Final Review

CSCE 747 - Lecture 26 - 04/26/2018

Final Exam

- Thursday, May 3rd, 9 - 11:30 AM
  - The usual room
- Final is focused on post-midterm material.
- Practice exam online.
  - Let's go over it.
  - Recommended Practice:
    - I'll ask question. You pause and answer. Resume, and listen to my answer.

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

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

Question 1

1. In general, we need either stubs or drivers but not both, when testing a module.
   a. True
   b. False

2. Which of the following may be logically inferred from the post-condition of a sorting routine, sort(array, size) that sorts elements in ascending order?
   a. size > 0
   b. ∃ i, j, 0 <= i < j < size : array[i] = array[j]
   c. ∀ i, j, 0 <= i < j < size : array[i] < array[j]
   d. ∀ i, j, 0 <= i < j < size : array[i] <= array[j]

Question 1

1. In general, we need either stubs or drivers but not both, when testing a module.
   a. True
   b. False

2. Which of the following may be logically inferred from the post-condition of a sorting routine, sort(array, size) that sorts elements in ascending order?
   a. size > 0
   b. ∃ i, j, 0 <= i < j < size : array[i] = array[j]
   c. ∀ i, j, 0 <= i < j < size : array[i] < array[j]
   d. ∀ i, j, 0 <= i < j < size : array[i] <= array[j]

Question 1

1. 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
2. Self-check oracles do not require the expected output for judging whether a program passed or failed a test.
   a. True
   b. False
3. 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

Question 1

1. 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
2. Self-check oracles do not require the expected output for judging whether a program passed or failed a test.
   a. True
   b. False
3. 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

Question 2

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

Define what a “local” search and a “global” search is. Contrast the two approaches. What are the strengths and weaknesses of each?

Question 2 - Answer

Local Search:

- Formulates one solution.
- Attempts to improve it by making a small change (looking around the local neighborhood).

Global Search:

- Tries multiple solutions at once.
- More freely samples from the whole space (making random guesses, large changes, discarding existing solutions)

Question 2 - Answer

Local Search:

- Very fast, easy to implement, easy to understand
- Depend strongly on initial guess

Global Search:

- Harder to implement and slower (more complex)
- Do not become stuck in local optima

Question 3

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;
    }
}
```

1. Create an equivalent mutant.

Question 3

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;
    }
}
```

1. Create an equivalent mutant.

```
    } else if (value > A[mid]) {
        return bSearch(A, value,
    mid+1, end);
    } else {
    }
    return mid;
}
```

SES - End Block Shift

Question 3

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;
    }
}
```

1. Create an invalid mutant.

Question 3

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;
    }
}
```

1. Create an invalid mutant.

```
    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

Question 3

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;
    }
}
```

1. Create a valid-but-not-useful mutant.

Question 3

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;
    }
}
```

1. Create a valid-but-not-useful mutant.

```
bSearch(A, value, start, end) {
    if (end > start)
        return -1;
    mid = (start + end) / 2;
```

ROR - Relational Operator Replacement

Question 3

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;
    }
}
```

1. Create a useful mutant.

Question 3

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;
    }
}
```

1. Create a useful mutant.

```
    } else if (value > A[mid]) {
        return bSearch(A, value,
    mid+2, end);
    } else {
        return mid;
    }

}
```

CRP - Constant for Constant Replacement

Question 4

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?

Question 4 - Answer

Tells us one of the following is an issue:

- The model
  - Fault in the model, bad assumptions, incorrect interpretation of requirements
- The property
  - Property not formulated correctly.
- The requirements
  - Contradictory or incorrect requirements.

Question 5

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.

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the rate of fault occurrence?

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the rate of fault occurrence?

64/168 hours = 0.38/hour = 3.04/8 hour work day

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the probability of failure on demand?

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the probability of failure on demand?

64/972 = 0.066

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the availability?

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What is the availability?

It was down for (37*2) = 74 minutes out of 168 hours = 74/10089 minutes = 0.7% of the time. Availability = 99.3%

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What additional information would you need to calculate the mean time between failures?

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- What additional information would you need to calculate the mean time between failures?

Timestamps of when the system went down.

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- Is the product ready to ship? If not, why not?

Question 5

Want: **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.

Currently: 972 requests.. The product failed a total of 64 times (37 crashes, 27 incorrect computations). It takes 2 minutes to restart.

- Is the product ready to ship? If not, why not?

No. Availability, POFOD are good. ROCOF is too low. How would you improve it?

Question 6

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
- E: for some path from a state

Question 6

Traffic-light controller, 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**

Question 6

State variables:

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

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

Formulate a safety property in CTL.

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**

Question 6

State variables:

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

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

AG (pedestrian_light = walk -> traffic_light != green)

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**

Question 6

State variables:

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

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

Formulate a liveness property in LTL.

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**

Question 6

State variables:

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

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

G (traffic_light = RED & button = RESET -> F (traffic_light = green))

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**

Question 6

State variables:

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

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

Write a trap-property that can be used to derive a test case to exercise the scenario "pedestrian obtains right-of-way to cross the road after pressing the button".

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**

Question 6

State variables:

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

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

Property in temporal logic:
G (button = SET -> F (pedestrian_light = WALK))

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**

Question 6

State variables:

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

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

Negate to get trap property:
G !(button = SET -> F (pedestrian_light = WALK))

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**

Question 7

```
int sum(int arr[], int n){
    int s = 0;
    while (n > 0){
        n = n - 1;
        s = s + arr[n];
    }
    return s;
}
```

What are the pre and post-conditions of this method?

Question 7

