DIT635 - Practice Examination

There are a total of 16 questions on the test. On all essay type questions, you will receive points based on the quality of the answer - not the quantity. If you hand-write any elements, make sure they are legible - illegible answers will not be graded.

(Note: This is longer than the real exam! I want to give you plenty of material to study.)

Question 1 (Warm Up)

Note - Multiple answers may be correct. Indicate all answers that apply.

1. A program may be correct, yet not reliable.
    a. True
    b. False

2. If a system is on an average down for a total 30 minutes during any 24-hour period:
    a. Its availability is about 98% (approximated to the nearest integer)
    b. Its reliability is about 98% (approximated to the nearest integer)
    c. Its mean time between failures is 23.5 hours
    d. Its maintenance window is 30 minutes

3. In general, we need either drivers or mock objects, but not both, when testing a module.
    a. True
    b. False

4. If a temporal property holds for a finite-state model of a system, it holds for any implementation that conforms to the model.
    a. True
    b. False

5. 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

6. 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 combination of condition values in every decision.
    d. Every path in the program.

7. The Category-Partition Testing technique requires identification of:
    a. Testing Choices
    b. Representative Values
    c. Def-Use pairs
    d. Pairwise combinations

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

9. The statement coverage criterion never requires as many test cases to satisfy as branch coverage criterion.
    a. True
    b. False

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

11. 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.

Question 2 (Quality Scenarios)

Consider the software for air-traffic control at an airport (say, GOT). Air traffic control (ATC) is a service provided by ground-based air traffic controllers (the users of this system) who direct aircraft on the ground and through controlled airspace with the help of the software. The purpose of this software is to prevent collisions, organize and expedite the flow of air traffic, and provide information and other support for pilots.

The software offers the following features:

- Monitors the location of all aircraft in a user's assigned airspace.
- Communication with the pilots by radio.
- Generation of routes for individual aircraft, intended to prevent collisions.
- Scheduling of takeoff for planes, intended to prevent potential collisions.
- Alerts of potential collisions based on current bearing of all aircraft.
    - To prevent collisions, ATC applies a set of traffic separation rules, which ensure each aircraft maintains a minimum amount of empty space around it at all times.
    - The route advice can be either of "mandatory" priority (to prevent an imminent collision, pilots should follow this command unless there is a good reason not to) or "advisory" priority (this advice is likely to result in a safe route, but a pilot can choose to ignore it).

You may add additional features or make decisions on how these features are implemented, as long as they fit the overall purpose of the system. In any case, state any assumptions that you make.

Identify one performance, one availability, and one security requirement that you think would be necessary for this software and develop a quality attribute scenario for each.

Question 3 (Quality)

Regarding performance:

1. What is the difference between response time and throughput?
2. Describe a situation where a system could display excellent throughput, but poor response time, and vice versa.

Question 4 (Quality)

You are building a web store that you feel will unseat Amazon as the king of online shops. Your marketing department has come back with figures stating that - to accomplish your goal - your shop will need an **availability** of at least 99%, a **probability of failure on demand** of less than 0.1, and a **rate of fault occurrence** of less than 2 failures per 8-hour work period.

You have recently finished a testing period of one week (seven full 24-hour days). During this time, 972 requests were served to the page. The product failed a total of 64 times. 37 of those resulted in a system crash, while the remaining 27 resulted in incorrect shopping cart totals. When the system crashes, it takes 2 minutes to restart it.

1. What is the rate of fault occurrence?
2. What is the probability of failure on demand?
3. What is the availability?
4. Is the product ready to ship? If not, why not?

Question 5 (System Testing - Category-Partition Method)

(Note - this is a much longer question than would be on the real exam. However, it is a nice detailed example for studying.)

The airport connection check is a high-level function exposed by the API of a travel reservation system. It is intended to check the validity of a single connection between two flights in an itinerary. For example, a traveler may intend to fly from Gothenburg to Los Angeles, but there is a connection through Frankfurt. Therefore, their itinerary is Gothenburg -> Frankfurt (Flight A) and Frankfurt -> Los Angeles (Flight B).

