Swansea University
Department of Computer Science

U Berger
October 18, 2022

CSC385/CSCM85 Modelling and Verification Techniques

Lab 2

The Dining Philosophers' Problem (Dijkstra 1971)

The tasks below may be done in groups of up to four students. To obtain full marks, all tasks must be completed. For fewer or incompletely accomplished tasks, partial marks will be awarded. Marks will be awarded only for work that is demonstrated during the lab session. Write your answers in the file `verification-lab2-Oct22-template.csp`.

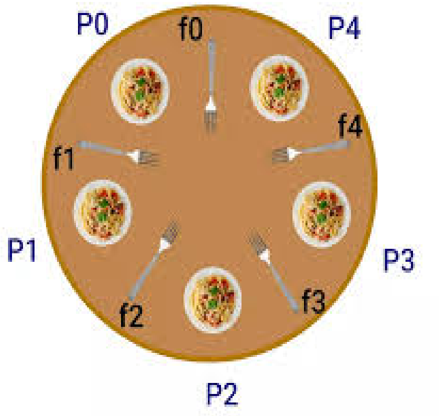
This lab assignment is worth 30 marks out of 100 lab marks. It needs to be completed and assessed in one of the lab sessions on the 18th, 21st, 25th, or 28th of October.

Introduction A number of philosophers (say 5) do what philosophers do: They think. After a while, they get hungry. So, they sit down at the dining table (taking their allocated seats), pick up the fork to their left, then the fork to their right and start eating. After having eaten enough, they put down the forks, get up and go back to their business of thinking.

The problem is that a philosopher's right fork is the same as the right neighbour's left fork. Therefore, if two neighbouring philosophers want to eat at they same time they are competing for a fork.

Can the philosophers get stuck?

We assume that the fork to the left of a philosopher has the same index as the philosopher, and indices increase counterclockwise (modulo 5, of course), as shown in the image below.[1]



[1]https://en.wikipedia.org/wiki/Dining_philosophers_problem

We model the dining philosophers as a CSP process and investigate its properties.

First, we fix the number M of philosophers and forks, and name the philosophers and forks by the numbers $0, \dots, M-1$.

```
M = 5
I = {0..M-1}
```

The name of the right neighbour of philosopher n is $(n+1)\%M$ where $\%$ is the modulus function ($k\%M = k$ if $k \in I$ and $M\%M = 0$).

```
right(n) = (n+1)%M
```

The fork with number n is the fork to the left of philosopher n. The fork with number right(n) is the fork to the right of philosopher n. We assume that the philosophers always pick up and put down the fork to their left first:

```
first_up(n) = n
second_up(n) = right(n)
first_down(n) = n
second_down(n) = right(n)
```

The reason for introducing these functions is that later we want to change the habits of the philosophers, and these functions allow us to do this easily, without having to edit CSP processes.

We introduce channels for thinking, eating, putting down and picking up forks:

```
channel think, eat : I
channel up, down: I.I
```

up.n.m means that philosopher n picks up fork m, down.n.m means that philosopher n puts down fork m. Hence, for example, `up.n.second_up(n)` is the event that philosopher n picks up her second fork. Assuming $M = 5$, this means that, for example, `up.4.second_up(4) = up.4.0`, since `second_up(4) = 0`.

Task 1. (Modelling the philosophers)

Define a parametrised process $P(n)$ that models the behaviour of Philosopher n.

More precisely, use the functions and channels defined above to express that Philosopher n:

thinks,

then picks up her first fork,

then picks up her second fork,

then eats,

then puts down her first fork,

then puts down her second fork,

then repeats.

```
P(n) = think.n -> ...
```

Use the replicated form of interleaving to define a process Phils that models all philosophers acting in parallel without communication between each other.

```
Phils = ...
```

5 marks

See the course notes, or the process Forks below, for examples of how to use the replicated form of interleaving.

Remark: You may want to draw the graphs of some of the processes you are constructing (for example Phils). In that case, change the value of M to 3. Otherwise, the processes may be too complex to draw their graphs.

