

Spider for a Traffic Light

DR. GERARD ALLWEIN

CHRISTOPHER BELMONTE

*Center for High Assurance Computer Systems Branch
Information Technology Division*

June 23, 2022

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 23-06-2022		2. REPORT TYPE NRL Memorandum Report		3. DATES COVERED (From - To) 1 Oct 2021 – 30 Sept 2022	
4. TITLE AND SUBTITLE Spider for a Traffic Light				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER 062235N	
6. AUTHOR(S) Dr. Gerard Allwein and Christopher Belmonte				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER 6C59	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Research Laboratory 4555 Overlook Avenue, SW Washington, DC 20375-5320				8. PERFORMING ORGANIZATION REPORT NUMBER NRL/5540/MR--2022/5	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 875 N. Randolph Street Arlington VA 22217-1995				10. SPONSOR / MONITOR'S ACRONYM(S) ONR	
				11. SPONSOR / MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This is a report on a spider for a traffic light application. A spider is a realtime monitor of field programmable gate array (FPGA) or complex programmable logic device (CPLD) code written in VHDL. Each spider is associated with specific logic statements expressing conditions in the code being monitored. Each spider also contains mechanisms for mitigating the effects of an exploit, either malicious, due to an error in design, or due to a hardware fault. The spider for this example was hand compiled from logic statements into VHDL code that is combined with the traffic light code before vendor tools compile everything into an internal representation. Spiders can be written in any language provided there is a translator into a language vendor supplied tools support. Eventually, spiders will be automatically compiled from logic statements and mitigation code to produce either VHDL code or ReWire code. ReWire has its own compiler that produces either VHDL or Verilog code, which would then subsequently be included in the application.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT U	18. NUMBER OF PAGES 25	19a. NAME OF RESPONSIBLE PERSON Dr. Gerard Allwein
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (include area code) (202) 404-3748

This page intentionally left blank.

CONTENTS

EXECUTIVE SUMMARY	E-1
1. INTRODUCTION	1
2. TRAFFIC LIGHT.....	2
2.1 Code Structure	4
2.2 Controller.....	4
2.3 Monitoring Conditions	7
2.4 Distributed Structure.....	10
3. CONCLUSIONS	10
4. APPENDIX	11
4.1 Wrapper Code	11
4.2 Traffic Light Controller	14
4.3 Spider.....	17
REFERENCES	21

This page intentionally left blank

EXECUTIVE SUMMARY

This is a report on a spider for a traffic light application. A spider is a realtime monitor of field programmable gate array (FPGA) or complex programmable device (CPLD) an application. The application and the spider are both written in VHDL. Generically, a spider is associated with specific logic statements expressing high assurance conditions in the application being monitored. The spider also contained mechanisms for mitigating the effects of an exploit, either malicious, due to an error in design, or due to a hardware fault. The spider for the traffic light controller was hand compiled from logic statements into VHDL code and combined with the traffic light code. The combination was then compiled by vendor tools into an internal representation that is downloaded into the FPGA. Eventually, spiders will be automatically compiled to produce either VHDL code or ReWire code. ReWire has its own compiler that produces either VHDL or Verilog code, which would then subsequently be included in the application. The NRL Memo Report “Spiders for FPGA Applications” is a prelude to this current report and covers theoretical ground.

This page intentionally left blank

Spider for a Traffic Light

1. INTRODUCTION

This is the second report for the NRL Base Program 6.2 (Work Unit 55 T012) Realtime Monitors (Spiders) for FPGA Applications. The first report, “Spiders for FPGA Applications”, detailed the logical prelude necessary to understand this report. That report also covered spider features not used in the spider for traffic light controller.

Intuitively, the notion is that spider sits above an application with its legs embedded in the application. The legs are signal lines and the body is the center for recognition of error conditions and mitigation of those conditions. The FPGA application will be called the *plant*, a term lifted from realtime control of industrial machines. We also use the term *spider* as verb, i.e., to *spider an application* is to have a realtime monitor for that application.

The intuitive picture is

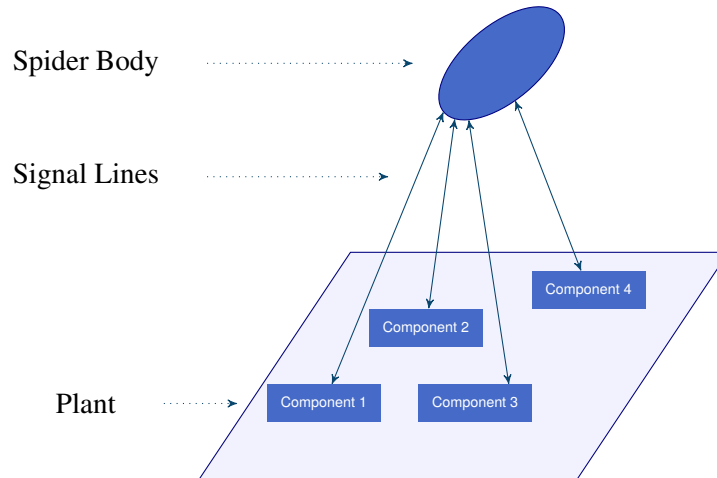


Figure 1.1: Intuitive Diagram of Spider and its Plant

The code itself is not structured according to the diagram. The structure of the system as seen from the VHDL code is usually different. The signal lines between the spider and the plant are not drawn. The hierarchical structure is showing a possible code architecture, not the actual connections (as in the previous diagram).

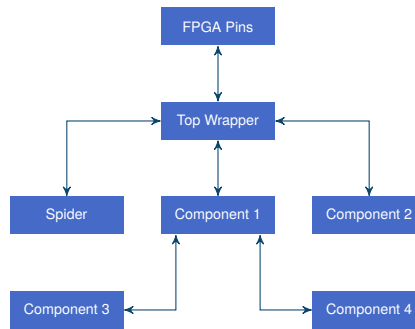


Figure 1.2: Code Structure of Spider and its Plant

The reason for this structure is that we wish to hide internal signal connections from the outside world. This structure also makes it relatively easy to place the spider and the plant into a larger system. There may be several spiders for any one plant, and spiders can spider other spiders. It is also possible to have a spider observing two different plants if, say, it is required to watch over interaction between the two plants.

