CSE503: SOFTWARE ENGINEERING
TESTING

David Notkin
Spring 2011

Andreas Zeller's talk

2

- Comments or questions?

Today

3

- Some basics of software testing
  - Characterizations of testing
  - Terminology
  - Basic approaches
  - Mutation testing
  - Random (feedback-directed) testing
- Next week: symbolic evaluation, concolic evaluation, automatic test generation and related topics – in some depth – many of the techniques used by Andreas (and others!)

An example from Bach

4

- Asks students to "try long inputs" for a test requiring an integer
- Interesting lengths are…?

Enter an integer:

Key boundaries: most not tried

- 16 digits+: loss of mathematical precision
- 23+: can't see all of the input
- 310+: input not understood as a number
- 1000+: exponentially increasing freeze when navigating to the end of the field by pressing <END>
- 23,829+: all text in field turns white
- 2,400,000: reproducible crash

- Why more not tried?
  - Seduced by what's visible
  - Think they need the specification to tell them the maximum – and if they have one, stop there
  - Satisfied by first boundary
  - Use linear lengthening strategy
  - Think "no one would do that"

Free association: "software testing"

- Shout it out!

- Have any of you worked as a software tester?

Many views of testing

- Showing you did something right vs. showing somebody else did something wrong
- Getting useful software into users' hands vs. stopping buggy software from getting into users' hands
- Finding defects vs. building confidence in properties
- Finding new bugs vs. making sure the rest of the team can make progress
- …

Steve McConnell

Testing by itself does not improve software quality. Test results are an indicator of quality, but in and of themselves, they don't improve it. Trying to improve software quality by increasing the amount of testing is like trying to lose weight by weighing yourself more often. What you eat before you step onto the scale determines how much you will weigh, and the software development techniques you use determine how many errors testing will find. If you want to lose weight, don't buy a new scale; change your diet. If you want to improve your software, don't test more; develop better.

Cem Kaner & James Bach

9

- "Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the software under test."
- "Testing is questioning a product in order to evaluate it.
  - "The 'questions' consist of ordinary questions about the idea or design of the product, or else questions implicit in the various ways of configuring and operating the product. The product 'answers' by exhibiting behavior, which the tester observes and evaluates."

Herb Simon (via wikipedia)

10

- "Satisficing … is a decision-making strategy which attempts to meet criteria for adequacy, rather than to identify an optimal solution. A satisficing strategy may often be (near) optimal if the costs of the decision-making process itself, such as the cost of obtaining complete information, are considered in the outcome calculus."
- "[Simon] pointed out that human beings lack the cognitive resources to maximize: we usually do not know the relevant probabilities of outcomes, we can rarely evaluate all outcomes with sufficient precision, and our memories are weak and unreliable. A more realistic approach to rationality takes into account these limitations: This is called bounded rationality."

Quotations

11

- "Beware of bugs in the above code; I have only proved it correct, not tried it." – D. Knuth
- "Program testing can be used to show the presence of bugs, but never to show their absence!" –E. Dijkstra
- "It is not a test that finds a bug but it is a human that finds a bug and a test plays a role in helping the human find it." – P. Soundarajan
-

  MJ quotation?

Terminology

12

- *Failure* – an externally-visible outcome that is inconsistent with the specification
  - This generally includes program crashes, exceptions that aren't handled, etc.
  - This also generally includes inconsistencies with the implicit specification
- *Fault* – an inconsistent internal state
  - These may or may not lead to failures
- *Defect* – the piece of code that leads to a failure or fault
- *Error* – the human misunderstanding that led to the defect

Tests vs. test inputs

13

- A *test* defines both inputs and expected outputs
  - The expected output for a test is usually called an *oracle*
- A *test input* defines only the inputs
  - These can be useful in identifying failures such as crashes – there is no output to compare to an oracle
  - They can be useful in assessing *coverage* properties
- Like most of the world, even in published papers, I may not be very careful about this distinction – but push if it's confusing!