```
int sum(int arr[], int n){
    int s = 0;
    while (n > 0){
        n = n - 1;
        s = s + arr[n];
    }
    return s;
}
```

(n = N, N = |arr|, N >= 0)

(s = S, S = $\sum_{i=0}^{N-1}$ arr[i])

What are the pre and post-conditions of this method?

Question 7

```
int sum(int arr[], int n){
    int s = 0;
    while (n > 0){
        ...
    }

    return s;
}
```

(n = N, N = |arr|, N >= 0)

(n = N, N = |arr|, N >= 0, s = 0)

If loop does not execute: (n = N, N = 0, |arr| = 0, s = 0)

(s = S, S = $\sum_{i=0}^{N-1}$ arr[i])

For each line of code, derive the state predicates that would be collected by symbolic execution.

Question 7

```
int sum(int arr[], int n){
    int s = 0;
    while (n > 0){

        n = n - 1;

        s = s + arr[n];

    }
    return s;
}
```

(n = N, N = |arr|, N >= 0)

(n = N, N = |arr|, N >= 0, s = 0)

On first entry: (n = N, N = |arr|, N > 0, s = 0)

On first entry: (n = N, N = N - 1, N >= 0, s = 0)

First: (n = N, N >= 0, s = S, S = S + arr[N])

If loop does not execute: (n = N, N = 0, |arr| = 0, s = 0)

(s = S, S = $\sum_{i=0}^{N-1} arr[i]$)

Question 7

```
    while (n > 0){

        n = n - 1;

        s = s + arr[n];

    }
    return s;
}
```

On first entry: (n = N, N = |arr|, N > 0, s = 0)

(n = N, N > 0, s = S, S= $\sum_{i=N}^{|arr|} arr[i]$)

On first entry: (n = N, N = N - 1, N >= 0, s = 0)

(n = N, N = N - 1, N >= 0, s = S, S= $\sum_{i=N}^{|arr|} arr[i]$)

First: (n = N, N >= 0, s = S, S = S + arr[N])

(n = N, N >= 0, s = S, S= $\sum_{i=N}^{|arr|} arr[i]$)

If loop does not execute: (n = N, N = 0, |arr| = 0, s = 0)

(n = N, N = 0, s = S, S= $\sum_{i=N=0}^{|arr|} arr[i]$)

(s = S, S = $\sum_{i=0}^{N-1} arr[i]$)

Question 8

Search-based test generation generally does not use coverage criteria directly in generation, but instead makes use of a scoring function to determine how clost tests are to achieving the current generation goal (such as coverage of the criterion). This is called the fitness function, or objective function.

Explain the fitness function formulation used by search-based generation to achieve branch coverage of a program. It may be helpful to come up with a code example.

Question 8 - Answer

- Normal measurement: Branches Covered / Total Branches
  - Not a good fitness function, as it does help the search find better tests.
  - Better fitness functions offer a *distance* from the target.

Question 8 - Answer

- Approach Level + Branch Distance
  - Approach Level: Count of the target branch's control-dependent nodes not yet executed.
  - Branch Distance: How close the targeted branch was to being taken.

Question 8 - Answer

- Instead of raw coverage, use the branch distance and approach level:

  fitness(s,b) = AL(s,b) + normalize(BD(s,b))

- Approach level - count of the branch's control-dependent nodes not yet executed.
- Branch distance - if the other branch is taken, measure how close the target branch was from being taken.

Question 8 - Answer

```
if(x < 10){ // Node 1
    // Do something.
}else if (x == 10){ // Node 2
    // Do something else.
}
```

- Goal, true branch of Node 2.
- If x<10 is true, approach level = 1
- If x==10 is reached, approach level = 0

Question 8 - Answer

```
if(x < 10){ // Node 1
    // Do something.
}else if (x == 10){ // Node 2
    // Do something else.
}
```

- Goal, true branch of Node 2.
- If x==10 evaluates to false, branch distance = (abs(x-10)+k).
- Closer x is to 10, closer the branch distance.

Question 9

Microwave controller

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

In CTL:

The microwave shall never cook when the door is open.

Question 9

Microwave controller

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

In CTL:

The microwave shall never cook when the door is open.

AG (Door = Open -> !Cooking)

Question 9

Microwave controller

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

In CTL:
The microwave shall cook only as long as there is remaining cook time.

Question 9

Microwave controller

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

In CTL:
The microwave shall cook only as long as there is remaining cook time.

AG (Cooking -> Timer > 0)

Question 9

Microwave controller

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

In CTL:
If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.

Question 9

Microwave controller

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

In CTL:

If the stop button is pressed when the microwave is not cooking, the remaining cook time shall be cleared.

AG (Button = Stop & !Cooking -> AX (Timer = 0))

Question 9

Microwave controller

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

In **LTL**:
It shall never be the case that the microwave can continue cooking indefinitely.

Question 9

Microwave controller

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

In LTL:
It shall never be the case that the microwave can continue cooking indefinitely.

G (Cooking -> F (!Cooking))

Question 9

Microwave controller

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

In LTL:
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.

Question 9

Microwave controller

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

In LTL:

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.

G (!Cooking U ((Button = Start & Door = Closed) & (Timer > 0)))

Question 9

Microwave controller

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

In LTL:
The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.

Question 9

Microwave controller

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

In LTL:

The microwave shall continue cooking when there is remaining cook time unless the stop button is pressed or the door is opened.

G ((Cooking & Timer > 0) -> X (((Cooking | (!Cooking & Button = Stop)) | (!Cooking & Door = Open)))

Question 10

You are testing the following method:

```
public double max(double a, double b);
```

Devise three executable test cases for this method in the JUnit notation. See the attached handout for a refresher on the notation.

Question 10

```
@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);
  }
```

Any other questions?

Thank you for a great semester!