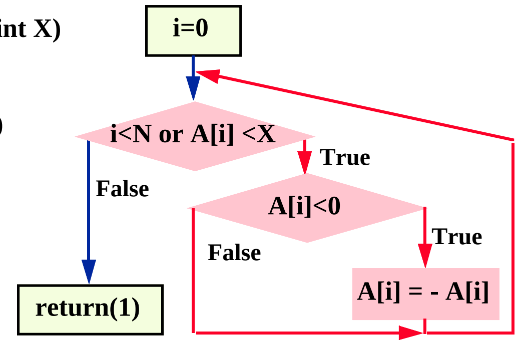
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
      if (A[i]<0)
         A[i] = - A[i];
   }
   return(1);
}
```



Both conditions (i<N), (A[i]<X) must be false and true for different tests. In this case, we must add tests that cause the while loop to exit for a value greater than X.
Faults that arise after several iterations of the loop would not be revealed.

Path Coverage

```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
      if (A[i]<0)
         A[i] = - A[i];
   }
   return(1);
}
```

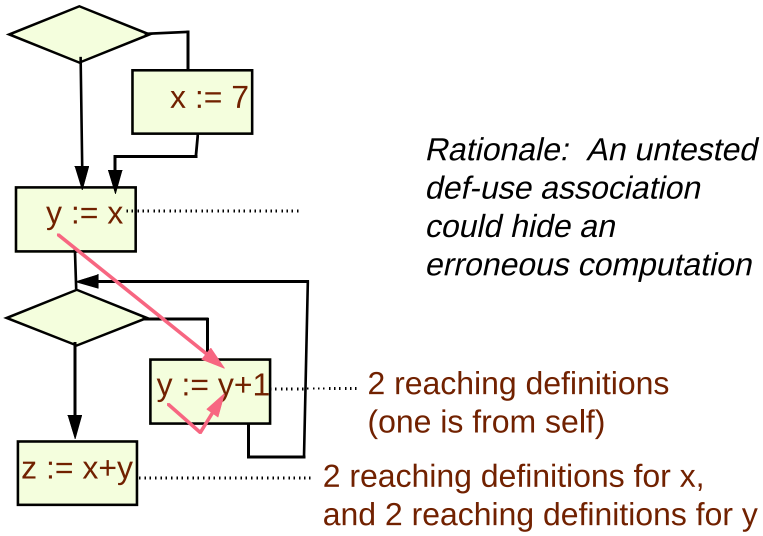


The loop must be iterated given number of times.
PROBLEM: uncontrolled growth of test sets. We need to select a significant subset of test cases.

Data flow testing: an example of partition testing

- Identify def-use pairs (reaching definitions) in program source code
- Coverage criterion: Each def-use pair must be executed at least once
- Rationale: Untested def-use pairs hide bad computations
  - Typical of coverage criteria: Justified as a lower bound on sufficient testing, not an upper bound

Data flow coverage criteria (ex.)



Rationale: An untested def-use association could hide an erroneous computation

Fault-based testing

- Given a fault model
  - hypothesized set of deviations from correct program
  - typically, simple syntactic *mutations*; relies on coupling of simple faults with complex faults
- Coverage criterion: Test set should be adequate to reveal (all, or x%) faults generated by the model
  - similar to hardware test coverage

Structural Coverage in Practice

- Statement and sometimes edge coverage is used in practice
  - Simple lower bounds on adequate testing; may even be harmful if inappropriately used for test selection
- Additional control flow heuristics sometimes used
  - Loops (never, once, many), combinations of conditions

The "subsumes" hierarchy

- Intuition: "stronger" criteria for better testing
  - Adequacy criterion A <u>subsumes</u> criterion B iff, for every program P, a test set that satisfies A for P, necessarily satisfies B for P
- Problems:
  - Unclear link to dependability
    - Although Hamlet & Taylor [86] has been widely misinterpreted
  - Does not consider cost
    - Partly addressed by Frankl et al (fault detection efficiency), but still does not address Ntafos' arguments for cheaper random testing

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
- Unsatisfactory approaches:
  - Manual justification for omitting each impossible test case (esp. for more demanding criteria)
  - Adequacy “scores” based on coverage
    - example: 95% statement coverage, 80% def-use coverage

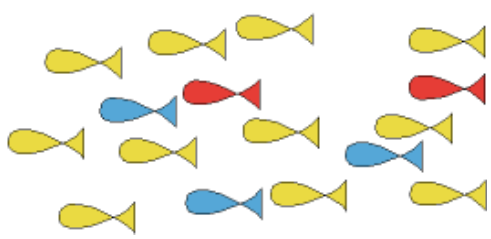
Challenges in Structural Coverage

- Interprocedural and gross-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 Budget Coverage Criterion

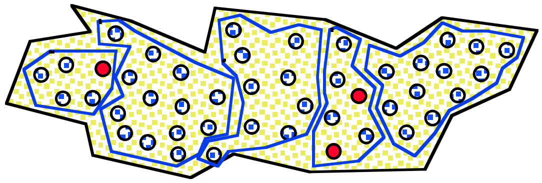
- Industry's answer to "when is testing done"
  - When the money is used up
  - When the deadline is reached
- *This is sometimes a rational approach!*
  - *Implication 1:* Adequacy criteria answer the wrong question. Selection is more important.
  - *Implication 2:* Practical comparision of approaches must consider the cost of test case selection

Selection vs. Adequacy: Mutation Testing Example



- Red fish = real program faults (unknown population)
- Blue fish = seeded faults (e.g., mutations) or representative behaviors (known population)
- Adequacy: count blue fish caught, estimate red fish
- Misuse for selection: use special bait to catch blue fish

Partition Testing: Summary



- Non-random selection for fault detection
  - as versus statistical reasoning about reliability
- Specification-based partitioning is the primary systematic technique
  - at unit, subsystem, and system levels
- Structural criteria indicate “holes” in the tests
  - but satisfying a structural criterion guarantees nothing