Structural Testing

Stuart Anderson

Types of Testing

When we write unit tests we consider:

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

- These are two different kinds of test: where we consider details of the implementation (as in 2 and 3) – known as **white box testing** – and where we work from external descriptions, treating the implementation as an opaque artefact with inputs and outputs: **black box testing** (as in 1).

- We also distinguish between tests which involve executing the code (**dynamic tests**, which we have mainly been looking at) and those which do not: **static tests** (code review, for example).

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
  - **Concurrent access to shared variables by threads** (3)
  - **Capitalization errors** (2)
  - **Null pointers** (1)

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() == ?
```


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;
  }
}
```

Synchronization... Even more important with shared **external** resources...

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().

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.

Basic Blocks

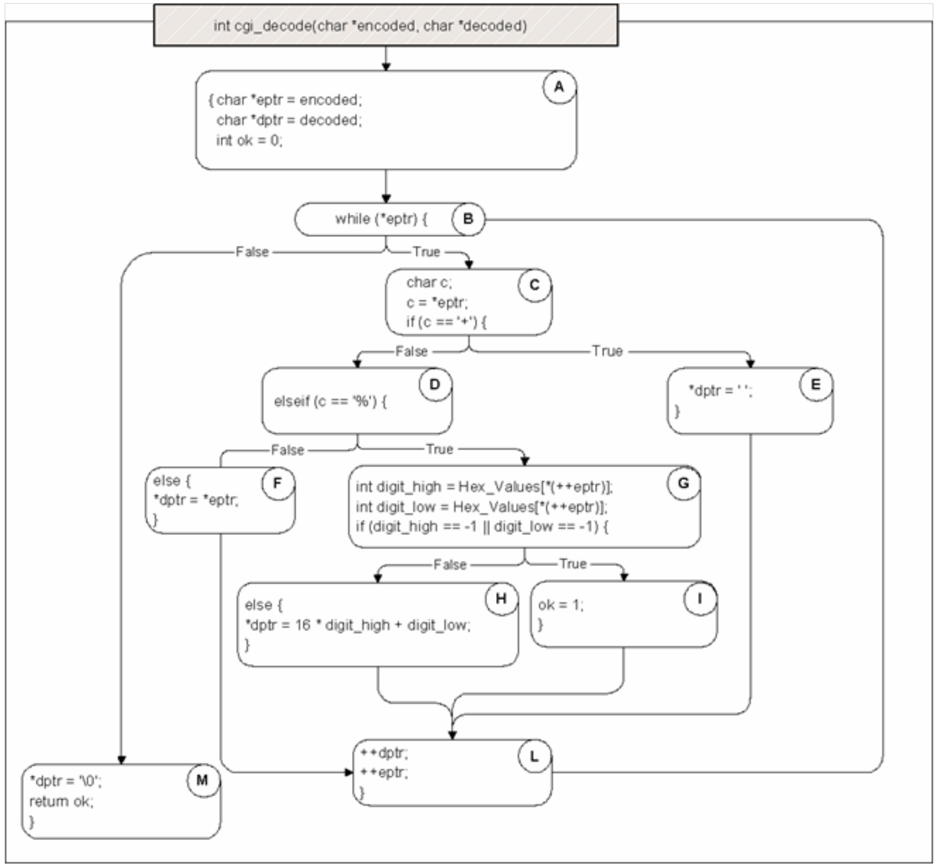
- A basic block has at most one entry point and usually at most two exit points.

