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 this section will be described the modelisation through PRISM [1] of a *DPM* (*Dynamic Power Management*) system, following the PRISM tutorial found at [2]. 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 [3] 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 [3], 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 [3] is proposed in Code 1, as seen in [2].

```
1 // Simple dynamic power management (DPM) model
2 // Based on:
3 // Qinru Qiu, Qing Wu and Massoud Pedram
4 // 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;

    // 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

```

```
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 [3], with only the *SQ* and the *SP* components. Source [2].

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.

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

Now we add to the PRISM model shown in Code 1 and additional module, PM, responsible of implementing the Power Manager. The *PM* is the component of a DPM that is charged with waking the *SP* from sleep or putting it back to sleep according to a specific policy that might be dependent of several factors, such as the current state of the *SP* or of the *SQ*.

Code 2 shows the updated model with the added PM module, which employs a fairly naive policy.

```
1 // Simple dynamic power management (DPM) model
2 // 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;
// 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);
```

```
60
61      // Synchronise with service queue (SQ):
62
63      // If in the idle state, switch to busy when a request arrives in the queue
64      [request] sp=1 -> (sp'=2);
65      // If in other power states when a request arrives, do nothing
66      // (need to add this explicitly because otherwise SP blocks SQ from moving)
67      [request] sp!=1 -> (sp'=sp);
68
69      // Serve a request from the queue
70      [serve] sp=2 -> rate_serve : (sp'=2);
71      [serve_last] sp=2 -> rate_serve : (sp'=1);
72
73  endmodule
74
75  //-------------------------------------------------------------------
76
77  // Power Manager (PM)
78  // Controls power state of service provider
79  // (this is done via synchronisation on idle2sleep/sleep2idle actions)
80
81  // Bound on queue size, above which sleep2idle command is sent
82  const int q_trigger;
83
84  module PM
85
86      // Send sleep2idle command to SP (when queue is of size q_trigger or greater)
87      [sleep2idle] q>=q_trigger -> true;
88
89      // Send idle2sleep command to SP (when queue is empty)
90      [idle2sleep] q=0 -> true;
91
92  endmodule
93
94  //-------------------------------------------------------------------
95
96  // Reward structures
97
98  rewards "queue_size"
99      true : q;
100 endrewards
```

Code 2: PRISM code for the model of a DPM based on [3], with the *SQ*, *SP* and *PM* components. Source [2].

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

Answer:

The PM module has the function of wake the *SP* or to put it back to sleep according, in this case, to the current size of the queue.

In particular, when in the queue there are at least q_trigger elements the modules PM and SP can synchronise on the action sleep2idle (lines 56, 57 and 87), effectively waking up the *SP* in case it was still asleep, as the action name suggests. If, when the awakening is triggered, the queue q is empty then the *SP* is switched to the *idle* state, otherwise it's switched directly to the *busy* state, i.e. working on the request on top of the queue.

Whenever, instead, the queue becomes empty, the PM and SP modules can synchronize on the idle2sleep action (lines 59 and 90), effectively waking up the *SP*.

It is worth noticing how the operations of waking the *SP* up and putting it back to sleep are not immediate but instead, when the corresponding conditions are met, are characterised by rates (rate_s2i and rate_i2s, respectively). Also, in this second version of the model, the SP's power state is now initialised as 0 (i.e. *sleep* state), since now, thanks to the PM module, initialising the *SP* in the *sleep* state wouldn't end up in a deadlock any more.

□

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?

Answer:

After a few simulation steps, it can be seen that the model actually behaves as expected. In particular, between 0 and 4 requests the *SP* has no other option than staying in the *sleep* state and, thus, the model has just one available transition at each step, that is the arrival of new requests. When the queue reaches 5 requests, the waking condition is met, but sometimes the awakening is not immediate and some additional requests (usually 1 or 2) manage to arrive before the *SP* is properly awaken: this is due the fact that the awaking rate (rate_s2i) is slightly smaller than the request arrival rate (request), meaning that on average the arrival of a new request is slightly faster than the awakening of *SP*.

Similarly, once the queue has been emptied, the *SP* can be switched back to *sleep* right away or it could happen that a request manages to arrive before that happens, forcing the *SP* to serve it first by switching to the *busy* state.

□

Analysing the model

We'll analyse now various aspects of the model, exploiting the properties and analyses features that PRISM offers.

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.

Answer:

The resulting experiment plot, with steps of 0.5 seconds, is shown in Figure 1.

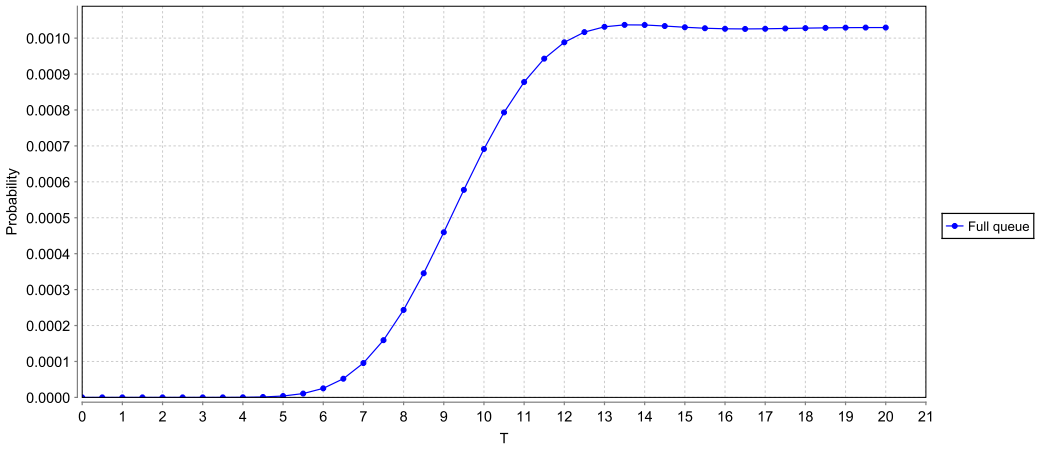

Figure 1: Experiment plot of the full queue between time 0 and 20.

□

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.

Answer:

From the extended plot shown in Figure 2 it seems that the probability of reaching a full queue indeed stabilises.

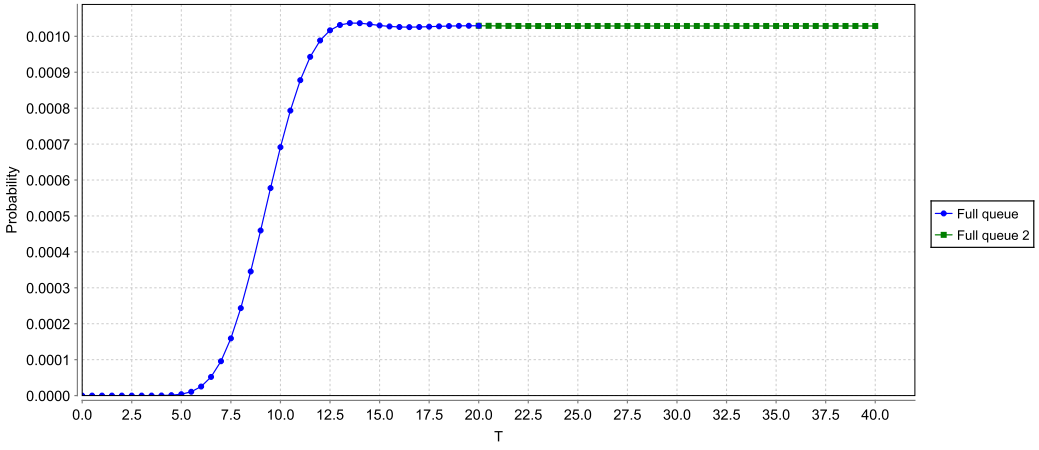

Figure 2: Experiment plot of the full queue between time 0 and 40.

In order to check the steady-state probability of the queue being full in a more direct fashion, the following property, that exploits the S operator, can be used:

