DIT636 / DAT560 - Practice Examination

There are a total of 14 questions on the practice exam (**there will be fewer on the real exam - we gave you some extra questions to study with**). On all essay type questions, you will receive points based on the quality of the answer - not the quantity. Write carefully - illegible answers will not be graded.

Question 1 (Warm Up)

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

1. A program may be reliable, yet not robust.
   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. A typical distribution of test types is 40% unit tests, 40% system tests, and 20% GUI/exploratory tests.
   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 test creation technique technique requires identification of:

e. **Choices**
f. **Representative Values**
g. Def-Use pairs
h. 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. Any program that has passed all test cases and has been released to the public is considered which of the following:
    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.

Sample Solution

Requirements should be specific and testable. Scenarios should have single stimuli and specific, measureable system responses

<u>Performance Requirement:</u> Under normal load, displayed aircraft positions shall be updated on a user's display in under 55 ms.

<u>Performance Scenario</u>: Responsiveness

- ***Overview: Check system responsiveness for displaying updated aircraft positions***
- ***System state: System is under normal load (defined as the deployment environment working correctly with less than 500 tracked aircraft).***
- ***Environment state: Less than 500 physical aircraft are in the airspace. All are being tracked successfully.***
- ***External stimulus: 50 Hz update of ATC system display.***
- ***Required system response: radar/sensor values are computed and fused, new position is displayed to the air traffic controller with maximum error of 5 meters.***
- ***Response measure: Fusion and display process completes in less than 45 ms 95% of the time, and in less than 50 ms 99% of the time. There is an absolute deadline of 55 ms.***

<u>Availability Requirement:</u> The system shall be able to tolerate the failure of any single server host, graphics card, display or network link.

<u>Availability Scenario:</u> primary display card fails during screen refresh

- ***Overview: One of the monitor display cards fails during transmission of a screen refresh***
- ***System state: System is working correctly under normal load with no failures.***
- ***Environment state: No relevant details.***
- ***External stimulus: A display card fails.***
- ***Required system response: The display window manager system will detect the failure within 10 ms and route display information through a spare redundant graphics card with no user-discernable change to ATC aircraft display. The graphics card failure will be displayed as an error message at the bottom right corner of the ATC display.***
- ***Response measure: There will be no loss in continuity of visual display and failover with visual warning will complete within 1s.***

<u>Security Requirement:</u> The system shall maintain audit logs of any logins to the ATC database, containing sufficient information to identify an attacker.

<u>Security Scenario:</u> malicious login and audit policy demonstration

- ***Overview: A malicious agent gains access to the flight records database in the ATC.***
- ***System state: The system is working correctly under normal load.***
- ***Environment state: No relevant environmental factors.***

- ***External stimulus: A malicious agent obtains access to the flight records database through password cracking, and downloads flight plans for commercial aircraft.***
- ***Required system response: An audit log will be updated with login and download information to support future prosecution of malicious users.***
- ***Response measure: The system audit contains time, IP address, and related information for the download. This information will assist in identifying and analyzing possible attacks.***

Question 3 (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?

Sample Solution

1. **64/168 hours = 0.38/hour = 3.04/8 hour work day**
2. **64/972 = 0.066**
3. **It was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time. Availability = 99.3%**
4. **No. Availability, POFOD are good. ROCOF is too low.**

Question 4 (System Testing)

Consider a personnel management program that offers an API where, among other functions, a user can apply for vacation time:

```
public boolean applyForVacation (String userID, String startingDate, String endingDate)
```

A user ID is a string in the format “firstname.lastname”, e.g., “gregory.gay”.
The two dates are strings in the format “YYYY-DD-MM”.

The function returns TRUE if the user was able to successfully apply for the vacation time. It returns FALSE if not. An exception can also be thrown if there is an error.

This function connects to a user database. Each user has the following relevant items stored in their database entry:

- User ID
- Quantity of remaining vacation days for the user
- An array containing already-scheduled vacation dates (as starting and ending date pairs)
- An array containing dates where vacation cannot be applied for (e.g., important meetings).

Perform category-partition testing for this function.

1. Identify choices (controllable aspects that can be varied when testing)
2. For each choice, identify representative input values.
3. For each value, apply constraints (IF, ERROR, SINGLE) if they make sense.

