Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 8 — January 20, 2010

Patrick Lam *version 1*

Building on the notion of a def-clear path:

Definition 1 *A* du-path *with respect to v is a simple path that is def-clear with respect to v from a node n_i, such that v is in def(n_i), to a node n_j, such that v is in use(n_j).*

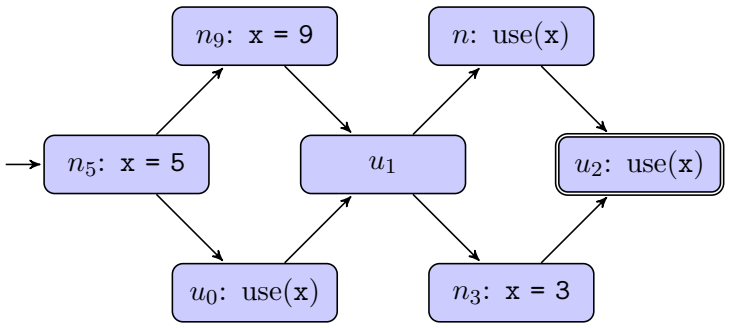
(This definition could be easily modified to use edges e_i and e_j).

Note the following three points about *du*-paths:

-
-
-

Coverage criteria using du-paths

We next create groups of *du*-paths. Consider again the following double-diamond graph D:



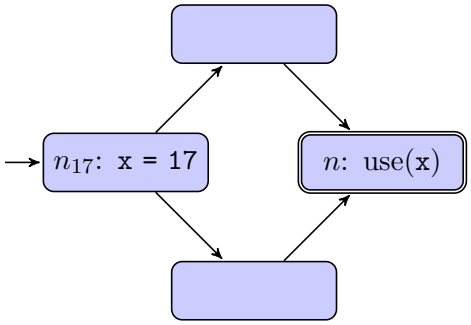
We will define two sets of *du*-paths:

- def-path sets: fix a def and a variable, e.g.
  - $\text{du}(n_5, x) =$
  - $\text{du}(n_3, x) =$
- def-pair sets: fix a def, a use, and a variable, e.g. $\text{du}(n_5, n, x) =$

These sets will give the notions of all-defs coverage (tour at least one *du*-path from each def-path set—a weak criterion); all-uses coverage (tour at least one *du*-path from each def-pair set); and all-du-paths coverage (tour all *du*-paths from each def-pair set).

How can there be multiple elements in a def-pair set?

Here's an example with two *du*-paths in a def-pair set.



We then have

$$\mathrm{du}(n_{17}, n, x) =$$

Note the general relation

$$\mathrm{du}(n_i, v) = \bigcup_{n_j} \mathrm{du}(n_i, n_j, v)$$

There are more def-pair sets than def-path sets. Cycles are always allowed.

Useful exercise. Create an example where one def-path set splits into several def-pair sets; you can get a smaller example than the one in the book.

Touring *du*-paths

Definition 2 *A test path p* du-tours *subpath d with respect to v if p tours d and d is def-clear with respect to v.*

We could allow or disallow def-clear sidetrips. Def-clear sidetrips make more paths tourable, so we choose to allow them. The definition then says that the part of p that *du*-tours d must be def-clear.

We can use the above definitions to provide 3 coverage criteria; we assume Best-Effort Touring.

Criterion 1 All-Defs Coverage *(ADC). For each def-path set $S = du(n, v)$, TR contains at least one path d in S.*

Criterion 2 All-Uses Coverage *(AUC). For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path d in S.*

Criterion 3 All-du-paths Coverage *(ADUPC). For each def-pair set $S = du(n_i, n_j, v)$, TR contains every path d in S.*

What do these criteria mean? For each def,

- ADC: reach at least one use;
- AUC: reach every use somehow;
- ADUPC: reach every use in every simple way (all *du*-paths).

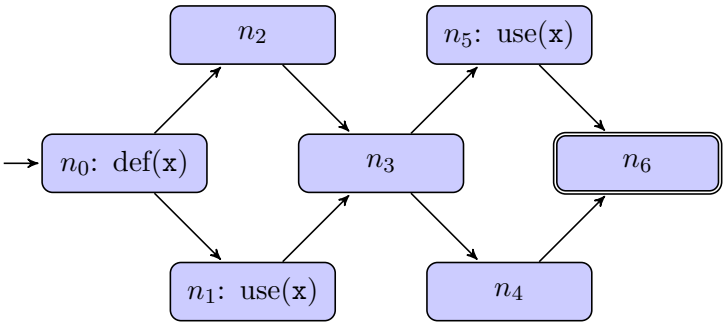
In the context of the earlier example,

- ADC requires:
- AUC requires:
- ADUPC requires: same as AUC since def-pair sets are all singletons.

