Group №3.

Configurations: <https://drive.google.com/file/d/1RrNXg5MweM_ssA0Zo1oQHFYiUmIWkDfX/view?usp=sharing>

Assignment: <https://drive.google.com/file/d/1WbpQbHhyce_NxLsVLWEu40z3HUmOKlv6/view?usp=sharing>

Model (should be a Github link): <https://pastebin.com/qt0Li63a>

Authors: Denis Maximenko, Alfred Mohenu, Maxim Shmakov

# Introduction

Overall, we were asked to:

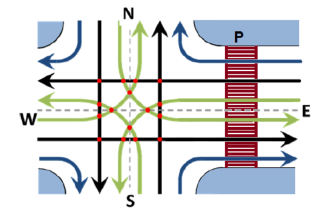
1. Model the traffic light controller.
2. Verify the model for safety, liveness, and fairness properties
3. Prepare a short report including your specific configuration, necessary comments, and verification results.

For that, we were to use the Promela language and Spin model checker. This report contains a description of our approach and Promela model, as well as verification results

# Modeling the intersection

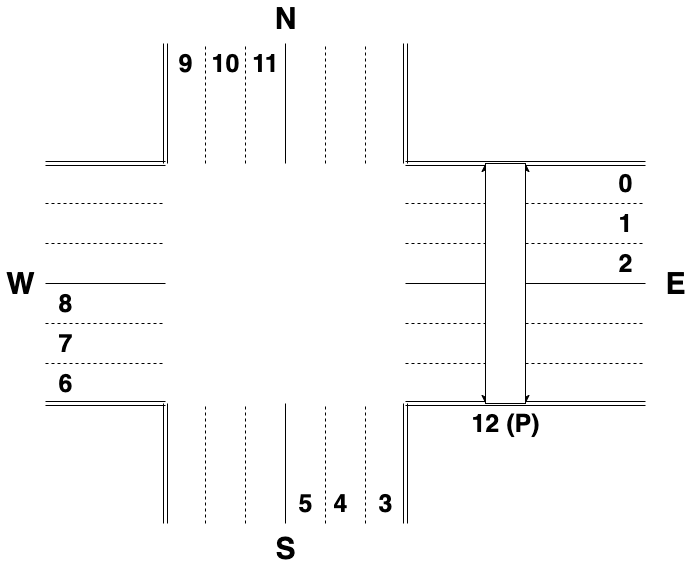
## Traffic light rules

In our assignment, we needed to model a sub-configuration of crossroads, but we decided to take it one step further. Let's consider the full configuration of the crossroads:

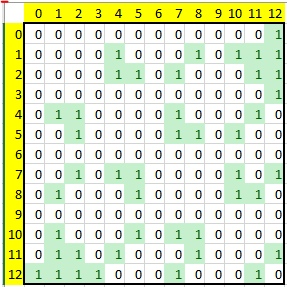


For convenience, we developed a common numeration system for each of those directions, which is as follows:

* We start from E->N at 0
* We then go clockwise, assigning each next lane a higher number (+1) That is 0 through 11 for car lanes
* Finally, we assign number 12 to the pedestrian crossing, which can also be considered as a traffic lane



We can make a table of intersections between lanes (red dots on the scheme):



Rules/Conditions for traffic lights can then be represented in the form of bitmasks:

Lane 0: **1** 000 000 000 000, intersects 12

Lane 1: **1** **11**0 **1**00 0**1**0 000, intersects 4, 8, 10, 11, 12

Lane 2: **1** **1**00 0**1**0 **11**0 000, intersects 4, 5, 7, 11, 12

Lane 3: **1** 000 000 000 000, intersects 12

Lane 4: 0 000 000 000 **11**0, intersects 1, 2, 7, 11

Lane 5: 0 0**1**0 **11**0 000 **1**00, intersects 2, 7, 8, 10

Lane 6: 0 000 000 000 000, no intersections

Lane 7: **1** 0**1**0 000 **11**0 **1**00, intersects 2, 4, 5, 10, 12

Lane 8: 0 **11**0 000 **1**00 0**1**0, intersects 1, 5, 10, 11

Lane 9: 0 000 000 000 000, no intersections

Lane 10: 0 000 **11**0 **1**00 0**1**0, intersects 1, 5, 7, 8

Lane 11: **1** 000 **1**00 0**1**0 **11**0, intersects 1, 2, 4, 8, 12

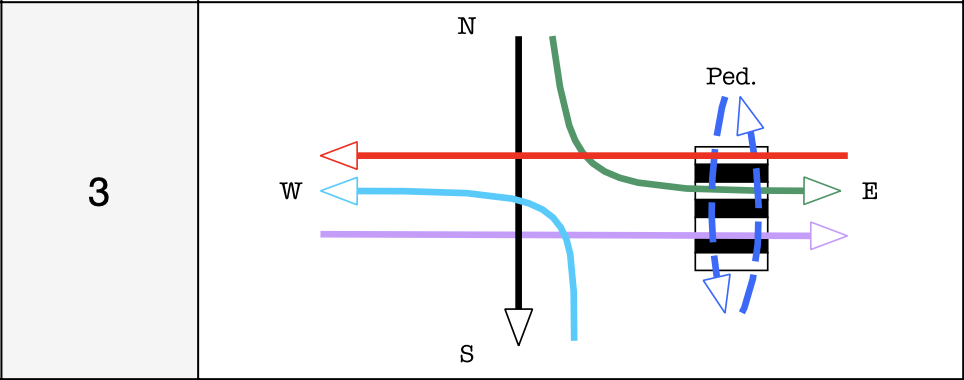
Lane 12: 0 **1**00 0**1**0 00**1** **111**, intersects 0, 1, 2, 3, 7, 11

Each “1” in a bitmask represents an intersection with another lane.

We then store the intersection state in an integer variable, with “1” representing green light, and “0” for red.

With that, we can check our conditions with a single bitwise operation.

Our sub-configuration:



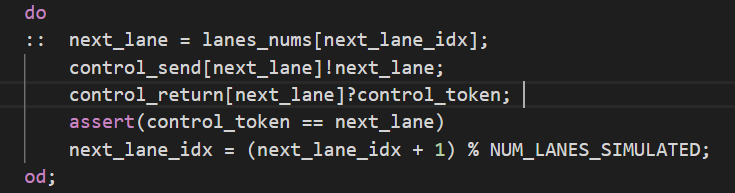
## Promela model

Our lights are controlled by a common process for the whole intersection, which synchronizes traffic lights using semaphores. Only one traffic light can change the intersection state at any point in time.

In our model we have several process types:

* **car\_spawner** is responsible for generating the cars on the lanes;
* **pedestrian\_spawner** is responsible for generating the pedestrians on the lanes;
* **traffic\_light** is a model of a single traffic light;
* **intersection\_controller** sends tokens to control synchronization channels to control the intersection. This controller allows instances of **traffic\_light** proctype to check the state of the intersection and change it if possible;
* **init** process (single instance) is used to bootstrap the crossroads simulation.

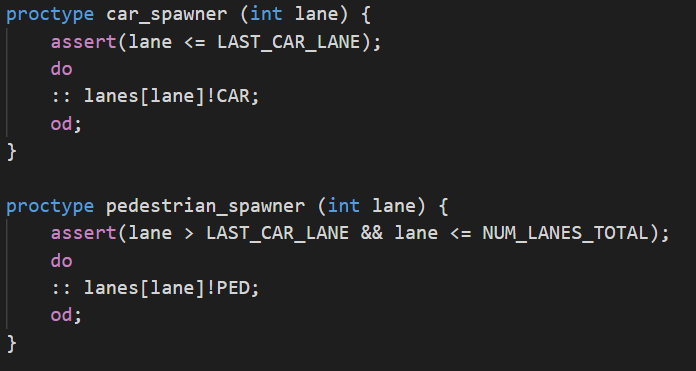
The **intersection\_controller** goes through all **traffic\_light** processes granting them control, to one process at a time:

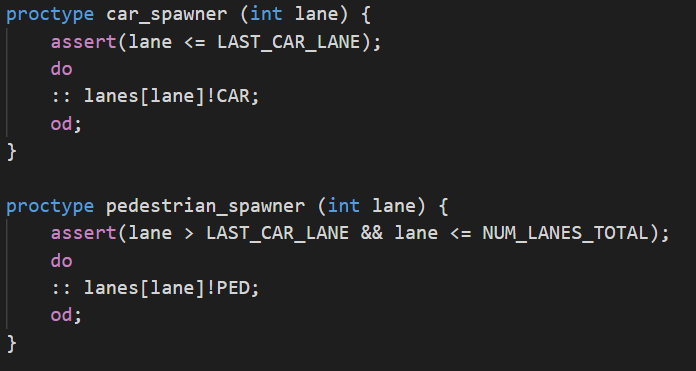


The **traffic\_light** processes obey the following algorithm:

1. Wait for cars
2. Wait for control token
3. Check intersection state (one bitwise operation: (state & conditions[lane]) == 0)
   1. If possible, the traffic light turns itself **GREEN**
4. Return control token
   1. If traffic light is not green, return to step (2)
5. Allow the cars to pass
6. Wait for control token
7. Turn **RED**
8. Return control token
9. Go to step (1) (repeat)

The **car\_spawner and pedestrian\_spawner** process types just repeatedly spawn traffic actors:





Such a generalized approach allowed us to switch the sub-configuration at the last day, when we found out that we are actually group 3 and not group 2, which we have been using in Discord the whole time.



# Verification of the model

## Safety

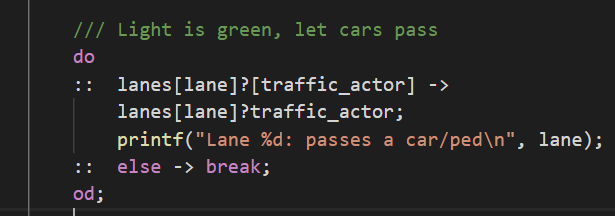
Safety property can be confirmed using a search depth of 309 950, so we can use -m400000 parameter. We also use -DNOREDUCE to enforce exhaustive verification.

***spin -search -DNOREDUCE -m400000 -ltl safety assignment.pml***

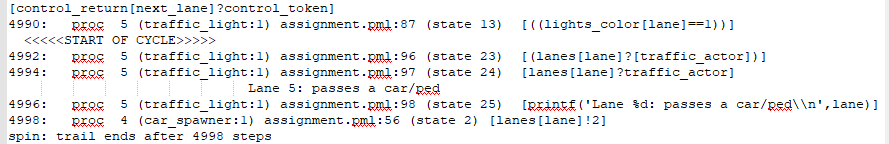
## Fairness → Liveness

We were asked to “check liveness with assumption of fairness”, which translates to “Fairness → Liveness” implication. Because the combined LTL formula seemed to be too complex to handle for our machines (compilation froze for a long time), we checked each component separately.

Initially, fairness property did not hold for our system (acceptance cycle found in generated *never* claim). This happened because in our initial version traffic light could just loop in GREEN state while cars were being generated:



Trail output:

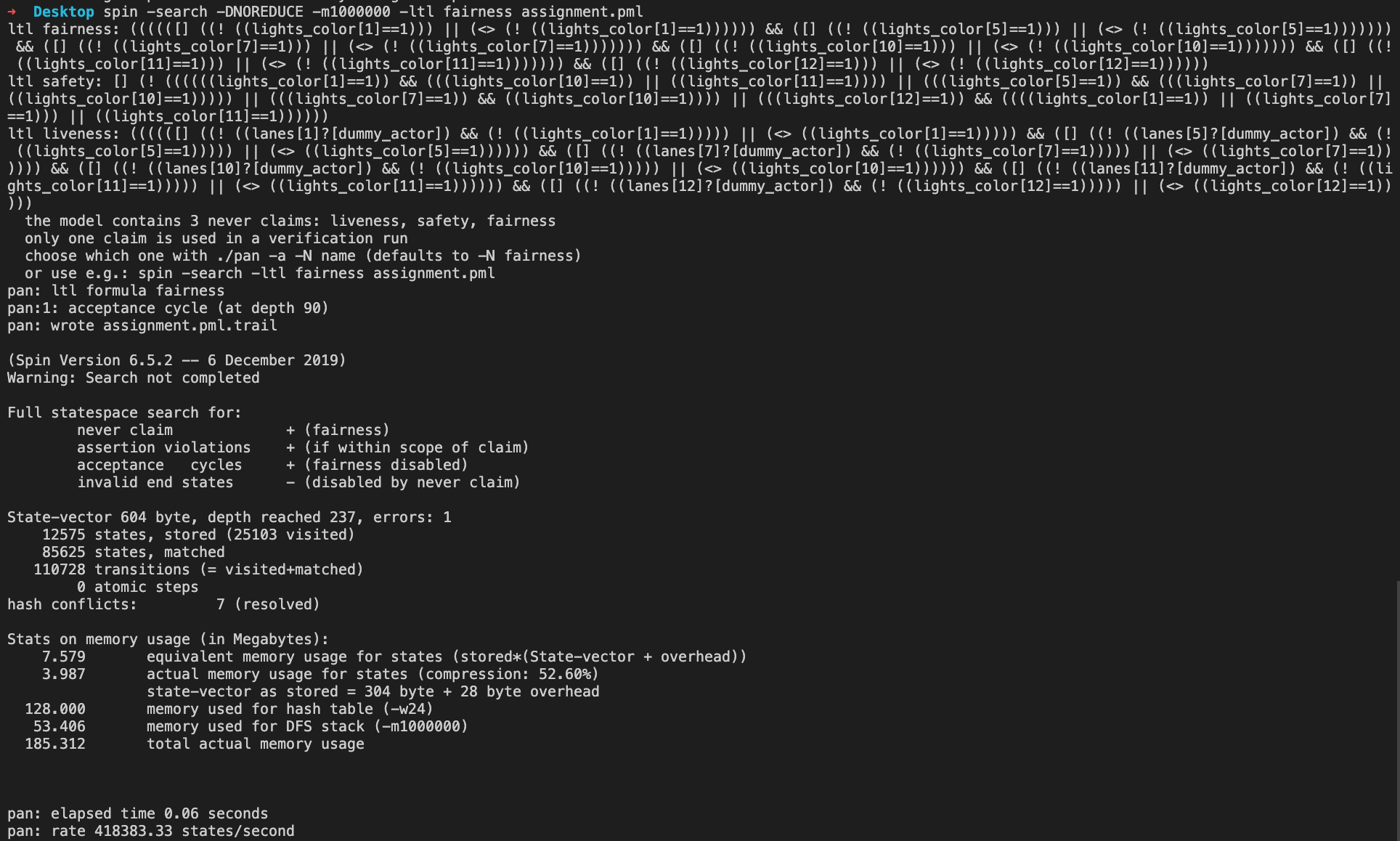


I.e. “assuming fairness” means that we assume that traffic cannot be infinite, but in our default implementation, it can be.

And because the fairness property does not hold, “Fairness → Liveness” implication holds

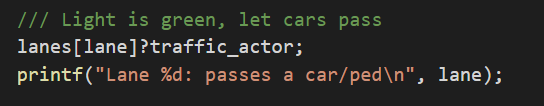
(0 → x = 1).

Below is the verification output for that version of program:

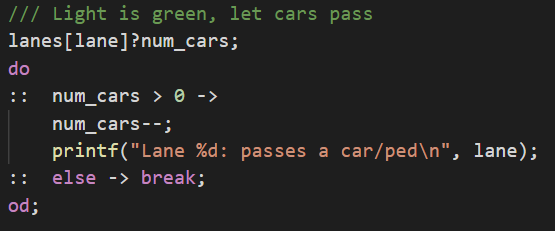
***spin -search -DNOREDUCE -m1000000 -ltl fairness assignment.pml***