You do not need to create test specifications or concrete test cases. For invalid input, **do not** just write “invalid” - be specific. If you wish to make any additional assumptions about the functionality of this method, state them in your answer.

Sample Solution

Note that your solution may not match this exactly, but should contain but should be detailed and account for normal and error scenarios.

- Choice: Value of userID
  - Existing user
  - Non-existing user **[error] [property not-exist]**
  - Null **[error]**
  - Malformed user ID (not in format “firstname.lastname”) **[error]**
- Choice: Value of starting date
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Null **[error]**
  - Malformed date (not in format “YYYY-MM-DD”) **[error]**
- Choice: Value of ending date
  - Valid date
  - Date before the current date **[error]**
  - Current date **[single]**
  - Date before the starting date **[error]**
  - Date same as the starting date **[single]**
  - Null **[error]**
  - Malformed date (not in format “YYYY-MM-DD”) **[error]**
- Choice: Remaining vacation time for the userID
  (Note: We are assuming the database schema prevents storing malformed/invalid values)
  - 0 days remaining
  - 1 day remaining, 1 day applied for **[single]**
  - Number of days remaining < number applied for
  - Number of days remaining = number applied for **[single]**
  - Number of days remaining > number applied for
  - User does not exist **[if not-exist]**
- Choice: Conflicts with vacation time
  (Note: We are assuming the database schema prevents storing malformed/invalid date ranges)
  - No conflicts with already-scheduled vacation or banned dates
  - Banned date(s) fall within the starting and ending dates applied for
  - Starting date falls within already-scheduled vacation time
  - Ending date falls within already-scheduled vacation time
  - Already-scheduled vacation time falls within starting and ending dates applied for
  - The starting and ending dates fall within already-scheduled vacation time
  - User does not exist **[if not-exist]**

Question 5 (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. Those actions must relate to the tour.

Sample Solution

1. **The supermodel tour is focused on testing the GUI of the application. It is not as concerned with functional correctness as the other tours (e.g., that the correct data is displayed on the screen). Rather, it is more concerned with the visual appearance of the GUI and whether it is correct. It focused on whether graphical elements display in the correct locations and without "glitches" (e.g., rendering errors, size or rotation issues). It also examines timing aspects of the GUI, such as how long it takes for a mouse cursor to move, text to update on the screen, for new screens to be drawn, etc. This tour can also look for typos in displayed text, or for usability issues (e.g., suggestions on how to make the GUI easier for new users to learn how to work with). Accessibility standards can be checked during this tour as well (e.g., color blindness, dyslexia, screen reader compatibility).**
2. **For the banking website, you might examine:**
    a. **Click on a drop down menu and ensure that the menu displays quickly, that all required items are present and displayed correctly, and that the menu does not cause any graphical issues when it appears over other on-screen items.**
    b. **When an account is selected, ensure that account information is displayed on the screen, that it is displayed in the correct locations, and that this information is easy for a user to see if they are searching for it on the screen (e.g., that good font, color, and size choices are made). Check that screen readers can read this to a blind person.**
    c. **When the user goes to edit personal information, ensure that the existing information is displayed on the screen and that edited segments are refreshed and displayed to the user correctly.**

Question 6 (Unit Testing)

Account
- name - personnummer - balance
Account (name, personnummer, Balance) withdraw (double amount) deposit (double amount) changeName(String name) getName() getPersonnummer() getBalance()

You are testing the class depicted to the left.

Write JUnit-format test cases to do the following:

1. Create a test case that checks a normal usage of the methods of this class.
2. Create two test cases reflecting either error-handling scenarios or quality attributes (e.g., performance or reliability).

Sample Solution

Withdraw money, verify balance (normal functionality).

```
@Test
public void testWithdraw_normal() {
   // Setup
   Account account = new Account("Test McTest", "19850101-1001", 48.5);
   // Test Steps
   double toWithdraw = 16.0; //Input
   account.withdraw(toWithdraw);
   double actual = account.getBalance();
   double expectedBalance = 32.5; // Oracle
   assertEquals(expected, actual); // Oracle
}
```

Withdraw more than is in balance.
(should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_moreThanBalance() {
   // Setup
   Account account = new Account("Test McTest", "19850101-1001", 48.5);
   // Test Steps
   double toWithdraw = 100.0; //Input
   Throwable exception = assertThrows(
      () -> { account.withdraw(toWithdraw); } );
   assertEquals("Amount 100.00 is greater than balance 48.50",
            exception.getMessage()); // Oracle
}
```

Withdraw a negative amount.
(should throw an exception with appropriate error message)

