Exam Software Testing & Verification 2011/2012

25 may 2012, 9:00–12:00,BBL-065

Lecturer: Wishnu Prasetya

1. [1.5pt] Consider this program:

```
String foo(c:List<String>) {
   if (c==null)
        c = new List<String>() ;
   String found   = null ;
   int k = 0 ;
   for (s:String in c) {
      if (s.contains("foo")) {
         found = s ;
         k++ ;
      }
   }
   return found + k;
}
```

   (a) Give a control flow graph that correponds to the program. It should be made clear in your drawing which group of instructions and expressions is represented by each node.

   (b) Give the definition of Edge-Pair Coverage, and give the smallest possible set of feasible test-paths that would give you maximum Edge-Pair Coverage.

   Answer: EPC: the TR consists of all paths in the program CFG, of length at most 2.

   Note btw that I asked you to give test paths, not the TR itself.

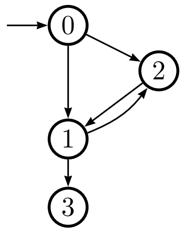
2. [1.5pt] Prime paths.

   (a) Give the definitions of *simple path* and *prime path*.

   Answer: Simple path: is a path in the given CFG where every node appears just once, except the first and the last nodes.

   Prime path: is a simple path that is not a subpath of another simple path (so it is maximal).

   (b) Consider the control flow graph below; 1 is the starting node, and 3 is the exit node.



   List all prime paths in the graph.

   Answer: 013, 012, 0213, 121, 212

(c) A coverage criterion C_1 *subsumes* another criterion C_2 if all test-set that satisfies C_1 also satisfies C_2. Give a test-set on the above graph that shows that the Edge Coverage criterion *does not* subsume the Prime-path Coverage criterion.

Answer: 01213, 0213 cover all edges, but not the prime path 212.

(d) Give a definition of *tour with detour*. That is, when does a test-path t can be said to tour a target path u with *detour*?

Answer: t tours u with detour if every node of u appears in t and in the same order (u is a subsequence of t)

(e) Give a simple program that shows a situation where it is necessary to weaken the concept of 'tour' to 'tour with detour'; specify the problematic prime-path in that example program.

Answer: (not a direct answer of the question:) In the CFG it appears that there is an edge from $i \to j$, but this edge is actually infeasible. There is however an alternative path from i to j. In this situation we may opt to allow detour in our touring definition. See also the example in the book.

3. [1.5pt] Consider the following control flow graph. The nodes are decorated with def and use information; the edges have no def nor use.

$$\begin{tikzcd}
{} \arrow[r] & 0 : def = \{x, y\} \arrow[d] & \\
& 1 : def{=}\{x, y\}, use{=}\{x, y\} \arrow[d] & \\
& 2 : use{=}\{x\} \arrow[r] \arrow[d] & 3 : use{=}\{y\} \arrow[ul] \arrow[dl] \\
& 4 : use{=}\{x, y\} &
\end{tikzcd}$$

State 0 is the initial-state, and state 4 is the end-state. If no def or use is mentioned on a node, it means the corresponding decoration is empty on that node.

(a) Give all members of $du(1, x)$ and $du(1, 1, x)$.

Answer: The def/use decoration in node 1 is ambiguous as it does not specify the exact order of the def and use within the node. I take this into account when I reviewed your answers, and accept all possibe interpretations. For the answer below, I assume both uses occur before the defs.

$du(1, x) = 12, 124, 1234, 1231$

$du(1, 1, x) = 1231$

(b) Give a test-set for the above graph, that shows that full All-Defs Coverage does *not* subsume All-Uses Coverage. Hint: such a test-set gives full coverage of the first, but not the second.

Answer: Note that for ADC, for each $du(i, var)$, the TR only needs to include one element of the du.

So, just a single test 0124 will give full ADC. For example it tours one member of $du(1, x)$, and thus covers the test requirement for that du. However we miss the only member of $du(1, 1, x)$, and thus fails to give full AUC.

Also note that I asked for test-cases, not TR.

4. [1pt] Consider the classes `A` and `B` below; notice that the method `f` of `A` calls the method `judge` of `B`.

```
 1 class A {
 2   b : B ;
 3   ...
 4   String f(int x) {
 5      int y = x*x ;
 6      if (x<0)
 7          y = 0 ;
 8      r = b.judge(y) ;
 9      return r
10   }
11 }
12
13 class B {
14   ...
15   String judge(int y) {
16     if (reader == null)
17         return null ;
18     int x = reader.readFromFile("data") ;
19     if (y>x)
20         return "ok"
21     return "not␣ok"
22   }
23 }
```

We want to test the integration between `f` and `judge` (in other words, we want to do integration testing).

