DIT635 - Practice Final

There are a total of 15 questions and 100 points available on the test. 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.

Question 1

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 mock objects or drivers 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. Self-check oracles (assertions) do not require the expected output for judging whether a program passed or failed a test.
    a. **True**
    b. False

6. Object-oriented design and implementation typically have an impact on verification such that OO specific approaches are required for:
    a. **Unit Testing**
    b. **Integration Testing**
    c. System Testing
    d. Acceptance Testing

7. 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**

8. 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.

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

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

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

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

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

14. 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.

Problem 2

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.

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

Performance Requirement: Under normal load, displayed aircraft positions shall be updated at least every 50 ms.

Performance Scenario: 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 aircraft currently being tracked.***
- ***External stimulus: 50 Hz update of ATC system display.***
- ***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 50 ms.***

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

Availability Scenario: ***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: display card fails***
- ***Required system response: display window manager system will detect failure within 10 ms and route display information through spare redundant graphics card with no user-discernable change to ATC aircraft display. Graphics card failure will be displayed as error message at bottom right hand of ATC display.***
- ***Response measure: no loss in continuity of visual display and failover with visual warning completes within 1s.***

Security Requirement: ***The system shall maintain audit logs of any logins to the ATC database.***

Security Scenario: ***malicious login***

- ***Overview: a malicious agent gains access to flight records database in the ATC.***
- ***System state: System is working correctly under normal load.***
- ***Environment state: No relevant environmental factors.***
- ***External stimulus: malicious agent obtains access to the flight records database through password cracking, and downloads flight plans for commercial aircraft.***
- ***Required system response: audit log contains login and download information to support future prosecution of user.***
- ***Response measure: system audit contains time, IP address, and related information for the download.***

Problem 3

Regarding performance:

1. What is the difference between response time and throughput?

Response time is from the client's perspective: how long does it take to service my request? Throughput is from the server's perspective. How many requests can be processed in a given time period?

2. Describe a situation where a system could display excellent throughput but poor response time and vice versa.

A system may involve several processors working in tandem to solve a particular problem. It may therefore be able to process very large volumes of transactions (high throughput) due to partitioning the problem into segments that are handled sequentially, while still exhibiting poor response time (each segment takes time t, with number of segments s, so the total response time is t*s).

Instead, imagine a single processor system that processes requests sequentially. If there are few requests, it will have better response time than the pipelined system because there is no latency in servicing the request. However, it will have very poor throughput under heavy load.

Question 4

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?

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. How would you improve it?**

Question 5

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.

Here are some examples: Please also explain your tests.

```
@Test
  public void aLarger() {
    double a = 16.0;
    double b = 10.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertTrue("should be larger", actual>b);
    assertEquals(expected, actual);
  }
@Test
  public void bLarger() {
    double a = 10.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertThat("b should be larger", b>a);
    assertEquals(expected, actual);
  }
@Test
  public void bothEqual() {
    double a = 16.0;
    double b = 16.0;
    double expected = 16.0;
    double actual = max(a,b);
    assertEquals(a,b);
    assertEquals(expected, actual);
  }
@Test
  public void bothNegative() {
    double a = -2.0;
    double b = -1.0;
    double expected = -1.0;
    double actual = max(a,b);
    assertTrue("should be negative",actual<0);
    assertEquals(expected, actual);
  }
```

Question 6

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?

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 7

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?

