Software Testing
Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg
http://www.comp.nus.edu.sg/~abhik

Testing
- Most common form of SW checking.
  - Run program on selected inputs.
  - Observe outputs.
  - Match outputs against expectation.
- Programmer’s expectation of outputs.
  - May not capture program as a mathematical function.
    - Requires very deep understanding in the first place
  - But expected o/p for specific i/p

The trivial example
```c
int factorial(int n)
{
    if (n == 1) return 1;
    else return n*fact(n-1);
}
```
Could capture programmer’s expectation as
```
factorial(n) = n!   for all n
```
Now find suitable \( n \), and test the output of the factorial function against the expected output given by \( n! \).

Most of the times, this is not possible since the programmer does not have such a deep understanding of the program he/she is writing.

Why test software?
- Quote by Knuth
  - “Beware of bugs in this program. I have only proved it correct, not tried it.”
- Quote by Dijkstra
  - “Testing only shows the presence of bugs, not their absence”.
- Which SW will you put in your mother’s car?
  - Verified based on a model, but not tested.
  - “Thoroughly” tested, but not verified.

Programmer’s expectation
- Often of the form
  - For input \( i==0 \), expect output \( o == 5 \), or
  - For input \( i == 0 \), expect output \( o > 0 \), or
  - … in real-life, can be even of the form
- Run program against input \( i==0 \)
- Observe output
  - What is the observed output
  - Observed output \( o == 0 \)
  - Doesn’t look right, investigate
- Debug program for its execution with input \( i==0 \)

A Trivial Exercise in Testing
- Function triangle takes three integers \( a,b,c \) which are the length of triangle sides; calculates whether the triangle is equilateral, isosceles, or scalene.
- Try to write down test cases for this function (due to Myers)
How thorough can we be?

- Do you have a test case for an equilateral triangle?
- Do you have a test case for an isosceles triangle?
- Do you have at least three test cases for isosceles triangles, where all permutations are considered (e.g., (x,x,y), (x,y,x), (y,x,x))?
- Do you have a test case for an admissible scalene triangle?
- Do you have a test case with zero side zero?
- Do you have a test case with non-positive values?
- Do you have a test case with non-negative values but non-integers?
- Do you have a test case where the sum of two sides equals the third one?
- Do you have at least three such test cases, considering all permutations of sides are considered?
- Do you have a test case where the sum of the two smaller inputs is greater than the third one?
- Do you have test cases where the inputs exceed the MAXINT?
- Do you have test cases with non-integer values but numbers?

Testing isn’t so trivial!

- Myers 1979: this example should demonstrate that testing even a trivial program is not an easy task. Consider the problem of testing an air traffic guidance system with 100,000 instructions, a compiler or just a payroll program.

Why is testing important?

- As SW grows more complex
  - Less % of time in initial coding, modeling, requirements.
  - Greater % of time in testing & maintenance
    - Maintaining the SW as SW ages
    - Regression testing: testing a SW after changes, and see if any previously working functionality breaks.
    - Crucial in any large SW development project.

The high-level view

- Unit testing (we focus more on this now!)
  - Structural or Functional approaches
  - A unit can be a function or in the case of O-O programs, say a class
  - We will discuss control flow coverage criteria in details.
- Testing full programs
  - Integration testing
  - Regression testing (Check that the program still works after a feature is added to a tested program)
    - Discuss more in 3rd lecture
  - Stress testing (e.g. web-service with many users)

Common terminology

- Test case
  - A test input (or its execution trace)
- Test suite
  - Set of test cases
- Test purpose
  - A formal specification to guide testing
    - e.g., a regular expression which the test case should satisfy
- Coverage criterion
  - A guide to exhaustively cover program structure.
    - e.g., Statement coverage, Cond. coverage, Path coverage.

Structural vs. Functional Testing

- Functional (Black Box)
  - Boundary Value Testing
  - Equivalence Class Testing
  - Decision Table based Testing
- Structural (Glass Box or White Box)
  - Control flow Coverage Criteria
  - Data flow Coverage Criteria
Boundary value

Checking a “month” input variable for boundary values 0, 13
Can check for simple errors like

\[
\text{if } (\text{month} \geq 0) \land (\text{month} < 13)
\]

Need to get the boundary values by equivalence partitioning, or by general intuition (e.g. in the case of “month” variable)

Equivalence Partitioning

- Name is suggestive
  - “month” variable --- \( \leq 0, 1..12, > 12 \)
  - Can have different handling for diff. values
    - if \((\text{month} \geq 0) \land (\text{month} < 13)\)
    - if \((\text{month} < 4) \lor \ldots \)
    - else: /* different financial year */ ...
- Partitions \( \leq 0, 1..3, 4..12, > 12 \)
- Strictly speaking, a white box testing method

Decision Table based testing

Create such a decision table to check complex conditions. Based on the tester’s knowledge of what the program should do, rather than the structure of the program.

Decision Table based testing

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
</tr>
</tbody>
</table>

Structural (White Box) Testing

- Look inside the code
  - Discussion of control flow coverage criteria
    - Statement coverage
    - Branch coverage
    - ...