(a) List all the *coupling variables* and their corresponding *coupling du-paths* between `f` and `judge`. You can use line numbers to identify the nodes in your paths.

Answer: Coupling vars are variables which are defined in the caller and used in the callee, or the other way around. These are y, r.

A coupling du-path for y is a du-path from the last def of y in the (in the above case) f to its first use in the *judge*.

Coupling du-paths for y are:

Def y at line 5:	5,(if-false)8,16,(if-true)17,18
Def y at line 5:	5,(if-false)8,16,(if-false),18
Def y at line 7:	7,8,16,(if-true)17,18
Def y at line 7:	7,8,16,(if-false),18

Coupling du-paths for r are:

Def r/*return* at line 17 :	17,8
Def r/*return* at line 20 :	20,8
Def r/*return* at line 21 :	21,8

(b) Specify the test-requirements (the TRs) for All-Coupling-Def Coverage and for All-Coupling-Uses Coverage for the above example. You can specify the TRs in terms of paths.

Answer: For ACDC we only need to include one path per def of each coupling variable above. So, for example:

Def y at line 5:	5,(if-false)8,16,(if-true)17,18
Def y at line 7:	7,8,16,(if-true)17,18
Def r/*return* at line 17 :	17,8
Def r/*return* at line 20 :	20,8
Def r/*return* at line 21 :	21,8

Note that this time you are asked to get the TR, and not test-cases.

Because the first uses of both y and r occur in only one location (repectively), then the above TR is also good for ACUC.

5. [1pt] Consider this function:

$$foo(p : Person, i : Insurance, a : Address)$$

The tester decided to divide the domain of each parameter above into the following blocks:

- $Person$: $Child, Adult, Senior$
- $Insurance$: $None, Standard, Premium$
- $Address$: $Fixed, Moving$

We will only consider abstract test-cases, which are expressed in terms of the blocks above (you don't have to specify the concrete values for p, i, and a).

(a) Give a test-set that would give full Pair Wise Coverage.

Answer: 9 test-cases will be sufficient, e.g.:

CNF	CSF	CPM
ANF	ASM	APF
SNM	SSF	SPM

(b) Suppose we add a constraint that if the address is $Moving$, then the type of insurance must be $Premium$, is it then still possible to get full Pair Wise Coverage? Explain your answer.

Answer: No, because then the combination NM and SM become unfeasible.

(c) The tester chooses this test-case as a *base test*:

$$foo(Child, Premium, Fixed)$$

Note that this implies that those blocks have been chosen as base blocks. Give a minimalistic test-set that gives maximal Base Choice Coverage. Take the constraint given in (b) into account.

Answer: CPF
CPM
CNF
CSF
APF
SPF

Note that we do not violate the given constraint.

6. [1.5pt] Consider this predicate (it has three clauses):

$$isMasterStudent(x) \ \vee \ (isStudent(x) \wedge (x.name = "foo"))$$

For this predicate, a test-case is just values assigned to the clauses. There is a constraint on the clauses, namely that $isMasterStudent(x) \Rightarrow isStudent(x)$; where $\Rightarrow$ means 'implies'. A test-case is only *feasible* if it satisfies this constraint. A test-set is feasible if it only contains feasible test-cases.

(a) Give a feasible test-set (for the above predicate) that shows that Predicate Coverage *does not* subsume Clause Coverage.

Answer:

iM	iS	iF	ϕ
1	1	0	1
0	1	0	0

Note that the tests respect the constraint $iM \Rightarrow iS$.

(b) Give a feasible test-set that gives full Correlated Active Clause Coverage, and *indicates for each test-case, which clause(s) it activates.*

Hint: a truth-table can help you.

Answer:

$p_{iM} = \neg iS \vee \neg iF$
$p_{iS} = \neg iM \wedge iF$
$p_{iF} = \neg iM \wedge iS$

	iM	iS	iF	ϕ	p_{iM}	p_{iS}	p_{iF}	note
1	1	1	1	1				
2	1	1	0	1	Y1			
3	1	0	1	1	Y1			violating constraint
4	1	0	0	1	Y1			violating constraint
5	0	1	1	1		Y1	Y1	
6	0	1	0	0	Y0		Y0	
7	0	0	1	0	Y0	Y0		
8	0	0	0	0	Y0			

We can select for example the pairs (2,6) that activates iM, (5,7) for iS, and (5,6) for iF. Hence, 2,6,5,6 as test-cases will do.

(c) What is the difference between Correlated Active Clause Coverage and Restricted Active Clause Coverage? Does the test-set you gave in (b) give full Restricted Active Clause Coverage?

Answer: In both CACC and RACC not only that for each major clause we have to cover 1/0, but the corresponding value of the formula f must also change. However, in RACC the used values of the minor clauses must be the same, whereas in CACC this does not have to be the case.

