Software Testing, Quality Assurance & Maintenance (ECE453/CS447/CS647/SE465): Midterm Solutions

February 10, 2009

This open-book midterm has 5 questions and 90 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1 (20 points): First-uses

(10) In the context of a single method,

Criterion A. *For each def-pair set $S = (n_i, n_j, v)$, TR contains all du-paths d in S.*

Criterion B. *For each def-pair set $S = (n_i, n_j, v)$, such that n_j is the first use in its basic block, TR contains all du-paths d in S.*

Identify and explain the subsumption relationships between these criteria, giving examples as necessary.

In brief, **A** and **B** are identical: visiting the first use in a basic block ensures that you visit all uses in the block. (This insight is enough to get full marks.) More formally, observe that, by definition, **A** includes all test requirements induced by **B**. To show equivalence of **A** and **B**, we must also show that **B** subsumes **A**. Let a test set T satisfy **B**. To show subsumption, T must also satisfy **A**. Consider an arbitrary test requirement in **A**; such a test requirement is an arbitrary du-path p from def d to use u. We must show that T tours p. 1) If u is the first use in its basic block, then **B** will explicitly include this p and hence T tours test requirement p. 2) Otherwise, u is not the first use in its basic block b. Statements in basic blocks are totally ordered, so there exists an unambiguous first use u_1 in b. (Since we are positing the existence of a du-path p from d to u, then there can be no def on the unique path between u_1 and u.) Furthermore, we know that **B** includes the prefix

p_1 of p which goes from d to u_1, so that T includes a test path t that tours p_1. Test paths start at initial nodes and final nodes; they never stop in the middle of a basic block. Hence path t (and test set T) must tour p as a consequence of touring p_1, as required.

(10) If n_i and n_j are in different methods, linked by a method call, define:

Criterion A'. *For each def-pair set $S = (n_i, n_j, v)$, TR contains all du-paths d in S.*

Criterion B'. *For each def-pair set $S = (n_i, n_j, v)$, such that n_j is a first use in its method, TR contains all du-paths d in S.*

Identify and explain the subsumption relationships between A' and B', giving examples as necessary.

Clearly, **A'** still subsumes **B'**. Here is an example of a case where **B'** does not subsume **A'**.

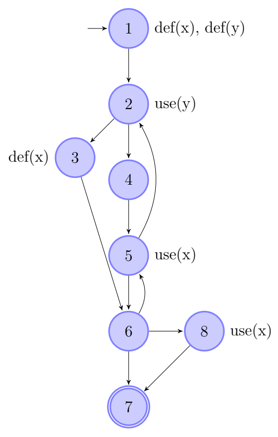
```
void caller() {
  def(x);
  callee(x);
}

void callee(x) {
  use(x);
  if (?) {
    use(x); // (*)
  } else { }
  return;
}
```

Note that **A'** contains the du-path involving (*) and **B'** has no such requirement; it could be satisfied with a test set that only executes the empty else branch of the `if` statement.

Question 2 (20 points): Coverage Criteria

Here is a control-flow graph.



(a) Consider the set of test paths:

$$[1, 2, 3, 6, 8, 7], [1, 2, 4, 5, 6, 7], [1, 2, 4, 5, 2, 3, 6, 5, 6, 7].$$

Which of the following coverage criteria does the test set cover? Complete Path Coverage (CPC); Prime Path Coverage (PPC); Complete Round Trip Coverage (CRTC); Simple Round Trip Coverage (SRTC); Edge-Pair Coverage (EPC); Edge Coverage (EC); Node Coverage (NC); All-du-Paths-Coverage (ADUPC); All-Uses Coverage (AUC); All-Defs Coverage (ADC). (+1 for including or omitting each criterion correctly).

This question should not take too long, if you don't go and enumerate all the criteria explicitly. You don't need to do that; just look for paths that violate each criterion. They're not too hard to find. Also, using the subsumption chart saves time.

- CPC: no, the CFG is cyclic and the test set is finite.
- PPC: no, missing $[2, 3, 6, 5, 2]$. (You can do this by inspection).
- CRTC: no, the above path is also a missing round trip.
- SRTC: no, no paths get back to 3 after hitting 3 once.
- EPC: Let's verify EC first. If we fail EC, then we fail EPC too.

