Goals of testing

- Reveal faults
    - Correctness
    - Reliability
    - Usability
    - Robustness
    - Performance

Top-down/Bottom-up

- Bottom-up
    - Lowest level modules tested first
        - Don't depend on any other modules
        - Driver
            - Auxiliary code that calls the module
- Top-down
    - Executive module tested first
        - Stub
            - Auxiliary code that simulates the results of a routine

Facts About Testing

- Question "does program P obey specification S" is undecidable!
- Every testing technique embodies some compromise between accuracy and computational cost
- Facts
    - Inaccuracy is not a limitation of the technique
    - It is theoretically impossible to devise a completely accurate technique
    - Every practical technique must sacrifice accuracy in some way

Cost/benefit

- Testing takes more than 50% of the total cost of software development
    - More for critical software
- Software quality will become the dominant success criterion

Types of Verification

- Execution-based Verification
- Non-execution based Verification

- Discussion

Execution-based Verification

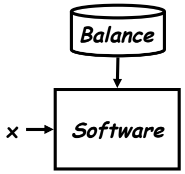
- Generating and *executing* test cases on the software
- Types of testing
  - Testing to specifications
    - Black-box testing
  - Testing to code
    - Glass-box (white-box) testing

  - Remember: difference is in generating test cases only! Verification of correctness is usually done via specifications in both cases

---

Black-box Testing

- Discussion: MAC/ATM machine example
  - Specs
    - Cannot withdraw more than $300
    - Cannot withdraw more than your account balance



White-box Testing

- Example

```
     x: 1..1000;
1    INPUT-FROM-USER(x);
     If (x <= 300) {
2              INPUT-FROM-FILE(BALANCE);
               If (x <= BALANCE)
3                        GiveMoney x;
4              else Print "You don't have $x in your account!!"}
     else
5              Print "You cannot withdraw more than $300";
6    Eject Card;
```

Discussion

- Which is superior?
- Neither can be done exhaustively
  - Too many test cases
- Each technique has its strengths – use both
  - Generally, first use black-box
  - Then white-box for missed code
- Accept that all faults cannot be detected
  - When to stop?

Determining Adequacy

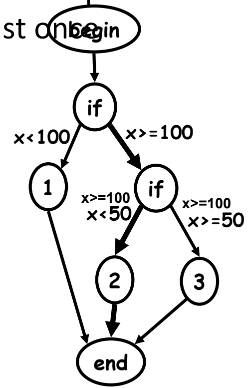
- Statement coverage
  - Statements
- Branch coverage
  - Both IF and ELSE
- Path coverage
- All-def-use-path coverage

- Philosophy: what does it all mean?
  - Does coverage guarantee absence of faults?
- Can we always get 100% coverage?

Surprise Quiz

- Determine test cases so that each print statement is executed at least once

```
input(x);
if (x < 100)
        print "Line 1";
else {
        if (x < 50) print "Line 2"
        else print "Line 3";
}
```



Sampling the State Space

- If (i == j)
  - Do something wrong
- Else
  - Do the right thing
- Endif

- Uniform sampling of the input space
- Test adequacy criteria
  - Designed to insure behaviors chosen are appropriately distributed to increase the likelihood of revealing errors

Non-execution Based

- Key idea
  - Review by a team of experts: syntax checker?
- Code readings
- Walkthroughs
  - Manual simulation by team leader
- Inspections
  - Developer narrates the reading
- Formal verification of correctness
  - Very expensive
  - Justified in critical applications
- Semi-formal: some assertions

Non-execution Based

- JPL
  - On the average, 2 hour inspection
  - 4 major and 14 minor faults
  - Saved $25,000 per inspection
- Rate of faults
  - Decreases exponentially by phase
- Cleanroom approach
  - Incremental development, formal specs and design, readings, inspections

Boundary-value Analysis

- Partition the program domain into input classes
- Choose test data that lies both inside each input class and at the boundary of each class
- Select input that causes output at each class boundary and within each class
- Also known as stress testing

Testing Approaches

- Top-down
- Bottom-up
- Big bang

- Unit testing
- Integration testing
- Stubs
- System testing

Glossary

- Fault
  - An incorrect step, process, or data definition in a computer program
- Error (ISO)
  - A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition
- Failure (IEEE)
  - The inability of a system or component to perform its required functions within specified performance requirements

