TDTS07 Lab Report

Justus Rossmeier

2024-02-20

Contents

1 Modeling and Simulation with SystemC

1.1 System Model (`Controller`)

The modeled crossing is depicted in Figure 1 and contains one lane per incoming car, with cars only ever passing the crossing straight without turning. As soon as coming from a side, the respective sensor is activated and stays active as long as cars are waiting on that side. If the light is green, one car per second will pass the crossing, ultimately cleaning out the queue if no new cars are arriving.

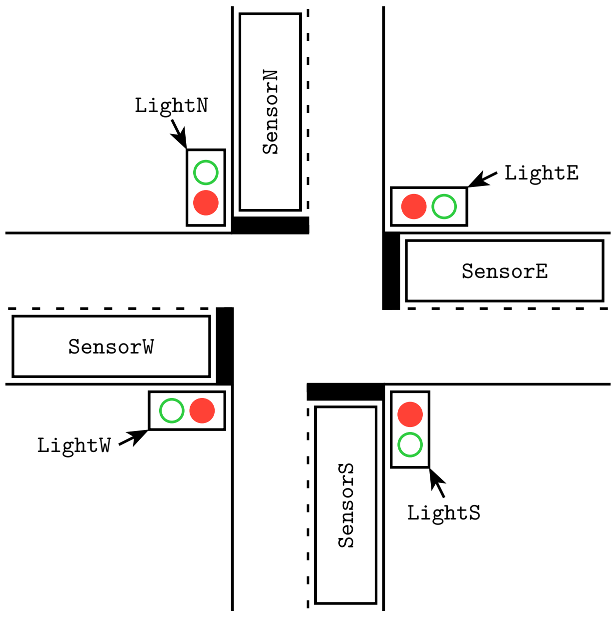

Figure 1: System overview

Each sensor is modeled as an input `sc_in<bool>` and each light as an output `sc_out<int>`, where the `int` is actually an instance of the enum `LightColor` in the code but `int` is used instead to work nicely with the included tracing of SystemC.

The controller generally operates by setting a direction's light to green as long as the respective sensor indicates that cars are waiting. To ensure safe operation, the controller maintains a `state` indicating which direction cars are allowed to pass. This can be `Idle`, `NorthSouth`, or `WestEast`. Each light will only be turned green if the controller is in the respective state. If a sensor turns on and the state is idle, the state will be changed to the required state by the respective light. If the state does not permit the light to turn green, a state change will be requested and scheduled by triggering an `sc_event` after a specified `max_wait_time`. When this event is triggered, a state change is forced by first turning all lights red and then changing the state to the opposite of the previous one. If all lights turn red because no more cars are detected, the controller either changes to the other state if it has been previously requested or returns to the `Idle` state.

1.2 Traffic Generator (`Generator`)

There are 4 traffic generators in the simulation. Each one is connected to one side of the crossing. Accordingly, the generator provides the following interface:

- `sc_in<int> light` for getting the light signal's color relevant for the generator
- `sc_out<bool> sensor` for feeding back sensor data to the controller
- `sc_out<int> queue_length` for being able to trace and debug the internal queue length of the generator

The generator handles both the arrival of new cars, which occur at fixed times and are random with a likelihood depending on the generator's predefined `pressure` value

$$P_{\text{arrival}} = \begin{cases} \min(1.0, \text{pressure}) & \text{if queue\_length} = 0 \\ \min\left(1.0, 1.0 - \frac{\text{queue\_length}}{\text{pressure}}\right) & \text{else} \end{cases}$$

and the passing of cars over the crossing whenever the light signal is green. Both those events are evaluated and executed once a second with a 100 ms delay between both events to better visualize the events in the trace file.

1.3 Testbench (`TestBench`)

The test bench contains an `sc_main` function that instantiates one `Controller` with one `Generator` per side. All relevant signals are initiated and connected to the entities as well as traced to an output file.

1.4 Results

To verify functionality, the generator is tested with a scenario that visualizes all edge cases. To achieve reproducibility, the random generator is initialized with a fixed seed at the beginning of the simulation. To nicely reproduce edge cases, the following pressure values are used for the `Generators`:

- North: 10
- South: 5
- West: 2
- East: 0.2

The high pressure for north and south cause a constant flow of cars from the respective directions, so both lights are essentially always green. The state is only changed after the timer for the `max_wait_time` has expired. The pressure from the west and east side is considerably lower so that sometimes the queues are emptied out and the state is changed back to `NorthSouth` before the timer's expiry. The very low pressure on the east side is additionally there to ensure that that individual light turns red when no cars are arriving anymore, even if the opposite light stays green. Figure 2 shows the resulting traces. It can be seen that all rules and intended behaviors are fulfilled. Only opposing lights are green at the same time (value of 1 in the plot) and the periods get scheduled based on need with a maximum switching time of 10 seconds.

