For all questions that mention FSP, you must write them in the LTSA and your code must be runnable. Except for Q5.

Recommendations for students:

- Students will have a rich format text box for each question,
- This text box allows students to copy and paste solutions directly from Microsoft Word into any file format's text box or attach any file format to that box if needed.
- Please finish the assignment on your local computer, and save each solution as a separate file in an appropriate format. And, in the end, **either copy and paste your solution to each avenue textbox or attach your solution.**
- PLEASE **DO NOT** USE THE **SUBMIT** button every time to **SAVE** your work. Only use the submit button for your final submission.
- Make sure to **save your work** for each question in the avenue **by clicking on the outside of the textbox box** in the avenue, which will allow avenue you to save your response.
- **You must** submit Java files for any Java question.
- You must submit *.txt* or *.LTSA* file(s) for your LTSA questions and paste your code inside the text box. We must be able to copy and paste your code into their respective IDEs and run it without any issues. Please **double-check your code** (i.e. Java, LTSA, etc. before submitting it). The non-working code will be **heavily** penalized or given a zero. The TA's should be able to copy/paste your code and run it without any issues.
- For hand-drawn diagrams, you can attach a photo, we recommend **PNG** over JPG.
- Students can easily create a snapshot of their LTS diagrams by using **the windows** snipping **tool (**Grab for Mac**).**
- Students, who do not have Microsoft Word on their computer, are suggested to use google document editor (Google Docs).
- For questions that do not involve using the LTSA tool or producing FSPs, hand-drawn pictures are allowed, but a solution should be in image format inserted or attached into the textbox (if a student need to scan a handwritten note and insert it into the textbox is encouraged to use a smartphone app called CamScanner).
- There will be a mark deduction for not following the submission instruction. Students must submit their assignments to Avenue. Any problem with Avenue, please discuss with Mahdee Jodayree <mahdijaf@yahoo.com>, the head TA for this course.

For students:

If anything is ambiguous or unclear, please let us know!!!

Question 1.A)

There are a total of six states (PHIL, SERVANT, COOKIES, COLA, SCOOKIES, SCOLA) for this FSP/LTSA code

Instead of creating 3 different philosophers, you must first define a single philosopher. (Note: There is a single state for the philosopher and you must describe its actions similar to the example in the tutorial, however, read the question carefully). Then, use guarded actions (there are 3 guarded actions) to describe the composition of all 3 philosophers by using guarded actions.

The following LTSA example: **chapter3 → Switches**, shows how to create a single state for a **switch**, and then create a composition that shows how 3 switches would run concurrently. This can be applied to the philosophers

LTSA/FSP CODE:

// This is the description for PHIL

```
PHIL = (think -> PHIL | get_cookie -> eat_cookie -> PHIL
                      | get_cola -> drink_cola -> PHIL ).

const K = 3 // Constant K (=3)

range Phil = 1..K // Range from 1 to K (3)

 || PHILS = ( forall[i : Phil] phil[i] : PHIL ).
```

/ This is how you can create 3 compositions of a single state. You can choose any number for philosophers but we recommend that you choose 3. */*

// Now, you must create a state for SERVANT

```
SERVANT = (...)
```

/* Now, create a state for COOKIES; it should have 4 guarded actions (0..3). BUT, there are 2 possibilities; one *if statement*, and the section action for the guarded actions.

Note: There is a single bar/pipe for COOKIES. *See Assignment #1, Question 10 for more information*

AND, don't forget to define the range for COOKIES, similar to how the range for PHIL was defined. Define the range *before* the COOKIES state is defined. */

/* Next, create a state for COLA. It should have a total of 3 guarded actions (0..2). The COLA state is very similar to the COOKIES state. And remember to define the range for COLA, *before* you define the state for COLA. *For reference, please refer to assignment #1, question 10.* */

/* Now, define two properties called SCOLA and SCOOKIE. ***Mahdee, the GOAT, has already defined SCOLA for you***. *Refer to Lecture 11 for more information.* */

property SCOLA = SCOLA[M],

SCOLA[s: Cola] = (when (s==0) fill_cola-> SCOLA[M]).

