Software Testing, Quality Assurance & Maintenance—Lecture 21

Patrick Lam

February 27, 2015

Today

Midterm Review!

Coverage

Idea: find a reduced space and cover it with tests.

Seen so far: graph (structural, dataflow); syntax.

How Code Goes Bad

- **Fault** (also known as a bug): A static defect in software—incorrect lines of code.
- **Error**: An incorrect internal state—follows execution of a fault, but not necessarily observed yet.
- **Failure**: External, incorrect behaviour with respect to the expected behaviour—must be visible (e.g. EPIC FAIL).

— and —

to manifest a failure (RIP):

1. Fault must be **reachable**;
2. Program state subsequent to reaching fault must be incorrect: **infection**; and
3. Infected state must **propagate** to output to cause a visible failure.

Talking about Coverage Criteria

Let's go top-down.

Coverage criterion:
imposes a set of **test requirements** (TRs);
a **test set** may cover a set of TRs.

Test requirement:
a condition that some **test case** must satisfy.

Test set:
a collection of **test cases**.

Test case:
a set of inputs & corresponding expected outputs
(+prefix values, +postfix values).

Subsumption

Sometimes, any test set that satisfies criterion X
will also satisfy criterion Y.

Then we say that X **subsumes** Y.

This is a mostly-theoretical point;
in particular, some criteria are hard to use,
and it's always impractical to get to 100% anyway.

There is a subsumption chart in the notes.
Look at it.

Part I

Graph Coverage

Terms for Graphs

- path, subpath, subsequence;
- test path;
  - test path(s) induced by a test case ($path_G(t)$)
  - nondeterminism and test paths;
- reachability: semantic and syntactic.

Note: sets of test paths satisfy coverage criteria,
but we run test cases.

Criteria You'll Encounter Later

- Node Coverage (NC).
  (aka statement coverage)
- Edge Coverage (EC).
  (aka branch coverage)

No one but us cares about these graph criteria

- Edge Pair Coverage (EPC)
- Prime Path Coverage (PPC)
- Complete Path Coverage (CPC)
- Prime Path Coverage (PPC)
- Simple/Complete Round Trip Coverage (SRTC/CRTC)
- Specified Path Coverage (SPC)
- Bridge Coverage (BC)

(. . . but they are fair game for exams.)

Graphs and Code

Be able to draw Control Flow Graphs.
(& basic blocks)

Data flow criteria

Seem like a good idea:

focus on movement of data around a program.

Terms:

- def, use, du-pair;
- def *reaches* a use along a *def-clear* path;
- du-path;
- def-path set, def-pair set;
- All-Defs Coverage, All-Uses Coverage.

Data flow criteria and Call Graphs

nodes: methods; edges: method calls.

last-def/first-use optimization

Graph Coverage Criteria in Practice

Industry uses statement coverage and branch coverage.
(easy to measure)

But, what do they mean?

1. Low coverage: your code is untested!
2. High coverage: . . . well, who knows?

Research shows that bigger suites detect more bugs,
and higher-coverage suites are bigger,
but not that higher coverage is intrinsically better.