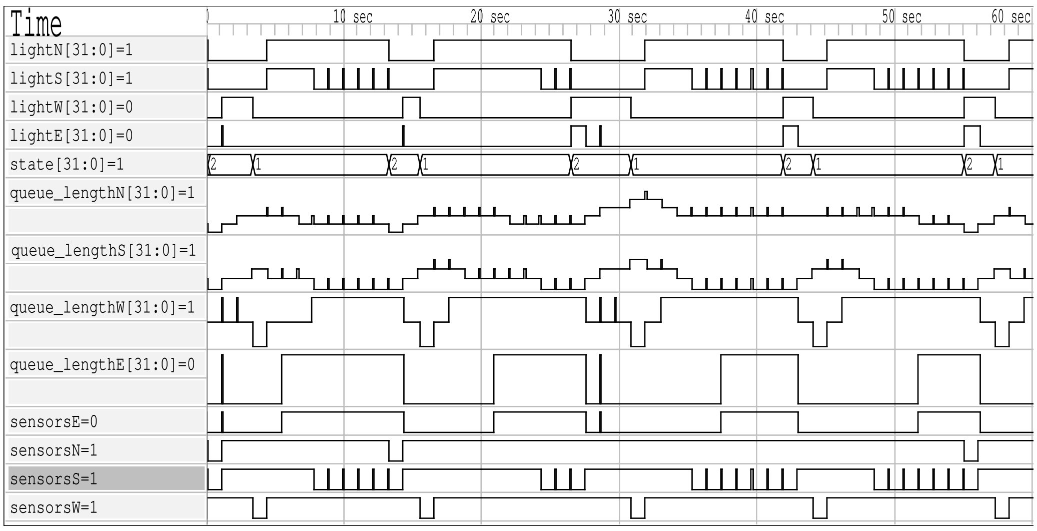

Figure 2: Waveforms of the simulation run

2 Formal verification with UPPAAL

2.1 Getting Started

- `E<> P.s3` is satisfied as there are clearly paths that lead to the `s3` state for the automation `P`.
- `A<> P.s3` is not satisfied. This is easier to understand using the equivalent `not E[] not P.s3`[1] where it becomes quite apparent that `P` is not in `s3` on many occasions.

2.2 Fischer

To adapt the example for higher values of `n`, the additional automats are just added to the system declarations. The query for verifying the mutex condition is generated using the python script shown in Listing 1.

```python
print(
    "A[] not ("
    + " or ".join(
        f"(P{i}.cs and (" + " or ".join(f"P{j}.cs" for j in range(i + 1, N + 1)) + "))"
        for i in range(1, N)
    )
    + ")"
)
```

Listing 1: Python code for generating the queries for any N

The results of the calculation times (measured with a smartphone stopwatch) are presented in Table 1. It is noteworthy that the first evaluation after altering the model is a substantially slower than subsequent evaluations. The times given are the ones of the subsequent evaluations.

N	time (s)
8	0.42
9	0.65
10	1.0
11	3.0

Table 1: Runtimes of the query evaluation

Linear regression with the model time $(N) = a \cdot b^N$ leads to $a = 2.32 \cdot 10^{-3}; b = 1.88$ and thus time$(12) = 4.6$ s.

When introducing a different minimum wait time $m \neq k$ for the transition from `wait` to `cs`, the mutex condition holds iff $m \leq k$. This makes intuitive sense because it is required that all processes have a chance to detect the foreign lock before proceeding to the critical section.

2.3 Traffic Light Controller

The controller consists of 5 different automata, where opposing directions use the same template. The different automata used are shown in Figure 3 which also shows that the automata for the different directions are structured nearly the same, with just the synchronization channels swapped.

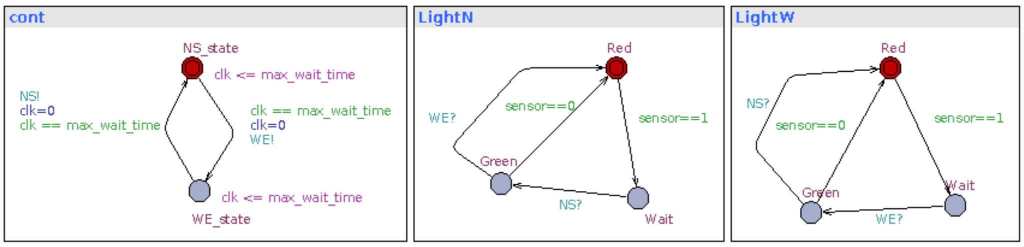

Figure 3: Different automata used for modeling the traffic light controller

[1]https://docs.uppaal.org/language-reference/query-semantics/symb_queries/

For simplicity in this model, different from the controller described in Section 1, the controller does not automatically switch to the other direction if no cars are coming, but is governed by a fixed timer that alternates between both possible directions. The queries ensuring the correct operation of the controller are shown in Listing 2 although the last query exists similarly for every direction.

```
A[] not deadlock
A[] not ((LightN.Green or LightS.Green) and (LightW.Green or LightE.Green))
E<> LightN.Green
```

