Testing
Part 2
Three Important Testing Questions

• How shall we generate/select test cases?

• Did this test execution succeed or fail?

• How do we know when we’ve tested enough?
1. How do we know when we’ve tested enough?

Test Coverage Measures
Structural Coverage Testing

• Idea
  – Code that has never been executed likely has bugs
    • At least the test suite is clearly not complete

• This leads to the notion of code coverage
  – Divide a program into elements (e.g., statements)
  – Define the coverage of a test suite to be

\[
\frac{\text{# of elements executed by suite}}{\text{# of elements in program}}
\]
Types of Program Elements

• Selection of test suite is based on some elements in the code
• Assumption: Executing the faulty element is a necessary condition for revealing a fault
• Several types of elements
  – Control flow: statement, branch, path
  – Condition: simple, multiple
  – Loop
  – Dataflow
  – Fault based (mutation)
  – …
Control Flow Graphs: The One Slide Tutorial

X := 3;
if (B > 0)
    Y := 0;
else
    Y := Z + W;
A = 2 * 3;

• A graph
• Nodes are basic blocks
  • statements
• Edges are transfers of control between basic blocks
Control Flow Graph

```c
void testme1(int x) {
    int j = 0;
    for (j=0; j < 2; j++) {
        if (x == j) {
            printf("Good\n");
        }
    }
    x = j;
}
```

- A graph
- Nodes are basic blocks
  - statements
- Edges are transfers of control between basic blocks
void testme1(int x) {
    int j = 0;
    for (j = 0; j < 2; j++) {
        if (x == j) {
            printf("Good\n");
        }
    }
    x = j;
}
Statement Coverage

• Test requirements: Statements in a program

\[
\frac{\text{# of executed statements}}{\text{# of statements in the program}}
\]
Statement Coverage in Practice

• The good old days: (Stucki 1973)
  – Only about 1/3 of NASA statements were executed under test before software was released

• Better results:
  – Microsoft reports 80–90% statement coverage
  – Boeing must get 100% statement coverage (feasible?) for all software

• 100% can be hard or impossible (why?)
Statement Coverage: Example

- Test requirements
  - Nodes 3, ..., 9
- Test cases
  - (x = 20, y = 30)

Any problems with this example?
Statement Coverage: Example

- Test requirements
  - Nodes 3, ..., 9

- Test cases
  - \((x = 20, y = 30)\)

- Such test does not reveal the fault at statement 7

- To reveal it, we need to traverse edge 4–7

=> Branch Coverage
Branch Coverage

• Test requirements: Branches in a program

# of executed branches

# of branches in the program
Branch Coverage: Example

- Test requirements
  - Edges 4–6, Edges 4–7
- Test cases
  - \((x = 20, y = 30)\)
  - \((x = 0, y = 30)\)
Branch Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 0;
9.     if (y > 0)
10.    w = y / z;
11.
12. }

• Branch Coverage
• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = -10)
• Is the test suite adequate for branch coverage?
Branch Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 0;
9.     if (y>0)
10.    w = y / z;
11. }

• Branch Coverage
• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = -10)
• Is the test suite adequate for branch coverage?
  • Yes, but it does not reveal the fault at statement 10
• Test case (x = 0, y = 22)
  - Reveals fault
Data Flow Coverage

• Test requirements: Def–use pairs in a program

\[
\frac{\text{# of executed def–use pairs}}{\text{# of def–use pairs in the program}}
\]
Data Flow Coverage: Example

```
1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 0;
9.     if (y>0)
10.        w = y / z;
11.
12. }
```

• Test Requirements:
  - DU pairs 6-10, 8-10

• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = 10)

• Reveals fault
Data Flow Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 1;
9.     if (y>0)
10.        w = y / z;
11.     else
12.         assert(0);
13. }

• Test Requirements:
  - DU pairs 6-10, 8-10
• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = 10)
• Is the test suite adequate for data flow coverage?
Data Flow Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 1;
9.     if (y > 0)
10.        w = y / z;
11.    else
12.        assert(0);
13. }

• Test Requirements:
  - DU pairs 6-10, 8-10

• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = 10)

