Probabilistic Model Checking Practicals

Practical 3

This practical concerns using the PRISM tool for the specification and analysis of continuous-time Markov chain (CTMC) 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. Dynamic power management (DPM) schemes

In this third practical we are going to model and analyse dynamic power management (DPM) schemes. A DPM scheme is a set of instructions used to switch between different power states of a computing device. Our model of a DPM system comprises three components:

- the *Service Queue (SQ)* into which requests for a particular service are placed;
- the *Service Provider (SP)* which responds to these requests;
- the *Power Manager (PM)* which controls the Service Provider.

We are going to consider a specific DPM system in which the service provider is a Fujitsu disk drive with 3 power states. In this case the SQ corresponds to read/write requests to the disk and, as stated, the SP is the actual disk drive with the power states: *sleep*, *idle* and *busy*. The power states of the drive have different power consumptions and also different response times to service requests.

Switches between power states are either performed by the Service Provider itself or initiated by the Power Manager. The SP controls switches between the *idle* and *busy* states. A switch from *idle* to *busy* occurs when a request arrives and a switch from *busy* to *idle* occurs when the final request in the queue is served. Switches between the *sleep* power state and the *idle* and *busy* states are under the control of the PM. The aim of the PM is to switch between these states such that power consumption is kept as low as possible whilst maintaining an acceptable level of service.

Our initial PRISM model is the following:

```
// Simple dynamic power management (DPM) model
// Based on:
// Qinru Qiu, Qing Wu and Massoud Pedram
// Stochastic modeling of a power-managed system: Construction and optimization
// Proc. International Symposium on Low Power Electronics and Design, pages 194--199, ACM Pres

ctmc

//-------------------------------------------------

// Service Queue (SQ)
// Stores requests which arrive into the system to be processed.

// Maximum queue size
const int q_max = 20;

// Request arrival rate
const double rate_arrive = 1/0.72; // (mean inter-arrival time is 0.72 seconds)

module SQ

        // q = number of requests currently in queue
        q : [0..q_max] init 0;

        // A request arrives
        [request] true -> rate_arrive : (q'=min(q+1,q_max));
        // A request is served
        [serve] q>1 -> (q'=q-1);
        // Last request is served
        [serve_last] q=1 -> (q'=q-1);

endmodule
```

```
//-------------------------------------------------

// Service Provider (SP)
// Processes requests from service queue.
// The SP has 3 power states: sleep, idle and busy

// Rate of service (average service time = 0.008s)
const double rate_serve = 1/0.008;

module SP

	// Power state of SP: 0=sleep, 1=idle, 2=busy
	sp : [0..2] init 1;

	// Synchronise with service queue (SQ):

	// If in the idle state, switch to busy when a request arrives in the queue
	[request] sp=1  -> (sp'=2);
	// If in other power states when a request arrives, do nothing
	// (need to add this explicitly because otherwise SP blocks SQ from moving)
	[request] sp!=1 -> (sp'=sp);

	// Serve a request from the queue
	[serve]      sp=2 -> rate_serve : (sp'=2);
	[serve_last] sp=2 -> rate_serve : (sp'=1);

endmodule

//-------------------------------------------------

// Reward structures

rewards "queue_size"
	true : q;
endrewards
```

View: printable version **Download:** power.sm

This model is a continuous-time Markov chain (CTMC), as denoted by the keyword at the start of the file. Presently, we just have two modules, representing the Service Queue (SQ) and Service Provider (SP). There are a couple of important things to note.

Firstly, since this is a CTMC, updates in guarded commands are labelled with rates, not probabilities. The first command in the module SQ, for example, which describes the arrival of a request into the queue, has the rate rate_arrive, which is a constant defined just above in the file.

Secondly, notice that all the commands are labelled with actions (in square brackets at the start of the line).

Read the section on synchronisation in the manual. Then, have a look at the definition of the SQ and SP modules, and try to understand what they describe.

Download the model file **power.sm** from above and load it into PRISM.

Use the PRISM simulator to generate some random paths through the model. Notice how, for a CTMC model like this, the elapsed time as the path progresses is displayed in the table. You will probably find that the size of the queue (q) never gets above 1. Why is this? Generate a path by hand where the queue reaches its maximum size (currently q_max=20). What happens when more requests arrive while the queue is full?

What is the size of the state space of this model? (i.e from the initial state, how many possible different states can be reached?) Go back to the "Model" tab of the GUI, select menu option "Model | Build model" and then look at the statistics displayed in the bottom left corner to check your answer.

2. Adding the power management control

We now extend our PRISM model by adding a Power Manager (PM) component. The PM functions by observing the state of the system (i.e. the SQ and the SP) and, based on this, instructs the SP when to change between the *sleep*

power state and the *idle* and *busy* states. In PRISM, we will model this via synchronisation between the existing SP module and a new module called PM. We will use the actions idle2sleep and sleep2idle which tell the SP to move between its *sleep* and *idle* power states. Here is the PRISM code for the model extended with the PM module.

```
// Simple dynamic power management (DPM) model
// Based on:
// Qinru Qiu, Qing Wu and Massoud Pedram
// Stochastic modeling of a power-managed system: Construction and optimization
// Proc. International Symposium on Low Power Electronics and Design, pages 194--199, ACM Pres

ctmc

//-------------------------------------------------

// Service Queue (SQ)
// Stores requests which arrive into the system to be processed.

// Maximum queue size
const int q_max = 20;

// Request arrival rate
const double rate_arrive = 1/0.72; // (mean inter-arrival time is 0.72 seconds)

module SQ

	// q = number of requests currently in queue
	q : [0..q_max] init 0;

	// A request arrives
	[request] true -> rate_arrive : (q'=min(q+1,q_max));
	// A request is served
	[serve] q>1 -> (q'=q-1);
	// Last request is served
	[serve_last] q=1 -> (q'=q-1);

endmodule

//-------------------------------------------------

// Service Provider (SP)
// Processes requests from service queue.
// The SP has 3 power states: sleep, idle and busy

// Rate of service (average service time = 0.008s)
const double rate_serve = 1/0.008;
// Rate of switching from sleep to idle (average transition time = 1.6s)
const double rate_s2i = 1/1.6;
// Rate of switching from idle to sleep (average transition time = 0.67s)
const double rate_i2s = 1/0.67;

module SP

	// Power state of SP: 0=sleep, 1=idle, 2=busy
	sp : [0..2] init 0;

	// Respond to controls from power manager (PM):

	// Switch from sleep state to idle state
	// (in fact, if the queue is non-empty, go straight to "busy" , rather than "idle")
	[sleep2idle] sp=0 & q=0 -> rate_s2i : (sp'=1);
	[sleep2idle] sp=0 & q>0 -> rate_s2i : (sp'=2);
	// Switch from idle state to sleep state
	[idle2sleep] sp=1 -> rate_i2s : (sp'=0);

	// Synchronise with service queue (SQ):

	// If in the idle state, switch to busy when a request arrives in the queue
	[request] sp=1  -> (sp'=2);
	// If in other power states when a request arrives, do nothing
	// (need to add this explicitly because otherwise SP blocks SQ from moving)
	[request] sp!=1 -> (sp'=sp);

	// Serve a request from the queue
```

```
        [serve]      sp=2 -> rate_serve : (sp'=2);
        [serve_last] sp=2 -> rate_serve : (sp'=1);

endmodule

//-------------------------------------------------

// Power Manager (PM)
// Controls power state of service provider
// (this is done via synchronisation on idle2sleep/sleep2idle actions)

// Bound on queue size, above which sleep2idle command is sent
const int q_trigger;

module PM

        // Send sleep2idle command to SP (when queue is of size q_trigger or greater)
        [sleep2idle] q>=q_trigger -> true;

        // Send idle2sleep command to SP (when queue is empty)
        [idle2sleep] q=0 -> true;

endmodule

//-------------------------------------------------

// Reward structures

rewards "queue_size"
        true : q;
endrewards
```

View: printable version Download: power_policy1.sm

The PM needs a *policy* in order to operate: a set of rules which dictate what power management instructions to send and when. This simple policy is to switch out of *sleep* mode when the request queue exceeds a certain size (denoted by the constant q_trigger) and to switch back to *sleep* mode when the queue is empty.

Look at the code we have added to the SP module and at the new PM module. Make sure you understand how they work.

Now use the simulator to generate a trace through this new model. When you create a new path, you have to specify a value for the constant q_trigger, because it is left undefined in the model. Try a value of 5 for now. Does this new model behave as you expect?

3. Analysing probabilistic performance measures

We will now use PRISM to analyse the behaviour of the power management system. More specifically, we will first study the probability that the service queue is full at a particular time instant.

Go to the "Properties" tab of the GUI, create a new constant called T, of type double and with no defined value. Then add the following property:

```
P=? [ F[T,T] q=q_max ]
```

Now, create an experiment based on this property, plotting a graph of its result for q_trigger equal to 5 and T from 0 up to 20, i.e. for the first 20 seconds of the system.

It seems that the transient probability of the queue being full stabilises after a short while. Using another experiment, plot values for the same property on the same graph, this time for T from 20 up to 40, and see if the probability remains the same. To confirm this, now create a property to check the long-run probability of the queue being full, using the S operator. Right click the new property and select "Verify". You will need to give a value for T but this is not used so you can enter anything. It turns out that the default iterative method in PRISM (Jacobi) for solving this kind of property does not converge in this case. Switch to the Gauss-Seidel method from the "Options" dialog and try again. Check that your result matches the graph.

4. Analysing reward-based performance measures

Next, we will study reward based properties using PRISM's support for costs and rewards.


Read the section on costs and rewards in the manual. Then, look at the rewards that have already been defined in this model.


Add these two properties which can be used to compute the transient and long-run expected queue size, respectively:

```
R{"queue_size"}=? [ I=T ]
R{"queue_size"}=? [ S ]
```


Plot the expected queue size at time T, for the first 20 seconds of the model. As above, also compute the long-run value and check that it matches the results on the graph.


Now create some experiments to analyse the transient and/or long-run expected queue size for a range of different values of the constant q_trigger. How does the performance of the system vary as this changes? What is the "best" value of q_trigger?

Another possible measure of the performance of the system is how many requests get lost, i.e. how many times a request arrives when the queue is currently full.


Add a second reward structure to the model called "lost", which assigns 1 to every transition of the model labelled with action request from a state where the queue is full. **[*]**


Now create a new property for checking the expected *cumulated* reward up until time T. How does this measure vary for different values of q_trigger? Plot a graph to show this.**[*]**

5. Analysing the power consumption

So far we have only analysed the performance of the system, i.e. how promptly requests are dealt with (one measure of which is the expected queue size) and how often requests are lost. However, there is a trade off between the performance of the system and the power consumption of the system (to optimise performance one can just keep everything "on", however this approach will consume the most power).

The disk drive in our example has the following power consumption:

- energy is used at rate 0.13, 0.95 or 2.15 in the power states *sleep*, *idle* and *busy*, respectively.
- transitions between power states have a fixed energy cost: going from *sleep* to *idle* uses 7.0 units and from *idle* to *sleep* uses 0.067 units.


Add a third reward structure to the model representing the power consumption. Use a cumulative reward property to investigate the energy consumption over time of the system. **[*]**


To illustrate the trade off between power consumption and performance, on the same graph plot how the long run average queue size and power consumption vary as q_trigger changes. **[*]**

6. Changing the power management scheme

Another approach to implementing the power manager is to use *stochastic* policies, which can also make random choices.


Replace the PM module in the existing model with the following:

```
// Probability of ordering the SP to sleep when queue empty
const double p_sleep;

module PM

	// i2s: true when idle2sleep command should be issued
	i2s : bool init false;
```

```
        // idle to sleep command triggered by request arrivals/services
        //  decide to perform command when queue becomes empty with probability p_sleep
        [serve_last] true -> p_sleep : (i2s'=true) + 1-p_sleep : (i2s'=false);
        // cancel idle to sleep command if a reqest arrives
        [request] true -> (i2s'=false);

        // Send idle2sleep command to SP
        [idle2sleep] i2s -> (i2s'=false);

        // Send sleep2idle command to SP (when queue is full)
        [sleep2idle] q=q_max -> true;

endmodule
```

 How well does this power management system perform? What effect does the value of the probability p_sleep have?

 Change this power manager such that, instead of switching on (sending the sleep2idle command to the PM) each time the queue becomes full, probabilistically chooses to switches on whenever a request arrives where the probability of switching increases as the request queue size increases. **[*]**

 Does this power management system improve performance or power consumption?

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 4: the PRISM code fragment for the reward structure "lost";
2. from part 4: the corresponding property and graph for the expected cumulated reward up until time T;
3. from part 5: a reward structure and property for analysing the expected power consumption over time;
4. from part 5: a graph showing the trade off between power consumption and performance (expected queue size);
5. from part 6: the PRISM code fragment for the new power manager that probabilistically switches on the disk drive.

[Back to index]