TDTS07 Lab 1 Report

Mario Impesi and Moritz Fröhlich

March 9, 2023

1 Implementation

1.1 Intersection

Our implementation of the intersection is a single centralized module called Intersection, which has four inputs and four outputs. The input ports are used to receive the information about cars arriving at a certain light, from the input generator module. The information about cars received by the Intersection is simply a boolean that signifies whether or not cars are present at a certain traffic light. The output ports are used to broadcast the status of each light, in particular to the monitor, as shown later. These ports are also of type boolean, where "true" signifies a green light, whereas "false" signifies a red light.

```
SC_MODULE(Intersection) {
    sc_in<bool> car_status[4];
    sc_out<bool> lights[4];
    ...
}
```

The intersection has one method and one thread.

Logic Method The method is called logic_method and it controls the logic behind the traffic lights. It is sensitive to the four inputs that signal the presence of cars and an event used for the timeout.

```
SC_METHOD(logic_method);
dont_initialize();
for (size_t i{0}; i < 4; ++i) {
  sensitive << car_status[i];
}
sensitive << timeouted;
```

The logic method is responsible for ensuring that the properties 1 and 2, namely the independence of the lights and the safety constraint, are satisfied. In particular, when there is a change in the car status it will apply the following algorithm for each light: if there are cars waiting and the light is currently red, change the light to green if the orthogonal lights are red. Otherwise, if there

are no cars waiting, set the light to red.
In this algorithm, a local copy of the output values is used to perform the checks. This is done to make sure that any changes to the output values made in the current iteration are immediately visible. If we used the output channels directly, any updated value would only come into effect on the successive iteration, thus causing problems to the logic.

Timeout Thread The thread is called timeout_thread, and it's used to ensure that the third property, namely the one about starvation, is satisfied.

```
SC_THREAD(timeout_thread);
```

This thread is active once every second, it checks that no light has been green for too long, comparing four local timers to a threshold amount that is a parameter of the Intersection itself. When a light has been green for too long, it is forcibly set to red.
The timeout thread only changes the local output values for the lights. The logic method is responsible for forwarding those changes to the actual output channels and it does that when it is triggered through the timeout event.
If the previous activation of this thread resulted in a timeout, on the next activation it sends a second notification on the timeout event. This is to make sure that if a light received a timeout, and after that there are no more changes to the input signals, it can still check if it is allowed to turn back to green.

1.2 Random Input Generator

The random input generator generates a variable number of cars to arrive at a certain traffic lights, and then removes one car from the simulated queue for every second that the light is set to green.

```
SC_MODULE(RandomGenerator) {
  sc_out<bool> car_signals[4];
  sc_in<bool> traffic_lights[4];
  int max_cars;
  int rate;

  SC_HAS_PROCESS(RandomGenerator);
  RandomGenerator(sc_module_name name, int max_cars, int rate);

  void generator_thread();
};
```

The random generator takes two parameters, one sets the maximum amount of cars that can be generated to join a queue, and the other is the rate at which these cars are generated (i.e. the probability of generating them at any point).

1.3 Deterministic Input Generator

The deterministic input generator generates signals based on input files. It is used to test edge cases and, in general, verify the three properties that the system should satisfy. The input files specify how many cars arrive at each light. The deterministic generator has one thread that runs every second. It checks the status of each traffic light: if the light is currently red and there are supposed to be cars arriving there, it notifies this by writing "true" on the respective signal; if the light is currently green, it lets one car pass the light by decreasing the respective car_count value.

