7.3 Case Study - FV of a traffic light controller

7.3.1 Model

This design represents a simple traffic light controller for a North-South and East-West intersection. The North-South is the main road, and is given the GREEN light unless a sensor on the East-West Street is activated. When that occurs, and the North-South light was GREEN for sufficient time, then the light will change to give way to the East-West traffic. The design also takes into account emergency vehicles that can activate an emergency sensor. When the emergency sensor is activated, then the North-South and East-West lights will turn RED, and will stay RED for a minimum period of 3 cycles. Figure 7.3.1-1 is a view of the intersection. The model has the following interfaces:

```
typedef enum { OFF,     // power off    // /ch7/7.3/trafficlightok.sv
               RED,     // red state
               YELLOW,  // yellow state
               GREEN,   // green state
               PRE_GREEN}  // state before green
               lights_t;

module trafficlight
 (
  output  lights_t ns_light,   // North/South light status, Main road
  output  lights_t ew_light,   // East/West light status
  input            ew_sensor,  // East/West sensor for new car
  input            emgcy_sensor, // emergency sensor
  input            reset_n,    // synchronous reset
  input            clk         // master clock
  );
   timeunit 1ns;    timeprecision 100ps;
```

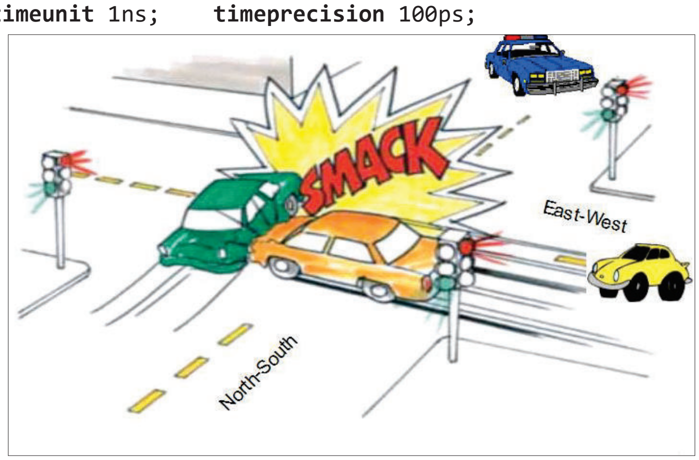


Figure 7.4.1-1 North-South and East-West intersection.

Since the traffic lights are timed, two timers are used in this first implementation:

```
logic [1:0]   ns_green_timer; // timer for NS light in Green
logic [1:0]   ew_green_timer; // timer for EW light in GREEN
```

A high-level view of the traffic light controller is shown in Figure 7.3.1-2. The architecture includes two FSMs and the timers. The operation of the machine is very simple:

1. North-South remains GREEN unless one of the East-West sensors is activated.
2. If the North-South light is RED and the North-South GREEN timer is 3, then the light will switch to GREEN.
3. If the North-South light is YELLOW, it will switch to RED.

4. If the North-South light is GREEN, and the emergency sensor is activated, the light will switch to YELLOW. Also, if the North-South timer is 3, and the East-West sensor is activated, the North-South light will switch to YELLOW.
5. The North-South GREEN timer is reset to zero at reset, or when the North-South light is YELLOW. Otherwise, it increments at every clock until it reaches the maximum count of 3.
6. The East-West light switches from RED to a PRE_GREEN state (to allow the North-South light to go to YELLOW) if the North-South timer is 3 and the East-West sensor is activated.
7. If the East-West is PRE_GREEN, it will switch to GREEN at the next clock.
8. If the East-West light is YELLOW, it will switch to RED.
9. If the East-West is GREEN and either the emergency sensor is activated or the East-West timer reaches a count of 3 then it switches to YELLOW.
10. The East-West GREEN timer is reset to zero at reset or when the East-West light is YELLOW. Otherwise, it increments at every clock until it reaches the maximum count of 3.

