

COMPGS03/COMP4023 Validation & Verification Test Cases

Outline

- Finding test cases.
- Readings
 - Myers chapter 4
 - Ostrand & Balcer Category-Partition Method
 - various other papers to give a more detailed perspective of testing research.

Exhaustive Testing?

- String triangle(int a, int b, int c)
 - 32 bit integers range -2^{31} to $2^{31}-1$
 - $2^{32} \times 2^{32} \times 2^{32}$ possible combinations
 - Not feasible to test them all
 - We want tests to run quickly (ideally a few seconds at most)
 - If method has parameters of other types (e.g., double, String), problem just gets worse.
- Hence, have to be very selective
 - Can only afford to focus on tests that have high probability of finding errors.
 - Exhaustive testing is not feasible in general.

Doubles...

- String triangle(double a, double b, double c)
 - Range of double is much larger.
 - A double value is only an approximation due to the way floating point numbers are represented.
 - Not all values can actually be represented.

Selecting Tests

- Myers list:
 - Equivalence Testing
 - Boundary-value analysis
 - Cause-effect graphing
 - Error guessing
 - Statement coverage
 - Decision coverage
 - Condition coverage
 - Decision-condition coverage
 - Multiple-condition coverage



Myers list these under Black Box testing but they work effectively with unit testing, as they fit in with the test design process.

Read Myers chapter 4 for detailed explanations/examples of these.

Equivalence Classes

- Tests divided into classes depending on input/initial state.
- Values in each class assumed to be result of equivalent computation.
- Allows a small number of representative values to be used for testing, to cover the entire equivalence class .
- Dramatically reduces number of test cases (based on input values) needed.

Example

String toString(int month, int day)

- Classes for valid days in months with 28, 29, 30, 31 days.
- Class for valid months (1-12).
- Classes for invalid months and days.
- Boundary tests:
 - {28 day month}{0, 1, 28, 29}
 - {29 day month}{0, 1, 28, 29, 30}
 - {30 day month}{0, 1, 30, 31}
 - {31 day month}{0, 1, 30, 31, 32}
 - Invalid months/days {-1, -1}{0, 0}{13, 32}{100, 100}, etc.

Equivalence Classes - Sqrt

- Consider writing a square root function

sqrt(double)

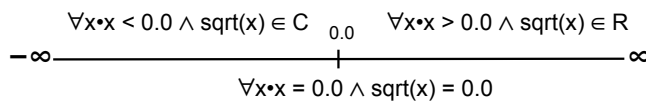
- Can initially partition input into:

$x < 0.0$

$x = 0.0$

$x > 0.0$

Also identifying boundaries and boundary values, e.g., -1.0, -0.000001, 0.0, 0.000001, 0.1, 1.0, 2.0



sqrt (2)

- Initial selection of test cases requires specification to be refined.
- What should sqrt applied to a negative number do?
 - Mathematically well defined.
 - But do we want to write a function that can return a double or a complex number?
 - In fact, can't do this in Java unless we have a more abstract type, such as Number
 - Standard Java doesn't have a Complex class or a rich enough number class hierarchy.
- So, sqrt of negative number is treated as an error or NaN.

sqrt(3) Boundary Values

- Does `sqrt(0.0000000001) == 0.00001`?
 - Precision implies there is another equivalence class covering number numbers outside the precision range.
- Or 0.0? Or something else?
- Depends on accuracy of floating point representation.
- How many decimal places of accuracy should we work with?
- What are the max/min boundaries?
 - Very small
 - Very large
 - $(1.123E70 + 1.0 = ?)$

Error Guessing?

- What other input do we test sqrt with?
 - 1.0, 10.0, 100.0, 1000.0, ...
 - 1.2345, 10.2345, 100.2345, ...
 - Why are these any better than any other?
 - Without knowledge about the method implementation, can only guess + look at (suspected) boundaries.

Example sqrt Method

Newton Raphson approximation

```
public static double sqrt(final double x) {
    final double precision = 0.0000001;
    if (x < 0.0) {
        return Double.NaN;
    }
    if (x <= precision) {
        return 0.0;
    }
    double a = 1.0;
    while (Math.abs(a * a - x) > precision) {
        a = (a + x / a) / 2;
    }
    return a;
}
```

Obvious boundary conditions

Potential overflow

Lots of other things to check

- Loop boundaries and correct number of iterations.
- Array/collection indexing.
- null pointers and initialisation.
- Underflow/overflow.
- Ranges and sub-ranges.
- Size and length.
- Side effects, I/O, non-local variables.
- Parameter values and ranges.
- In fact, just about everything!