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 8

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. For instance, if the arrival airport of Flight A differs from the departure airport of Flight B, the connection is invalid. Likewise, if the departure time of Flight B is too close to the arrival time of Flight A, the connection is invalid.

```
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 (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.

Recall the lecture on functional testing. The approximate process of taking the requirements and writing tests is to:

1. ***Refine the requirements so that they are 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. ***For each independently testable feature, you need to identify the parameters. These can be explicit (passed into the function) or implicit (environmental factors that influence the outcome of the function.***
4. ***Each parameter can be manipulated in many ways through testing. For each***

parameter, you must identify categories of input that can be chosen. These are things that you can control when you test, whether explicit variable values or factors under your control. For example, if an input is a data structure, the categories might include each field of that data structure that could influence the outcome. If that data structure is serialized from a file, then you can also control factors like whether that file exists, is corrupted, and so on.

5. ***You cannot exhaustively test a function, there are too many possible parameters. So, instead, you partition the input domain into representative regions (types) of input. For each category, identify representative input values for that category. If you try inputs from each of these regions, you are more likely to trigger a fault than through random testing alone. We discussed some methods of performing this partitioning in class.***
6. ***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 exercise, you have been given a testable feature, so you have been asked to perform the activity from Steps #3 - 5 above - identify the parameters, split the parameters into controllable input categories, then partition the categories into representative values.

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, each of those fields represents a controllable input category.

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 the representative values must reflect how multiple categories and variables can interact.

<u>Parameter: Arriving flight</u>

Flight code:

- ***Malformed (string that does not follow stated formatting convention)***
- ***not in database***
- ***valid***

Originating airport code:

- ***malformed (not a three-letter string)***
- ***not in database***
- ***valid city***

Scheduled departure time:

- ***malformed (not following formatting convention)***
- ***out of legal range (not a valid time)***
- ***legal***

Destination airport (transfer airport - where connection takes place):

- ***malformed (not a three-letter string)***
- ***not in database***
- ***valid city***

Scheduled arrival time (tA):
- ***malformed (not following formatting convention)***
- ***out of legal range (not a valid time)***
- ***legal***

<u>Parameter: Departing flight</u>

Flight code:
- ***Malformed (string that does not follow stated formatting convention)***
- ***not in database***
- ***valid***

Originating airport code:
- ***Malformed (not a three-letter string)***
- ***not in database***
- ***differs from transfer airport***
- ***same as transfer airport***

Scheduled departure time:
- ***malformed (not following formatting convention)***
- ***out of legal range (not a valid time)***
- ***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 a three-letter string)***
- ***not in database***
- ***valid city***

Scheduled arrival time:
- ***malformed (not following formatting convention)***
- ***out of legal range (not a valid time)***
- ***legal***

<u>Parameter: Database record</u>

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

Airport code:
- ***Malformed (not a three-letter string)***
- ***blank***
- ***valid***

Airport country:

- ***Malformed (non-string)***
- ***blank***
- ***Invalid (not a country)***
- ***valid***

Minimum connection time:

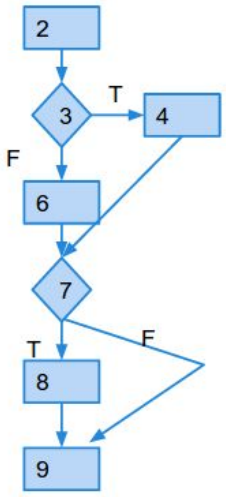
- ***malformed (not following formatting convention)***
- ***blank***
- ***invalid***
- ***valid***

Question 9

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.

```
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.  }
```




```
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 10

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 11

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?

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 12

Consider the following function:

```
bSearch(A, value, start, 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, even if it fits multiple categories.

1. Equivalent Mutant

```
bSearch(A, value, start, 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;
}
```

SES - End Block Shift

2. Invalid Mutant

```
bSearch(A, value, start, 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;
        }
}
```

SDL - Statement Deletion

3. Valid, but not Useful

```
bSearch(A, value, start, 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;
        }

}
```

ROR - Relational Operator Replacement

4. Useful Mutant

```
bSearch(A, value, start, 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;
```

```
        }
}
```

CRP - Constant for Constant Replacement

Question 13

Suppose that finite state verification of an abstract model of some software exposes a counter-example to a property that is expected to hold for true for the system. Briefly describe what follow-up actions would you take and why?

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 code, bad assumptions, incorrect interpretation).**
- **There is an issue with the property. The property may not say what you intended it to say.**
- **There is an issue with your requirements. Two requirements may contradict, or a requirement may be written incorrectly.**

The action you take depends on which is true. You should look at each angle, and find the source of the problem.

Question 14

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”.

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.

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.

   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 15

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.

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

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.

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)))**