```
@Test
public void testWithdraw_negative() {
   // Setup
   Account account = new Account("Test McTest", "19850101-1001", 48.5);
   // Test Steps
   double toWithdraw = -2.5; //Input
   Throwable exception = assertThrows(
      () -> { account.withdraw(toWithdraw); } );
   assertEquals("Cannot withdraw a negative amount: -2.50",
            exception.getMessage()); // Oracle
}
```

Question 7 (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?

Sample 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 requirements-based testing you are unlikely to cover much of those switch statements.***
    4. ***There may be support code that was not covered in the requirements (e.g., operating system or database interfacing, file I/O, user interface code).***
2. ***In general there will be optimizations, debug code, support 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. ***The criterion may call for an impossible combination of conditions within a decision statement. You may have also performed defensive programming, resulting in error-handling code that cannot actually be triggered. In addition, there may be unreachable or unused code that cannot be called directly or reached through normal execution paths.***

Question 8 (Structural Testing)

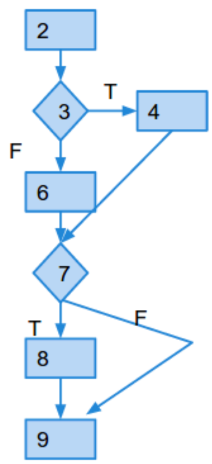
For the following function,

a. Draw the control flow graph for the program.
b. Develop test input that will provide statement coverage. For each test, note which lines are covered.
c. Develop test input that will provide branch coverage. For each test, note which branches are covered. You may reuse input from the previous problem.
d. Develop test input that will provide path coverage. For each test, note which paths are covered. You may reuse input from the previous problem.
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.   }
```

(Just including test input is sufficient - you do not need to write full JUnit cases)

Sample 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) covers lines 2, 3, 4, 7, 8, 9
(2, 3, 4) covers lines 2, 3, 6, 7, 8, 9

c) (3, 2, 4) covers branches 3T, 7T
(3, 4, 1) covers branches 3F, 7F

d) (4, 2, 5) covers path 3T, 7T
(4, 2, 1) covers path 3T, 7F
(2, 3, 4) covers path 3F, 7T
(2, 3, 1) covers path 3F, 7F

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 9 (Structural Testing - Data Flow)

The following function returns true if you can partition an array into one element and the rest, such that this element is equal to the product of all other elements excluding itself.
For example:

- **canPartition([2, 8, 4, 1])** returns true (8 = 2 * 4 * 1)
- **canPartition([-1, -10, 1, -2, 20])** returns false.
- **canPartition([-1, -20, 5, -1, -2, 2])** returns true (-20 = -1 * 5 * -1 * -2 * 2)

```
public static boolean canPartition(int[] arr) {
      Arrays.sort(arr);
      int product = 1;
      if ((Math.abs(arr[0]) >= arr[arr.length-1]) || arr[0] == 0) {
          for (int i = 1; i < arr.length; i++){
              product *= arr[i];
          }
          return arr[0] == product;
      } else{
         for (int i = 0; i < arr.length-1; i++){
             product *= arr[i];
         }
         return arr[arr.length-1] == product;
      }
  }
```

1. Identify the def-use pairs for all variables.
2. Identify test input that achieves all def-use pairs coverage.

Note: You may treat arrays as a single variable for purposes of defining DU pairs. This means that a definition to **arr[0]** or to array **arr** are both definitions of the same variable, and references to **arr[0]** or **arr.length** are both uses of the same variable.

Sample Solution

1. DU Pairs

arr	(1, 2), (2, 4), (2, 5), (2, 6), (2, 8), (2, 10), (2, 11), (2, 13)
product	(3, 6), (6, 6), (3, 8), (6, 8), (3, 11), (11, 11), (11, 13)
i	(5, 5), (5, 6), (10, 10), (10, 11)

2. Test Input

Input	**Additional DU Pairs Covered**
`[2, 8, 4, 1]`	arr: (1, 2), (2, 4), (2, 10), (2, 11), (2, 13) product: (3, 11), (11, 11), (11, 13) i: (10, 10), (10, 11)
`[-1, -10, 0, 10]`	arr: (2, 5), (2, 6), (2, 8) product: (3, 6), (6, 6), (6, 8) i: (5, 5), (5, 6)
`[0]`	arr: (3, 8)

