Technical University of Denmark

02246 Model Checking

Mandatory Assignment Part 1: Discrete Modelling and Verification

Authors:

Andreas Hallberg Kjeldsen
s092638@student.dtu.dk

Morten Chabert Eskesen
s133304@student.dtu.dk

October 21, 2013

Introductory notes

This report has been written by the both of us. All parts have been worked on together and therefore our responsibility for each assignment is equal.

Part A: Introductory Problems

A1) Practical Problems

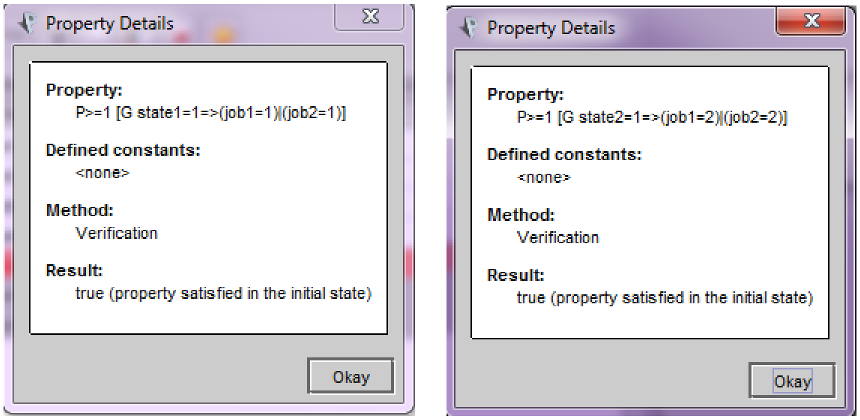
A1.1

For the FCFS scheduler, we would like to verify that whenever a client has an active job, the scheduler has that job somewhere in its queue. For example, in the case of the first client, we require that whenever $state_1 = 1$, then either $job_1 = 1$ or $job_2 = 1$.

A1.1a) Express this as two CTL properties - one for each client

$client_1$: $AG(state_1 = 1 \Rightarrow \neg(\neg job_1 = 1 \wedge \neg job_2 = 1))$
$client_2$: $AG(state_2 = 1 \Rightarrow \neg(\neg job_1 = 2 \wedge \neg job_2 = 2))$

A1.1b) Use PRISM to verify whether these properties hold of the FCFS scheduler model



Property:
P>=1 [G state1=1=>(job1=1)|(job2=1)]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1 [G state2=1=>(job1=2)|(job2=2)]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

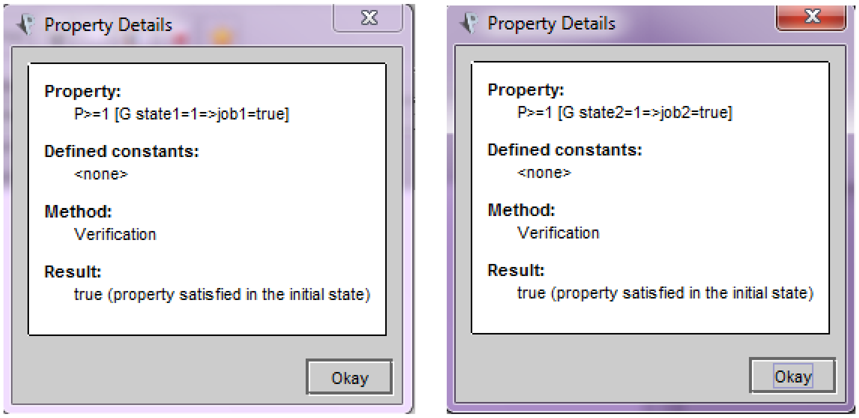
Figure A1.1b: The properties verified by PRISM.

A1.1c) Write down two similar properties for the SRT scheduler, explaining your construction

$client_1$: $AG(state_1 = 1 \Rightarrow job_1 = true)$
$client_2$: $AG(state_2 = 1 \Rightarrow job_2 = true)$

We require that whenever $state_1 = 1$ then $job_1 = true$ because there should be a job waiting in the queue when the $state_1 = 1$. The same goes for $client_2$.

A1.1d) Verify whether they hold of the model



Property:
P>=1 [G state1=1=>job1=true]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1 [G state2=1=>job2=true]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Figure A1.1d: The properties verified by PRISM.

A1.2

Add another client to the PRISM model of the FCFS scheduler. You will need to modify the *Scheduler* module to cope with the extra client, but for now do not increase the length of the queue.

A1.2a) Explain the changes that you made to the model, and argue why they satisfy the above instructions

In order for the *Scheduler* to cope with an extra client we first add an extra module called $client_3$ with the same commands as the two other clients with the names of the commands corresponding to $client_3$. We changed the finite range, which job_1 and job_2 can take their value over to $0 \ldots 3$. This does not increase the length of the queue because there is still only two jobs allowed in the queue (job_1 and job_2). We also added commands create3, serve3 and finish3 and only changed the values according to the number of $client_3$.

A1.2b) How many reachable states are in the new model?

In the new model there are 214 reachable states.

A1.2c) What will happen if the queue is full when a client attempts to create a job?

A client cannot create a job when the queue is full. This is because all the modules synchronize over all action names that appear syntactically in the modules. The commands create1, create2 and create3 are also in scheduler with a guard that specifies that the $job_2 = 0$ for creation of a job to be possible, and $job_2 = 0$ is only true if the queue is empty.

A1.2d) Do the properties you have previously verified still hold of the model? If not, why not?

Yes the properties previously verified still hold in the new model. They do because the clients' states will only be 1 if their job is in the scheduler since the modules are synchronized.

A1.3

Now modify the *Scheduler* module so that the queue is of length three.

A1.3a) Explain the changes that you made to the model, and argue why they are correct.

In order to modify the queue to have a length of three we add another job to the queue called job_3 which will hold the third job of the queue. We also changed the create commands to have the guard $job_3 = 0$ because now this is the last job in the queue, so when it is 0 there is a place for one more job. Furthermore we added another method for shifting the queue when there is an empty slot. The old command stays in place, but there is now another command with the guard $job_2 = 0$ & $job_3 > 0$ that shifts job_3 to job_2 so it is moved up in the queue. Since the commands have no action names the commands can always occur *independently* of what any other modules in the systems are doing - just so long as its guard is true.

A1.3b) How many reachable states are in the new model?

In the new model there are 1459 reachable states.

A1.3c) Do the properties now hold of the model? If not, why not?

The properties previously verified do not hold in the new model, because the queue is now of length 3. Which means that a job created by a client could be in scheduled as the last job, i.e. in job_3. Example: this would cause (for $client_1$) to have $state_1 = 1$ while $job_3 = 1$ because the job is at the end of the queue.

