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

This open-book midterm has 4 questions and 90 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1: Prime Path Coverage (25 points)

Consider the following code (modified from the GPLed library `pdfsam` *by Andrea Vacondio):*

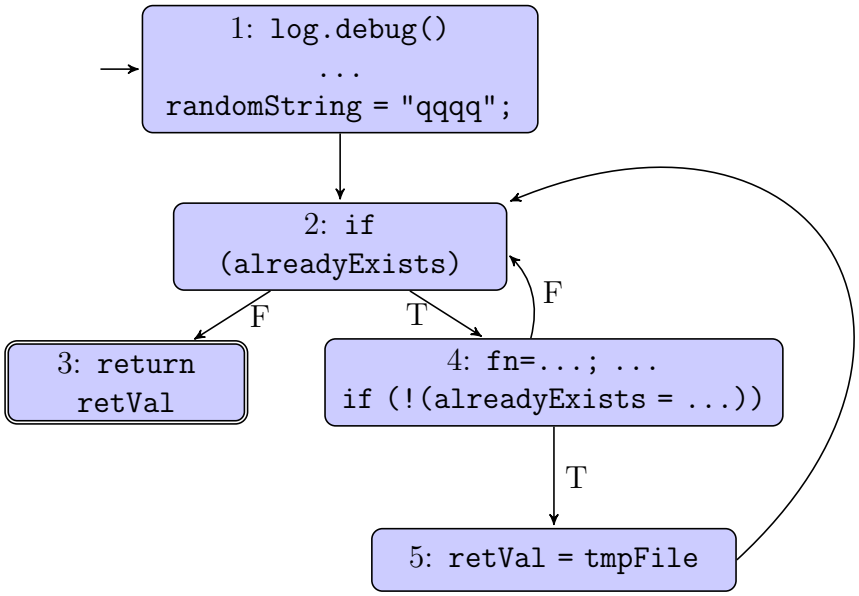
```
public static File generateTmpFile(String filePath){
   log.debug("Creating temporary file..");

   File retVal = null;
   boolean alreadyExists = true;
   int enthropy = 0;
   String fileName = "";
   String randomString = "qqqq"; // not so random anymore. -PL

   while(alreadyExists){
       fileName = FileUtility.BUFFER_NAME+randomString+
                   Integer.toString(++enthropy)+".pdf";
       File tmpFile = new File(filePath+File.separator+fileName);
       if (!(alreadyExists = tmpFile.exists())) {
           retVal = tmpFile;
       }
   }
   return retVal;
}
```

(5 points) Draw a control-flow graph for this method. (10 points) Enumerate the test requirements for Prime Path Coverage on your CFG. (10 points) Provide a test suite for this method which will satisfy prime path coverage using Best Effort Touring and explain why your test suite satisfies PPC. If there are infeasible test requirements, explain why they are infeasible. (A test case may assume that it starts in an empty directory, but may create new files in that directory before calling `generateTmpFile`*. Just write "*`create file X`*".)*

Here is a control-flow graph:



Prime path coverage imposes the following test requirements:

$$[1,2,3],[1,2,4,5],[2,4,5,2],[4,5,2,3],[4,5,2,4],[5,2,4,5],[2,4,2],[4,2,4].$$

The following test suite achieves PPC. Assume the directory is initially empty before each test case runs. Here are the test cases:

1. { } (no files): test path $[1,2,4,5,2,3]$
2. {`create file "qqqq1.pdf"`}: test path $[1,2,4,2,4,5,2,3]$

My reasoning is as follows:

- $[1,2,3]$: infeasible directly, but the sidetrip $[1,2,4,5,2,3]$ tours this TR (case 1)
- $[1,2,4,5]$: case 1
- $[2,4,5,2]$: case 1
- $[4,5,2,3]$: case 1

Two more test cases require going around the loop at least once.