• Is the test suite adequate for data flow coverage?
  - Yes, but it does not reveal the fault at statement 12

• Test case (x = 1, y = -1)
  - Reveals fault
All Path Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 1;
9.     if (y>0)
10.        w = y / z;
11.     else
12.  }
13.  }

• Test Requirements:
  - 4 paths

• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = 10)
  - (x = 1, y = -22)
  - (x = 1, y = -10)
All Path Coverage: Example

1. main() {
2.     int x, y, z, w;
3.     read(x);
4.     read(y);
5.     if (x != 0)
6.         z = x + 10;
7.     else
8.         z = 1;
9.     if (y>0)
10.        w = y / z;
11.     else
12.        w = 0;
13. }

• Test Requirements:
  - 4 paths

• Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = 10)
  - (x = 1, y = -22)
  - (x = 1, y = -10)

• Faulty if x = -10
  - Structural coverage may not reveal this error
Pitfalls of Coverage

```c
status = perform_operation();
if (status == FATAL_ERROR)
    exit(3);
```

- Coverage says the `exit` is never taken
- Straightforward to fix
  - Add a case with a fatal error
Example

```c
status = perform_operation();
if (status == FATAL_ERROR)
    exit(3);
```

• Coverage says the `exit` is never taken
• Straightforward to fix
  – Add a case with a fatal error

• But are there other error conditions that are not checked in the code?
  – Coverage does not help with faults of omission
Code Coverage (Cont.)

• Code coverage has proven value
  – It’s a real metric, though far from perfect

• But 100% coverage does not mean no bugs
  – Many bugs lurk in corner cases
  – E.g., a bug visible after loop executes 1,025 times

• And 100% coverage is almost never achieved
  – Products ship with < 60% coverage
  – High coverage may not even be economically desirable
    • May be better to devote more time to tricky parts with good coverage
Coverage: the Bad assumptions

Logical Fallacies

• TRUE  If low coverage then poor tests; Not low coverage, therefore not poor tests

• TRUE  If good tests then high coverage; High coverage, therefore good tests
Coverage: the Bad assumptions

Logical Fallacies

• TRUE  If low coverage then poor tests;
  FALSE  Not low coverage, therefore not poor tests

• TRUE  If good tests then high coverage;
  FALSE  High coverage, therefore good tests
Using Code Coverage

• Code coverage helps identify weak test suites

• Tricky bits with low coverage are a danger sign

• Areas with low coverage suggest something is missing in the test suite
The Lesson

• Code coverage can’t complain about missing code
  – The case not handled
Structural Coverage in Practice

• Statement and sometimes edge or condition coverage is used in practice
  – Simple lower bounds on adequate testing

• Additional control flow heuristics sometimes used
  – Loops (never, once, many), combinations of conditions

• See “How to Misuse Code Coverage” in course schedule
Standard Testing Questions

- How shall we generate/select test cases?
- Did this test execution succeed or fail?
- How do we know when we’ve tested enough?
2. Was this test execution correct?
What is an Oracle?

• Oracle = alternative realization of the specification

• Examples of oracles
  – The “eyeball oracle”
    • Expensive, not dependable, lack of automation
  – A prototype, or sub-optimal implementation
    • E.g., bubble-sort as oracle for quick sort
  – A manual list of expected results
Record-and-Replay Oracles

• Record prior runs
• Test recording is usually very fragile
  – Breaks if environment changes anything
  – E.g., location, background color of textbox

• More generally, automation tools cannot generalize
  – They literally record exactly what happened
  – If anything changes, the test breaks

• A hidden strength of manual testing
  – Because people are doing the tests, ability to adapt tests
    to slightly modified situations is built-in
Result Checking

• Easy to check the result of some algorithms
  – E.g., computing roots of polynomials, vs. checking that the result is correct
  – E.g., executing a query, vs. checking that the results meet the conditions

• Not easy to check that you got all results though!
Assertions

• Use `assert(...)` liberally
  – Documents important invariants
  – Makes your code self-checking
    • reduces propagation from fault to failure
  – And does it on every execution!
    • for a performance price
  – You still have to worry about coverage

• May need to write functions that check invariants

• Opinion: Most programmers don’t use `assert` enough
Standard Testing Questions

• How shall we generate/select test cases?

