Structural Testing 1

Conrad Hughes
School of Informatics

Slides thanks to Stuart Anderson

informatics

Summary

When we write unit tests we consider:

1. Specification-based tests using specifications or models
2. Checklists of commonly occurring errors
3. Structural Testing

Common Errors

- Can be from a particular programming community.
- Well-instrumented organisations monitor and summarise error occurrences.
- Professional good practice should make you sensitive to the errors you make personally.
- The following are the "top three" from David Reilly's top ten Java programming errors (linked from the practical).
- Use this as a checklist when you are looking to test systems – attempt to provoke errors in these classes. (e.g. number 4 in the "top ten" is that Java's arrays start at 0!)

3. Concurrent access to shared variables by threads

```
public class MyCounter {
  private int count = 0; // count starts at zero

  public void incCount(int amount) {
    count = count + amount;
  }

  public int getCount() {
    return count;
  }
}
…
               MyCounter c;
// Thread 1                    // Thread 2
c.incCount(1);                 c.incCount(1);
               // join
               c.getCount() == ?
```


3. Concurrent access to shared variables by threads

```
public class MyCounter {
  private int count = 0; // count starts at zero

  public synchronized void incCount(int amount) {
    count = count + amount;
  }

  public int getCount() {
    return count;
  }
}
```

Even more important with shared **external** resources…

2. Capitalization Errors

- Remember:
  - All methods and member variables in the Java API begin with lowercase letters.
  - All methods and member variables use capitalization where a new word begins e.g - getDoubleValue().

1. Null pointers

```
public static void main(String args[]) {
  String[] list = new String[3]; // Accept up to 3 parameters
  int index = 0;

  while( (index < args.length) && (index < 3) ) {
    list[index] = args[index];
    index++;
  }

  // Check all the parameters
  for(int i = 0; i < list.length; i++) {
    if(list[i].equals("-help")) {
      // .........
    } else if(list[i].equals("-cp")) {
      // .........
    }
    // [else .....]
  }
}
```


Structural Testing

- Testing that is based on the structure of the program.
- Usually better for finding defects than for exploring the behaviour of the system.
- Fundamental idea is that of "basic block" and flow graph – most work is defined in those terms.
- Two main approaches:
  - Control oriented: how much of the control aspect of the code has been explored?
  - Data oriented: how much of the definition/use relationship between data elements has been explored.
- See figures 12.1 and 12.2 of Pezzè and Young for an example of some code and its corresponding control flow graph.
- The code has null pointer errors.

Basic Blocks

- A basic block has at most one entry point and usually at most two exit points.
  - Can you think of exceptions to this?
- We decompose our program into basic blocks. These are the nodes of the control graph.
- The edges of the control graph indicate control flow – possibly under some conditions.

Code and Control Flow Graph Example

```
-17: int cgi_decode(char *encoded, char *decoded) {
-18:   char *eptr = encoded;
-19:   char *dptr = decoded;
*20:   int ok=0;
*21:   while (*eptr) {
-22:     char c;
*23:     c = *eptr;
-24:     /* Case 1: '+' maps to blank */
*25:     if (c == '+') {
*26:       *dptr = ' ';
*27:     } else if (c == '%') {
-28:       /* Case 2: '%xx' is hex for character xx */
-29:
*30:       int digit_high = Hex_Values[*(++eptr)];
*31:       int digit_low = Hex_Values[*(++eptr)];
*32:       if ( digit_high == -1 || digit_low == -1 ) {
-33:         /* *dptr='?'; */
*34:         ok=1; /* Bad return code */
-35:       } else {
*36:         *dptr = 16* digit_high + digit_low;
-37:       }
-38:
-39:       /* Case 3: All other chars map to themselves */
*40:     } else {
*41:       *dptr = *eptr;
-42:     }
*43:     ++dptr;
*44:     ++eptr;
-45:   }
*46:   *dptr = '\0'; /* Null terminator for string */
*47:   return ok;
-48: }
```

P&Y p.213-214, Figures 12.1 & 12.2

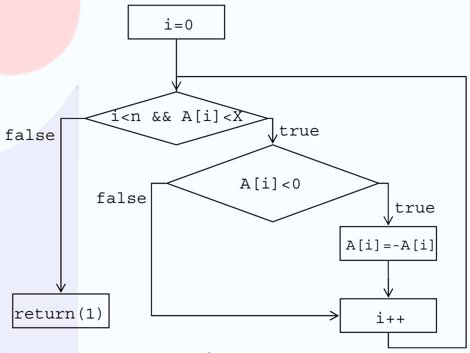
Some tests for the cgi program

- T_0 = { "", "test", "test+case%1Dadequacy"}
  - -> "", "test", "test case□adequacy"
- T_1 = {"adequate+test%0Dexecution%7U"}
  - -> "adequate test<CR>execution□"
- T_2 = {"%3D", "%A", "a+b", "test"}
  - -> "=", ?, "a b", "test"
- T_3 = { " ", "+%0D+%4J"}
  - -> " ", "<CR> □"
- T_4 = {"first+test%9Ktest%K9"}
  - -> "first test□test□"

Statement Testing

- ***Statement Adequacy***: all statements have been executed by at least one test.
- ***Statement Coverage***: for a particular test T, this is the quotient of the number of statements executed during a run of T (not counting repeats) and the number of statements in the program.
- The test set T is adequate if the Statement Coverage is 1.
- For our sample tests: T0 omits ok = 1 at line 34, T1 executes all the code as does T2.
- In general we do not know if statement coverage is achievable – why?
- All of this can be rephrased in terms of basic blocks – and we look at node coverage in the control-flow graph.
- Statement coverage is a basic measure but is a fairly poor test of how well we have exercised the code.

