Testing Research

Lecture 7a

Overview

- Testing research has a long history
  - At least to the 1960's
- Much work is focused on metrics
  - Assigning numbers to programs
  - Assigning numbers to test suites
  - Heavily influenced by industry practice
- More recent work focuses on deeper analysis
  - Semantic analysis, in the sense we understand it

Random Testing

- About $\frac{1}{4}$ of Unix utilities crash when fed random input strings
  - Up to 100,000 characters

- What does this say about testing?

- What does this say about Unix?

What it Says About Testing

- Randomization is a highly effective technique
  - And we use very little of it in software

- "A random walk through the state space"

- To say anything rigorous, must be able to characterize the distribution of inputs
  - Easy for string utilities (ala the paper)
  - Harder for systems with more arcane input
    - E.g., parsers for context-free grammars

What it Says About Unix

- What sort of bugs did they find?
  - Buffer overruns
  - Format string errors
  - Wild pointers/array out of bounds
  - Signed/unsigned characters
  - Failure to handle return codes
  - Race conditions
- Nearly all of these are problems with C!
  - Would disappear in Java
  - Exceptions are races & return codes

My Favorite Bug

csh !0%8f

- ! is the history lookup operator
  - No command beginning with 0%8f
- csh passes an error "0%8f: Not found" to an error printing routine
- Which prints it with printf()

Efficient Regression Testing

- Problem: Regression testing is expensive
- Observation: Changes don't affect every test
  - And tests that couldn't change need not be run
- Idea: Use a conservative static analysis to prune test suite

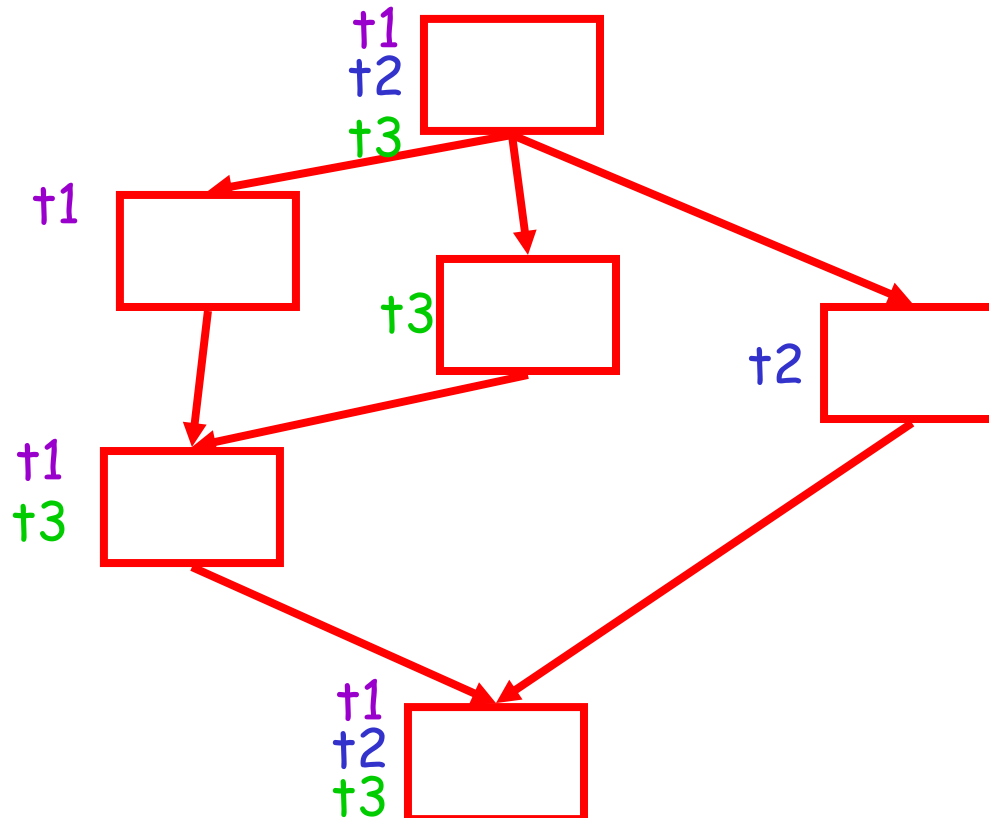
The Algorithm

Two pieces:

1. Run the tests and record for each basic block which tests reach that block

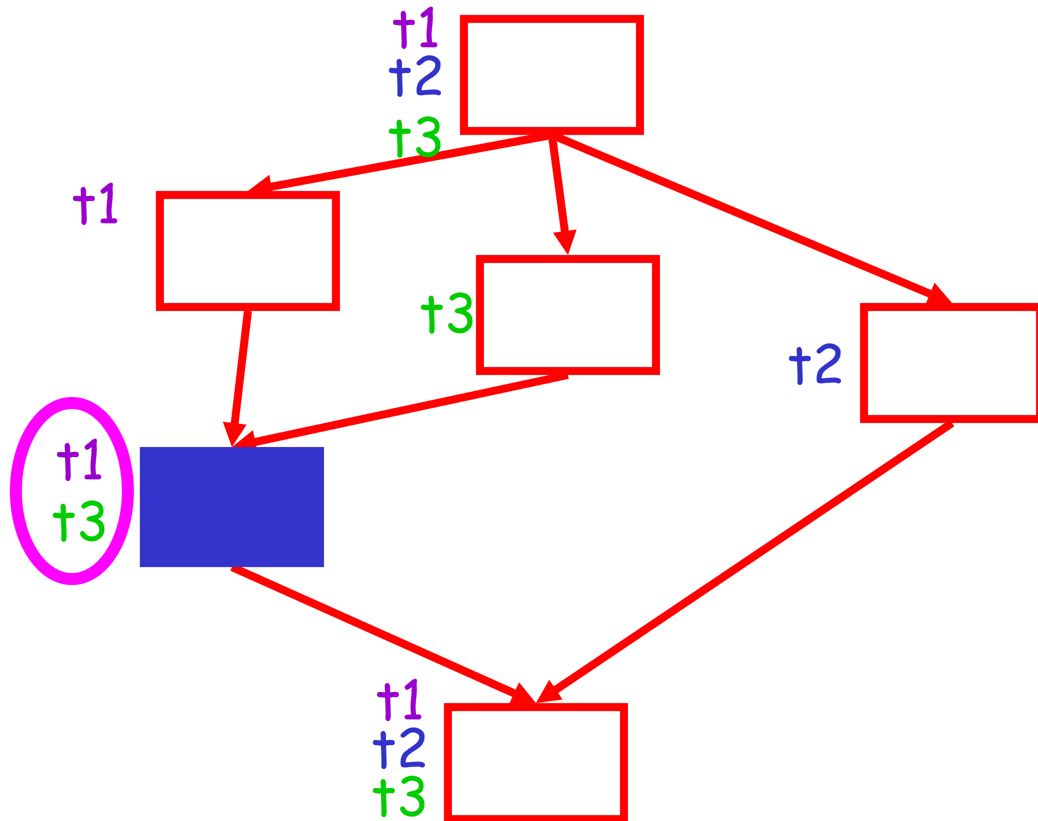
2. After modifications, do a DFS of the new control flow graph. Wherever it differs from the original control flow graph, run all tests that reach that point

Example



Label each node of the control flow graph with the set of tests that reach it.

Example (Cont.)



When a statement is modified, rerun just the tests reaching that statement

Experience

- This works
  - And it works better on larger programs
  - # of test cases to rerun reduced by > 90%

- Total cost less than cost of running all tests
  - Total cost = cost of tests run + cost of tool

- Why not use this?

What is a Good Test?

- We're implementing a function F on domain D
- A test set $T \subseteq D$ is *reliable* if for all programs P

$$(\forall t \in T.\ P(t) = F(t)) \Rightarrow (\forall t \in D.\ P(t) = F(t))$$

- Says that a good test set is one that implies the program meets its specification

Good News/Bad News

- Good News
  - There are interesting examples of reliable test sets
  - Example: *A function that sorts N numbers using comparisons sorts correctly iff it sorts all inputs consisting of 0,1 correctly*
  - This is a finite reliable test set

- Bad News
  - There is no effective method for generating finite reliable test sets

An Aside

- It's clear that reliable test sets must be impossible to compute in general
- But most programs are not diagonalizing Turing machines . . .
- It ought to be possible to characterize finite reliable test sets for certain classes of programs

What is a Good Test?

- We're implementing a function F on domain D
- A test set $T \subseteq D$ is *reliable* if for all programs P

$$(\forall t \in T.\ P(t) = F(t)) \Rightarrow (\forall t \in D.\ P(t) = F(t))$$

- equivalently, for all programs P

$$(\exists t \in D.\ P(t) \neq F(t)) \Rightarrow (\exists t \in T.\ P(t) \neq F(t))$$

- But we can't afford to quantify over all programs . . .

From Infinite to Finite

- We need to cut down the size of the problem
  - Check reliability wrt a smaller set of programs

- Idea: Just check a finite number of (systematic) variations on the program
  - E.g., replace x > 0 by x < 0
  - Replace I by I+1, I-1

- This is *mutation analysis*

Mutation Analysis

- Modify (mutate) each statement in the program in finitely many different ways
- Each modification is one *mutant*
- Check for adequacy wrt the set of mutants
  - Find a set of test cases that distinguishes the program from the mutants

What Justifies This?

- The “competent programmer assumption”
  The program is close to right to begin with

- It makes the infinite finite
  We will inevitably do this anyway; at least here it is clear what we are doing

The Plan

- Generate mutants of program P
- Generate tests
  - By some process
- For each test t
  - For each mutant M
    - If $M(t) \neq P(t)$ mark M as *killed*
- If the tests kill all mutants, the tests are reliable

The Problem

- This is dreadfully slow

- Lots of mutants
- Lots of tests
- Running each mutant on each test is expensive

- But early efforts more or less did exactly this

Better Algorithms

- Observation: Mutants are nearly the same as the original program
- Idea: Compile one program that incorporates and checks all of the mutations simultaneously
  - A so-called *meta-mutant*
- Weak mutation
  - Check only that mutant produces different state after mutation, not different final output

Metamutant with Weak Mutation

- Constructing a metamutant for weak mutation is straightforward

- A statement has a set of mutated statements
  - With any updates done to fresh variables

    $X := Y \ll 1$ $\quad$ $X_1 := Y \ll 2$ $\quad$ $X_2 := Y \gg 1$
  - After statement, check to see if values differ

    X == X_1 $\quad$ X == X_2

Comments

- A metamutant for weak mutation should be quite practical
  - Constant factor slowdown over original program
- If test suite fails to kill all mutants, then (maybe) its inadequate