Exam Software Testing & Verification 2016/2017

22nd May 2017, 13:15–15:00, EDUC-THEATRON

Lecturer: Wishnu Prasetya

1 Part I (4pt)

1. What does it mean for a coverage criterion C_1 to subsumes another coverage criterion C_2?

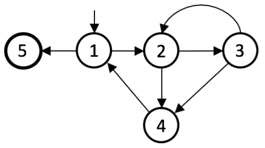
   Answer: Every test set that (fully) covers C_1 will also (fully) cover C_2.

2. What is a simple path and what is a prime path?

   Answer: A simple path in a graph: is a path in the graph such that no node in the path occurs twice, except if it is the first and the last one.

   Prime path: is a simple path that is not a subpath (is not toured by) of another simple path.

3. Consider the Control Flow Graph (CFG) below. Node-1 is the initial node, and node-5 is the exit node.

   List all prime paths in this CFG, **that starts in node-2**.

   Answer: There are 5 prime paths that start from node-2:

   The cycles: 232, 2412, 23412

   Non cycles: 23415

   Note that 2415 is not a prime path because it is a subpath of 32415, which is a simple (and prime) path.

4. Consider a program $P(x, y, z)$. The domains of x, y, and z are each partitioned into two blocks:

   $$\begin{array}{lll} x & \rightarrow & \{X_1, X_2\} \\ y & \rightarrow & \{Y_1, Y_2\} \\ z & \rightarrow & \{Z_1, Z_2\} \end{array}$$

   Give a smallest **test set** that would give full Pair-wise Coverage (PWC) over those blocks.

   Answer: Four test cases will do:

   (X_1, Y_1, Z_1)
   (X_1, Y_2, Z_2)
   (X_2, Y_1, Z_2)
   (X_2, Y_2, Z_1)

5. Consider again the program P in No. 4. We select (X_1, Y_1, Z_1) as a so-called base test-case. Give the smallest **test set** that would give full Base Choice Coverage (BCC).

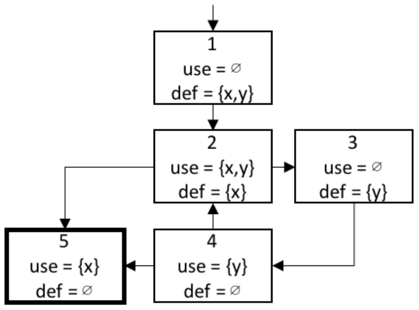
   Answer: The test set with the following test cases:

   (X_1, Y_1, Z_1) (the base test)(X_2, Y_1, Z_1)
   (X_1, Y_2, Z_1)
   (X_1, Y_1, Z_2)

6. Give the definition of "du-path of a variable x".

   Answer: Given the CFG of a program, a du-path of x is a *simple path* in the CFG starts with a node that contains a def to x and ends with a node that contains a use of x, and furthermore the path is def-clear with respect to x.

7. Consider the following CFG of some program. The nodes are annotated with defs and uses information. If the same variable occurs in the def and the use of the same node n, we assume that within this node the def of this variable occurs *after* its use.



   Which paths should be included as your test requirements (**TR**) to have full All-Defs Coverage (ADC)? Note that the program has two variables. Also note that you are asked to enumerate your TR, and *NOT* to specify a test set.

   Answer:

   The du-paths of x are: 12, 2342, 2345, 25. For ADC the TR is 12 and one of $\{2342, 2345, 25\}$.

   The du-paths of y are: 12, 34, 342. For ADC the TR is 12 and one of $\{34, 342\}$.

8. Consider a program $Q(s)$ that takes a string s as a parameter. The string has to follow the grammar shown below (described in the BNF notation):

$$\begin{array}{lcll} S & \rightarrow & \epsilon & \text{(RuleS1)} \\ S & \rightarrow & A & \text{(RuleS2)} \\ S & \rightarrow & B & \text{(RuleS3)} \\ A & \rightarrow & "a"\ S\ "a"\ S & \text{(RuleA)} \\ B & \rightarrow & "b"\ S\ "b"\ S & \text{(RuleB)} \end{array}$$

   The non-terminals are S, A, B with S as the starting symbol. Symbols between quotes are terminals.

   A tester wants to test the program Q using the string "*baabaa*" as the input. Prove that the string is a valid string in the above grammar. Prove this by showing a **derivation** that leads to this string.