• Did this test execution succeed or fail?

• How do we know when we’ve tested enough?
3. How Shall We Generate/Select Tests?

Automatic Test Generation
Fuzz Testing

• What is Fuzz Testing?
  – Throw random data at an app
  – See what breaks
Bugs Found Through Fuzzing

- About ¼ of Unix utilities crash when fed random input strings!!
- Microsoft estimates 20–25% of bugs found this way
- Examples
  - Microsoft JPEG parsing error
  - Linksys WRT54G wireless router crack
  - Device Driver crashes
  - Skype remote DOS from heap overflow
Office 2007: 4 bugs, 3 hours, 7 lines of python fuzzer


+ Unspecified Overflow in word 2007
  – Crash in wwlib.dll. Code execution is not trivial.

+ Word 2007 CPU exhaustion DOS
  – CPU shoots up to 100 %.

+ Word 2007 CPU exhaustion DOS + ding
  – CPU shoots up to 100 %, and windows goes .ding!.

+ Heap overflow in Windows HLP files
  – Funky heap overflow crash.
Random Bugs

• What sort of bugs does fuzz testing find?
  – Buffer overruns
  – Format string errors
  – Wild pointers/array out of bounds
  – Signed/unsigned characters
  – Failure to handle return codes
  – Race conditions
World’s Simplest Fuzzer

- while [ 1 ]; do cat /dev/urandom | nc -vv target port; done
  – (Sutton, 2007)

- Can be hard to figure out exactly what caused a crash
  – We’ll see a technique later
Why Fuzzers Work?

• Test engineers try to generate bad inputs
  – but human generated test cases tend to contain fewer tests than a fuzzer can produce.

• Test engineers may make implicit assumptions input
  – an automated, brainless fuzzer will just try anything
Fuzzing Techniques

• Dumb fuzzing
  – sends random bytes of data to a program and see what, if anything, happens

• Smart fuzzing
  – extends dumb fuzzing with more domain specific data
  – A fuzzer targeted at web applications might generate GET and POST queries using (and abusing) the variables that the form or page submits as well as adding in some random variables and values.
  – A fuzzer targeting a web browser might generate random input that conformed to HTML syntax, with random tags and attributes as well as abusing the defined tags.

• A combination of both dumb and smart fuzzing is likely the best approach.
Fuzz Generation: String Heuristics

• Very long strings, even of single character
• Strings of high-ASCII
• Non-alphanumerics:
  – !@#$%^&*()_+-+=:;'"
• Unicode:
  – non-ASCII malformed surrogate pairs, ASCII encoded in several bytes, etc.
• Directory traversal:
  – .., ../, ../.., ..../..../.. etc.
  – Sendmail vulnerability “//////////.... /”, “<<><><><>...<<>”
• Format string: %n%n%n%n %s%n%s%n
Fuzz + Enumeration

• Sometimes the strings come from an underlying “grammar”
  – In a compiler

• Mix fuzzing and enumeration
  – Enumerate strings from the grammar, with “holes”
  – Fill holes with fuzz
  – Command injection: “ls *”, “SELECT FROM”

• Useful in compiler testing
Web App Heuristics: URLs

• Directory traversal
• Fuzz file extensions in particular: .htm, .html, etc.
• Long URLs
• Query strings, names and values
• Malformed URLs with unescaped characters
Open Source Fuzzers

- (L)ibrary (E)xploit API – lxapi – A collection of python scripts for fuzzing
  
- ... many more
  
- Original Fuzz tool by Bart Miller:
Comments About Fuzz Testing

• Differs from Targeted Testing
  – May well miss dangerous paths
  – Will take paths no logical person would ever consider
  – Tests vastly more conditions and paths than a human could
  – Like unit testing, usually starts from a clean slate. Thus misses multicondition bugs
  – A **supplement** to traditional boundary value analysis and other human driven testing;
    • not a replacement
Limitations of Fuzz Testing

• Access control
• Checksums and file verification
• Grammars
• Multistep bugs
Limitations of Fuzzing