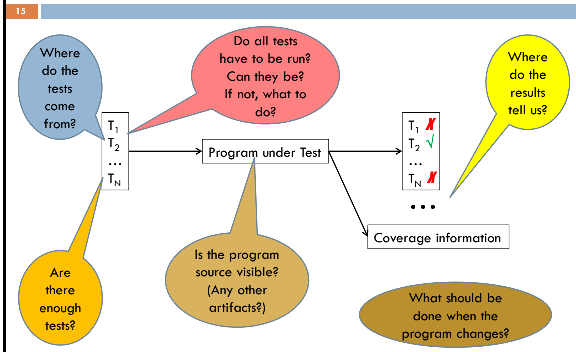
Do tests **pass** or **fail**?

14

- Does a test where the output matches the oracle **pass** or **fail**?
- Does a test input that terminates normally **pass** or **fail**?

Some key questions *Others?*

15



Testing theory

16

- Plenty of negative results
  - Nothing guarantees correctness
  - Statistical confidence is prohibitively expensive
  - Being systematic may not improve fault detection (as compared to simple random testing)
- "So what did you expect, decision procedures for undecidable questions?" – M. Young

When can we stop?

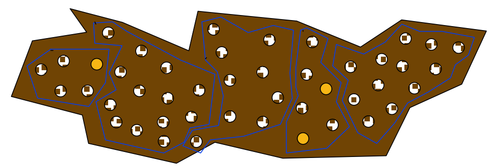
17

- Ideally: adequate testing ensures some property (proof by cases)
  - Goodenough & Gerhart, Weyuker & Ostrand
  - In reality, as impractical as other program proofs
- Practical adequacy criteria are really "inadequacy" criteria
  - If no case from class X has been chosen, surely more testing is needed …

Partition testing

18

- Basic idea: divide program input space into (quasi-) equivalence classes, selecting at least one test case from each class
- The devil is in the details – and there are many!



Structural coverage testing

19

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

Statement coverage

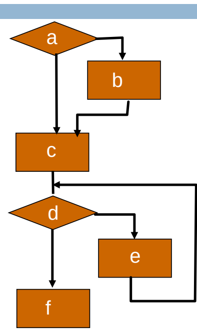
- Unsatisfying in trivial cases

```
if x > y then
  max := x
else
  max :=y
endif


if x < 0 then
  x := -x
endif
z := x;
```

Edge coverage

- Covering all basic blocks (nodes, statements) would not require edge **ac** to be covered
- Edge coverage requires all control flow graph edges to be coverage by at least one test



Condition coverage

- How to handle compound conditions?
  - `if (p != NULL) && (p->left < p->right) …`
- Is this a single conditional in the CFG? How do you handle short-circuit conditionals?
- Condition coverage treats these as separate conditions and requires tests that handle all combinations
- Modified Condition/Decision Coverage (MCDC)
  - Sufficient test cases to verify whether every condition can affect the result of the control structure
  - Required for aviation software by RCTA/DO-178B

Path coverage

- Edge coverage is in some sense very static
- Edges can be covered without covering actual paths (sequences of edges) that the program may execute
- All paths in a program may not be executable
  - Writing tests for these is hard ☺
  - Not shipping a program until these paths are executed does not provide a competitive advantage ☺

Path coverage

- The test suite {<x = 0, z = 1>, <x = 1, z = 3>} executes all edges, but…

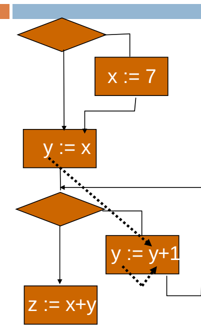
```
if x ≠ 0 then
  y := 5;
else
  z := z - x;
endif;
if z > 1 then
  z := z / x;
else
  z := 0;
end
```

Loop coverage

25

- Loop coverage also makes path coverage complex
  - Each added iteration through a loop introduces a new path
  - Since we can't in general bound the number of loop iterations, we often partition the paths for testing purposes
    - Never, once, many times …
    - 10 is a constant often used as a representation of "many"