This service will ensure that the connection through Frankfurt is a valid one. For example, if the arrival airport of Flight A differs from the departure airport of Flight B, the connection is invalid. That is, if we pass in two flights, and Flight A arrives in Frankfurt, but Flight B departs from Munich, it is not a valid connection.

Likewise, if the departure time of Flight B is too close to the arrival time of Flight A, the connection is invalid. If Flight A arrives in Frankfurt at 8:00, and Flight B departs at 8:05, there is not sufficient time to complete the customs process and board the flight.

```
validConnection(Flight flightA, Flight flightB)
                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 from the originating airport (in universal time).
- The destination airport code (three character string).
- The scheduled arrival time at the destination airport (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 to be valid).

`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 (Flight A does not land in the same location that Flight B departs from), or 4 for any other errors (malformed input or any other unexpected errors).

Design system test cases using the category-partition method for the `validConnection` function.

1. Identify choices (aspects that you control and that can vary the outcome) for the two input flights and the database.
2. For each choice, identify a set of representative values.

3. Apply ERROR, SINGLE, and IF constraints.
    a. ERROR = This representative value will trigger an error no matter that it is paired with.
    b. SINGLE = This representative value should give an OK response, but we want to make sure we try it once.
    c. IF = This representative value can only be used if a certain value is set for another choice.

Question 6 (System Testing - Combinatorial Interaction Testing)

You are designing system-level tests for a web browser with multiple configuration options. You have extracted the following choices, with the following representative values for each:

Allow Content to Load	**Notify About Pop-Ups**	**Allow Cookies**	**Warn About Add-Ons**	**Warn About Attack Sites**	**Warn About Forgeries**
Allow	Yes	Allow	Yes	Yes	Yes
Restrict	No	Restrict	No	No	No
Block		Block			

The full set of possible test specifications contains 144 options.

Create a covering array of specifications that covers all **pairwise value combinations** in fewer test specifications.

(hint: start with two variables with the most values and add additional variables one at a time)

Question 7 (Exploratory Testing)

Exploratory testing typically is guided by “tours”. Each tour describes a different way of thinking about the system-under-test, and prescribes how the tester should act when they explore the functionality of the system.

1. Describe one of the tours that we discussed in class.
2. Consider a banking website, where a user can do things like check their account balance, transfer funds between accounts, open new accounts, and edit their personal information. Describe three actions you might take during exploratory testing of this system, based on the tour you described above.

Question 8 (Unit Testing)

You are testing the following method:
```
public double max(double a, double b);
```

Devise four executable test cases for this method in the JUnit notation.

Question 9 (Structural Testing)

Consider the following situation: After *carefully and thoroughly* developing a collection of tests based on the requirements and your own intuition, 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?

Question 10 (Structural Testing)

For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage.
c. Develop test input that will provide branch coverage.
d. Develop test input that will provide path coverage.
e. Modify the program to introduce a fault so that you can demonstrate that even achieving path coverage will not guarantee that we will reveal all faults. Please explain how this fault is missed by your test cases.

```
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.  }
```

(Input -> Output pairs will be fine - no need for assertions or writing full JUnit cases)

Question 11 (Structural Testing - Data Flow)

Identify all DU Pairs in the following code, and provide test input that achieves all DU Pairs Coverage over the code (you do not need to provide assertions or full JUnit test cases). Explain which DU Pairs are covered by each test input.