- Probability of reaching an error can be astronomically low

```c
int x;
if (x == 94389) {
    ERROR;
}
```

Probability of hitting ERROR = $1/2^{32}$
Symbolic Execution

• Use symbolic values for input variables
• Execute the program symbolically on symbolic input values
• Collect symbolic path constraints
• Use constraint solver to generate test inputs for each execution path

• Lots of recent work on constraint solving
  – Well-engineered tools available (Z3, Yices)
  – Unfortunately, cannot cover in this class
Computation Tree: Execution Paths of a Program

- Can be seen as a **binary tree** with possibly infinite depth
  - Computation tree
- Each **node** represents the execution of a “if then else” statement
- Each **edge** represents the execution of a sequence of non-conditional statements
- Each path in the tree represents an equivalence class of inputs
void test_me(int x, int y) {
    if(2*x==y){
        if(x != y+10){
            printf("I am fine here");
        } else {
            printf("I should not reach here");
            ERROR;
        }
    } else {
    }
}
Active Checking for Errors

Divide by 0 Error

x = 3 / i;

Buffer Overflow

a[i] = 4;
Active Checking for Errors

Key: Add Checks Automatically and Try to Cover all Branches

Divide by 0 Error

\[
\text{if } (i \neq 0) \\
\quad x = 3 / i; \\
\text{else} \\
\quad \text{ERROR;}
\]

Buffer Overflow

\[
\text{if } (0 \leq i \land i < a.\text{length}) \\
\quad a[i] = 4; \\
\text{else} \\
\quad \text{ERROR;}
\]
Concrete Execution

```c
int x = read();
int y = read();
int t;

t = x;
x = x + y;
if (x > y) {
    y = 2*t;
    if (x+1 == y) {
        assert(false);
    }
}
```
Concrete Execution

```c
int x = read();
int y = read();
int t;

t = x;
x = x + y;
if (x > y) {
    y = 2*t;
    if (x+1 == y) {
        assert(false);
    }
}
```

```plaintext
x = 4
y = 9
t = 4
x = 13
(13 > 9)
y = 8
(14 == 8)
```
Symbolic Execution

```
int x = read();
int y = read();
int t;

t = x;
x = x + y;
if (x > y) {
    y = 2 * t;
    if (x + 1 == y) {
        assert(false);
    }
}
```
Symbolic Execution

```
int x = read();
int y = read();
int t;

t = x;
x = x + y;
if (x > y) {
    y = 2 * t;
    if (x+1 == y) {
        assert(false);
    }
}
```

Solve constraint:
Solution: $x = 0$, $y=0$

Solve constraint:
Solution: $x = 1$, $y=9$

Solve constraint:
Solution: $x = 2$, $y=1$
Concolic Approach

• Combine concrete and symbolic execution for unit testing
  – DART: Directed Automated Random Testing
  – Concrete + Symbolic = Concolic

• Why would you do this?
Concolic Approach

• Combine concrete and symbolic execution for unit testing

• Why would you do this?

• Concrete execution helps Symbolic execution to simplify complex and unmanageable symbolic expressions
  – By performing address resolution at run time
  – By replacing symbolic values by concrete values
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

- Random Test Driver:
  - random value for x and y
- Probability of reaching ERROR is extremely low
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution
- Concrete state: $x = 22, y = 7$
- Symbolic state: $x = x_0, y = y_0$
- Path condition:

```
x = 22, y = 7
```
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution
- **Concrete state:**
  - \( x = 22, y = 7, z = 14 \)

Symbolic Execution
- **Symbolic state:**
  - \( x = x_0, y = y_0, z = 2*y_0 \)

**Path condition:**
- \( x > y+10 \)
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution
- **Concrete state**
  - $x = 22$, $y = 7$, $z = 14$

Symbolic Execution
- **Symbolic state**
  - $x = x_0$, $y = y_0$, $z = 2y_0$
- **Path condition**
  - $2y_0 \neq x_0$
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