Data flow coverage criteria



- Idea: an untested def-use pair could hide an erroneous computation
- The increment of **y** has two reaching definitions
- The assignment to **z** has two reaching definitions for each of **x** and **y**
- There are many variants on this kind of approach

Structural coverage: challenges

27

- Interprocedural coverage
  - Interprocedural dataflow, call-graph coverage, etc.
- Regression testing
  - How to test version P' given that you've tested P
- Late binding in OO – coverage of polymorphism
- Infeasible behaviors: arises once you get past the most basic coverage criteria

Infeasibility problem

28

- Syntactically indicated behaviors that are not semantically possible
- Thus can't achieve "adequate" behavior of test suites
- Could
  - Manually justify each omission
  - Give adequacy "scores" – for example, 95% statement, 80% def-use, …
  - [Can be deceptive, of course]
- Fault-injection is another approach to infeasibility

Mutation testing

29

- Mutation testing is an approach to evaluate – and to improve – test suites
- Basic idea
  - Create small variants of the program under test
  - If the tests don't exhibit different behavior on the variants then the test suite is not sufficient
- The material on the following slides is due heavily to Pezzè and Young on fault-based testing

Estimation

30

- Given a big bowl of marbles, how can we estimate how many?
- Can't count every marble individually

What if I also…

31

- … have a bag of 100 other marbles of the same size, but a different color (say, black) and mix them in?
- Draw out 100 marbles at random and find 20 of them are black
- How many marbles did we start with?

Estimating test suite quality

32

- Now take a program with bugs and create 100 variations each with a new and distinct bug
  - Assume the new bugs are exactly like real bugs in every way
- Run the test suite on all 100 new variants
  - ... and the tests reveal 20 of the bugs
  - … and the other 80 program copies do not fail
- What does this tell us about the test suite?

Basic Assumptions

33

- The idea is to judge effectiveness of a test suite in finding real faults by measuring how well it finds seeded fake faults
- Valid to the extent that the seeded bugs are representative of real bugs: not necessarily identical but the differences should not affect the selection

Mutation testing

34

- A mutant is a copy of a program with a mutation: a syntactic change that represents a seeded bug
  - Ex: change `(i < 0)` to `(i <= 0)`
- Run the test suite on all the mutant programs
- A mutant is killed if it fails on at least one test case
  - That is, the mutant is distinguishable from the original program by the test suite, which adds confidence about the quality of the test suite
- If many mutants are killed, infer that the test suite is also effective at finding real bugs

Mutation testing assumptions

35

- Competent programmer hypothesis: programs are nearly correct
  - Real faults are small variations from the correct program and thus mutants are reasonable models of real buggy programs
- Coupling effect hypothesis: tests that find simple faults also find more complex faults
  - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults, too

Mutation Operators

36

- Syntactic change from legal program to legal program and are thus specific to each programming language
- Ex: constant for constant replacement
  - from `(x < 5)` to `(x < 12)`
  - Maybe select from constants found elsewhere in program text
- Ex: relational operator replacement
  - from `(x <= 5)` to `(x < 5)`
- Ex: variable initialization elimination
  - from `int x =5;` to `int x;`

Live mutants scenario

37

- Create 100 mutants from a program
  - Run the test suite on all 100 mutants, plus the original program
  - The original program passes all tests
  - 94 mutant programs are killed (fail at least one test)
  - 6 mutants remain *alive*
- What can we learn from the living mutants?

503 11sp © UW CSE • D. Notkin

How mutants survive

38

- A mutant may be equivalent to the original program
  - Maybe changing `(x < 0)` to `(x <= 0)` didn't change the output at all!
  - The seeded "fault" is not really a "fault" – determining this may be easy or hard or in the worst case undecidable
- Or the test suite could be inadequate
  - If the mutant could have been killed, but was not, it indicates a weakness in the test suite
  - But adding a test case for just this mutant is likely a bad idea – why?

