A (Mildly) Intelligent Traffic Light System (a Bluespec tutorial)

October 11, 2009

1 Introduction

In this document, as one of our introductory tutorials to Bluespec, we develop a Bluespec specification of an intelligent traffic light system at an intersection. We start with an informal description of the desired traffic light system, and then we develop a series of Bluespec specifications that incrementally solve more and more of the problem until we have a complete solution.

It is hoped that this tutorial will serve two purposes. First, it is provided as a further worked example for study by those who have already undertaken Bluespec training; but secondly, we believe that it is an introduction to Bluespec which should be understandable to anyone who has a basic knowledge of programming languages.

Working Bluespec source code is available for all these versions: version n is found in file `TL`n`.bsv`. Please refer to these sources continually as you read this document. The sources are included in Appendix A for reference, but you may also want to print them out separately or view them on your screen. Please also keep handy a copy of the Bluespec language manual for precise descriptions of grammar and semantics.

Along with the code files of the various versions there are testbenches for them: the testbench for file `TL`n`.bsv` is in file `TbTL`n`.bsv`. These testbenches are not described in this tutorial document; but an examination of them may reveal further features of Bluespec which might whet the reader's appetite for further study.

2 Informal description of the desired traffic light system

The road intersection and the traffic light system are depicted schematically in Figure 1[1]. It is a 4-way intersection, with roads to the North, South, East and West. The North-South route is a "major" route whereas the East and West routes are "minor" routes. The normal cycle of the lights is:

- Allow northbound and southbound traffic (both directions): 20 seconds green, 4 seconds amber, 2 seconds all-red;
- Allow eastbound traffic: 10 seconds green, 4 seconds amber, 2 seconds all-red;
- Allow westbound traffic: 10 seconds green, 4 seconds amber, 2 seconds all-red.

We refer to each green/amber/all-red set as a "sequence". The "all-red" periods are safety periods in which all lights at the intersection are red.

The light system has three kinds of "intelligent behavior". The first is that it can, on request, insert a pedestrian sequence into the normal sequence. The intersection is equipped with buttons that pedestrians can press. When pressed at any point in a sequence, then, after the all-red at the end of the current sequence, the pedestrian lights go green (10 seconds) while all road traffic lights remain red. Then, the pedestrian lights go amber (6 seconds), and there is again an all-red safety period (2 seconds), and the system goes to the green of the

[1] If you are from Australia or another "drive on the left" country, please consider the picture as a view from "down under" the roadway, instead of from above, which is where Americans tend to look at things from.

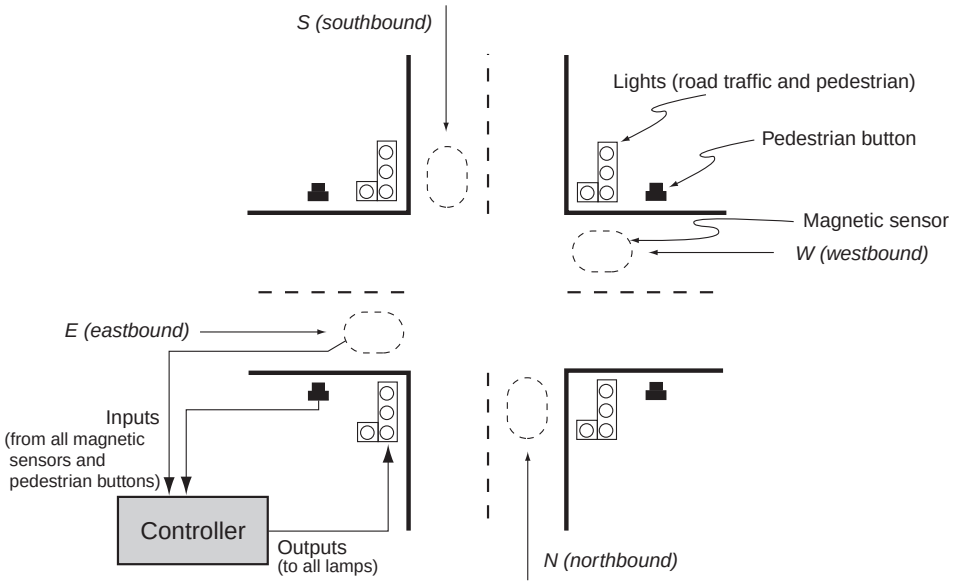

Figure 1: Schematic view of road intersection and traffic light system

next sequence. For example, if a pedestrian button is pressed during the eastbound green, then what will follow is the eastbound amber, all-red safety, pedestrian green, pedestrian amber, all-red safety, and then the westbound green, and so on. Once a pedestrian button has been pressed, the pedestrian buttons are ignored until the completion of the ensuing pedestrian sequence.

All four incoming roads have magnetic sensors that can detect whether or not a car is present on that incoming road. The second kind of intelligent behavior is that if no car is waiting on an incoming road, then the corresponding green/amber/all-red sequence is skipped. (This means that if no car is waiting on any of the four incoming roads, then all lights stay red.) Finally, the third kind of intelligent behavior is that once a particular traffic flow (N-S, E or W) has been given a green (because there was at least one car waiting), if the magnetic sensors then detect that there are no more cars in that flow, the green is transitioned immediately into its corresponding amber, instead of staying green for its full normal delay.[2] In this document, the suffixes N, S, NS, E and W always refer to northbound,

[2]After reading this document, you may wish to solidify your understanding of Bluespec by modifying the program to solve the following variation of the problem that favors the major north-south route:

- The default state, when there is no E-W traffic nor any pedestrian request, is north-south green with no time limit, instead of all-red. In this state, north-south traffic can flow without slowing.
- We trigger an exit out of the north-south green state only if there is a specific request (pedestrian request, or E-W traffic present). As before, we transition to `AmberNS` after `ns_green_delay` seconds or as soon as there is no more N-S traffic, whichever comes earlier.
- We short-circuit the north-south green state (skip it, or truncate it) only in if there is a specific request

southbound, northbound-and-southbound, eastbound and westbound, respectively.

Our goal is to build the box labelled "Controller" in the figure. It takes Boolean inputs from all the magnetic sensors and pedestrian buttons, and produces Boolean outputs that switch all the lamps on or off. We will assume that all the pedestrian buttons are OR'ed together.

3 Modules (= hardware)

Bluespec is a hardware description language, and the central construct to describe hardware is the `module`. A module describes a "black box" like the Controller in Figure 1. It will have some internal state, and it will have some input and output wires. It can have independent internal behavior: that is, it is clocked, and its internal state may change even if there are no changes on its external inputs. It will also typically have some externally visible behavior: that is, the values on its output wires may vary based on changes in input wires and internal state, and internal state may change in response to changes on the input wires, thereby affecting future outputs.

The external world (outside the "black box") can only interact with the module *via* its input and output wires. All the input and output wires are collectively called the *interface* of the module. We will further organize the interface into logically separate sub-groups called *methods*. In fact, the interface of a module will be precisely specified with a type T, and the module itself will have type[3] `Module#`(T). For an interface type T of a hardware module, there is a precise translation of T into input and output wires.

A subtlety: a `module` definition more precisely describes a *generator* of a piece of hardware rather than a piece of hardware, since it can be instantiated multiple times; that is, the module expression is a *schema* that describes any such instantiation. However, we are often sloppy about this distinction in text and oral discussion.

There is a strong analogy here with classes, objects and interfaces in object-oriented programming languages like Java and C++. A Bluespec module is like a class definition, including the definition of the private state encapsulated within objects (instances) of that class. A Bluespec module also plays the role of the class constructor, i.e., it creates objects of the class. Bluespec interface methods are exactly like class interface methods.

In general, modules are a powerful mechanism for structuring large designs. It is possible to have a lengthy and detailed discussion of this topic, about how modules relate to each other (hierarchically, as peers, with interaction disciplines that facilitate layout on a chip, and so on) but in this document our focus is on basic Bluespec concepts used within a single module.

(pedestrian request, or E-W traffic present).

The rest of the specification remains the same (amber lights, durations, *etc.*).

[3]This is actually a slight oversimplification, but it is true for all modules which are synthesized into hardware.

4 A word about types

Bluespec is a language with *static and strong typing*. Strong typing means means that every expression in a Bluespec program has a definite type and the system checks that the only operations allowed on values of type T are those that are meaningful for T. For example, it does not make sense to apply the square root operation to an internet address. Note that raw circuits, by themselves, do not enforce any such discipline. Since all values are ultimately represented as bits, and raw circuits just operate on bits, it is perfectly feasible to supply the bits representing an internet address to the inputs of a square root circuit. Thus, the type discipline which prevents such meaningless applications is a purely linguistic mechanism enforced by the language and the language system.

Static typing means that the type discipline is enforced at the level of source programs (that is, without having to execute the programs). In Bluespec, the compiler performs these checks to ensure that the type discipline is followed.

Types are also a central tool that allows us to think abstractly about values, without worrying about their representations in bits. Unlike traditional hardware description languages, in Bluespec the programmer does not much think directly in terms of bits. Instead, the programmer thinks about abstract entities like "internet addresses" or "hash table entries" or "packet headers" or "processor instructions" or "cache lines" or "queues" *etc.* This is just what we do in any high level language, where we think in terms of "structs" or "unions" or "arrays" or "objects" and so on, leaving it up to the compiler to choose the details of their representation in bits.

In Bluespec programs, we frequently see phrases like:

```
T x;
```

Such a phrase simply declares that we will be using a variable "`x`" with type "`T`". Note that this does not create any hardware for the variable; it merely reserves the name to refer to a value of the given type.

5 Version TL0: the normal sequence of states

This is a first cut and simply addresses the normal sequence. It does not address actual time durations of each state, nor any of the intelligent behaviors (pedestrian buttons, magnetic sensors).

The central piece of the code is in the lines:

```
module sysTL(TL);
   ...
   ...
endmodule: sysTL
```

which represent the Controller hardware, but let us work our way to this in stages. The outermost structure of the whole file is:

```
package TL0;
   ...
   ...
endpackage: TL0
```

It specifies that we are defining a Bluespec package called `TL0`.

Side Comment

Packages are Bluespec's mechanism for organizing large programs into separate files, and controlling which names from one file are visible within other files. It is purely a programming convenience and has nothing to do with hardware structure. Please see the Reference Guide for more details. Note that the colon and the name of the package after the `endpackage` keyword (and other similar keywords at the ends of things) are optional; they help the reader to keep track of the structure of large designs.

End of Side Comment

Lines such as these, beginning with two slashes, are just comments:

```
// Simple model of a traffic light
// (modeled after the light at the intersection of Rte 16 and Broadway
//  on the border between Arlington, MA and Somerville, MA)
```

The code:

```
interface TL;
endinterface: TL
```

defines a new type, an interface type called `TL`. We will be using this as the interface (connections to the external world) of our traffic light system. The current definition defines this to be empty (no connections); later we will add the pedestrian request button and magnetic sensors.

The code:

```
typedef enum {
   GreenNS, AmberNS, RedAfterNS,
   GreenE, AmberE, RedAfterE,
   GreenW, AmberW, RedAfterW} TLstates deriving (Eq, Bits);
```

defines a new data type `TLstates` as an enumeration of nine distinct values. The `deriving` clause instructs the Bluespec compiler to pick a way to compare these values for equality, and to pick a way to represent this type in bits (so that we can store such values in registers).

Side Comment

More formally, the `deriving` clause declares this new type as a member of two type classes:

- the `Eq` type class, allowing us to use the equality operator "`==`" between values of this type, and
- the `Bits` type class, allowing us to store these values in registers.

We could have used separate `instance` statements to do this, but then we'd have to specify explicitly the definitions of the "`==`" operator (for the `Eq` class) and the `pack` and `unpack` operators (for the `Bits`) class. Using a `deriving` clause tells the compiler to do this for us in the "obvious" way.

For some types the notion of equality is not straightforward. For example, equality of two hash tables may just require that both tables contain the same key-and-value pairs, not that they are stored in the same order (*e.g.,* if the order of insertion was different, the storage order may be different). In this case, we would need to define the algorithm for the "`==`" operator explicitly, in a way that ignores storage order.

End of Side Comment

The next line in the code:

```
module sysTL(TL);
```

declares that `sysTL` is a variable with an interface of type `TL`. The type of the module is *parameterized* by the type of its interface.

The rest of the file, until the line

```
endmodule: sysTL
```

gives the definition of this module. It consists of a series of statements, including definitions of the module's interface methods and its rules.

Consider the statements:

```
Reg#(TLstates) state <- mkReg(RedAfterW);
```

This could be written as two logically separate statements:

```
Reg#(TLstates) state();
mkReg#(RedAfterW) the_state(state);
```

but the abbreviation allows us to combine them in a single statement. The longer form corresponds to the SystemVerilog syntax for interface and module instantiation. The first line declares and (by virtue of the "()" at the end) instantiates an interface of type `Reg#(TLstates`. The second line instantiates a `mkReg` module with parameter (the initial value to be stored in the register) `RedAfterW`; the instantiation is named `the_state`, and it is hooked up to the `state` interface declared in the previous line. In the shorter form (which most users prefer) the `state` interface is similarly declared, and initialized with the result interface obtained by instantiating the `mkReg` module; the name of the instantiation will usually be the same as the interface name.

Side Comment

In general the use of the symbol "`<-`" indicates that the expression on its right evaluates to a value which specifies something to be "done"; doing it will result in a side effect (in this case the instantiation of the module) and returns a result (in this case the interface value) to be assigned to the thing on the left of the arrow.

End of Side Comment

`Reg` is a *polymorphic* type: it is parameterized by the type of the register contents; in this case, the register will contain values of the type `TLstates`. The application of `mkReg` creates (instantiates) a new register object initialized to contain the value `RedAfterW`. This choice of initial value is arbitrary—there is nothing in the original problem statement that specifies the initial state of the system.

The remainder of the module definition contains a collection of *rules* that specify state changes. For example, the rule:

```
rule fromGreenNS (state == GreenNS);
   state <= AmberNS;
endrule: fromGreenNS
```

says that when the current state is green for North-South traffic, the next state is amber for North-South traffic. The name `fromGreenNS` has no semantic significance. It is present purely to help in simulation/debugging: the execution engine uses these names to describe which rules are enabled, which rules are executed, and so on.

Within the rule, the expression:

```
   state <= AmberNS;
```

is actually a syntactic shorthand for:

```
state._write(AmberNS);
```

that is, it applies the `_write` method in the `state` object (which is a register) to the argument `AmberNS`.

Usually a module definition concludes with the definition of the methods of its interface; but in this case the interface is empty, and there aren't any methods to be defined.

5.1 The Test Program

Since our design has no interface, there is not much that a test program can do to test its correctness. All it does is instantiate the device under test ("DUT") and let it run for a while. The reader is invited to run the program to produce a waves file, and examine this to check that things are working (rules are firing, and the state is changing) as expected.

5.2 Version TL0a: add lamp outputs to the interface

Having modeled the state transitions, we add the actual lamp outputs to the interface. The interface type now has declarations like:

```
method Bool lampRedNS();
method Bool lampAmberNS();
method Bool lampGreenNS();
```

specifying a set of red, amber and green outputs for the lamps on the north-south roads. We assume a `True` output means the lamp is switched on, and a `False` output means that the lamp is switched off. There are similar definitions for the eastbound road and the westbound road.

These methods are defined in the interface section of the module expression as simple functions of the current state:

```
method lampRedNS() = (!(state == GreenNS || state == AmberNS));
method lampAmberNS() = (state == AmberNS);
method lampGreenNS() = (state == GreenNS);
```

The red lamps on the north-south roads are switched on *unless* that road has a green or amber. The amber lamps on the north-south road are switched on only in the `AmberNS` state, and similarly for the green lamps. There are corresponding definitions for the lamps controlling eastbound and westbound traffic.

The test program for this modification is now a little more elaborate, since the DUT is now producing output controlling the lamps. The test program has a rule which fires any time any of the lamps goes on or off, and prints a line saying what changed.

5.3 Version TL0b: a small variation collapsing the red states

Observe that in all three red states we will be driving the same lamps (all the red lamps) and so we might want to collapse them into a single red state. However, the three red states have different next-green states. So, in this version, we collapse all reds into a single `AllRed` state, but before going into this state we need to remember the next green to follow. Later on, this collapsing-into-one will turn out to be even more useful because we will then have a single state in which to decide whether to insert a pedestrian sequence, instead of replicating it for each of the reds.

Here is the new type definition for the states:

```
typedef enum {
  AllRed,
  GreenNS, AmberNS,
  GreenE, AmberE,
  GreenW, AmberW} TLstates deriving (Eq, Bits);
```

In the module definition, we declare and allocate another register to remember the next green after `AllRed`:

```
Reg#(TLstates) next_green <-   mkReg(GreenNS);
```

The choice of initial value `GreenNS` is arbitrary (it implies that the first green will be north-south) since the original problem statement doesn't say anything about initialization. A rule like this:

```
rule fromAmberNS (state == AmberNS);
   state <= AllRed;
   next_green <= GreenE;
endrule: fromAmberNS
```

makes the transition from amber to red, but records the green that should follow the red. And this rule:

```
rule fromAllRed (state == AllRed);
   state <= next_green;
endrule: fromAllRed
```

implements the transition from red to the remembered green.
In the first rule above, the two lines of the body:

```
   state <= AllRed;
   next_green <= GreenE;
```

specify a composite action consisting of the two sub-actions:

```
   state <= AllRed;
```

and

```
   next_green <= GreenE;
```

Although the body looks similar to a block of statements in C, there are some fundamental differences. There is *no* temporal ordering between these two sub-actions. They are performed simultaneously as one atomic step. The two sub-actions could be swapped in the program text with no change in meaning. This principle applies to *all* sub-actions within a rule. For example, a rule may contain a method-call, which in turn may contains more sub-actions and method-calls which, in turn, may contain more sub-actions and method-calls, and so on—the entire collection of sub-actions within the purview of a rule are executed simultaneously as a single atomic step. A further nuance is that all expressions in the rule are evaluated simultaneously, before any updates are performed. A common trap for the programmer is to misunderstand the following:

```
rule inc_x (x == 10);
   x <= x + 1;
   y <= x;
endrule
```

In a sequential language such as C, one expects the variable `x` to be incremented to 11, and then the variable `y` gets this new value 11. In Bluespec, because the statements are simultaneous and atomic, both right-hand side expressions are evaluated simultaneously and both variables are updated simultaneously. Thus, `y` will get the value 10.

6 Version TL1: adding the time delays

In this version we add timing, and we arrange things so that each state is maintained for the required number of seconds. In this version we shall assume that the clock ticks just once per second—later we shall adapt the design to accommodate a more realistic clock speed.

The line:

```
typedef UInt#(5) Time32;
```

says that "`Time32`" is a synonym for the type "`UInt#(5)`", which comprises unsigned integers that can be represented in 5 bits. Our counter will be able to count durations up to $2^5 = 32$ seconds. The Bluespec compiler will check statically that we do not try to compare this value with time periods that are wider than 5 bits.

In the module definition, the lines:

```
Reg#(Time32) secs <- mkReg(0);
```

declares and allocates the counter (5-bit register), with initial value 0.

The lines:

```
Time32 allRedDelay = 2;
Time32 amberDelay = 4;
Time32 ns_green_delay = 20;
Time32 ew_green_delay = 10;
```

essentially define some variables to be used as symbolic constants for the required time delays. In each line, the first word declares the type of the variable, and the final part binds a value to the variable.

The rules which change `state` (thereby also changing the state of the lamps) each set the `secs` register back to zero. If none of these rules fires, we wish for the counter simply to be incremented. The incrementing rule (`inc_sec`) cannot fire in the same cycle as any of the others (because all the rules both read and write the `secs` register), so it is necessary to specify that the state-changing rules take precedence over the incrementing one—that is to say, if any of the state-changing rules fires, then the execution of the incrementing rule is pre-empted. This is done by the "preempts" attributes attached to each of the state-changing rules.

Note the occurrence of the "+1"'s in the conditions of the rules. This is to ensure that the state changes happen on the correct cycle. At each change, the counter is reset to 0; so n cycles later it achieves the value n. We wish the state to change when the counter equals the required delay; so we must decide to fire the state-changing rule when the counter is one less than this.

6.1 Version TL1a: an alternative way of specifying pre-emption

In this version we show a slightly different way of specifying the precedence of the state-changing rules over the incrementing rule. Previously, the

```
rule
   ...
endrule
```

construct always appeared as part of a module body; it defined the rule, and made it part of the behavior of the module. Here, instead, we define one or more rules (using the same syntax as before) within a set of

```
rules
   ...
endrules
```

brackets (note the plurals). The result is a value of type `Rules`, and it is not yet part of the behavior of any module. Such values may be manipulated just like any other values—they may be passed as arguments or results of functions, made elements of arrays and so on. Only when a value `r` of type `Rules` is used in the special statement

```
addRules(r);
```

does it become part of the behavior of the module containing that statement.

Side Comment

So, for example, the fragment

```
rule foo (x != 0);
   x <= x - 1;
endrule
```

is precisely equivalent to

```
Rules r =
  (rules
      rule foo (x != 0);
         x <= x - 1;
      endrule
   endrules);
addRule(r);
```

provided only that the definition of `r` does not clash with any other use of that name.

End of Side Comment

In the present example, we define two values of type `Rules`: the value `low_priority_rule` contains the incrementing rule, and `high_priority_rules` contains all the state-changing rules. The Prelude function `rJoinPreempts` combines these two values into a single value of the same type, in such a way that all the rules which are part of its first argument pre-empt any which are part of the second. The value produced as the result of this function is added to the behavior of the module by `addRules`, as described above.

Actually, in an example as simple as this one, it is probably preferable not to use this extra apparatus, but to use the technique shown in Version TL1. The techniques of Version TL1a, however, prove particularly useful in designs whose size is specified by a a parameter which might affect the number of rules: then it is difficult to specify precedence by attributes

using the explicit names of the rules (perhaps because rules might be generated inside a loop or a function which gives each of them the same name). We use this technique again in the final version of this design, below, to illustrate this point.

7 Version TL2: adding the pedestrian request button and pedestrian lamps

So far, the circuit we have described has no input interfaces—it is a free-running system just executing a particular sequence of state transitions, and its external outputs control the lamps. Now, we add an external input: the pedestrian button, and more external outputs to control the pedestrian lamps. (Physically there are four input buttons, one at each corner of the intersection, but since the whole intersection goes into pedestrian mode when any of the buttons are pushed, we assume the four buttons are *or*'ed together externally, so there is only one input to our controller.)

Here are the new interface methods:

```
interface TL;
   method Action ped_button_push();
   ...

   method Bool lampRedPed();
   method Bool lampAmberPed();
   method Bool lampGreenPed();
endinterface: TL
```

The three new lamp outputs are similar to the earlier lamp outputs. The one new input has type `Action`. We extend the `TLstates` type to have two new states for the pedestrian green and amber:

```
typedef enum {
   AllRed,
   GreenNS, AmberNS,
   GreenE, AmberE,
   GreenW, AmberW,
   GreenPed, AmberPed} TLstates deriving (Eq, Bits);
```

In the module definition we have a new state variable:

```
Reg#(Bool) ped_button_pushed <- mkReg(False);
```

which is just a register remembering whether or not the button has been pushed. It is set when the button is pushed, and reset by the system when is has responded with a pedestrian sequence. We have arbitrarily chosen its initial value to be `False`.

We introduce two variables used as symbolic constants for the delays in the pedestrian sequence:

```
Time32 pedGreenDelay = 10;
Time32 pedAmberDelay = 6;
```

In the interface section of the module, we specify the lamp outputs like the previous lamp outputs, and the action that is performed when the pedestrian button is pushed:

```
method Action ped_button_push();
   ped_button_pushed <= True;
endmethod: ped_button_push

method lampRedPed() = (!(state == GreenPed || state == AmberPed));
method lampAmberPed() = (state == AmberPed);
method lampGreenPed() = (state == GreenPed);
```

The input action simply sets the Boolean register `ped_button_pushed` to the value `True`. Note that if the button is pushed again while this register is set, it will have no further effect.

Turning to the rules section, the only interesting new rule arising from this change is:

```
rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
   state <= (ped_button_pushed ? GreenPed : next_green);
   secs <= 0;
   ped_button_pushed <= False;
endrule: fromAllRed
```

At the end of the `AllRed` period, if the pedestrian button had been pressed, we transition into pedestrian green; otherwise we transition into the remembered next green. We also reset the register `ped_button_pushed` getting it ready to accept another pedestrian request. (The problem statement did not specify what should happen if the pedestrian button is pushed again during a pedestrian sequence. In this design, it will start another pedestrian sequence after the current one.)

As before, when transitioning out of an amber, we set `next_green` appropriately, except that when we transition out of `AmberPed` we leave the `next_green` untouched.

7.1 A faster clock

For this version we have also taken the opportunity to allow a more realistic clock cycle time, instead of the 1Hz previously used. We use a second counter which is decremented once per cycle and reaches zero once per second, when it triggers an increment of the original seconds counter `secs`.

This involves a new datatype `CtrSize` for the new counter `cycle_ctr`, and a new constant of that type (`clocks_per_sec`) to reset the register each second. (The chosen value for this constant, 100, is of course still unrealistic; but it can easily be changed to something more sensible, and in the meantime it keeps the duration of simulation runs within tolerable limits.)

There is a new rule, `dec_cycle_ctr`, to decrement the new counter; and the rule which increments the seconds counter now fires only when the new counter, which it resets, has reached zero. Notice that, now that cycles are no longer the same as seconds, and the state

changes on the cycle (not the second) after the seconds counter has reached the trigger point, there is no need now for the "+1"'s in the state-changing rules' conditions.

7.2 The test program

The rule `inc_ctr` in this and subsequent test programs now contains conditional actions which cause input to the dut on particular cycles. The reader is invited to experiment with other inputs, and to check that the behavior is as desired.

7.3 Version TL2a: using a common "next state" function

In this version we do not add any new behavior; we simply introduce a local function definition to make the code a bit more perspicuous. Inside the module expression, we define the following function that captures the common activity of transitioning to a next state and reseting the timer to zero:

```
function Action next_state(TLstates ns);
   action
      state <= ns;
      secs <= 0;
   endaction
endfunction: next_state
```

The first line says that `next_state` is a function, with one argument of type `TLstates`, producing a result of type `Action`; this action is specified by the function body.

The function is used, for example, in this rule:

```
rule fromGreenPed (state == GreenPed && secs >= pedGreenDelay);
   next_state(AmberPed);
endrule: fromGreenPed
```

where it is applied to the argument `AmberPed`. The action produced by the function call is taken as the body of the rule.

8 Version TL3: adding magnetic sensors

In this version we add the magnetic sensors in the roadway that detect incoming cars, and the second intelligent behavior (skip a sequence for which there are no cars waiting). First, we add four new inputs to the interface of the circuit:

```
interface
   (* always_enabled *)
    method Action set_car_state_N(Bool x);
   (* always_enabled *)
    method Action set_car_state_S(Bool x);
```

```
   (* always_enabled *)
    method Action set_car_state_E(Bool x);
   (* always_enabled *)
    method Action set_car_state_W(Bool x);
   ...
endinterface
```

The sensor input could be encoded in several ways; for example:

- a voltage level (car present/absent)
- a pulse every time the sensor voltage changes
- a pulse plus a value specifying the sensor voltage, generated periodically or whenever it changes
- *etc.*

This is a design choice. If we had omitted the "`always_enabled`" attributes above, we would be making the third choice—the *enable* signals would supply the pulses which would trigger the sampling of the values. But the attributes cause this wire not to be present, and the voltage is sampled on every clock: thus we have effectively made the first choice. (The third choice might be slightly easier to use in a simulator—we would just need to perform an action every time we want to change the state of the sensor).

In the module definition, we add some registers to remember each of the interface actions. Since our traffic light system always does northbound and southbound traffic together, we might think we need only three registers, since we only need to remember whether the northbound *or* the southbound sensor was activated. It is instructive to see what happens if we do this. We add the new registers:

```
Reg#(Bool) car_present_NS <- mkReg(True);
Reg#(Bool) car_present_E <- mkReg(True);
Reg#(Bool) car_present_W <- mkReg(True);
```

We initialize these registers to `True` (arbitrary choice: assume that we initialize the system with cars present on all four incoming roads).

Next, we define a help-function that tells us, for a particular green state, which green would normally be next (after the amber and red, and assuming no pedestrian request).

```
function TLstates green_seq(TLstates x);
   case (x)
      GreenNS: return (GreenE);
      GreenE:  return (GreenW);
      GreenW:  return (GreenNS);
   endcase
endfunction
```

Note that this function will give an undefined result if applied to a state other than the three green states listed. This is OK because we are only ever going to apply it to one of those three states.

We also define a help-function that tells us, for a particular green state, whether a car is currently present and waiting for that green.

```
function Bool car_present(TLstates x);
   case (x)
      GreenNS: return (car_present_NS);
      GreenE:  return (car_present_E);
      GreenW:  return (car_present_W);
   endcase
endfunction
```

In the interface section of the module definition, we define the four new interface elements:

```
method Action set_car_state_N(b); car_present_NS <= b; endmethod
method Action set_car_state_S(b); car_present_NS <= b; endmethod
method Action set_car_state_E(b); car_present_E <= b; endmethod
method Action set_car_state_W(b); car_present_W <= b; endmethod
```

When each action is performed, we remember the value passed in, in the corresponding register. Note that the "N" and the "S" methods each write to the "`car_present_NS`" register.

In the rules section, most of the action is concentrated in the all-red state, when we are just about to leave that state.

```
rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
   if (ped_button_pushed)
      action
         ped_button_pushed <= False;
         next_state(GreenPed);
      endaction
   else if (car_present(next_green))
      next_state(next_green);
   else if (car_present(green_seq(next_green)))
      next_state(green_seq(next_green));
   else if (car_present(green_seq(green_seq(next_green))))
      next_state(green_seq(green_seq(next_green)));
   else
      noAction;
endrule: fromAllRed
```

Suppose `next_green` was g_1, and the logically following road-traffic greens are g_2 and g_3, respectively (*e.g.,* g_1 may be GreenW, in which case g_2 is GreenNS, and g_3 is GreenE). The first part of the `if` is as before: if the pedestrian button is pushed we transition into the green pedestrian state. The rest of the (nested) conditional expression specifies that if there is a car waiting for g_1 we transition to g_1, otherwise if there is a car waiting for g_2 we transition to g_2, otherwise if there is a car waiting for g_3 we transition to g_3, and otherwise we stay where we are, in all-red. (Note that we might think that we could emphasise that we are staying in all-red by saying "`next_state(AllRed)`" instead of "`noAction`"; but this would also have the effect of resetting the seconds counter, which we do not want.)

8.0.1 Debugging this version

Let us attempt to compile this version and its testbench to Verilog RTL. The dut itself compiles, but we get the warning

```
Warning: "TL3x.bsv", line 52, column 8: (G0036)
  Rule "set_car_state_N" will appear to fire before "set_car_state_S" when
  both fire in the same clock cycle, affecting:
    calls to car_present_NS.write vs. car_present_NS.write
```

This should worry us: it says that when rules containing the methods `set_car_state_N` and `set_car_state_S` fire in the same cycle, the former will always appear to happen first. Since (as the warning says) both of them write to the register `car_present_NS`, the first write will always be overwritten and not observed. Since these two methods are both marked as always enabled, this will happen on every cycle, so the value of the "N" detector will never be recorded. Sure enough, if we examine the generated Verilog for this design, we find in the initial comment

```
// set_car_state_N_x              I     1 unused
```

indicating that this input is completely ignored.
Compilation of the testbench actually fails, giving the error message

```
Error: "TbTL3x.bsv", line 40, column 9: (G0004)
  Rule RL_detect_cars uses methods that conflict in parallel:
    dut.set_car_state_N carN.read
  and
    dut.set_car_state_S carS.read
```

This happens because the testbench includes a rule, `detect_cars`, which attempts to call all four `set_ca_state` methods simultaneously. But, according to the information from the previous compilation, the "N" one must happen *before* the "S" one, so this is impossible (it would imply writing two possibly different values to the same register simultaneously).

The problem is quite simple to correct. We abandon the attempt to use just one register for North-South traffic, and define two of them:

```
Reg#(Bool) car_present_N <- mkReg(True);
Reg#(Bool) car_present_S <- mkReg(True);
```

We change the two methods that wrote to the original single register so that they now each write to their own one. Finally we define

```
Bool car_present_NS = car_present_N || car_present_S;
```

which provides the `car_present_NS` value, used by the rest of the design, in terms of the two new registers. (Note that this is like Verilog's "continous assign": the two values on the right are dynamically varying, and so is the result.) The rest of the design is unchanged.

8.1 Version TL3a: adding the third intelligent behavior

In this version, we add the third intelligent behavior: when we are in a green (and this would only happen because a car was waiting for it) and we now detect that there are no more cars waiting for that green, then we immediately cut short the green and go into amber, rather than waiting for the entire green delay time. This is simply encoded by changing the transitions out of green, such as:

```
rule fromGreenNS (state == GreenNS &&
                  (secs >= ns_green_delay || !car_present_NS));
   next_state(AmberNS);
endrule: fromGreenNS
```

that is, we transition out of the state when either the time has expired or no car is present for that flow.