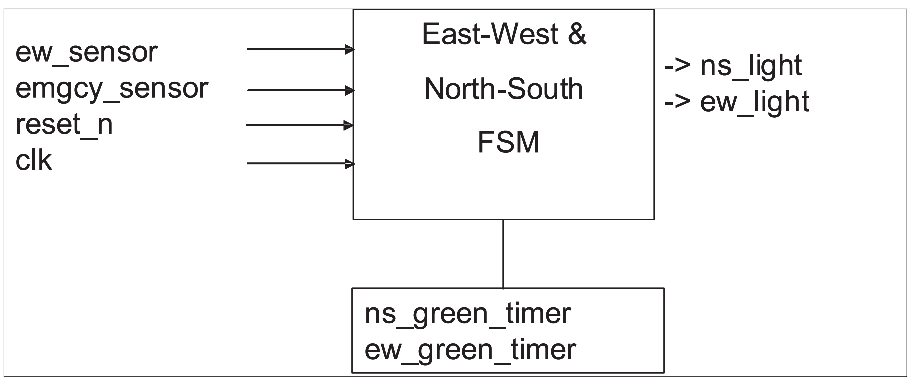


Figure 7.4.1-2 High-Level View of Traffic Light Controller

Advance Notification:

On the surface, this architecture looks acceptable. However, it is flawed, as there are illegal behaviors, as demonstrated by formal verification tools. For example, it is possible for both the North-South light and the East-West light to be GREEN in the same cycle.

Even though the implementation of this synthesizable model is flawed, it is used in this book to demonstrate the value and power of ABV with SystemVerilog Assertions in formal verification. Later on in this chapter, a good traffic light controller is modeled and verified.

7.3.2 SystemVerilog Assertions for traffic light controller

Based upon the requirements, several assertions can be expressed prior to writing the RTL. Below (in file *tlight/tlight_props1.sv*[60]) are some example assertions written for this design. Note that in order to later bind this checker module to the actual design, all its ports are declared as inputs. Its ports are all inputs and outputs of the actual design, plus some internal states needed to model the requirements. The internal states are defined in compilation unit scope as enumerations so they can be easily shared in the RTL. In this case we choose to include the internal timer *ns_green_timer* in the port list.

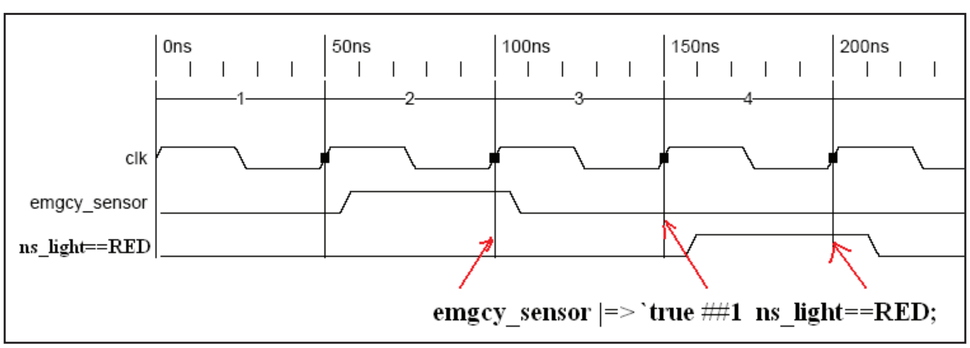
60 Please refer to file model ***tlight/trafficlight.sv*** for the complete RTL code.

```
`define true 1
`ifndef MULTIPLE_FILE_COMPILE
  typedef enum {OFF, RED, YELLOW, GREEN, PRE_GREEN} lights_t;   `endif
module tlight_props  ( // ch7/7.3/tlight_props2.sv
   input  lights_t ns_light,  // North/South light status, Main road
   input  lights_t ew_light,  // East/West light status
   input          ew_sensor, // East/West sensor for new car
   input          emgcy_sensor, // emergency sensor
   input          reset_n,   // synchronous reset
   input          clk,       // master clock
   input [1:0]  ns_green_timer
   );
  parameter FAIL   = 1'b0;

  // **************************************************
  // Safety property
  property Never_NS_EW_ALL_GREEN;
      disable iff (!reset_n)
     not (ns_light==GREEN && ew_light==GREEN);
  endproperty : Never_NS_EW_ALL_GREEN
  Never_NS_EW_ALL_GREEN_1 : assert property(@ (posedge clk) Never_NS_EW_ALL_GREEN);
```