```
S=? [ q=q_max ]
```

By trying to evaluate the property using the default iterative method (Jacobi) the evaluation indeed does not converge and PRISM prompts an error. Instead, using the Gauss-Seidel iterative method, the computed steady-state probability of the queue being full results being 0.0010287642322871614. Watching at the plot in Figure 2 it looks like the probability of a full queue stabilises slightly above 0.0010, which matches with the obtained result.

□

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

Answer:

The only reward that has been defined in the model in Code 2 is queue_size which, at any given time, is equivalent to the current number of requests present in the queue.

□

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.

Answer:

The experiment plot is shown in Figure 3.

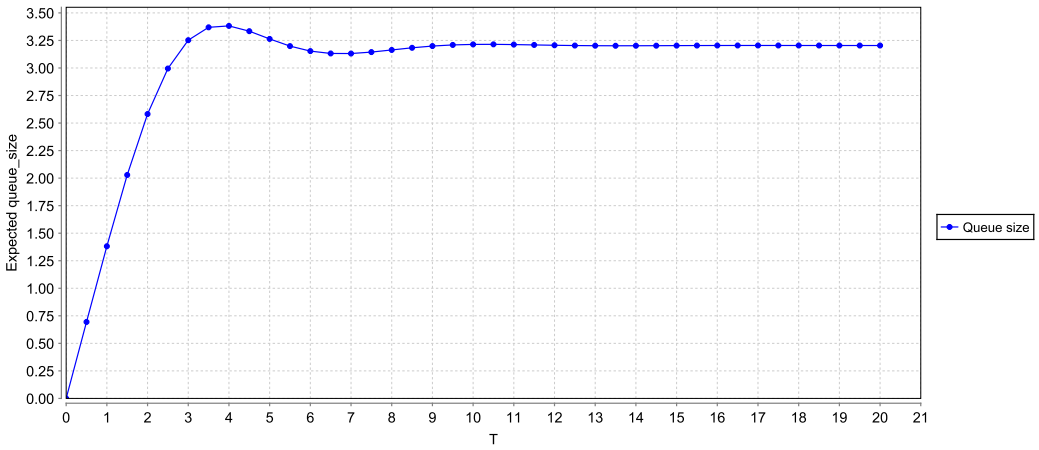

Figure 3: Experiment plot of the queue size between time 0 and 20.

Evaluating instead the steady-state property, it can be seen that the number of requests in the queue at steady-state is 3.2038846166301242, which seems to match to the transient results shown in the plot in Figure 3.

□

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?

Answer:

Transient analysis results between time 0 and 40 and for q_trigger between 0 and 21 are shown in Figure 4. The last value, q_trigger=21, has been included to show what happens in a scenario where the *SP* never wakes.

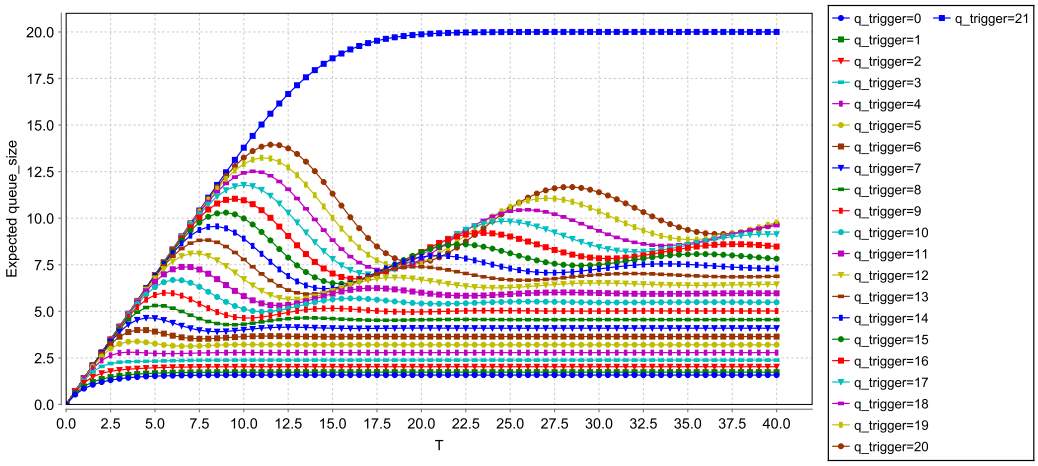

Figure 4: Experiment plot of the queue size between time 0 and 40 and for q_trigger between 0 and 21.

As expected, the average queue size is higher for higher values of q_trigger. After some time this value tends to stabilise, although for higher values of q_trigger it seems as it needs more time in order to reach the steady-state. For q_trigger=21, as expected, the queue fills completely and never gets the change to get emptied. According to the plot in Figure 4 we could say that the "best" values for q_trigger are the lowest (from 0 to 5), since for those values the average queue size stabilises earlier, thus producing a more predictable behaviour.

In Figure 5 is instead shown the average queue size at steady-state, for the same range of values for q_trigger.

As expected, the average queue size rises as q_trigger grows and it's exactly 20 for q_trigger$\geq$ 21. But is can also be observed that the growth is not exactly linear: for example for q_trigger=1 the average queue size is 1.734, while for q_trigger=3 it's 2.383.

Ideally, we would like to have the queue as empty as possible at any time, because that would mean that requests are handled rapidly and there is more room for future requests, while at the same time consuming as few resources as possible, for example by keeping the *SP* in the *sleep* state for longer. In Figure 6 we are showing then the *queue size coefficient*, evaluated as q/q_trigger, in order to find the best trade-off between these two factors. In particular, the coefficient is added to the PRISM model by including the following reward structure:

```
rewards "queue_size_coefficient"
    q_trigger=0 : 0;
    q_trigger!=0 : q/q_trigger;
endrewards
```

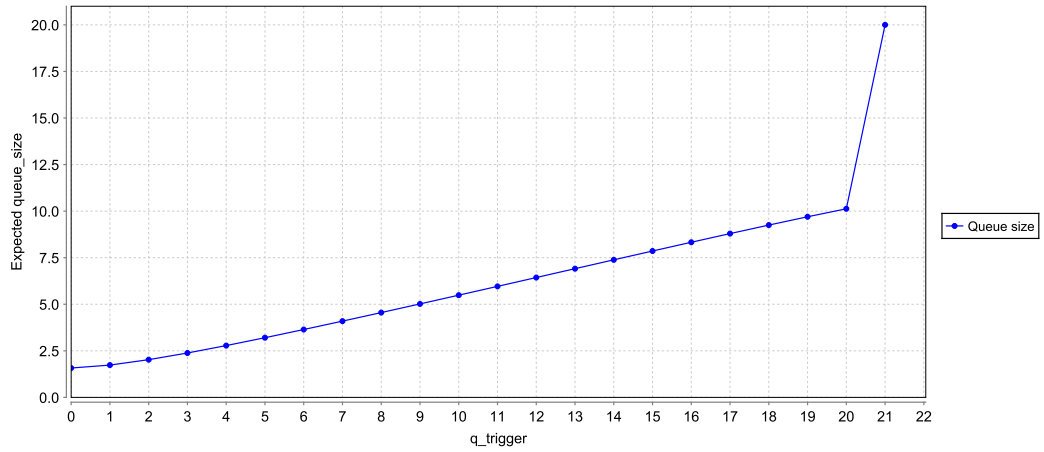

Figure 5: Experiment plot of the queue size at steady-state for q_trigger between 0 and 21.

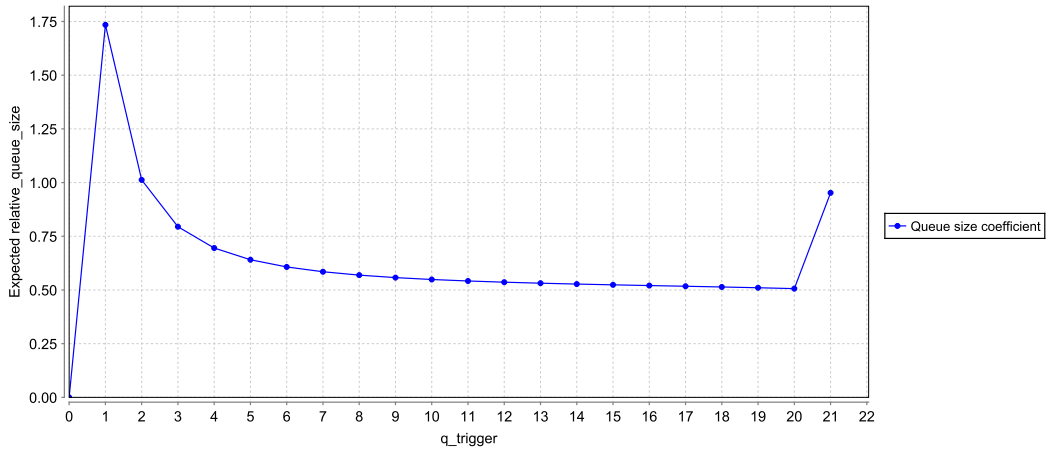

Figure 6: Experiment plot of the queue size coefficient at steady-state for q_trigger between 0 and 21.

The results in Figure 6 shows that, for q_trigger=1 and 2, the coefficient is above 1, meaning that the average queue size is usually higher than the value of q_trigger. For q_trigger$\geq$ 3 instead, the coefficient is lower than zero and decreasing, except for q_trigger =21, which becomes higher again. According to this experiment then, we are able to say that the "best" value for q_trigger is 20, since in this case the queue size coefficient is at it's lowest (looking at the plot in Figure 5 we can see that for q_trigger=20 the average queue size is 10.126).

□

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.

Answer:

The reward "lost" is simply included in the PRISM model by adding at the end the following lines:

```
rewards "lost"
    [request] q=q_max : 1;
endrewards
```

□

Now create a new property to check the expected *cumulated* reward up until time T. Don't forget to specify which reward structure you want in the property. (You might want to look at this section of the manual.) How does this measure vary for different values of q_trigger?

Answer:

In Figure 7 are shown the results for the cumulative lost requests between time 0 and 40 and for values of q_trigger between 0 and 20. The experiment is conducted using the following property:

```
R{"lost"}=? [ C<=T ]
```

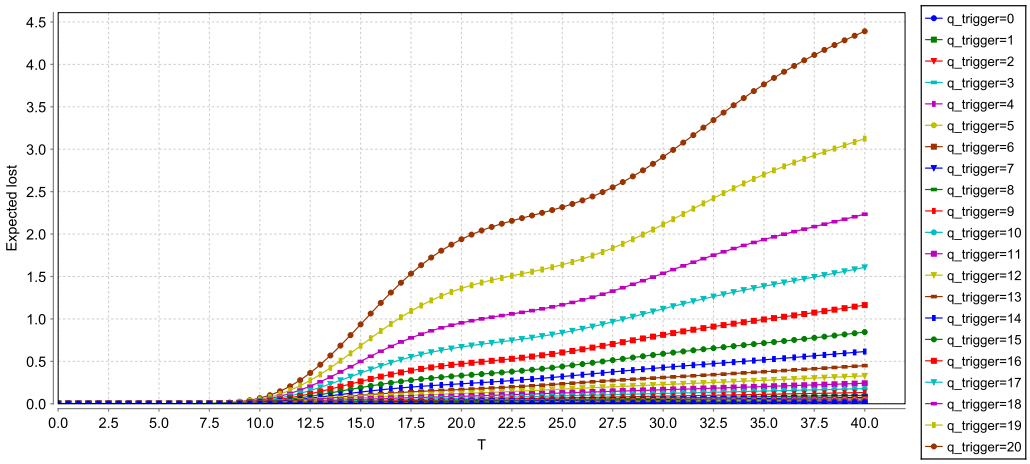

Figure 7: Experiment plot of the cumulative lost requests between time 0 and 40 and for q_trigger between 0 and 20.

As expected, the cumulative lost requests are more as time goes on, but also for higher values of q_trigger this values grows at higher rates. This is intuitive since when the *SP* only wakes up after more requests have arrived in the queue, the chances of reaching the queue saturation are higher.

□

Let's imagine now to introduce a measure of the actual power consumption. In particular, energy consumption rates are 0.13, 0.95 and 2.15, for the power states *sleep*, *idle* and *busy*, respectively. On top of that, switching from *sleep* to *idle* costs 7.0 energy units, while from *idle* to *sleep* costs 0.067 energy 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.

Answer:

First of all, in order to add the desired reward structure, the following rates have to be included in the model:

```
const double rate_consume_sleep = 0.13;
const double rate_consume_idle = 0.95;
const double rate_consume_busy = 2.15;
```

Then, inside the SP module, the following transitions have to be included:

```
[consume_sleep] sp=0 -> rate_consume_sleep : true;
[consume_idle] sp=1 -> rate_consume_idle : true;
[consume_busy] sp=2 -> rate_consume_busy : true;
```

Finally, the "consumption" reward structure can be defined as follows:

```
rewards "consumption"
    [sleep2idle] true : 7;
    [idle2sleep] true : 0.067;
    [consume_sleep] true : 1;
    [consume_idle] true : 1;
    [consume_busy] true : 1;
endrewards
```

Experiment results are shown in Figure 8.

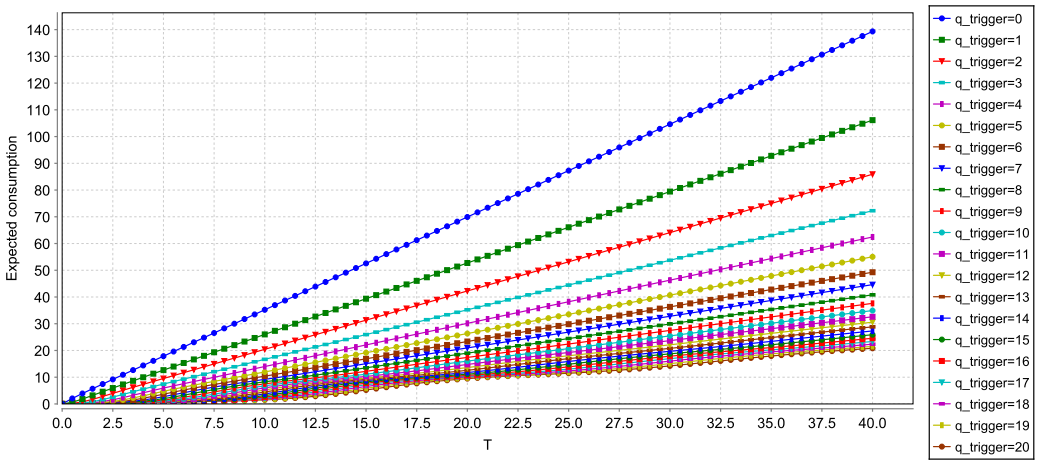

Figure 8: Experiment plot of the cumulative energy consumption between time 0 and 40 and for q_trigger between 0 and 20.

As expected, the cumulative energy consumption grows as time goes on, but it can also be observed that the energy consumption is lower for higher values of q_trigger, which is obvious considering that in these scenarios the *SP* spends, on average, more time in the *sleep* state, consuming less energy.

□

- -

Extensions

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;

    // Updates to i2s variable triggered by request arrivals/services
    [serve_last] true -> p_sleep : (i2s'=true) + 1-p_sleep : (i2s'=false);
    [request] true -> (i2s'=false);

    // Send idle2sleep command to SP (when queue empty, with probability p_sleep)
    [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?

Answer:

The results of the experiments concerning the cumulative energy consumption with the stochastic policy are shown in Figure 9.

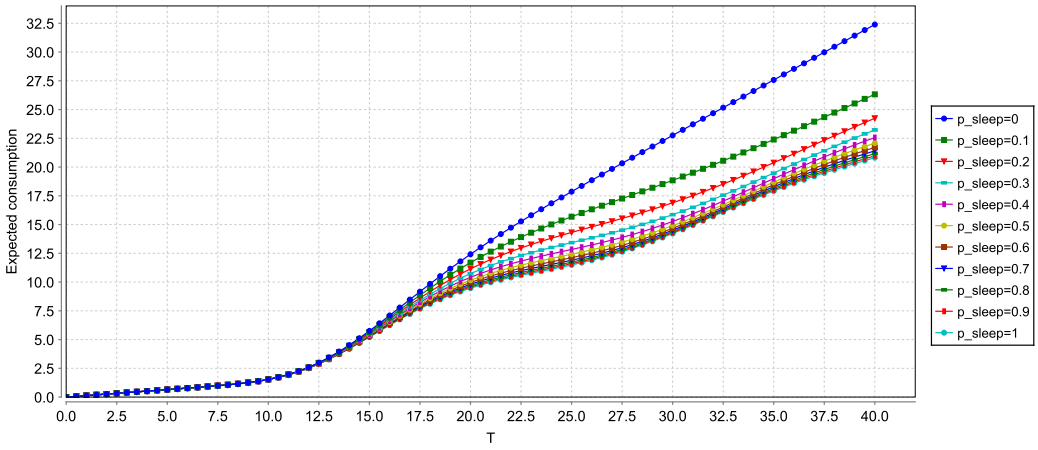

Figure 9: Experiment plot of the cumulative energy consumption employing the stochastic policy, between time 0 and 40 and for p_sleep between 0 and 1, with step 0.1.

Again, the energy consumption grows as the time goes on, but also it tends to be lower for higher values of p_sleep, yielding top performance when p_sleep=1, i.e. when the *SP* always gets put to sleep after the queue has been emptied. On top of that, we can notice that while for the previous simpler policy the energy consumption, after 40 seconds, could reach, on average, almost 140 units (see Figure 8), with this new policy the maximum at T=40 is 32.392, for p_sleep=0, making this second policy way more performing than the previous one.

□

Can you think of any ways of improving these power management strategies? Implement them in the PRISM model and see how well they perform.

Answer:

A possible way of minimising even further the energy consumption is to reverse the idea behind the stochastic policy introduced previously. In particular, we now want to always put the *SP* to sleep when the queue has been emptied and wake it up only when the queue is full but only with a certain probability. This policy is implemented by the following code for the PM module:

```
// Probability of remaining asleep even with a full queue
const double p_sleep;

module PM

    // s2i: true when sleep2idle command should be issued
    s2i : bool init false;

    // Updates to i2s variable triggered by request arrivals/services
    [request] q<q_max -> (s2i'=false);
    [request] q=q_max -> p_sleep : (s2i'=false) + 1-p_sleep : (s2i'=true);

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

    // Send sleep2idle command to SP (when queue is full)
    [sleep2idle] s2i -> (s2i'=false);

endmodule
```

The results of the experimentation with this new policy are shown in Figure 10.

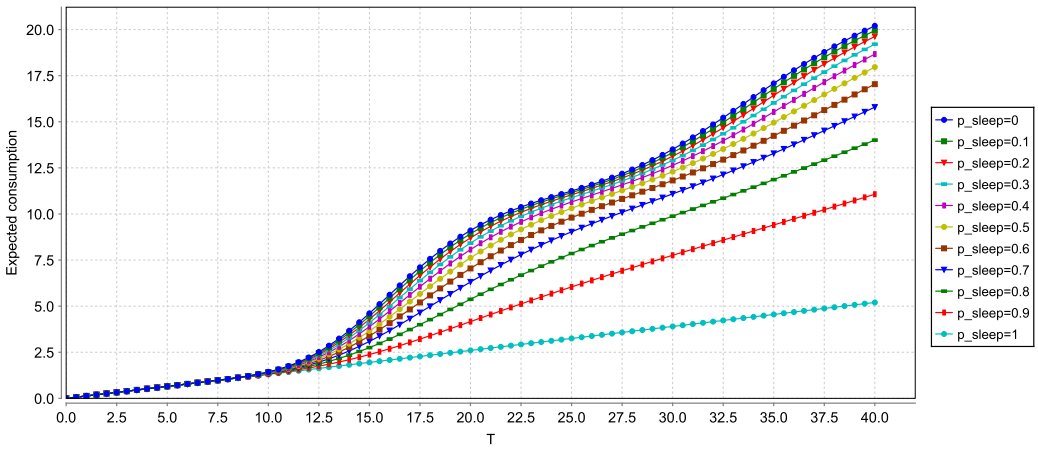

Figure 10: Experiment plot of the cumulative energy consumption employing a new custom policy, between time 0 and 40 and for p_sleep between 0 and 1.

It can be observed that, by increasing the value of p_sleep, i.e. the probability of remaining asleep even with a full queue, the cumulated energy consumption tends to be lower. This policy basically tries to minimise the number of times the *SP* wakes up, being the wake

up process a highly consuming action. Clearly, with this policy the number of lost requests is expected to be higher, but in a scenario where energy consumption is a central problem and some lost request is not such a big deal, this might be a good solution. In particular, the "best" case would be that with p_sleep=1, but that would mean that the *SP* never wakes up, which would be unrealistic.

In order to further analyse the model, and devise smarter policies, both the energy consumption and the number of lost requests should be taken into account, in order to find the best trade-off.

□

LINFA: Smart drugs restocking

Smart drugs restocking

The pharmaceutical logistic chain is a very complex system, as it includes several actors and critical points (such as elevated drug costs, need for transport at monitored temperature, chance of expiration of goods, stock management, irregularity of demands, several possible logistic approaches, etc), which render the problem hard to optimize without the proper tools for decision support. In this context, possible unavailability could lead to critical situations, sometime even catastrophic, as it wouldn't be possible any more to guarantee the correct execution of one or more healthcare protocol, thus affecting the patients' health. Furthermore, orders are typically carried out on a daily basis and in a manual fashion, without the support of any decision support system. All these reasons makes restocking schedule hard, thus leading to unproductive stocks and higher stocking costs.

In this scenario fits the project *LINFA* (i.e. *Logistica INtelligent del FArmaco*, or *Smart Drug Logistic*, from Italian), that aims to develop an IT system for support to processes of drugs logistic management, in the context of healthcare or local companies. LINFA aims to increase efficiency, effectiveness and predictability of the process of drugs and medical devices restocking, within healthcare structures, through methods of predictive analysis and optimization, advanced logistic techniques and tracking features through the use of RFID technologies or the integration of healthcare and administrative information stream.

For sake of simplicity, the following assumptions will be made in order to build a model:

- a single ward is present;
- the ward has a fixed number of beds (40) and a fixed maximum storage capacity (40);
- patients are indistinguishable;
- there is only one type of drug;
- patients can arrive through emergencies (i.e. according to a random variable) or scheduled examinations (i.e. according to a fixed constant);
- each day, patients can leave the ward with a certain probability;
- each day, patients can consume up to 3 units of drug;
- for each missing drug, an urgent order is issued, which arrives immediately;
- restock orders are issued at the end of each day and arrive immediately;

- the drug can be restocked in quantities of 0 (i.e. no order), 10, 20, 30 or 40.

The model in Code 3 models a generic ward with the above specifications and 3 days of lookahead. The model is used to support decision to the restock order of the current day, so it starts at the end of the current day (i.e. after patients arrival/discharge and drug consumption, but before the restock order). The arrival of patients and the consumption of drugs have been modelled through custom distributions, while drug restock orders have been modelled through non-deterministic choices, effectively representing the user's choice.

