UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO

Formal Methods

Hackathon

João Gabriel Reis Saraiva de Andrade

Natal - RN

July. 2023

Traffic lights controller 1

Project Status

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
CTX	OK	OK	0	0	0	-
CTX_i	OK	OK	3	3	0	-
M0_to_complete	OK	OK	46	42	4	-

Image 1: Traffic controller 1 project status

The M0_to_complete was the only component with Proof obligation remaining unproved, being 4 on total.

Explanations

```
    ----
VARIABLES
    rs, /* red signals */
    os, /* orange signals */
    gs, /* green signals */
    bs, /* blinking (orange signals) */
    ds, // the lights are off
    rpc, //red pedestrian crossing
    gpc //green pedestrian crossing
INVARIANT
3/3 rs <: SIGNALS &
1/1 os <: SIGNALS &
3/3 gs <: SIGNALS &
    bs <: SIGNALS &
    ds <: SIGNALS &
    (bs = SIGNALS or bs ={}) & /* all blinking or none blinking */
5/9 rs \/ os \/ gs \/ bs \/ ds = SIGNALS & /* coherency */
9/9 rs /\ os /\ gs /\ bs /\ ds = {}  /* coherency */
    // No more than one same lane as destination
6/7 & gs : {{},{S1,S3},{S2,S4}} //determines that only these configurations are possible as green signals
2/2 & rpc <: CROSSINGS
2/2 & gpc <: CROSSINGS
2/3 & rpc \/ gpc = CROSSINGS
2/3 & rpc /\ gpc = {}
5/7 & PEDESTRIAN_CROSSINGS[ran(TRANSITIONS[gs])] /\ gpc = {}
INITIALISATION
2/2 rs := {} ||
2/2 os := {} ||
4/4 gs := {} ||
2/2 bs := {} ||
2/2 ds := SIGNALS ||
2/2 rpc := CROSSINGS ||
3/3 gpc := {}
```

Image 2: M0_to_complete variables, invariant and initialization

According to what's on display by Image 2, the main alterations came from the adding of the pedestrian crossings to the model, with them were added the rpc and gpc variables as well as the PEDESTRIAN_CROSSINGS relation and the CROSSINGS set as shown in the image 3.

Other alterations to the machine came from the addition of the ds variable, to indicate the off state on the signals lights.

```
MACHINE
    CTX
SETS
    LANES = {L1, L2, L3, L4};
    SIGNALS = {S1, S2, S3, S4};
    CROSSINGS = {PC1,PC2,PC3,PC4}
CONSTANTS
    INTERSECT_LANES,
    TRANSITIONS,
    PEDESTRIAN_CROSSINGS

PROPERTIES
    INTERSECT_LANES <: LANES * LANES &
    TRANSITIONS : SIGNALS <-> ( LANES * LANES ) &
    PEDESTRIAN_CROSSINGS: LANES <-> CROSSINGS &
    INTERSECT_LANES = {
        L1 |-> L2, L1 |-> L4,
        L2 |-> L1, L2 |-> L3,
        L3 |-> L2, L3 |-> L4,
        L4 |-> L1, L4 |-> L3} &
    TRANSITIONS = {
        S1 |-> (L1 |-> L1),
        S1 |-> (L1 |-> L4),
        S2 |-> (L2 |-> L2),
        S2 |-> (L2 |-> L1),
        S3 |-> (L3 |-> L3),
        S3 |-> (L3 |-> L2),
        S4 |-> (L4 |-> L4),
        S4 |-> (L4 |-> L3)
    }
    &
    PEDESTRIAN_CROSSINGS = {
        L1 |-> PC1,
        L2 |-> PC2,
        L3 |-> PC3,
        L4 |-> PC4,
        L3 |-> PC1,
        L4 |-> PC2,
        L1 |-> PC3,
        L2 |-> PC4
    }
END
```

Image 3: CTX machine

```
turn_off =
BEGIN
    rs := {} ||
    os := {} ||
    gs := {} ||
    bs := {} ||
    ds := SIGNALS
END;

reset_turn_on =
PRE ds = SIGNALS
THEN
    rs := {} ||
    os := {} ||
    gs := {} ||
    bs := SIGNALS ||
    ds := {}
END;

start_exploitation = /* moving from all blinking to all orange */
PRE
    bs = SIGNALS
THEN
    bs,os := {}, SIGNALS
END;
```

Image 4: turn_off, reset_turn_on and start_explotation operations from M0_to_complete