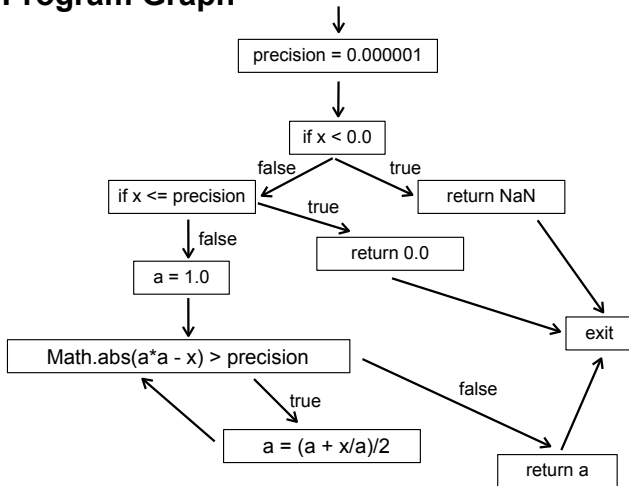
Coverage

- Want enough tests to ensure that all code is “covered” by a test.
 - Lines of code.
 - Branches, loops.
 - All paths of execution.
- Code coverage tools measure how effectively tests achieve coverage.
 - e.g., Clover

Paths

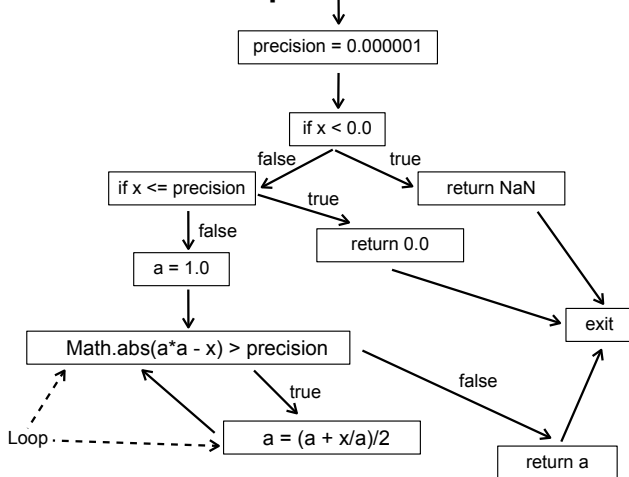
- Can represent code as a graph
 - node -- statement
 - edge -- transition from one statement to another
 - fixed or determined by decision
- Can identify
 - set of paths to test
 - if path is feasible (or find dead code)
 - loops increase number of paths (can be infinite...)