// Now, define a state/property for SCOOKIE. Note: SCOOKIE is **very** similar to SCOLA

/* Finally, you need to create a composition, which is a composition of all previous states that you have created (i.e. PHILS, COOKIES, COLA, SERVANT, SCOLA, SCOOKIES). *Note: The final composition should not have any mutual exclusion.* */

/* **Note:** SCOLA and SCOOKIE are **property** so they must start with Scola [M] and the conditions end with **M** and **N** instead of **0**. However, COOKIES and COLA are guarded actions so they must start with **COLA[0]**, **COOKIES[0]** */

Question 1.B)

Refer to Lecture Notes 3 on creating an elementary Petri Net

Question 1.C)

Refer to Lecture Notes 9, slide 21 - 22, on creating a Place/Transition Petri Net

Remember to add the transition numbers

Question 1.D)

Refer to Lecture Notes 9, slide 23 - 24, on creating a Colored Petri Net

Remember to indicate what the labels are. For example, at the top, write:

color Philosophers = with ph_1| ph_2 | ... | ph_k

Question 1.E)

Based on your answers from (A), (B), (C), and (D), discuss the differences between the FSP and the Petri Nets, and the differences between the Petri nets themselves.

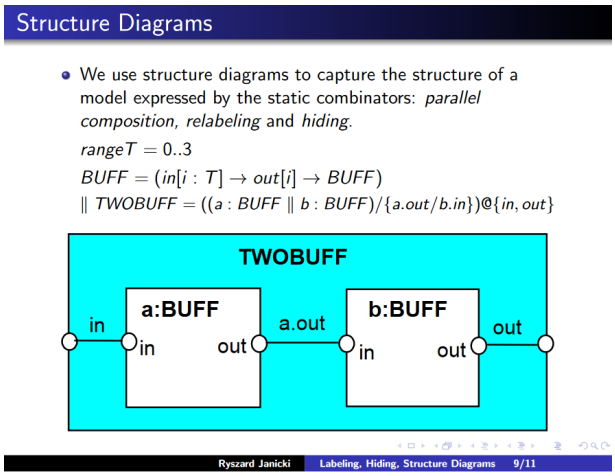
Question 1.F) You can create a java program similar to the one in assignment 1 for the radio question. It will start and prompt the user of the available options, and then it will continue and prompt the new status of the system to the user and prompt new options to the user and ask the user to enter his option.

Question 2.A)

Please see lecture 5 slides 9, for the diagram

You will have your Final composition (label only) on the top, below that you will have the Director state below the final composition and below that, you will have your three other states.

Inside your diagram, there should be a total of 4 white boxes



You must define a state for **DOOR** and define its actions based on the actions described in the question. Door has a single action and it returns to itself.

You must create another state for the director and define their actions. **Director** has several actions and it is the most important state.

The state for the CONTROL contains 5 guarded actions (0 ...4) with different conditions, Very similar to (assignment 1, Q10). Lecture 7 Slide 5 - Lecture 8 - SLide 4

In the Control state, there should be three if statements

If i = 0 then the proper action is is-zero and it will return to control[i]

If i <N then the value of *i* must increment and it must go to Control [i+1]

If i > 0 the value of i decrements and it should go to CONTRO [i-1]

```
CARPARKCONTROL(N=4) = SPACES[N],
SPACES[i:0..N] = (when(i>0) arrive->SPACES[i-1]
                 |when(i<N) depart->SPACES[i+1]
                 ).

ARRIVALS    = (arrive->ARRIVALS).
DEPARTURES = (depart->DEPARTURES).

||CARPARK =
      (ARRIVALS||CARPARKCONTROL(4)||DEPARTURES).
```

SPACES[*i* : 0..*N*] expands to:

SPACES[0] = (*depart* → *SPACES*[1]
SPACES[1] = (*arrive* → *SPACES*[0] | *depart* → *SPACES*[2])
SPACES[2] = (*arrive* → *SPACES*[1] | *depart* → *SPACES*[3])
SPACES[3] = (*arrive* → *SPACES*[2] | *depart* → *SPACES*[4])
SPACES[4] = (*arrive* → *SPACES*[3]

- Blue expansion is used when LTS is produced! This is also and abstract **monitor**
- Guarded actions are used to control *arrive* and *depart*

