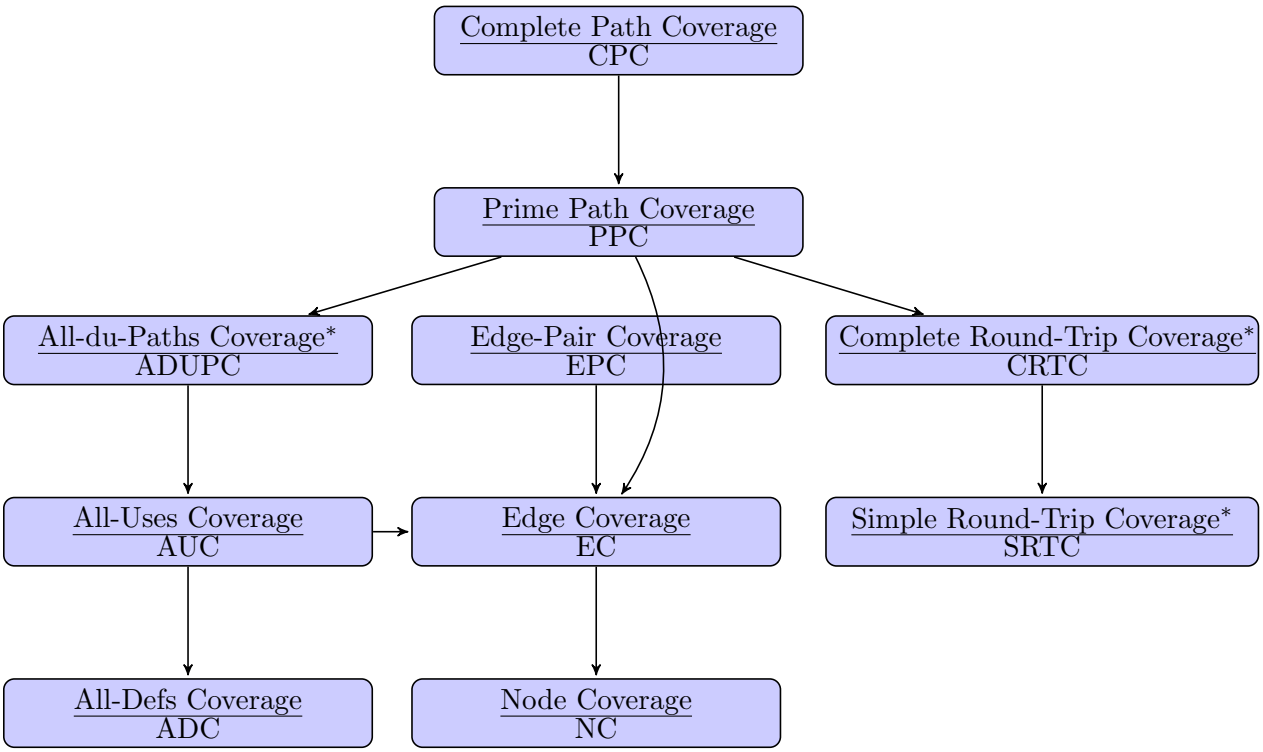
Software Testing, Quality Assurance and Maintenance Winter 2015

Lecture 16 — February 9, 2015

Patrick Lam *version 2*

Subsumption Chart

Here are the subsumption relationships between our graph criteria.



- We know EPC subsumes EC subsumes NC from before, and clearly CRTC subsumes SRTC. Also, CPC subsumes PPC and PPC subsumes EPC. PPC subsumes CRTC, because simple paths include round trips.
- In this offering, we didn't talk about ADUPC at all, and we only touched briefly on CRTC and SRTC.

Assumptions for dataflow criteria:

1. every use preceded by a def (guaranteed by Java)
2. every def reaches at least one use

3. for every node with multiple outgoing edges, at least one variable is used on each out edge, and the same variables are used on each out edge.

Then:

- AUC subsumes ADC.
- Each edge has at least one use, so AUC subsumes EC.
- Finally, each *du*-path is also a simple path, so PPC subsumes ADUPC (not discussed this term, but ADUPC subsumes AUC). (Note that prime paths are simpler to compute than data flow relationships, especially in the presence of pointers.