The behavior of the operations shown on the Image 4 are as follows: turn_off operation turns every light to the off state; reset_turn_on turns every light from off to blinking; start_exploration turns every light from blinking to orange.

```
    orange_to_red = /* moving from all orange to all red */
    PRE /*Had to change the precondition and the operations because the requirement for the os = {} on red_to_green1
and 2 led to a state where the lights became fixed and couldn't change.*/
        not (os={})
    THEN
3/3     os,rs := {}, rs\/os
    END;

    red_to_green1 = // set S1 and S3 to green
    PRE
        gs = {} & {S1,S3} <: rs & os = {} & PEDESTRIAN_CROSSINGS[ran(TRANSITIONS[{S1,S3}])] /\ gpc = {} & (rs-{S1,S3})
\/ os \/ gs \/ {S1,S3} \/ bs \/ ds = SIGNALS
    THEN
5/5     gs := {S1,S3} ||rs := rs-{S1,S3}
    END;

    red_to_green2 = // set S2 and S4 to green
    PRE
        gs = {} & {S2,S4} <: rs & os = {} & PEDESTRIAN_CROSSINGS[ran(TRANSITIONS[{S2,S4}])] /\ gpc = {} & (rs-{S2,S4})
\/ os \/ gs \/ {S2,S4} \/ bs \/ ds = SIGNALS
    THEN
5/5     gs := {S2,S4} ||rs := rs-{S2,S4}
    END;
```

Image 5: orange_to_red, red_to_green1 and red_to_green2 operations from M0_to_complete

Orange_to_red changes the state of the lights that are currently orange to red, as can be seen on the commentary shown within Image 5 the precondition had to be changed to accommodate red_to_green1 and red_to_green2 behavior in a way that the machine wouldn't be unable to continue changing the state of the lights.

The red_to_green1 and red_to_green2 operations are very similar, their behavior is changing a pair of lights from red to green. The precondition of both ensures that a light wont turn green if a pedestrian light is green.

```
    green_to_orange = // Set green signals to orange
    PRE
        not(gs={})
    THEN
4/5     os := os \/ gs || gs := {}
    END;

    red_to_green = //it breaks the invariant (on purpose)
    PRE not(rs = {})
2/5 THEN gs := rs\/gs || rs := {}
    END;

    rpc_to_gpc(cr) = // Turns a red pedestrian light into green
    PRE cr: CROSSINGS & cr : rpc & (rpc-{cr}) \/ (gpc \/ {cr}) = CROSSINGS & ((rpc-{cr}) /\ (gpc \/ {cr})) = {} &
(PEDESTRIAN_CROSSINGS~[{cr}] /\ ran(TRANSITIONS[gs])) = {} & PEDESTRIAN_CROSSINGS[ran(TRANSITIONS[gs])] /\ (gpc \/
{cr}) = {} //a rpc can only be turned into a gpc if there are no gs that allow passage through the lanes that intersect
with the crossing
2/2 THEN gpc:= gpc \/ {cr} || rpc := rpc-{cr}
    END;

    gpc_to_rpc(cr) = // Turns a green pedestrian light into red
    PRE cr: CROSSINGS & cr : gpc
5/5 THEN rpc:= rpc \/ {cr} || gpc := gpc-{cr}
    END

END
```

Image 6: green_to_orange, red_to_green, rpc_to_gpc and gpc_to_rpc operations from M0_to_complete

The green_to_orange operation changes the state of all green signals to orange. There is a proof obligation that wasn't proven automatically but it's intuitively correct. The missing proof is *rs ⋁ (os ⋁ gs) ⋁ {} ⋁ bs ⋁ ds = SIGNALS*. Given that gs will be assigned all the elements from rs and rs will be assigned empty, it follows that the union of all the possible signal states will remain equal to *SIGNALS*.

The red_to_green operation is purposefully wrong and made to test the invariant. As shown in the Images 7 and 8, when the operation is present there will be an invariant violation. The missing proofs are: *{} ⋁ os ⋁ (rs ⋁ gs) ⋁ bs ⋁ ds = SIGNALS*, a similar case to the one from green_to_orange; *rs ⋁ gs : {{}, {S1, S3}, {S2, S4}}*, this is impossible to prove as correct, it's what causes the invariant to break; *PEDESTRIAN_CROSSINGS[ran(TRANSITIONS[rs ⋁ gs])] /\ gpc = {}*, also possible to break the invariant if there is a green pedestrian light.