Statement coverage

Make the branch condition true
\((X = 1, Y = 1, Z = 2, W = 1)\)

Edge coverage

Make the branch condition true/false
\((X = 1, Y = 1, Z = 2, W = 2)\)
\((X = 1, Y = 1, Z = 2, W = 2)\)
Condition coverage

For each executable condition \( c \)
- Check whether it can be both true or false
- \( c \) could be unsatisfiable or valid in all pgm. executions
- For all such conditions \( c \), \( c \) should be true in at least one test in the test suite, and \( c \) should be false in at least one test in the test suite.

\[
\begin{align*}
Y &= Y + 1 \\
\text{true} & \rightarrow X = Y < W \\
\text{false} & \rightarrow X = X - 1
\end{align*}
\]

Path coverage

- Cover all paths in the program
  - Unboundedly many, unless loops can be bounded.
  - Lot of infeasible paths i.e. paths which do not form execution trace for any input.
    - Infeasible path detection will help test-suite construction.
    - A technique to help exercise new paths with new tests
      - Attempts to achieve path coverage
        - Basic idea: concrete and symbolic execution at the same time.

Example program

```c
if (Climb)
    separation = Up;
else
    separation = Up + 100;
if (separation > 150)
    upward = 1;
else
    upward = 0;
if (upward > 0)
    printf("Upward");
else
    printf("Downward");
```

Directed Automated Random Testing

- Start with a random input \( I \).
- Execute program \( P \) with \( I \)
  - Suppose \( I \) executes path \( p \) in program \( P \).
  - While executing \( p \), collect a symbolic formula \( f \) which captures the set of all inputs which execute path \( p \) in program \( P \).
  - Minimally change \( f \), to produce a formula \( f' \)
    - Solve \( f' \) to get a new input \( I' \) which executes a path \( p' \) different from path \( p \).
**Example program**

```c
if (Climb)
  separation = Up;
else
  separation = Up + 100;
if (separation > 150)
  upward = 1;
else
  upward = 0;
if (upward >0)
  printf("Upward");
else
  printf("Downward");
```

**Generating new tests**

- The path condition calculated
  - Climb ==0 ∧ Up + 100 > 150 ∧ upward > 0
- Minimally modify the condition
  - Climb ==0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)
- Corresponding to the path ...

**Infeasible path!!**

```c
if (Climb)
  separation = Up;
else
  separation = Up + 100;
if (separation > 150)
  upward = 1;
else
  upward = 0;
if (upward >0)
  printf("Upward");
else
  printf("Downward");
```

**Generating new tests**

- The path condition calculated
  - Climb ==0 ∧ Up + 100 > 150 ∧ upward > 0
- Minimally modify the condition
  - Climb ==0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)
- Corresponding to infeasible path!
- Modify a bit more
  - Climb == 0 ∧ ¬(Up + 100 > 150)
- Corresponding to the path ...

**Feasible path**

```c
if (Climb)
  separation = Up;
else
  separation = Up + 100;
if (separation > 150)
  upward = 1;
else
  upward = 0;
if (upward >0)
  printf("Upward");
else
  printf("Downward");
```

**Generating new tests**

- The path condition calculated
  - Climb ==0 ∧ Up + 100 > 150 ∧ upward > 0
- Minimally modify the condition
  - Climb ==0 ∧ Up + 100 > 150 ∧ ¬(upward > 0)
- Corresponding to infeasible path!
- Modify a bit more
  - Climb == 0 ∧ ¬(Up + 100 > 150)
- Solve to get another test input
  - Climb == 0, Up == 0
- Continue in this fashion.
Structural Testing (continued)

- Coverage Criteria
  - Control flow based
    - Statement, Edge, Condition, Path
  - Data flow based
    - All defs, All uses etc
  - Why need it?
    - Control flow criteria (except path coverage) do not exercise the use of a variable definition and the data flow.

```
int P1(int flag) {
    int x, y, z;
    if (x > 3)
        z = z + 1;
    else
        x = flag;
    if (y == 4)
        y = y + 1;
    else
        x = 1;
    if (x < 2)
        z = z / 2;
    else
        z = z - 1;
    y = x - z;
    if (y > 0)
        z = x + y;
    else
        z = -1;
    return z;
}
```

---

**def(x)**

Nodes where variable x is assigned

**p-use(x)**

Nodes where variable x is used in a predicate.

**c-use(x)**

Nodes where variable x is used in any expression other than a predicate (say rhs of assignment).

**def-clear(x)**

Set of paths which do not contain any node in def(x)

Typically consider acyclic paths
Given variable x, and s ∈ def(x)
dpu(s,x) = ( s | 3 def-clear(x) path from s to s' and  s' ∈ p-use(x) }

Given variable x, and s ∈ def(x)
dcu(s,x) = ( s | 3 def-clear(x) path from s to s' and  s' ∈ c-use(x) }

Coverage criteria
- **All defs**
  - For each variable x, and def. s ∈ def(x)
  - Include at least one def-clear(x) path from s to at least one node in dpu(s,x) ∪ dcu(s,x).

- **All uses**
  - For each variable x, and def. s ∈ def(x)
  - Include at least one def-clear(x) path from s to each node in dpu(s,x) and to each node in dcu(s,x).

- **All du-paths**
  - For each variable x, and def. s ∈ def(x)
  - Include all def-clear(x) path from s to each node in dpu(s,x) and to each node in dcu(s,x).

- **In terms of power**
  - All du-paths > All uses > All defs

Reading
- Most of the basic stuff is folklore.
- For a recent work on symbolic execution based testing see
  - [http://srl.cs.berkeley.edu/~ksen/slides/dart-fm.ppt](http://srl.cs.berkeley.edu/~ksen/slides/dart-fm.ppt)
  - This covers the portion on “Directed Automated Random Testing”. 