2. TRAFFIC LIGHT

The traffic light is a relatively simple application that allowed us to see what is required for building a spider. The traffic light controller works on a four-way intersection controlling the usual green-yellow-red lights for all directions. There is a timer so that the lights do not stay in one configuration over a time limit.

The specification of the traffic light controller is as follows. Consider the traffic intersection below:

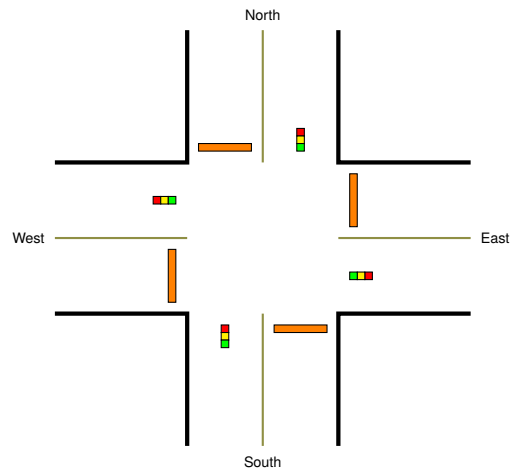


Figure 2.1: Traffic Flow Diagram

Requirements:

1. Produce source code exhibiting the following behavior:
 - a. The North/South lights act as a pair and the East/West lights act as a pair, the lights will be in synchronization at all times. There is no turn signal.
 - b. The light is a standard traffic and should follow the typical sequencing of green, yellow, red light, repeat.
 - c. By default each pair shall be active (green) for 10 seconds, after which it will begin the transition through yellow to inactive (red) before the other light pair becomes active.
 - d. If a car shows up at the red light pair, and there are no cars present at the green light pair, then the green light shall only be active for a total of 5 seconds before switching; i.e., a car arriving at the red light before the green light pair was active for 5 seconds would cause the green light pair to change at 5 seconds, a car arriving at the red light after the green light pair was active for 5 seconds would immediately cause the green light pair to switch.
2. Provide a test bench that verifies the above behavior, and includes at a minimum the following cases:
 - a. The default behavior of the light.
 - b. A car arrives at the red light and a car is already present at the green light.
 - c. A car arrives at the red light before the 5 second mark and no car is present at the green light.
 - d. A car arrives at the red light after the 5 second mark and no car is present at the green light.

The spider for this controller will only attempt to watch over the sequencing of the lights with minimal attention paid to time intervals. The reason for this is that not much would have been learned by the spider being more intrusive. Also, there is a trade off in how complicated the spider can be with respect to the number of conditions being spidered in the controller. The code for both the spider and the controller is compiled together by vendor tools.

The code is recorded in the Appendix to this report. It is divided into the the top wrapper, the traffic light controller, and the spider. The code is written in VHDL and would require you understand that language in order to make sense of the code.

The spider has two general conditions that it watches for:

C1: detecting invalid outputs, e.g., both sets of lights being green at the same time,

C2: monitoring FSM timed transitions, e.g., tracking the max time for each state and flagging deviations, etc.

Note that in C2, minimum times spent in each traffic light color are not monitored. A more intrusive spider would check for this.

Mitigation should include upon recognition of an exploit:

M1: Flashing red lights in all (some) directions.

M2: Forcing the FSM controller into a start state.

2.1 Code Structure

The intuitive diagram of the spider and the light controller is

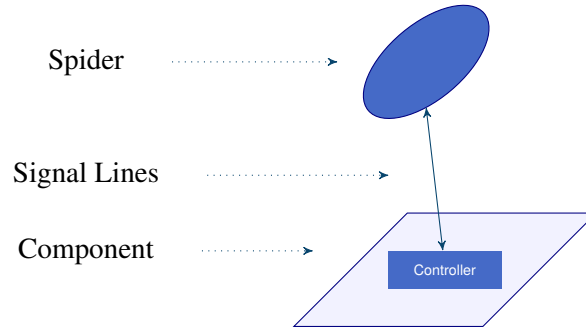


Figure 2.2: Traffic Light Spider + Controller

The VHDL code has three files, `top_wrap`, `top_modded`, and `light_spider_modded`, which, in this instance, correspond to three components, called *entities* in VHDL. VHDL allows for entities to contain other entities, i.e., components containing other components but we do not need such extra structure here. From now on, we will revert to using the term *component*, and the files will be called `wrapper`, `controller`, and `spider`. We will also use these names for the components whose names are eponymous with the files containing them.

The component structure is

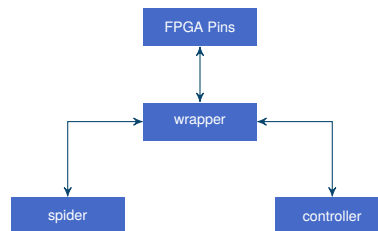


Figure 2.3: Traffic Light Component Structure

2.2 Controller

The controller has five states:

S0: North/South lights green, East/West lights red.

S1: North/South lights yellow, East/West lights red.

S2: North/South lights red, East/West lights green.

S3: North/South lights red, East/West lights yellow.

S4: Fault detected by spider, flash the red lights.

The states of the controller have the above descriptive comments as to what they represent. However, these are only comments and without formal proof, they may or may not be correct; they are in this instance, but in general require proof. The reason they are descriptive is that the main process that decides on state changes is not the process that sets the light colors. The only reason we can with confidence say they are accurate is that the the process that sets the states does so when the state variable in the main processes changes state to a state, say, s_2 , and the lights sets the lights for the state s_2 . Still, this is not formal proof. For a real formal proof, we would have to axiomatize VHDL, which we will not be doing in this project. Rather, we will eventually use ReWire in which case we can use equational logic where the equations can be lifted directly from the code.

In the conclusions, we note that mitigation would be best kept in the spider. If this is the case, this state would cease to exist in the plant. However, then more care must be taken to construct the plant so as not to interfere when mitigation is taking place.

The controller has a wrapper called `top` and three processes: `main`, `lights`, and `timer` for respectively (1) deciding on state changes, (2) setting the lights, and (3) timer to flash the lights when necessary. The main process does not really have that name but is unnamed and residing in the `top` wrapper. These items have the following configuration:

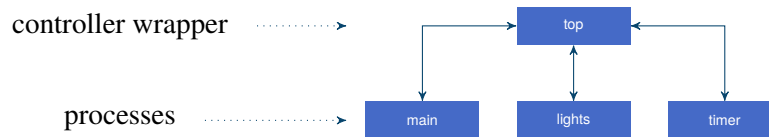


Figure 2.4: Controller Process Structure

The controller has two variables used for controlling the lights, `ns_lights` and `ew_lights` for north-south and east-west lights. The north and south light always mirror each other as do the east-west lights. Each can be in green, yellow, or red.