Program Graph



© 2007, Graham Roberts

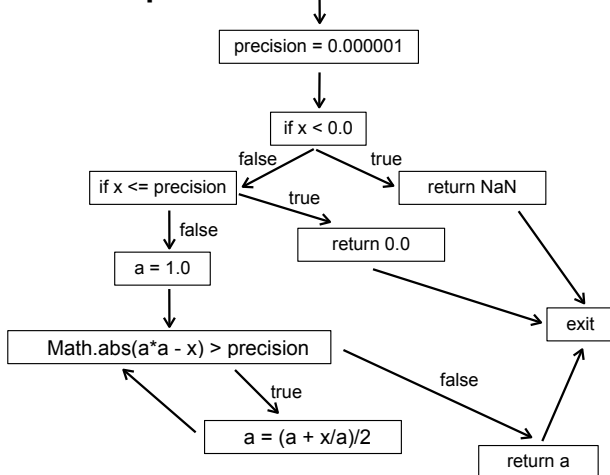
16

Path for $n < 0.0$ equivalence class

© 2007, Graham Roberts

17

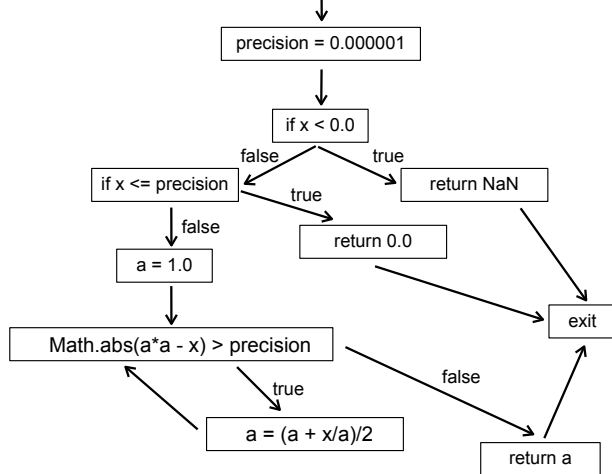
n is below precision threshold



© 2007, Graham Roberts

18

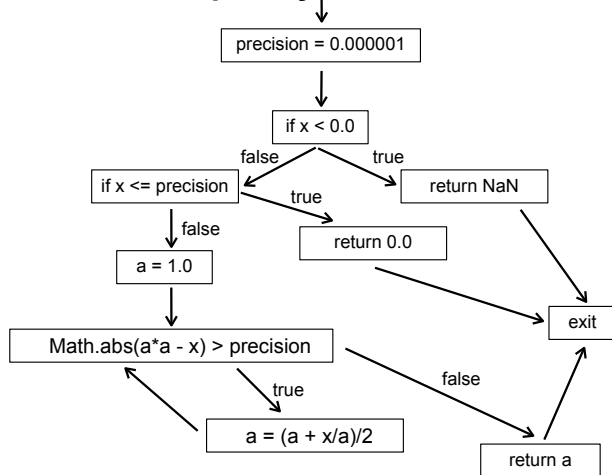
Path where loop body is executed



© 2007, Graham Roberts

19

Path where loop body is not executed



© 2007, Graham Roberts

20

Triangle (familiar!)

```

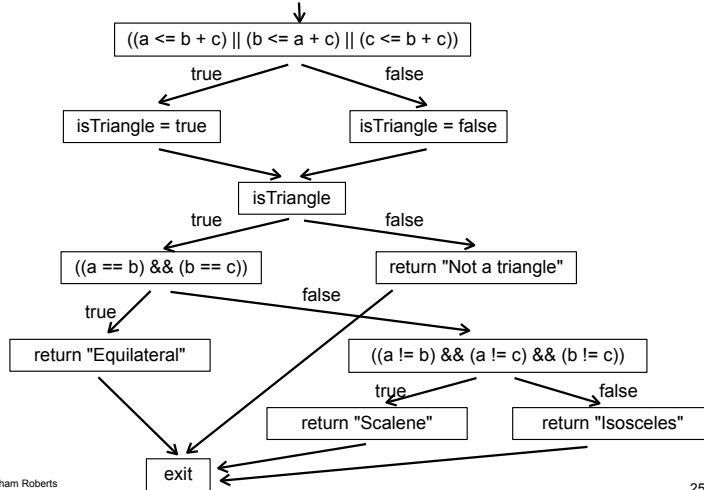
String triangle(double a, double b, double c) {
    boolean isTriangle = false;
    if ((a <= b + c) || (b <= a + c) || (c <= b + c)) { isTriangle = true;}
    if (isTriangle) {
        if ((a == b) && (b == c)) { return "Equilateral";}
        else
            if ((a != b) && (a != c) && (b != c)) { return "Scalene"; }
        else { return "Isosceles"; }
    }
    else { return "Not a Triangle"; }
}
  
```

Spot the errors...

© 2007, Graham Roberts

21

Program Graph - not all paths are possible



Statement Coverage

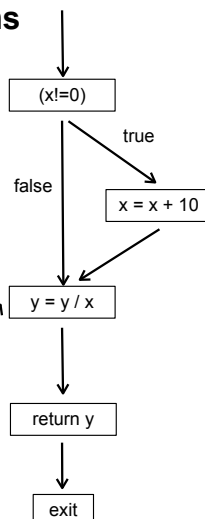
- Enough tests to ensure each statement or line of code is run at least once.
 - Also referred to as basic code coverage.
- Test cases must cover all if statement branches, loop bodies, etc.
- “Dead code” is unreachable and can be removed.
 - Some language compilers can detect unreachable code.

Statement coverage limitations

```

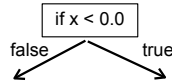
int f(int x, int y) {
  if (x != 0) { x = x + 10; }
  y = y / x;
  return y;
}
  
```

Test case $[x = 1, y = 2]$ ($x \neq 0$, any y) achieves coverage. But fails to find error when $x = 0$. Only nodes covered.



Decision Coverage

- Enough tests such that each decision evaluates to true and false at least once.
 - Superset of statement coverage.
- Each edge executed.