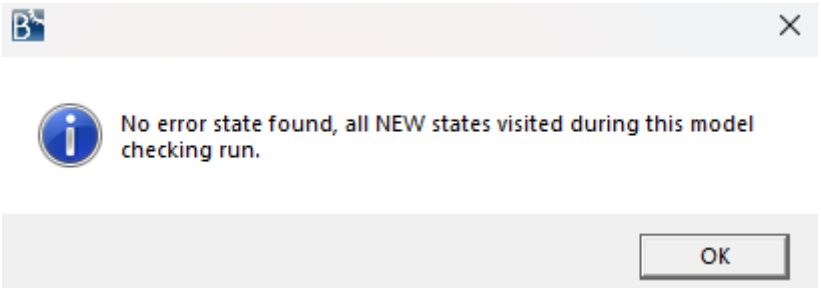

Image 7: model check without the red_to_green operation

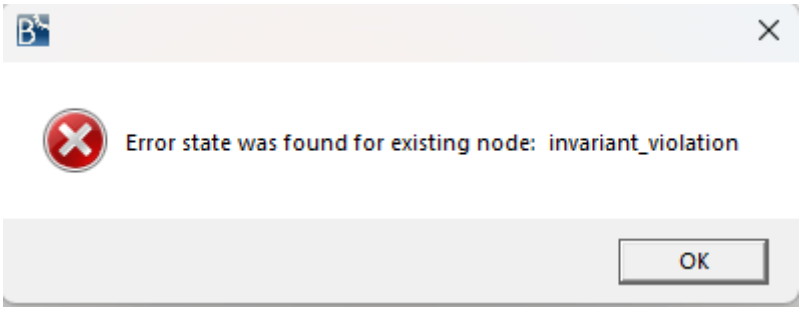

Image 8: model check with the red_to_green operation

The rpc_to_gpc and gpc_to_rpc operations are responsible for changing a pedestrian light from red to green and green to red, respectively. The way the street was created makes it impossible to have a pedestrian light turn green if any light is not red, that is expressed in the invariant of rpc_to_gpc.

Efficiency

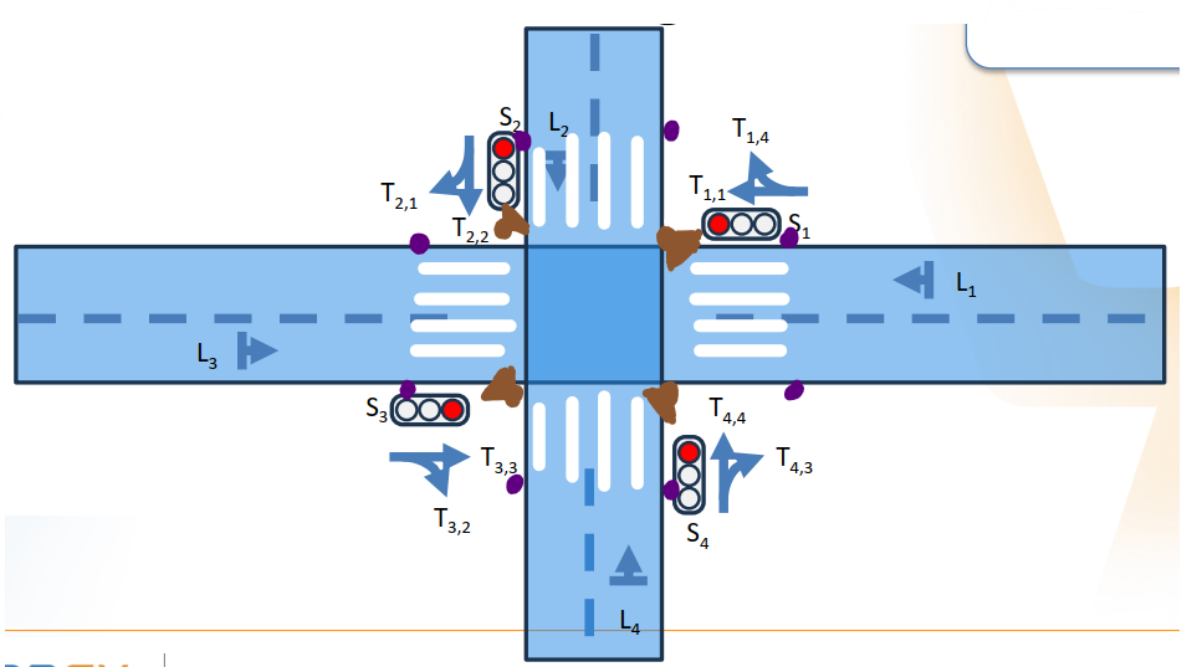

Image 9: Traffic lights 1 problem with drawings of pedestrian crossing and extra equipment.

In the Image 9 the white lines represent the pedestrian crossings, the brown triangles represent cameras that with the help of image recognition software can be able to identify the flow of traffic though the intersection and the purple dots represent buttons that allow the pedestrians to request for the pedestrian lights to be turned green.

The cameras cannot be used as a reliable method for identifying cars and pedestrians with enough accuracy that makes it safe but it can be used to estimate the flow and the data used to determine what lights should be staying green for longer.

The purple buttons make it possible to allow the pedestrian lights to turn green. It's impossible for a pedestrian to cross safely though any of the crossing if at least one light is not red, that way the presence of buttons would allow for a better flow of cars, given that all the signal lights would only turn red at the same time if a pedestrian pressed the button.

Traffic lights controller 2

Modeling

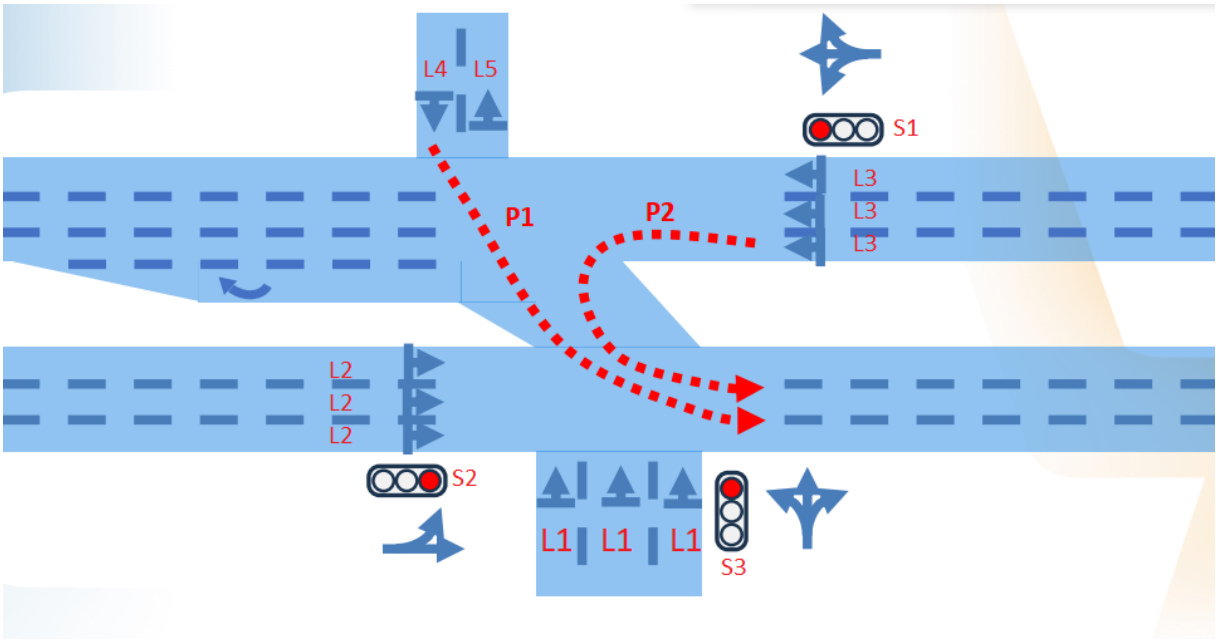

Image 10: Traffic lights controller 2 visual model with added Signal and lanes names

Proposed scheduling

A possible schedule for the signals depends on whether or not P1 is allowed, for safety reasons P1 should only be allowed if all the 3 signals are red but even then there would be no way of communicating that to the drivers coming from the road where P1 starts, so P1 shouldn't be allowed and won't be considered for the rest of the modelling.
A possible scheduling for the signals is as follow, assuming the lights go from green to orange to red to green:
First scenario:

- S1 is red
- S2 is red
- S3 is green

Second scenario:

- S1 is green
- S2 is green
- S3 is red

In this scenario P2 is possible but the drivers need to be very careful when merging into the lanes blocked by S2.

Project Status

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
CTXtf2	OK	OK	0	0	0	-
CTXtf2_i	OK	OK	1	1	0	-
TrafficLights2	OK	OK	27	26	1	-

Image 11: Traffic controller 2 project status

The Traffic controller 2 project was simpler and as such there were fewer proof obligations and only one unproved.

Explanations

```
SETS
    LANES = {L1, L2, L3, L4, L5};
    SIGNALS = {S1, S2, S3}
CONSTANTS
    TRANSITIONS

PROPERTIES
    TRANSITIONS : SIGNALS <-> ( LANES * LANES ) &
    TRANSITIONS = {
        S1 |-> (L3 |-> L1),
        S2 |-> (L2 |-> L3),
        S3 |-> (L1 |-> L2),
        S3 |-> (L1 |-> L3)
    }
END
```