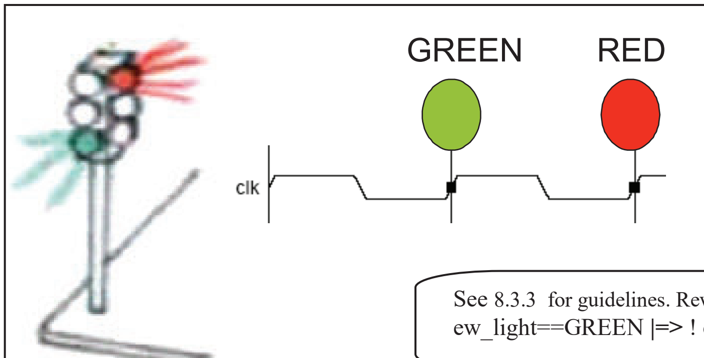
> The following is preferred (see 8.3.3)
> (ns_light==GREEN |-> !ew_light==GREEN)

```
  // **************************************************
  // State of lights at reset
  property nsLightAtReset;
     // disable iff (!reset_n)  // <-- this causes the assertion to always be vacuous
     reset_n==1'b0 |=> ns_light==OFF;
  endproperty : nsLightAtReset
  nsLightAtReset_1 : assert property(@ (posedge clk) nsLightAtReset);
  //
  property ewLightAtReset;
     // disable iff (!reset_n) // <-- this causes the assertion to always be vacuous
    reset_n==1'b0 |=> ew_light==OFF;  // RED???
   endproperty : ewLightAtReset
  ewLightAtReset_1 : assert property(@ (posedge clk) ewLightAtReset);
  // **************************************************
 // State of lights during emergency
  // Lights switch from GREEN to YELLOW to RED
  property NsLightsWhenEmergency;
     disable iff (!reset_n)
    emgcy_sensor |=> `true[*2] ##1  ns_light==RED;
   endproperty : NsLightsWhenEmergency
  NsLightsWhenEmergency_1 : assert property(@ (posedge clk) NsLightsWhenEmergency);
```




```
  property EwLightsWhenEmergency;
     disable iff (!reset_n)
     emgcy_sensor |=> `true[*2] ##1 ew_light==RED;
  endproperty : EwLightsWhenEmergency
  EwLightsWhenEmergency_1 : assert property(@ (posedge clk) EwLightsWhenEmergency);
```

```
// Safety, GREEN to RED is illegal.  Need YELLOW
  property NsNeverFromGreenToRed;
     disable iff (!reset_n)
     not(ns_light==GREEN ##1 ns_light==RED);
  endproperty : NsNeverFromGreenToRed
  NsNeverFromGreenToRed_1 : assert property(@ (posedge clk) NsNeverFromGreenToRed);
```

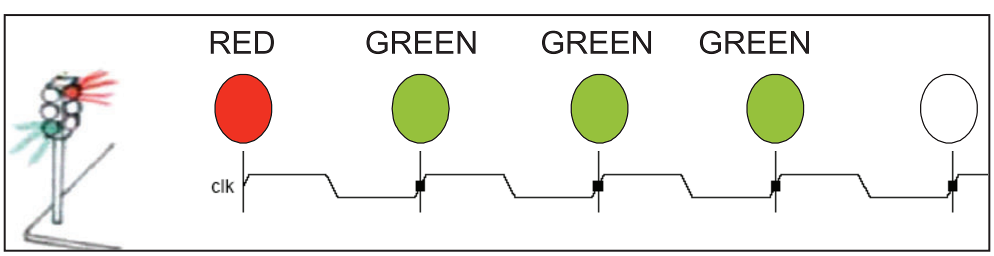


See 8.3.3 for guidelines. Rewrite as
ew_light==GREEN |=> ! ew_light==RED

```
property EwNeverFromGreenToRed;
    disable iff (!reset_n)
     not(ew_light==GREEN ##1 ew_light==RED);
endproperty : EwNeverFromGreenToRed
 EwNeverFromGreenToRed_1 : assert property(@ (posedge clk) EwNeverFromGreenToRed);
// ****************************************************
 // The NorthSouth light is the main street light.
// If ns is green and no emergency or ew sensor, then next cycle is also GREEN
property NsGreenNext;
   (ns_light==GREEN) && ($past(emgcy_sensor)==1'b0 && reset_n==1'b1)
                  |=> ns_light==GREEN;
endproperty : NsGreenNext
NsGreenNext_1:  assert property (@ (posedge clk) NsGreenNext);
```

