Testing Research
Lecture 3
CS195

Outline

- Overview of testing research
  - Definitions, goals
- Three topics
  - Random testing
  - Efficient regression testing
  - Mutation analysis

Overview

- Testing research has a long history
  - At least to the 1960s
- 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

What is a Good Test?

- Attempt 1:
  If program $P$ implements function $F$ on domain $D$, then a test set $T \subseteq D$ is reliable if
    $$(\forall t \in T, P(t) = F(t)) \Rightarrow \forall t \in D, P(t) = F(t)$$
- 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 01 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
Adequacy

- Reliability is not useful if we don’t have a full reliable test set
  - Then it is possible that the program passes every test, but is not the program we want

- A different definition
  If program P implements function F on domain D, then a test set \( T \subseteq D \) is adequate if
  \( (\forall \text{progs } Q. Q(D) \neq F(D)) \implies \exists t \in T. Q(t) \neq F(t) \)

Adequacy

- Adequacy just says that the test suite must make every incorrect program fail
  - This seems to be what we really want

Random Testing

- About 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 (aka the paper)
  - Harder for systems with more arcane input
    - E.g., parser for context-free grammar

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 10%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

![Diagram] Label each node of the control flow graph with the set of tests that reach it.

Example (Cont.)

![Diagram] 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?

Adequacy (Review)

If program P implements function F on domain D, then a test set \( T \subseteq D \) is adequate if
\[
(\forall q \in Q, Q(D) \neq P(D)) \Rightarrow \exists t \in T, Q(t) \neq F(t)
\]

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 adequacy w/r.t 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 infinitely many different ways

- Each modification is one mutant

- Check for adequacy w/r.t 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

- This already generalizes existing metrics
  If it is not the end of the road, at least it is a step forward

The Plan

- Generate mutants of program $P$

- Generate tests
  - By some process

- For each test $t$
  - For each mutant $M$
    - If $M(t) = P(t)$ mark $M$ as killed
  - If the tests kill all mutants, the tests are adequate

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

Simplifications

- To make progress, we can either
  - Strengthen our algorithms
  - Weaken our problem

- To weaken the problem
  - Selective mutation
    - Don't try all of the mutants
  - Weak mutation
    - Check only that mutant produces different state after mutation, not different final output
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

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
    \[ X := Y < 1 \quad X_1 := Y < 2 \quad X_2 := Y > 1 \]
  - 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

• Paper doesn’t discuss metamutants with weak mutation explicitly
  - Not clear what is actually done
  - Not clear how to build a metamutant for stronger mutation models

Generating Tests

• Mutation analysis seeks to generate adequate test sets automatically

• Must determine inputs such that
  - Mutated statement is reached
  - Mutated statement produces a result different from original

Automatic Test Generation

• This is not easy to do

• Approaches
  - Use weakest-preconditions
  - Work backwards from statement to inputs
    - Take short paths through loops
    - Generate symbolic constraints on inputs that must be satisfied
    - Solve for inputs

Automatic Test Generation (Cont.)

• Work forwards from inputs
  - Try arbitrary inputs
  - Adjust as necessary to take desired branches

• Hybrids
  - Keep track of sets of values
  - When things get too complicated, commit to single value for one variable
    - Reduced dimension of problem
Comments on Test Generation

- Apparently works well for
  - Small programs
  - Without pointers
  - For certain classes of mutants

- So not very clear how well it works in general
  - Note: Solutions for pointers are proposed

A Problem

- What if a mutant is equivalent to the original?

  - Then no test will kill it

- In practice, this is a real problem
  - Not easily solved
    - Try to prove programs equivalence automatically
    - Often required manual intervention
  - Undermines the metric

Opinions

- Mutation analysis is a good idea
  - For all the reasons cited before
  - Also technically interesting
  - And there is probably more to do . . .

- How important is automatic test generation?
  - Still must manually look at output of tests
    - This is a big chunk of the work, anyway
  - Automatic tests likely to be weird
    - Both good and bad
  - Mutation analysis makes sense without automatic test generation