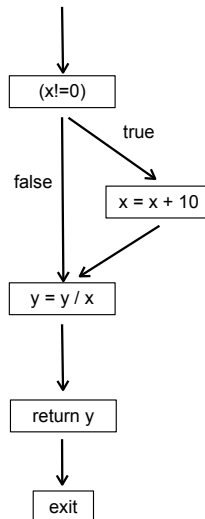


Decision coverage

```

int f(int x, int y) {
  if (x != 0) { x = x + 10; }
  y = y / x;
  return y;
}
  
```

Want to check $(x \neq 0, \text{ any } y)$ and $(x = 0, \text{ any } y)$ to cover nodes and paths.
Use test cases $[x=1, y=2]$ and $[x=0, y=2]$

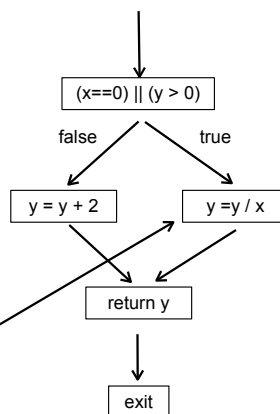


But...

```

int f(int x, int y) {
  if ((x == 0) || (y > 0)) { y = y / x; }
  else { y = y + 2; }
  return y;
}
  
```

Try $[x=5, y=5]$, $[x=5, y=-5]$.
Achieves decision coverage but fails to detect error if $x = 0$.



Condition Coverage

- Enough tests such that each condition in a decision evaluates to true and false at least once, each decision evaluates to true and false at least once.
- Each edge executed *and* each atomic decision is evaluated to true and false.
 - $(a < b) \ \&\& \ (c > d)$
 - sub-expressions $a < b$ and $c > d$ must each evaluate to true and false.

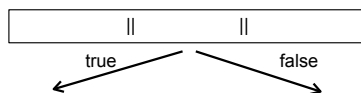
Draw-up a table

$(x == 0) \ || \ (y < 0)$

Test Case	$x == 0$	$y < 0$	$(x == 0) \ \ (y < 0)$
$x=5, y=5$	F	T	T
$x=0, y=5$	T	T	T
$x=0, y=-1$	T	F	T
$x=5, y=-1$	F	F	F

Triangle Condition Coverage

- A lot more complex than sqrt



- Unit testing aims for simple methods
 - Split up triangle into smaller methods.
 - Reduce number of test cases for each part.

Exercise

```
int f(int x, int y) {
  while ((x > 0) || (y > 0)) {
    if (x >= y) { x = x - 1;}
    else { y = y - 1;}
  }
  return x + y;
}
```

Identify a set of test cases to achieve:

- Statement coverage
- Decision coverage
- Condition coverage

Mutation Testing

- A mutation is a small change made in program code.
- Mutation testing uses mutations to measure the effectiveness of a test suite in finding defects.
 - The *adequacy* of a test suite.
 - Systematically apply mutations to a program to create a sequence of mutants.
 - Each mutant is the result of a single change.
 - Run the test suite on each mutant and see if mutation is detected by a test.
 - Can be fully automated.
- Tools available for range of languages, notably Java.
 - *μJava* (muJava), www.ise.gmu.edu
 - Jester (JUnit test tester), jester.sourceforge.net

Mutation Testing (2)

- Concerned with the Residual Defect Density (RDD)
 - A measure of the defects remaining in code.
 - % of defects per thousand lines.
- rk/n gives estimate of RDD
 - r = estimate of defect rate based on experience of previous development.
 - k = number of mutants that did not result in errors being detected.
 - n = total number of mutants tested.
- k/n gives adequacy of test in finding defects.

Mutation Testing (3)

- Value mutations
 - literal values, loop bounds, parameters
 - e.g., add/subtract one
- Decision mutations
 - Modify boolean expressions
 - e.g., $x < y \Rightarrow x > y$
- Statement mutations
 - delete line of code
 - reorder lines

```
int n = 0;
for (int i = 0 ; i < a.length && a[i] > 5 ; i++) {
  if (a[i] == -1) n++ ;
  if (n > 10) break ;
}
```

Many mutations possible even in a small section of code.

Summary

- Looked at ways of identifying potential tests.
- Should read Myers chapter 4 for detailed examples.
- All useful techniques/ideas for unit testing.
- Keep code simple to facilitate testing!