Modelling forks A fork can be picked up by anybody and then then put down by anybody. But if a fork has been picked up, the same fork can't be picked up again before it hasn't been been put down.

```
F(n) = [] m : I @ up.m.n -> [] k : I @ down.k.n -> F(n)
```

Forks run in parallel and don't communicate with each other.

```
Forks = ||| n : I @ F(n)
```

Task 2. (Dining Philosophers and their analysis)

The dining Philosophers can decide to eat at any time but they have to respect the restrictions imposed by the process Forks. This means that the philosophers and the forks must synchronise on picking-up and putting-down forks

Use the generalised parallel to model this behaviours.

```
DinPhils = Phils [| ... |] Forks
```

Use a suitable assertion to check whether it is possible that the dining philosophers get stuck, that is, whether there is a deadlock.

```
assert ...
```

Use FDR's debugging facility to understand the result.

5 marks

Task 3. (Avoiding deadlock by changing a philosopher's habits) Try to avoid deadlock by changing one philosophers habits, say, that of philosopher number 0, so that she picks up the right fork first.

```
first_up(n) = if n == 0 then right(n) else n
second_up(n) = ...
```

Check that now the Dining Philosophers can't get stuck. **5 marks**

Avoiding deadlock with the help of a butler Since it is not realistic to assume that philosopher will change their habits, we instead employ a butler who makes sure that not all philosophers are seated at the same time.

We define functions inc and dec that increase respectively decrease a number between 0 and M by one but capping the result if it would be negative or greater than M.

```
inc(n) = if n < M then n+1 else n
dec(n) = if n > 0 then n-1 else n
```

The butler restricts the number of seated philosophers. A philosopher sits down when they pick their first fork, and they get up when they put down their second fork.

Hence, we define the set of sitting-down events and the set of getting-up events as

```
Sit_Down = { up.n.first_up(n) | n <- I }
Get_Up = { down.n.second_down(n) | n <- I }
```

The butler records how many philosophers are currently seated and makes sure that this number is always less than M.

```
Butler(k) = (if k < M-1
             then [] sit : Sit_Down @ sit -> Butler(inc(k))
             else STOP)
          [] ([] getup : Get_Up @ getup -> Butler(dec(k)))
```

Task 4. (Controlling the dining philosophers by the butler)

Define a process DinPhilsB that models the dining philosophers controlled by the Butler. Then check whether this process is deadlock free.

5 marks

Controlling the number of eating philosophers. Due to rising energy prices and staff shortage, catering can only serve food for half of the philosophers, and therefore need to make sure that at most half of the philosophers are eating at any time.

Note that the event eat.n signals that Philosopher n has *started* eating. To detect when she has *finished* eating, we use the event that she puts down her first fork. We define the set of these finish-eating events as

```
FinishEating = { down.n.first_down(n) | n <- I }
```

We define a process Catering(k) that keeps track of the number k of philosophers who are currently eating. It uses a channel, eating, reporting that number.

```
channel eating : Int

Catering(k) =
    eating.k ->  ( eat?n -> Catering(inc(k))
                    []
                  ([] f : FinishEating @ f -> Catering(dec(k))) )
```

The philosophers monitored by catering are modelled by running them in parallel with Catering(0) (at the beginning, no one is eating) but synchronising on the events that signal beginning and finishing of eating:

```
DinPhilsC = DinPhils [| union({|eat|}, Finish_Eating) |] Catering(0)
DinPhilsBC = DinPhilsB [| union({|eat|}, FinishEating) |] Catering(0)
```

Task 5. (Checking catering's workload)

We define a parameterised process CateringSpec(m) that serves as the specification that at most m philosophers are currently eating:

```
CateringSpec(m) = eating ? k : {0..m} -> CateringSpec(m)
```

Use trace refinement to check that at any time at most $M/2$ philosophers are eating.

In addition, verify that this number cannot be lowered.

Don't forget to hide the events that are irrelevant!

5 marks

Task 6. (Verifying that controlled philosophers are not impeded)

The philosophers that are controlled by the butler, DinPhilsB, are subject to extra synchronisation requirements which might restrict their ability to think and eat.

Show that this is not the case by checking that these DinPhilsB can perform the same sequences of think and eat actions as the original process DinPhils.

Again, don't forget to hide the irrelevant events.

5 marks