Dataflow Graph Coverage for Source Code

Last time, we saw how to construct graphs which summarized a control-flow graph's structure. Let's enrich our CFGs with definitions and uses to enable the use of our dataflow criteria.

Definitions. Here are some Java statements which correspond to definitions.

- `x = 5`: `x` occurs on the left-hand side of an assignment statement;
- `foo(T x) { ... }` : implicit definition for `x` at the start of a method;
- `bar(x)`: during a call to `bar`, `x` might be defined if `x` is a C++ reference parameter.
- (subsumed by others): `x` is an input to the program.

Examples:

Uses. The book lists a number of cases of uses, but it boils down to "`x` occurs in an expression that the program evaluates." Examples: RHS of an assignment, or as part of a method parameter, or in a conditional.

Complications. As I said before, the situation in real-life is more complicated: we've assumed that `x` is a local variable with scalar type.

- What if `x` is a static field, an array, an object, or an instance field?
- How do we deal with aliasing?

One answer is to be conservative and note that we've said that a definition d reaches a use u if it is possible that the address defined at d refers to the same address used at u. For instance:

```
class C { int f; }
void foo(C q) { use(q.f); }

x = new C(); x.f = 5;
y = new C(); y.f = 2;

foo(x);
foo(y);
```

Our definition says that both definitions reach the use.

Exercise. Consider the following graph:

- $N = \{0, 1, 2, 3, 4, 5, 6, 7\}$
- $N_0 = \{0\}$
- $N_f = \{7\}$
- $E = \{(0,1), (1,2), (1,7), (2,3), (2,4), (3,2), (4,5), (4,6), (5,6), (6,1)\}$
- test paths:
  - $t_1 = [0, 1, 7]$
  - $t_2 = [0, 1, 2, 4, 6, 1, 7]$
  - $t_3 = [0, 1, 2, 4, 5, 6, 1, 7]$
  - $t_4 = [0, 1, 2, 3, 2, 4, 6, 1, 7]$
- def(0) = def(3) = use(5) = use(7) = { x }

(a) Draw the graph.

(b) List all *du*-paths with respect to x. Include all *du*-paths, even those that are subpaths of others.

(c) List a minimal test set (using the given paths) satisfying all-defs coverage with respect to x.

(d) List a minimal test set (using the given paths) satisfying all-uses coverage with respect to x.

Compiler tidbit. In a compiler, we use intermediate representations to simplify expressions, including definitions and uses. For instance, we would simplify:

```
x  = foo(y + 1, z * 2)
```

Basic blocks and defs/uses. Basic blocks can rule out some definitions and uses as irrelevant.

- Defs: consider the last definition of a variable in a basic block. (If we're not sure whether `x` and `y` are aliased, leave both of them.)
- Uses: consider only uses that aren't dominated by a definition of the same variable in the same basic block, e.g. `y = 5; use(y)` is not interesting.

Graph Coverage for Design Elements

We next move beyond single methods to "design elements", which include multiple methods, classes, modules, packages, etc. Usually people refer to such testing as "integration testing".

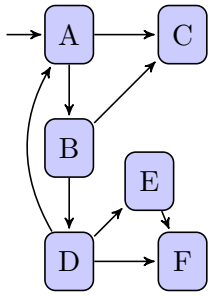
Structural Graph Coverage.

We want to create graphs that represent relationships—*couplings*—between design elements.

Call Graphs. Perhaps the most common interprocedural graph is the *call graph*.

- design elements, or nodes, are methods (or larger program subsystems)
- couplings, or edges, are method calls.

Consider the following example.



- For method coverage, must call each method.
- For edge coverage, must visit each edge; in this particular case, must get to both A and C from both of their respective callees.

Like any other type of testing, call graph-based testing may require test harnesses. Imagine, for instance, a library that implements a stack; it exposes `push` and `pop` methods, but needs a test driver to exercise these methods.

Data Flow Graph Coverage for Design Elements

The structural coverage criteria for design elements were not very satisfying: basically we only had call graphs. Let's instead talk about data-bound relationships between design elements.

- *caller:* unit that invokes the callee;
- *actual parameter:* value passed to the callee by the caller; and
- *formal parameter:* placeholder for the incoming variable.

Illustration.

caller: `foo(actual1, actual2);`
callee: `void foo(int formal1, int formal2) { }`