CSCE 747 - Midterm
Name:

This is a 75-minute exam. On all essay type questions, you will receive points based on the quality of the answer - not the quantity.

Make an effort to write legibly. Illegible answers will not be graded and awarded 0 points.

There are a total of 8 questions and 100 points available on the test.

Question 1

Multiple solutions may apply. Select all that are applicable.

1. A test suite that meets a stronger coverage criterion will find any defects that are detected by any test suite that meets only a weaker coverage criterion
   a. True
   b. False

False

2. A test suite that is known to achieve Modified Condition/Decision Coverage (MC/DC) for a given program, when executed, will exercise, at least once:
   a. Every statement in the program.
   b. Every branch in the program.
   c. Every LCSAJ in the program.
   d. Every path in the program.

A, b

3. Possible sources of information for functional testing include:
   a. Requirements Specification
   b. User Manuals
   c. Program Source Code
   d. Domain Experts

A, b, d

4. Category-Partition Testing technique requires identification of:
   a. Parameter characteristics
   b. Representative values
   c. Def-Use pairs
   d. Pairwise combinations

A, b

5. Validation activities can only be performed once the complete system has been built.
    a. True
    b. False

False

6. Statement coverage criterion never requires as many test cases to satisfy as branch coverage criterion.
    a. True
    b. False

False

7. Requirement specifications are not needed for generating inputs to satisfy structural coverage of program code.
    a. True
    b. False

True

8. A system that fails to meet its user's needs may still be:
    a. Correct with respect to its specification.
    b. Safe to operate.
    c. Robust in the presence of exceptional conditions.
    d. Considered to have passed verification.

A, B, C, D

Question 2

Consider the following situation:
After *carefully and thoroughly* developing a collection of requirements-based tests and running your test suite, you determine that you have achieved only 60% statement coverage. You are surprised (and saddened), since you had done a very thorough job developing the requirements-based tests and you expected the result to be closer to 100%.

1. Briefly describe two (2) things that might have happened to account for the fact that 40% of the code was not exercised during the requirements-based tests.
2. Should you, in general, be able to expect 100% statement coverage through thorough requirements-based testing alone (why or why not)?
3. Some structural criteria, such as MC/DC, prescribe obligations that are impossible to satisfy. What are two reasons why a test obligation may be impossible to satisfy?

Suggested Solution:

1. *There are several reasons. The most obvious one being doing a poor job finding the black-box test cases. Since we assume we did a good job, this is not the case.*
    1. *We are missing requirements. The requirements document is incomplete and somewhere along the development of the software these missing requirements have been informally filled in by the development team, but the requirements were never added to the requirements document. Developing black-box tests from an incomplete specification to test a more complete implementation will naturally lead to poor coverage.*
    2. *We have large amounts of dead or inactivated code. The software may have gone through several major changes and code needed for an earlier version is now not used. This code will not be covered. Also, debugging code deactivated through some global variable will not be covered. Furthermore, any malicious code may not get covered. There are many reasons why unneeded or undesirable code might make it into the software—this code is likely to not be covered with your black-box tests.*
    3. *There may be valid optimizations in the code. The programmers might have done some very smart things in terms of optimizing the code, but this leads to a potentially large code base that is only used in various special cases. For example, the programmer might have used some lookup tables for various trigonometric functions (implemented as a switch statement) instead of the built in trigonometric functions. With black box testing you are unlikely to cover much of those switch statements.*
2. *In general there will be optimizations, debug code, exception handling, etc. in the program that the black-box testing is quite unlikely to reveal. Thus it is highly unlikely that we will get close to 100% through black-box testing alone.*

3. *Impossible combination of conditions, defensive programming, unreachable/unused code*

Question 3

In class we discussed the importance of defining a test case for each requirement. What are the two primary benefits of defining this test case?

Suggested Solution:

1. *A test case will greatly help us in the integration testing phase. Now our testing groups can start defining test cases and procedures early and be ready when the system is coming on-line.*
2. *Test cases force us to write testable (thus, pretty good) requirements. If a requirement is not testable, we simply cannot write a test case.*

Question 4

The airport connection check is part of a travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary.

validConnection(Flight arrivingFlight, Flight departingFlight) returns ValidityCode.

A Flight is a data structure consisting of:

- A unique identifying flight code (string, three characters followed by four numbers).
- The originating airport code (three character string).
- The scheduled departure time (in universal time).
- The destination airport code (three character string).
- The scheduled arrival time (in universal time).

There is also a flight database, where each record contains:

- Three-letter airport code (three character string).
- Airport country (two character string).
- Minimum connection time (integer, minimum number of minutes that must be allowed for flight connections).

ValidityCode is an integer with value 0 for OK, 1 for invalid airport code, 2 for a connection that is too short, 3 for flights that do not connect (arrivingFlight does not land in the same location as departingFlight), or 4 for any other errors (malformed input or any other unexpected errors).

In order to design requirements-based test cases, perform category-partition testing using this specification for the validConnection function.

