Software Testing
Lecture 3
Coverage

Justin Pearson

2017
Approaches to testing

- **Black Box Testing**: Test without looking at the code/hardware
- **White Box Testing (clear box testing)**: Test the internal structure of the software

There is also grey box testing where you look for test cases that cover the specification and cover some aspect of the code. It is a grey area.
It is all about coverage

- Black box testing: test by covering the specification
- White box testing: test by covering the source code
  - Execution paths
  - Statements
  - Decision coverage
  - ...

Short version:
- Complete coverage hard to define or impossible;
- So we have to find some approximation.
Turing’s halting problem

Does this program halt?

```c
main() {
    int i = 0;
    int z = 0;
    for (i = 0; i < 10; i++) {
        z = z + 1;
    }
}
```
Turing’s halting problem

Does this program halt?

```c
int i = 0;
int z = 0;
while (1 != 0) {
    z = z + 1;
}
```
Turing’s halting problem

- Can I write a program that takes *any* program and decides if it halts?

Seems that is might be possible, but it is mathematically impossible.
Turing’s halting problem

Proof

- Enumerate all programs. There are infinitely many, but still a countable number\(^1\).
- Give each program a number.
- The function

\[
h(i, x) = \begin{cases} 
1 & \text{if program } i \text{ halts on input } x \\
0 & \text{otherwise}
\end{cases}
\]

is not computable. That is there no always halting program that implements \(h\).

\(^1\)You can put them in an infinitely long list.
Given

\[ h(i, x) = \begin{cases} 
1 & \text{if program } i \text{ halts on input } x \\
0 & \text{otherwise}
\end{cases} \]

Define

\[ g(i) = \begin{cases} 
0 & \text{if } h(i, i) = 0 \\
\text{loop forever} & \text{otherwise}
\end{cases} \]

g is a program it has a number, let's call it \( G \).

What of \( h(G, G) \)? Two possibilities:

\[ h(G, G) = 1 \] then \( g \) halts on input \( G \), so \( g(G) = 0 \) which implies \( h(G, G) = 0 \), hence a contradiction.

\[ h(G, G) = 0 \] then \( g \) loops forever on input \( g \). Which implies \( h(G, G) \) is not equal to 0, hence contradiction.

Proof strategy is often referred to as a diagonal argument.
Common Caveats

▶ Halting function should work on *all* programs.
▶ Finite memory, finite number of registers a computer is just a finite state machine.
▶ So it is possible to write a function that decides if all programs up to a given size terminate, but not very efficiently.
▶ Also knowing that the program halts for all memory sizes up to certain value does not necessarily tell you anything about bigger sizes.
▶ How big is big enough?
Rice’s Theorem

- All interesting properties are non-computable.
- Ask yourself: Is what I’m trying to do equivalent to the halting problem?
- Are all execution paths covered?
  - If you could solve that problem, then you would solve the halting problem.
- This is the origin of

  “Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger Dijkstra.
Pragmatics

- Admit we can not decide properties on all programs.
- Do our best on most programs.
- Or be content with approximations such as:
  - Definitely yes
  - Definitely no
  - I have no idea.
Coverage criteria include:

▶ Function coverage — Has each function (or subroutine) in the program been called?
▶ Statement coverage — Has each statement in the program been executed?
▶ Branch coverage — Has each branch of each control structure (such as in if and case statements) been executed?
▶ Loop coverage — Have we done a representative number of iterations of all the loops.
Statement and Branch coverage

- Statement coverage $\subseteq$ Branch coverage

```c
silly(int x) {
    int y=0;
    if (x==1) {
        y=100;
    }
    twonk(y);
}
```

- The test case silly(1) covers all statements in the program, but it does not cover all branches. We never test the case when $x \neq 1$. 

Statement and Branch coverage

- Even branch coverage is a blunt instrument.

```c
int silly(int x) {
    int y = 0;
    while (x >= 0) {
        y = y + x;
        x--
    }
    return (y);
}
```

- The test case `silly(1)` runs the loop once. A while loop is a branch with a goto statement.
- But what about running the loop zero times, lots of times?
- Halting problem again, for all most all loop you can’t decide how many times to run the loop.
Control Flow Graphs

Control flow graphs models the control structures of the program. We can use them to reason about executions and test cases.