See 8.3.3 for guidelines. Rewrite as
(ew_light==GREEN |-> !ns_light==YELLOW)
and
(ns_light==GREEN |-> !ew_light==YELLOW)

```
   // GREEN-YELLOW at the same time
  property NeverGreenYellow;
     not ((ew_light==GREEN && ns_light==YELLOW) ||
          (ns_light==GREEN && ew_light==YELLOW));
  endproperty : NeverGreenYellow
  NeverGreenYellow_1: assert property (@ (posedge clk) NeverGreenYellow);
  // ******************************************************
// The NorthSouth light is the main street light.
// It must remain GREEN for ns_green_timer == 3 before it can switch.
// Timer ns_green_timer will count to 3, and remain at 3 until light changes.
   property NsGreenForMin3Ccyles;
    @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
      $rose(ns_light==GREEN) && !$past(emgcy_sensor) |=>
             ns_light==GREEN[*2]; // abort emgcy_sensor);
   endproperty : NsGreenForMin3Ccyles
  NsGreenForMin3Ccyles_1 : assert property (NsGreenForMin3Ccyles);
```




```
  // ***************************************************
```

// East-West North-South Lights with East-West sensor
// If ew_sensor is activated (new car), then light will switch for the ew_light
// when minimum time for ns_light is satisfied. ew_green_timer will count to 3,
// at which time, the ns_green_timer will regain control of GREEN.

```
   property EwNewSensorActivation;
    @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
     ( (ew_sensor==1'b1) && $rose(ns_green_timer==2'b11 )) &&
                       !$past(emgcy_sensor) && ns_light!= RED
                              |=>   ns_light==YELLOW ##1 ew_light==GREEN;
   endproperty : EwNewSensorActivation
   EwNewSensorActivation_1 : assert property (EwNewSensorActivation);
// End of new properties 09/10/09
 endmodule : tlight_props

bind trafficlight tlight_props tlight_props1 (.*);
```

7.3.3 Verification

The above model was verified with *OneSpin 360 MV*, and it revealed several failures in the design, as shown in Figure 7.4.3-1.[61] In that figure, "fail (9)" means that the tool detected a violation of the property starting 9 cycles after reset.

As an example of debugging a failing property, Figure 7.4.3-2 shows the debugging view for the first property "*Never_NS_EW_ALL_GREEN*":

> The left part of the debugging window shows an interactive view of the property, with the failing parts highlighted in red (see *ch6/tlight/1_some_fail.png* file for a color view of a larger image). The waveform shows that indeed in cycle 0, both the EW and the NS lights are green[62]. Further, it indicates that some steps earlier, the emergency sensor, and the EW sensor were activated. To explore this situation, the time-point "-2" has been selected (indicated by the yellow vertical bar). The active source code annotation in the upper right corner shows the critical part of the DUV, with the active source lines marked in red: the root cause is the conditional transition from RED to the PRE_GREEN state, the condition
>
> ```
> if (ns_green_timer==3'b11 && ew_sensor==1'b1)
> ```
>
> being satisfied although in fact the NS light is not green, but being switched to green in the same step.

The sequence of events leading to this situation is fairly complex, and would have required extensive simulation with pseudo-random patterns to arrive at the failed situation. The bug, together with the other bugs detected by the formal tool, led to a thorough redesign of the controller, as discussed in the next section.

---

[61] *OneSpin*'s *360 MV* is a family of formal verification tools ranging from fully automatic RTL checks for large designs all the way to *OneSpin*'s patented gap-free verification. http://www.onespin-solutions.com/

[62] The cycles are numbered such that the property always starts at cycle 0, while the reset cycle is at some negative number, not shown in the figure.