Image 12: CTXtf2 machine

The simplicity of the model made it possible to remove the *INTERSECT_LANES* constant from it.

```
VARIABLES
    rs, /* red signals */
    os, /* orange signals */
    gs, /* green signals */
    bs, /* blinking (orange signals) */
    ds // the lights are off
INVARIANT
    rs <: SIGNALS &
    os <: SIGNALS &
    gs <: SIGNALS &
    bs <: SIGNALS &
    ds <: SIGNALS &
    (bs = SIGNALS or bs ={}) & /* all blinking or none blinking */
    rs \/ os \/ gs \/ bs \/ ds = SIGNALS & /* coherency */
    rs /\ os /\ gs /\ bs /\ ds = {}  /* coherency */
    // No more than one same lane as destination
    & gs : {{},{S3},{S1,S2}} //determines that only these configurations are possible as green signals
INITIALISATION
    rs := {} ||
    os := {} ||
    gs := {} ||
    bs := {} ||
    ds := SIGNALS
```

Image 13: Variables, Invariant and Initialization of TrafficLights2 machine

In the same manner, the variables, invariant and initialization were simplified, the most notable change is the last line of the invariant that now contains the green lights scenarios identified during the proposed scheduling.

```
OPERATIONS
    turn_off =
    BEGIN
        rs := {} ||
        os := {} ||
        gs := {} ||
        bs := {} ||
        ds := SIGNALS
    END;

    reset_turn_on =
    PRE ds = SIGNALS
    THEN
        rs := {} ||
        os := {} ||
        gs := {} ||
        bs := SIGNALS ||
        ds := {}
    END;

    start_exploitation = /* moving from all blinking to all orange */
    PRE
        bs = SIGNALS
    THEN
        bs,os := {}, SIGNALS
    END;

    orange_to_red = /* moving from all orange to all red */
    PRE not (os={})
    THEN
        os,rs := {}, rs\/os
    END;
```

Image 13: turn_off, reset_turn_on, start_exploitation and orange_to_red operations from Trafficlights2

The operations presented on the Image 13 had no change made to them.

```
    red_to_green1 = // set S3 to green
    PRE
        gs = {} & {S3} <: rs & os = {} & (rs-{S3}) \/ os \/ gs \/ {S3} \/ bs \/ ds = SIGNALS
    THEN
5/5     gs := {S3} ||rs := rs-{S3}
    END;

    red_to_green2 = // set S1 and S2 to green
    PRE
        gs = {} & {S1,S2} <: rs & os = {} & (rs-{S1, S2}) \/ os \/ {S1, S2} \/ bs \/ ds = SIGNALS
    THEN
4/4     gs := {S1,S2} ||rs := rs-{S1,S2}
    END;

    green_to_orange = // Set green signals to orange
    PRE
        not(gs={})
    THEN
3/4     os := os \/ gs || gs := {}
    END
```

Image 14: red_to_green1, red_to_green2 and green_to_orange operations from Trafficlights2

The red_to_green1 and red_to_green2 had to be changed to the new possible members of the gs set, the red_to_green1 now sets S3 to green and the red_to_green2 sets S1 and S2 to green.

The missing proof in green_to_orange is $rs \setminus / (os \setminus / gs) \setminus / \{\} \setminus / bs \setminus / ds = SIGNALS$. The same missing proof from the traffic controller 1 problem and as such, can be proven in the same manner.

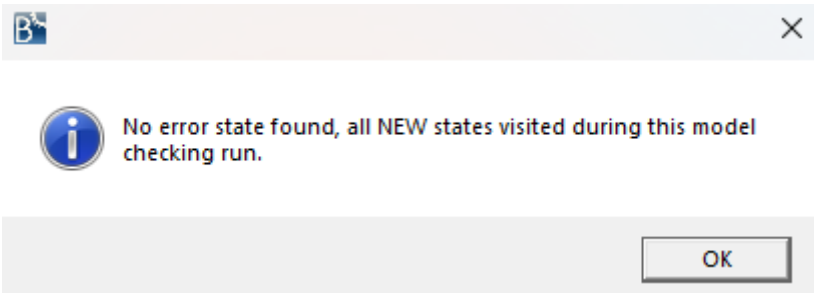

Image 15: Model check result of Trafficlights2

Using prob, the model check came with no error, as shown in image 15.

Supervisor

Project Status

Component	TypeChecked	POs Generated	Proof Obligations	Proved	Unproved	B0 Checked
g_operators	OK	OK	37	37	0	OK
g_standard_types	OK	OK	0	0	0	OK
g_types	OK	OK	3	3	0	OK
g_types_i	OK	OK	10	10	0	OK
inputs	OK	OK	0	0	0	OK
inputs_i	OK	OK	6	6	0	OK
io_constants	OK	OK	0	0	0	OK
lchip_configuration	OK	OK	0	0	0	OK
lchip_interface	OK	OK	1	1	0	OK
logic	OK	OK	0	0	0	OK
logic_i	OK	OK	231	175	56	OK
outputs	OK	OK	0	0	0	OK
outputs_i	OK	OK	4	4	0	OK
safety_variables	OK	OK	0	0	0	OK
user_component	OK	OK	0	0	0	OK
user_component_i	OK	OK	2	2	0	OK
user_configuration	OK	OK	0	0	0	OK
user_configuration_i	OK	OK	13	11	2	OK
user_ctx	OK	OK	0	0	0	OK
user_ctx_i	OK	OK	2	2	0	OK

Image 16: supervisor Project Status

The logic_i component was the one with the most Proof obligations and the one with the most unproved proofs as well.