The next-state relations in the main process of controller can be enumerated. The code could be automatically processed to produce the enumeration. In general, large state machines in applications are avoided due to code complexity. This entails that the next-state relations are relatively small as well. The trade-off comes in concurrence relations (see *Spiders for FPGA Applications* and Section 2.4 below), as there are more of them with many small FSMs. Processing the machines to extract the concurrence relations will be necessary.

The state machine of Figure 2.5 below has several event types that are generated from conditionals in the code. An example of how this occurs is

```

1 case state is
2   when s0 =>
3     if ((t0 = '1') and (t1 = '1')) then
4       state <= s1;
5     elsif (t2 = '1') then
6       state <= s0;
  
```

where we let

$$e_0 \stackrel{\text{def}}{=} (t_0 = 1) \wedge (t_1 = 1) \qquad e_1 \stackrel{\text{def}}{=} (t_2 = 1) \wedge \neg e_0$$

Note that the extra conjunction $\neg e_0 = \neg((t_0 = 1) \wedge (t_1 = 1))$ is needed in defining e_1 since VHDL executes the **elsif** only when the **if** fails.

Event types generate next-state relations. The entire next state relation is the set theoretical union of the individual next-state relations. However, the union washes out the contribution of the individual event types and hence only useful for more abstract properties than we deal with here. We list the event types (next-state relations) here:

Next-State Relation	Event Type	Prescription
$\mathcal{H}_0$	e_0	$\text{rst} = '1' \text{ or } \text{fault} = '1'$
$\mathcal{H}_1$	e_1	$(\text{ew_car} = '0' \text{ and } \text{count} < \text{clk_limit}) \text{ or } (\text{ns_car} = '1' \text{ and } \text{count} < \text{clk_limit}) \text{ or } (\text{count} < "0100")$
$\mathcal{H}_2$	e_2	$\neg \mathcal{H}_1$
$\mathcal{H}_3$	e_3	$\text{count} < "0101"$
$\mathcal{H}_4$	e_4	$\neg \mathcal{H}_3$
$\mathcal{H}_5$	e_5	$(\text{ns_car} = '0' \text{ and } \text{count} < \text{clk_limit}) \text{ or } (\text{ew_car} = '1' \text{ and } \text{count} < \text{clk_limit}) \text{ or } (\text{count} < "0100")$
$\mathcal{H}_6$	e_6	$\neg \mathcal{H}_5$
$\mathcal{H}_7$	e_7	$\text{rst} = '0'$
$\mathcal{H}_8$	e_8	$\neg \mathcal{H}_7$

The state machine (see code for traffic light controller in the Appendix) has only four states. Those four states now have outbound arrows indicating the event types. The event types represent the **if** statement structure in the code.

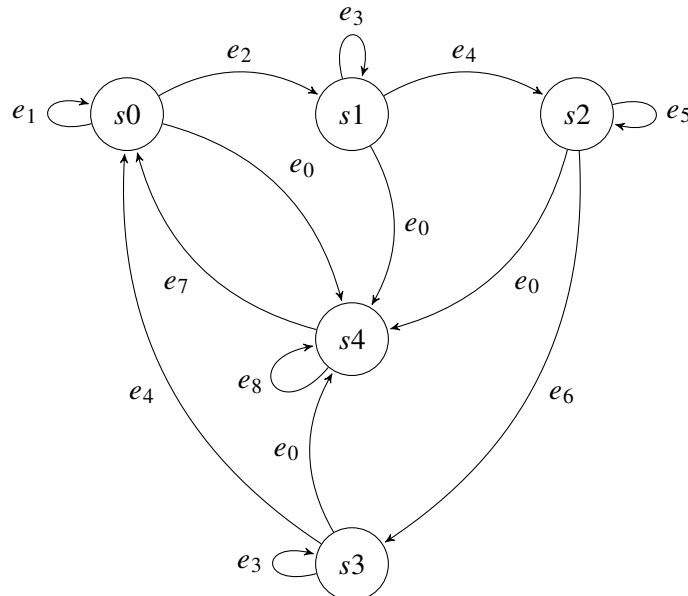


Figure 2.5: Traffic Light FSM

The process of the controller that sets the lights has a state machine which is isomorphic to the main controller process. That process uses the same state symbols as the main process and so the isomorphism is identifying the correct states. Hence we can safely elide this and consider that process as folded into the main process. There is another process of the controller that handles counting clock cycles for flashing the light and has no internal states; so this process can safely thought of as being folded into the main controller process.

The local next state relations for the locality of the controller are H_0 through H_{10} and can be read off the diagram:

Next State Relation $\mathcal{K}$	
Relation	Tuples
$\mathcal{H}_0$	$\{\langle s0, s4 \rangle, \langle s1, s4 \rangle, \langle s2, s4 \rangle, \langle s3, s4 \rangle, \langle s4, s4 \rangle\}$
$\mathcal{H}_1$	$\{\langle s0, s0 \rangle\}$
$\mathcal{H}_2$	$\{\langle s0, s1 \rangle\}$
$\mathcal{H}_3$	$\{\langle s1, s1 \rangle, \langle s3, s3 \rangle\}$,
$\mathcal{H}_4$	$\{\langle s1s2 \rangle, \langle s3, s0 \rangle\}$
$\mathcal{H}_5$	$\{\langle s2, s2 \rangle\}$
$\mathcal{H}_6$	$\{\langle s2, s3 \rangle\}$
$\mathcal{H}_7$	$\{\langle s4, s0 \rangle\}$
$\mathcal{H}_8$	$\{\langle s4, s4 \rangle\}$

Table 2.1: Next State Relations for the Traffic Light Controller

Were we to check state sequencing in the spider of the controller, these relations would be used for designing the checks. We did not do so in order to get a lightweight spider. We bring them up here to show the relationship between event types and next state relations. The overall next state relation for the controller is

$$\mathcal{H} \stackrel{\text{def}}{=} \bigcup_{0 \leq i \leq 8} \mathcal{H}_i.$$

The relation $\mathcal{H}$ would be of use for abstract properties such as “is state i accessible from state j ”.