A1.3d) Can you give an upper bound on the number of reachable states for a model with n clients, and a queue of length m?

There is no way to determine an upper bound on the number of reachable states for a model with n clients and a queue of length m. A model with 2 clients and a queue of length 2 has 83 reachable states. With 3 clients and a queue of length 2 there is 214 reachable states whilst with 3 client and a queue of length 3 there is 1459 reachable states. There is no coherence between these numbers so it is impossible to give an upper bound. Although making the queue longer will increase the number of reachable states a lot more than adding another client.

A1.4

Consider the CTL properties Φ and AG Φ, where Φ is an atomic property.

A1.4a) What are their semantics, and how do they differ?

$AG\ \Phi$ specifies that from all the paths from this state Φ should hold. Whereas property Φ should only hold in that state.

A1.4b) Are their semantics different in the version of PRISM that you use?

The semantics are different in the version of PRISM we use. If the property Φ should hold in all reachable states it should be written $AG\ \Phi$. Because if only Φ has been written as the property - this version of PRISM will only check if the property Φ holds in the *initial* state.

A1.4c) How would you solve the classical model checking problem $M, s \models \Phi$ as a problem of the form $\forall s' : M, s' \models \Phi'$?

The problem $M, s \models \Phi$ means that in M there is a state s that satisfies Φ. The problem $\forall s' : M, s' \models \Phi'$ means that for every state s' in M, Φ' is satisfied. Assuming all states are reachable, we can model the first problem as the second problem by specifying $\Phi' = \exists\Diamond\Phi$.
$\forall s' : M, s' \models \exists\Diamond\Phi$. This states that for every s' in M there exists a path that at some point satisfies Φ.

A1.4d) How would you solve the model checking problem $\forall s' : M, s' \models \Phi$ as a problem of the form $M, s \models \Phi'$

The problem $\forall s' : M, s' \models \Phi$ means that for every state s' in M, Φ is satisfied. The problem $M, s \models \Phi'$ means that in M there is a state s that satisfies Φ'. Assuming all states are reachable, we model the first problem as the second problem by specifying $\Phi' = \forall\Box\Phi$.
$M, s \models \forall\Box\Phi$. This states that in M there is a state s for which all reachable paths and all subsequent subpaths satisfies Φ.

A2) Theoretical Problems

A2.1

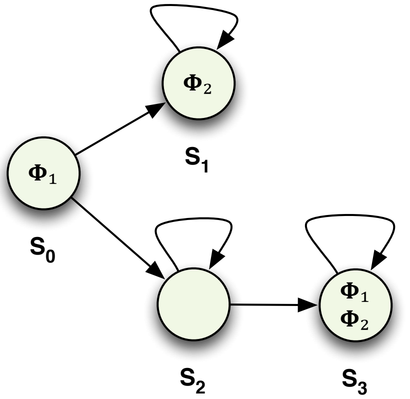


Figure 1: A transition system

Consider the transition system, shown graphically in **Figure 1**. The states are represented by circles, whose names are shown beneath them, and whose labels are shown inside them. The initial state is s_0.

A2.1a) Write down this transition system formally, as a tuple $(S, \rightarrow, S_0, AP, L)$

The transition system will formally be written as:
$\langle \{s_0, s_1, s_2, s_3\}, \{(s_0 \rightarrow s_1), (s_1 \rightarrow s_1), (s_0 \rightarrow s_2), (s_2 \rightarrow s_2), (s_2 \rightarrow s_3)\}, s_0,$
$\{\Phi_1, \Phi_2\}, \{(s_0, \{\Phi_1\}), (s_1, \{\Phi_2\}), (s_2, \{\}), (s_3, \{\Phi_1, \Phi_2\})\}\rangle$

A2.1b) By directly reasoning with the semantics of CTL, determine whether the following properties hold of the state s_0

i. $AF\Phi_2$: Does not hold as s_2 can loop infinitely and therefore s_3 is never reached.

ii. $AX\Phi_2$: Does not hold as $s_0 \rightarrow s_2$ isn't allowed.

iii. $EF\Phi_1$: Holds ($s_0 \rightarrow s_2 \rightarrow s_3$).

iv. $A[\Phi_1 U \Phi_2]$: Does not hold as $s_0 \rightarrow s_2$ isn't allowed.

A2.2

For each of the following pairs of CTL formulae, determine whether (a) they are equivalent, (b) one implies the other, or (c) neither implies the other. Explain your reasoning.

A2.2a) $EX\ EF\ \Phi$ and $EF\ EX\ \Phi$: They are equal
($EX\ EF\ \Phi \equiv EF\ EX\ \Phi$).
The first formulae dictates that somewhere there is a state which at some point has a descendant that contains Φ. The second formulae dictates that somewhere there is a state which immediate descendant contains Φ.

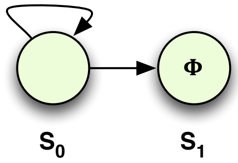


Figure A2.2a: A transition system.

In Figure A2.2a both formulae will be satisfied. For the first formulae we match s_0 with EX then somewhere in the descendant path of s_0 we reach s_1 which satisfies $EF\ \Phi$. For the second formulae we keep staying in s_0 for however long it might be and the final time we hit s_0 we will match EF then immediately afterwards we will go to s_1 which satisfies $EX\ \Phi$.

A2.2b) $AX\ AF\ \Phi$ **and** $AF\ AX\ \Phi$**:** They are equal ($AX\ AF\ \Phi \equiv AF\ AX\ \Phi$).
The first formulae dictates that for every state all the descendant states of the immediate descendant must somewhere have a state that contains Φ. The second formulae dictates that all states must have a descendant which all immediate descendants contains Φ.

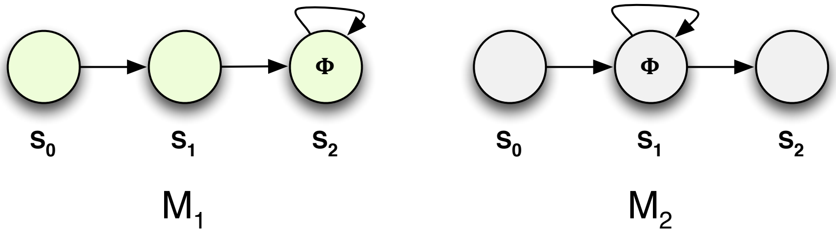


Figure A2.2b: Two transition systems M_1 and M_2.

In Figure A2.2b the transition system M_1 satisfies both formulae. Transition system M_2 will not satisfy any of the formulae as state s_2 will never be satisfied.