1. Identify parameters.
2. Identify categories for each parameter.
3. Identify representative values (choices) for each category.

Suggested Solution:
The following are the essential input partitions. Others are possible.

Recall the lectures on requirements-based testing. The approximate process of taking a requirement and writing tests is to:

1. *Refine the requirement so that it is testable.*
2. *As a requirement is just a property of the software, you usually can't directly "test the requirement." Instead, you need to use a function of the software and show that the requirement holds during that execution. So, the next step is to identify the independently testable features of the software.*
3. *Now that you have those, you need to look at the parameters and figure out which inputs to pass in. You cannot exhaustively test a function, there are too many possible parameters.* ***So, instead, you partition the input domain into representative regions.*** *If you try inputs from each of these areas, you are more likely to trigger a fault than through random testing alone. We discussed some methods of doing so during Lecture 8.*
4. *Once you have the inputs partitioned, you can form abstract test cases for which you can transform into actual test cases by coming up with concrete input values from the identified partitions.*

In this question, you have been given a testable feature, so you have been asked to perform the activity from Step #3 above - given the parameters, partition the inputs into representative regions. This function has two explicit inputs - an arriving flight and a departing flight - and an implicit input - an airport connection database.

A flight is a complex data structure containing several fields, so for each field, you need to partition the inputs. Remember that the function's parameters may influence each other (testing this function requires considering both the arriving and departing flight's field values as well as what is in the database), so that may influence the partitions.

Parameter: Arriving flight

Flight code:
malformed
not in database
valid

Originating airport code:
malformed
not in database
valid city

Scheduled departure time:
syntactically malformed
out of legal range

legal

Destination airport (transfer airport - where connection takes place):
malformed
not in database
valid city

Scheduled arrival time (tA):
syntactically malformed
out of legal range
legal

<u>Parameter: Departing flight</u>

Flight code:
malformed
not in database
valid

Originating airport code:
malformed
not in database
differs from transfer airport
same as transfer airport

Scheduled departure time:
syntactically malformed
out of legal range
before arriving flight time (tA)
between tA and tA + minimum connection time (CT)
equal to tA + CT
greater than tA + CT

Destination airport code:
malformed
not in database
valid city

Scheduled arrival time:
syntactically malformed
out of legal range
legal

<u>Parameter: Database record</u>

This parameter refers to the database time record corresponding to the transfer airport.

Airport code:

malformed
not found in database
valid

Airport country:
malformed
invalid
valid

Minimum connection time:
not found in database
invalid
valid

Question 5

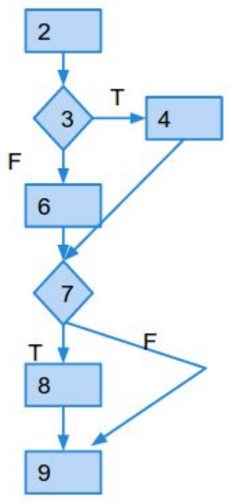
For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage. (Input output pairs will be fine.)
c. Develop test input that will provide branch coverage.
d. Develop test input that will provide full-path coverage.
e. Modify the program to introduce a fault so that you can demonstrate that even achieving full path coverage will not guarantee that we will reveal all faults. Please explain how this fault is missed in your example.

```
int findMax(int a, int b, int c)
{
   int temp;
   if (a>b)
      temp=a;
   else
      temp=b;

   if (c>temp)
      temp = c;
   return temp;
}
```

Suggested Solution



```
1. int findMax(int a, int b, int c) {
2.   int temp;
3.   if (a>b)
4.        temp=a;
5.   else
6.        temp=b;
7.  if (c>temp)
8.        temp = c;
9.  return temp;
10. }
```

a)
b) (3, 2, 4); (2, 3, 4)
c) (3, 2, 4); (3, 4, 1)
d) (4, 2, 5); (4, 2, 1); (2, 3, 4); (2, 3, 1)
e) If we have (a>b+1) in the first condition as opposed to (a>b), the tests in part D will not reveal this flaw. Only a boundary value test will.

Question 6

Identify all DU Pairs in the following code:

```
1.
2.  /* External file hex_values.h defined Hex_Values[128]
3.  * with value 0 to 15 for the legal hex digits
4.  * and value -1 for each illegal digit including special
5.  * characters */
6.
7.  #include “hex_values.h”
8.  /** Translate a string from the CGI encoding to plain
9.  * acsii text. ‘+’ becomes space, %xx becomes byte with hex
10. * value xx, other alphanumeric characters map to themselves
11. * Returns 0 for success, positive for erroneous input.
12. * 1 = bad hexadecimal digit.
13. */
14. int cgi_decode(char *encoded, char *decoded){
15.           char *eptr = encoded;
16.           char *dptr = decoded;
17.           int ok = 0;
18.           while(*eptr){
19.                  char c;
20.                  c = *eptr;
21.
22.                  if(c ==’+’){   /* Case 1: ‘+’ maps to blank*/
23.                         *dptr = ‘ ‘;
24.                  } else if(c == ‘%’){  /* Case 2: ‘%xx’ = char xx*/
25.                         int digit_high = Hex_Values[*(++eptr)];
26.                         int digit_low = Hex_Values[*(++eptr)];
27.                         if(digit_high == -1 || digit_low == -1){
28.                                /* *dptr=’?’ */
29.                                ok = 1; /* Bad return code */
30.                         }else{
31.                                *dptr = 16 * digit_high + digit_low;
32.                         }
33.                  }else{ /*Case 3: All other chars map to themselves*/
34.                         *dptr = *eptr;
35.                  }
36.                  ++dptr;
37.                  ++eptr;
38.           }
39.           *dptr = ‘\0’;
40.           return ok;
41.    }
```