Ryszard Janicki Monitors, Condition Synchronization & Semaphores 5/14

Nested Monitors and Semaphores

- Suppose that, in place of using the *count* variable and condition synchronization directly, we instead use two semaphores *full* and *empty* to reflect the state of the buffer.

```
const Max = 5
range Int = 0..Max

SEMAPHORE ...as before...

BUFFER =  (put -> empty.down ->full.up ->BUFFER
          |get -> full.down ->empty.up ->BUFFER
          ).

PRODUCER = (put -> PRODUCER).
CONSUMER = (get -> CONSUMER).

||BOUNDEDBUFFER = (PRODUCER|| BUFFER || CONSUMER
                  ||empty:SEMAPHORE(5)
                  ||full:SEMAPHORE(0)
                  )@{put,get}.
```

Does this behave as desired?

- **It deadlocks after the trace *get*!!!**
- Why? *CONSUMER* tries to get a character, but the buffer is empty. It blocks and releases the lock on the semaphore *full*. PRODUCER tries to put a character into the buffer, but also blocks.
- It is called *nested monitor problem*.

Ryszard Janicki Semaphores, Monitors and Buffers & Semaphores 4/12

In the end, there is a final composition which is a composition of all previously defined states (we defined 3 states in this question), there is no mutual exclusion in this composition however you must add labelling for this composition, the actions for the east door and west door must be shown in the state Door). Please watch tutorials to understand how to add labelling to the composition.

Question 2.B)

Model the FSP in 2.A with any Petri Net (your choice)

Question 2. C)

There should be a total of 4 classes as mentioned in the question, MUSEUM, Director, Control Use inheritance as well.

When the program starts it will ask you which door you want to enter from and once the user chooses the door to enter, once the user enters the door, the program must make sure to use atomicity to increment the value of the number of people who are entering the garden.

In one of the tutorials, I explained this lecture slide.

Question 3)

~~You must create a state for car-park control which should have 4 guarded actions (or five guarded actions). With two different conditions (similar to assignment 1 Q10). LEcture note 7, slide 5 but you must write a safety property. Please read the hint for Q1 and understand how to write a safety property.~~

~~You must create a state for arrivals, it has only a single action. And another state for departures has only a single action.~~

~~Then you must create a composition for car-park which would be the composition of all 3 states that you have created so far.~~

To make things easier, start off with the CARPARKCONTROL example in Lecture Notes 7, slide 5. You can also find this in the LTSA tool (i.e. *File* → *Examples* → *chapter5* → *Carpark*).

From here, you need to add the following:

/* Create a property called OVERFLOW with 4 guarded actions.

The OVERFLOW property does not have any *if statements* however it defines what two actions of arriving and departing would do. It is **very** similar to CARPARKCONTROL, minus the *if statements*.

See example below. */

```
//(five guarded actions, 0.. 4,
```

```
property OVERFLOW(N=4) = . . . // very similar to CARPARKCONTROL
```

/* Create a composition called CHECK_CARPARK and this would be the composition of your previously defined property (OVERFLOW) and previously defined composition (CARPARK) */

/* Now, define a progress property for *arrive*. Refer to Lecture Notes 11, slide 21 for more information. But, it looks like this: */

```
progress ENTER = {arrive}
```

/* Finally, define a final composition, which is the composition of CARPARK, while giving depart a lower priority. In the following example, "action" is given a lower priority: */

|| STATE_A = STATE_B >> {action}

/* **Answer this question:** Will starvation occur when car departure has lower priority than car arrival? It would be a good idea to use the LTS diagram and/or animator tool*/

Question 4)

There are a total of 4 different states. They are FORK, PHIL, USE_FORKS, PUT_FORKS. And you must define each state appropriately.

The main point of this question is to avoid deadlock. This is the difference between this question and Slide 19 of Lecture Notes 9. To reiterate, in this question you must avoid deadlock.