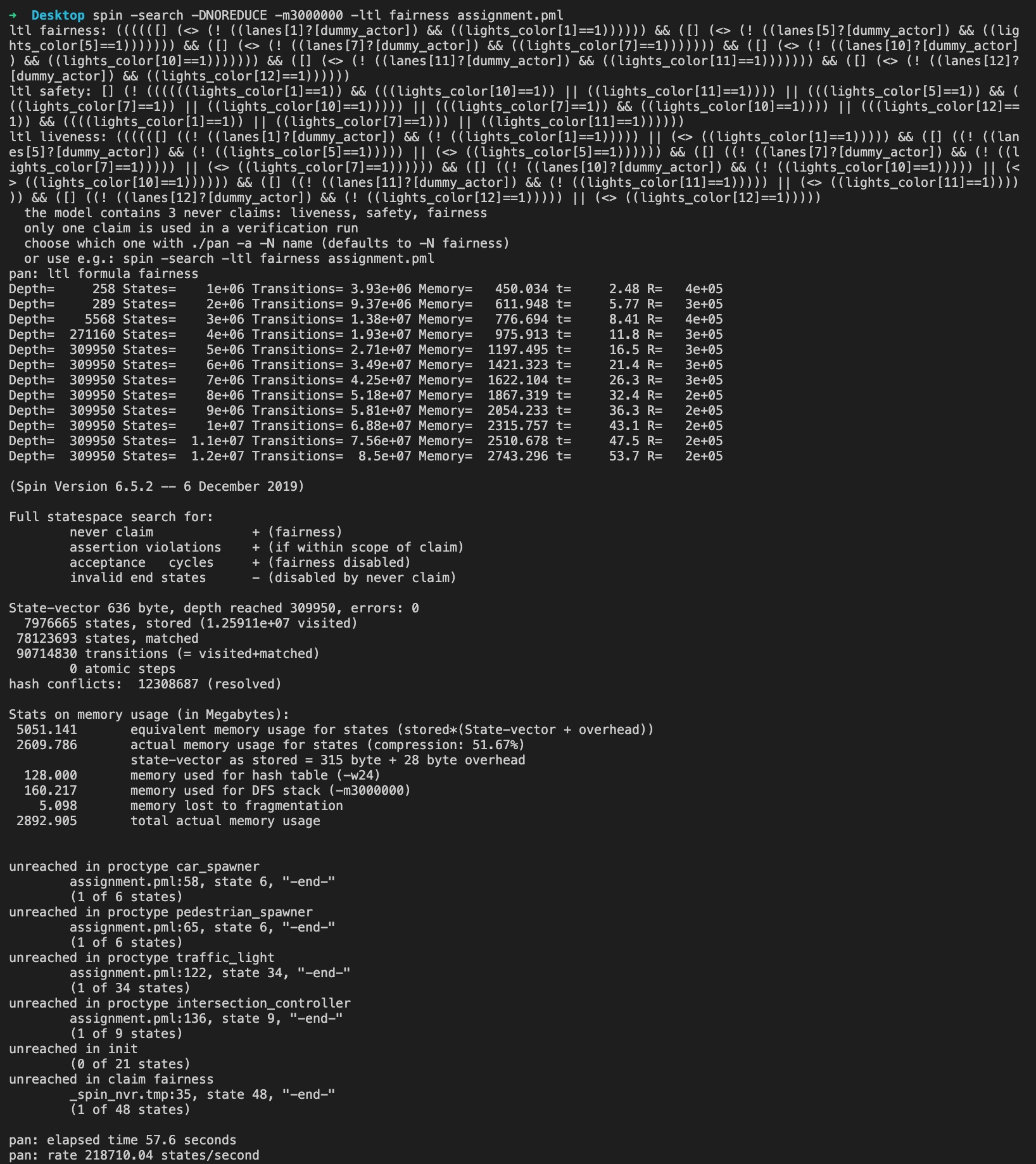
***spin -search -DNOREDUCE -m1000000 -ltl liveness assignment.pml***

There are several ways to ensure fairness in our system. One is just passing one car at a time:



Another way is to generate number of cars in current group, instead of separate cars:



In both cases, fairness property now holds for our system:

And so does liveness property: