Exam Software Testing & Verification 2013/2014

4th June 2014, 13:15–15:15, BBL-165

Lecturer: Wishnu Prasetya

1. [CFG-based testing, 2 pt] Consider this program:

```
1  Integer isMemberOf(u:String, s:List<String>) {
2     if (s==null)
3         return null ;
4     int k = 0 ;
5     for (v:String in s) {
6        if (u.equals(v))
7           return k ;
8        k++ ;
9     }
10    return -1 ;
11 }
```

   (a) Give a control flow graph that correponds to the program. Label each node with the line numbers of the statements it represents.

   (b) Give a set of test paths that would give full node coverage, but *not* full edge coverage.

   (c) Suppose in the context where `isMemberOf` is called, the list `s` is never null nor empty. List all prime paths which are impossible to be toured, and specify for each of them if it can still be toured with detour or sidetrip.

   (d) Given a CFG G, let's define the TR (Test Requirement) of *k-path coverage* to consist of all paths in G of length up to k. As in A&O, we define the length of a path to be the number of edges that the path consists of. Prove that the prime path coverage criterion does *not subsume* k-path coverage, for $k \geq 3$.

2. [Black-box partition-based testing, 2 pt] To test a program P we have identified *browser*, *user*, and *query* to be three *characteristics* that influence the behavior of P. So abstractly, a test-case for P is a tuple $P(b, u, q)$ specifying the values of each of these characteristics that are to be used in the test-case.

   We decide to partition these characteristics into blocks as shown below; the names between brackets are abbreviations you can use to refer to them.

Characteristic	#blocks	*Blocks*
browser	4	Chrome (BC) , Mozilla (BM), IE (BIE), Opera (BO)
user	3	Member (UM), Admin (BA), Intruder (UI)
query	3	Normal (QT), WithInjection (QWI), Illegal (QI)

   (a) Give a smallest possible test set that would give you *full* pair-wise as well as each-choice coverage.

   (b) For each of the following constraints, indicate whether it is still possible to give full pair-wise coverage, when the constraint is imposed. Motivate your answer.

i. Users of types 'Member' and 'Admin' always submit normal queries.
ii. Intruders will try all types of browsers.
iii. Users of type 'Member', when they use Chrome, cannot submit illegal queries.

(c) Consider now the following test cases:

- $tc_1 \;=\; P(IE, Admin, WithInjection)$
- $tc_2 \;=\; P(IE, Intruder, WithInjection)$

Give a smallest possible test set that would give full Multiple Base Choice Coverage (MBCC) using the above test cases as the *base tests*; the base chocies are thus the blocks listed above.

(d) Suppose we have $C_1, ..., C_k$ as characteristics, and each C_i is divided into $|C_i|$ number of blocks. Suppose we have N number of base tests, such that for characteristic i, its number of base choices is m_i. Give a formula that specifies general minimum on the number of test cases that will give you full MBCC coverage, based on those N base tests. *General* here means, that in *any situation* the number of needed test cases would be at least that specified minimum, although in *some situation* you may be able to eliminate some duplicates and hence needing less then that general minimum.

3. [Predicate testing, 1.5 pt] Consider a program implementing this predicate f, consisting of three clauses a, b, c, which are assumed to be independent of each other:

$$(a \Rightarrow b) \;=\; c$$

We will abstractly describe test cases and test requirements for f in terms of combinations of the values of (a, b, c).

(a) Complete the truth table below. In the f-column, fills in the value of the predicate f on the corresponding combination of the clauses; and in the last column, specifies which clauses are *activated*. Please stick to the given order of the combinations.

a	b	c	f	activated clauses
0	0	0	...	...
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

(b) Give a minimum test set that gives full *clause coverage* but *not* full *predicate coverage*.

