Concurrency Exercises 5: Properties of concurrent systems

Topic 5: Properties


\[ \text{FORK} = (\text{get} \rightarrow \text{put} \rightarrow \text{FORK}). \]

\[ \text{PHIL} = (\text{sitdown} \rightarrow \text{right.get} \rightarrow \text{left.get} \rightarrow \text{eat} \rightarrow \text{left.put} \rightarrow \text{right.put} \rightarrow \text{arise} \rightarrow \text{PHIL}). \]

\[ \Box \text{DINERS(N=5)} = \forall [i:0..N-1] (\phi[i]:\text{PHIL} || \{\phi[i].\text{left}, \phi[((i-1)+N)\%N].\text{right}\}::\text{FORK}). \]

Modify this model to include a ‘butler’ so that only N-1 philosophers are hungry or eating at any time, i.e. the butler process only permits up to N-1 philosophers to sit down at the table to attempt to eat. Show that the solution is deadlock-free and check using the LTSA tool.

Answer:

\[ \Box \text{SITTING(N=5)} = \{\phi[i:0..N-1]::\text{BUTLER}(N-1)\}::\text{FORK}. \]

\[ \text{FORK} = (\text{get} \rightarrow \text{put} \rightarrow \text{FORK}). \]

\[ \text{PHIL} = (\text{sitdown} \rightarrow \text{right.get} \rightarrow \text{left.get} \rightarrow \text{eat} \rightarrow \text{left.put} \rightarrow \text{right.put} \rightarrow \text{arise} \rightarrow \text{PHIL}). \]
2. Now modify your FSP description so that each philosopher must acquire the two forks together as an atomic step. Show that the solution is deadlock-free.

**Answer:**

FORK = (get -> put -> FORK).


||DINERS2(N=5) =
forall [i:0..N-1] (phil[i]:PHIL2 || {phil[i], phil[((i-1)+N)%N]}.::FORK ).

3. Think about other deadlock-free solutions to the Dining Philosophers Problem, constructing FSP models and Java implementations. Hint: Consider the properties required of a system to ensure the possibility of deadlock. How can these be avoided? See the course textbook for a discussion introducing a variety of solutions.

4. Can you make the above solutions fair, so that a hungry philosopher will eventually get to eat?

**Answer:**

**One solution:** We need to ensure that the butler chooses fairly between hungry philosophers so that everyone waiting will eventually be served. Having a blocked-queue semaphore for the Butler is one way to achieve this fairness.

5. From Magee and Kramer: The figure below depicts a maze. Write a description of the maze in FSP, which, using deadlock analysis, finds the shortest path out of the maze starting at any square.

Hint: at each numbered square in the maze, a directional arrow (N,S,E,W) can be used to indicate an allowed path to another square.

**Answer:**
MAZE(Start=8) = P[Start],
P[0] = (north->STOP|east->P[1]),
P[6] = (north->P[3]),
P[7] = (east ->P[8]),

||GETOUT = MAZE(7).

6. Write an algorithm for deadlock detection in FSP processes or LTS graphs.

**Answer:**

**One solution:** Assume LTS has been constructed and use a standard Breadth-First Search. Maintain a FIFO state queue $Q$ and a visited stateset $S$. If $Q = \{\}$ then no deadlock. Otherwise, for each $q \in Q$: if $q \in S$ skip else if $\text{succ}(q) = \{\}$ then Deadlock else add $\text{succ}(q) - S$ to (tail of) $Q$ (i.e. add list of all successor states not in $S$).

**Another Solution:** Perform the above but without constructing the LTS, i.e. on-the-fly directly from the FSP process term. What is a ‘state’? How to do ‘state-equivalence’?

7. A lift has a maximum capacity of $N$ people. In a model of the lift control system, passengers entering the lift are signalled by an enter action and passengers leaving the lift are signalled by an exit action. Specify a safety property in FSP, which when composed with the lift model, will check that the system never allows the lift that it controls to have more than $N$ occupants.

**Answer:**

```
const N = 10

property LIFTCAPACITY = LIFT[0],
LIFT[i:0..10] = (enter -> LIFT[i+1]
  |when(i>0) exit  -> LIFT[i-1]
  |when(i==0)exit -> LIFT[0]
).
```

8. Specify an FSP safety property for the Car Park problem (5.1.1 in Magee and Kramer’s Concurrency book), which asserts that the carpark does not overflow.

**Answer:**

From Magee & Kramer: The following property only allows arrival for $N$ cars:

```
property OVERFLOW(N=4) = OVERFLOW[0],
OVERFLOW[i:0..N] = (arrive -> OVERFLOW[i+1]
```

**Answer:**

const False = 0
const True = 1
range Bool = False..True
set BoolActions = {setTrue, setFalse, [False], [True]}

BOOLVAR = VAL[False],
VAL[v:Bool] = (setTrue -> VAL[True]
  | setFalse -> VAL[False]
  | [v] -> VAL[v]
).

||FLAGS = (flag1:BOOLVAR || flag2:BOOLVAR).

NEIGHBOUR1 = (flag1.setTrue -> TEST),
TEST = (flag2[b:Bool] ->
  if(b) then
    (flag1.setFalse -> NEIGHBOUR1)
  else
    (enter -> exit -> flag1.setFalse -> NEIGHBOUR1)
)+{{flag1,flag2}.BoolActions}.

NEIGHBOUR2 = (flag2.setTrue -> TEST),
TEST = (flag1[b:Bool] ->
  if(b) then
    (flag2.setFalse -> NEIGHBOUR2)
  else
    (enter -> exit-> flag2.setFalse -> NEIGHBOUR2)
)+{{flag1,flag2}.BoolActions}.

property SAFETY = (n1.enter -> n1.exit -> SAFETY | n2.enter -> n2.exit -> SAFETY).

||FIELD = (n1:NEIGHBOUR1 || n2:NEIGHBOUR2 || {n1,n2}::FLAGS || SAFETY).

progress ENTER1 = {n1.enter} //NEIGHBOUR 1 always gets to enter
progress ENTER2 = {n2.enter} //NEIGHBOUR 2 always gets to enter

/* greedy neighbours - make setting the flags high priority - eagerness to enter*/

||GREEDY = FIELD << {{n1,n2}.flag1,flag2}.setTrue}. 
/* progress violations show situation where neither neighbour enters
* each continually retests the lock
*/