1.4 Monitor

The monitor module has as inputs both the car status generated from the generator, and the traffic lights status. It has on method, sensitive to the changing of the traffic lights.

```
SC_MODULE(Monitor) {
  sc_in<bool> car_status[4];
  sc_in<bool> traffic_lights[4];
  std::ostream& os{std::cout};
  int timers[4];
  int timeout;

  SC_HAS_PROCESS(Monitor);
  Monitor(sc_module_name name, int timeout);

  void check_traffic_lights_status();
};
```

Whenever a traffic lights changes to green, it checks that:

1. there are cars at that particular light, because otherwise we expect it to turn red. (property 1)
2. the orthogonal ones are red; In this case, it also updates local timers to check that the timeout is working properly. (property 2)
3. it hasn't been green for too long, using a timeout value that is a parameter of the module. (property 3)

2 Simulation

Terminology In this section we assign cardinal directions to the light indices to improve illustration. We assign light 0 to be the south light, light 1 to be the west light, light 2 to be the north light and light 3 to be the east light.

```
At time: 1 s
0: GREEN;Cars here
1: RED;No cars
2: GREEN;Cars here
3: RED;No cars
At time: 3 s
0: RED;No cars
1: RED;No cars
2: GREEN;Cars here
3: RED;No cars
At time: 5 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
```

Figure 1: Output checks/property1

First we checked the 3 properties individually using 3 deterministic scenarios that we supplied to the deterministic input generator. Then we used the random input generator with multiple edge-case inputs to test all 3 properties at once. Since we are using assertions in our monitor, the simulation should end prematurely with an exception if any of the properties are not satisified at any time during the simulation.

To start a deterministic simulation, the testbench executable (in our case `intersection.x`) has to be supplied with exactly 3 arguments. The first argument is the simulation time, the second argument is the time after which a light will be timouted and the third argument is the name of a file containing the scenario to simulate. A scenario consists of the initial car counts at each of the four lights, seperated by whitespaces. E.g. `0 1 2 3` will cause the simulator to use 0, 1, 2, 3 cars respectively at lights 0, 1, 2 and 3.

2.1 Property 1: The lights should work independently

To verify this property we supplied a scenario in which there is 1 car the southern light, 3 cars at the northern light and no cars on the other lights. We ran the simulator using the command `./intersection.x 100 100 checks/property1` from the root of the project. The output is shown in Figure 1.

We can observe that the southern light turns red in time 2, after letting the one car pass whilst the northern light stays green until letting all 3 cars pass. This shows that opposing lights are not working together. Note: since our monitor is sensitive to the light signals, we will only observe the southern light going red in the third time-step and the northern light going red in the 5th time-step.

```
At time: 1 s
0: GREEN;Cars here
1: RED;Cars here
2: RED;No cars
3: RED;No cars
At time: 12 s
0: RED;No cars
1: GREEN;Cars here
2: RED;No cars
3: RED;No cars
At time: 23 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
```

Figure 2: Output checks/property2

2.2 Property 2: Orthogonal exclusion

To verify this property, we supplied a scenario in which there are 10 cars at the southern light and 10 cars at the western light. We ran the simulator using the command `./intersection.x 100 100 checks/property2` from the root of the project. The output is shown in Figure 2.

We can observe that at any point, at most one of the lights is active. The southern light goes green first (our implementation favors lower indices) and stays green for exactly 10 time-steps. Then the western light goes green and stays green for exactly t10 time-steps as well until both lights turn red in time-step 23.

2.3 Property 3: No starvation

To verify this property we supplied a scenario in which there are 2000 cars at the southern light and no cars at any other light. We ran the simulator using the command `./intersection.x 10000 100 checks/property3` from the root of the project. The truncated output is shown in Figure 3.

We can observe that the southern light is timed out repeatedly after exactly 100 time steps, until there are no more cars. Doing so ensures that none of the traffic lights starve. Specifically, the southern light is timeouted after exactly 100 time-steps and turns green again during the next time-step, resetting its timeout timer. See time-steps 101, 102 and 202, 203 for example.
If more than this one light had cars our implementation would attempt to turn any of those lights green, since the timeouted light is only allowed to turn green again in the time-step after it was timeouted.

```
At time: 1 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 101 s
0: RED;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 102 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 202 s
0: RED;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 203 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 303 s
0: RED;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
[TRUNCATED]
```