(c) Give for each clause, the combinations that would give it full *Restricted Active Clause Coverage*. If this is not possible, use *Correlated Active Clause Coverage* instead for that clause. Use the table below, and try to minimize the total set of test requirements you end up with:

activated clause	*combinations to for RACC*	*else, combinations for CACC*
a	..., ...	
b	...	
c		

(d) Name which clauses can be feasibly covered with *General Inactive Clause Coverage*, and which of them can still be feasibly covered with *Restricted Inactive Clause Coverage*?

4. [Predicate testing, 2 pt] Consider a predicate f, specified by the following Karnaugh map:

$cd \downarrow \setminus ab \rightarrow$	00	01	11	10
00	0	1	1	0
01	1	1	1	1
11	1	1	0	0
10	0	0	0	0

(a) Give a *minimal* DNF describing f and another minimal one describing $\neg f$.

(b) Give a smallest possible test set that gives full *Implicant Coverage* (with respect to the DNFs in (a)).

(c) Give a smallest possible test set that gives full *Unique True Point Coverage* (with respect to the DNFs in (a)).

(d) Prove that *Unique True Point and Near False Point Coverage* (CUTPNFP) does not subsume Unique True Point Coverage.

5. [Complex Input, 1 pt] If e is a regular expression, let $[e]$ denote either an empty string or sentences you can obtain from e.

Consider a program $P(s)$ where s is a string whose syntax is specified by this regular expression:

$$1^*[a|b|c](0|1)^+c$$

(a, b, c above simply represent the corresponding literal characters)

(a) Give a finite state automaton M that equivalently describes the syntax.

(b) We also want to do negative tests on P by giving it invalid s. Notice that any invalid string must contain a valid prefix. An invalid string is *prefix-minimal* if you cannot make it shorter by dropping its last element while still being invalid. Propose an algorithm to generate *all* prefix-minimal invalid inputs for P of length $\leq k$.

You can assume the finite state automaton M to be described by a tuple (S, s_0, F, E, R) where S is its set of states, $s_0 \in S$ is its initial state, $F \subseteq S$ is its set of final states, E is the set of labels decorating the arrows, and $R : S \rightarrow E \rightarrow \{S\}$ is a function describing the arrows; $R\ s\ \alpha$ specifies the set of states connected by an arrow from s, labelled with α.

6. [Integration testing, 1.5 pt] Consider the classes `Game` below, that contains two method: `move` and `activate`. The first calls the latter (line 12).

```
1  class Game {
2    int instance ;
3    Collection<GameObject>[] state ;
4    ...
5
6    String move(String id, Vector v) {
7        if (mangled(id))
8            id = fix(id) ;
9        if (v.isNegative())
10           v = v.normalize() ;
11
12       GameObject o = activate(id) ;   // **
13
14       if (o==null) {
15           o = new GameObject(v) ;
16           Collection S = state[instance] ;
17           S.add(o) ;
18       }
19       else
20           o.move(v) ;
21       return o
22   }
23
24   GameObject activate(String id) {
25     if (state[instance] == null)
26         state[instance] = new Tree() ;
27     for (GameObject o : state[instance])
28        if (o.id == id) {
29          o.active = true ;
30          return o ;
31          }
32     return null
33   }
34 }
```

Suppose we want to test the integration between the method `move` and `activate`, and we want to apply the intergration testing approach as in A&O. We will define a variable x to be *defined* at line number i, if the line contains an assignment of either of these forms:

$$x = ... \;\; , \;\; x.fieldname = ... \;\; , \;\; x[e] = ...$$

Similarly, x is *used* at line number i, if the line contains an evaluation of an expression of either of this form: x, $x.fieldname$, $x[e]$.

(a) List all the **coupling variables** that couple `move` and `activate`. For each, specify all its **coupling du-paths**. Use line numbers to identify the nodes in your paths.

(b) Specify a minimalistic TR for each of the following coverage criteria. Express the TR in terms of a set of coupling paths from (a).

   i. **All-Coupling-Def Coverage**
   ii. **All-Coupling-Use Coverage**
   iii. **All-Coupling-du-Path Coverage**