503 11sp © UW CSE • D. Notkin

Weak mutation: a variation

39

- There are lots of mutants – the number of mutants grows with the square of program size
- Running each test case to completion on every mutant is expensive
- Instead execute a "meta-mutant" that has many of the seeded faults in addition to executing the original program
  - Mark a seeded fault as "killed" as soon as a difference in an intermediate state is found – don't wait for program completion
  - Restart with new mutant selection after each "kill"

503 11sp © UW CSE • D. Notkin

Statistical Mutation: another variation

40

- Running each test case on every mutant is expensive, even if we don't run each test case separately to completion
- Approach: Create a random sample of mutants
  - May be just as good for assessing a test suite
  - Doesn't work if test cases are designed to kill particular mutants

503 11sp © UW CSE • D. Notkin

In real life …

- Fault-based testing is a widely used in semiconductor manufacturing
  - With good fault models of typical manufacturing faults, e.g., "stuck-at-one" for a transistor
  - But fault-based testing for design errors – as in software – is more challenging
- Mutation testing is not widely used in industry
  - But plays a role in software testing research, to compare effectiveness of testing techniques
- Some use of fault models to design test cases is important and widely practiced

Summary

- If bugs were marbles …
  - We could get some nice black marbles to judge the quality of test suites
- Since bugs aren't marbles …
  - Mutation testing rests on some troubling assumptions about seeded faults, which may not be statistically representative of real faults
- Nonetheless …
  - A model of typical or important faults is invaluable information for designing and assessing test suites

Their slide deck (partial)

Feedback-directed Random Test Generation

(to appear in ICSE 2007)

Carlos Pacheco, Michael Ernst — MIT

Shuvendu Lahiri, Thomas Ball — Microsoft Research

January 19, 2007

Their slide deck (partial)

Random testing

- Select inputs at random from a program's input space
- Check that program behaves correctly on each input

- An attractive error-detection technique
  - Easy to implement and use
  - Yields lots of test inputs
  - Finds errors
    - Miller et al. 1990: Unix utilities
    - Kropp et al.1998: OS services
    - Forrester et al. 2000: GUI applications
    - Claessen et al. 2000: functional programs
    - Csallner et al. 2005, Pacheco et al. 2005: object-oriented programs
    - Groce et al. 2007: flash memory, file systems

Evaluations of random testing

Their slide deck (partial)

- Theoretical work suggests that random testing is as effective as more systematic input generation techniques (Duran 1984, Hamlet 1990)
- Some empirical studies suggest systematic is more effective than random
  - Ferguson et al. 1996: compare with chaining
  - Marinov et al. 2003: compare with bounded exhaustive
  - Visser et al. 2006: compare with model checking and symbolic execution

Studies are performed on small benchmarks,
they do not measure error revealing effectiveness,
and they use completely undirected random test generation.

Contributions

Their slide deck (partial)

- We propose feedback-directed random test generation
  - Randomized creation of new test inputs is guided by feedback about the execution of previous inputs
  - Goal is to avoid *redundant* and *illegal* inputs
- Empirical evaluation
  - Evaluate coverage *and* error-detection ability on a large number of widely-used, well-tested libraries (780KLOC)
  - Compare against systematic input generation
  - Compare against undirected random input generation

Random testing: pitfalls

Their slide deck (partial)

1. Useful test
```
Set t = new HashSet();
s.add("hi");
assertTrue(s.equals(s));
```

2. Redundant test
```
Set t = new HashSet();
s.add("hi");
s.isEmpty();
assertTrue(s.equals(s));
```

3. Useful test
```
Date d = new Date(2006, 2, 14);
assertTrue(d.equals(d));
```

4. Illegal test
```
Date d = new Date(2006, 2, 14);
d.setMonth(-1); // pre: argument >= 0
assertTrue(d.equals(d));
```

5. Illegal test
```
Date d = new Date(2006, 2, 14);
d.setMonth(-1);
d.setDay(5);
assertTrue(d.equals(d));
```

do not output

do not even create

Feedback-directed random test generation

Their slide deck (partial)

