EE360T/382V Software Testing

khurshid@ece.utexas.edu

March 19, 2018

Overview

Today

- Complete Chapter 5: Syntax-based testing

Next class – Exam 2 review

Homework– Problem Set 3 due (originally): 3/9

- You can submit until 3/19 11:59am with no penalty

Exam 2 – March 26, in-class

- Closed book, no cheat-sheet

EE360T/382V Software Testing

khurshid@ece.utexas.edu

Syntax-based testing (Chapter 5)*

*Introduction to Software Testing by Ammann and Offutt

Testing programs with mutation

Insight: in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault

Approach: given a program *p*

1. Create mutants of *p*
2. Remove redundant mutants (if feasible)
3. Generate a test suite for *p*
4. Run each test on *p* and its mutants to check mutant killing
5. Compute the mutation score for the test suite
6. Check *p*'s outputs for tests that kill some mutant(s)

Designing mutation operators

Two common strategies for operator design:

- Mimic developer mistakes, e.g., '<' → '>'
- Follow common heuristics, e.g., *failOnZero*(…) uses the heuristic "evaluate expression to 0"

Having more mutation operators means more mutants

Two common ways to control number of mutants

- Randomly sample from total mutants
- Only use *effective* mutation operators
  - Subset *E* of mutation operators *O* is effective if tests that kill mutants created by *E* also kill mutants created by *O* – *E* with high probability

Mutation operators for Java (1)

ABS – absolute value insertion

- *abs*(), *negAbs*(), *failOnZero*()

AOR – arithmetic operator replacement

- +, -, *, /, %, remove an operand/operator

ROR – relational operator replacement

- >, >=, <, <=, ==, !=; *false*, *true*

COR – conditional operator replacement

- &&, ||, &, |, ^; *false*, *true*

SOR – shift operator replacement

LOR – logical operator replacement

Mutation operators for Java (2)

ASR – assignment operator replacement

- *a* = *b*, *a* += *b*, *a* -= *b*, *a* *= *b*, *a* /= *b*, *a* %= *b*, *a* <<= *b*, *a* >>= *b*, *a* >>>= *b*, *a* &= *b*, *a* |= *b*, or *a* ^= *b*

UOI – unary operator insertion

- Arithmetic '+'/'-', conditional '!', logical '~'

UOD – unary operator deletion

SVR – scalar variable replacement

- "*x* = *a* * *b*" →
  "*x* = *a* * *a*", "*a* = *a* * *b*", "*x* = *x* * *b*", "*x* = *a* * *x*", "*x* = *b* * *b*", or "*b* = *a* * *b*"

FSR – failure statement replacement

5.3 Integration and OO testing

Integration mutation focuses on component connections

IPVR – integration parameter variable replacement

- "*m*(*x*)" → "*m*(*y*)"

IUOI – integration unary operator insertion

IPEX – integration parameter exchange

- "*m*(*a*, *b*)" → "*m*(*b*, *a*)"

IMCD – integration method call deletion

- replace call with constant if return value used

IREM – integration return expression modification

- Apply *UOI* or *AOR* to return expression

OO mutation operators focus on OO features

5.5 Input space grammars

Recall we can use a BNF grammar to describe inputs

Inputs for a class of programs (e.g., web services) can be defined using the eXtensible Markup Language (XML)

- XML *schemas* describe a form of grammar

Recall also mutation allows testing with invalid inputs

- When mutating grammars, the mutants themselves are tests
  - Killing mutants does not apply

Mutation operators for grammars

Nonterminal replacement – every nonterminal in a production is replaced by other nonterminals

Terminal replacement – every terminal in a production is replaced by other terminals

Terminal and nonterminal deletion – every terminal and nonterminal in a production is deleted

Terminal and nonterminal duplication – every terminal and nonterminal in a production is duplicated

Exercise

Recall our example grammar

$S \rightarrow M$

$M \rightarrow I\ N$

$I \rightarrow$ add | remove

$N \rightarrow D^{1-3}$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Show example applications of mutation operators for grammars

?/!