- EC: yes, edges are $(1,2)$, $(2,3)$, $(2,4)$, $(3,6)$, $(4,5)$, $(5,2)$, $(5,6)$, $(6,5)$, $(6,7)$, $(6,8)$, $(8,7)$. (Count the edges to check that you didn't forget any). Verify that test set contains each of these edges.
- NC: yes, we satisfy EC.
- EPC: no; now we have to create edge-pairs until we find one that doesn't hit. I started with $(1,2,3)$ and reached $(5,6,8)$, which this test set does not hit.
- ADC: yes, def at 1 satisfied by $[1,2,4,5]$, and def at 3 satisfied by $[3,6,5]$.
- AUC: no, collect du-pair sets $du(1,5,x)$, $du(1,8,x)$, $du(3,5,x)$, $du(3,8,x)$. Note that we never go from 1 to 8 without redefining x at 3.
- ADUPC: no, because we don't satisfy AUC.

(b) What about this set of test paths:

$$[1,2,3,6,5,2,4,5,6,7], [1,2,3,6,5,2,3,6,7], [1,2,4,5,2,4,5,6,7]$$

- CPC: no.
- PPC: no, missing $[5,6,5]$.
- CRTC: no, still missing $[5,6,5]$.
- SRTC: yes.
- EPC: no, $(5,6,5)$ is a valid edge-pair as well.
- EC: no. Note that we don't even hit NC; we miss 8.
- NC: no.
- ADC: yes, same as before: def at 1 satisfied by $[1,2,4,5]$, and def at 3 satisfied by $[3,6,5]$.
- AUC: no, we don't hit use at 8.
- ADUPC: no, because we don't satisfy AUC.

Question 3 (20 points): True or False

(I gave a bonus point here.) True or false:

1. Coverage criterion C_1 imposes an infeasible test requirement `tr` on a program P. Coverage criterion C_2 also imposes `tr`. There exists a test set that fully meets C_2. **False. You just can't satisfy `tr`.**

2. Prime path coverage can always be satisfied with simple test paths. **False, say $p = 2 - 3 - 2$ is a prime path; initial node 1, final node 4. Then you must use a non-simple path e.g. $1–2–3–2–4$ to visit p.**

3. A prime path in a graph is never a suffix of another prime path in the same graph. **True, definition of prime path.**

4. Complete graph coverage is possible for all graphs. **False, cycles; we only consider finite test sets.**

5. A def-pair set is a union of def-path sets for that def[1]. **False.**

6. Satisfying coupling inter-procedural data-flow coverage is always easier than satisfying full inter-procedural data-flow coverage. **True.**

7. An unreachable program fault can be detected using testing. **False.**

8. If test set T_1 achieves a higher coverage level than T_2 on a set of test requirements TR, then T_1 will detect more defects than T_2. **False.**

9. The control-flow graph corresponding to a Java method is sometimes not a Single-Entry/Single-Exit graph. **True, multiple returns.**

10. If test path p tours subpath q with sidetrips, then p also tours q with detours. **True.**

11. All-du-Paths is equivalent to All-Uses in all graphs without cycles. **False.**

12. Defs and uses for the same variable may appear on the same control-flow graph node. **True, x++.**

[1]Don't believe everything you read in textbooks!

13. A test case will always exclusively visit syntactically reachable nodes in a control-flow graph. **True.**

14. If p tours q, and q tours r with detours, then p also tours r with detours. **True.**

15. A basic block b may have two predecessors b_0 and b_1. **True.**

16. After executing a round-trip in a control-flow graph, the program's state is identical to its state preceding the round-trip. **False. Sort of true for FSMs, but not for CFGs.**

17. Any path can be composed by concatenating prime paths. **False; the statement is only true for simple paths.**

18. Dead code makes it impossible to achieve node coverage. **False. Dead code is unreachable, and node coverage only talks about reachable nodes. (Pretty much everyone said true here.)**

19. A sidetrip is always a simple path. **False, there are no restrictions on sidetrips.**

20. Fewer paths are semantically possible through a program than are syntactically reachable. **True.**

21. It is conceptually possible to cover all-du-paths for a shared-data coupling variable. **True (it's just hard).**

Question 4 (10 points): AUC vs EC

We've stated that All-Uses Coverage subsumes Edge Coverage. (1) Under what conditions does this subsumption relationship hold? (9) Give an example where AUC does not subsume EC; you may falsify the conditions you stated in the first part.