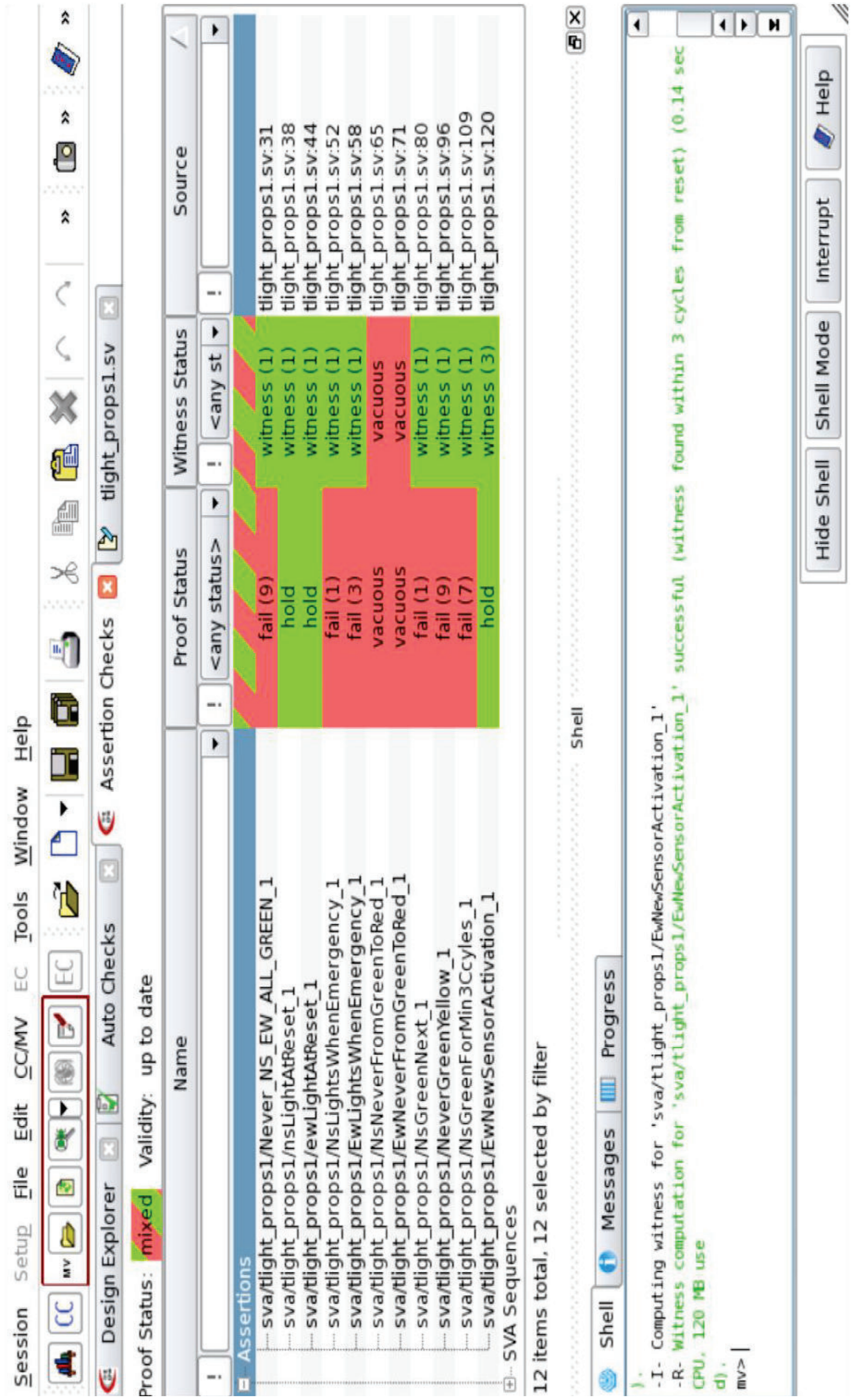


Figure 7.4.3-1 Summary of Formal Verification Results (ch7/tlight/1_some_fail.png)
The screenshots in this section were made with *OneSpin 360 MV*,
courtesy of *OneSpin Solutions*