A2.2c) $AG\ EF\ \Phi$ **and** $EF\ AG\ \Phi$**:** The first formulae implies the second formulae ($AG\ EF\ \Phi \Rightarrow EF\ AG\ \Phi$).

If $AG\ EF\ \Phi$ satisfies a transition system then $EF\ AG\ \Phi$ will satisfy the same transition system, but not the other way around. Figure A2.2c shows two transition systems, M_1 and M_2. M_1 will satisfy both formulae whereas M_2 will only satisfy the second formulae.

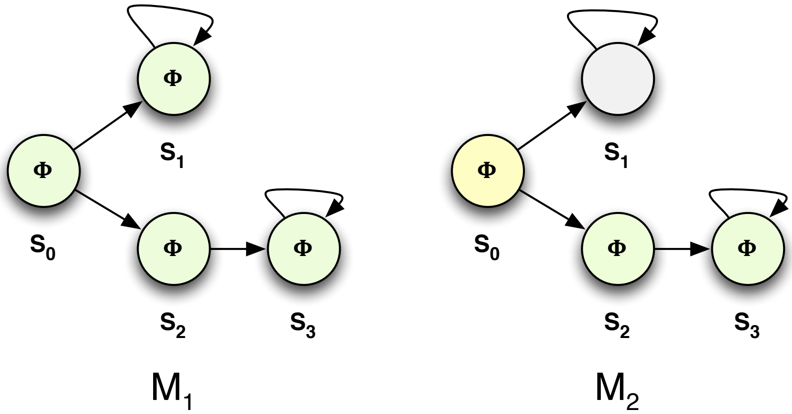


Figure A2.2c: Two transition systems M_1 and M_2.

Green indicates a state that is satisfied for both formulae. Yellow indicates a state that is satisfied by just the second formulae. Grey indicates a state that is satisfied by none of the formulae.

A2.2d) $AG\ (\Phi_1 \wedge \Phi_2)$ **and** $(AG\ \Phi_1) \wedge (AG\ \Phi_2)$**:** They are equal $(AG(\Phi_1 \wedge \Phi_2) \equiv (AG\ \Phi_1) \wedge (AG\ \Phi_2))$.
The first formulae dictates that every state reachable must have Φ_1 and Φ_2. The second formulae firstly dictates that every state reachable must have Φ_1 and secondly dictates that every state reachable must have Φ_2. Thous for both of the formulae to be satisfied every reachable state must have both Φ_1 and Φ_2.
Also it's a distribute law in CTL, see page 330 of the course book.

A2.2e) $EF\ (\Phi_1 \wedge \Phi_2)$ **and** $(EF\ \Phi_1) \wedge (EF\ \Phi_2)$**:** The first formulae implies the second formulae $(EF\ (\Phi_1 \wedge \Phi_2) \Rightarrow (EF\ \Phi_1) \wedge (EF\ \Phi_2))$.
The first formulae dictates that there must exist a reachable state having both Φ_1 and Φ_2. The second formulae firstly dictates that there must exist a reachable state containing Φ_1 and secondly it dictates that there must exist a reachable state containing Φ_2. Thous if the first

formulae is satisfied the second formulae must also be. On the contrary if the second formulae is satisfied the first formulae may not necessary also be, as Φ_1 and Φ_2 can be in different states but in the same tree, this would satisfy the second formulae but not the first.

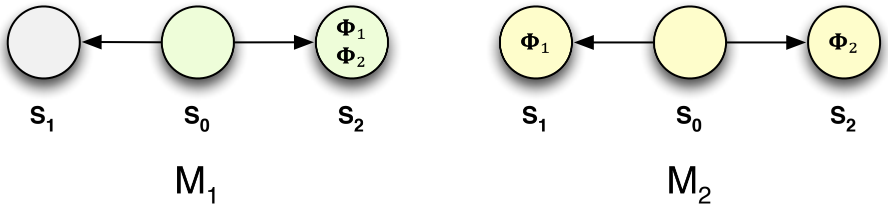


Figure A2.2e: Two transition systems M_1 and M_2.

Green indicates a state that is satisfied for both formulae. Yellow indicates a state that is satisfied by just the second formulae. Grey indicates a state that is satisfied by none of the formulae.

2A.3

Write down a CTL* formula for each of the following properties, which are described in natural language. In each case, argue whether or not the property can also expressed in CTL.

2A.3a) There is a path on which Φ holds infinitely often

CTL*: $\exists\Box\Phi$
CTL: Would be the same as for CTL*.

2A.3b) For all paths, Φ_1 holds along the path until Φ_2 holds of a state and Φ_3 holds of the state that immediately follows

CTL*: $\forall(\Phi_1 U \Phi_2) \bigcirc \Phi_3$
CTL: In CTL a path quantifier $(\exists, \forall)$ has to be in front of a state operator $(\Diamond, \Box, \bigcirc, U, W)$, therefore the formulae for CTL* is not possible to use with CTL. Furthermore, the CTL* formulae cannot be translated into CTL due to Φ_3 having to follow Φ_2 immediately.

2A.3c) There is a path on which either Φ_1 eventually holds or Φ_2 eventually holds

CTL*: $\exists\Diamond\left((\Phi_1 \wedge \neg\Phi_2) \vee (\neg\Phi_1 \wedge \Phi_2)\right)$
CTL: Would be the same as for CTL*.

2A.3d) For all paths, either Φ_1 always holds or Φ_2 always holds

CTL*: $\forall(\Box\Phi_1 \vee \Box\Phi_2)$
CTL: Not possible to create due to the $\Box$ having to be inside the parentheses.

2A.4

Encode the transition system in **Figure 1** as a PRISM module, using a variable s, such that $0 \leq s \leq 3$, to represent the state. Define Φ_1 and Φ_2 as PRISM labels.

The PRISM module:

```
mdp

module figure
  s : [0..3] init 0;

  [proceed] s = 0 -> (s’ = 1);
  [proceed] s = 0 -> (s’ = 2);
  [proceed] s = 1 -> (s’ = 1);
  [proceed] s = 2 -> (s’ = 2);
  [proceed] s = 2 -> (s’ = 3);
  [proceed] s = 3 -> (s’ = 3);
endmodule

label "phi1" = s=0 | s=3;
label "phi2" = s=1 | s=3;
```

Listing 1: PRISM module encoding the transition system in **Figure1**.

The PRISM properties:

```
P>=1 [F "phi2"]

P>=1 [X "phi1"]

!P<=0 [F "phi1"]

P>=1 [ ("phi1") U ("phi2") ]
```

Listing 2: PRISM properties for **Figure1**.

Part B: Intermediate Problems

B1) Practical Problems

Round-robin Scheduler

In this part we will model a round-robin scheduler (found in the file RR.nm). A round-robin scheduler cycles between jobs and executes each task for just one time unit before moving onto the next. We model the scheduler by modifying the SRT scheduler which was handed out with the assignment (SRT.nm). In order to model a round-robin scheduler from the SRT scheduler we need to introduce some variables to the *Scheduler* module.

```
turn : [0..2] init 0; // Who is next?
job1time : bool init false; // Has job1 used up its time?
job2time : bool init false; // Has job2 used up its time?
```

The turn variable specifies which job is next to be served. When turn=1 the next job to be served is job_1 which is the job of $client_1$.
The $job_X time$ variable specifies if job_X has used up all its serving time in the scheduler. So if $job_X time$ is true the next job should be served whereas if $job_X time$ is false this job should be served next. It should only be served next because there are only 2 jobs in this scheduler but if there were more jobs it should be served soon because it is not necessarily that job's turn next when more jobs are cycled.
We also introduce some new commands with no action names in the module. These commands can always occur independently of any other module in the system. The guard of these commands just has to be true.

```
[] job1=true & job1time=true & turn=1 ⇒ (job1time′=false);
[] job2=true & job2time=true & turn=2 ⇒ (job2time′=false);
```

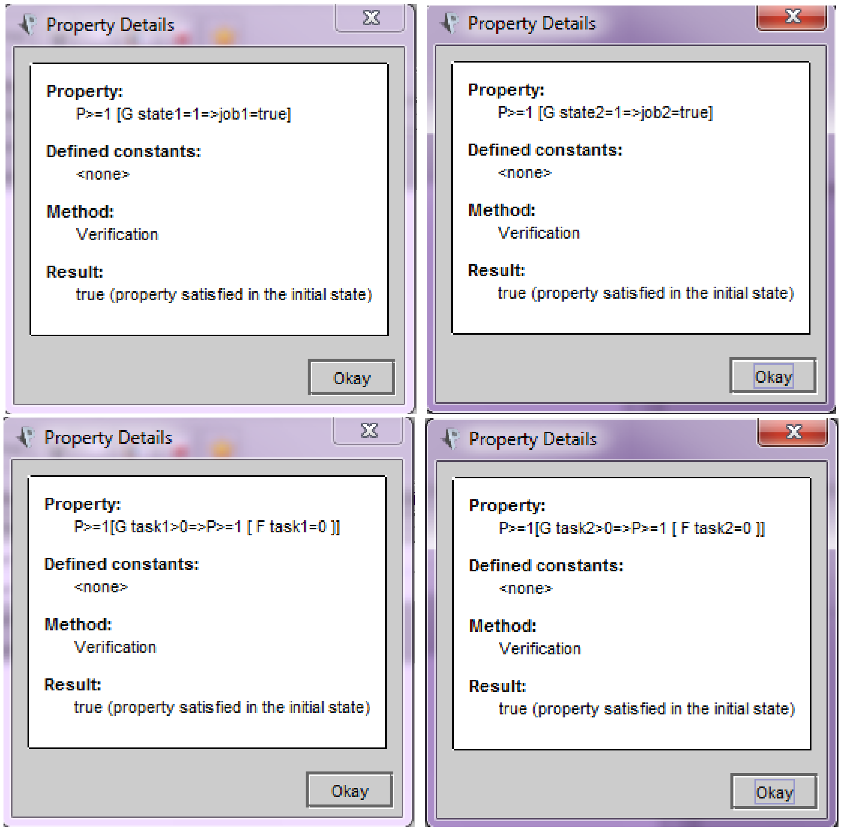
These commands are used to cycle between the jobs. If job_1 has used up all its time in the scheduler but it is now job_1's turn in the queue job_1time should not be true because of the guard in the serve commands which we will look at next. It should then be modified to false so it can be served by the scheduler. The same goes for job_2.
The serve commands have been modified in order to be able to cycle the jobs in a round-robin fashion.

$$
\begin{gathered}
\text{[serve1] job1=true \& job2=false} \Rightarrow \text{true;}\\
\text{[serve2] job1=false \& job2=true} \Rightarrow \text{true;}\\
\text{[serve1] job1=true \& job2=true \& turn=1 \& job1time=false} \Rightarrow\\
\text{(job1time}'\text{=true) \& (turn}'\text{=2);}\\
\text{[serve2] job1=true \& job2=true \& turn=2 \& job2time=false} \Rightarrow\\
\text{(job2time}'\text{=true) \& (turn}'\text{=1);}
\end{gathered}
$$

It serves the jobs if there is no other job in the queue which is still intact from the model of the SRT scheduler. That should not be modified. However it should only serve a job if it is that job's turn to be served and the job has not used up its time being served. The guard specifies that that the turn variable has to be equal to the client's number ($client_1$, $client_2$) and its jobtime has to be false.
We have also made some additions to the *create* and *finish* commands. The create command now has two different kinds for each client. One of the commands has the guard job_X=false and turn=0. This command is only used if there has not been any jobs scheduled in the round-robin scheduler before the creation of the new job. When turn=0 there has not been scheduled any jobs yet because turn is initialized as 0 and then turn is only altered between 1 and 2. The other create command is used when there has been created jobs previously which means that the turn variable has been used and therefore turn > 0.

$$
\begin{gathered}
\text{[create1] job1=false \& turn>0} \Rightarrow \text{(job1}'\text{=true);}\\
\text{[create1] job1=false \& turn=0} \Rightarrow \text{(job1}'\text{=true) \& (turn}'\text{=1);}\\
\text{[create2] job2=false \& turn>0} \Rightarrow \text{(job2}'\text{=true);}\\
\text{[create2] job2=false \& turn=0} \Rightarrow \text{(job2}'\text{=true) \& (turn}'\text{=2);}\\
\dots\\
\text{[finish1] job1=true} \Rightarrow \text{(job1}'\text{=false) \& (job1time}'\text{=false);}\\
\text{[finish2] job2=true} \Rightarrow \text{(job2}'\text{=false) \& (job2time}'\text{=false);}
\end{gathered}
$$

The finish commands has been altered to also set the variable $job_X time$ to false. Since the job is now finished, its time in the queue should not be specified as used up. It should therefore be set as false so it does not affect the next job created by the same client.
This model of a round-robin scheduler has 277 reachable states.
We have specified some properties in CTL which we previously have verified for the FCFS and SRT schedulers. We will verify these properties in PRISM.
$client_1$: $AG(state_1 = 1 \Rightarrow job_1 = true)$
$client_2$: $AG(state_2 = 1 \Rightarrow job_2 = true)$
$client_1$: $AG(task_1 > 0 \Rightarrow AF task_1 = 0)$
$client_2$: $AG(task_2 > 0 \Rightarrow AF task_2 = 0)$



Property:
P>=1 [G state1=1=>job1=true]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1 [G state2=1=>job2=true]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1[G task1>0=>P>=1 [F task1=0]]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1[G task2>0=>P>=1 [F task2=0]]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Figure B1.1: The properties verified by PRISM.

Priority First Come First Served Scheduler

In this part we will model a FCFS scheduler that is able to handle tasks with two different priority levels. A FCFS scheduler serves a job until it is finished however this variant can stop a job if a new job is created with higher priority. In order to accommodate jobs having a priority the *Client* module have to be modified to have an additional variable called *priority* which can be either 1 or 2. When the priority is 2 the job has a higher priority than a job with priority 1. The create commands have been altered to 10 new ones so the client is able to create jobs with both priority 1 and priority 2.

```
priority1 : [1..2] init 1; // Priority of the job
[create1] state1=0 ⇒ (state1′=1) & (task1′=1) & (priority1′=1);
...
[create1] state1=0 ⇒ (state1′=1) & (task1′=1) & (priority1′=2);
...
```

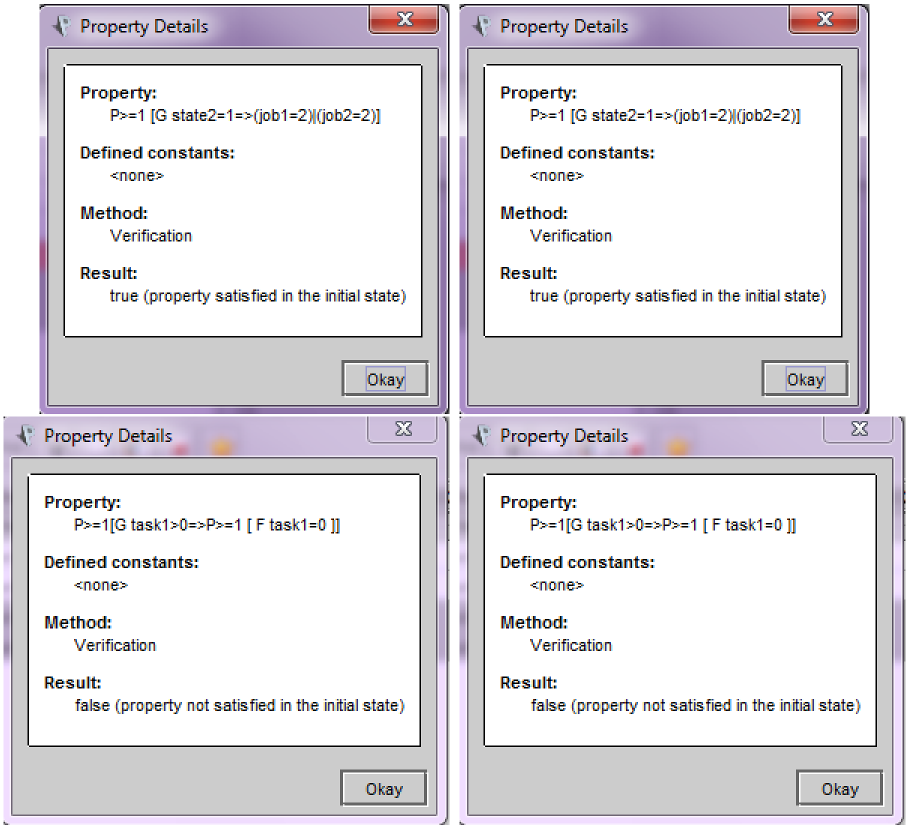
The *Scheduler* module is modified so that when a new job is created and it has a higher priority than a job in the queue it moves ahead of that job. This is done in the module by creating some new create commands in the Scheduler module.

```
[create1] job2=0 ⇒ (job2′=1);
[create2] job2=0 ⇒ (job2′=2);
[create1] job2=0 & priority1=2 & priority2=1 ⇒ (job2′=job1) & (job1′=1);
[create2] job2=0 & priority2=2 & priority1=1 ⇒ (job2′=job1) & (job1′=2);
```

When a new job is created by any of the clients and it has a priority of 2 and the existing job in the queue has a priority of 1 then the new job moves ahead in the queue and the existing job moves back. If the priorities of the jobs are the same no changes should be made and likewise when the existing job in the queue has a priority of 2 and the newly added job has a priority of 1 then no changes should be made. This however involves pre-empting. If a job is running and a job is created with a higher priority it will interrupt the running job and the new job will jump ahead in the queue. This can result in starvation of a job meaning that a job will never be served which will also mean that the client who created that job will never be able to create a new job. This model of FCFS with priority has 360 reachable states.

We have specified some properties in CTL which we previously have verified for the FCFS and SRT schedulers. We will verify these properties in PRISM.
$client_1$: $AG(state_1 = 1 \Rightarrow \neg(\neg job_1 = 1 \land \neg job_2 = 1))$
$client_2$: $AG(state_2 = 1 \Rightarrow \neg(\neg job_1 = 2 \land \neg job_2 = 2))$
$client_1$: $AG(task_1 > 0 \Rightarrow AF task_1 = 0)$
$client_2$: $AG(task_2 > 0 \Rightarrow AF task_2 = 0)$

The two last properties failed verification because this model could result in starvation of a job as explained earlier. This means that a job's length (which is specified in the variable task) will never reach 0.



Property:
P>=1 [G state2=1=>(job1=2)|(job2=2)]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1 [G state2=1=>(job1=2)|(job2=2)]

Defined constants:
<none>

Method:
Verification

Result:
true (property satisfied in the initial state)

Property:
P>=1[G task1>0=>P>=1 [F task1=0]]

Defined constants:
<none>

Method:
Verification

Result:
false (property not satisfied in the initial state)

Property:
P>=1[G task1>0=>P>=1 [F task1=0]]

Defined constants:
<none>

Method:
Verification

Result:
false (property not satisfied in the initial state)

Figure B1.2: The properties verified by PRISM.

B2) Theoretical Problems

Bisimulation relations

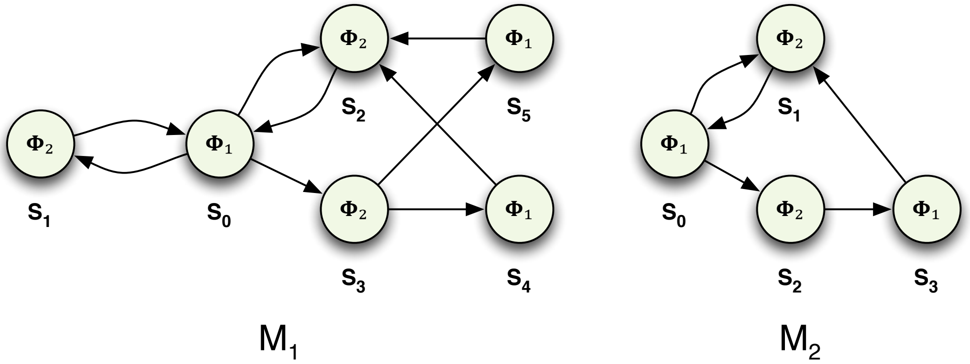


Figure 2: Two transition systems, M_1 and M_2

The two transition systems in Figure 2: $M_1 = (S_1, \rightarrow_1, \{s_0\}, AP, L_1)$ and $M_2 = (S_2, \rightarrow_2, \{s_0\}, AP, L_2)$ where $AP = \{\phi_1, \phi_2\}$.
We have to find the coarsest bisimulation relation between the states of the two structures: $R_{12} \subseteq S_1 \times S_2$.
The first step to find this, is finding the states that have the same labels. This gives us:

$$R_{12} = \left\{ \begin{array}{c} (s_0, s_0), (s_0, s_3), (s_1, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_2), \\ (s_3, s_1), (s_3, s_2), (s_4, s_0), (s_4, s_3), (s_5, s_0), (s_5, s_3) \end{array} \right\}$$

The next step in finding the coarsest bisimulation relation between two transition systems is to remove the entries that does not satisfy the mutual simulation conditions. Doing so will give us the following:

$$R_{12} = \left\{ \begin{array}{c} (s_0, s_0), (s_0, s_3), (s_1, s_1), (s_1, s_2), (s_2, s_1), (s_2, s_2), \\ (s_3, s_1), (s_3, s_2), (s_4, s_0), (s_4, s_3), (s_5, s_0), (s_5, s_3) \end{array} \right\}$$

R_{12} is a bisimulation relation of M_1 and M_2 ($M_1 \sim M_2$) because the initial states are bisimilar: $s_0 R s_0$. It is also the coarsest bisimulation relation because we started by finding all the states that have the same label and put them in the relation. Then we proceeded to remove the entries that does not satisfy

the mutual simulation conditions which led us to removing none. We did not remove any entries because every state in the two transition systems that have the same label have an incoming edge to a state with another label and an outgoing edge to a state with another label.
We have to find the coarsest bisimulation relation between the states of M_1: $R_{11} \subseteq S_1 \times S_1$.
We do the same thing as we did last time - we find the states that have the same labels. This gives us:

$$R_{11} = \left\{ \begin{array}{c} (s_0, s_0), (s_0, s_4), (s_0, s_5), (s_1, s_1), (s_1, s_2), (s_1, s_3), \\ (s_2, s_1), (s_2, s_2), (s_2, s_3), (s_3, s_1), (s_3, s_2), (s_3, s_3), \\ (s_4, s_0), (s_4, s_4), (s_4, s_5), (s_5, s_0), (s_5, s_4), (s_5, s_5) \end{array} \right\}$$

The next step is to remove the entries that does not satisfy the mutual simulation conditions which will give us the following:

$$R_{11} = \left\{ \begin{array}{c} (s_0, s_0), (s_0, s_4), (s_0, s_5), (s_1, s_1), (s_1, s_2), (s_1, s_3), \\ (s_2, s_1), (s_2, s_2), (s_2, s_3), (s_3, s_1), (s_3, s_2), (s_3, s_3), \\ (s_4, s_0), (s_4, s_4), (s_4, s_5), (s_5, s_0), (s_5, s_4), (s_5, s_5) \end{array} \right\}$$

R_{11} is a bisimulation relation of M_1 and M_1 ($M_1 \sim M_1$) because the initial states are bisimiliar: $s_0 R s_0$. It is maximal because all states with the same label is in the relation. No entries have been removed due to every state having an incoming edge from and an outgoing edge to a state of the other label.
We can use this to construct the bisimulation quotient $M_1/\sim$ of M_1. Then we have to find the bisimilar states and since no entries was removed due to satisfying the mutual simulation conditions all states with the same label are bisimilar.
$s_0 \sim_T s_4$, $s_0 \sim_T s_5$, $s_4 \sim_T s_5$, $s_1 \sim_T s_2$, $s_1 \sim_T s_3$ and $s_2 \sim_T s_3$.
This gives us the bisimulation quotient $M_1/\sim$ of M_1. This is shown in the Figure below.

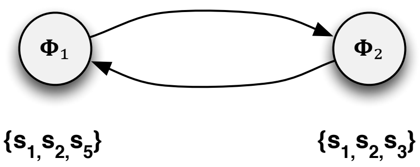


Figure B2.1e: Bisimulation quotient of $M_1/\sim$ of M_1.

We want to find out if this is bisimilar to M_2. We take the same approach as before by putting all the states with the same labels in the relation.

$$R = \left\{ \begin{array}{c} (\{s_0, s_4, s_5\}, s_0), (\{s_0, s_4, s_5\}, s_3), \\ (\{s_1, s_2, s_3\}, s_1), (\{s_1, s_2, s_3\}, s_2) \end{array} \right\}$$

We want to remove the entries in this relation which do not satisfy the mutual simulation conditions. This gives us the following relation:

$$R = \left\{ \begin{array}{c} (\{s_0, s_4, s_5\}, s_0), (\{s_0, s_4, s_5\}, s_3), \\ (\{s_1, s_2, s_3\}, s_1), (\{s_1, s_2, s_3\}, s_2) \end{array} \right\}$$

Again no entries was removed because every state with label ϕ_1 has an outgoing edge to and an incoming edge from a state that has label ϕ_2 and vice versa. This is therefore the coarsest bisimulation relation since all the states with the same label are in the relation and none was removed for not satisfying the mutual simulation conditions.

Satisfiability

Let's consider the following CTL formula

$$\Phi = \neg(EX\Phi_1) \wedge (AF\Phi_2)$$

where Φ_1 and Φ_2 are atomic propositions. We can convert Φ into $\exists$-normal form, which is a normalization of the used operators. The allowed operators are: $\neg, \wedge, \exists\bigcirc, \exists\cup, \exists\Box$. We will call the $\exists$NF version of Φ for Φ'.

$$\Phi' = \neg(\exists \bigcirc \Phi_1) \wedge (\neg\exists\Box(\neg\Phi_2))$$

Further we can draw an abstract syntax tree for Φ' to better illustrate it.

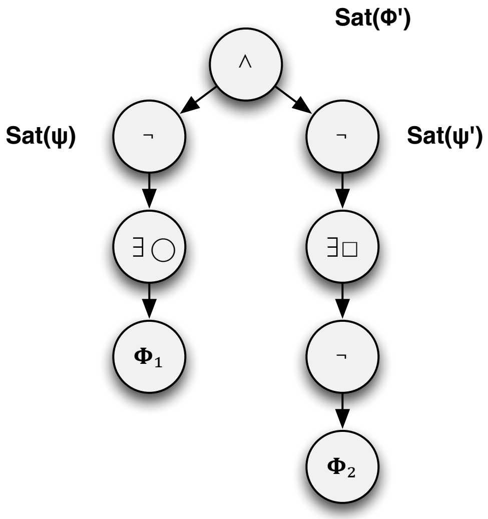


Figure 3: An abstract syntax tree of Φ'.

A formula can be split into sub-terms, often denoted as Ψ. In Figure 3, the sub-terms of Φ' is shown.

$$\Psi = \neg(\exists \bigcirc \Phi_1)$$
$$\Psi' = \neg(\exists \Box(\neg \Phi_2))$$

We wish to define how Φ is satisfied. We can do this by using the $Sat()$ keyword. $Sat(\Phi)$ means the set of states that satisfy Φ. For each of our sub-terms (Ψ, Ψ') and for Φ', we wish to list the set of states which satisfy them.

$$Sat(\Psi) = Sat\left(\neg(\exists \bigcirc \Phi_1)\right) = S \setminus \{s \in S : Post(s) \bigcap Sat(\Phi_1) \neq \emptyset\}$$

$$Sat(\Psi') = Sat\left(\neg(\exists \Box(\neg \Phi_2))\right) = S \backslash \{T \subseteq Sat(\neg \Phi_2)\ and\ s \in T \Rightarrow Post(s) \bigcap T \neq \emptyset\}$$

$$Sat(\Phi') = Sat\left(\neg(\exists \bigcirc \Phi_1) \wedge (\neg \exists \Box(\neg \Phi_2))\right) = Sat(\Psi) \bigcap Sat(\Psi')$$

We can calculate the set of states satisfying $Sat(\Phi) = Sat(\Phi')$ for M_1 from Figure 2. To start finding out which states are satisfied for $Sat(\Phi)$ we start off with the left side of the $\wedge$ symbol $(\neg(EX\Phi_1))$. We have to find states which does not have an immediate path to a state containing Φ_1. We will call this set for T_{11}.

$$T_{11} = \{s_0, s_4, s_5\}$$

Now for the right part of the $\wedge$ symbol ($AF\Phi_2$). We have to find states which all paths from the state at some point reaches Φ_2. We will call this set for T_{12}.

$$T_{12} = \{s_0, s_1, s_2, s_3, s_4, s_5\}$$

We can now calculate the set of states that match both the left and right side of the $\wedge$ symbol, which is the union of the two sets. We will call this set for T_1.

$$T_1 = T_{11} \cup T_{12} = \{s_0, s_4, s_5\}$$

Now on to finding out which states are satisfied for $Sat(\Phi')$. Same procedure as before. First the set of states that match the left side of the $\wedge$ symbol ($\neg(\exists\bigcirc\Phi_1)$). As before we have to find a state which do not have an immediate path to a state containing Φ_1. We will call this set for T_{21}.

$$T_{21} = \{s_0, s_4, s_5\}$$

The right part of the $\wedge$ symbol ($\neg\exists\Box(\neg\Phi_2)$) is next. We have to find states where no path exists where the paths from the state and all subsequent paths never contains Φ_2. We will call this set T_{22}.

$$T_{22} = \{s_0, s_1, s_2, s_3, s_4, s_5\}$$

We can yet again calculate the set of states that match both the left and right side of the $\wedge$ symbol, which is the union of the two sets. We will call this set for T_2.

$$T_2 = T_{21} \cup T_{22} = \{s_0, s_4, s_5\}$$

At last we can calculate the set of states that satisfy the following statement $Sat(\Phi) = Sat(\Phi')$.

$$Sat(\Phi) \cup Sat(\Phi') = T_1 \cup T_2 = \{s_0, s_4, s_5\}$$

Recursiveness

A recursive subterm of Φ is Ψ' ($\neg(\exists\Box\Phi_2)$). Using the following expansion law:

$$\exists\Box\Phi = \Phi \wedge \exists \bigcirc \exists\Box\Phi$$

We can see that the $\exists\Box\Phi$ part is recursive and thereby a *fixed point.* Fixed points can either be a *least* fixed point or a *greatest* fixed point. As the names

suggests the least fixed point is the smallest set of satisfying states while the greatest fixed point is the largest set of satisfying states.
To compute the smallest and largest set we have to make use of the expansion laws and the fact that we have a recursive expression.

Least fixed point

An expression containing a least fixed point could be $\exists(\Phi_1 \cup \Phi_2)$. The expansion laws dictates that the expression could be expanded to

$$\Phi_2 \vee (\Phi_1 \wedge \exists \bigcirc \exists(\Phi_1 \cup \Phi 2))$$

Hence $\exists(\Phi_1 \cup \Phi_2)$ is the fixed point.
This can be expressed as

$$Sat\,(\exists(\Phi_1 \cup \Phi_2)) = Sat(\Phi_2) \cup \{s \in Sat(\Phi_1) : Post(s) \cap T \neq \emptyset\} \subseteq T$$

Where T is the smallest set of satisfying states.

Greatest fixed point

An expression containing a greatest fixed point could be $\exists\Box\Phi$. The expansion laws dictates that the expression could be expanded to

$$\Phi \wedge \exists \bigcirc \exists\Box\Phi$$

Hence $\exists\Box\Phi$ is the fixed point.
This can be expressed as

$$Sat(\exists\Box\Phi) = \{s \in Sat(\Phi) : Post(s) \cap T \neq \emptyset) \supseteq T$$

Where T is the largest set of satisfying states.

For our expression the fixed point used is a greatest fixed point as shown above.

Part C: Advanced Problems

Practical Problems

In the practical part of part B we modeled a round-robin scheduler. In this part we have modified the scheduler to include priorities (the model can be found in the Weighted-RR.nm file). When a scheduler has to include priorities it is important to decide how these priorities should affect the scheduling of jobs. In our specification the scheduler chooses the job with the highest priority and it serves it till its finished. There is however one exception to this. If a job has waited 5 turns this job will be served even though it may have a priority lower than the job that has been put on hold. This is a very simple but not a very convenient way to avoid starvation of a job because this could keep a job with a higher priority in the queue while serving a less important job. If the priority is equal the scheduler cycles between the jobs in a regular round-robin fashion. We use the notion that a job can either have a priority of 1 or 2 where 2 is the highest priority. This is specified by the client when the client creates a job. The *Scheduler* module has been changed to accommodate priorities. It introduces two new variables $wait_1$ and $wait_2$ which specifies how long a job has waited and it is an integer between 0 and 5 with an initial value of 0. $wait_1$ specifies how long job_1 has waited and $wait_2$ specifies how long job_2 has waited.
The finish commands has been altered to accommodate the wait variables because if a job has finished it should not affect the clients next job that the previous job waited 4 turns for its turn. So the variable is reset to 0.

```
[finish1] job1=true ⇒ (job1′=false) & (job1time′=false) & (wait1′=0);
[finish2] job2=true ⇒ (job2′=false) & (job2time′=false) & (wait2′=0);
```

The serve commands have also been altered to comply with jobs having priorities and waiting time. However one thing that should not be changed is the fact that if only one client has a job in the scheduler that job should of course be served. This can be seen in the first two lines below.

1. [serve1] job1=true & job2=false ⇒ true;
2. [serve2] job1=false & job2=true ⇒ true;
3. [serve1] job1=true & job2=true & turn=1 & job1time=false & priority1=priority2 & wait2<5 ⇒ (job1time′=true) & (turn′=2);
4. [serve2] job1=true & job2=true & turn=2 & job2time=false & priority2=priority1 & wait1<5 ⇒ (job2time′=true) & (turn′=1);
5. [serve1] job1=true & job2=true & wait1=5 ⇒ (turn′=2);
6. [serve2] job1=true & job2=true & wait2=5 ⇒ (turn′=1);
7. [serve1] job1=true & job2=true & priority1>priority2 & wait2<5 ⇒ (turn′=2) & (wait2′=wait2+1);
8. [serve2] job1=true & job2=true & priority2>priority1 & wait1<5 ⇒ (turn′=1) & (wait1′=wait1+1);

The first two conditions in every guard above are very simple - the two job's have be to true meaning both clients have a job in the scheduler. Line 3 and 4 specifies the regular round-robin. If the priorities are equal and a job has not waited 5 turns for its turn then they should be cycled in a regular round-robin fashion. Line 5 and 6 specifies that if one job has waited 5 turns then that job should be served until it is finished and then the wait variable is reset to 0. However the update in these lines are interesting it changes the turn variable so it is the other job's turn to be served. This is done so that when job_1 is finished and that $client_1$ enters another job with the same priority as job_2 it will be job_2s turn to be served. This is a minor detail done in order to faster clear out jobs in the scheduler. Line 7 and 8 serves a job if its priority is higher than the other job's priority and it the other job has not waited 5 turns. The update again changes the turn variable to the other job for the same reason as line 5 and 6. It also increments the wait variable for the other job so the other job does not have to wait too long to be served. The Weighted-RR model has 3655 reachable states.
We have specified some properties in CTL which we will use PRISM to verify for our model. The properties specified below can also be found specified in PRISM notation in the Weighted-RR.pctl file.

No.	CTL	Verified
1.	$AG(state_1 = 1 \Rightarrow job_1 = true)$	✓
2.	$AG(state_2 = 1 \Rightarrow job_2 = true)$	✓
3.	$AG(task_1 > 0 \Rightarrow AFtask_1 = 0)$	✓
4.	$AG(task_2 > 0 \Rightarrow AFtask_2 = 0)$	✓
5.	$A(task_1 > 0 \wedge wait_1 = 5Utask1 = 0 \wedge job1 = false$	✓
6.	$AG(wait_1 = 5 \wedge task_1 = 1 \Rightarrow (AXtask_1 = 0))$	÷
7.	$AG(wait_1 = 5 \wedge task_1 = 1 \Rightarrow (EXtask_1 = 0))$	✓
8.	$AG(wait_1 = 5 \wedge task_2 = 1 \Rightarrow (EXtask_2 = 0))$	÷
9.	$AG(wait_2 = 5 \wedge task_1 = 1 \Rightarrow (EXtask_1 = 0))$	÷
10.	$AG(priority_2 > priority_1 \wedge task_2 = 1 \wedge wait_1 < 5 \Rightarrow (EXtask_2 = 0))$	✓
11.	$AG(priority_2 > priority_1 \wedge task_2 = 1 \wedge wait_1 < 5 \Rightarrow (EXtask_1 = 0))$	÷

No. 1 is a property that specify that if $state_1$ is 1 then job_1 should be true meaning that when $client_1$ has created a job it should be in the scheduler. The same goes for $client_2$ which is specified by no. 2 in the list.
No. 3 and 4. specify that it is globally the case whenever $task_1$ and $task_2$ is larger than 0 then it should eventually hold that $task_1$ and $task_2$ is 0 which means that the job has to eventually finish. This property could not be verified if the scheduler did not avoid starvation because then a job would never finish and it would just be in the scheduler forever. We prevent this by having a maximum waiting period of 5 turns as explained earlier.
No. 5 specifies that when job_1 has waited 5 turns and the job's length is larger than 0 it should hold until $task_1 = 0$ and $job_1 =$ false. This is a property to verify that when a job has waited 5 turns it should be served till it is finished.
No. 6 specifies that it is globally the case that if a job has waited 5 turns and the length of its job is only 1 then in all the next states the length of the job should be 0, i.e. the job is finished. This property has not been verified and it should not be because in the scheduler there are two actions that can happen independently of all the other modules and this could create a new state where some of the variables have changed.
No. 7 specifies that it is globally the case that if a job has waited 5 turns and the length of its job is only 1 then in one of the next states the length of job should be 0. This property has been verified. This verifies that in one of the next states it is the job's turn to be served when it has waited 5 turns, however this is not enough to verify that the job will always be served ahead of the other job because of its waiting period.

This brings us to no. 8 and 9. No. 8 and 9 specify that if a job has waited 5 turns and the length of the other job is 0 then in one of the next states the length of the other job should be 0. This is not verified by PRISM which means that it is modeled correctly. It means that if a job has waited 5 turns the other job will never be served ahead of that job.
No. 10 specifies that if a job has a higher priority than the other job and its length is 1 and the other job has not waited 5 turns then in one of the next states the length of the job should be 0. This has been verified by PRISM. It verifies that it in one of the next states it is the job's turn to be served when it has a higher priority than the other job. Again however this is not enough to verify that the job will always be served ahead of the other job if it has a higher priority and the other job has not waited 5 turns.
This brings us to no. 11. No. 11 specifies that if a job has a higher priority than the other job and its length is 1 and the other job has not waited 5 turns then in one of the next states the length of the other job should be 0. This is not verified by PRISM which means that it is modeled correctly. It means that if a job has a higher priority and the other job has not waited 5 turns the other job will never be served ahead of that job.