Software Verification and Validation

White-box testing. Test coverage

11 October 2017

White-box Testing

Tests are generated based on *internal structure* of code
Other (better) names: glass box, clear box, open box

Another classification:
  behavioral testing (black-box) / *structural* (white-box)

Comparison:
– black-box: at any level / white-box: mostly module/unit testing
– white-box: code change ⇒ tests change
– white-box: easier detection of *coding errors*,
  but cannot detect *omission errors* (in code or spec)

Program structure: Terminology
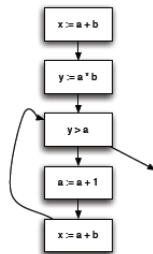
Control flow graph (CFG)
graph representation of program
and implicitly its execution paths

nodes = instructions
edges (labeled w. conditions):
  sequencing between instructions

```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```


Usually, straight-line code is grouped together ⇒

basic block =
  a sequence of statements with just one entry and one exit point
  (no jumps into middle of code, or from code outside)

Image: https://vinaytech.wordpress.com/2008/10/04/abstract-syntax-tree/

Code coverage

= a criterion to measure if a set of tests is *adequate*

What good are such criteria ? For questions as:

What program properties should we examine ?

What test data do we select for such properties ?

What *quantitative* objectives do we set for testing ?

Did we test enough ?

Burnstein, Practical Software Testing

The impossible ideal: test all program executions

$\Rightarrow$ i.e., *all paths* through the CFG

But: number of program paths usually infinite (loops, recursion)
  also: one path, multiple data (proper equivalence classes?)

$\Rightarrow$ must choose modest structural criteria

$\Rightarrow$ but not arbitrary – chosen *judiciously*

Testing axioms (Weyuker)

Antiextensionality:

There are equivalent programs P și Q such that a test suite T is adequate for P but not for Q.

i.e. *semantically* equivalent programs may need different test suites

General Multiple Change:

There are programs P and Q that have the same form (structure) and a test suite T which is adequate for P but not for Q.

i.e. *syntactically* close programs may need different test suites

Criteria: Line coverage

also: statement coverage, basic block coverage

Sufficient tests to execute each program statement
Obviously a necessary criterion (not executed = not tested)
obviously also insufficient

```c
char a[5], *s = NULL;
if (len < 5)
  s = a;
*s = 't';
```

Test with `len = 4` covers all statements; misses error

Branch coverage

also: decision coverage

Tests every possible value of a decision (true/false)
more precise definition: also tests every entry and exit from program
  usually implies statement coverage
  (every instruction is on some branch; see exception below)

The following are also decisions / branches
  `switch/case` statements (multiple branches)
  exception handling (hard to test, often neglected)
    every *potential* exception is a branching point
    code that looks straight-line in reality isn't

Caution: functions or side-effects in decisions:
`if (a && (b || f(x, y))`
  does not call f if a and b both true
  $\Rightarrow$ a case where branch coverage does not subsume line coverage

Condition coverage

A *condition* is an *elementary boolean expression* in a *decision*
  needs tests for each possible value of a condition
  apparently more complex than decision coverage, but does not
subsume it

Example
```
if (x > 5 && y == 3) /*some code */
```
Two tests: x = 6, y = 2   and   x = 4, y = 3
generate all possible condition values (T and F, F and T)
but follow the same branch (`false`)

Condition/decision coverage

Simultaneously covers *both* criteria

May need more tests than individual methods or just recombining them

Example

