# 02246 Mandatory Assignment
# L03 - Temporal Logics*

To be submitted on DTU Learn - see deadline on DTU Learn

You are encouraged to work in groups, but you must clearly identify the contributions of each group member, and you will be jointly responsible for the finished report. Register your group on DTU Learn before submitting as group submission.

Answers to all parts should be typed up using LaTeX and submitted electronically as a PDF report using the provided template. Drawings and formulae may be handwritten and scanned. More detailed instructions as to the style of answer we expect for each part are included below.

Group contribution: We have worked together on every aspect of this assignment. Problems have been solved and discussed together. We have both had a part in the formulation and answer of every problem, and it is therefore difficult to put names on any specific section.

*Thanks to Michael Smith (original author), and Lijun Zhang, Kebin Zeng and Flemming Nielson and Alberto Lluch Lafuente (contributors)

# L03 - Temporal Logics

## L03P: Practical Problems

**L03P.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 $job_2 = 1$.

a) Express this as two CTL properties — one for each client.

Assuming that we do not have to use the minimal CTL syntax we can write

$$\forall\Box(state_1 = 1 \rightarrow ((job_1 = 1 \wedge \neg job_2 = 1) \vee (\neg job_1 = 1 \wedge job_2 = 1))),$$

and

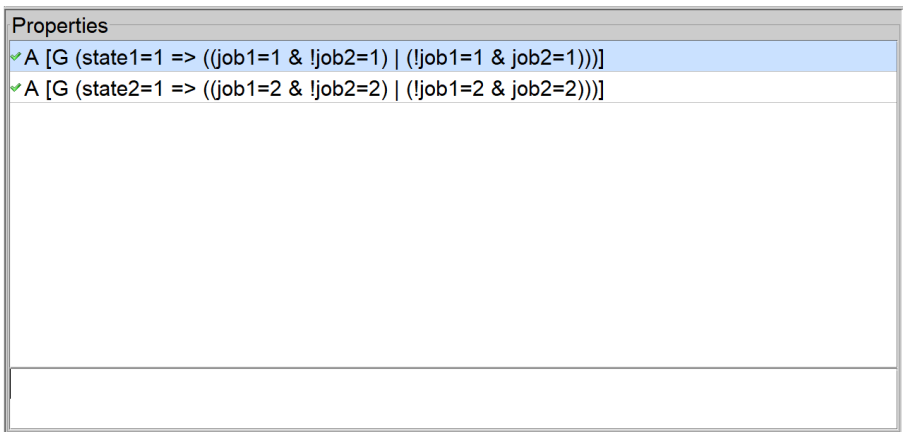
$$\forall\Box(state_2 = 1 \rightarrow ((job_1 = 2 \wedge \neg job_2 = 2) \vee (\neg job_1 = 2 \wedge job_2 = 2))).$$

b) Use PRISM to verify whether these properties hold in the FCFS scheduler model. Provide a screenshot showing that this is the case.

We write the following properties for PRISM

```
A [G(state1=1 => ((job1=1 & !job2=1) | (!job1=1 & job2=1)))]
A [G(state2=1 => ((job1=2 & !job2=2) | (!job1=2 & job2=2)))]
```

and is given the output when verifying




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

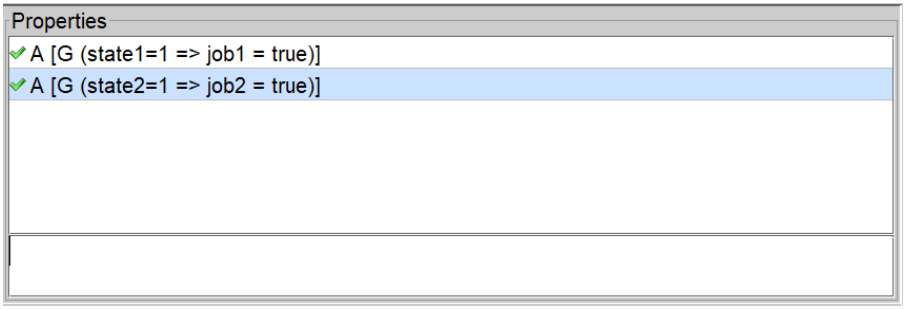
We express the requirement that whenever $state_1 = 1$, i.e. $client_1$ has an active job, then $job_1 = true$ must be true, as the scheduler then recognizes that $client_1$ has a waiting job. This should of course hold for all states in every execution path. Thus we have

$$\forall\ \Box(state_1 = 1\ \rightarrow job_1 = true)$$

and

$$\forall\ \Box(state_2 = 1\ \rightarrow job_2 = true).$$

d) Verify whether they hold in the model. Provide a screenshot showing the result.
Using PRISM we find that the above properties hold for the SRT scheduler.




