Report for Stochastic Model Checking

Academic Year 2016/2017

Tommaso Papini
tommaso.papini@unifi.it
DT21263

PRISM Tutorial Part 3: Dynamic power management

Dynamic power management

In Section will be described the modelization through PRISM of a *DPM* (*Dynamic Power Management*) system, following the PRISM tutorial found at [1]. DPMs are used to apply different power usages to some computing device, according to a predefined strategy that takes into account the current state of the device. This kind of systems have been studied largely in literature, for example in [2] where a DPM for a Fujitsu disk drive has been studied.

A generic DPM system is made of three distinct components:

- *Service Queue* (*SQ*): holds the requests that the Service Provider will have to serve, in an ordered fashion, and can have finite queue capacity;
- *Service Provider* (*SP*): serves, one at a time, the requests stored in the Service Queue, serving each time the request at the head of the queue;
- *Power Manager* (*PM*): can change the power state of the Service Provider according to certain policies.

The *SP* could be anything that is a computing device that serves requests, such as a disk drive as in [2], but also a CPU or a Web Server.

At any given time, the *SP* is in one of three possible power states, each of which:

- *sleep*: the *SP* is in a low-power consumption mode and is unable to serve any request unless explicitly awaken by the *PM*;
- *idle*: the *SP* is awake but currently not serving any request, so any newly arriving request will be served immediately by the *SP*;

- *busy*: the *SP* is currently serving a request and will be available to serve the next in queue as soon as it's finished.

Ideally, when in the *sleep* state the *SP* will be requiring little to none power, when in the *idle* state it will require more, as it is awake and ready to serve requests, while when *busy* it will require even more, as the *SP* in that case is actively working on a request. The *PM* is charged with employing a power consumption strategy by switching the *SP*'s power state, in order to maximise the availability of the service while minimising the overall power consumption.

A first PRISM model for a DPM based on [2] is proposed in Code 1, as seen in [1].

```
// 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 Press, 1999

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;
```

```
47
48     // Synchronise with service queue (SQ):
49
50     // If in the idle state, switch to busy when a request arrives in the queue
51     [request] sp=1 -> (sp'=2);
52     // If in other power states when a request arrives, do nothing
53     // (need to add this explicitly because otherwise SP blocks SQ from moving)
54     [request] sp!=1 -> (sp'=sp);
55
56     // Serve a request from the queue
57     [serve] sp=2 -> rate_serve : (sp'=2);
58     [serve_last] sp=2 -> rate_serve : (sp'=1);
59
60 endmodule
61
62 //----------------------------------------------------------
63
64 // Reward structures
65
66 rewards "queue_size"
67     true : q;
68 endrewards
```

Code 1: PRISM code for the model of a DPM based on [2]. Source [1].

In this first model version shown in Code 1, only the two modules, for the *SQ* and the *SP* components, are introduced. This is an already working strategy, even without the *PM* component, which would only add a smarter policy for the *SP*'s power management.

It is worth noticing that the model shown in Code 1 is described as a *Continuous Time Markov Chain* (*CTMC*), which allows the use of rates when defining the firing of transitions instead of simple probabilities.

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.

Answer:

The SQ and SP modules implement, respectively, the *SQ* and *SP* components of the DPM system. Both modules synchronise on three different kinds of actions: request, indicating the arrival of a new request, serve, indicating that a request (except the last) has been served, and serve_last, indicating that the last request in the queue has been served.

The SQ module keeps a variable, q, that represents the current number of request in the queue, with maximum capacity given by the constant q_max. When, with the predefined arrival rate rate_arrive, a new request arrives (line 26) the queue is updated accordingly, eventually discarding requests in excess in case of a full queue. When a request is served by the SP module, whether it was the last (line 30) or not (line 28), the queue is decreased accordingly. The distinction for these two cases might seem useless on the SQ side, but it will prove useful for the SP module.

The SP module only keeps a variable, sp, indicating the current power state (0 meaning *sleep*, 1 *idle* and 2 *busy*), which starts in the *idle* state. When a new request arrives, the *SP* is switched to the *busy* state in case it was *idle* (line 51), indicating that it started working on that request right away, while if the power state is either *sleep* or *busy* then it's kept the same (line 54). When a request that was not the last in the queue is served (line 52), with service rate defined by the variable rate_serve, the *SP* is kept in the busy state, meaning that it starts working on the next request in line. Otherwise, if the last request is served (line 58), employing the same service rate, the *SP* is switched back to the *idle* state.