Structural Testing

- Coverage-based testing
  - Test cases to satisfy statement coverage
  - Or branch coverage, etc
- Complexity-based testing
  - Cyclomatic complexity
    - Graph representation
    - Find the basis set
    - # Of braches + 1

Mutation Testing

- Errors are introduced in the program to produce “mutants”
- Run test suite on all mutants and the original program

Test Case Generation

- Test input to the software
- Some researchers/authors also define the test case to contain the expected output for the test input

Category-partition Method

- Key idea
  - Method for creating functional test suites
  - Role of test engineer
    - Analyze the system specification
    - Write a series of formal test specifications
  - Automatic generator
    - Produces test descriptions

AI Planning Method

- Key idea
  - Input to command-driven software is a sequence of commands
  - The sequence is like a plan
- Scenario to test
  - Initial state
  - Goal state

Example

- VCR command-line software
- Commands
  - Rewind
    - If at the end of tape
  - Play
    - If fully rewound
  - Eject
    - If at the end of tape
  - Load
    - If VCR has no tape

Preconditions & Effects

- Rewind
  - Precondition: if at end of tape
  - Effects: at beginning of tape
- Play
  - Precondition: if at beginning of tape
  - Effects: at end of tape
- Eject
  - Precondition: if at end of tape
  - Effects: VCR has no tape
- Load
  - Precondition: if VCR has no tape
  - Effects: VCR has tape

Preconditions & Effects

- Rewind
  - Precondition: end_of_tape
  - Effects: ¬end_of_tape
- Play
  - Precondition: ¬end_of_tape
  - Effects: end_of_tape
- Eject
  - Precondition: end_of_tape
  - Effects: ¬has_tape
- Load
  - Precondition: ¬has_tape
  - Effects: has_tape

Initial and Goal States

- Initial state
  - end_of_tape
- Goal state
  - ¬end_of_tape
- Plan?
  - Rewind

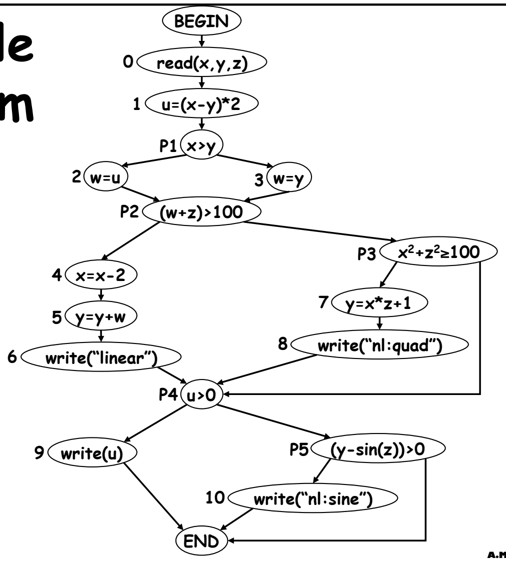
Initial and Goal States

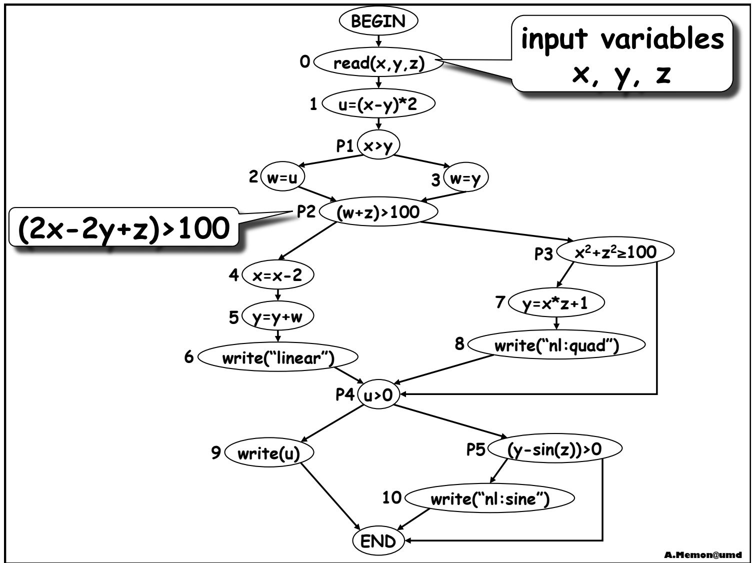
- Initial state
  - ¬end_of_tape & has_tape
- Goal state
  - ¬has_tape
- Plan?
  - Play
  - Eject