FORK

- The FORK state has 2 possibilities of action; very similar to lecture notes, but before you start any action, you must make sure that you acquire that fork

PHIL

- The PHIL state differs from lecture notes. In order to avoid deadlock, it must contain an *acquired* action

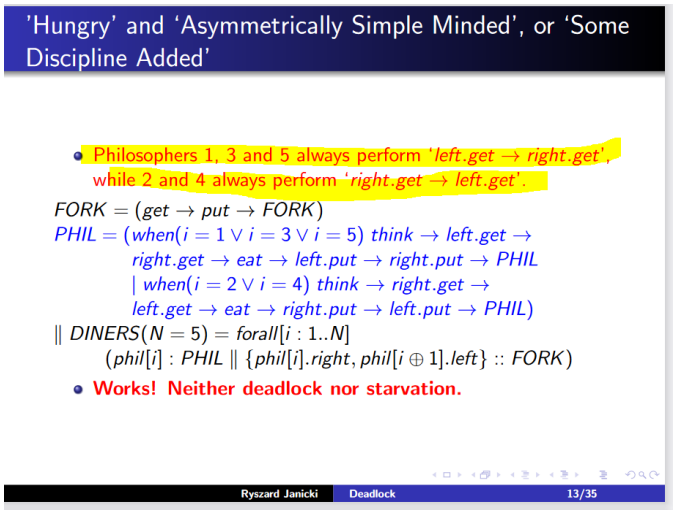
USE_FORKS

- This State will have TWO major sets of actions(you have 1 bar in the middle),
- either philosopher will
    - Take the left fork and then the right fork then eat and then put the fork down
    - or
    - The philosopher will take the right fork and then the left fork and then eat and then put down the forks.
- This state is picking up the forks, eating, and then PUTTING THEM BACK DOWN
- Make sure you avoid deadlock in this state

PUT_FORKS

- This state will also have TWO major sets of actions, in one set,
    - will put down the left fork and then the right fork and return to philosopher state
    - or
    - Will first put down the right fork and then the left fork.
- This state is putting the forks down, one-by-one
- Make sure you avoid deadlock in this state

Lecture 9 Slide 13



In the end, there is a composition which is a composition of 4 philosophers and all philosophers are **mutually excluded** to the appropriate fork. For more information, please refer to: File → Examples → chapter6 → DiningPhilosophers, in the LTSA tool. The composition done in this example is **very** similar to the composition you must do for this question; but it ain't identical!

Refer to Lecture Notes 9, Slide 11 for guidance about the actions for USE_FORKS, PUT_FORKS. *The key takeaway is that you should account for both scenarios; which fork will be picked up?*

Question 5)

There is no LTSA code for this question, however, you must write an FSP (not an FSP code for LTSA). This means your solution does not need to have any commas and dots

Your solution must start, similar to the following line.

- Notation: for get_j^i, put_j^i, i - philosopher number, j - fork number

$$FORK_1 = (get_1^1 \rightarrow put_1^1 \rightarrow FORK_1 \mid get_1^5 \rightarrow put_1^5 \rightarrow FORK_1)$$

I. Define Fork1, Fork2, … all the way to Fork5.

II. Define Phil1, Phil1, …, all the way to Phil5

III. Create a composition state for all the Forks and Phils

Then, draw Petri nets according to your solution. Similar to Lecture 9 - Slide 23

You need to add two more places (circles)

One is called one fork

The second one is called eat. Mμμ mm

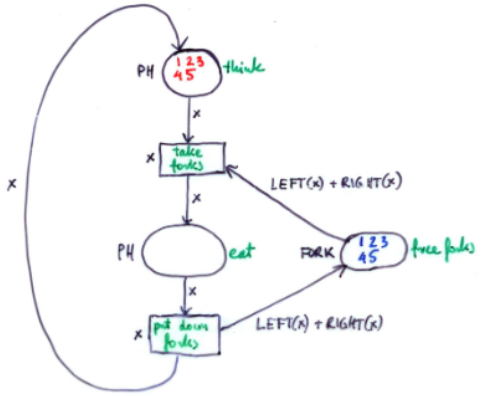
You need to add two more transitions (squares) and rename your previous transitions

The new transitions are **to take the 2nd fork** and **put down the 1st fork**. So think about this.

You need to add two more arcs (arrows) to that petri nets.

In the lecture Slide, in the firing sequence there are **2 *funs***, but in your solution you must have a total of **4 *funs***.