Nodes versus edges. So far, we've assumed definitions and uses occur on nodes.

- uses on edges ("*p*-uses") work as well;
- defs on edges are trickier, because a *du*-path from an edge to an edge may not be simple. (We could make things work out with more work.)

Another example.

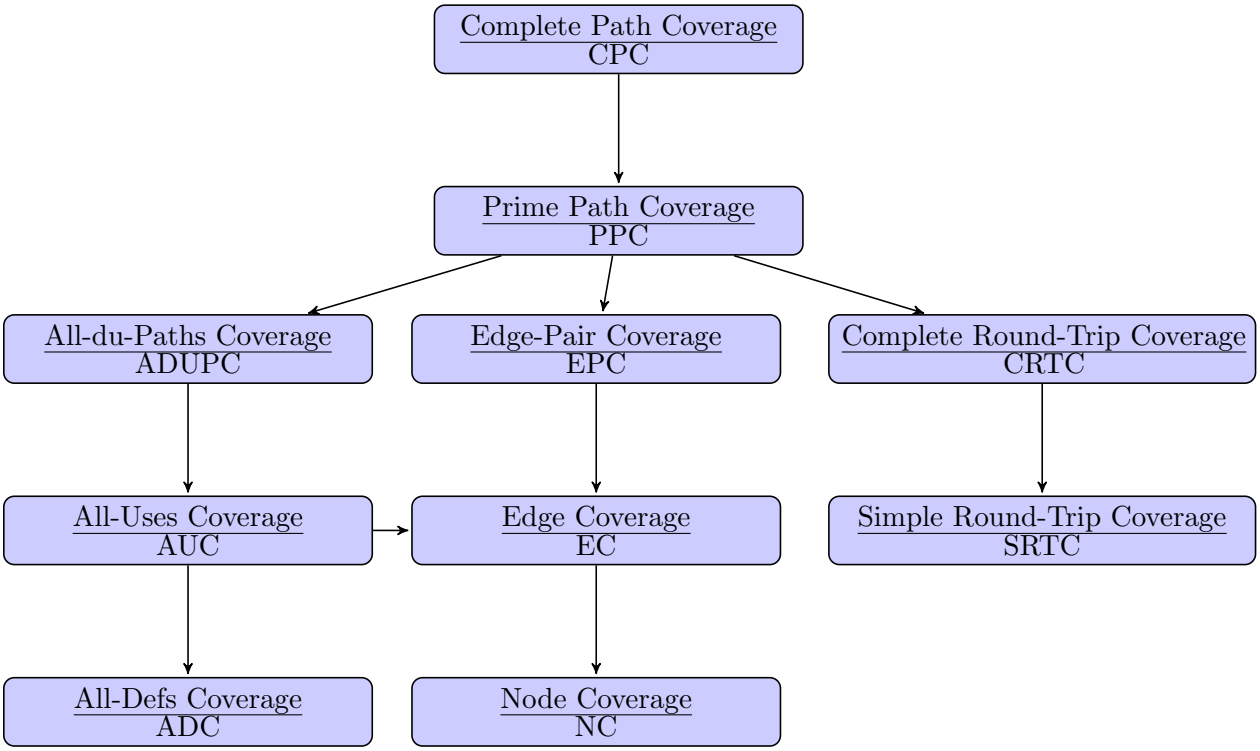


Some test sets that meet these criteria:

- ADC:
- AUC:
- ADUPC:

Subsumption Chart

Here are the subsumption relationships between our graph criteria. Assume Best-Effort Touring.



- Best-Effort Touring saves us from infeasible TRs.
- We know EPC subsumes EC subsumes NC from before, and clearly CRTC subsumes SRTC. Also, CPC subsumes PPC subsumes EPC, and PPC subsumes CRTC, because simple paths include round trips.

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 and ADUPC subsumes AUC.
- Each edge has at least one use, so AUC subsumes EC.
- Finally, each *du*-path is also a simple path, so PPC subsumes ADUPC. (Note that prime paths are simpler to compute than data flow relationships, especially in the presence of pointers.