Iterative Relaxation

- Key idea
  - Path-oriented testing
  - Problem: generation of test data that causes a program to follow a given path
- Technique
  - Choose arbitrary input
  - Iteratively refine it until all the branch predicates on the given path evaluate to the desired outcome

Example Program





input variables x, y, z

(2x-2y+z)>100

Test Coverage & Adequacy

- How much testing is enough?
- When to stop testing
- Test data selection criteria
- Test data adequacy criteria
  - Stopping rule
  - Degree of adequacy
- Test coverage criteria
- Objective measurement of test quality

Preliminaries

- Test data selection
  - What test cases
- Test data adequacy criteria
  - When to stop testing
- Examples
  - Statement coverage
  - Branch coverage
  - Def-use coverage
  - Path coverage

Uses of Test Adequacy

- Objectives of testing
- In terms that can be measured
  - For example branch coverage
- Two levels of testing
  - First as a stopping rule
  - Then as a guideline for additional test cases

Categories of Criteria

- Specification based
  - All-combination criterion
    - Choices
  - Each-choice-used criterion
- Program based
  - Statement
  - Branch
- Note that in both the above types, the correctness of the output must be checked against the specifications

Classification according to underlying testing approach

- Structural testing
  - Coverage of a particular set of elements in the structure of the program
- Fault-based testing
  - Some measurement of the fault detecting ability of test sets
- Error-based testing
  - Check on some error-prone points

Structural Testing

- Program-based structural testing
  - Control-flow based adequacy criteria
    - Statement coverage
    - Branch coverage
    - Path coverage
      - Length-i path coverage
    - Multiple condition coverage
      - All possible combinations of truth values of predicates
  - Data-flow based adequacy criteria

Structural Testing

- Data-flow based adequacy criteria
  - All definitions criterion
    - Each definition to some *reachable* use
  - All uses criterion
    - Definition to each reachable use
  - All def-use criterion
    - Each definition to each reachable use

Fault-based Adequacy

- Error seeding
  - Introducing artificial faults to estimate the actual number of faults
- Program mutation testing
  - Distinguishing between original and *mutants*
    - Competent programmer assumption
      - Mutants are close to the program
    - Coupling effect assumption
      - Simple and complex errors are coupled

Test Oracles

- Discussion
  - Automation of oracle necessary
  - Expected behavior given
  - Necessary parts of an oracle

Test Oracle

- A test oracle determines whether a system behaves correctly for test execution

- Webster dictionary - oracle
  - A person giving wise or authoritative decisions or opinions
  - An authoritative or wise expression or answer

Purpose of Test Oracle

- Sequential systems
  - Check functionality
- Reactive (event-driven) systems
  - Check functionality
  - Timing
  - Safety

Reactive Systems

- Complete specification requires use of multiple computational paradigms
- Oracles must judge all behavioral aspects in comparison with all system specifications and requirements
- Hence oracles may be developed directly from formal specifications

Parts of an Oracle

- Oracle information
  - Specifies what constitutes correct behavior
    - Examples: input/output pairs, embedded assertions
- Oracle procedure
  - Verifies the test execution results with respect to the oracle information
    - Examples: equality
- Test monitor
  - Captures the execution information from the run-time environment
    - Examples
      - Simple systems: directly from output
      - Reactive systems: events, timing information, stimuli, and responses

Regression Testing

- Developed first version of software
- Adequately tested the first version
- Modified the software; Version 2 now needs to be tested
- How to test version 2?
- Approaches
  - Retest entire software from scratch
  - Only test the changed parts, ignoring unchanged parts since they have already been tested
  - Could modifications have adversely affected unchanged parts of the software?

Regression Testing

- “Software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected unchanged portions of the program.”