Coloured Petri Nets



colour PH = with ph1 | ph2 | ph3 | ph4 | ph5
colour Fork = with f1 | f2 | f3 | f4 | f5
LEFT : PH → FORK, RIGHT : PH → FORK
var x : PH
fun LEFT x = case of ph1 ⇒ f2 | ph2 ⇒ f3 | ph3 ⇒ f4 |
ph4 ⇒ f5 | ph5 ⇒ f1
fun RIGHT x = case of ph1 ⇒ f1 | ph2 ⇒ f2 | ph3 ⇒ f3 |
ph4 ⇒ f4 | ph5 ⇒ f5

Question 6)

LTSA code for this question is very similar to Assignment 1 question 10, however, there are a total of 11 guarded actions (0 ….10).

*Don't forget to prepend **property** to the state, and remember that a maximum of 10 people are allowed on the elevator at any given time.*

Question 7)

There is no LTSA coding for this question, however, you must first draw LTS by hand then you must use your hand-drawn diagram and write a safety property and this property to find which actions would lead to ERROR and specify that in your new handwritten FSP.

Question 8.A)

// Create 3 different states for each smoker.

SMOKER_T = (get_paper -> get_match->roll_cigarette -> smoke_cigarette ->SMOKER_T)

SMOKER_P = (...)

SMOKER_M = (...)

// Create a state for TOBACCO

TOBACCO = (delivered -> picked -> TOBACCO)

// Create a state for PAPER

PAPER = (...)

// Create a state for MATCH

MATCH = (...)

/* Create 3 different states for each agent that will be delivering each item. The job of the agent is to **deliver** the appropriate materials. */

AGENT_T = (… → … → … → AGENT_T)

AGENT_P = (…)

AGENT_M = (…)

/* Create a state for RULE. The RULE makes sure that the appropriate action of smoking is completed */

RULE = (... → … → RULE)

/* Create 1 composition for SMOKERS; this should be the composition of 3 different smokers. **IMPORTANT:** The name/label you give the smoker here, is the name/label you will use everywhere else. Look at the example below */

SMOKERS = (snoop_D :: SMOKER_T || … || …)

/* Create 1 composition for all RESOURCES; this is the composition of 3 different resources that we previously defined and mutually excluded to 4 different actions. Think about **which smoker** needs what resource. */

RESOURCES = (… || … || …)

/* Create a composition for the AGENTS and RULE; this is the composition of all 3 agents and the RULE state, and mutually excluded to 4 different actions */

AGENT_RULE = (… || … || … || …)

// Finally, compose all 3 composed states in a final composition

CIG_SMOKERS = (SMOKERS || RESOURCES || AGENT_RULE) / {...}

In the end, there must be a final composition that would be composed (composition) of all three previously defined compositions.

Question 8.B)

/* Add safe actions to your FSP code from part (A). You can start this off by defining a property called CORRECT_PICKUP, and the actions are safe for the respective smoker */

property CORRECT_PICKUP = (… → … → CORRECT_PICKUP

| … → … → CORRECT_PICKUP

| … → … → CORRECT_PICKUP

).

/* Finally, compose your property with the other 3 states, **and label it***/

SAFE_CIG_SMOKERS = (SMOKERS || RESOURCES || AGENT_RULE || CORRECT_PICKUP) / { **…** }

Question 8. C)

Take your FSP code from (A) and modify it to fit the question's requirements

'ask first, do later' paradigm (this is similar to the acquire and release concept, we want to avoid deadlock)

For this part, your need to modify the three smokers states.

remember there are **Smoker_T**, **Smoker_P**, **Smoker_M**

and add an action similar to acquire to the **beginning** of these states.

so for example for **Smoker_T**, you can add *no_tobacco* to the beginning of this state.

but remember whatever action you add to the beginning of these states, this action must be a **common action** with some other state.

This means you also need to modify the **AGENT_T, AGENT_P, AGENT_M** and

for example, no_tobacco must be also in the middle of these state

you must insert no-tobacco between can_deliver and deliver the paper

But remember you need to do this for all three states of **Smoker_T**, **Smoker_P**, **Smoker_M accordingly.**

Question 8.D)

Please let us know if you have any questions about this