Answer:
$$\begin{aligned} S &\to B \\ &\to bSbS \\ &\to bAbS \\ &\to baSaSbS \\ &\to ba\epsilon aSbS \\ &\to ba\epsilon a\epsilon bS \\ &\to ba\epsilon a\epsilon baSaS \\ &\to ba\epsilon a\epsilon ba\epsilon aS \\ &\to ba\epsilon a\epsilon ba\epsilon a\epsilon \end{aligned}$$

9. Give the definition of "coupling du-path".

   Answer: Assume a program A and B where $B(...)$ is called somewhere in A.

   A coupling du-path between module A and B is a du-path the combined CFG of A and B that: (1) goes through a call site of $B(...)$ in A, and (2) it starts with at last-def of a coupling variable v (that is, a coupling variable that couples A and B) in one module, and ends in the first-use of v in the other module.

10. Name two uses of mutation testing.

    Answer:

    Your AO book suggests two application:

    (1) if some inputs of the program under test are strings which are required to follow certain formats defined by some grammars, we can mutate the grammars as a way to systematically generate negative tests.

    (2) we can introduce mutations into the source code of the program under test to artificially introduce errors. This is useful to test the strength of your test suite. (note that this is not a test against the program under test, but rather a test against its test suite)

2 Part II (5pt)

1. Consider a program with N parameters: $P(X_1, ..., X_N)$; $N>0$. The domain of X_N is divided into k blocks; we name them $B_1 ... B_k$. The domains of other parameters are left undivided; so each X_i with $i \neq N$ only has one block; we name it C_i.

   (a) Suppose we want to do pair-wise testing over those blocks. How many pairs of blocks have to be covered in total?

   Answer:

   We will assume that k is at least one, else no test can be constructed for P (it would not have any meaningful input).

   Firstly, if $N = 1$ then there is no pair to cover.

   Now, for $N \geq 2$, let's first take a look at an example with $N = 6$. There are $k * 5$ pairs between X_N and the others X_i. There are 4 pairs between X_1 and the other pairs which we have not counted yet; there are 3 pairs between X_2 and the other pairs which we have not counted yet, and so on. So, there are these number of pairs for $N = 6$

   $$5k + 4 + 3 + 2 + 1 + 0$$

   More generally:

   $$k(N-1) + (N-2) + ... + 0$$

   Or:

   $$k(N-1) + \sum_{i=0}^{N-2} i$$

This can be simplified further, knowing that the sum $\sum_{i=0}^{n} i$ is equal to $n(n+1)/2$. So:

$$k(N-1) + (N-2)(N-1)/2$$

(b) Give a minimal test set that would deliver full Pair-wise Coverage (PWC) over the blocks.

Answer: Fortunately, you can cover all the pairs with much less number of test cases than the number of pairs. These test cases will do:

$$\{(C_1, ..., C_{N-1}, B_m) \mid 1 \leq m \leq k\ \}$$

2. Prove that All-du-Paths Coverage (ADUPC) does **not** subsume Prime Path Coverage.

Answer: You only need to show one counter example. Consider the CFG in Part-I no 7. In the answer there I have listed all du-paths with respect to x as well as y. The following test set that covers all of them:

$$12345, 123425$$

However, this test set does not cover the prime path 3423. This demonstrates that ADUPC does not subsume PPC.

3. Consider the program fragment below. Assume that all array accesses are valid.

```
1  module A {
2    int[] zap(int N) {
3        int[] a = mkarray(...) ;
4        int[] b = mkarray(...) ;
5        int k = 0 ;
6        while (k<N) {
7            foo(a,k) ;  // call point (*)
8            if (b[k]>0)
9                b[k] = a[k]
10           k++ ;
11       }
12       return b ;
13   }
14 }
15
16 module B {
17  foo(int[] a, int k) {
18     if (k>9)
19       return ;
20     a[k] = a[k] + a[k+1] ;
21     if(a[k]<0) {
22       a[k] = 0 ;
23     }
24  }
25 }
```

The def and use of an array `a` is defined as follows:

- An assignment `a` =... is a def of `a`.
- An assignment `a`[*e*] = ... is a def of `a`.
- An expression `a`[*e*] in the guard of an **if** or a **while** is a use of `a`.
- An expression `a`[*e*] in the right-hand-side of an assignment is a use of `a`.

Furthermore:

- Passing a **variable** `v` directly in a program call such as `f(v)` does **not** in itself count as a use of `v`. The defs and uses of `v` inside *f* do count as defs and uses of `v`.