It is worth noticing that, since by definition only the *PM* component can decide when the *SP* has to wake, in this first model, if the *SP* starts in the *sleep* state, it would have no way of awaking.

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

Answer:

The model power.sm loaded into PRISM is shown in Figure 1.

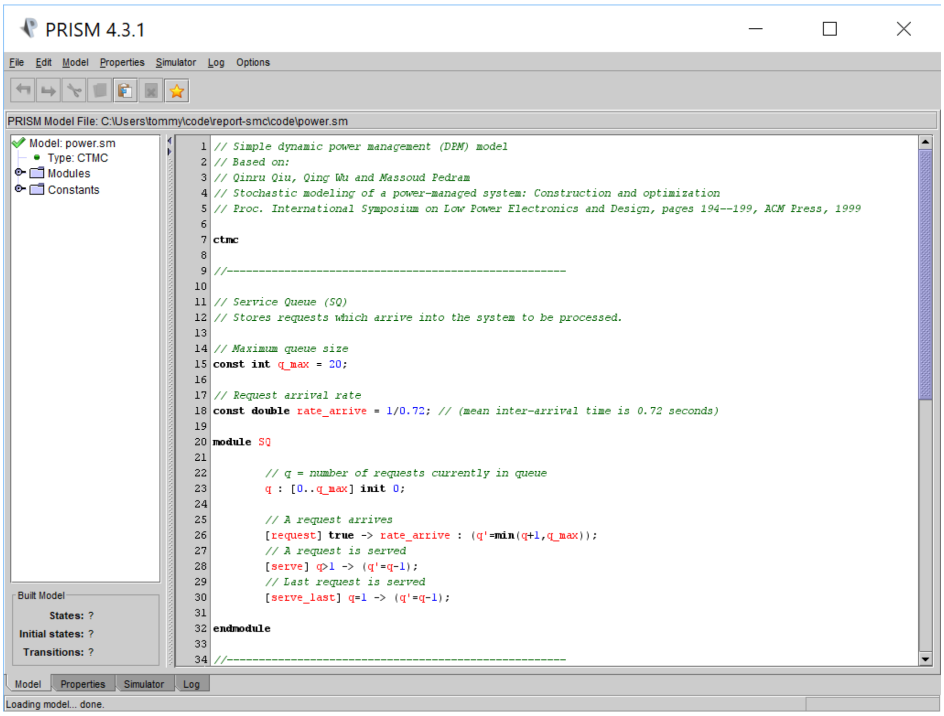

Figure 1: Model power.sm loaded into PRISM.

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?

Answer:

By repeatedly selecting the "Simulate" button in the PRISM simulator with 1 step it can effectively be seen that the queue size almost never exceeds 1. This because of how arrival and service rates are defined: while the arrival rate of new requests is $1.3\overline{8}$, meaning that on average a new request arrives every 0.72 seconds, the service rate is 125, meaning that on average a request is served in 0.008 seconds. This means that, when a request is in the queue, the service time of that request is, on average, 90 times faster than the arrival of a new request. So, although possible, it's just highly unlikely that, with rates so defined, a new request arrives before a service is finished.

If instead the action request is repeatedly selected in the PRISM simulator in order to force the arrival of new requests before the service, a full queue can be reached. In this case, as expected, more request arrivals end up in the refusal of these new requests, simply no including them in the queue, which remains at it's full capacity (`q_max`=20 in this case). It is worth noticing that, in case a new request arrives before a service is completed, even though the `serve` action remains enabled, it's probability distribution remains the same, since these are all exponentially distributed transitions and thus memoryless.

□

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.

Answer:

By selecting the "Build model" feature, it can be seen that the state space of this model is made of a total of 21 states. Intuitively, these are made of a state where the *SP* is *idle* and the queue empty and 20 states where the *SP* is *busy* and the queue has a different number of pending requests (from 1 to 20).

□

Adding the power management control

References

[1] PRISM Team. PRISM Tutorial Part 3: Dynamic power management. http://www.prismmodelchecker.org/tutorial/power.php. [Online; accessed 18-September-2017].

[2] Qiu, Q., Qu, Q., and Pedram, M. Stochastic modeling of a power-managed system-construction and optimization. *IEEE Transactions on computer-aided design of integrated circuits and systems 20*, 10 (2001), 1200–1217.