**L03P.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.

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

First of all, we create `client3` in the exact same way as `client2` was created with the exception of naming the variables with '3' instead of '2'.

```
module client3 = client1 [state1=state3,
                          task1=task3,
                          create1=create3,
                          serve1=serve3,
                          finish1=finish3 ]
endmodule
```

We add the possibility of `job1` and `job2` having the value 3, as `client3` then possibly can be placed in the queue.

```
    job1 : [0..3] init 0; // First job in the queue
    job2 : [0..3] init 0; // Second job in the queue
```

We then add `create3`, `serve3`, `finish3` actions similar to the actions of 1 and 2.

```
    [create3] job2=0 -> (job2'=3);
    ⋮
    [serve3] job1=3 -> true;
    ⋮
    [finish3] job1=3 -> (job1'=0);
```

Finally, we append "`|| client3`" to the system definition.

```
system
    scheduler || client1 || client2 || client3
endsystem
```

This adds `client3`, in the same way that `client2` was added, i.e. client3 maintains exactly the same behavior as `client1` and `client2` respectively. Also we do not extend the queue, as we do not add a `job3` variable for a third spot in the queue. Thus this satisfies the given description.

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

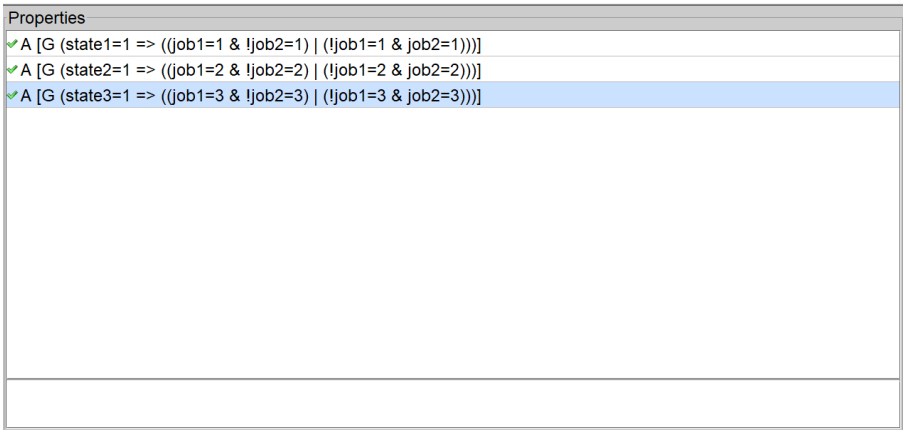
Using PRISM, we find that there are 214 reachable states in the new system.

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

In that case, the guard of the `createX` command is not satisfied, and therefore nothing will happen. The resulting state of such action is part of the unreachable states.

d) Do the properties you have previously verified still hold in the model? If not, why not? Provide a screenshot showing the results.

We show with PRISM that the properties still hold. Note that we included a property for the added third client.




This does indeed make sense, as we specifically specify only to create a new job if the last place in the queue is empty in the lines

```
[create1] job2=0 -> (job2'=1);
[create2] job2=0 -> (job2'=2);
[create3] job2=0 -> (job2'=3);
```

**L03P.3** Now additionally modify the *Scheduler* module so that the queue is of length three.

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

First of all, we add a variable `job3` representing the third job in queue. Then we update the actions by the `createX` call, so when placing a new job at the end of the queue instead of updating the variable `job2`, we update `job3`.

For the part that shifts the queue if there is an empty slot, we add a command that also moves a job from the third to the second slot in the queue if there is space.

```
[] job2=0 & job3>0 -> (job2'=job3) & (job3'=0);
[] job1=0 & job2>0 -> (job1'=job2) & (job2'=0);
```