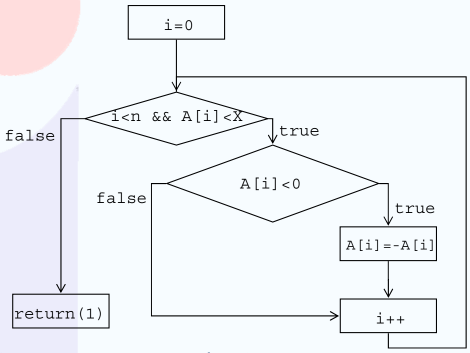
Statement Coverage - Example



Branch Coverage

- Statement Coverage gives fairly poor coverage of the flow of control in systems.
- For example, we can only guarantee to consider arriving at some basic block from one of its predecessors.
- **Branch adequacy** attempts to resolve that:
  - Let T be a test suite for a program P. T satisfies *the branch adequacy criterion* if for each branch B of P there exists at least one test case that exercises B.
- The *branch coverage* for a test suite is the ratio of branches tested by the suite and the number of branches in the program under test.
- As usual it is undecidable whether there exists a test suite satisfying the branch adequacy criterion.

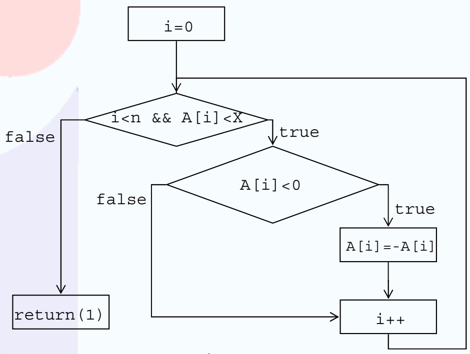
Branch Coverage – Example



Condition Coverage

- There are issues concerning the adequacy of branch coverage in environments where we allow compound conditions (because we might take a particular branch for different reasons).
- This is exacerbated when we have "shortcut conditions" that do not evaluate some of the condition code.
- We frame this in terms of "basic conditions" i.e. comparisons, basic properties etc.
- The *basic condition adequacy criterion* is:
  - Let T be a test suite for program P. T covers all the basic conditions of P iff each basic condition of P evaluates to *true* under some test in T and evaluates to *false* under some test in T.
- Possible to extend to a "compound" condition adequacy where all boolean subformulae in conditions evaluate to both true and false.

Condition Coverage – Example



Compound Condition Coverage

a && b && c && d && e

Test Case	a	b	c	d	e
(1)	True	True	True	True	True
(2)	True	True	True	True	False
(3)	True	True	True	False	–
(4)	True	True	False	–	–
(5)	True	False	–	–	–
(6)	False	–	–	–	–

(((a || b) && c) || d) && e

Test Case	a	b	c	d	e
(1)	True	–	True	–	True
(2)	False	True	True	–	True
(3)	True	–	False	True	True
(4)	False	True	False	True	True
(5)	False	False	–	True	True
(6)	True	–	True	–	False
(7)	False	True	True	–	False
(8)	True	–	False	True	False
(9)	False	True	False	True	False
(10)	False	False	–	True	False
(11)	True	–	False	False	–
(12)	False	True	False	False	–
(13)	False	False	–	False	–

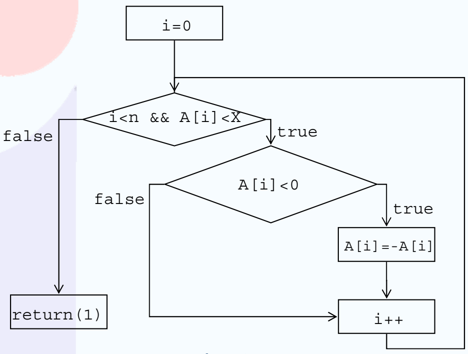
P&Y p.221

Finally, MC/DC:
Modified Condition/Decision Coverage,
aka Modified Condition Adequacy Criterion:
- Satisfiable with N + 1 test cases (N variables).
- Good compromise, required in aviation quality standards.

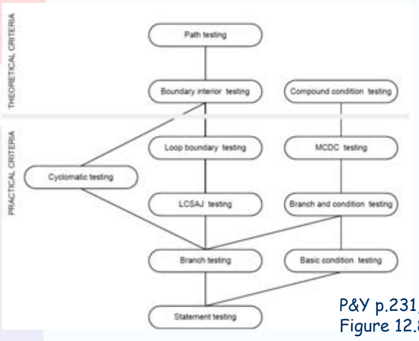
Path Coverage

- Condition coverage still gives us a poor coverage of historical executions of the system.
- Path coverage is better:
  - Let T be a test suite for program P. T satisfies the *path adequacy criterion* for P iff for each path p of P there exists at least one testcase in T that causes the execution of p.
- Infeasible for all but trivial programs.
- Coverage notion is the ratio of covered paths to total number of paths – tends to zero for programs with unbounded loops.
  - Why?
- Approach is to consider "unrolling" the code finitely
- Loop boundary coverage, each loop is executed:
  - Zero times
  - Once
  - More than once

Path Coverage – Example



Summary – Subsumption Relations



P&Y p.231, Figure 12.8