2.3 Monitoring Conditions

The logic conditions in the spider that were monitored can be collected into *invariants*. That is, they must hold throughout the entire state machine of the traffic light controller. This feature allowed us to not track in the spider the state transitions of the controller. In a more expensive spider, say one where we really want to know precisely what failed in the controller, we would need to track state transitions. The companion report explains some of the machinery necessary to do this.

The spider has two states:

Q0: Checking for faults.

Q1: Fault found, waiting for reset.

The logic fault conditions are

F0: $((\text{ns_lights} = \text{green}) \vee (\text{ns_lights} = \text{yellow})) \wedge (\text{ew_lights} \neq \text{red})$

F1: $((\text{ew_lights} = \text{green}) \vee (\text{ew_lights} = \text{yellow})) \wedge (\text{ns_lights} \neq \text{red})$

F2: $(\text{ns_lights} = \text{red}) \wedge (\text{ew_lights} = \text{red})$

where $\vee$ stands for disjunction, i.e., *or*, and $\wedge$ stands for conjunction, i.e., *and*. Other logic fault conditions

F3: $(\text{ns_lights} = \text{green}) \wedge ((\text{ns_lights} = \text{red}) \vee (\text{ns_lights} = \text{yellow}))$

F4: $(\text{ns_lights} = \text{red}) \wedge ((\text{ns_lights} = \text{green}) \vee (\text{ns_lights} = \text{yellow}))$

F5: $(\text{ns_lights} = \text{yellow}) \wedge ((\text{ns_lights} = \text{red}) \vee (\text{ns_lights} = \text{green}))$

F6: $(\text{ew_lights} = \text{green}) \wedge ((\text{ew_lights} = \text{red}) \vee (\text{ew_lights} = \text{yellow}))$

F7: $(\text{ew_lights} = \text{red}) \wedge ((\text{ew_lights} = \text{green}) \vee (\text{ew_lights} = \text{yellow}))$

F8: $(\text{ew_lights} = \text{yellow}) \wedge ((\text{ew_lights} = \text{red}) \vee (\text{ew_lights} = \text{green}))$

The check time limit event types are

F9: $(\text{ns_count} > \text{clk_limit} + 1) \wedge (\text{DBG_TRIG_ENBL} \vee \neg (\text{ns_lights_r}(2) = '1' \wedge \text{ns_lights}(2) = '0'))$
 $\wedge (\text{ns_lights}(0) = '1' \vee \text{ns_lights}(1) = '1')$

F10: $(\text{ew_count} > \text{clk_limit} + 1) \wedge (\text{DBG_TRIG_ENBL} \vee \neg (\text{ew_lights_r}(2) = '1' \wedge \text{ew_lights}(2) = '0'))$
 $\wedge (\text{ew_lights_r}(2) \neq '1' \wedge (\text{ew_lights}(0) \neq '1' \vee \text{ew_lights}(1) = '1'))$

The event types F9 and F10 require some explanation. The `clock_limit` check is obviously checking for a timing fault. The `DBG_TRIG_ENBL` is for debugging the spider. When it is not set, extra checks are made. We left this in so that it is clear that the spider and its job of checking errors needs to be checked for bugs just as the plant. A further spider could have been written that would check for the conditions following the `DBG_TRIG_ENBL` under the right circumstances. We did not do so in the interest of simplicity.

We define the event type

$$t_0 \stackrel{\text{def}}{=} \bigvee_{0 \leq i \leq 10} F_i = F_0 \vee F_1 \vee \cdots \vee F_{10},$$

That is, t_0 is a label for the disjunction on the right. t_0 's description is `fault_i = '1'` because whenever one of these conditions is satisfied in the code, `fault` is set to 1. The event types for the spider state machine are

Next-State Relation	Event Type Prescription	
$\mathcal{K}_0$	t_0	$\text{fault_i} = '1'$
$\mathcal{K}_1$	t_1	$\neg \mathcal{K}_0$
$\mathcal{K}_2$	t_2	$\text{spider_rst} = '0' \text{ and } \text{fault_i} = '0'$
$\mathcal{K}_3$	t_3	$\neg \mathcal{K}_2$

The state machine for the spider is

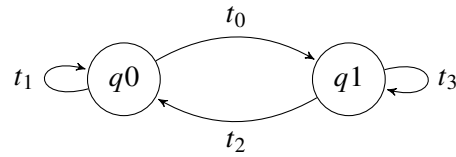


Figure 2.6: Spider State Machine

The local next state relations for the locality of the spider are $\mathcal{K}_0$ through $\mathcal{K}_3$ and can be read off the diagram:

Concurrence Relation $\mathcal{F}$	
Relation	Tuples
$\mathcal{K}_0$	$\{\langle q0, q1 \rangle\}$
$\mathcal{K}_1$	$\{\langle q0, q0 \rangle\}$
$\mathcal{K}_2$	$\{\langle q1, q0 \rangle\}$
$\mathcal{K}_3$	$\{\langle q1, q1 \rangle\}$

Table 2.2: Next State Relations for the Spider

The overall next state relation for the controller is

$$\mathcal{K} \stackrel{\text{def}}{=} \bigcup_{0 \leq i \leq 3} \mathcal{K}_i.$$

The spider checked for fault conditions F0 through F10. If one of these faults occurred, the spider in state Q1 caused the lights to blink red by signaling to the controller to enter its state S4.

2.4 Distributed Structure

The sole distributed relation $\mathcal{F} \subseteq H \times K$ pairs possible states from each state machine that can operate in parallel. We do not use this relation in defining any distributed logic connectives for the reason that all we used were invariants that hold regardless of the states in which the spider and the controller find themselves. The distributed relation $\mathcal{F}$ would be of use if the spider checked state changes in the controller.

The distributed relation $\mathcal{F}$ showing the states in the spider and controller that can concur is

Concurrence Relation $\mathcal{F}$	
Relation	Tuples
$\mathcal{F}$	$\{\langle q0, s0 \rangle, \langle q0, s1 \rangle,$ $\langle q0, s2 \rangle, \langle q0, s3 \rangle,$ $\langle q1, s4 \rangle\}$

Table 2.3: Concurrence Relation for the Spider and the Traffic Light Controller

3. CONCLUSIONS

We learned quite a bit about spiders and plants from the traffic light system. The notion of *invariants* came about by trying to write a simple spider and realizing that to check everything the traffic light controller did required essentially a copy of the controller in the spider. Disregarding mitigation code, a similar effect can be had by running two copies of the controller and comparing their outputs. However, this will not check design errors in the controller and there is no recovery mechanism when some error is detected.

Our spider for the traffic light controller used a minimal set of states to watch over all the states of the controller. Put in different terms, we did not wish to trust that the outputs were set correctly. Rather, the conditions in which the spider is interested are sections of the cross product of the outputs. A section is a particular element of that cross product, i.e., a tuple of values for all the outputs, or at least the outputs in which the spider is interested.

Apparently in VHDL, an architecture cannot read its own outputs. To read the values, those outputs must be stored in buffers. Hence the use of similar sounding signals in the architecture which record the values of those outputs. The use of those kinds of signals is slightly different between the spider and the controller. This can cause a confusion in anyone reviewing the spider + controller combination. The result is that at the clock strike, the similar signals get set to the current new values of the outputs as well as the outputs.

Typically, the plant is produced first and then a spider written. Optimally, it would be best for the spider to be written at the same time as the plant. The main reason for this has to do with the signals a spider requires in order to adequately monitor a plant. Every signal the spider requires from the plant must be exported from the plant's code so that the spider can use it. This feature means that it is a good idea to build a wrapper module for the spider-plant combination with an eye toward's integrating this combination with a

larger system. It might easily be the case that there is more than one spider in a system, each concerned with a specific module as its plant. It may even be that for increased high assurance, there are spiders monitoring other spiders or spider-plant combinations.

There appear to be at least two philosophies for writing a spider with a specific plant in mind. One is that the mitigation code resides mostly in the spider and the other where the mitigation code resides mostly in the plant. Remember that both are usually compiled together by vendor tools. There must be a minimal amount of mitigation code in the spider even if most of it resides in the plant. That minimal amount directed the plant's mitigation code. There might also be a minimal amount of mitigation code in the plant for the case where the plant should not operate as normal while mitigation is occurring. However, it might be possible to put no mitigation code in the plant and have the spider gate the plant's signals to prevent the plant from interfering with the mitigation.

One good reason for putting the mitigation entirely (if possible) in the spider is that this avoids touching the module being spidered, the con is that this would reduce the insight we have into that module. The primary reason that the separation would be useful is if/when the spider is autogenerated and integrated into the design at a higher level, the auto-generation of the spider would not involve updating existing modules.

There is a security consideration that it might be good to keep the mitigation logic totally within the spider itself. Assume the plant consists of "secure" code with its own provenance. The spider acts as a 'gating' function for signals being protected and the untrusted module (spider) does not directly touch the secure code. This begs the question of how secure is the spider itself. This could be secured using the same mechanisms as the plant. However, the plant might come from a trusted source and access to the source code may not even be possible. In this latter case, mitigation necessarily must be performed in the spider.

4. APPENDIX

4.1 Wrapper Code

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  library xil_defaultlib ;
5
6  entity top_wrap is
7    generic (
8      blank           : std_logic_vector(2 downto 0):= "000" ;   -- ? blank ?
9      green           : std_logic_vector(2 downto 0):= "001" ;
10     yellow          : std_logic_vector(2 downto 0):= "010" ;
11     red             : std_logic_vector(2 downto 0):= "100" ;
12     flash_red_period : std_logic_vector(3 downto 0):= "1000";
13     clk_limit        : std_logic_vector(3 downto 0):= "1001";
14     error_none       : std_logic_vector(2 downto 0):= "000" ;
15     error_reg        : std_logic_vector(2 downto 0):= "001" ;
16     error_mit        : std_logic_vector(2 downto 0):= "010");
17  port (
18    -- inputs
19    clk           : in    std_logic ;           -- clock signal
20    spider_rst    : in    std_logic ;           -- spider reset signal to put spider

```

```

21                                     -- in quiescent state
22     ns_car      : in    std_logic;      -- signal to represent presence of
23                                     -- a car at N/S intersection
24     ew_car      : in    std_logic;      -- signal to represent presence of
25                                     -- a car at E/W intersection
26     -- outputs
27     error       : out   std_logic_vector (2 downto 0);  -- "001" = error, "010" = error in
28                                     -- mitigation
29     fault       : out   std_logic;      -- signal to light controller that a fault
30                                     -- has been detected
31     ns_lights   : out   std_logic_vector (2 downto 0) := "001"; -- North/South intersection lights;
32                                     -- see generic above for values
33     ew_lights   : out   std_logic_vector (2 downto 0) := "100"; -- East/West intersection lights;
34                                     -- see generic above for values
35 end top_wrap;
36 architecture rtl of top_wrap is
37     component top is
38         generic (
39             blank      : std_logic_vector(2 downto 0) := blank ;      -- "000" ; -- ? blank ?
40             green      : std_logic_vector(2 downto 0) := green ;      -- "001" ;
41             yellow     : std_logic_vector(2 downto 0) := yellow;      -- "010" ;
42             red        : std_logic_vector(2 downto 0) := red ;        -- "100" ;
43             flash_red_period : std_logic_vector(3 downto 0) := flash_red_period; -- "1000";
44             clk_limit   : std_logic_vector(3 downto 0) := "1001");
45         port (
46             -- inputs
47             clk        : in    std_logic;      -- clock signal
48             rst        : in    std_logic;      -- reset signal to set N/W lights to
49                                     -- green & E/W lights to red
50             ns_car     : in    std_logic;      -- signal to represent presence of
51                                     -- a car at N/S intersection
52             ew_car     : in    std_logic;      -- signal to represent presence of
53                                     -- a car at E/W intersection
54             fault      : in    std_logic;      -- signal from spider that a fault
55                                     -- has been detected by the spider
56             -- outputs
57             ns_lights   : out   std_logic_vector (2 downto 0) := "001"; -- North/South intersection
58                                     -- lights; see generic above
59                                     -- for values
60             ew_lights   : out   std_logic_vector (2 downto 0) := "100"; -- East/West intersection
61                                     -- lights; see generic above
62                                     -- for values
63     end component;
64
65     component spider_top is
66         generic (
67             green      : std_logic_vector(2 downto 0) := green ;      -- "001" ;
68             yellow     : std_logic_vector(2 downto 0) := yellow;      -- "010" ;
69             red        : std_logic_vector(2 downto 0) := red ;        -- "100" ;
70             clk_limit   : std_logic_vector(3 downto 0) := clk_limit ;  -- "1001";
71             error_none  : std_logic_vector(2 downto 0) := error_none;  -- "000" ;
72             error_reg   : std_logic_vector(2 downto 0) := error_reg ;  -- "001" ;

```

```

73     error_mit    : std_logic_vector(2 downto 0):= error_mit;           -- "010" );
74     port (
75         -- inputs
76         clk       : in  std_logic;           -- clock signal
77         spider_rst : in  std_logic;           -- spider reset signal to put spider
78                                           -- in quiescent state
79         ns_lights  : in  std_logic_vector (2 downto 0);           -- North/South intersection
80                                           -- lights; "001" = green, "010" = yellow, "100" = red
81         ew_lights  : in  std_logic_vector (2 downto 0);           -- East/West intersection
82                                           -- lights; "001" = green, "010" = yellow, "100" = red
83         -- outputs
84         error      : out std_logic_vector (2 downto 0);           -- "001" = error, "010" = error in mitigation
85         rst        : out std_logic;           -- "001" = error, presumably we will want more
86         fault      : out std_logic;           -- signal to light controller that a fault
87                                           -- has been detected
88     end component;
89
90     signal rst_i           : std_logic;
91     signal fault_i         : std_logic;
92     signal ns_lights_i     : std_logic_vector (2 downto 0);
93     signal ew_lights_i     : std_logic_vector (2 downto 0);
94     signal error_i         : std_logic_vector (2 downto 0);
95
96     begin
97
98     fault <= fault_i;
99     ns_lights <= ns_lights_i;
100    ew_lights <= ew_lights_i;
101
102
103    top_i: top
104        generic map (
105            blank      => blank      ,
106            green      => green      ,
107            yellow     => yellow     ,
108            red        => red        ,
109            flash_red_period => flash_red_period,
110            clk_limit  => clk_limit  )
111        port map (
112            clk        => clk        ,
113            rst        => rst_i      ,
114            ns_car     => ns_car     ,
115            ew_car     => ew_car     ,
116            fault      => fault_i    ,
117            ns_lights  => ns_lights_i ,
118            ew_lights  => ew_lights_i );
119
120    spider_i: spider_top
121        generic map (
122            green      => green      ,
123            yellow     => yellow     ,
124            red        => red        ,

```

```

125         clk_limit      => clk_limit  ,
126         error_none     => error_none ,
127         error_reg       => error_reg  ,
128         error_mit       => error_mit  )
129     port map (
130         clk              => clk        ,
131         spider_rst       => spider_rst ,
132         ns_lights        => ns_lights_i,
133         ew_lights        => ew_lights_i,
134         error            => error       ,
135         rst              => rst_i       ,
136         fault            => fault_i     );
137
138 end rtl;

```

4.2 Traffic Light Controller

```

1
2  -- Module to control traffic lights at a four-way intersection
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.std_logic_unsigned.all;
7  use ieee.numeric_std.all;
8  entity top is
9      generic (
10         blank          : std_logic_vector(2 downto 0) := "000";  -- ? blank ?
11         green          : std_logic_vector(2 downto 0) := "001";
12         yellow         : std_logic_vector(2 downto 0) := "010";
13         red            : std_logic_vector(2 downto 0) := "100";
14         flash_red_period : std_logic_vector(3 downto 0) := "1000";
15         clk_limit       : std_logic_vector(3 downto 0) := "1001");
16     port (
17         -- inputs
18         clk          : in    std_logic;          -- clock signal
19         rst          : in    std_logic;          -- reset signal to set N/W lights to
20                                                -- green & E/W lights to red
21         ns_car       : in    std_logic;          -- signal to represent presence of
22                                                -- a car at N/S intersection
23         ew_car       : in    std_logic;          -- signal to represent presence of
24                                                -- a car at E/W intersection
25         fault        : in    std_logic;          -- signal from spider that a fault has
26                                                -- been detected by the spider
27         -- outputs
28         ns_lights     : out   std_logic_vector (2 downto 0) := "001";  -- North/South intersection lights;
29                                                -- see generic above for values
30         ew_lights     : out   std_logic_vector (2 downto 0) := "100";  -- East/West intersection lights;
31                                                -- see generic above for values
32     end top;
33
34 architecture rtl of top is
35     type state_type is (s0, s1, s2, s3, s4);

```

```

36  signal state      : state_type;
37  signal count      : std_logic_vector(3 downto 0);
38  signal flash_count : std_logic_vector(3 downto 0) := (others => '0');
39
40  signal ns_lights_i  : std_logic_vector (2 downto 0);    -- North/South intersection lights; see generic above
41                                     -- for values
42  signal ew_lights_i  : std_logic_vector (2 downto 0);    -- East/West intersection lights; see generic above
43                                     -- for values
44  signal blank_r      : std_logic := '0';
45
46
47  begin
48
49  --reset(rst) issue  @rising edge make a transition to red light blanking state
50  --                  @falling edge make a transition to NS-GR EW-RD (s0)
51  --if fault exist    make a transition to red light blanking state
52
53  ns_lights <= ns_lights_i;
54  ew_lights <= ew_lights_i;
55
56
57  process (clk)
58  begin
59      -- case s0: North/South lights green, East/West lights red
60      -- case s1: North/South lights yellow, East/West lights red
61      -- case s2: North/South lights red, East/West lights green
62      -- case s3: North/South lights red, East/West lights yellow
63      -- case s4: fault detected by spider, flash the red lights
64  if (clk'event and clk = '1') then
65
66      if (rst = '1' or fault = '1' ) then
67          state <= s4;                                -- (e9) flash red until the reset is taken away
68          count <= "0000";
69      else
70
71          case state is
72          when s0 =>
73              if ((ew_car = '0' and count < clk_limit) or ---- (e1): no car present on EW and count less than 9
74                  (ns_car = '1' and count < clk_limit) or ---- car present on NS and count less than 9
75                  (count < "0100")) then ---- count less than 4 (car exist does not matter)
76                  state <= s0; -- stay in the same state until condition are false (0-5 sec don't check car
77                               -- is there)
78                  count <= count + 1;
79              else ---- (e2):
80                  state <= s1;
81                  count <= "0000";
82              end if;
83          when s1 =>
84              if (count < "0101") then ---- (e3) count less than 5? (error)
85                  state <= s1;
86                  count <= count + 1;
87              else ---- (e4)

```