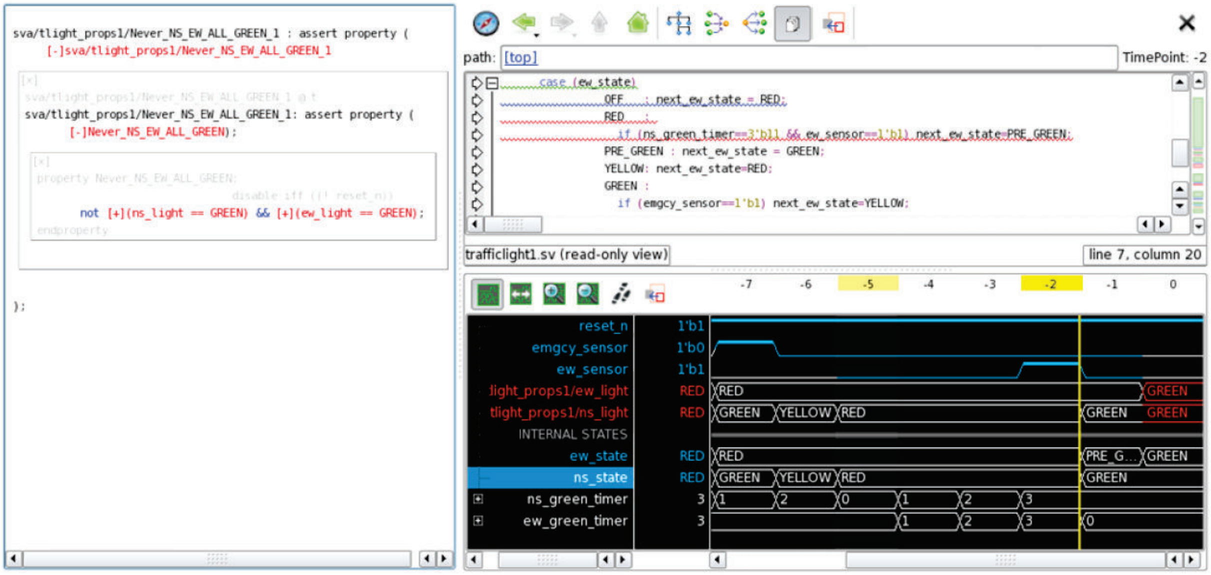


Figure 7.4.3-2 Counterexample: all lights are green, *ch7/tlight/2_debug_allgreen.png*

7.3.4 Good Traffic Light Controller

The traffic light controller demonstrated in the previously section represents a model where the East-West and North-South FSMs are loosely tied, and there lies the source of errors such as both East-West and North-South lights turning GREEN in the same cycle. Of course, that erroneous condition only happens under specific sequences, as demonstrated by formal verification.

To resolve this issue, a centralized architecture will be used. The new design relies on the North-South FSM being the master controller. The East-West slave FSM makes an *ew_green_request* whenever it wants access to the light. That request is granted with the *ew_green_grant* handshake. It is provided by the North-South FSM when the North-South goes YELLOW. In addition, to maintain this centralized control, the North-South FSM will inform the East-West FSM to go RED with the *ew_to_red_cmd* command. That command is a function of the value of the emergency sensor, the value of the East-West sensor, and the length of time that East-West light stayed GREEN. A centralized *North-South* GREEN timer, instead of two independent timers, controls that time. Figure 7.4.4-1 represents a high-level view of this architecture.

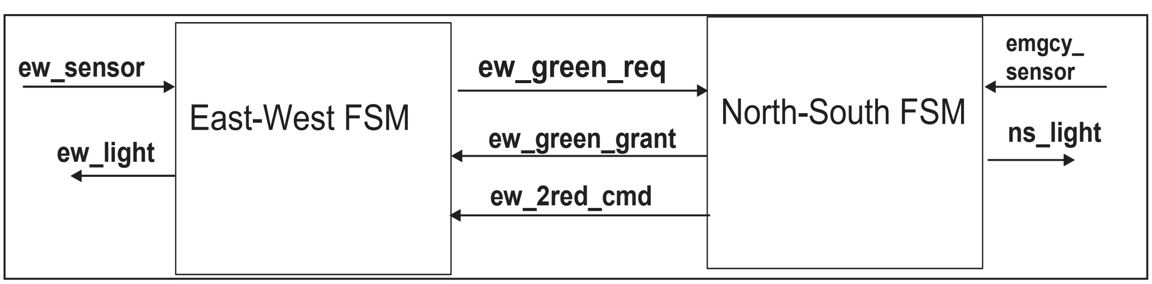


Figure 7.4.4-1 Good traffic light controller architecture

The last property "*EwNewSensorActivation*" fails now, although this held on the first (faulty) design! This property describes how the EW light is switched to green after activation of the EW sensor.

```
property EwNewSensorActivation;
   @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
     ew_sensor==1'b1 && $rose(ns_green_timer==2'b11)
      && !$past(emgcy_sensor) && ns_light!= RED |->
         ##[1:3] ns_light==YELLOW ##1 ew_light==GREEN;
endproperty : EwNewSensorActivation
EwNewSensorActivation_1 : assert property (EwNewSensorActivation);
```