Definitions and Uses:

Variable	Definitions	Uses
*encoded	14	15
*decoded	14	16
*eptr	15, 25, 26, 37	18, 20, 25, 26, 34
eptr	15, 25, 26, 37	15, 18, 20, 25, 26, 34, 37
*dptr	16, 23, 31, 34, 36, 39	
dptr	16, 36	16, 23, 31, 34, 36, 39
ok	17, 29	40
c	20	22, 24
digit_high	25	27, 31
digit_low	26	27, 31
Hex_Values	-	25, 26

Def-Use Pairs:

Variable	DU Pairs
*encoded	(14, 15)
*decoded	(14, 16)
*eptr	(15, 18), (15, 20), (15,25), (15, 34), (25, 26), (26, 37), (37, 18), (37, 20), (37,25), (37, 34)
eptr	(15, 15), (15, 18), (15, 20), (15, 25), (15, 34), (15, 37), (25, 26), (26, 37), (37, 18), (37, 20), (37, 26), (37, 34), (37, 37)
dptr	(16, 16) , (16, 23), (16, 31), (16, 34), (16, 36), (16, 39), (36, 23), (36, 31), (36, 34), (36, 36), (36, 39)
ok	(17, 40), (29, 40)
c	(20, 22), (20, 24)
digit_high	(25, 27), (25, 31)
digit_low	(26, 27), (26, 31)

Question 7

In a directed graph with a designated exit node, we say that a node m post-dominates another node n, if m appears on every path from n to the exit node.

Let us write *m pdom n* to mean that m post-dominates n, and *pdom(n)* to mean the set of all post-dominators of n, i.e., *{m | m pdom n}*.

Answer the following, providing justification for each:

1. Does *b pdom b* hold true for all b?
2. Can both *a pdom b* and *b pdom a* hold true for two different nodes a and b?
3. If both *c pdom b* and *b pdom a* hold true, what can you say about the relationship between c and a?
4. If both *c pdom a* and *b pdom a* hold true, what can you say about the relationship between c and b?

Suggested Solution:

1. *Yes. Trivially, the node b must appear on every path from the exit node to itself. Thus, every node is its own "trivial post-dominator". Thus, pdom is reflexive.*
2. *No, unless a = b (or yes, only if a = b). If a = b, result follows from (a). If a and b are distinct and a pdom b, every path to the exit must pass through a after reaching b. If b pdom a is true, that implies that every path to the exit must pass through b after passing through a. Both cannot be true at once, that would imply there is some path from a to the exit that does not pass through b. Thus, pdom is anti-symmetric.*
3. *c pdom a holds true. If there is a path from a to the exit, b appears on that path because b pdom a. Then, c must appear on the sub-path from b to the exit because c pdom b. Thus, pdom is transitive.*
4. *Either c pdom b or b pdom c holds true. Otherwise, b and c will be distinct nodes and there will be a path from b to the exit that does not pass through c, and a path from c to the exit that does not pass through b. Suppose there is a path from a to the exit. Now, because c pdom a and b pdom a, both c and b must appear on that path.*

Question 8

In class, we discussed various forms of oracles – such as a model, a second implementation, properties, self-checks, a team of experts, etc. Provide a comparative analysis of three different kinds of oracles of your choice, defining what they are and addressing their strengths and weaknesses with respect to key attributes relevant to the verification process (e.g., cost, accuracy, completeness).

Suggested Solution:
Expected-Value Oracle: Exact definition of the expected output given a concrete input. Most common form of oracle.

Self-Check Oracle: A property that must be met by the output, regardless of the value of the output.

Model: A finite-state machine representing the abstract behavior of a function in a variety of situations.

Expected-value oracles are less expensive to design (per-test) than models and self-checks, but only work for one test input at a time. Self-checks and Models can handle a wider variety of situations - potentially any input to the function. However, they may miss a fault that does not violate the property, or meets the same abstract output class as the expected result. Models are the most expensive to design, as they require development of a full model of execution. Models may be inaccurate as well, as they represent simplified versions of a program, and may not reflect enough of the details that are needed for the real execution of the program. Self-Checks may also be inaccurate, as they can only tell when a program violates the specified property - incorrect output could still meet the property.

Cost (per-test) (least to greatest): expected value, self-check, model
Completeness (least to greatest): expected value, self-check, model
Accuracy (least to greatest): self-check, model, expected value