- Build test inputs incrementally
  - New test inputs extend previous ones
  - In our context, a test input is a method sequence
- As soon as a test input is created, execute it
- Use execution results to guide generation
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

Technique input/output

Their slide deck (partial)

- Input:
  - classes under test
  - time limit
  - set of contracts
    - Method contracts (e.g. "o.hashCode() throws no exception")
    - Object invariants (e.g. "o.equals(o) == true")
- Output: contract-violating test cases. Example:

no contracts violated up to last method call

```
HashMap h = new HashMap();
Collection c = h.values();
Object[] a = c.toArray();
LinkedList l = new LinkedList();
l.addFirst(a);
TreeSet t = new TreeSet(l);
Set u = Collections.unmodifiableSet(t);
assertTrue(u.equals(u));
```

fails when executed

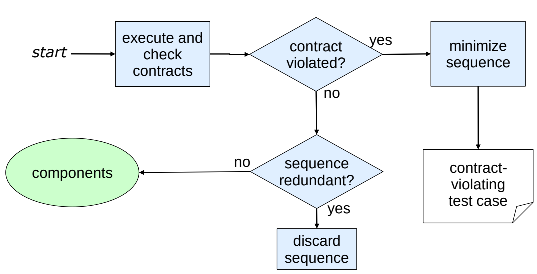
Technique

Their slide deck (partial)

1. Seed components
   components = { int i = 0; boolean b = false; ... }
2. Do until time limit expires:
   a. Create a new sequence
      i. Randomly pick a method call $m(T_1 \ldots T_k)/T_{ret}$
      ii. For each input parameter of type T_i, randomly pick a sequence S_i from the components that constructs an object v_i of type Ti
      iii. Create new sequence $S_{new} = S_1; \ldots ; S_k ; T_{ret}\ v_{new} = m(v1\ldots vk);$
      iv. if S_{new} was previously created (lexically), go to i
   b. Classify the new sequence S_{new}
      a. May discard, output as test case, or add to components

Classifying a sequence

Their slide deck (partial)



Redundant sequences

Their slide deck (partial)

- During generation, maintain a set of all objects created.
- A sequence is redundant if all the objects created during its execution are members of the above set (using `equals` to compare)
- Could also use more sophisticated state equivalence methods
  - E.g. heap canonicalization

Their slide deck (partial)

Coverage achieved by Randoop

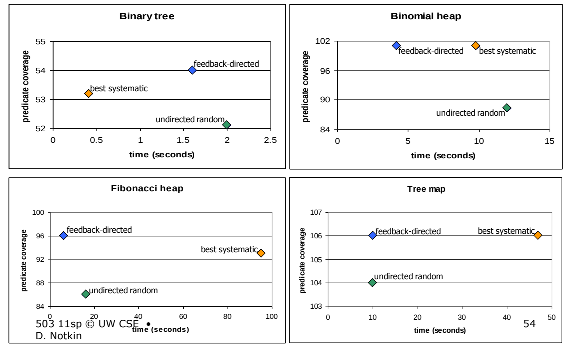
- Comparable with exhaustive/symbolic techniques

data structure	time (s)	branch cov.
Bounded stack (30 LOC)	1	100%
Unbounded stack (59 LOC)	1	100%
BS Tree (91 LOC)	1	96%
Binomial heap (309 LOC)	1	84%
Linked list (253 LOC)	1	100%
Tree map (370 LOC)	1	81%
Heap array (71 LOC)	1	100%

503 11sp © UW CSE • D. Notkin 53

Their slide deck (partial)

Predicate coverage



503 11sp © UW CSE • D. Notkin 54

Their slide deck (partial)

Evaluation: summary

- Feedback-directed random test generation:
  - Is effective at finding errors
    - Discovered several errors in real code (e.g. JDK, .NET framework core libraries)
  - Can outperform systematic input generation
    - On previous benchmarks and metrics (coverage), and
    - On a new, larger corpus of subjects, measuring error detection
  - Can outperform undirected random test generation

503 11sp © UW CSE • D. Notkin 55