**Concrete Execution**

**Symbolic Execution**

- **Concrete State**: `x = 22, y = 7, z = 14`
- **Symbolic State**: `x = x₀, y = y₀, z = 2*y₀`
- **Path Condition**: `2*y₀ != x₀`
- **Solution**: `x₀ = 2, y₀ = 1`
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution
- **Concrete State**: \( x = 2, y = 1 \)
- **Symbolic State**: \( x = x_0, y = y_0 \)
- **Path Condition**

Symbolic Execution

---

133
Concolic Testing Approach

```cpp
int double (int v) {
    return 2*v;
}

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

Concrete Execution

**Concrete State**: $x = 2, y = 1, z = 2$

Symbolic Execution

**Symbolic State**: $x = x_0, y = y_0, z = 2y_0$

Path Condition

134
Concolic Testing Approach

```
int double (int v) {
    return 2*v;
}

void testme (int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution

Symbolic Execution

<table>
<thead>
<tr>
<th>concrete state</th>
<th>symbolic state</th>
<th>path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 2$, $y = 1$, $z = 2$</td>
<td>$x = x_0$, $y = y_0$, $z = 2y_0$</td>
<td>$2y_0 == x_0$</td>
</tr>
<tr>
<td>$x_0 &gt; y_0 + 10$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Concrete state:
- $x = 2$
- $y = 1$
- $z = 2$

Symbolic state:
- $x = x_0$
- $y = y_0$
- $z = 2y_0$

Path condition:
- $2y_0 == x_0$
- $x_0 > y_0 + 10$
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution

```
int x = 2, y = 1,
z = 2
```

Symbolic Execution

```
Solve: (2*y₀ == x₀) && (x₀ > y₀ + 10)
```

Solution: $x₀ = 30, y₀ = 15$

```
2*y₀ == x₀
x₀ > y₀ + 10
```

137
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution:
- `x = 30, y = 15`

Symbolic Execution:
- `x = x_0, y = y_0`

Path Condition:
- If `x > y + 10`
Concolic Testing Approach

```c
int double (int v) {
    return 2*v;
}

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

Concrete Execution

Symbolic Execution

concrete state

symbolic state

path condition

x = 30, y = 15

x = x₀, y = y₀ + 10

Program Error

x₀

x₀ >

2*ₚ₀ ==

ERROR
Traverse all Paths

• Traverse all execution paths one by one to detect errors
  – assertion violations
  – program crash
  – uncaught exceptions

• combine with valgrind to discover memory errors
Traverse all Paths

- Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

- combine with valgrind to discover memory errors
Traverse all Paths

• Traverse all execution paths one by one to detect errors
  - assertion violations
  - program crash
  - uncaught exceptions

• combine with valgrind to discover memory errors
Concolic Testing: A Middle Approach

Random Testing
- Complex programs
- Efficient
- Less coverage
- No false positive

Concolic Testing
- Complex programs
- +/- Somewhat efficient
- High coverage
- No false positive

Symbolic Testing
- Simple programs
- Not efficient
- High coverage
- False positive
Limitations: A Comparative View

Concolic: Broad, shallow

Random: Narrow, deep
Hybrid Concolic Testing

Interleave Random Testing and Concolic Testing to increase coverage

Key idea:
– Random testing leads to state s deep inside program
– Concolic testing explores path starting in s and gives good local coverage

Good for: Reactive programs (periodic input from environment)

Not good for: Transformational programs

Reference Paper: Majumdar, Sen, Hybrid Concolic Testing
http://srl.cs.berkeley.edu/~ksen/pubs/paper/hybrid.ps

Aside: Motivated by similar idea used in VLSI design validation:
Ganai et al. 1999, Ho et al. 2000
Hybrid Concolic Testing

- Interleave Random Testing and Concolic Testing to increase coverage

while (not required coverage) {
  while (not saturation) {
    perform random testing;
    Checkpoint;
  }
  while (not increase in coverage) {
    perform concolic testing;
    Restore;
  }
}

Deep, broad, hybrid Search
Further reading


See also: Zoltán MICSKEI’s page with a recent overview on tools:
http://home.mit.bme.hu/~micskeiz/pages/code_based_test_generation.html
Acknowledgements

• Some of the slides were adapted from the lecture materials by Rupak Majumdar