Regression Testing Vs. Development Testing

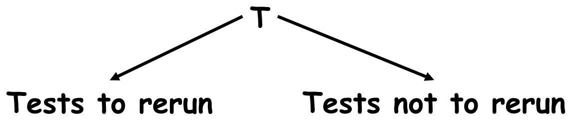
- During regression testing, an established test set may be available for reuse

- Approaches
  - Retest all
  - Selective retest (selective regression testing) ← main focus of research

Formal Definition

- Given a program P,
- Its modified version P’, and
- A test set T
  - Used previously to test P
- Find a way, making use of T to gain sufficient confidence in the correctness of P’

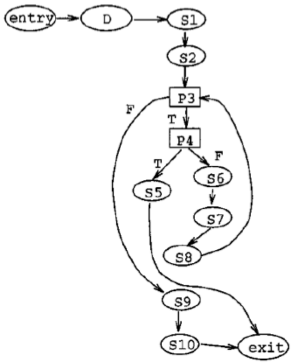
Selective Retesting



T

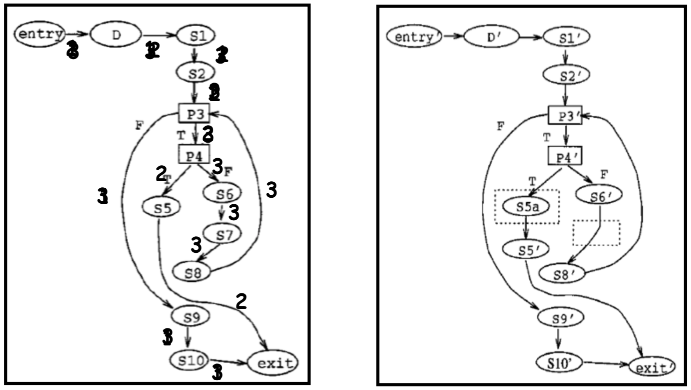
Tests to rerun **Tests not to rerun**

- Tests to rerun
  - Select those tests that will produce different output when run on P’
    - Modification-revealing test cases
    - It is impossible to always find the set of modification-revealing test cases – (we cannot predict when P’ will halt for a test)
  - Select modification-traversing test cases
    - If it executes a new or modified statement in P’ or misses a statement in P’ that it executed in P



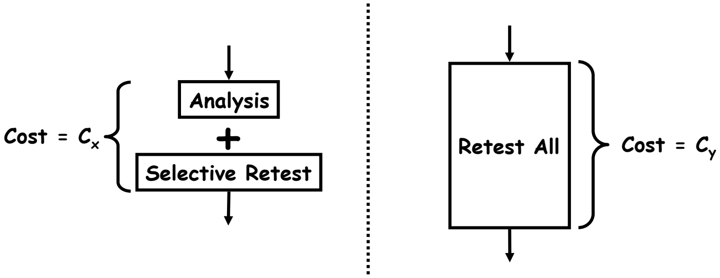
```
        Procedure avg

S1.  count = 0
S2.  fread(fileptr,n)
P3. while (not EOF) do
P4.    if (n<0)
S5.       return(error)
       else
S6.       numarray[count] = n
S7.       count++
       endif
S8.    fread(fileptr,n)
    endwhile
S9. avg = calcavg(numarray,count)
S10. return(avg)
```

Fig. 1. Procedure avg and its CFG.



T' = {t2, t3}

Cost of Regression Testing



Cost = C_x

Analysis

+

Selective Retest

Retest All

Cost = C_y

We want $C_x < C_y$

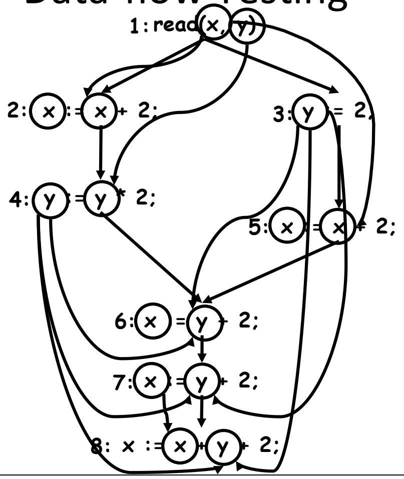
Key is the test selection algorithm/technique

We want to maintain the same "quality of testing"

Factors to Consider

- Testing costs
- Fault-detection ability
- Test suite size vs. Fault-detection ability
- Specific situations where one technique is superior to another

Data-flow Testing



```
1: read(x, y)
2: x := x + 2;
3: y := 2;
4: y := y + 2;
5: x := x + 2;
6: x := y + 2;
7: x := y + 2;
8: x := x + y + 2;
```

A.Memon@umd