Homework (Week 5)

Submission: Due on Friday, 15th of July, 11am Sydney Time. Please submit using the CSE Give System either online or via this command on a CSE terminal:

```
give cs3151 hw5 hw5.pdf philosophers.pml merge.pml
```

Late submissions are accepted up to five days after the deadline, but at a penalty: 5% off your total mark per day.

Develop solutions to each of the following problems in Promela, in a separate Promela file.

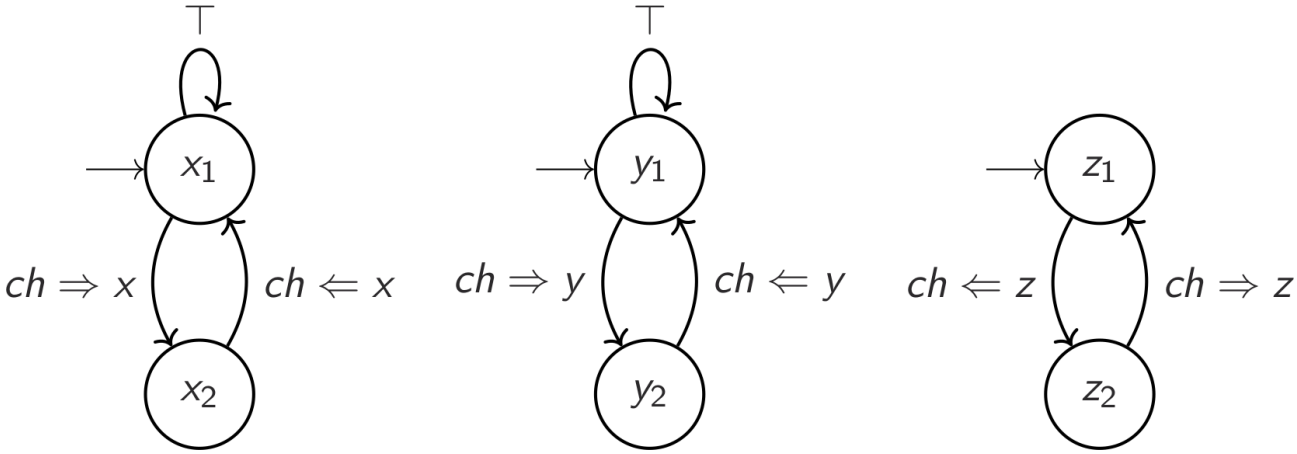
Reasoning about message passing (5 marks)

Here is a critical section algorithm that uses synchronous message passing:

Algorithm 2.1: Mutual exclusion with server		
channel ch of bool		
x	**y**	**z**
forever do x1: ch ⇒ x x2: ch ⇐ x	forever do y1: ch ⇒ y y2: ch ⇐ y	forever do z1: ch ⇐ z z2: ch ⇒ z

Only processes **x** and **y** are competing for the critical section; **z** is an auxiliary process. The critical sections are at lines `x2` and `y2`; `x_1` and `y_1` are the non-critical sections. The program variables x,y,z are just dummies; their values and types are unimportant.

The transition diagrams for these processes are as follows.



(The self-loop is not depicted in the code above; it represents the ability to stay in the non-critical section forever).

1. Construct the closed product of these transition diagrams. The initial state will be $\langle x_1 , y_1 , z_1 \rangle$.
2. It's obvious from inspection of the closed product that this algorithm satisfies mutual exclusion. Why?
3. Does this algorithm satisfy eventual entry? Briefly motivate.
4. Does this algorithm still work if we flip all inputs to outputs, and vice versa? Brifely motivate.
5. The algorithm behaves oddly if we make **ch** asynchronous. Describe a scenario that (a) assumes an asynchronous, reliable channel; (b) goes on forever in a cycle; and (c) takes transitions other than the self-loops at `x1` and `y1` infinitely often; and (d) never visits locations `x2` and `y2`.

Submit your answers in a pdf file called `hw5.pdf`.

Philosophers (4 marks)

Develop a solution for the dining philosophers problem using only message passing, under the additional restriction that each channel must be connected to exactly one sender and exactly one receiver.

By way of a hint, the following things do not work:

- Having only 5 processes, one for each philosopher.
- Having only 5 channels, one for each fork.

Configure the solution to run forever, in a 5 philosopher scenario.

Put all your work in a file called `philosophers.pml`. Do not include any other files.

Fair Merge (4 marks)

Develop an algorithm to merge two sequences of data. A process called `merge` is given three channel parameters of type `byte`, receives data on two input channels and interleaves the data on the output channel, such that if the two inputs are sorted (in ascending order), then the output is sorted.

Try to implement a fair merge that is free from starvation of both input channels when possible. This means that you should try to make sure every input stream is always eventually read from. This requirement will sometimes be impossible to reconcile with the sortedness requirement. If so, keeping the outputs sorted takes priority. For example, if the two input channels transmit infinite streams of 1:s and 2:s, respectively, no 2:s should be sent on the output channel.

Assume that the value 255 is a special `EOF` signal, and no further data will be sent on a channel after `EOF` is sent. Your merge process should terminate if all data has been transmitted. Assume that an `EOF` will be sent at the end of the data stream (if it ends).

Put all your work in a file called `merge.pml`. Do not include any other files.