Explanations

```
SEES
    g_types
VALUES
    MINCT = 5000; // 5 seconds
    MINRY = 51 //51 milliseconds, this exists because if the lights flicker too quickly it could kill the board
END
```

Image 17: user_ctx_i constant values

The MINCT constant corresponds to 5 seconds and will be used to represent the minimum allowed time for a light to change state without creating an alert.

```
CONCRETE_VARIABLES
    board_0_O1,
    board_0_O2,
    ctime, //Current time
    l1d, //This and the two following variables hold the last time that any of the lights were turned green or stopped being green
    l2d,
    l3d,
    l1s, //This and the two following variables hold the last known value for each input.
    l2s,
    l3s,
    relayTime //Time the checkAbnormalBehavior was last called
INVARIANT
    board_0_O1 : uint8_t &
    board_0_O2 : uint8_t &
    ctime : uint32_t &
    l1d : uint32_t &
    l2d : uint32_t &
    l3d : uint32_t &
    l1s : uint8_t &
    l2s : uint8_t &
    l3s : uint8_t &
    relayTime : uint32_t


INITIALISATION
    board_0_O1 := IO_OFF;
    board_0_O2 := IO_OFF;
    ctime := 0;
    l1d := 0;
    l2d := 0;
    l3d := 0;
    l1s := IO_OFF;
    l2s := IO_OFF;
    l3s := IO_OFF;
    relayTime := 51
```

Image 18: Concrete variables, invariant and initialisation from logic_i

Most of the variables were created to support the check in a change of state with the lights.

```
    LOCAL_OPERATIONS
        r1,r2 <-- supervisor(i1,i2,i3)=
        PRE
            r1 : uint8_t & r2: uint8_t & i1 : uint8_t & i2 : uint8_t & i3 : uint8_t & board_0_O1: uint8_t & board_0_O2:uint8_t & ctime :
    uint32_t & l1d : uint32_t & l2d : uint32_t & l3d : uint32_t & l1s : uint8_t & l2s : uint8_t & l3s : uint8_t & relayTime : uint32_t
        THEN
            r1 :: uint8_t||
            r2 :: uint8_t
        END;
        result <-- allLightsGreen(i1,i2,i3)=
        PRE
1/1         result : uint8_t & i1 : uint8_t & i2 : uint8_t & i3 : uint8_t & board_0_O1: uint8_t
        THEN
            IF(i1 = IO_ON) THEN
                IF(i2 = IO_ON) THEN
                    IF(i3 = IO_ON) THEN
                        result := IO_ON
                        ELSE result := board_0_O1
                    END
                    ELSE result := board_0_O1
                END
                ELSE result := board_0_O1
            END
        END;
        result <-- checkAbnormalBehavior(i1,i2,i3)=
        PRE
72/72           result : uint8_t & i1 : uint8_t & i2 : uint8_t & i3 : uint8_t & ctime : uint32_t & l1d : uint32_t & l2d : uint32_t & l3d : uint32_t
    & l1s : uint8_t & l2s : uint8_t & l3s : uint8_t & relayTime : uint32_t
        THEN
            result :: uint8_t||
10/18       l1d :: uint32_t||
11/19       l2d :: uint32_t ||
13/21       l3d :: uint32_t ||
 9/17       l1s :: uint8_t ||
 9/17       l2s :: uint8_t ||
 9/17       l3s :: uint8_t ||
27/35       relayTime :: uint32_t
        END
```

Image 19: Local operations from logic_i

```
        user_logic =
        BEGIN
            ctime <-- get_ms_tick;
            VAR i1_, i2_, i3_ IN
                i1_ :( i1_ : uint8_t );
                i2_ :( i2_ : uint8_t );
                i3_ :( i3_ : uint8_t );

                i1_ <-- get_board_0_I1;
                i2_ <-- get_board_0_I2;
                i3_ <-- get_board_0_I3;
                board_0_O1,board_0_O2 <-- supervisor(i1_, i2_, i3_)
            END
        END;

        r1,r2 <-- supervisor(i1,i2,i3)=
        BEGIN
  1/1       r1 <-- allLightsGreen(i1,i2,i3);
            VAR dslc IN //dslc = difference since last call to checkAbnormalBehavior
                dslc : (dslc : uint32_t);
  8/8           dslc := sub_uint32(ctime,relayTime);
                IF(board_0_O2 = IO_ON) THEN
                    IF(dslc < 1000)THEN
                        r2 := board_0_O2
                    ELSE
36/64                   r2 <-- checkAbnormalBehavior(i1,i2,i3)
                    END
                ELSE
36/64               r2 <-- checkAbnormalBehavior(i1,i2,i3)
                END
            END
        END;
```

Image 20: user_logic and supervisor operations

The user_logic operation gets the input from the pins on the board and sets the output of supervisor to the two variables that correspond to the output on the board.

Supervisor is responsible for calling allLightsGreen and checkAbnormalBehavior. Important to note that the logic for not killing the relays on the board is within supervisor.

```
result <-- allLightsGreen(i1,i2,i3)=//returns IO_ON if all lights are on or if it has returned IO_ON previously
BEGIN
    result := board_0_O1;
    IF(i1 = IO_ON) THEN
        IF(i2 = IO_ON) THEN
            IF(i3 = IO_ON) THEN
                result := IO_ON
            END
        END
    END
END;
```

Image 21: allLightsGreen operation

The allLightsGreen operation will turn the output on if it was already on or if all the inputs are on at the same time.

```
result <-- checkAbnormalBehavior(i1,i2,i3)=//returns IO_ON if an abnormal behavior is detected
BEGIN
    result := IO_OFF;
    VAR d1, d2, d3 IN //deltas between the ctime and l1d,l2d,l3d
        d1 : (d1:uint32_t);
        d2 : (d2:uint32_t);
        d3 : (d3:uint32_t);
2/2     d1 := sub_uint32(ctime,l1d);
1/1     d2 := sub_uint32(ctime,l2d);
1/1     d3 := sub_uint32(ctime,l3d);
        IF (i1 = l1s)THEN
            skip
        ELSE
            l1s := i1;
1/1         l1d := ctime;
            IF(d1<MINCT) THEN
                result := IO_ON;
4/4             relayTime := ctime
            END
        END;
        IF(i2 = l2s) THEN
            skip
        ELSE
            l2s := i2;
2/2         l2d := ctime;
            IF(d2<MINCT) THEN
                result := IO_ON;
6/6             relayTime := ctime
            END
        END;
        IF(i3 = l3s) THEN
            skip
        ELSE
            l3s := i3;
4/4         l3d := ctime;
            IF(d3<MINCT) THEN
                result := IO_ON;
9/9             relayTime := ctime
            END
        END
    END
END;
```

Image 22: checkAbnormalBehavior operation

The checkAbnormalBehavior will verify the time since the state of an input was last changed and if that time is less than 5 seconds it will turn the output on. It does that for the 3 inputs on the board.

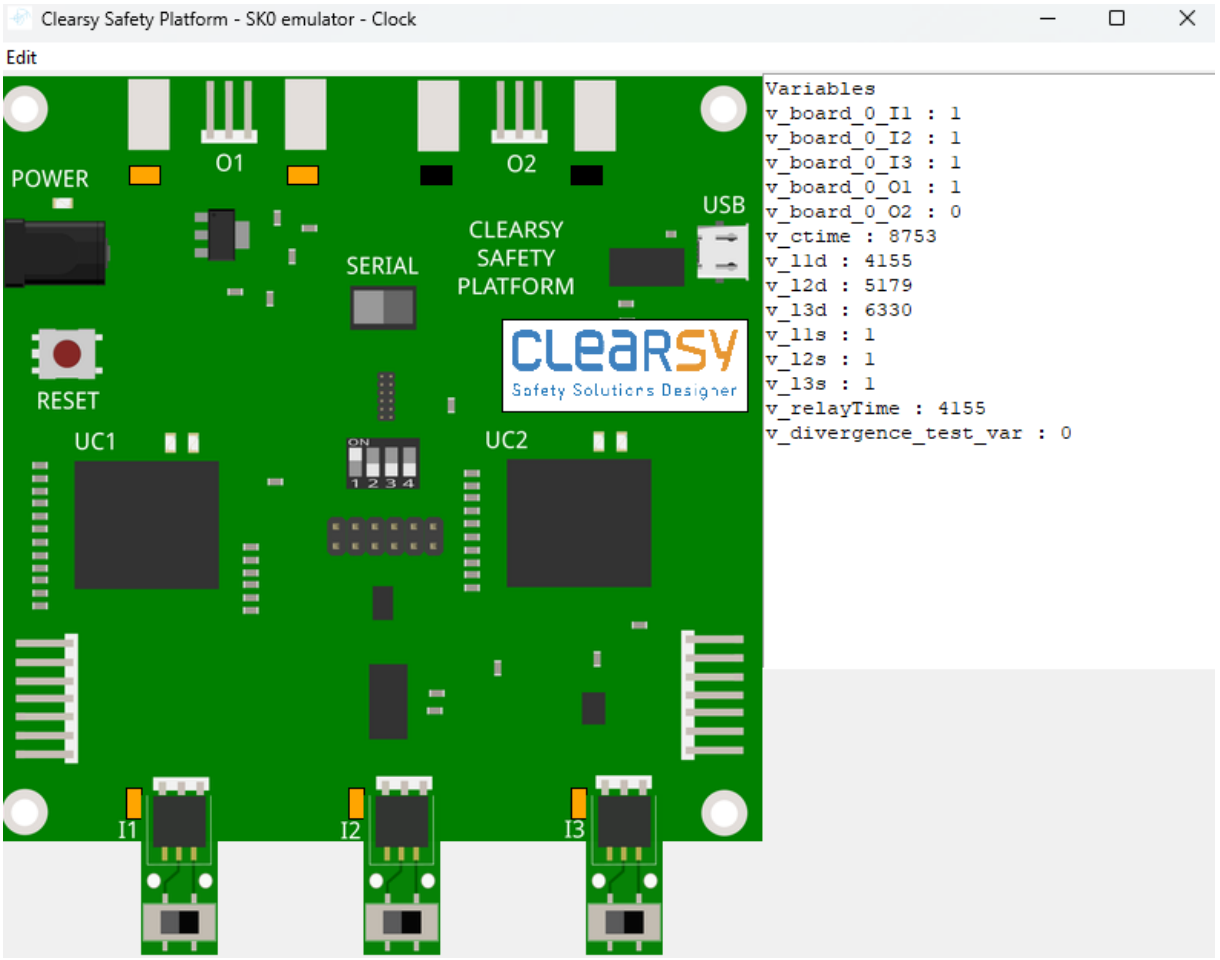

Image 23: testing the supervisor with the simulator

The simulator presented problems with my computer, most notably the v_ctime variable was not updating correctly, the problem was not found but it was not present the next day.

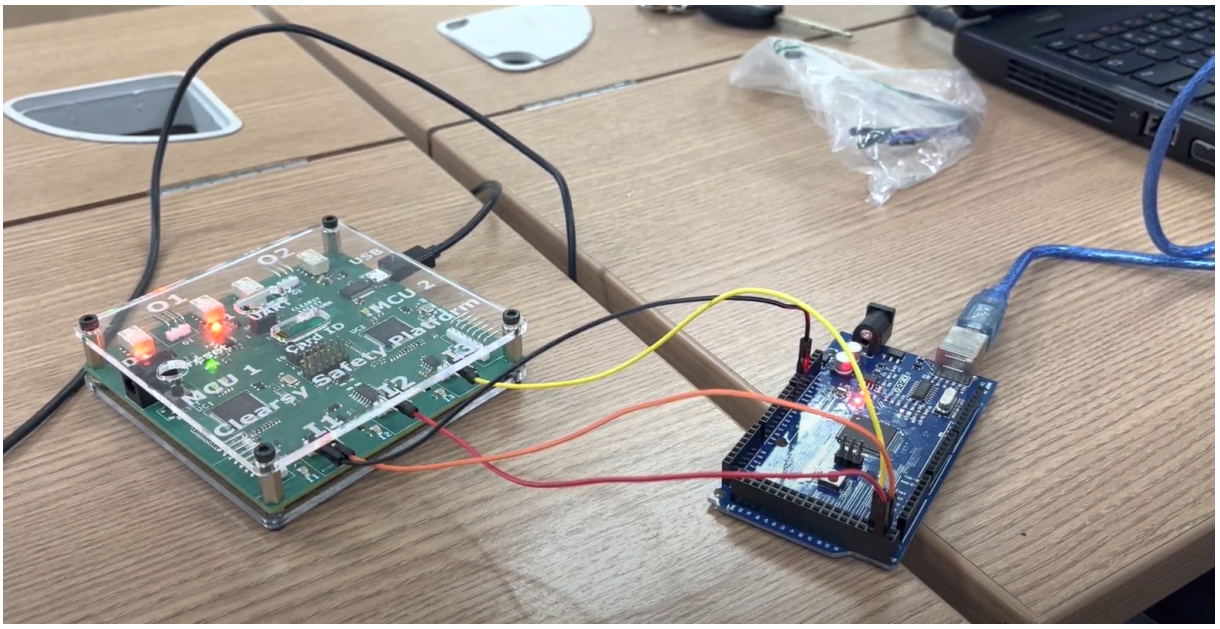

Image 24: testing with the board

The program was tested on two boards and was giving different results from the expected. Later, when using a physical button the problems disappeared, we couldn’t find the reason for that but our theory was insufficient power coming from the arduino.