```

88         state <= s2;
89         count <= "0000";
90     end if;
91     when s2 =>
92         if ((ns_car = '0' and count < clk_limit) or ---- (e5) no car present on NS and count less than 9
93             (ew_car = '1' and count < clk_limit) or ---- car present on EW and count less than 9
94             (count < "0100")) then ---- count less than 4? (error)
95             state <= s2;
96             count <= count + 1;
97         else ---- (e6)
98             state <= s3;
99             count <= "0000";
100        end if;
101        when s3 =>
102            if (count < "0101") then ---- (e3) count is less than 5? (error)
103                state <= s3;
104                count <= count + 1;
105            else ---- (e4)
106                state <= s0;
107                count <= "0000";
108            end if;
109            when s4 => ---- (e7) note: a reset is required to get out of s4
110                if (rst = '0') then
111                    state <= s0;
112                end if; ---- (e8) no change in state
113            when others =>
114                state <= s0;
115        end case;
116    end if;
117 end if;
118 end process;
119
120 -- this special flash counter is work only when reset is asserted.
121 process (clk)
122 begin
123     if (clk'event and clk = '1') then
124         ---- flash_red_period is less than 8? (error) flash_count <= "0000";
125         if rst = '0' then
126             flash_count <= (others => '0');
127             blank_r <= '0';
128         elsif (unsigned(flash_count) > unsigned(flash_red_period)) then
129             flash_count <= (others => '0');
130             blank_r <= not blank_r;
131             --wf(blank_r = '0') then
132             -- blank_r <= '1';
133             --else
134             -- blank_r <= '0';
135             --end if;
136         else
137             flash_count <= flash_count + 1;
138         end if;
139     end if;

```

```

140 end process;
141
142 lights : process(state, blank_r)
143 begin
144
145     case state is
146         when s0 =>
147             ns_lights_i <= green;
148             ew_lights_i <= red;
149
150         when s1 =>
151             ns_lights_i <= yellow;
152             ew_lights_i <= red;
153
154         when s2 =>
155             ns_lights_i <= red;
156             ew_lights_i <= green;
157
158         when s3 =>
159             ns_lights_i <= red;
160             ew_lights_i <= yellow;
161
162         when s4 =>
163
164             if (blank_r = '0') then
165                 ns_lights_i <= blank;
166                 ew_lights_i <= blank;
167
168             else
169                 ew_lights_i <= red;
170                 ns_lights_i <= red;
171
172             end if;
173
174         when others =>
175             ns_lights_i <= green;
176             ew_lights_i <= red;
177     end case;
178
179 end process;
180
181 end rtl;

```

4.3 Spider

- 1 -- Spider Module (prenatal) for checking traffic light controller at a four-way intersection
- 2
- 3 -- In this version, only simple conditions are checked, namely invalid outputs.
- 4 -- The idea was to construct the minimal spider; hence there is only a single state, q_0
- 5 -- Under the understanding of a minimal spider, the controller has not been changed.
- 6 -- interested what we can see from the outside. Sooner or later, the internal states of the
- 7 -- controller should be exported.

```

8
9  library ieee;
10 use ieee.std_logic_1164.all;
11 use ieee.std_logic_unsigned.all;
12
13 entity spider_top is
14     generic (
15         DBG_TRIG_ENBL : boolean := false;
16         green         : std_logic_vector(2 downto 0) := "001" ;
17         yellow        : std_logic_vector(2 downto 0) := "010" ;
18         red           : std_logic_vector(2 downto 0) := "100" ;
19         clk_limit      : std_logic_vector(3 downto 0) := "1001";
20         error_none     : std_logic_vector(2 downto 0) := "000" ;
21         error_reg      : std_logic_vector(2 downto 0) := "001" ;
22         error_mit      : std_logic_vector(2 downto 0) := "010";
23     port (
24         -- inputs
25         clk          : in  std_logic;           -- clock signal
26         spider_rst    : in  std_logic;           -- spider reset signal to put spider in
27                                                    -- quiescent state
28         ns_lights     : in  std_logic_vector (2 downto 0); -- North/South intersection
29                                                    -- lights; "001" = green, "010" = yellow, "100" = red
30         ew_lights     : in  std_logic_vector (2 downto 0); -- East/West intersection
31                                                    -- lights; "001" = green, "010" = yellow, "100" = red
32         -- outputs
33         error         : out std_logic_vector (2 downto 0); -- "001" = error, "010" = error
34                                                    -- in mitigation
35         rst           : out std_logic;           -- "001" = error, presumably we will
36                                                    -- want more
37         fault         : out std_logic;           -- signal to light controller that a
38                                                    -- fault has been detected
39     end spider_top;
40
41 architecture rtl of spider_top is
42     type state_type is (q0, q1);
43     signal state      : state_type := q0;
44     signal ns_count   : std_logic_vector(3 downto 0);
45     signal ew_count   : std_logic_vector(3 downto 0);
46     signal error_count : std_logic_vector(3 downto 0);
47     signal fault_i     : std_logic;
48     signal error_code  : natural;
49     signal ns_lights_r : std_logic_vector (2 downto 0); -- North/South intersection lights;
50                                                    -- "001" = green, "010" = yellow, "100" = red
51     signal ew_lights_r : std_logic_vector (2 downto 0); -- East/West intersection lights;
52                                                    -- "001" = green, "010" = yellow, "100" = red
53 begin
54     fault <= fault_i ;
55
56     process (clk, spider_rst)
57     begin
58         -- states for the spider
59         -- case q0: fault conditions being checked normally

```