9 Version TL4: common rule-building function calls

We observe that in the previous version certain groups of rules are similar in structure. For example, the following rule specification:

```
rule fromGreenNS (state == GreenNS &&
                  (secs >= ns_green_delay || !car_present_NS));
   next_state(AmberNS);
endrule: fromGreenNS
```

is broadly repeated, except in the details, in the rules `fromGreenE` and `fromGreenW`. We can capture the common structure in a function definition and abstract out the detailed differences into parameters. The function returns a result of type `Rules`:

```
function Rules make_from_green_rule(TLstates green_state,
                                    Time32 delay,
                                    Bool car_is_present,
                                    TLstates ns);
   return (rules
             rule from_green (state == green_state &&
                              (secs >= delay ||
                               !car_is_present));
                next_state(ns);
             endrule
           endrules);
endfunction
```

We define a similar function for transitions from amber. We then declare an array containing elements of type `Rules`:

```
Rules hprs[7];
```

and initialize most of its elements (element 0 is dealt with as a special case) by calls of these two functions:

```
hprs[1] = make_from_green_rule(GreenNS, ns_green_delay, car_present_NS, AmberNS);
hprs[2] = make_from_amber_rule(AmberNS, GreenE);
hprs[3] = make_from_green_rule(GreenE, ew_green_delay, car_present_E, AmberE);
hprs[4] = make_from_amber_rule(AmberE, GreenW);
hprs[5] = make_from_green_rule(GreenW, ew_green_delay, car_present_W, AmberW);
hprs[6] = make_from_amber_rule(AmberW, GreenNS);
```

We finally define `high_priority_rules`, using a `for`-loop, as the result of joining all these elements together; and we combine this with the `low_priority_rule` using `preempts`, as in Version TL1a.

10 Conclusion

The primary purpose of this tutorial is to develop some familiarity with basic Bluespec notation and concepts: simple types, structure of a module, simple interfaces, registers, state within a module, rules, combining rules, simple local function definitions and the "look and feel" of Bluespec programs. This tutorial does not address the question of composition and interaction between modules, which is the next important topic for the Bluespec programmer and is essential for effective "programming-in-the-large". The current example only defines a single module and it only uses primitive modules (in particular, registers).

A The source files

A.1 The source file TL0.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL0;

// Simple model of a traffic light
// (modeled after the light at the intersection of Rte 16 and Broadway
//  on the border between Arlington, MA and Somerville, MA)

// Version 0: just model the normal cycle of states

// An empty interface:
interface TL;
endinterface: TL

typedef enum {
   GreenNS, AmberNS, RedAfterNS,
   GreenE, AmberE, RedAfterE,
   GreenW, AmberW, RedAfterW} TLstates deriving (Eq, Bits);

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(RedAfterW);

   rule fromGreenNS (state == GreenNS);
      state <= AmberNS;
   endrule: fromGreenNS

   rule fromAmberNS (state == AmberNS);
      state <= RedAfterNS;
   endrule: fromAmberNS

   rule fromRedAfterNS (state == RedAfterNS);
      state <= GreenE;
   endrule: fromRedAfterNS

   rule fromGreenE (state == GreenE);
      state <= AmberE;
   endrule: fromGreenE

   rule fromAmberE (state == AmberE);
      state <= RedAfterE;
```

```
   endrule: fromAmberE

   rule fromRedAfterE (state == RedAfterE);
      state <= GreenW;
   endrule: fromRedAfterE

   rule fromGreenW (state == GreenW);
      state <= AmberW;
   endrule: fromGreenW

   rule fromAmberW (state == AmberW);
      state <= RedAfterW;
   endrule: fromAmberW

   rule fromRedAfterW (state == RedAfterW);
      state <= GreenNS;
   endrule: fromRedAfterW

endmodule: sysTL

endpackage: TL0
```

A.2 The source file TL0a.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL0a;

// Simple model of a traffic light
// (modeled after the light at the intersection of Rte 16 and Broadway
//  on the border between Arlington, MA and Somerville, MA)

// Version 0a: Add interface outputs for the lamps

(* always_ready *)
interface TL;
   method Bool lampRedNS();
   method Bool lampAmberNS();
   method Bool lampGreenNS();

   method Bool lampRedE();
   method Bool lampAmberE();
   method Bool lampGreenE();

   method Bool lampRedW();
   method Bool lampAmberW();
   method Bool lampGreenW();
endinterface: TL

typedef enum {
  GreenNS, AmberNS, RedAfterNS,
  GreenE, AmberE, RedAfterE,
  GreenW, AmberW, RedAfterW} TLstates deriving (Eq, Bits);

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(RedAfterW);

   rule fromGreenNS (state == GreenNS);
      state <= AmberNS;
   endrule: fromGreenNS

   rule fromAmberNS (state == AmberNS);
      state <= RedAfterNS;
   endrule: fromAmberNS

   rule fromRedAfterNS (state == RedAfterNS);
      state <= GreenE;
```

```
   endrule: fromRedAfterNS

   rule fromGreenE (state == GreenE);
      state <= AmberE;
   endrule: fromGreenE

   rule fromAmberE (state == AmberE);
      state <= RedAfterE;
   endrule: fromAmberE

   rule fromRedAfterE (state == RedAfterE);
      state <= GreenW;
   endrule: fromRedAfterE

   rule fromGreenW (state == GreenW);
      state <= AmberW;
   endrule: fromGreenW

   rule fromAmberW (state == AmberW);
      state <= RedAfterW;
   endrule: fromAmberW

   rule fromRedAfterW (state == RedAfterW);
      state <= GreenNS;
   endrule: fromRedAfterW

   method lampRedNS() = (!(state == GreenNS || state == AmberNS));
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = (!(state == GreenE || state == AmberE));
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = (!(state == GreenW || state == AmberW));
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
endmodule: sysTL

endpackage
```