- Nodes: Statements of sequence of statements
- Edges: Transfer of control
- Basic Block: Sequence of statements with no transfer of control.
If statements

```c
if (x < y)
{
    y = 0;
}
else
{
    x = y;
}
```
If statements

```c
if (x < y)
{
    y = 0;
}
```

![Flowchart diagram]
If return statements

```python
if (x<y)
{
    return;
}
print(x);
return;
```

Note that we do not collapse the two return statements.
Loops

\[
\text{for}\ (i=0; \ i<x; \ i++) \ \{ \\
\quad \text{loop\_body}(); \\
\}
\]
• Other program constructs are easy to do.
• Each node only to be labelled with one basic block.
• Beware of hidden control structures. (C’s case statement).
Other definitions possible.

- A path is a sequence of nodes.
- The length of a path is the number of edges. A path with only one node, hence no edge has length 0.
- Subpath is sub-sequence of a path.
- Reach\( (n) \) the set of nodes that can be reach via a directed path from the node \( n \).
Paths include: [0, 3, 8], [0, 4, 8], [0, 4, 9], [1, 4, 8], . . . , [4, 8], . . .
Reach($n$)

- Reach(3) = \{8\}, Reach(1) = \{4, 9, 8, 6, 10\}.
Two notions in program analysis:

- Syntactic reach.
- Semantic Reach.

```c
main() {
    for (i = 0; true; i++) {
        if (f(i) == 0) { break(0); }
    }
    X();
}
```

`X()` is syntactically reachable, but semantically you have to infer something about `f()`. 
Test Path

- A test path starts at an initial node and ends in a final node.
- Test paths represent execution of test cases.
  - Some paths can be executed by many test cases.
  - Some paths can not be executed by any test cases (halting problem again).
Test and Test Paths

Many to one. Deterministic software, each test path has identical execution.

Many to many, non-deterministic software (you’ll meet it all the time) a test can execute many test paths.
Testing and Covering Graphs

- **Test Requirements (TR)**: Describe properties of test paths
- **Test Criterion**: Rules that define test requirements
- **Satisfaction**: Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in \( \text{path}(T) \) that meets the test requirement.

General idea in testing. Define your test requirements separately from the tests cases. Reformulate your requirements into test criteria and then try to find test paths that satisfy your test criteria.
Node Coverage — Statement Coverage

- Node Coverage (NC) : Test set $T$ satisfies node coverage on graph $G$ iff for every syntactically reachable node $n$ in $N$, there is some path $p$ in path($T$) such that $p$ visits $n$. 
Edge Coverage — Branch Coverage

- Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in $G$.

  Is there any difference between node and edge coverage?
Difference between node and edge coverage

- **Node coverage**
  - Test requirement (TR) = \(\{0, 1, 2\}\).
  - Test path = [0, 1, 2].

- **Edge Coverage**
  - Test requirement (TR) = \(\{(0, 1), (0, 2), (1, 2)\}\).
  - Test paths = [0, 1, 2], [0, 2].
Complete Path Coverage

- Require that all paths are covered.

Often there are too many paths. So various approximations.
- Require that all paths up to length $k$ are covered.
  - $k = 0$, node coverage.
  - $k = 1$, edge coverage.
  - $k = 2$, edge pair coverage.
Structural Coverage Example

Node Coverage: $TR = \{0, 1, 2, 3, 4, 5, 6\}$, test paths $= \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$.

Edge Coverage: $TR =$
$= \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (4, 6), (5, 4)\}$.

Complete Path Coverage. Test paths $[0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 1, 2, 4, 5, 4, 6], [0, 1, 2, 4, 5, 4, 5, 4, 6]$, etc.
Structural Coverage Example

Edge-Pair Coverage: \( TR = \{[0, 1, 2], [0, 2, 3], [0, 2, 4], [1, 2, 3], [1, 2, 4], [2, 3, 6], [2, 4, 5], [2, 4, 6], [4, 5, 4], [5, 4, 5], [5, 4, 6]\} \)

Test Paths
- \([0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 2, 3, 6]\)
- \([0, 2, 4, 5, 4, 5, 4, 6]\).
Loops

There is a lot of theory, most of it is unsatisfactory.