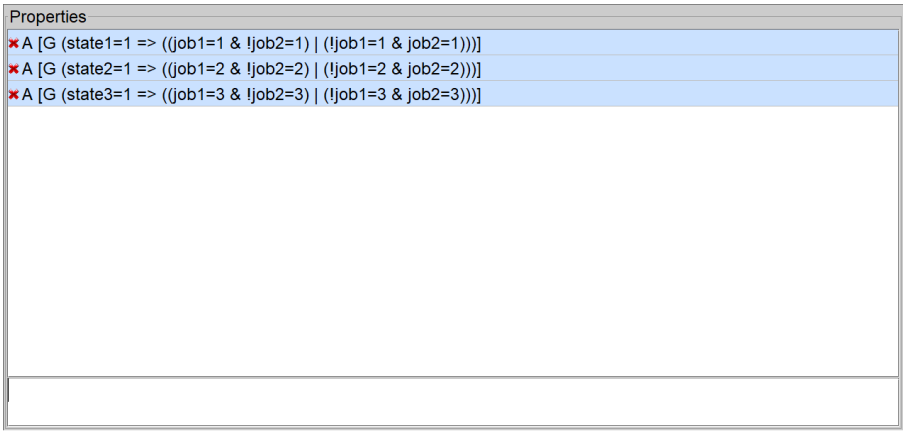
It behaves in the exact same way as the two-spot queue. Only jobs enter spot 3 and are shifted through the queue until they reach the first spot and can be executed.

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

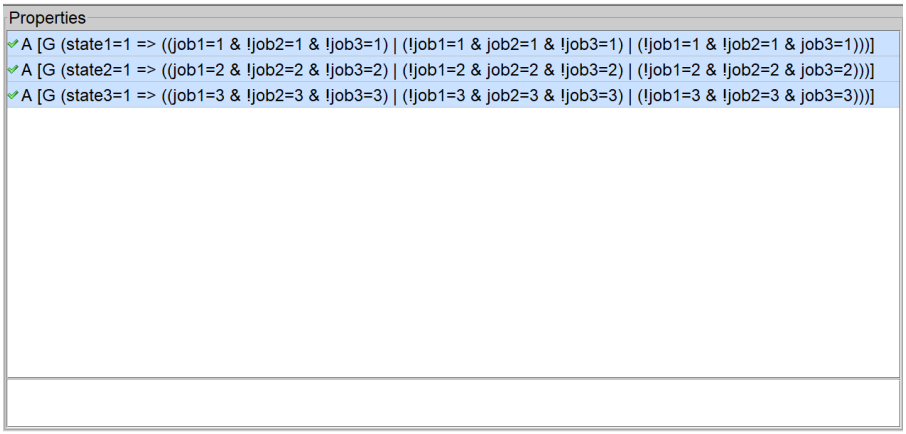
Using PRISM we find that there exist 1458 reachable states.

c) Do the properties now hold in the model? If not, why not? Provide a screenshot showing the results.

With the addition of a new spot in the queue, the properties do not hold. This is due to the fact that our property more precisely states that each job needs to be either number one or two in queue, - which obviously does not hold when the job is number three in queue.




If however, we extend the properties to include the third sport in the queue we find that the properties hold.



## L03T: Theoretical Problems

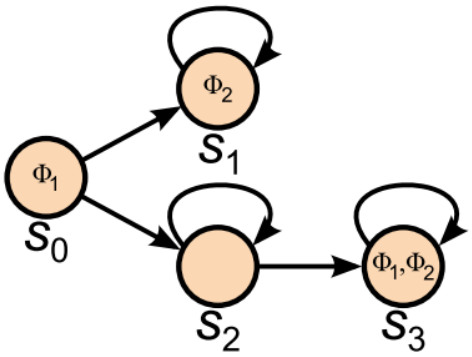



Figure 1: A transition system

**L03T.1** 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 $s_0$. Determine whether the following properties hold in state $s_0$. You can encode the transition system in Figure 1 as a PRISM module, and you can use PRISM to check if your answers are correct but you have to explain why they hold or do not hold using the formal semantics of CTL.

a) $AF\ \Phi_2$.

False. We can construct a counter-example giving the following lasso: we can avoid $\Phi_2$ forever by going to state $S_2$ from $S_0$ and from there use the self-loop infinitely, that is, stay in $S_2$ forever and thus not end in $\Phi_2$.

b) $AX\ \Phi_2$.

False. If we go to $S_2$ we will not be at $\Phi_2$ in one step.

c) $EF\ \Phi_1$.

True. Clearly, we will be at a state satisfying $\Phi_1$ eventually, as we are starting in $S_0$ which already satisfies $\Phi_1$.

d) $A[\Phi_1\ U\ \Phi_2]$.