The above proposes test-cases also give RACC.

(d) Give a feasible test-set that gives maximum Restricted Inactive Clause Coverage, and *indicates for each test-case, which clause(s) it makes inactivate.*

Answer: In RAIC for each major clause we have to make it inactive. We need to cover all four combinations of the value of the major clause and the value of f, namely $00, 01, 10, 11$. For the cases where the value of f is 0, the chosen values of the minor clauses must be the same. Similarly when for the cases where f is 1.

Note that some of the cases may be unfeasible.

Not that when for example p_{iM} is false, iM becomes inactive.

	iM	iS	iF	ϕ	p_{iM}	p_{iS}	p_{iF}	note
1	1	1	1	1	X11	X11	X11	
2	1	1	0	1	Y1	X11	X01	
3	1	0	1	1	Y1	X01	X11	violating constraint
4	1	0	0	1	Y1	X01	X01	violating constraint
5	0	1	1	1	X01	Y1	Y1	
6	0	1	0	0	Y0	X10	Y0	
7	0	0	1	0	Y0	Y0	X10	
8	0	0	0	0	Y0	X00	X00	

We can see above that 1,2,7,8 give full RAIC for the major clause iF.

1,3,6,8 would give full RAIC for the major clause iS. Unfortunately 3 is not allowed. We can instead use the pair (2,4) instead of (1,3). But 4 is also not allowed. So the best we can give is for example 1,6,8.

Full RAIC for iM is not possible. The best we can give is 1,5.

So, a test-set with maximum RAIC is 1,2,5,6,7,8.

7. [1pt] Mutation.

   (a) Suppose we have a program $P(s : String)$ whose input s is described by the following BNF:

(1)	S	$::=$	ϵ
(2)	S	$::=$	$S\ Brace$
(3)	S	$::=$	$S\ Curly$
(4)	$Brace$	$::=$	"()"
(5)	$Brace$	$::=$	"(" $Brace$ ")"
(6)	$Curly$	$::=$	"{}"

   where S is the starting symbol. Quoted texts are terminals.

   Give as precise as possible a definition of what it means for a test-set on P to have a full Poduction Coverage.

   Give such a test-set; indicate for each test-case which production rules it covers.

   Answer: *Full Production Coverage*: for each production rule, there exists a derivation sequence deriving one of the test input, such that the rule is applied in the derivation.

   Example tests set: (), (()), {}

   (b) Suppose we also want to do negative testing on $P(s)$. That is, we want to test how it deals with invalid s. We introduce a single mutation operator o that can be applied to any of the above production rules to mutate the rule. Explain how to do the following:

      i. Negative test on P that gives full Mutation Operator Coverage.
      Answer: Generate inputs so that for each mutation operator (there is only one above), there is an input that is derived using a production rule, mutated by the operator.

      ii. Negative test on P that gives full Mutation Production Coverage.
      Answer: Generate inputs so that for each mutation operator o (there is only one above), and for each production rule r, there is an input that is derived using r mutated with o.

8. [1pt] Consider the following classes. Notice that `A2` is a subclass of `A1`, which is a subclass of `A`, and notice the overriding of the method `g`.

```
class A {
   int x ;
   A a ;
   public f() { g() ; x = a.x }
   public g() { a = new A() ; a.x = ... }
   public h() { x = 0 ; a.x = 0 }
}

class A1 extends A {
  override public g() { super.g() ; a.x = ... }
}

class A2 extends A1 {
  override public g() { h() }
}
```

(a) Draw the Yo-Yo graph of the above classes.

(b) What is the purpose of a Yo-Yo graph?

Answer: It is used to abstractly visualize how polymorphism in OO (the one that is due to inheritence) influence your objects' behavior. Behavior is viewed abstractly in terms of call sequences between methods accross the given inheritence chain/hierarchy.

(c) Give an example of either data flow anomaly or a State Definition Anomaly.

Answer: Data flow anomaly occurs when there is a use of some variable x which has not been defined. Overriding a method may lead to such an anomaly, as is the case above. The method f() contains a use of a.x. In the class A and A1, this use is preceeded by a def, namely in g. But A2 overrides g() where this def now disappear. So In A2.f() we have this anomaly.

This is not necessarily an error (it could be that A2.f is only called after a.x has been set). But it indicates potential to make errors (when it turns out there there is a part in the program that calls A2.f with unset a.x).

SDA. Let A1 be a subclass of A. SDA occurs when an object of A1 causes some promised properties (in particular state related properties) of A to break. E.g. it could be that A1 changes the part of the state defined by A inconsistently (from A's perspective).

The above case of data flow anomaly is an instance of SDA.