- $[2,4,2]$: case 2
- $[4,2,4]$: case 2

Finally, two cases are infeasible:

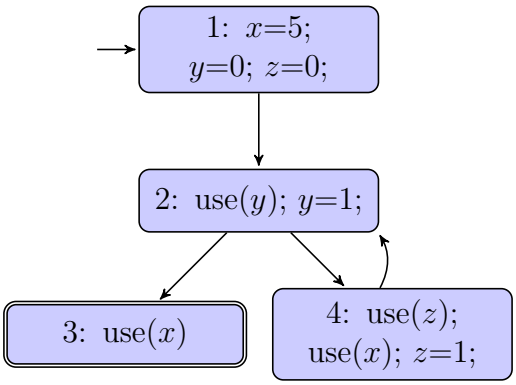
- $[4,5,2,4]$: infeasible, executing 5 will always cause the F branch to be taken at 2.
- $[5,2,4,5]$: same

Question 2: Comparing ADUPC and PPC (30 points)

Draw a control-flow graph, annotated with the relevant definitions and uses, where ADUPC and PPC impose the same test requirements. (List these test requirements.) **Your CFG must contain a loop.**

This question was fairly tricky, so we were lenient while marking it. We gave 5 points for recognizing that because PPC subsumes ADUPC, you only need to find a case where ADUPC imposes the same test requirements as PPC, and we gave 10 points for having a graph of the appropriate shape, leaving 10 points for the annotations (2 points for putting a def and a use somewhere, 3 points for at least using multiple variables) and 5 points for the test requirements.

Here is that graph that I came up with:



The test requirements for PPC are:

$$[1, 2, 3], [1, 2, 4], [2, 4, 2], [4, 2, 4].$$

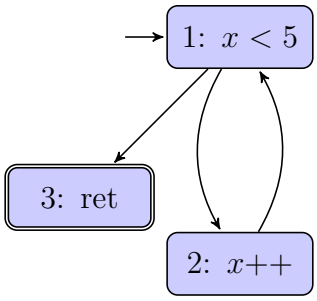
I therefore used different variables to take care of each of this requirements. In particular, x imposes the test requirements $[1, 2, 4]$ and $[1, 2, 3]$. Then y imposes the test requirement $[2, 4, 2]$ based on the definition in 2 and the use in 2 (because the use precedes the def), and z imposes the requirement $[4, 2, 4]$ based on the definition in 4 and the use in 4.

We gave full credit if any test set which would satisfy ADUPC on your example would also satisfy PPC.

Question 3: Comparing EPC and EC (10 points)

Edge-pair coverage ought to impose more test requirements than edge coverage. (6 points) Write a Java method where EPC imposes a test requirement that EC doesn't impose. (You can do this with 3 lines of code.) Draw the CFG and write out the test requirements for both EPC and EC. (2 points) Produce a test set that satisfies EC but not EPC. (2 points) Produce a test set that satisfies EPC.

I drew the CFG first and then came up with a corresponding method.



A corresponding Java method would be:

```
void m(int x) {
  while (x < 5) x++;
  return;
}
```

Coming up with the test requirements for whatever you drew was worth 2 points. In my case, the test requirements for EC are:

$$[1, 2], [2, 1], [1, 3],$$

while the test requirements for EPC include those for EC and also the pairs:

$$[1, 2, 1], [2, 1, 2], [2, 1, 3].$$

The test set $\langle 4 \rangle$ causes the test path $[1, 2, 1, 3]$, which meets EC but not EPC (since it is missing $[2, 1, 2]$); the test set $\langle 3 \rangle$ meets both EC and EPC, since it leads to the test path $[1, 2, 1, 2, 1, 3]$.

Question 4: Creating a Finite State Machine (25 points)

Read the attached excerpt from RFC 4254, "The Secure Shell (SSH) Connection protocol". This excerpt descripts the protocol for handling an SSH channel. (a) (10 points) Describe the abstract states in this protocol. (We've seen how to create a single FSM for a specification. If you think you can create interacting server and client FSMs, go for it.) (b) (10 points) Describe the transitions between states. (c) (5 points) Draw the FSM for opening, using, and closing SSH channels.

Note: Avoid creating an FSM that looks like a control-flow graph.

I apologize for the poor quality of the photocopy.