```

60 -- case q1: fault detected, awaiting reset
61
62 -- Reset is a bit tricky to code without causing a race condition. We check for both rising and falling edges
63 -- with the idea that the rst to the controller should not be bouncing around, hence we waited until the
64 -- spider's reset has fallen and then reset the controller. It is a bit unclear about how this is wired up
65 -- external to the spider and the controller. Right now, it is assumed the rst the controller is only set
66 -- in the spider. However, it could be both could be set at the same time external to the spider.
67
68 -- There is also an assumption that the controller's clock is sufficiently fast to catch the rising edge of
69 -- its spider's reset and the falling edge will have of the spider's reset will have sufficient time elapsed
70 -- for the controller to recognize its reset before the spider sets that signal back to '0';
71
72
73 if (spider_rst = '1') then
74
75     ns_count <= "0000";    -- counter for ns_light duration since changed from last change
76     ew_count <= "0000";    -- counter for ew_light duration since changed from last change
77     error_count <= "0000";
78     error <= error_none;
79     rst <= '1';
80     fault_i <= '0';
81     state <= q1;
82     error_code <= 0;        --error_code to distinguish the fault why it is generated.
83     ns_lights_r <= "000"; --register ns_lights and ew_lights to detect the light has been changed.
84     ew_lights_r <= "000";
85
86 elseif (clk'event and clk = '1') then
87     rst <= '0';
88     --register ns_lights and ew_lights to detect the light has been changed.
89     ns_lights_r <= ns_lights;
90     ew_lights_r <= ew_lights;
91
92     if (ns_lights_r /= ns_lights) then --register ns_lights and ew_lights to detect the light
93                                         -- has been changed.
94         ns_count <= (others => '0');
95     else
96         ns_count <= ns_count + 1;
97     end if;
98     if (ew_lights_r /= ew_lights) then --register ns_lights and ew_lights to detect the light
99                                         -- has been changed.
100         ew_count <= (others => '0');
101     else
102         ew_count <= ew_count + 1;
103     end if;
104     case state is
105         when q0 =>
106
107             if ((( ns_lights = green) or (ns_lights = yellow)) and (ew_lights /= red)) then
108                 error <= error_reg;
109                 fault_i <= '1';
110                 error_count <= error_count + 1;
111                 error_code <= 1;

```

```

112      elsif (((ew_lights = green) or (ew_lights = yellow)) and (ns_lights /= red)) then
113          error <= error_reg;
114          fault_i <= '1';
115          error_count <= error_count + 1;
116          error_code <= 2;
117      elsif ((ns_lights = red) and (ew_lights = red)) then
118          error <= error_reg;
119          fault_i <= '1';
120          error_count <= error_count + 1;
121          error_code <= 3;
122      end if;
123
124      if ((( ns_lights and red) = red) and (((ns_lights and green) = green) or
125          (( ns_lights and yellow) = yellow))) then
126          error <= error_reg;
127          fault_i <= '1';
128          error_count <= error_count + 1;
129          error_code <= 4;
130      elsif ((( ns_lights and green) = green) and (((ns_lights and red) = red) or
131          (( ns_lights and yellow) = yellow))) then
132          error <= error_reg;
133          fault_i <= '1';
134          error_count <= error_count + 1;
135          error_code <= 5;
136      elsif ((( ns_lights and yellow) = yellow) and (((ns_lights and red) = red) or
137          (( ns_lights and green) = green))) then
138          error <= error_reg;
139          fault_i <= '1';
140          error_count <= error_count + 1;
141          error_code <= 6;
142      end if;
143
144      if ((( ew_lights and red) = red) and (((ew_lights and green) = green) or
145          ((ew_lights and yellow) = yellow))) then
146          error <= error_reg;
147          fault_i <= '1';
148          error_count <= error_count + 1;
149          error_code <= 7;
150      elsif ((( ew_lights and green) = green) and (((ew_lights and red) = red) or
151          ((ew_lights and yellow) = yellow))) then
152          error <= error_reg;
153          fault_i <= '1';
154          error_count <= error_count + 1;
155          error_code <= 8;
156      elsif ((( ew_lights and yellow) = yellow) and (((ew_lights and red) = red) or
157          ((ew_lights and green) = green))) then
158          error <= error_reg;
159          fault_i <= '1';
160          error_count <= error_count + 1;
161          error_code <= 9;
162      end if;
163

```

```

164      -- We choose a minimal condition here in lieu of recreating the entire controller's time checks
165      if (ns_count > clk_limit+1 and (DBG_TRIG_ENBL or not (ns_lights_r(2) = '1' and ns_lights(2) = '0'))
166          and (ns_lights(0) = '1' or ns_lights(1)='1'))
167      or (ew_count > clk_limit+1 and (DBG_TRIG_ENBL or not (ew_lights_r(2) = '1' and ew_lights(2) = '0'))
168          and (ew_lights_r(2) /= '1') and ( ew_lights(0) /= '1' or ew_lights(1)='1'))
169          then
170          fault_i <= '1';
171          error_code <= 10;
172      end if;
173
174      if ( fault_i = '1') then
175          -- flip to the main fault_i state
176          state <= q1;
177      end if;
178
179      when q1 =>
180          if (spider_rst= '0' and fault_i = '0') then
181              state <= q0;
182          end if;
183          -- awaiting reset, we minimally spider the flashing red lights
184          if ((ns_lights /= red) and (ew_lights = red)) then
185              error <= error_mit;
186          elsif ((ns_lights = red) and (ew_lights /= red)) then
187              error <= error_mit;
188          end if;
189
190      when others =>
191          state <= q0;
192      end case;
193
194  end if;
195 end process;
196
197 end rtl;

```

REFERENCES

1. A. Procter, W.L. Harrison, I. Graves, M. Becchi, and G. Allwein, “A Principled Approach to Secure Multi-Core Processor Design with ReWire,” Proceedings of the Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (ACM Digital Library), 2016.
2. W.L. Harrison, A. Procter, I. Graves, M. Becchi, and G. Allwein, “A Programming Model for Reconfigurable Computing Based in Functional Concurrency,” Proceedings of the Proceedings of the 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2016) (IEEE), 2016, pp. 1–8.
3. P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” Proceedings of the Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA (ACM Press), 1977, pp. 238?–252.

4. G. Allwein and W.L. Harrison, “Distributed Modal Logic,” in K. Bimbó, ed., *J. Michael Dunn on Information Based Logic: Outstanding Contributions to Logic*, pp. 331–362 (Springer-Verlag, 2016).
5. J.M. Dunn and G. Hardegree, *Algebraic Methods in Philosophical Logic*, Oxford Logic Guides 41 (Oxford University Press, 2001).
6. C. A. R. Hoare, “An Axiomatic Basis for Computer Programming,” *Communications of the ACM* **12**, 576–580, 583 (1969).
7. K. R. Apt, “Ten Years of Hoare’s Logic: A Survey–Part 1,” *ACM Transactions on Programming Languages and Systems* **3**, 431–483 (1981).
8. D. Barker-Plummer, J. Barwise, and J. Etchemendy, *Language, Proof, and Logic*, second ed. (CSLI Publications, 2011).
9. P. J. Freyd and A. Scedrov, *Categories, Allegories* (North–Holland, 1990).