Figure 3: Truncated output checks/property3

```
At time: 1 s
0: GREEN;Cars here
1: RED;Cars here
2: GREEN;Cars here
3: RED;Cars here
At time: 2 s
0: RED;Cars here
1: GREEN;Cars here
2: RED;Cars here
3: GREEN;Cars here
At time: 3 s
0: GREEN;Cars here
1: RED;Cars here
2: GREEN;Cars here
3: RED;Cars here
At time: 4 s
0: RED;Cars here
1: GREEN;Cars here
2: RED;Cars here
3: GREEN;Cars here
[TRUNCATED]
```

Figure 4: Truncated output `./intersection.x 10000 1 500 100`

2.4 Edge-case simulations

For starting a random simulation, the testbench executable must be supplied with exactly four arguments: The simulation time, the time after which lights will be timeouted, the amount of cars to generate once cars run out and the likelyhood of cars being generated. E.g. running `./intersection.x 100 10 20 33` means that the simulation will run for 100 time-steps, lights will be timouted after 10 time-steps and once a light turns red because there are no more cars to service, there is a 33% chance in any of the following time-step to generate between 1 and 20 new cars.

1 time-step timeout In this scenario we verify that all properties still hold, if any light is only allowed to be turned on for one time-step at a time. We supplied `./intersection.x 10000 1 500 100` and observed the truncated output shown in Figure 4.

We can observe that after every time-step, green lights switch between the orthogonal lights (southern and northern light green in t implies that western and northern light green in $t + 1$. Additionally, the simulation is completed without any assertion error, proving that all properties hold.

```
At time: 8 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: GREEN;Cars here
At time: 9 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 16 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 17 s
0: RED;No cars
1: GREEN;Cars here
2: RED;No cars
3: RED;No cars
At time: 18 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 21 s
[TRUNCATED]
```

Figure 5: Truncated output `./intersection.x 10000 1000 1 10`

Low frequency car arrival In this scenario we verify that all properties still hold, if cars are only generated rarely, that is, lights should turn on rarely (10% chance to generate a single car in every time-step). We supplied `./intersection.x 10000 1000 1 10` and observed the truncated output shown in Figure 5.

The simulation completes without any assertion error, meaning that all 3 properties are satisfied. Additionally we can observe that the first time a light turns green is the western light time-step 8, which as expected, is turned off again in time-step 9, since only one car was generated. The next time a car is generated is in time-step 16 etc.

High frequency car arrival In this scenario we verify that all properties still hold, if cars are generated at a high frequency. Specifically, one car is generated with a likelihood of 80%. We supplied `./intersection.x 10000 1000 1 80`

```
At time: 1 s
0: RED;No cars
1: RED;No cars
2: GREEN;Cars here
3: RED;No cars
At time: 2 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;Cars here
At time: 3 s
0: RED;No cars
1: GREEN;Cars here
2: RED;No cars
3: GREEN;Cars here
[TRUNCATED]
```

Figure 6: Truncated output `./intersection.x 10000 1000 1 80`

and observed the truncated output shown in Figure 6.

The simulation completes without any assertion error, meaning that all 3 properties are satisfied. The truncated output shows that there is a rapid change of cars arriving and leaving. For example there are cars at the northern light in time-step 1, then there are cars at the southern and the eastern light in time-step 2 and cars at the northern and the eastern light in time-step 3. Even under rapidly required change, the properties still hold.

Average scenario Finally, we simulated an average, that is a non-edge case, scenario supplied using `./intersection 10000 6 8 20` in which lights are timed out after 6 time-steps and between 1 and 8 cars will arrive with 20% probability. The truncated output is shown in Figure 7.

The simulation finishes with no assertion failures, meaning that all properties are satisfied.

```
At time: 1 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: GREEN;Cars here
At time: 3 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 6 s
0: RED;No cars
1: GREEN;Cars here
2: RED;No cars
3: RED;No cars
At time: 7 s
0: RED;No cars
1: RED;No cars
2: RED;No cars
3: RED;No cars
At time: 9 s
0: GREEN;Cars here
1: RED;No cars
2: RED;No cars
3: RED;No cars
[TRUNCATED]
```

Figure 7: Truncated output `./intersection.x 10000 6 8 20`