TDTS07 Lab 2 Report

Mario Impesi and Moritz Fröhlich

March 9, 2023

1 Fischer 1

It would take roughly 115 seconds to verify the mutual exclusion property for 12 processes. The measuring results indicate the expected exponential growth of verification time in relation to the number of processes. Our measurement are shown in Table 1. The measurements were taken over ThinLinc, which is why they are probably larger than the times taken on one of the lab computers.

Process count	Mutex verification time in s
8	$< 1s$
9	$2s$
10	$6s$
11	$27s$
12	$115s$

Table 1: Mutex requirement verification times for the Fischer protocol

2 Fischer 2

When we change the guard from $x > k, id == i$ to $x > m, id == i$, with $m = 1$, the mutual exclusion condition on the critical section is no longer verified. This is because one or more processes can go from state `req` to state `wait` and then directly to the critical section. Increasing m we discover that the mutual exclusion condition holds as long as m is greater than or equal to k. In this case, once a process reaches the `wait` state, it has to wait enough seconds to allow any other processes in `req` to advance, thereby changing the `id`.

3 Traffic Light Controller

Our intersection model is decentralized and consists of four traffic light controller instances (on for each traffic light). An image of the controller template is shown in Figure 1.

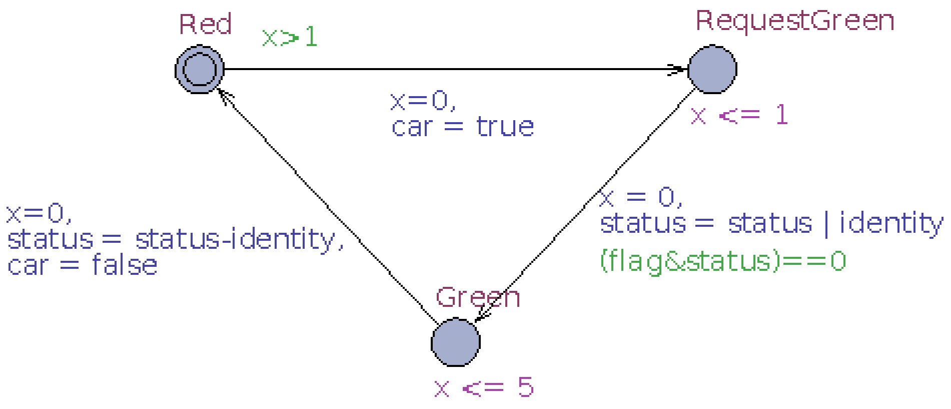

Figure 1: Template for a light controller

Each light controller has a local clock x that is used to ensure that: cars arrive at the light, any light will eventually turn green and a light can only stay green for a limited amount of time. We encoded the global status of the intersection, that is, which light controllers are currently turned on and which aren't, in a global variable `status` using bit-flags. Specifically, each light is assigned an `identity` corresponding to a power of 2. Since we have four lights (1, 2, 3, 4), they are assigned identities $1, 2, 4, 8$ respectively. In addition, each light is passed the (bit)-`flag` of the lights that are orthogonal to them. For example: Light 1 is passed the flag $2 + 8 = 10$, which is the combined bit-flag of lights 2 and 4. For a light to be able to go green, it checks that the bit-wise AND of the flag and the global status is 0. This will only be the case, if none of the orthogonal lights have flipped their bit, i.e. none of them are green.

If a light turns green, it flips the identity bit of the global status and if it goes from green to red, it flips it again. This way the global status always contains information about the state of all four controllers.

Liveliness verification We are verifying the liveliness property using the query `P1.car --> P1.Green` and `A[] not deadlock`. The first query checks that if a light requests to go green because of car arrival, all of the computation paths will eventually lead to the green state for the light. The second query just checks that the system cannot go into a state of deadlock. Both of the queries are satisfied for four instances of traffic light controllers.

Safety verification We are verifying the safety property using the query `A[] !((P1.Green or P3.Green) and (P2.Green or P4.Green))`. This query checks that in any state, none of the lights that are orthogonal to each other are simultaneously in the green state, but allows for opposing lights to be green at the same time. This query is also satisfied.

4 Alternating Bit Protocol

4.1 System Model

To model a system that implements the Alternating Bit Protocol, we used three UPPAAL templates: one sender, one receiver and one unreliable channel. The system has four global channels (send, receive, send_ack and receive_ack) and two global integer variables (ack and msg).

