Fundamentals of Dynamic Testing

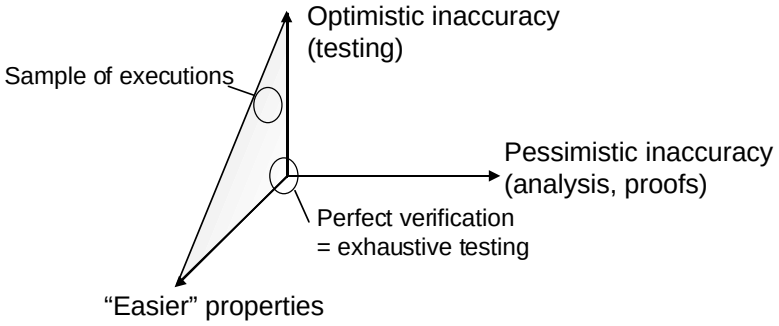
- *Three questions:*
  - How shall we tell if a test has succeeded or failed?
  - How shall we select test cases?
  - How do we know when we're done?

Historically approached in the opposite order

What is "thorough" testing?

- Can we test exhaustively?
  - Why or why not?
  - And if not, why bother?

Accuracy / Feasibility Tradeoffs



- 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

Granularity levels

- **Acceptance testing**: the software behavior is compared with end user requirements
- **System testing**: the software behavior is compared with the requirements specifications
- **Integration testing**: checking the behavior of module cooperation.
- **Unit testing**: checking the behavior of single modules
- **Regression testing**: to check the behavior of new releases

Testing activities before coding

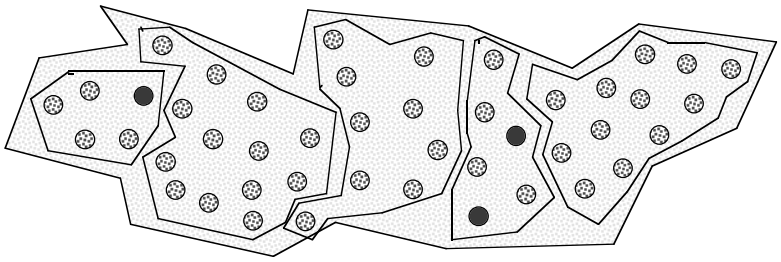
- **Planning**
  - acceptance test planning (requirements elicitation)
  - system test planning (requirements specifications)
  - Integration & unit test planning (architectural design)
- **Generation**
  - create functional system tests (requirement specifications)
  - generate test oracles (detailed design)
  - generate black box unit tests (detailed design)

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



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

Exercise

An event queue in a simulation system is a priority queue where events are extracted according to their time-stamps (earliest time-stamps first, last-in among events with the same time-stamp)

- Devise a set of functional test cases for an event queue

Structural Coverage Testing

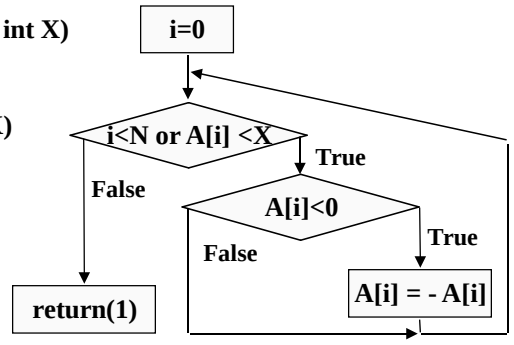
- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
  - Path coverage
  - Data flow (syntactic dependency) coverage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

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

Statement Coverage

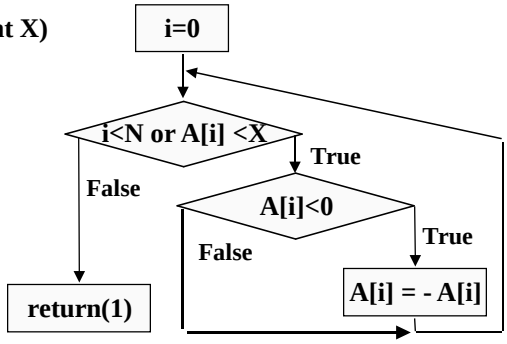
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
       if (A[i]<0)
           A[i] = - A[i];
   }
   return(1);
}
```



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

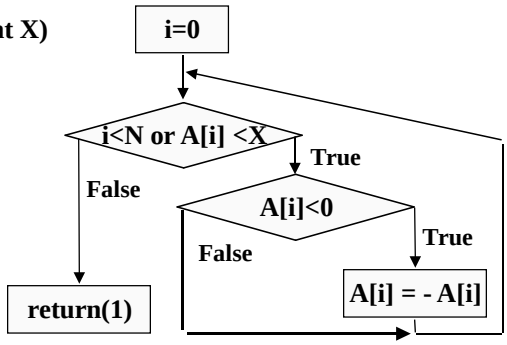
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
       if (A[i]<0)
           A[i] = - A[i];
   }
   return(1);
}
```



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

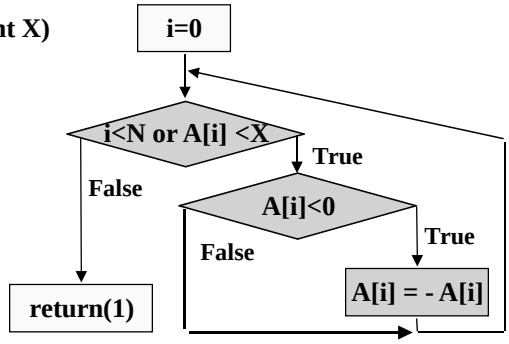
```
int select(int A[], int N, int X)
{
   int i=0;
   while (i<N or A[i] <X)
   {
       if (A[i]<0)
           A[i] = - A[i];
   }
   return(1);
}
```



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);
}
```

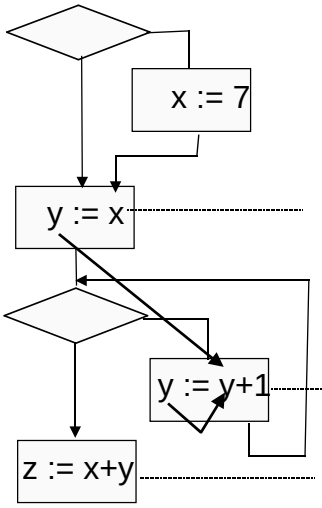


The loop must be iterated given number of times.
PROBLEM: uncontrolled growth of test sets. We need to select a significant subset of test cases.

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

2 reaching definitions (one is from self)

2 reaching definitions for x, and 2 reaching definitions for y

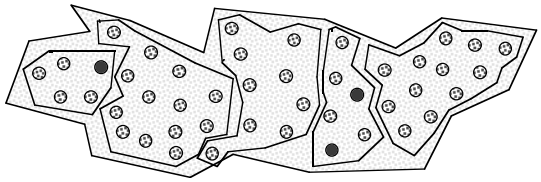
Fault-based testing

- 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

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