Question 10 (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

Sample Solution

1. **Equivalent Mutant - SES (end block shift) - Result will be the same.**

```
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;
}
```

2. **Invalid Mutant - SDL (statement deletion) - Will not compile.**

```
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;
	}
}
```

3. **Valid, but not Useful - ROR (relational operator replacement) - will almost always fail if method is called correctly (as long as end!=start).**

```
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;
        }

}
```

4. **Useful Mutant - CRP (constant-for-constant replacement) - Will not always fail. Requires input that triggers that specific else if, and may still return the right result as long as it doesn't skip the correct entry.**

```
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 + 2, end);
        } else {
                return mid;
        }

}
```

Question 11 (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 you would take, and why you would take them.

Sample Solution

This tells us that a property we expect to hold is not held by the model. This implies one of the following:

- **There is an issue with the model. The model is made by interpreting the requirements, and there could be a mistake in the model (fault in the model code, bad assumptions, incorrect interpretation of requirements).**
- **There is an issue with the property. The property may not say what you intended it to say. It can be difficult to formulate a property in temporal logic.**
- **There is an issue with your requirements. The requirement may be incorrect, unclear, or incomplete.**

The action you take depends on which of the above is true. You should look at each angle, and find the source of the problem. If the model is incorrect, you should locate and correct the fault. If the property is incorrect, it should be reformulated. If the requirement is incorrect, it should be reformulated - then the property must also be rewritten to match. Fixing the requirement may also require updating the model as well or updating related requirements.

Question 12 (Finite State Verification)

Temporal Operators: A quick reference list. p is a Boolean predicate or atomic variable.

- G p: p holds globally at every state on the path from now until the end
- F p: p holds at some future state on the path (but not all future states)
- X p: p holds at the next 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 reaching out from a state, used in CTL as a modifier for the above properties (AG p)
- E: for one or more paths reaching out from a state (but not all), used in CTL as a modifier for the above properties (EF p)

An LTL example:

- G (MESSAGE_SENT -> F (MESSAGE_RECEIVED))
- It is always true (G), that if the message is sent (property MESSAGE_SENT is true), then at some point after it is sent (F), the message will be received (property MESSAGE_RECEIVED will become true).
    - More simply: A sent message will always be received eventually.

A CTL example:

- EG (WIND -> AF (RAIN))
- There is a potential future where it is a certainty (EG) that - if there is wind (property WIND is true) - it will always be followed eventually (AF) by rain (property RAIN will become true).
    - More simply: There is some probability that wind will inevitably lead to eventual rain, but we have not established this fact for certain.

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.

Sample Solution

1. **AG (pedestrian_light = walk -> traffic_light != green)**
   The pedestrian light cannot indicate that I should walk when the traffic light is green. This is a safety property. We are saying that something should NEVER happen.

2. **G (traffic_light = RED & button = RESET -> F (traffic_light = green))**
   If the light is red, and the button is reset, then eventually, the light will turn green. This is a liveness property, as we assert that something will eventually happen, but we do not know how long it will take.

3. **First, we should formulate the property in a temporal logic, than translate into a trap property:**
   G (button = SET -> F (pedestrian_light = WALK))
   This states that, no matter what happens, if the button is pressed, then eventually the pedestrian light will signal that I can cross the street. This is a liveness property (again, we do not know how long it will take).

   A trap property takes a property we know to be true (like this), then negates it. By negating it, we assert that this property is NOT true. The negated form is:
   G !(button = SET -> F (pedestrian_light = walk))

   Because it is actually true, the model checker gives us a counter-example showing one concrete scenario where the property is true. This is a test case we can use to test our real program.

Question 13 (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.

Sample Solution

CTL:

1. **AG (Door = Open -> !Cooking)**
2. **AG (Cooking -> Timer > 0)**
3. **AG (Button = Stop & !Cooking -> AX (Timer = 0))**

LTL:

1. **G (Cooking -> F (!Cooking))**
2. **G (!Cooking U ((Button = Start & Door = Closed) & (Timer > 0)))**
3. **G ((Cooking & Timer > 0) -> X (((Cooking | (!Cooking & Button = Stop)) | (!Cooking & Door = Open)))**