```
1. public int inflections(int[] a, int n) {
2.    int v = 0; // number of inflections
3.    int d = 0; // current run direction (+/-)
4.    while (n > 1) {
5.          n = n - 1;
6.          if ((d * (a[n]-a[n-1])) < 0) // direction change
7.                v = v + 1; // => inflection point
8.          if (a[n] != a[n-1])
9.                d = a[n] - a[n-1]; // record direction
10.   }
11.   return v;
12. }
```

Question 12 (Mutation Testing)

Consider the following function:

```
public void bSearch(int[] A, int value, int start, int end) {
        if (end <= start)
                return -1;
        mid = (start + end) / 2;
        if (A[mid] > value) {
                return bSearch(A, value, start, mid);
        } else if (value > A[mid]) {
                return bSearch(A, value, mid+1, end);
        } else {
                return mid;
        }
}
```

Give an example, with a brief justification, for each of the following kinds of mutants that may be derived from the code by applying mutation operators of your choice. Do not reuse a mutation operator, even if it fits multiple categories.

1. Equivalent Mutant
2. Invalid Mutant
3. Valid, but not Useful Mutant
4. Useful Mutant

Question 13 (Automated Test Generation)

Metaheuristic search techniques can be divided into local and global search techniques.

1. Define what a “local” search and a “global” search is.
2. Contrast the two approaches. What are the strengths and weaknesses of each?
3. Choose one search algorithm and briefly explain how it works. State whether it is a global or local search, and explain why it belongs to that category.

Question 14 (Finite State Verification)

Suppose that finite state verification of an abstract model of some software exposes a counter-example to a property that is expected to hold true for the system. This means that the model can be shown to not satisfy the property.

Briefly describe what follow-up actions would you take, and why you would take them?

Question 15 (Finite State Verification)

Temporal Operators: A quick reference list.

- G p: p holds globally at every state on the path
- F p: p holds at some state on the path
- X p: p holds at the next (second) state on the path
- p U q: q holds at some state on the path and p holds at every state before the first state at which q holds.
- A: for all paths from a state, used in CTL as a modifier for the above properties (AG p)
- E: for some path from a state, used in CTL as a modifier for the above properties (EF p)

Consider a finite state model of a traffic-light controller similar to the one discussed in the homework, with a pedestrian crossing and a button to request right-of-way to cross the road.

State variables:

- **traffic_light: {RED, YELLOW, GREEN}**
- **pedestrian_light: {WAIT, WALK, FLASH}**
- **button: {RESET, SET}**

Initially: **traffic_light = RED, pedestrian_light = WAIT, button = RESET**

Transitions:
pedestrian_light:

- **WAIT → WALK if traffic_light = RED**
- **WAIT → WAIT otherwise**
- **WALK → {WALK, FLASH}**
- **FLASH → {FLASH, WAIT}**

traffic_light:

- **RED → GREEN if button = RESET**
- **RED → RED otherwise**
- **GREEN → {GREEN, YELLOW} if button = SET**
- **GREEN → GREEN otherwise**
- **YELLOW→ {YELLOW, RED}**

button:

- **SET → RESET if pedestrian_light = WALK**
- **SET → SET otherwise**
- **RESET → {RESET, SET} if traffic_light = GREEN**
- **RESET → RESET otherwise**

1. Briefly describe a safety-property (nothing “bad” ever happens) for this model and formulate it in CTL.
2. Briefly describe a liveness-property (something “good” eventually happens) for this model and formulate it in LTL.
3. Write a trap-property that can be used to derive a test case using the model-checker to exercise the scenario “pedestrian obtains right-of-way to cross the road after pressing the button”.

   A trap property is when you write a normal property that is expected to hold, then you negate it (saying that the property will NOT be true). The verification framework will then produce a counter-example indicating that the property actually can be met - including a concrete set of input steps that will lead to the property being true.

Question 16 (Finite State Verification)

Consider a simple microwave controller modeled as a finite state machine using the following state variables:

- Door: {Open, Closed} -- sensor input indicating state of the door
- Button: {None, Start, Stop} -- button press (assumes at most one at a time)
- Timer: 0...999 -- (remaining) seconds to cook
- Cooking: Boolean -- state of the heating element

Formulate the following informal requirements in CTL:

1. The microwave shall never cook when the door is open.
2. The microwave shall cook only as long as there is some remaining cook time.
3. If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.

Formulate the following informal requirements in LTL:

1. It shall never be the case that the microwave can continue cooking indefinitely.
2. The only way to initiate cooking shall be pressing the start button when the door is closed and the remaining cook time is not zero.
3. The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.