  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



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

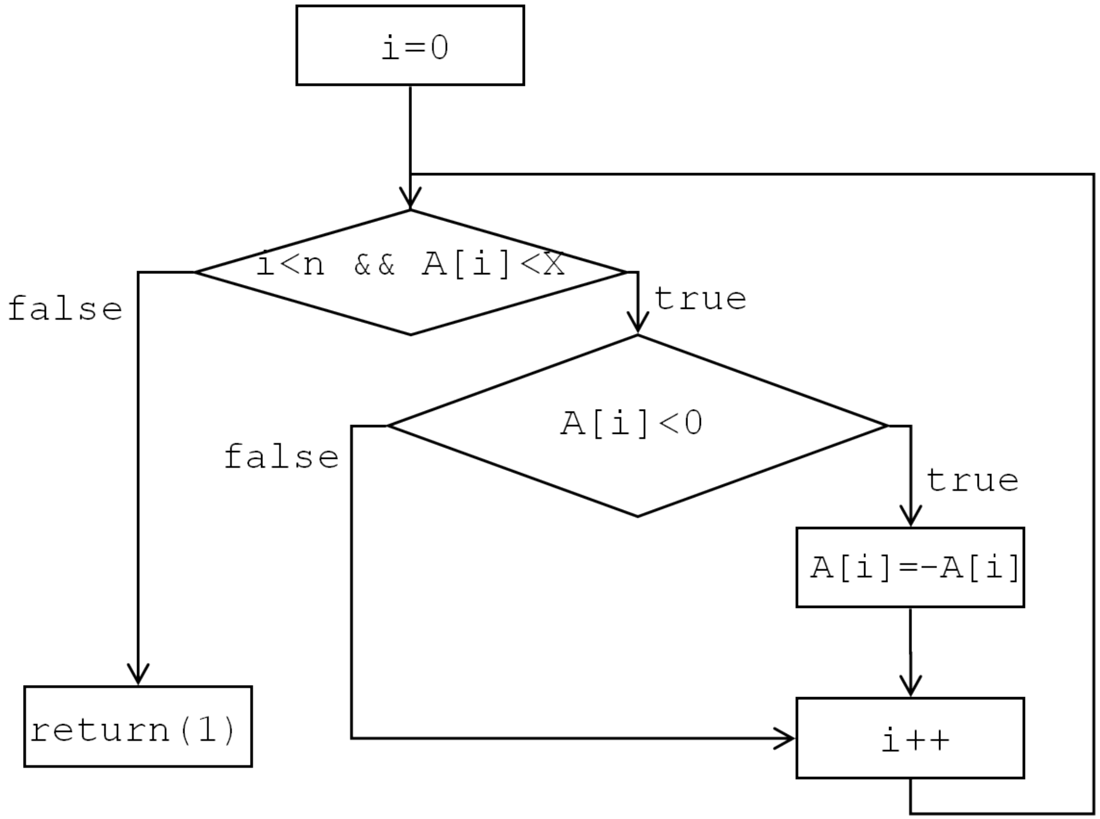
Some tests for the cgi program

- T_0 = { " ", "test", "test+case%1Dadequacy" }
  → " ", "test", "testcase□adequacy"
- T_1 = { "adequate+test%0Dexecution%7U" }
  → "adequate test<CR>execution□"
- $T2$ = { "%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: T_0 omits ok $=$ 1 at line 34, T_1 executes all the code as does T_2.
- 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.

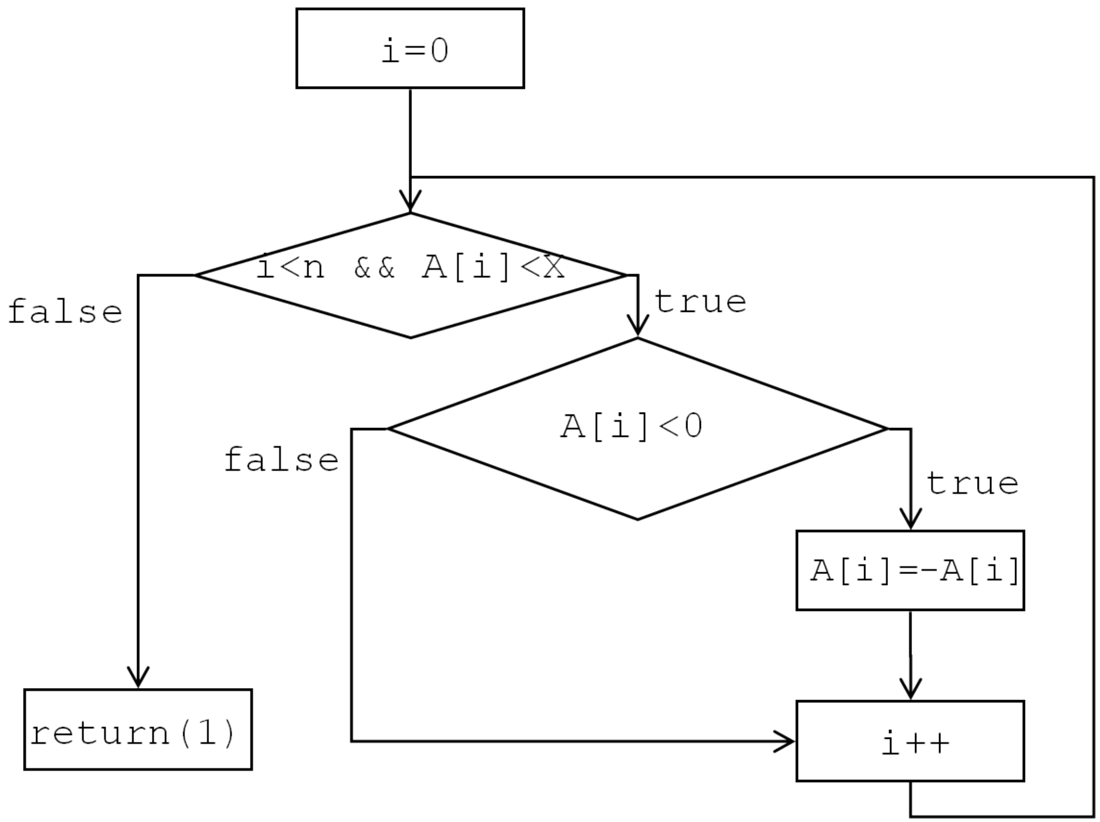
Statement Coverage



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.

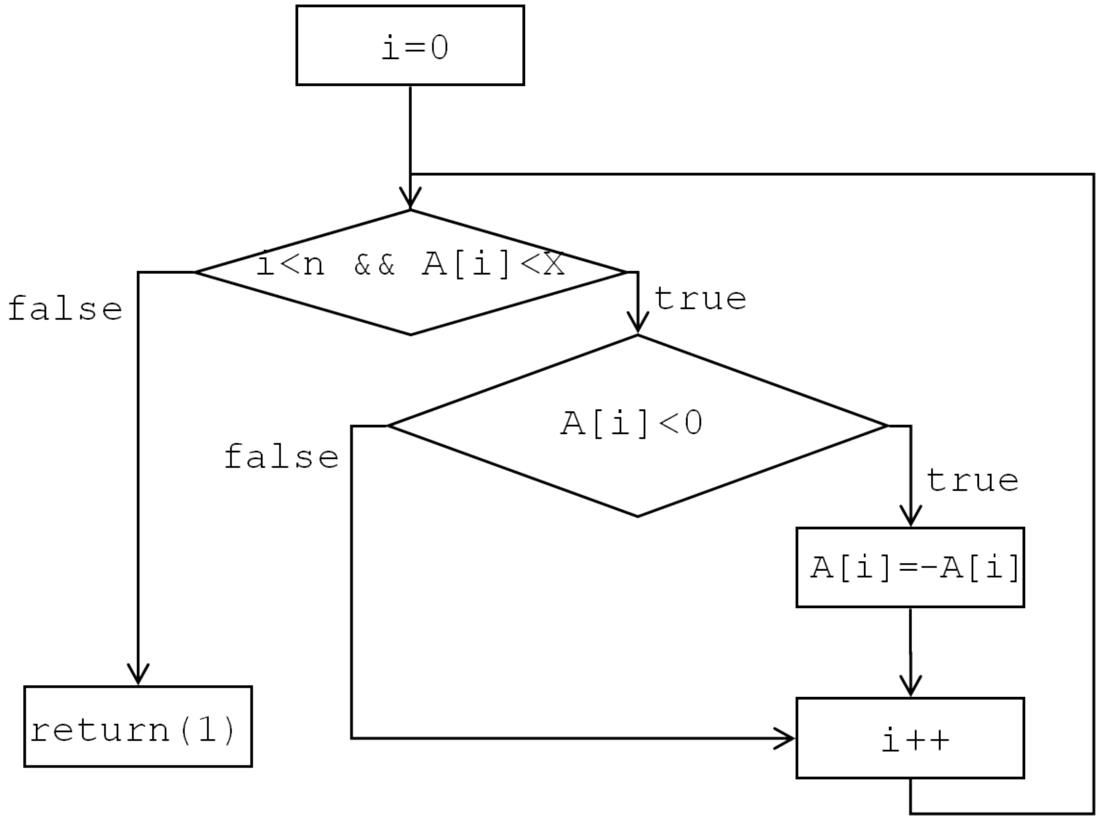
Branch Coverage



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



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 [P&Y p.221]

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**	–

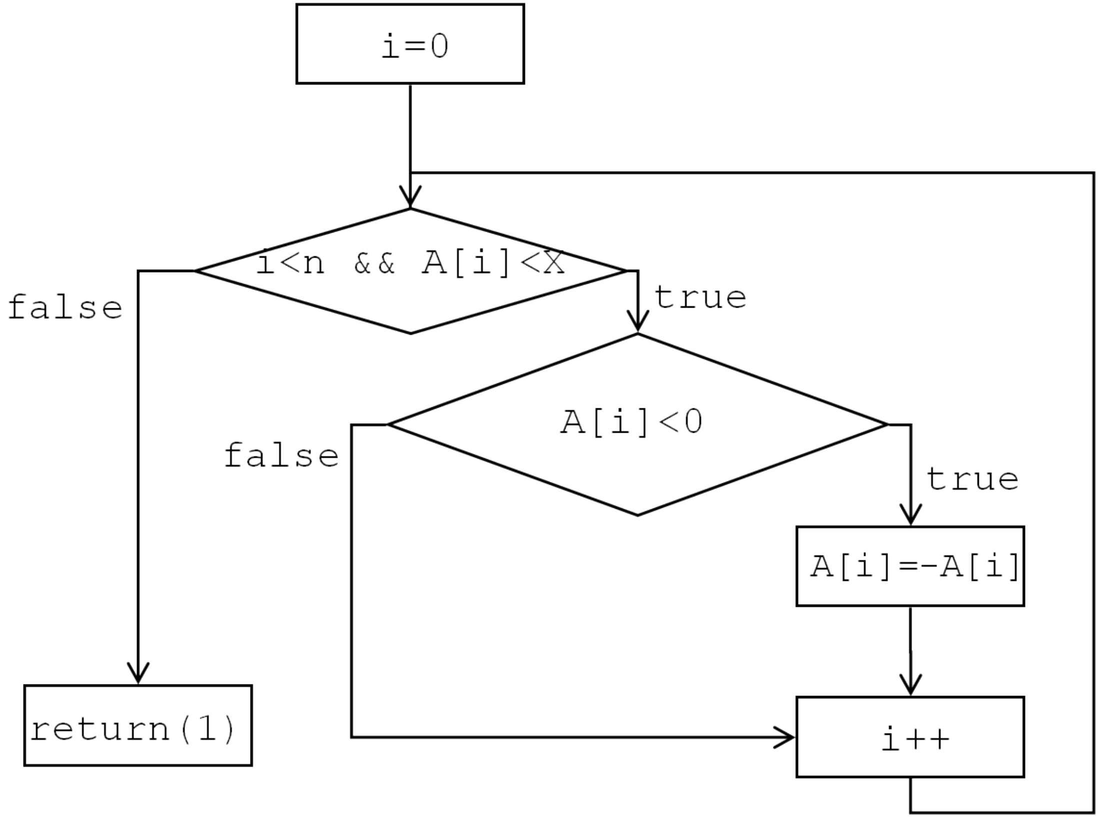
Finally, Modified Condition(MC)/Decision Coverage(DC), aka **Modified Condition Adequacy Criterion**:

- Satisfiable with N + 1 test cases (N variables).
- Good compromise, required in aviation quality standards.

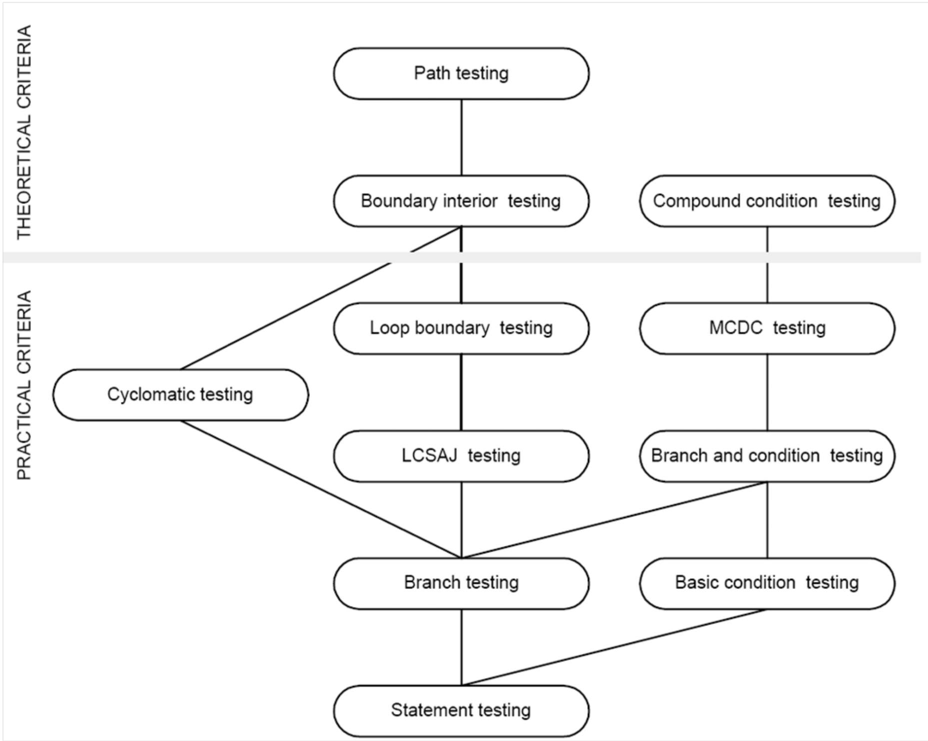
Path Coverage

- 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



Subsumption Relations



[P&Y p.231, Figure 12.8]

Readings

Required Readings

- **Textbook (Pezzè and Young):** Chapter 12, Structural Testing

Suggested Readings

- Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29, 4 (December 1997), 366-427.

  http://dx.doi.org/10.1145/267580.267590