Analyzing the counterexample in figure 7.4.4-2 reveals that the EW light is indeed switched to green, but only *four cycles after* the sensor activation, rather than two cycles as predicted by the property. This is caused by the momentum of the FSMs. Figure 7.4.4-2 indicated that there is a witness for this property, i.e. it is possible to meet the predicted timing.

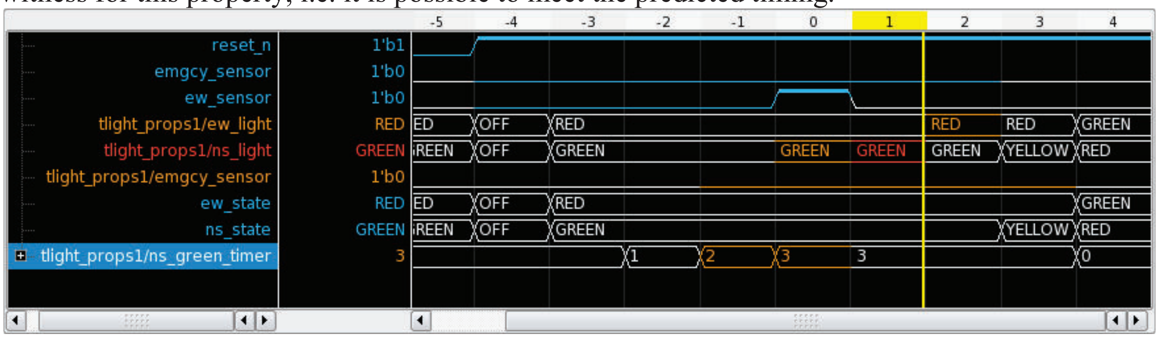


Figure 7.4.4-2 Debugging property EwNeverSensorActivation_1,
ch7/tlight/4_debugEWSensor.png

Therefore the property needs to be modified to allow for a *more liberal* timing:

```
 property EwNewSensorActivation;
   @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
     ew_sensor==1'b1 && $rose(ns_green_timer==2'b11)
      && !$past(emgcy_sensor) && ns_light!= RED |->
         ##[1:3] ns_light==YELLOW ##1 ew_light==GREEN;
endproperty : EwNewSensorActivation
EwNewSensorActivation_1 : assert property (EwNewSensorActivation);
```

This property holds. However, inspecting the property again, we find that its antecedent describes a rather special situation, namely that the *ns_green_timer* expires in the same instant when the *ew_sen*sor is activated. Also the additional assumption *ns_light != RED* raises the question whether we need to examine further cases before we can claim that this feature is truly verified. A more powerful way to write the "*EwNewSensorActivation*" is shown below, by introducing a formal argument for the delay between sensor activation and switching to green:

```
property EwNewSensorActivation_w(int del);
   @ (posedge clk) disable iff (!reset_n || emgcy_sensor)
  ew_sensor |->  ##[0:del] ew_light==GREEN;
 endproperty : EwNewSensorActivation_w
```

This property can be called with different delay values to explore the actual worst case delay:

```
EwNewSensorActivation3: assert property (EwNewSensorActivation_w(3));
…
EwNewSensorActivation11 : assert property (EwNewSensorActivation_w(11));
EwNewSensorActivation12 : assert property (EwNewSensorActivation_w(12));
```

It turns out that even a delay of 10 can occur, but the property does hold with delay 11 or more, as is shown in the final result in figure 7.4.4-3. The final properties are in file *ch7*/tlight/tlight_props2.sv.