- Don’t be content with branch coverage
- Look at your loops.
  - Try to get them to execute zero times, once and many times.
Loops

- If a graph contains a loop then it has an infinite number of paths.
- Thus you can not ask for complete path coverage.
- Attempts to deal with loops:
  - 1970s: Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s: Execute each loop, exactly once (formalised)
  - 1990s: Execute loops 0 times, once, more than once (informal description)
  - 2000s: Prime paths
Simple and Prime Paths

- A path is simple if no node appears more than once except possible that the first and last node can be the same.
- A prime path of a graph is a simple path that is not a sub-path of any other simple path.
Simple Paths

- [0],[1],[2],[3]
- [0, 1],[0, 2],[1, 3],[2, 3],[3, 0]
- [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0],[3, 0, 1]
- [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1].
Prime Paths

Remove all simple paths that can be extended (either direction) to a longer simple path.

- [0], [1], [2], [3]
- [0, 1], [0, 2], [1, 3], [2, 3], [3, 0]
- [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1]
- [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1].

In this case the prime paths are all the longest simple paths. Not always the case.
Simple Paths

Enumerate all simple paths of length, 1, 2, 3, … then remove simple paths that can be extended. You will be left with the prime paths.

- [1], [2], [3], [4]
- [1, 2], [2, 3], [2, 4], [3, 2]
- [1, 2, 3], [1, 2, 4], [2, 3, 2]
- We have to be careful about the paths of length 4.
  - [1, 2, 3, 2] is not a simple path. Repeats 2 which is not at the beginning or the end.
- In fact there are no simple paths of length 4 in this graph.
Prime Paths

Enumerate all simple paths of length, 1, 2, 3, \ldots then remove simple paths that can be extended. You will be left with the prime paths.

\[
\begin{align*}
&x = 0 \\
&i++ \quad \downarrow \\
&\text{body} \\
&3 \quad 4
\end{align*}
\]

- \([1, 2, 3], [1, 2, 4], [2, 3, 2]\)
Prime Paths to Test Paths

$x = 0$

- $[1, 2, 3] \rightarrow [1, 2, 3, 4]$
  - Execute loop once.
- $[1, 2, 4] \rightarrow [1, 2, 4]$
  - Execute loop zero times.
- $[2, 3, 2] \rightarrow [1, 2, 3, 2, 4]$
  - Execute loop more than once.

<body

i++
Simple Paths

- \[1, 2, 3, 4, 5, 6, 7\]
- \[1, 2, 3, 4, 5, 6, 3, 6, 3\]
- \[1, 2, 3, 4, 5, 6, 3, 4\]
- \[1, 2, 3, 4, 5, 6, 3, 4, 5\]
- \[1, 2, 3, 4, 5, 6, 3, 4, 6\]
- \[1, 2, 3, 4, 5, 6, 3, 4, 5\]
- \[1, 2, 3, 4, 5, 6, 3, 5, 6, 3\]
- \[1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 3\]
Prime Paths

1 → 2 → 3 → 4 → 5 → 6 → 7

- [1], [2], [3], [4], [5], [6], [7], [8]
- [1, 2], [2, 3], [3, 4], [3, 7], [4, 5], [4, 6], [5, 6], [6, 3], [6, 3], [6, 4]
- [1, 2, 3], [2, 3, 4], [2, 3, 7], [3, 4, 5], [3, 4, 6], [4, 5, 6], [4, 6, 3], [4, 6, 3], [5, 6, 3], [5, 6, 3], [6, 3, 4]
- [1, 2, 3, 4], [1, 2, 3, 7], [2, 3, 4, 5], [2, 3, 4, 6], [3, 4, 5, 6], [3, 4, 6, 3], [4, 5, 6, 3], [5, 6, 3, 4], [6, 3, 4, 5], [4, 6, 3, 4], [5, 6, 3, 4]
- [1, 2, 3, 4, 5], [1, 2, 3, 4, 6], [2, 3, 4, 5, 6], [3, 4, 5, 6, 3]
- [1, 2, 3, 4, 5, 6]
Prime Paths

- $[1, 2, 3, 7]! \rightarrow [1, 2, 3, 7]$
  - Do the loop zero times.

- $[3, 4, 6, 3] \rightarrow [1, 2, 3, 4, 6, 3, 7]$
  - Do the loop once and do not do the if

- $[6, 3, 4, 5] \rightarrow [1, 2, 3, 4, 6, 3, 4, 5, 6, 3, 7],$
  - Do the loop twice, once with the if and once without.

- $[4, 6, 3, 4] \rightarrow [1, 2, 3, 4, 6, 3, 4, 6, 3, 7]$
  - Do the loop twice, both times without taking the if.