Listing 2: Queries used to verify the model

2.4 Alternating Bit Protocol

The send events (`s0`, `s1`, `sack0`, `sack1`) are modeled using synchronous channels that are written to be the sender/receiver and read by the channel. The corresponding receive events (`r0`, `r1`, `rack0`, `rack1`) are modeled as broadcast channels, as the receiver/sender might not always be able to receive them depending on their current state. To make the usage of clocks and synchronization channels more intuitive, the same logical state is partially split up in several states where all but the last are "urgent" so the synchronization still works correctly.

2.4.1 Sender

The sender automat depicted in Figure 4 generates a new message after being idle for `gen = 3` time-units. Each new message is sent to the correct channel according to the current status of the sender. After sending, if the sender receives the correct ack message, it changes state and goes back to idle. If the timeout (default 5 s) is reached or the wrong ack message is received, the transmission is attempted again.

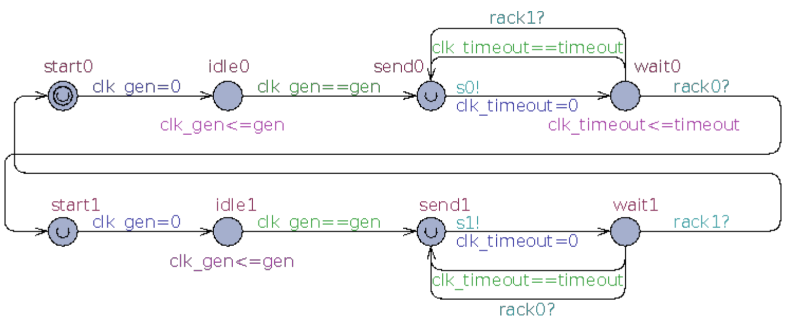

Figure 4: The sender

2.4.2 Channel

The channel is depcited in Figure 5. It models a binary symmetrical channel for both the data as well as the ack messages. It receives any of the potential send events (`s0`, `s1`, `sack0`, `sack1`) and emits the corresponding or opposite receive event (`r0`, `r1`, `rack0`, `rack1`). The error probabilities are modeled by exploiting the fact that UPPAAL randomly chooses one of all possible edges every time. The modeled error probability $P_{\text{error}} = \frac{1}{4}$ is achieved by having 3 "correct" edges and 1 "error" edge for each possible event.

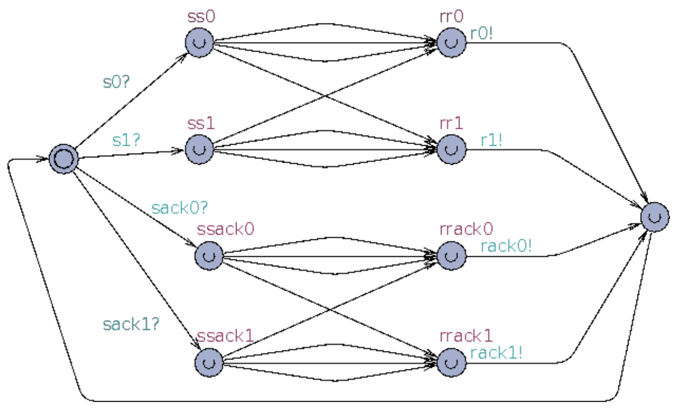

Figure 5: The channel

2.4.3 Receiver

The receiver is depicted in Figure 6. It works very similarly to the sender. When receiving the correct message for the current state, it emits the corresponding ack and changes state. When timing out or receiving the wrong message, it emits the opposite ack and goes back to waiting.

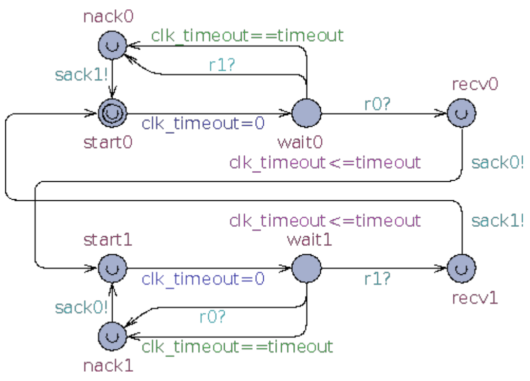

Figure 6: The receiver

2.4.4 Verification

The following queries are used to verify correct function

- `S.send0 --> R.recv0`: ensure that all messages sent by the sender are eventually received by the receiver. This passes with using an ideal channel, but does not pass using the binary symmetrical channel described in Section 2.4.2.
- `R.recv0 --> C.ssack0` and `R.recv0 --> (C.rrack0 or C.rrack1)`: ensure that the receiver tries to acknoledge received messages either successfully or unsuccessfully (because of the lossy channel)
- `A[] not deadlock`: ensure that the whole system can not end up in a deadlock condition