Your task: list all du-paths that should be included as test requirements (**TR**) for achieving full **All-coupling-use Coverage**. **For each path, mention the name of the coupling variable that the path is associated to.**

Use line numbers to describe your paths. For example: [2,3,4,5,6] describes the path from the start of `zap` up to its loop-head.

Answer: There are two coupling variables that concern the call point marked above, namely a and k. Let us first list all the coupling du-paths of each:

couple var a:	3..6[true]..**7**..18[false]..20
	20..21[false]..24..**7**..8[true]..9
	22..24..**7**..8[true]..9
couple var k	5..6[true]..**7**..18
	10..6[true]..**7**..18

For each coupling variable, notice that each coupling path has a different pair of last-def and first-use. So, for ACU we have to include all paths above in the TR.

3 Part III (1pt)

1. Consider the following binding triples within `foo()`, and their corresponding coupling paths. In all cases, the context variable is o. All named paths are distinct.

binding triples			
coupling sequence	runtime type of context-var	coupling variables	coupling paths
C_1	A	$\{o.x\}$	σ_1
C_1	B	$\{o.y\}$	σ_2
C_2	A	$\{o.x, o.y\}$	τ_1 for $o.x$ τ_2 and τ_3 for $o.y$
C_2	B	$\{o.x\}$	τ_4

   (a) Give a minimal set of paths that should be included as the test requirements (**TR**) for achieving full All-Coupling-Defs-Uses (ACDU).

   Answer: For ACDU, for every coupling var v of every coupling sequence, the TR should include at least one of v's coupling path. This TR will do:

   - For C_1, we have two coupling variables $o.x$ and $o.y$, each only has one coupling path. We have to cover them both. So: $\{\sigma_1, \sigma_2\}$.
   - For C_2, we also have $o.x$ and $o.y$ as the coupling variables. For $o.x$ we have to cover one of $\{\tau_1, \tau_4\}$. For $o.y$ we have to cover one of $\{\tau_2, \tau_3\}$.

   (b) Give a minimal addition to the TR in (a) to specify the TR for achieving full All-Poly-Coupling-Defs-Uses (APCDU) coverage.

   Answer: For APCDU, for every coupling var v of every binding triple, TR should include at least one of vs coupling path. The answer for this question depends on what you choose in (a).

   For example, **suppose** for C_2 you decide to include $\{\tau_1, \tau_2\}$ in the TR. In this case we only miss the TR for the last binding triple of C_2. There is only one path there, namely τ_4. So in this case adding τ_4 to the TR in (a) will do.

2. Consider a program $Q(s)$ that takes a string s as a parameter. The string has to follow the grammar shown below (described in the BNF notation):

$$\begin{array}{lll} S & \rightarrow & \epsilon \\ S & \rightarrow & "x" \\ S & \rightarrow & B \\ B & \rightarrow & "<"\ S\ ">"\ S \end{array} \quad \begin{array}{l} \text{(RuleS1)} \\ \text{(RuleS2)} \\ \text{(RuleS3)} \\ \text{(RuleB)} \end{array}$$

   Give a minimalistic test set for Q that would give full All Rule-Rule Coverage (ARRC) with respect to the above grammar. Only rule-combinations that are feasible need to be covered. We favor a larger test set with shorter test cases (and we do **not** favor a small test set with longer test cases).

   It may help (but you don't have to) if you present your test cases in the form of derivation trees.

   Answer: Only the following combinations need to be covered:

$$RuleB;\ 2\ R_1\ ; 4\ R_2$$

   where R_1, R_2 are either $RuleS1, RuleS2, RuleS3$. So in total there are 9 combinations to cover.

The following test cases will cover them all:

" $<>$ "	covers	$RuleB$; 2 $RuleS1$; 4 $RuleS1$
" $<> x$"	covers	$RuleB$; 2 $RuleS1$; 4 $RuleS2$
" $<><>$ "	covers	$RuleB$; 2 $RuleS1$; 4 $RuleS3$
" $< x >$ "	covers	$RuleB$; 2 $RuleS2$; 4 $RuleS1$
" $< x > x$"	covers	$RuleB$; 2 $RuleS2$; 4 $RuleS2$
" $< x ><>$ "	covers	$RuleB$; 2 $RuleS2$; 4 $RuleS3$
" $<<>>$ "	covers	$RuleB$; 2 $RuleS3$; 4 $RuleS1$
" $<<>> x$"	covers	$RuleB$; 2 $RuleS3$; 4 $RuleS2$
" $<<>><>$ "	covers	$RuleB$; 2 $RuleS3$; 4 $RuleS3$