

Probabilistic Model Checking Practicals

Practical 4

This practical concerns using the PRISM tool for the specification and analysis of Markov decision process models.

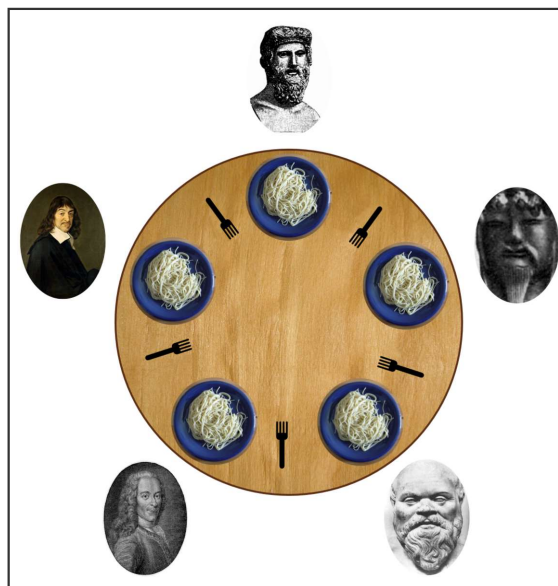
Items that should be included in your practical report are marked with a **[*]** below (and also summarised at the end of this page).

1. The Dining philosophers problem

The dining philosophers problem is an example of a large class of concurrency problems that attempt to deal with allocating a set number of resources among several processes. The problem originates with Edsger Dijkstra, who in 1971 set an examination question where five computers competed for access to five shared tape drives. The problem was then retold by Tony Hoare as the dining philosophers problem.

The problem supposes a certain number of philosophers, seated around a circular table, who spend their lives alternating between thinking and eating. There is a fork placed between each pair of neighbouring philosophers which both can access and a philosopher must be in possession of both forks in order to eat. A philosopher may only pick up one fork at a time and when a philosopher finishes eating, both forks are placed back on the table and he/she returns to thinking. Notice that, since the philosophers share forks, it is not possible for all to eat at the same time.

Below we represent graphically the case when there are five philosophers seated around the table.



A solution to this problem is an algorithm for each of the philosophers to follow that will ensure that:

- there will never be a situation of *deadlock* (i.e. where no philosopher gets to eat)
- there will never be a situation of *resource starvation* (i.e. where a philosopher wants to eat but never does so).

For an example of a situation that can lead to deadlock, suppose that all philosophers become hungry at the same time and reach out and take their left fork, then if all reach for the right fork no one can move.

2. A Randomised Solution

We now consider Lehmann and Rabin's randomised solution to this problem [LR81]. It ensures that the system is deadlock-free, i.e. that, if a philosopher is hungry, then eventually a philosopher eats.

The PRISM model in the case of three philosophers is given below.

```

// randomized dining philosophers [LR81]
// three philosophers

mdp

// formulae
// left fork free (left neighbour is philosopher 2)
formula lfree = (p2>=0 & p2<=4) | p2=6 | p2=10;
// right fork free (left neighbour is philosopher 3)
formula rfree = (p3>=0 & p3<=3) | p3=5 | p3=7 | p3=11;

module phil1

    p1: [0..11];

    [] p1=0 -> (p1'=1); // trying
    [] p1=1 -> 0.5 : (p1'=2) + 0.5 : (p1'=3); // draw randomly
    [] p1=2 & lfree -> (p1'=4); // pick up left
    [] p1=3 & rfree -> (p1'=5); // pick up right
    [] p1=4 & rfree -> (p1'=8); // pick up right (got left)
    [] p1=4 & !rfree -> (p1'=6); // right not free (got left)
    [] p1=5 & lfree -> (p1'=8); // pick up left (got right)
    [] p1=5 & !lfree -> (p1'=7); // left not free (got right)
    [] p1=6 -> (p1'=1); // put down left
    [] p1=7 -> (p1'=1); // put down right
    [] p1=8 -> (p1'=9); // move to eating (got forks)
    [] p1=9 -> (p1'=10); // finished eating and put down left
    [] p1=9 -> (p1'=11); // finished eating and put down right
    [] p1=10 -> (p1'=0); // put down right and return to think
    [] p1=11 -> (p1'=0); // put down left and return to think

endmodule

// construct further modules through renaming
module phil2 = phil1 [ p1=p2, p2=p3, p3=p1 ] endmodule
module phil3 = phil1 [ p1=p3, p2=p1, p3=p2 ] endmodule

// labels

// a philosopher is hungry
label "hungry" = ((p1>0)&(p1<8))|((p2>0)&(p2<8))|((p3>0)&(p3<8));
// a philosopher is eating
label "eat" = ((p1>=8)&(p1<=9))|((p2>=8)&(p2<=9))|((p3>=8)&(p3<=9));
// philosopher 1 is hungry
label "hungry1" = (p1>0)&(p1<8);
// philosopher 2 is hungry
label "hungry2" = (p2>0)&(p2<8);
// philosopher 3 is hungry
label "hungry3" = (p3>0)&(p3<8);
// philosopher 1 is eating
label "eat1" = (p1>=8)&(p1<=9);
// philosopher 2 is eating
label "eat2" = (p2>=8)&(p2<=9);
// philosopher 3 is eating
label "eat3" = (p3>=8)&(p3<=9);

```

View: [printable version](#)Download: [lr3.nm](#)

This model is a Markov decision process (MDP), as denoted by the keyword at the start of the file. We have three modules, representing the three philosophers.


Notice that, after a philosopher eats (e.g. when $p1=9$ for philosopher 1), the nondeterministic choice as to which fork he/she puts down first is represented by two commands being enabled (i.e. there are multiple commands with overlapping guard $p1=9$).


Notice also the use of the PRISM formulae (`lfree` and `rfree`) to represent expressions that are re-used several times in the file. In this model, these formulae are used to identify states of the left (or right) neighbour of a philosopher in which he/she is not using his/her fork. As a consequence, this requires the renaming of each of the variables $p1$, $p2$ and $p3$ when constructing the modules for philosophers 2 and 3. This renaming corresponds to the fact that the left and right neighbours of philosopher 2 and 3 differ from that of philosopher 1.


The basics steps of the algorithm for each philosopher are:


1. move from thinking to hungry (to simplify the analysis we assume that philosophers always become hungry);
2. when hungry, randomly choose to try and pick up the left or right fork;
3. wait until the fork is down and then pick it up;
4. if the other fork is free, pick it up; otherwise, put the original fork down (and return to step 1);
5. eat (since in possession of both forks);
6. when finished eating, put both forks down in any order and return to thinking.

 Download the model file **lr3.nm** from above and load it into PRISM.

 Before starting you should modify one of the settings in PRISM: launch the "Options" dialog from the main menu and set the "engine" to be "MTBDD" (this might already be the case if you saved your setting in a previous practical). This is because not all the model checking algorithms used in the practical are currently supported by the other engines.


 Use the PRISM simulator to generate some random paths through the model. Notice how, for an MDP model like this, multiple probability distributions may be available in a state. Generate a path by hand where one of the philosophers eats.

 By using module renaming, write versions of the algorithm when there are 4 and 5 philosophers. One thing to take into account is whether the neighbours of the philosophers change. [*]

 For each of the models, add labels corresponding to whether each individual philosopher is hungry/eating.

3. Analysing Lehmann and Rabin's Solution

We will now use PRISM to analyse the above model.


 Load the following properties file into PRISM:

```
// deadlock freeness
filter(forall, "hungry" => P>=1 [ F "eat" ])


// no resource starvation
filter(forall, "hungry1" => P>=1 [ F "eat1" ])
filter(forall, "hungry2" => P>=1 [ F "eat2" ])
filter(forall, "hungry3" => P>=1 [ F "eat3" ])
```


View: [printable version](#)

Download: [lr3.pctl](#)

 Verify the loaded properties to check that the model is deadlock-free, but is *not* resource starvation-free. Also verify your models for four and five philosophers against these specifications.

 Read the section of the manual on [filters](#) as we will use these in our specifications.

 Construct properties to compute, from the (single) state where all philosophers are thinking, both the minimum and maximum probability that a philosopher eats within k steps. (Hint: look up the **state** filter in the manual page above.) Plot results for the case of three, four and five philosophers.

 Construct reward structures and properties to find both the minimum and maximum expected time for a philosopher to eat from the state where *all* philosophers are thinking. Verify these properties on the models of three and four philosophers. Describe a strategy that achieves the minimum expected time. [*]

4. A Resource Starvation-Free Solution

Lehmann and Rabin solved the resource-starvation problem by extending the above approach so that the philosophers are courteous. More precisely: a philosopher will not pick up his/her neighbour's fork (when it has no forks) if that neighbour is trying to eat and has not eaten since the philosopher's most recent meal.

This is achieved by using shared variables between neighbouring philosophers and shown by the following pseudo-code.

```

var left-signal, right-signal : {On,Off};
*** left-signal is shared with left neighbour initially set to Off ***
*** It is set to On when one becomes hungry and restored to Off only after eating. ***
*** The left neighbour may read it but not change it and refers to it as right-neighbour-signal . ***
*** The case is symmetric for the right-signal. ***


read only var left-neighbour-signal ,right-neighbour-signal : {On,Off};
*** left-neighbour-signal is left neighbour's right signal. ***

var left-last,right-last : {Left,Neutral,Right};
*** left-last is shared with left neighbour and both may change it. ***
*** It indicates who ate last : left from fork or right from fork ***
*** Initially the value is Neutral (as no one has eaten) ***
*** left-last is the same as left neighbour's right-last. ***

1. trying:=true *** move from thinking to hungry ***
2. left-signal:=On; right-signal:=On
3. while trying do
4.   draw a random element s of {Right,Left}; *** with equal probabilities ***
5.   wait until s fork is down and (s-neighbour-signal=Off or s-last=Neutral or s-last=s)
6.   then lift fork s
7.   if R(s) fork is down *** R(Right)= Left and R(left)=Right ***
8.   then lift it and trying:=false
9.   else put down fork s
10. od
11. eat;
12. left-signal:=Off; right-signal:=Off;
13. left-last:=Right; right-last:=Left;
14. put down both forks *** one at a time in any order ***


```


We will now extend our PRISM code to specify Lehmann and Rabin's courteous dining philosophers.


 First, save a new copy of the model `lr3.nm` as `clr3.nm`.


Next, we will add a number of constants, variables and formulae that will be needed in the specification.


 Add constants for **NEUTRAL**, **LEFT** and **RIGHT** which take the values 0, 1 and 2 respectively.

 To represent the variables `left-signal` and `right-signal`, add the variables `left_signal1` and `right_signal1` to philosopher 1 and add similar variables for philosophers 2 and 3 through renaming. Define suitable ranges and initial values for these variables given that `left-signal` and `right-signal` take two values (Off and On).

 In the pseudo code there are two shared variables, `left-last` and `right-last`, that we will represent in PRISM using global variables. Read the sections on [global variables](#) in the manual. Now add global variables `fork1_last`, `fork2_last` and `fork3_last` which will be used to represent the variables `left-last` and `right-last` in the pseudo code above. Define suitable ranges and initial values for these variables given that these variables take the values **NEUTRAL**, **LEFT** and **RIGHT** (and the constants we introduced above).

 Draw a diagram of the three philosophers (`phil1`, `phil2` and `phil3`) and three forks (`fork1`, `fork2` and `fork3`) as this will help understand the relationships between the philosophers and the forks. For example, this should make it easier to work out which fork/philosopher is to the right/left of philosopher 1.

 Write formulae for philosopher 1 being courteous when waiting to pick up his/her left fork and when waiting to pick up his/her right fork. Now rename the appropriate variables in the modules of philosopher 2 and 3 such that after renaming these formulae they correctly model these philosophers being courteous.

 Using the above constants, variables and formulae modify the commands of `phil1` to match the pseudo code for the courteous dining philosophers defined above. Note that you will probably have to extend the range of `p1` and you will also have to change the labels at the end of the file to match your modifications. Notice since you have already changed the renaming to match the new variables and formulae the modules `phil2` and `phil3` should also now correctly model the courteous dining philosophers. [*]

5. Analysing the Courteous Philosophers



Check the protocol is resource starvation-free and deadlock-free.



From any state where *all* philosophers are hungry, find the minimum and maximum expected time for any philosopher to eat and, for each philosopher, from any state where he/she is hungry, the minimum and maximum expected time for that philosopher to eat. [*]

Practical Report

To get this practical signed off, you should provide the demonstrator with a brief practical report comprising the starred items from the text above, i.e.:

1. from part 2: the PRISM code for the model for five philosophers;
 2. from part 3: the minimum and maximum expected time for a philosopher to eat and a description of an adversary which achieves the minimum;
 3. from part 4: the PRISM code for the courteous model for three philosophers;
 4. from part 5: the minimum and maximum expected time for any philosopher to eat and each philosopher to eat.
-

[[Back to index](#)]



Site hosted at the Department of Computer Science, University of Oxford