Software Testing: Review

Verification vs. Validation

- Verification:
  - ?????
- Validation:
  - ?????

http://www.umsec.umn.edu
Verification vs. Validation: Right Definition

- Verification:
  - “are we building the product right?”
  - The software should conform to its specification
- Validation:
  - “are we building the right product?”
  - The software should do what the user really requires

Assuring that a software system meets a user’s needs

Dynamic and Static Verification

- Dynamic V & V
  - Concerned with exercising and observing product behavior
  - Testing
- Static V & V
  - Concerned with analysis of the static system representation to discover problems
  - Proofs
  - Inspections
Static and Dynamic V&V

Requirements Specification → High-Level Design → Formal Specification → Detailed Design → Program

Static Verification

Prototype → Dynamic Evaluation

Definitions of Testing

- The process of executing a program (or part of a program) with the intention of finding errors (Myers, via Humphrey)
- The purpose of testing is to find errors
  - Testing is the process of trying to discover every conceivable fault or weakness in a work product (Myers, via Kit)
- The process of searching for errors (Kaner)
What is a Test?

- Test Data
  - Inputs which have been devised to test the system
- Output
- Correct result?
- Oracle
- Software under Test
- Test Cases

Test Data and Test Cases

- Test data
  - Inputs which have been devised to test the system
- Test cases
  - Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification

http://www.umsec.umn.edu
Bugs? What is That?

• Failure
  ◆ An execution that yields an erroneous result
• Fault
  ◆ The source of the failure
• Error
  ◆ The mistake that led to the fault being introduced in the code

Axiom of Testing

“Program testing can be used to show the presence of bugs, but never their absence.”

Dijkstra, 1969
Black and White Box

• Black box testing:
  ♦ Designed without knowledge of the program’s internal structure and design
  ♦ Based on functional requirements

• White box testing:
  ♦ Examines the internal design of the program
  ♦ Requires detailed knowledge of its structure

Black-Box Testing

• Approach to testing where the program is considered as a “black-box”
• The program test cases are based on the system specification
• Test planning can begin early in the software process
Black-Box Testing

- Input test data
- Software
- Output results

Input causing anomalous behavior

Output which reveal the presence of defects

Equivalence Partitioning

- Software
- Input test data
- Output results

http://www.umsec.umn.edu
Structural Coverage Testing

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

Statement Coverage

```c
int select(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] < X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```

One test datum (N=1, A[0]=-7, X=9) is enough to guarantee statement coverage of function select.
Faults in handling positive values of A[i] would not be revealed.
Branch Coverage

We must add a test datum \((N=1, A[0]=7, X=9)\) to cover branch False of the if statement. Faults in handling positive values of \(A[i]\) would not be revealed. Faults in exiting the loop with condition \(A[i] < X\) would not be revealed.

```c
int select(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] < X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```

Path Coverage

The loop must be iterated given number of times. PROBLEM: uncontrolled growth of test sets. We need to select a significant subset of test cases.

```c
int select(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] < X)
    {
        if (A[i]<0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}
```
Condition Coverage

int select(int A[], int N, int X)
{
    int i=0;
    while (i<N and A[i] < X)
    {
        if (A[i] < 0)
            A[i] = - A[i];
        i++;
    }
    return(1);
}

Both conditions (i<N), (A[i]<X) must be false and true for different tests. In this case, we must add tests that cause the while loop to exit for a value greater than X. Faults that arise after several iterations of the loop would not be revealed.

Basic Condition Coverage

• Make each condition both True and False

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Cond 1</th>
<th>Cond 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
### Compound Condition Coverage

- Evaluate every combination of the conditions

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Cond 1</th>
<th>Cond 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

### Compound Coverage (cont.)

- May lead to a lot of test cases

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Cond 1</th>
<th>Cond 2</th>
<th>Cond 3</th>
<th>Cond 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>2</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>3</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>4</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>9</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>10</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>11</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>12</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>13</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>14</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>15</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>16</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
The Infeasibility Problem

- Syntactically indicated behaviors (paths, data flows, etc.) are often impossible
  - Infeasible control flow, data flow, and data states
- Adequacy criteria are typically impossible to satisfy
- Unsatisfactory approaches:
  - Manual justification for omitting each impossible test case (esp. for more demanding criteria)
  - Adequacy “scores” based on coverage
    - example: 95% statement coverage, 80% def-use coverage

The Budget Coverage Criterion

- Industry’s answer to “when is testing done”
  - When the money is used up
  - When the deadline is reached
- This is sometimes a rational approach!
  - Implication 1:
    - Adequacy criteria answer the wrong question. Selection is more important.
  - Implication 2:
    - Practical comparison of approaches must consider the cost of test case selection
Top-down testing

Test Drivers

Level N-1

Level N

Testing Sequence

Bottom-Up Testing

Test Drivers

Level N-1

Level N

Testing Sequence
Create Scaffolding

Goal
To setup the environment for executing the test

Driver
- initialization of non-local variables
- initialization of parameters
- activation of the unit

Oracle
- check the correspondence between the produced and the expected result

Program Unit

Stub
- “templates” of modules used by the unit (functions called by the unit)
- “templates” of any other entity used by the unit

Generate Drivers and Stubs

generic (for all tests)  specific (for subsets of tests)

Brute force coding
- interactive: ask user for values
- automatic: (approximately) compute required values

From driver/stub specs
- parsing the unit to partially generate the framework
- add scripts to fill in the framework
Problems and Tradeoffs

effort in test execution and regression testing

- poorly designed drivers/stubs
  - low effort in development
  - high effort in test execution and regression testing
- high effort in development
- low effort in test execution and regression testing
- well designed drivers/stubs

effort in developing drivers/stubs

Who Should Test?

- Developer
  - Understands the system
  - But, will test gently
  - And, is driven by deadlines

- Independent tester
  - Must learn system
  - But, will attempt to break it
  - And, is driven by “quality”

http://www.umsec.umn.edu
Axioms of Testing

- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases
- Assign your best programmers to testing
- Exhaustive testing is impossible

Axioms of Testing

- You cannot test a program completely
- Even if you do find the last bug, you’ll never know it
- It takes more time than you have to test less than you’d like
- You will run out of time before you run out of test cases