```
mdp

const int maxPat = 40; // maximum number of patients that can be hosted in the hospital ward
const int maxDrugs = 40; // maximum number of drugs that can be hold in the hospital ward

const double probExit = 0.6; // probability to leave the hospital ward

// probability to consume 0, 1, 2 or 3 drugs
const double consume0 = 0.2;
const double consume1 = 0.4;
const double consume2 = 0.3;
const double consume3 = 0.1;

const int costOrder = 1; // the cost to be paid to order a single drug
const int costUrgentOrder = 100; // the cost to be paid for urgent orders
const double costStorage = 5; // the cost to be paid for drugs that remain in hospital ward

// distribution of the arrivals from the ER (everyday)
const double probER_0 = 0.1;
const double probER_1 = 0.2;
const double probER_2 = 0.4;
const double probER_3 = 0.2;
const double probER_4 = 0.1;

const int patSched1 = 2; // scheduled patients for day 1
const int patSched2 = 3; // scheduled patients for day 2
const int patSched3 = 2; // scheduled patients for day 3

module hospitalWard
    s : [0..30] init 0; // state variable
    stockDrugs : [0..maxDrugs] init 40;
    n : [0..maxPat] init 0;
    day : [0..3] init 0;
    tmp : [0..maxPat] init 0; // support variable
    missingDrugs: [0..3] init 0; // support variable indicating missing drugs

    // decide how many drugs must be ordered
    [doNotOrder] s=0 -> (s'=1); // not order
    [order10] s=0 -> (stockDrugs'=max(0,min(maxDrugs,stockDrugs+10))) & (s'=1); // order 10 drugs
    [order20] s=0 -> (stockDrugs'=max(0,min(maxDrugs,stockDrugs+20))) & (s'=1); // order 20 drugs
    [order30] s=0 -> (stockDrugs'=max(0,min(maxDrugs,stockDrugs+30))) & (s'=1); // order 30 drugs
    [order40] s=0 -> (stockDrugs'=max(0,min(maxDrugs,stockDrugs+40))) & (s'=1); // order 40 drugs

    // day 1, 2, 3
    [] s=1 -> (s'=10) & (tmp'=n) & (day'=day+1);

    // moving patients out of the hospital ward
    [] s=10 & tmp>0 & (n>0) -> (1-probExit) : (n'=n) & (tmp'=tmp-1) + probExit : (n'=n-1) & (
        tmp'=tmp-1);
```