```
if (x > 5 && y == 3) /*some code */
```

two tests are still enough: x = 6, y = 3, and x = 4, y = 2

May be insufficient: the effect of some conditions may *mask* others

Multiple condition coverage

Tests all combinations for the *subexpressions* (conditions) of the decision

Exponential in number of conditions (2^n tests for n conditions)
   $\Rightarrow$ often too expensive to implement

In pratice, some of the 2^n combinations
   – may be irrelevant (for short-circuit evaluation)
   – may be infeasible (when conditions are not independent)
$\Rightarrow$ in general, this requirement is not justified

Modified Condition/Decision Coverage

One of the strongest criteria; initially developed at Boeing
is a requirement in avionics/safety-critical systems (standard DO-178B)

Complete requirements for an MC/DC test suite:
All program entry and exit points covered
Each decision exercised on both branches
Each condition takes both values
Each condition is shown to affect its enclosing decision
(keep other conditions fixed, varying condition of interest)

Same tests, whether language has short-circuit evaluation or not.

Constructing an MC/DC test suite

Start from base cases `&&` and `||` with two conditions
AND operator `&&` has a single case (`t t`) with result `t`.
Changing any condition to `f`, result becomes `f`.
Likewise for `||` (dual operator), switching `t` and `f`.

a	b	a && b	
f	t	f	(1)
t	f	f	(2)
t	t	t	(3)

a: (1, 3)
b: (2, 3)

a	b	a || b	
t	f	t	(1)
f	t	t	(2)
f	f	f	(3)

We indicate the pair of tests relevant for each condition:
(1, 3) shows a may influence decision; likewise, (2, 3) for b.

For *n* conditions: a test with all the same, *n* tests with one each flipped

a	b	c	a && b && c	
f	t	t	f	(1)
t	f	t	f	(2)
t	t	f	f	(3)
t	t	t	t	(4)

a: (1, 4)
b: (2, 4)
c: (3, 4)

MC/DC Construction Example

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

Start from innermost expression(s), `d && e` (watch precedence!)

d	e	d && e	
f	t	f	(1)
t	f	f	(2)
t	t	t	(3)

d: (1, 3)
e: (2, 3)

We then add `c ||`.

Since `||` with `f` does not change truth, add `c=f` to all tests (1-3).
For the new test (4), choose test with `f` result (2) and add `c=t`.

c	d	e	c || d && e	
f	f	t	f	(1)
f	t	f	f	(2)
f	t	t	t	(3)
t	t	f	t	(4)

Now also shows effect of c:

c: (2, 4)
d: (1, 3)
e: (2, 3)

MC/DC example (cont.)

Now add a `&& b && `. To previous tests, add a=t, b=t.
Then choose a test with t result (4), flip in turm a and b to f,
showing a and b influence decision:

a	b	c	d	e	a && b && (c || d && e)	
t	t	f	f	t	f	(1)
t	t	f	t	f	f	(2)
t	t	f	t	t	t	(3)
t	t	t	t	f	t	(4)
f	t	t	t	f	f	(5)
t	f	t	t	f	f	(6)

a: (4, 5)
b: (4, 6)
c: (2, 4)
d: (1, 3)
e: (2, 3)

Each test pair has one condition shown to influence outcome,
all other conditions have the same value in both tests.

By construction, it follows that *n* variables need $n + 1$ tests.

MC/DC coverage: example 2

Consider `a && b || c && d` .
We write tests for both subexpressions (given by precedence)

a	b	a && b			a: (1', 3')		c	d	c && d	
f	t	f	(1')		b: (2', 3')		f	t	f	(1")
t	f	f	(2')		c: (1", 3')		t	f	f	(2")
t	t	t	(3')		d: (2", 3")		t	t	t	(3")

We combine with `||`. Since `||` with `f` has no effect, choose one `f` test
from each group (1' + 1") and combine with all tests in the other group.

a	b	c	d	a && b || c && d		
f	t	f	t	f	(1=1'+1")	a: (1, 5)
f	t	t	f	f	(2=1'+2")	b: (4, 5)
f	t	t	t	t	(3=1'+3")	c: (1, 3)
t	f	f	t	f	(4=2'+1")	d: (2, 3)
t	t	f	t	t	(5=3'+1")	

We have thus kept the influence of each individual condition.

MC/DC in real code

The above analysis is valid for *independent conditions*
  it's always possible to generate the designed tests
In reality, conditions may be *coupled* (correlated)

Example: (z - x >= 3 && z - y >= 1 || y < 5) && x <= 3
To have z - x >= 3 influence the condition, we'd need
x <= 3,   and y >= 5,   and z - y >= 1
But from these, we get z - x >= 3, thus the condition can't be false,
and can't influence the decision!

$\Rightarrow$ trying to get MC/DC coverage, we can detect if a condition is written
needlessly complex, or has irrelevant parts (a possible logic error)

In this case, since z - x < 3 cannt have an effect, the condition can be
rewritten setting z - x >= 3 to *true*:

$$(z - y >= 1 \; || \; y < 5) \; \&\& \; x <= 3$$

Unique-Cause MC/DC vs. Masking MC/DC

Unique-Cause MC/DC
the initially presented variant: the influence of a condition must be
shown keeping all other conditions unchanged
  may be impossible to achieve for coupled conditions

Masking MC/DC
a relaxed variant:
in the test pair, not all conditions must have same value,
but both combinations must show the effect of the scrutinized condition

In practice: combination
unique cause for all independent conditions
masking MC/DC for coupled conditions

Predicate coverage

or predicate-complete coverage [T. Ball, 2004]

Previous criteria do NOT correlate multiple decisions
$\Rightarrow$ e.g. combinations of successive `if` statements in the program
$\Rightarrow$ we need a criterion closer to *path coverage*
(which would cover all execution paths)

Approach: identify n relevant *predicates* (conditions) in the program
Try to generate all $S \cdot 2^n$ possible combinations
  S states (program locations), n predicates
$\Rightarrow$ correlates between them all states and predicates in the program

Predicate coverage example [T. Ball]