- $[5, 6, 3, 4] \rightarrow [1, 2, 3, 4, 5, 6, 3, 4, 6, 3, 7]$
  - Do the loop twice, take the if once and once without, other way around from the previous case.

- $[3, 4, 5, 6, 3] \rightarrow [1, 2, 3, 4, 5, 6, 3, 7]$
Prime Paths: Summary

- Prime paths give you a good way of deriving a set of test cases that cover various combinations of loops and branches.
- There is no formal guarantee about completeness. As in all testing it just formalises a good compromise.
Model, define, and approximate

- Model what you want to test.
- Define coverage criteria.
- If coverage criteria is undecidable or requires too many test cases then approximate.
Separate test requirements and test cases

- Have a reason for a test.
- Test requirements are the reasons for tests.
- You need to find test cases satisfying test cases.
Example

```c
int count_spaces(char* str) {
    int length, i, count;
    count = 0;
    length = strlen(str);
    for(i=1; i<length; i++) {
        if(str[i] == ' ')
            count++;
    }
}
```
First Divide into Basic Blocks

```c
int count = 0;
length = strlen(str);

for(i=1; i<length; i++)
    if(str[i] == ' ')
        count++;

return(count);
```
```c
int count = 0;
length = strlen(str);

for (i = 1; i < length; i++)
    count++;

if (str[i] == ',')
    return (count);
```
Test Path

- Remember a test path is a path that starts at an entry node and leaves at an exit node.
Node Coverage

```c
int count = 0;
length = strlen(str);

for(i=1; i<length; i++)
    count++;

if(str[i] == ' ')
   return(count);
```

TR = {1, 2, 3, 4, 5, 6, 7},
Test path is
[1, 2, 3, 4, 5, 6, 3, 7]
Grey box testing

- Our test path [1, 2, 3, 4, 5, 6, 3, 7] requires the loop to execute exactly once and it to detect one space. So we might try the test case (" ",1) but this won’t work. Don’t forget that i=1 in the loop body.
- Instead we have to use the test case ("H ",1)
- By thinking about what the code should do, and trying to construct a test case corresponding to a path, we have uncovered a fault.
Edge Coverage

```c
int count = 0;
length = strlen(str);

for(i=1; i<length; i++)
    count++;

if(str[i] == ',')
    return(count);
```

Test paths are

- $[1, 2, 3, 4, 5, 6, 3, 7]$,
- $[1, 2, 3, 4, 6, 3, 7]$,
- $[1, 2, 3, 7]$.
Test Cases

- [1, 2, 3, 4, 5, 6, 3, 7] (" ", 1)
- [1, 2, 3, 4, 6, 7] ("H", 0)
- [1, 2, 3, 7] (" ", 0)
Relaxing test cases

- As we have seen, sometimes we have infeasible test cases.
  - This could because there is a fault.
  - Or, that we have to do other things to get to the code. There might be a bit of setup code that we have to call first that is not in our path.
- Before we introduced the notion of a path touring another path.
- A path $p$ tours the path $s$ if $s$ is a sub-sequence of $p$.
  - $[1, 2, 3, 4, 6, 3, 4, 6, 3, 7]$ tours the test path $[1, 2, 3, 4, 6, 3]$ it also tours many other paths including $[4, 6, 3, 7]$.
  - Don’t forget the difference between a test path and a path.
Relaxing Test Cases

- A test path $p$ is set to *tour* sub-path $q$ with *side-trips* if every edge that is in $q$ is also in $p$ in the same order.
- A test path $p$ is set to *tour* sub-path $q$ with *detours* if every node that is in $q$ is also in $p$ in the same order.
The path [0, 1, 2, 3, 2, 4, 5] tours the path [0, 1, 2, 4, 5] with side trips.
The path $[0, 1, 2, 3, 4, 5]$ tours the path $[0, 1, 2, 4, 5]$ with side trips.
Infeasible test requirements

- An infeasible test requirement *cannot be satisfied*
  - Unreachable statement (dead code)
  - Can only be executed if a contradiction occurs $X > 0 \land X < 0$.
  - Always check against the specification, it could be a fault.
Infeasible test requirements

- Most test criteria have some infeasible test requirements.
- It is usually undecidable to decide if all test requirements are feasible (halting problem again).
- Allowing side trips might weaken the test cases, but allows more feasible test cases.
- Practical recommendation, best effort touring. Allow as many as possible without side-trips, only allow side-trips on infeasible test paths.