```
49      [] s=10 & tmp=0 -> (s'=11);
50
51      // scheduled arrivals
52      [] s=11 & day=1 -> (s'=12) & (n'=min(maxPat,n+patSched1)); // day1
53      [] s=11 & day=2 -> (s'=12) & (n'=min(maxPat,n+patSched2)); // day2
54      [] s=11 & day=3 -> (s'=12) & (n'=min(maxPat,n+patSched3)); // day3
55
56      // ER arrivals
57      [] s=12 ->  probER_0 : (s'=13) +
58          probER_1 : (s'=13) & (n'=n+min(maxPat-n,1)) +
59          probER_2 : (s'=13) & (n'=n+min(maxPat-n,2)) +
60          probER_3 : (s'=13) & (n'=n+min(maxPat-n,3)) +
61          probER_4 : (s'=13) & (n'=n+min(maxPat-n,4));
62
63      [] s=13 -> (s'=20) & (tmp'=n);
64
65      // drugs consumption
66      [] s=20 & tmp>0 -> consume0: (tmp'=tmp-1) +
67          consume1: (missingDrugs'=max(0,-(stockDrugs-1))) & (stockDrugs'=max(0,stockDrugs-1)) &
                (tmp'=tmp-1) +
68          consume2: (missingDrugs'=max(0,-(stockDrugs-2))) & (stockDrugs'=max(0,stockDrugs-2)) &
                (tmp'=tmp-1) +
69          consume3: (missingDrugs'=max(0,-(stockDrugs-3))) & (stockDrugs'=max(0,stockDrugs-3)) &
                (tmp'=tmp-1);
70
71      [missingDrugs] s=20 & missingDrugs>0 -> missingDrugs'=0;
72
73      [consumptionDone] s=20 & tmp=0 -> (s'=21);
74
75      // end of the day
76      [] (s=21) & (day<3) -> (s'=0) ; // end of the day 1 or 2
77      [] (s=21) & (day=3) -> (s'=30); // end of the day 3
78
79      []s=30 -> (s'=30);
80
81  endmodule
82
83  //-----------------------//
84
85  rewards "totalCost"
86      [consumptionDone] true: costStorage*stockDrugs;
87
88      [missingDrugs] true : costUrgentOrder*missingDrugs;
89
90      [doNotOrder] true : 0;
91      [order10] true : 10*costOrder;
92      [order20] true : 20*costOrder;
93      [order30] true : 30*costOrder;
94      [order40] true : 40*costOrder;
95  endrewards
```

Code 3: PRISM code for the model of a hospital ward.

Look at the model in Code 3 and describe how it works.

Answer:

The first thing that can be seen about the model is that it is defined as a Markov Decision Process (MDP). This is due the presence of non-deterministic choices at the end of each day, representing drug orders.

In the first part of the code, before the beginning of the main module (lines 3-27), constants and custom distributions are defined. In particular, it can be seen that the probability for each patient to leave the hospital at the end of each day is set to 0.6, meaning that the sojourn time of each patient in the hospital is distributed as a geometric distribution. The other two distributions, regarding the patients arrival from emergencies and the drugs consumption, are instead manually defined in the ranges of $[0, 4]$ (lines 19-23) and $[0, 3]$ (lines 9-12) respectively. Arrivals of scheduled patients for the three subsequent days of lookahead are defined as fixed variables (lines 25-27). Also, cost weights are defined for the order cost of a single drug unit, the cost for urgent drug reorders and the stocking cost for keeping each drug unit in stock in the right conditions.

The main module hospitalWard firstly defines several internal variables (lines 30-35), such as the number of hospitalised patients n, the number of current drugs in stock stockDrugs, the current day day, or support variables, such as s or tmp. In particular s will be used to describe the internal state for each day of the ward.

States s=0,1 (lines 38-45) are used to model the drug order for the current day and for the first and second following days. In particular, lines 38-42 represents all the possible orders that can be issued, modelled through non-deterministic choices. Also, each choice has a label associated, that will be used in the cost reward computation. After the order choice has been made, the day variable is incremented and the tmp support variable is set to the number of patients currently hospitalised, n.

States s=10,11,12,13 are used to model the arrival and discharge of patients in the ward. State s=10 (lines 48-49) of the ward is used to cycle through all the currently hospitalised patients to decide which of them will leave the ward, with probability probExit. In this scenario the tmp variable is used to cycle through all the patients. State s=11 (lines 52-54) is used to model the arrival of scheduled patients, the exact number of which is indicated by three constants. State s=12 (lines 57-61) instead is used to model the arrival of new patients through emergencies, employing the custom distribution defined. State s=13 is then used to move on to the drug consumption phase.

State s=20 is used to model drug consumption by the currently hospitalised patients. A custom consumption distribution is used and a cycle is implemented through the use of the tmp variable as seen for the leaving patients (lines 66-69). On top of that, for each patient, if after the drug consumption some drugs result to be missing (i.e. some drug was needed but there wasn't enough in stock), then the special action with label missingDrugs is executed (line 71), which is needed to compute the correct cost reward. When drugs have been administered to all the patients (line 73) the module will then either start again (line 76) or stop definitely (line 77) in case the three following days have been completely modelled.

Lastly, the only reward defined, totalCost, is used to cumulate the total cost coming from different sources. In particular, every time the consumption of drug is finished for a certain day, the remaining drugs in stock produce a cost determined by the weight costStorage. Each time one or more drugs result missing during administration, and therefore an urgent order has to be issued, each missing drug produce a cost defined by the weight costUrgentOrder. Regular orders also produce a cost, in particular a cost of 1 for each drug ordered.

□

References

[1] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)* (2011), G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806 of *LNCS*, Springer, pp. 585–591.

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

[3] 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.