The conditions (p50) are: (1) every use preceded by a def; (2) every def reaches at least one use; (3) multiple outgoing edges imply at least one variable used on each out edge, and same variables used on each out edge.

Falsify (3); here's a case where you can satisfy AUC without EC:

```
def(x);
if (?) {
  use(x);
} else { }
```

To achieve AUC, we need to test the def-use pair, but we don't need to visit the else-branch of the `if` statement. Condition (3) would require a use of some variable on the else branch, which would imply that all-uses would have to cover the else branch as well. This condition is not quite realistic, because sometimes you don't actually use a tested variable at a conditional.

Question 5 (20 points): Control-Flow Graphs

Create a control-flow graph of the following method (5 points) and provide inputs that achieve edge coverage (7 points) and all-defs coverage (7 points) on your control-flow graph. (At conditionals, you may put uses either on the edges or on the nodes.)

We consider `x.put()` *to be a use of* `x`*. (1 point) What type of coupling is going on with* `rhsToContainingStatement`*?*

Examples of statements `s` *are* `x = 5, x = y + z, x = new Foo()` *(all* `AssignStmts`*) and* `return;`*. You may assume that "5" is not an* `Expr`*.*

```
public void Foo(LinkedList units)
{
    rhsToContainingStmt = new HashMap<Value, Unit>();
    emptySet = new ToppedSet(new ArraySparseSet());

    unitToGenerateSet = new HashMap();
    Iterator unitIt = units.iterator();

    while(unitIt.hasNext())
    {
      Unit s = (Unit) unitIt.next();

      FlowSet genSet = emptySet.clone();
      if (s instanceof AssignStmt)
      {
        AssignStmt as = (AssignStmt)s;
```

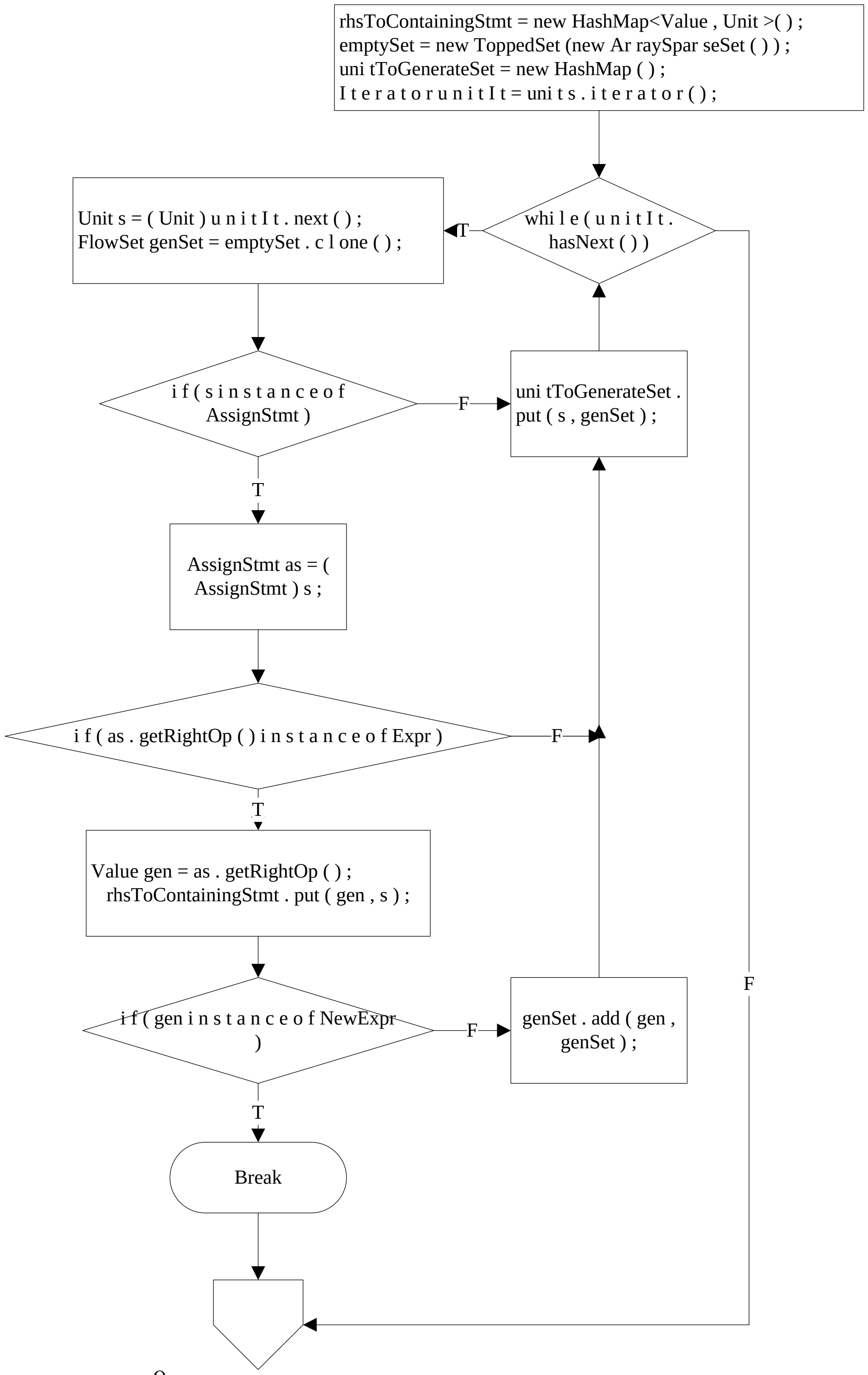
```
            if (as.getRightOp() instanceof Expr)
            {
              Value gen = as.getRightOp();
              rhsToContainingStmt.put(gen, s);

              if (gen instanceof NewExpr)
                  break;
              genSet.add(gen, genSet);
            }
          }

          unitToGenerateSet.put(s, genSet);
        }
}
```