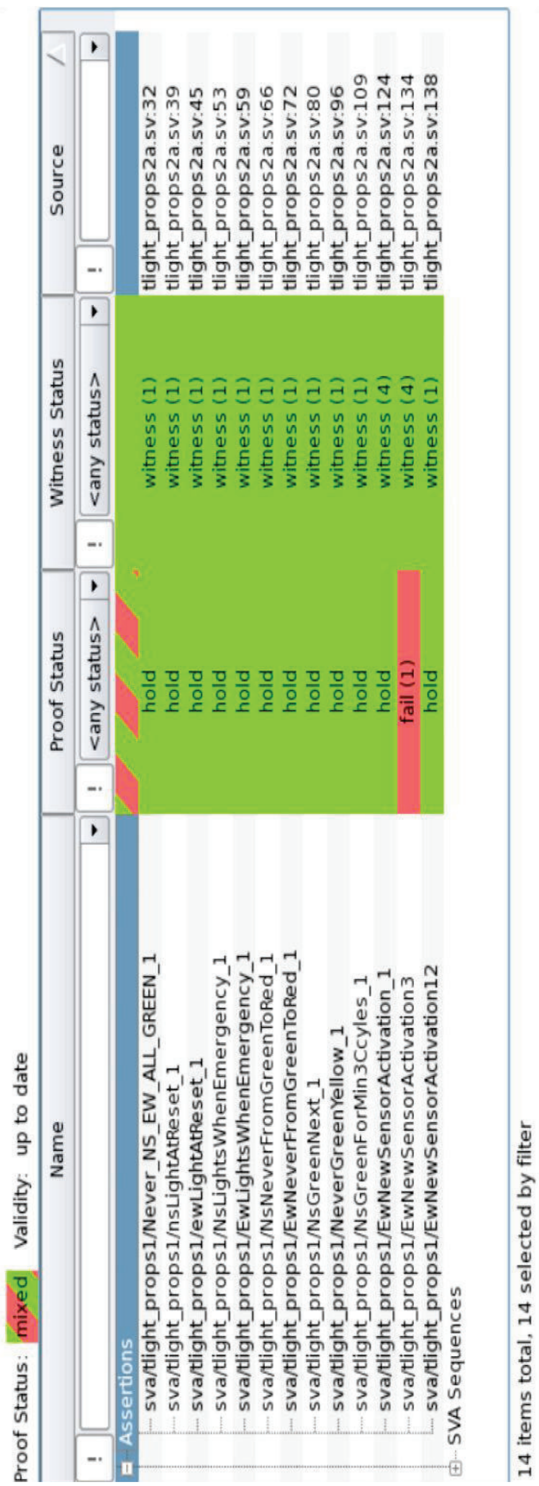


Figure 7.4.4-3 Final result for traffic light controller, ***ch7/tlight/5_finalresult.png***

Reflections:
Some reflections on the derivation of this model are in order:

1. It took about 10 iterations to arrive at a working model.

2. Each iteration went fairly fast because formal verification was used without a testbench. As the design matured, and more analysis was done, it took longer between iterations (early mortality effect). This methodology is analogous to the edit-compilation/synthesis process used in RTL design and synthesis to eliminate the gross errors.

3. Linting of the model significantly helped in debugging the model prior to performing formal verification.

4. Formal verification quickly arrived at errors in the design, along with counterexamples that demonstrated the problems. Corrections of these errors were quickly verified with another run of the formal verification tool.

5. Formal verification also demonstrated that some properties needed modifications since they were not properly expressed. These incorrectly stated properties challenged the author of the properties in his understanding of the requirements. That cycle caused a better understanding of the operation of the machine and the requirements.

A debugger closely integrated with the verification tool helped greatly in understanding the root cause of the failures.

8 SYSTEMVERILOG ASSERTIONS GUIDELINES

This chapter provides a rich set of guidelines in using SystemVerilog Assertions. These guidelines emerged from experience with usage of Assertion-Based Verification, vendor's recommendations, code reviews, and SystemVerilog 1800 documentation. Those guidelines are supplemental to those addressed throughout this book.

It is recommended that when Assertion-Based Verification is used in a project, all designers and verification engineers involved in that project use that methodology. A general guideline in the use of assertions is to ensure a common "feel and look" throughout the design. This includes among other things, naming convention, directory structure, error reporting, initialization, synchronization or properties, use of clocks and default clock, defining resets in properties, the declaration of properties and tasks in packages, interfaces, and/or modules. A consistency in the verification environment will speed up the design and debug process and significantly improves maintainability. Sporadic and partial usage of assertions in a project defuses the benefits of ABV.