3 Design-space exploration with MPARM

3.1 Energy Minimization

The results of the different parametrizations are shown in Table 2. Run 1 uses the default settings of the simulator. After observing the very low cache miss rates, the cache sizes were reduced (eg. runs 2 and 3), resulting in run 2 for optimal cache sizes. Then the frequency divider was systematically increased (runs 4 to 6), with diminishing returns for a divisor larger than 6. Thus the result of run 6 is considered optimized and still contains considerable headroom to the 20 ms time limit as well.

Run	Parameters	Time	Energy	D-Miss	I-Miss
1	`-F0,1 --dt=4 --ds=12 --it=1 --is=13`	1.69 ms	45.03 µJ	1.13 %	0.61 %
2	`-F0,1 --dt=4 --ds=9 --it=1 --is=9`	1.74 ms	29.77 µJ	2.35 %	0.92 %
3	`-F0,1 --dt=4 --ds=9 --it=1 --is=8`	1.94 ms	33.13 µJ	2.15 %	2.62 %
4	`-F0,2 --dt=4 --ds=9 --it=1 --is=9`	3.58 ms	18.90 µJ	2.35 %	0.92 %
5	`-F0,3 --dt=4 --ds=9 --it=1 --is=9`	5.10 ms	18.02 µJ	2.35 %	0.92 %
6	`-F0,6 --dt=4 --ds=9 --it=1 --is=9`	9.68 ms	17.83 µJ	2.35 %	0.92 %

Table 2: Results of the different runs

3.2 Concurrency Optimization

Without any further modifications, the most apparent difference between the two versions is that the shared version takes more time with 14.1 ms vs 10.0 ms with the queue version. However, the queue version has a higher relative bus occupation of 56.25% vs 44.89%.

To reduce the shared version's bus occupation even further, the core clocks are lowered. This is done by setting the `-Fx,y` arguments for the simulation. Lowering all core clocks by a factor of two gives a lower bus occupation of only 23.57% but a total runtime of 26.3 ms which does not satisfy the time requirement of 20.0 ms. Using one full-speed and two half speed cores results in bus usages of around 30%. The configuration `-F0,2 -F1,2 -F2,1` is able to complete the decoding in 18.8 ms with a bus occupation of 31.78%.

3.3 Mapping/Scheduling

The critical path of the problem is T1 $\longrightarrow$ T3 $\longrightarrow$ T5. By executing this path immediately and without any unnecessary delays between tasks, an optimal schedule can be achieved. The most straightforward solution to this problem is to execute this critical path on one processor and the other tasks (T2 and T4) on the other processor. Such a schedule with SL = 35 ms is shown in Figure 7.

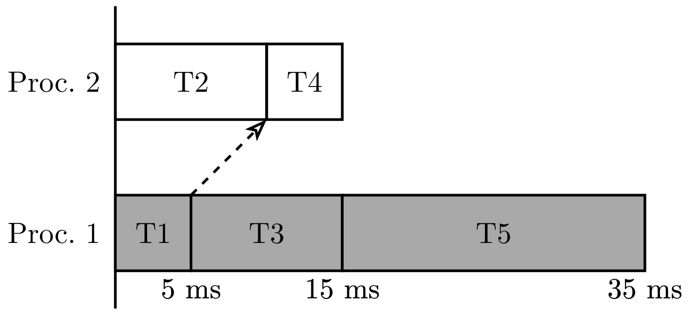

Figure 7: Schedule with changed task assigments

The same optimal schedule length can however also be reached while still keeping the original assignments of tasks to processors by evaluating the T2 $\longrightarrow$ T4 path while T5 is running on the other processor. The resulting schedule in shown in Figure 8.

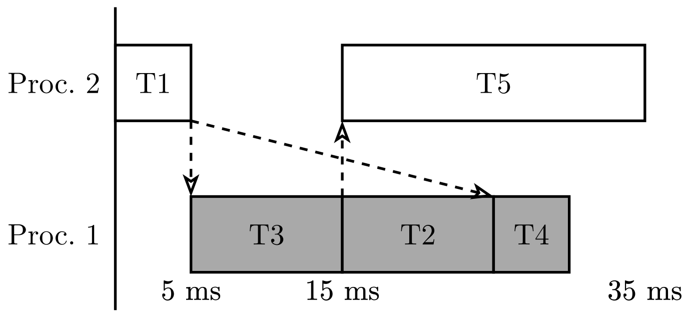

Figure 8: Schedule with original task assignments