```
void partition(int a[], int n) { // assume(n>2);
  int pivot = a[0];
  int lo = 1, hi = n-1;
  while (lo <= hi) {
    while (a[lo] <= pivot)
      lo++;
    while (a[hi] > pivot)
      hi--;
  if (lo < hi)
    swap(a,lo,hi);
  }
}
```

Is it correct? Do you detect an error?

Relevant predicates: branch conditions
lo <= hi, lo < hi, a[lo] <= pivot, a[hi] > pivot

Coverage criteria for cycles

[Beizer, Software Testing Techniques]

For simple cycles
– zero iterations (cycle is skipped)
  possibly also: negative counter – correct behavior?
– one iteration
– two iterations (may catch *initialization errors)*
– one typical intermediate value
– N-1 iterations
– N iterations
– try to force N+1 iterations (more than assumed max)

For nonzero minimum: try min-1, min, min+1 ...

Coverage for multiple cycles [Beizer]

1. minimal number of outer iterations
   try inner cycle completely (as independent cycle)
2. continue following cycles outwards
   – with inner cycle at typical iteration count
   – vary count for current cycle
3. finally, vary all cycles together from min to max

Other path testing criteria

Boundary interior path testing

- all paths that traverse a cycle once, without repetition
  (boundary test)

- all paths that repeat a test, at most once
  (interior test)

Linear Code Sequence and Jump (LCSAJ)

an LCSAJ sequence: straight line code followed by a jump

length N LCSAJ criterion: N such consecutive sequences

$N = 1$ ensures line coverage

$N = 2$ ensures branch coverage (even more)

Mutation-based testing

Try changing decisions/statements according to some patterns
to detect if the program runs differently

Examples:
– $<$ changed to $<=$, etc.
– $+1$ changed to -1 or ignored
– limits changed by ± 1
– a || b changed to a, resp. b
  (is the test relevant?); same for a && b

If a mutation is not caught ("mutant not killed") by any test
$\Rightarrow$ either tests are insufficient
$\Rightarrow$ or program may be wrong (or has irrelevant code)

Dataflow coverage criteria

Criteria so far: linked to program *control flow*
Alternative: criteria linked to *data flow* (*dataflow coverage*)

Key notions:
– variable *definition* (**def**): place where it's assigned
– variable **use**: place where it is read
  (used in expression or tested in condition)

Various coverage criteria, e.g.: all-defs, all-uses
def-use coverage: cover each *feasible* pair of def-use with a test case

How relevant is coverage?

Errors increase with complexity

Cyclomatic complexity (of CFG)
$= E - V + 2$ (E = edges, V = nodes)
is a good measure for complexity

We want better coverage for more complex code

But: code coverage is not an absolute measure for test quality
cf. Brian Marick: How to misuse code coverage