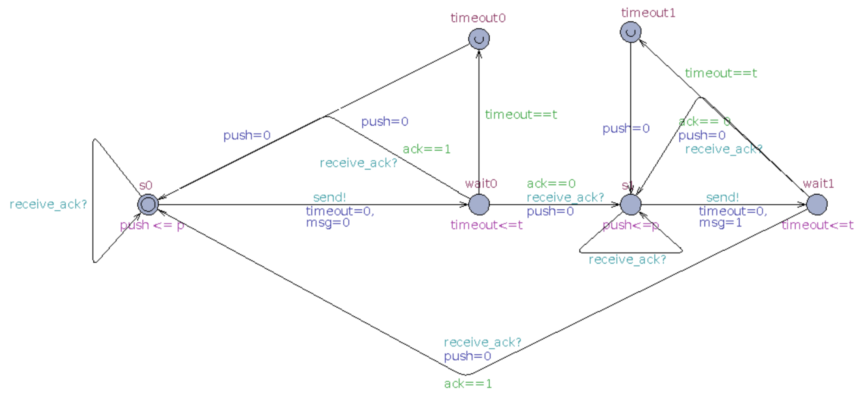

Figure 2: Sender

Sender The sender has two local clocks. One is called `timeout` and the other is `push`. The sender starts in `s0`. In this state it could receive an old acknowledgment from the receiver, which will be ignored. To simulate the willingness from the sender to actually send a message, we have included an invariant that uses the `push` clock. From `s0`, the sender will send a message with the value `0`, and it will move to state `wait0`, after resetting the `timeout` clock and after synchronizing with the channel, through `send`. At this point the sender is waiting to receive an acknowledgment from the receiver. If it receives one, it first checks that it's an acknowledgment for the correct message, in this case `0`. If it is, it will move ahead to `s1`, which works in a mirrored way to `s0`. If the acknowledgment is wrong, or nothing was received before the `timeout` triggered, then the sender will return to `s0` and send the message again.

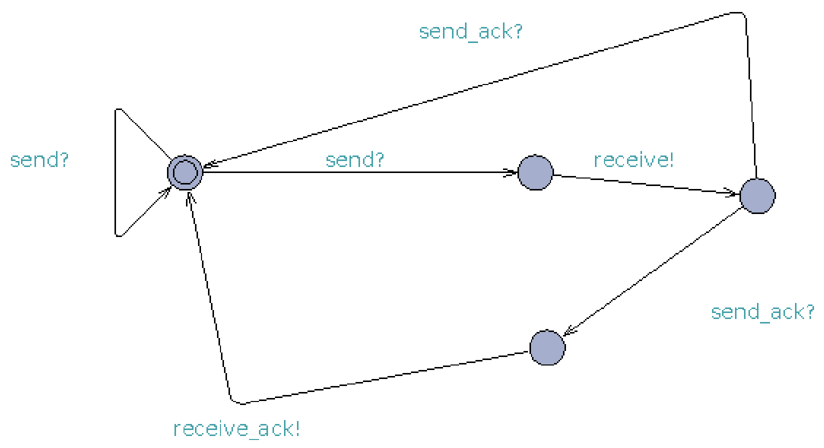

Figure 3: Channel

Channel The channel starts in a state where it waits for the sender to send. When it receives the signal to send from the sender, it has to options. It could *lose* the message, and stay in the same state, or it could successfully send the message to the receiver, by synchronizing with the receiver through the `receive` channel. At this point, the channel waits for the receiver to send an acknowledgment, which can also sometimes be lost, or delivered to the sender.

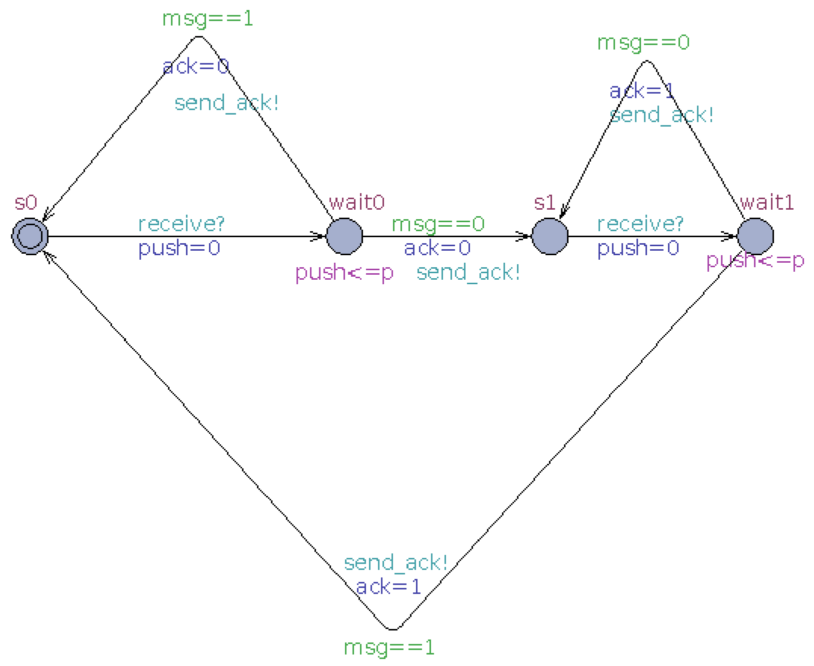

Figure 4: Receiver

Receiver The receiver has one local clock called `push`. The receiver starts in the state `s0`, where it waits to receive something from the sender, through the channel. When it does, it moves to the state `wait0`. In this state it looks at the message received: if the message is the correct one, it moves on to state `s1`, which works in a mirrored way to `s0`; otherwise, it will go back to `s0`. In both cases, it will send an acknowledgment to the sender.

4.2 Property Verification

The first property (messages sent by the sender are eventually received by the receiver) is not verified because every message can potentially be lost. The second property (the receiver might send an acknowledgment) is satisfied because the channel is unreliable so the acknowledgment can be sent or it can be lost. We check this property with two queries that verify that both outcomes are possible:

1. $E <>$ `S.timeout1`,
2. $E <>$ `S.timeout0`

The third property (the system can not deadlock) is satisfied. We check this with the query: `A[] not deadlock`.