False. The statement means that we for all paths starting in $S_0$ always reach a state where $\Phi_2$ holds by following a path of states where $\Phi_1$ holds. Thus a counter example is the path $S_0S_2S_3$ where neither $\Phi_1$ nor $\Phi_2$ holds in the state $S_2$.

**L03T.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.

a) $EX\ EF\ \Phi$ and $EF\ EX\ \Phi$.

They are equivalent. The reasoning is as follows - for any transition system exactly one of the following cases trivially hold:

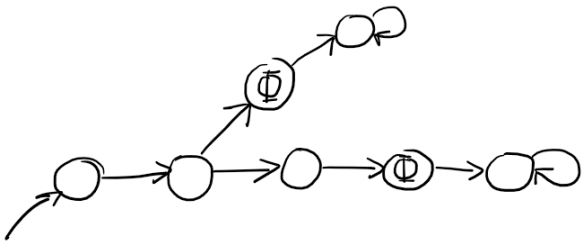
(a) Either we can reach a state where $\Phi$ is satisfied in one or more steps, or

(b) we cannot.

In case (a) the formulae respectively represent "one step followed by zero or more steps before reaching $\Phi$" or "zero or more steps followed by one step before reaching $\Phi$" which in both cases means "one or more steps before reaching $\Phi$". Hence both CTL formulae are satisfied in case (a). In case (b) none of the CTL formulae are satisfied. Thus we conclude they are equivalent.

b) $AX\ AF\ \Phi$ and $AF\ AX\ \Phi$.

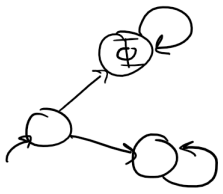
They are not equivalent by the counter example below, where $AX\ AF\ \Phi$ is satisfied, but $AF\ AX\ \Phi$ is not.




We note however that $AF\ AX\ \Phi$ implies $AX\ AF\ \Phi$. We argue this by the fact that $AX\ AF\ \Phi$ means that we for all paths find $\Phi$ in one or more steps. $AF\ AX$ means that all paths eventually lead to a state from where the next step certainly is $\Phi$. This implies that (after the first step) all paths lead to $\Phi$, namely $AX\ AF\ \Phi$.

c) $AG\ EF\ \Phi$ and $EF\ AG\ \Phi$.

We first argue that $EF\ AG\ \Phi$ does not imply $AG\ EF\ \Phi$ by the following counter-example where $EF\ AG\ \Phi$ is satisfied but $AG\ EF\ \Phi$ is not.




Now, $AG\ EF\ \Phi$ implies that $EF\ \Phi$ must hold in all reachable states, in particular in the last[1] state of all paths. This in turn implies that $EF\ AG\ \Phi$.

d) $AG\ (\Phi_1 \wedge \Phi_2)$ and $(AG\ \Phi_1) \wedge (AG\ \Phi_2)$.

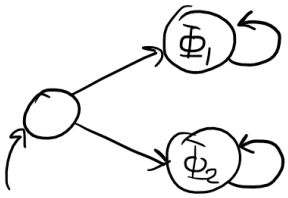
These are equivalent. $AG\ (\Phi_1 \wedge \Phi_2)$ is true only if all states on every path satisfy both $\Phi_1$ and $\Phi_2$. This implies that both $AG\ \Phi_1$ and $AG\ \Phi_2$ are true.

Now if $(AG\ \Phi_1) \wedge (AG\ \Phi_2)$ clearly all states on every path must satisfy both $\Phi_1$ and $\Phi_2$ and thus $AG\ (\Phi_1 \wedge \Phi_2)$ is true.

e) $EF\ (\Phi_1 \wedge \Phi_2)$ and $(EF\ \Phi_1) \wedge (EF\ \Phi_2)$.

Firstly, we will show that $(EF\ \Phi_1) \wedge (EF\ \Phi_2)$ does not imply $EF\ (\Phi_1 \wedge \Phi_2)$ by giving a counter example.

[1] By *last*, we mean a state from where there is only a self-loop going out.




Now we will argue that $EF\ (\Phi_1 \wedge \Phi_2)$ implies $(EF\ \Phi_1) \wedge (EF\ \Phi_2)$. If $EF\ (\Phi_1 \wedge \Phi_2)$ then it follows that $(\Phi_1 \wedge \Phi_2)$ is satisfied in some state $s$ on a path. Then clearly $\Phi_1$ is satisfied in $s$, and $\Phi_2$ is satisfied in $s$. Hence $(EF\ \Phi_1) \wedge (EF\ \Phi_2)$ is true and the implication follows.