BLG 475E: Software Quality and Testing
Fall 2017-18

Fault-Based Testing

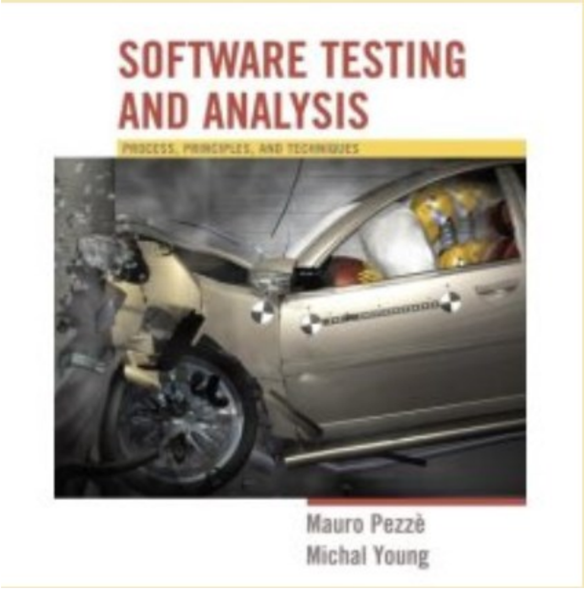
Dr. Ayşe Tosun

ISTANBUL TEKNİK ÜNİVERSİTESİ

1773

Agenda

- Chapter 16
  - Rationale of fault-based testing
  - Quality of test cases
  - Valid and invalid uses
  - Mutation testing



Estimation

- Suppose we have a big bowl of marbles. How can we estimate how many?
  - I don't want to count every marble individually
  - I have a bag of 100 other marbles of the same size, but a different color
  - What if I mix them?
  - Later, I draw out 100 marbles at random.
- 20 of them are black. How many marbles were in the bowl to begin with?

Test suite effectiveness

- Instead of marbles, if I seeded 100 bugs into my program, and my test suite can reveal 20 of the bugs
- What can I infer about my test suite?

- Valid to the extent that the seeded bugs are representative of real bugs
  - Not necessarily identical (e.g., black marbles are not identical to other marbles); but the differences should not affect the selection

Fault-based testing

- To select test cases that would distinguish the program under test from alternative programs that contain hypothetical faults
- To modify the program under test and produce hypothetically faulty programs
- ‘Fault seeding’

Mutation testing

- A *mutant* is a copy of a program with a *mutation*
- A *mutation* is a syntactic change (a seeded bug)
  - Example: change (i < 0) to (i <= 0)

- Run test suite on all the mutant programs
- A mutant is *killed* if it fails on at least one test case

- 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
    - => 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

```

/** Convert each line from standard input */
void transduce() {
    #define BUFLEN 1000
    char buf[BUFLEN]; /* Accumulate line into this buffer */
    int pos=0; /* Index for next character in buffer */

    char inChar; /* Next character from input */

    int atCR = 0; /* 0="within line", 1="optional DOS LF" */

    while ((inChar = getchar()) != EOF ) {
      switch (inChar) {
      case LF:
        if (atCR) { /* Optional DOS LF */
          atCR = 0;
        } else {     /* Encountered CR within line */
          emit(buf, pos);
          pos=0;
        }
        break;
      case CR:
        emit(buf, pos);
        pos=0;
        atCR = 1;
      break;
    default:
      if (pos >= BUFLEN-2) fail("Buffer overflow");
      buf[pos++] = inChar;
    }/* switch */
  }
  if (pos > 0) {
      emit(buf, pos);
  }
}
```

Figure 16.2 from Pezze and Young

Mutation Operators

- Syntactic change from legal program to legal program
  - So: Specific to each programming language. C++ mutations don’t work for Java, Java mutations don’t work for Python
- Examples:
  - crp: constant for constant replacement
    - for instance: from (x < 5) to (x < 12)
    - select from constants found somewhere in program text
  - ror: relational operator replacement
    - for instance: from (x <= 5) to (x < 5)
  - vie: variable initialization elimination
    - change int x =5; to int x;

Mutation Operators

- Operand modifications
- Expression modifications
  - e.g. absolute value insertion
  - aor: arithmetic operator replacement
    - $e1 \psi e2 \neq e1 \varphi e2$
    - replace arithmetic operator ψ with arithmetic operator φ
- Statement modifications
  - e.g. Deleting a statement
  - Replacing the label of one case in a switch with another
  - Move } one statement earlier or later

Example

- Assume you have a test suite for the program in Figure 16.2
  - TS = {1U, 1D, 2U, 2D, 2M, End, Long}
- Adequacy criteria of the tests %25

ID	Operator	Line	Original/Mutant	Which tests catch
Mi	ror	28	(pos >= BUFLEN-2) (pos == BUFLEN - 2)	None
Mj	ror	32	(pos >0) (pos >= 0)	1D,2U,2D, 2M
Mk	sdl	16	arCR = 0 nothing	None
Ml	ssr	16	atCR = 0	None

Live Mutants

- Scenario:
  - Create 100 mutants from the 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 whether a mutant is equivalent may be easy or hard; in the worst case it is undecideable
- 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. We care about the real bugs, not the fakes!

```
int edit1( char *s1, char *s2) {
        if (*s1 == 0) {
        if (*s2 == 0) return TRUE;
        /* Try inserting a character in s1 or deleting in s2
        if (*(s2+1)==0) return TRUE;
        return FALSE;
        }
        if (*s2 == 0) { /* Only match is by deleting last char
        if (*(s1 + 1) == 0) return TRUE;
        return FALSE;
        }
        /* Now we know that neither string is empty */
        if (*s1 == *s2) {
        return edit1(s1 +1, s2 +1);
        }
/* Mismatch; only dist 1 possibilities are identical strings by
* inserting, deleting, or substituting character
*/
/* Substitution: We "look past" the mismatched character
if (strcmp(s1+1, s2+1) == 0) return TRUE;
/* Deletion: look past character in s1 */
if (strcmp(s1+1, s2) == 0) return TRUE;
/* Insertion: look past character in s2 */
if (strcmp(s1, s2+1) == 0) return TRUE;
return FALSE;
}
```

Change with s1 + 1

Figure 16.1

Variations on Mutation

- Weak mutation
- Statistical mutation

Weak mutation

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the square of program size
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as "killed" as soon as a difference in intermediate state is found
    - Without waiting for program completion
    - Restart with new mutant selection after each "kill"

Statistical Mutation

- Problem: There are lots of mutants. Running each test case on every mutant is expensive
  - It's just too expensive to create N^2 mutants for a program of N lines (even if we don't run each test case separately to completion)
- Approach: Just create a random sample of mutants
  - May be just as good for *assessing* a test suite
    - Provided we don't design test cases to kill particular mutants (which would be like selectively picking out black marbles anyway)

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* is more challenging (as in software)
- Mutation testing is not widely used in industry
  - But plays a role in *software testing* research, to compare effectiveness of testing techniques
  - Some tools are Pit, Judy, Muclipse, etc.
- Some use of fault models to design test cases is important and widely practiced