CSE P503:
Principles of Software Engineering

David Notkin
Spring 2009

Tonight's agenda

- Testing: various
- May 21st
- One-minute paper

Mutation testing

- 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

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

Groups of 3-4

What if I also…

- … 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

- 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

- 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

- 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

- 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

- 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

- 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?

How mutants survive

- 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 undecideable
- 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 a bad idea – why?

Weak mutation: a variation

- 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"

Statistical Mutation: another variation

- 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

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

Symbolic execution

- Example from Visser, Pasareanu & Mehlitz

Concrete execution

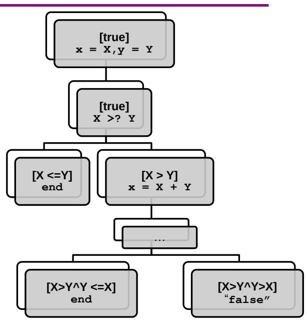
	[x = 1 ; y = 0]	[x = 0 ; y = 1]
`int x, y;`		
`if (x > y) {`	1 >? 0	0 >? 1
`x = x + y;`	x = 1 + 0 = 1	
`y = x – y;`	y = 1 – 0 = 1	
`x = x – y;`	x = 1 – 1 = 0	
`if (x – y > 0)`	0 – 1 >? 0	
`assert(false)`		
`}`		

Symbolic execution example

	[x = X ; y = Y]		
`int x, y;`			
`if (x > y) {`	X >? Y		
	F	T	
`x = x + y;`		x = X + Y	
`y = x – y;`		y = (X + Y) – Y = X	
`x = x – y;`		x = (X + Y) – X = Y	
`if (x – y > 0)`		Y – X >? 0	
`assert(false)`		F	T
`}`			"false"

What's really going on?

- Create a symbolic execution tree
- At nodes with predicates explicitly track path conditions
- Solving path conditions – "how do you get to this point in the execution tree?" – defines test inputs
- Goal: define test inputs that reach all reachable statements



Example: from Sen and Agha

```
int double (int v){
  return 2*v;
}
void testme (int x, int y){
  z = double (y);
  if (z == x) {
    if (x > y+10) {
      ERROR;
    }
  }
}
```

- Half of the groups: directly find concrete inputs that exercise all reachable statements
- Other half: do this using symbolic analysis

Groups of 3-4

Possible weaknesses of each?

Aside: test inputs vs. test cases

- Just to be clear…
- Although not used consistently, it is useful to distinguish *test inputs* (what goes in) from *test cases* (what goes in associated with what goes out)
  – That is, is there an oracle?
- Useful without oracles for what purposes?

Concolic testing:

- Basically, combine concrete and symbolic execution
- More precisely…
  – Generate a random concrete input
  – Execute the program on that input both concretely and symbolically simultaneously
  – Follow the concrete execution and maintain the path conditions along with the corresponding symbolic execution
  – Use the path conditions collected by this guided process to constrain the generation of inputs for the next iteration
  – Repeat until test inputs are produced to exercise all feasible paths

Concolic examples

- Standard approach applied to data structures (which are notoriously difficult to test)
- Variation that addresses situations where the constraints are hard to solve
- From Sen and Agha
- From Sağlam

Concolic: discussion

422 K. Sen and G. Agha

Table 1. Results for testing synchronized Collection classes of JDK 1.4. R/D/L/E stands for data race/deadlock/infinite loop/uncaught exceptions

Name	Run time in seconds	# of Paths	# of Threads	% Branch Coverage	# of Funs Tested	# of Bugs R/D/L/E
Vector	5519	20000	5	76.38	16	1/9/0/2
ArrayList	6811	20000	5	75	16	3/9/0/3
LinkedList	4401	11523	5	82.05	15	3/3/1/1
LinkedHashSet	7303	20000	5	67.39	20	3/9/0/2
TreeSet	7333	20000	5	54.93	26	4/9/0/2
HashSet	7449	20000	5	69.56	20	19/9/0/2

Test-driven development

- From www.agiledata.org: “Test-driven design (TDD) is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation. In other words, it’s one way to think through your design before your write your functional code. Another view is that TDD is a programming technique. As Ron Jeffries likes to say, the goal of TDD is to write clean code that works. I think that there is merit in both arguments, although I lean towards the specification view, but I leave it for you to decide.” [Scott Ambler]
- TDD = test-first design + refactoring [Ambler]
- “XP requires the buy-in of functional business groups outside of dev. TDD is a piece of it we can take with us and apply without needing the cooperation of anyone outside of the development team.” [John Roth]

Wikipedia sez

- “**Test-driven development** (TDD) is a software development technique that uses short development iterations based on pre-written test cases that define desired improvements or new functions. Each iteration produces code necessary to pass that iteration's tests. Finally, the programmer or team refactors the code to accommodate changes. A key TDD concept is that preparing tests before coding facilitates rapid feedback changes. Note that test-driven development is a software design method, not merely a method of testing.”

So, what’s the scoop?

- Fad? Real deal?

Discussion

Next Thursday

- Two choices
  - Michael Jackson video on your own; a serious one-page assessment
  - Curriculum development on Thursday night in groups
- Information on both goes out by email and on the web site tomorrow

N-version programming

- Idea: mimic hardware reliability using redundancy

$$P = \prod_{i=1}^{n} p_i$$

- Probability of a component failing is p_i
- Given independent failures, the probability of the whole system failing is the product of those failure rates
- Why not try it in software?

Optional…

- One-minute paper: Key point? Open question? Mid-course correction?