I used the following states:

uninitialized connection	UNINIT
originator is opening channel	REQUESTED
remote side agrees and opens channel	OPEN
channel full	FULL
one side has EOFed	EOF1
one side has EOFed and channel full	EOF1_FULL
both sides have EOFed	EOF2
close requested	CLOSE_REQ
close acknowledged	CLOSE_ACK

We'll call one of the sides the local side and the other side the remote side. Here are the transitions between states that I found:

(a) UNINIT→REQ: local requests channel

(b) REQ→OPEN: remote opens

(c) OPEN→OPEN: request/receive larger window

(d) OPEN→OPEN: pre: window space available; cmd: send data

(e) OPEN→FULL: pre: data will fill window; cmd: send data

(f) FULL→OPEN: consume data or receive larger windowsize

(g) OPEN→EOF1: one side sends EOF

(h) FULL→EOF1_FULL: one side sends EOF

(i) EOF1→EOF1: pre: window space available; cmd: other side sends data

(j) EOF1→EOF1_FULL: pre: data will fill window; cmd: send data

(k) EOF1_FULL→EOF1: consume data or receive larger windowsize

(l) EOF1→EOF2: other side sends EOF

(m) FULL→CLOSE_REQ: request close

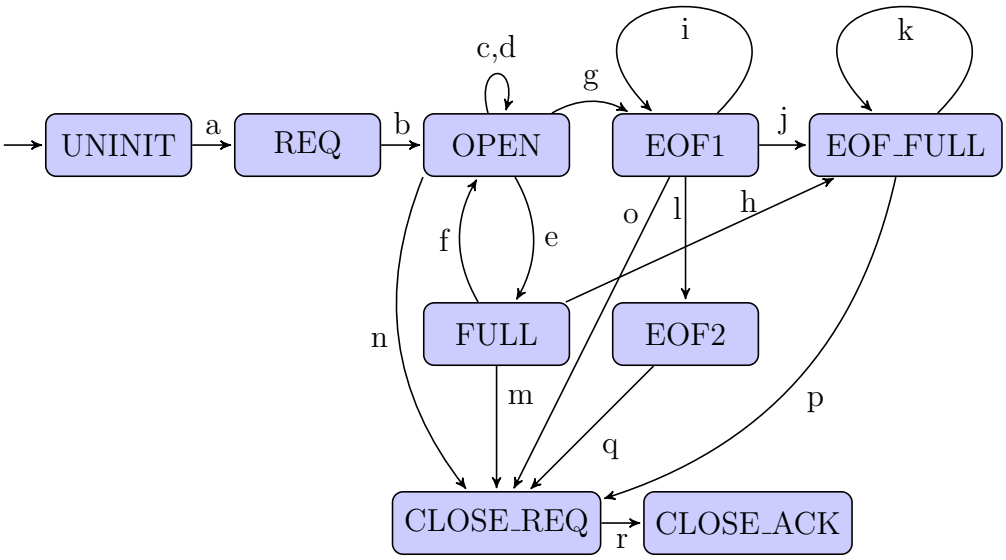
(n) OPEN→CLOSE_REQ: request close

(o) EOF1→CLOSE_REQ: request close

(p) EOF1_FULL→CLOSE_REQ: request close

(q) EOF2→CLOSE_REQ: request close

(r) CLOSE_REQ→CLOSE_ACK: ack close



This question was, of course, subjective. Here are some of the guidelines that we used to assign marks. We assigned points with a bit more leeway than what was written on the assignment sheet (giving higher marks than we could if we stuck strictly to the points distribution). It seems that people did not have enough time to write the answers as requested. So the most important factor was that the solution you provided clearly reflected the points given below:

- Your solution must reflect the fact that two systems are interacting.
- It must be clear that a sequence OPEN REQUEST / OPEN ACKNOWLEDGED has to occur before data can be sent or received.
- It must be clear that a CLOSE REQUEST / CLOSE ACKNOWLEDGED sequence has to occur when a connection is being closed.
- It must be clear that a side can still receive messages, even if it has already sent a SSH_MSH_CHANNEL_EOF.
- The FSM must allow sending and receiving data.