Part II

Syntax-Based Testing

Two Ways of using Grammars

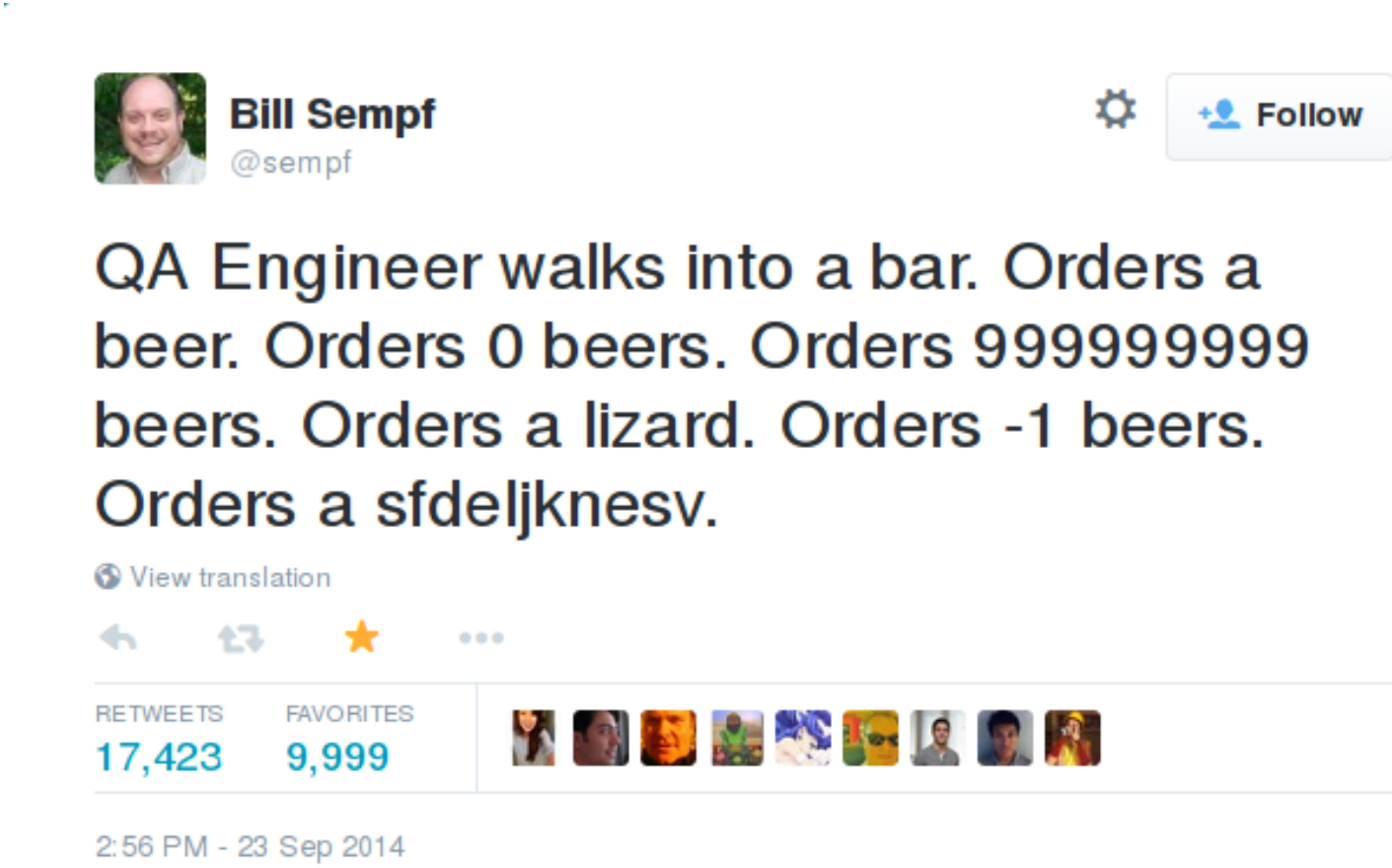
- input spaces
- programs (mutation testing)

Input Spaces

Test case generation:

- generate valid inputs from grammar;
- generate invalid inputs by modifying grammar.

Can use grammar mutation operators to modify grammar.



https://twitter.com/sempf/status/514473420277694465

Mutation Testing

http://en.wikipedia.org/wiki/File:Xav-lopr.png

Mutation Testing: Key Idea

Improve test suites
by forcing them to detect
known-wrong programs.

Carrying out Mutation Testing

1. generate mutants;
2. eliminate equivalent mutants;
3. ensure tests **kill** enough mutants (add tests if necessary).

Why Mutation Testing is Hard

Tools can generate mutants for you.

But you must still:

1. figure out equivalence;
2. design a test case
   to kill the mutant.

Is mutation testing all worth it?

Yes, probably.

Test suites which kill mutants
also detect real bugs.

Part III

Concurrency Bugs

Problems with concurrency

- race conditions;
- atomicity violations;
- deadlocks.

Detecting lock problems: paired calls

```
/* 2.4.0:drivers/sound/cmpci.c:cm_midi_release: */
lock_kernel(); // [PL: GRAB THE LOCK]
if (file->f_mode & FMODE_WRITE) {
  add_wait_queue(&s->midi.owait, &wait);
  ...
  if (file->f_flags & O_NONBLOCK) {
    remove_wait_queue(&s->midi.owait, &wait);
    set_current_state(TASK_RUNNING);
    return -EBUSY; // [PL: OH NOES!!1]
  }
  ...
}
unlock_kernel();
```

Problem: lock() and unlock() must be paired!

aComment: inferring lock disciplines

- extract locking-related annotations from code;
- extract locking-related annotations from comments;
- propagate annotations to callers.

Beliefs

MUST beliefs:

violations are clearly wrong.

MAY beliefs:

- need more evidence of wrongdoing.

General statistical technique

"a(); . . . b();" implies MAY-belief that a() followed by b().
(is it real or fantasy? we don't know!)

Algorithm:

- assume every `a`–`b` is a valid pair;
- emit "check" for each path with "a()" and then "b()";
- emit "error" for each path with "a()" and no "b()".

(actually, prefilter functions that look paired).

Part IV

Tool Support

Laundry List I

- iComment/aComment
- FindBugs
- Java Path Finder
- Korat
- Randoop
- Daikon
- ESC/Java
- Valgrind
- Flawfinder

Laundry List II

- Clang static analyzer
- cppcheck, sparse, splint
- pex
- KLEE
- Coverity, CodeSonar, Visual Studio
- PCLint, PVS-Studio, Fortify
- Intel Parallel Studio XE
- KlocWork Insight
- ScalaTest, ScalaCheck, Jacoco
- Atlassian Bamboo