By the way, the code is modified from the Soot compiler framework. It originally populated a hash map from right-hand sides of expressions to their containing statements; I changed it to make a more interesting exam question.

The control-flow graph is fairly straightforward to draw. The attached CFG is due to TA Mohammad Yazdandoost. You only had to make clear which statements were in each block, not include the full statements.


rhsToContainingStmt = new HashMap<Value , Unit >( ) ;
emptySet = new ToppedSet (new Ar raySpar seSet ( )) ;
uni tToGenerateSet = new HashMap ( ) ;
I t e r a t o r u n i t I t = unit s . i t e r a t o r ( ) ;
whi l e (u n i t I t . hasNext ( ))
T
Unit s = (Unit) u n i t I t . next ( ) ;
FlowSet genSet = emptySet . c l one ( ) ;
i f (s i n s t a n c e o f AssignStmt)
F
uni tToGenerateSet . put (s , genSet) ;
T
AssignStmt as = (AssignStmt) s ;
i f (as . getRightOp ( ) i n s t a n c e o f Expr)
F
T
Value gen = as . getRightOp ( ) ;
rhsToContainingStmt . put (gen , s) ;
i f (gen i n s t a n c e o f NewExpr)
F
genSet . add (gen , genSet) ;
T
Break
F

Note that **shared data coupling** is going on with `rhsToContainingStmt`, since it is being written to here and (presumably) read from later.

To achieve edge coverage, you simply need to include every kind of statement that I've demonstrated:

```
[], [return], [x = 5], [x = new Foo()], [x = y + z]
```

(Many solutions are possible for this part of the question. You have to recognize that the function is taking a list, and you have to include a non-assignment statement, a statement with something that's not an assignment, an assignment with a `new` expression, and an assignment with some non-`new` expression.)

Finally, the defs are of `rhsToContainingStmt`, `emptySet`, `unitToGenerateSet`, `unitIt`, `s`, `genSet`, `as`, and `gen`. You should identify the defs in your solution (otherwise how can you know that you hit all defs?) Since I didn't ask for a minimal set, you might as well start with the test cases that you used for edge coverage and notice that this set also gets you all-defs coverage.