A.3 The source file TL0b.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL0b;

// Simple model of a traffic light
// (modeled after the light at the intersection of Rte 16 and Broadway
//  on the border between Arlington, MA and Somerville, MA)

// Version 0b: collapsed all the Reds into one, just remembering the
//             next green.  Retained separate Ambers, because each
//             Amber controls a different set of lights

(* always_ready *)
interface TL;
   method Bool lampRedNS();
   method Bool lampAmberNS();
   method Bool lampGreenNS();

   method Bool lampRedE();
   method Bool lampAmberE();
   method Bool lampGreenE();

   method Bool lampRedW();
   method Bool lampAmberW();
   method Bool lampGreenW();
endinterface: TL

typedef enum {
  AllRed,
  GreenNS, AmberNS,
  GreenE, AmberE,
  GreenW, AmberW} TLstates deriving (Eq, Bits);

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <-   mkReg(GreenNS);

   rule fromAllRed (state == AllRed);
      state <= next_green;
   endrule: fromAllRed

   rule fromGreenNS (state == GreenNS);
      state <= AmberNS;
```

```
   endrule: fromGreenNS

   rule fromAmberNS (state == AmberNS);
      state <= AllRed;
      next_green <= GreenE;
   endrule: fromAmberNS

   rule fromGreenE (state == GreenE);
      state <= AmberE;
   endrule: fromGreenE

   rule fromAmberE (state == AmberE);
      state <= AllRed;
      next_green <= GreenW;
   endrule: fromAmberE

   rule fromGreenW (state == GreenW);
      state <= AmberW;
   endrule: fromGreenW

   rule fromAmberW (state == AmberW);
      state <= AllRed;
      next_green <= GreenNS;
   endrule: fromAmberW

   method lampRedNS() = (!(state == GreenNS || state == AmberNS));
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = (!(state == GreenE || state == AmberE));
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = (!(state == GreenW || state == AmberW));
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
endmodule: sysTL

endpackage
```

A.4 The source file TL1.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL1;

// Simple model of a traffic light

// Version 1: add a specific time delay for each state

(* always_ready *)
interface TL;
   method Bool lampRedNS();
   method Bool lampAmberNS();
   method Bool lampGreenNS();

   method Bool lampRedE();
   method Bool lampAmberE();
   method Bool lampGreenE();

   method Bool lampRedW();
   method Bool lampAmberW();
   method Bool lampGreenW();
endinterface: TL

typedef enum {
   AllRed,
   GreenNS, AmberNS,
   GreenE, AmberE,
   GreenW, AmberW} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
   Reg#(Time32) secs <- mkReg(0);

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;

   // The default rule, which fires (every second) only if no other can:
   rule inc_sec;
```

```
   secs <= secs + 1;
endrule: inc_sec

(* preempts = "fromAllRed, inc_sec" *)
rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
   state <= next_green;
   secs <= 0;
endrule: fromAllRed

(* preempts = "fromGreenNS, inc_sec" *)
rule fromGreenNS (state == GreenNS && secs + 1 >= ns_green_delay);
   state <= AmberNS;
   secs <= 0;
endrule: fromGreenNS

(* preempts = "fromAmberNS, inc_sec" *)
rule fromAmberNS (state == AmberNS && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
   next_green <= GreenE;
endrule: fromAmberNS

(* preempts = "fromGreenE, inc_sec" *)
rule fromGreenE (state == GreenE && secs + 1 >= ew_green_delay);
   state <= AmberE;
   secs <= 0;
endrule: fromGreenE

(* preempts = "fromAmberE, inc_sec" *)
rule fromAmberE (state == AmberE && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
   next_green <= GreenW;
endrule: fromAmberE

(* preempts = "fromGreenW, inc_sec" *)
rule fromGreenW (state == GreenW && secs + 1 >= ew_green_delay);
   state <= AmberW;
   secs <= 0;
endrule: fromGreenW

(* preempts = "fromAmberW, inc_sec" *)
rule fromAmberW (state == AmberW && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
```

```
      next_green <= GreenNS;
   endrule: fromAmberW

   method lampRedNS() = (!(state == GreenNS || state == AmberNS));
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = (!(state == GreenE || state == AmberE));
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = (!(state == GreenW || state == AmberW));
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
endmodule: sysTL

endpackage
```

A.5 The source file TL1a.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL1a;

// Simple model of a traffic light

// Version 1a: alternative notation for pre-emption of rules

(* always_ready *)
interface TL;
   method Bool lampRedNS();
   method Bool lampAmberNS();
   method Bool lampGreenNS();

   method Bool lampRedE();
   method Bool lampAmberE();
   method Bool lampGreenE();

   method Bool lampRedW();
   method Bool lampAmberW();
   method Bool lampGreenW();
endinterface: TL

typedef enum {
   AllRed,
   GreenNS, AmberNS,
   GreenE, AmberE,
   GreenW, AmberW} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
   Reg#(Time32) secs <- mkReg(0);

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;

   // The default rule, which fires (every second) only if no other can:
   Rules low_priority_rule =
```

```
    (rules
rule inc_sec (True);
   secs <= secs + 1;
endrule: inc_sec
     endrules);

 Rules high_priority_rules =
    (rules
rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
   state <= next_green;
   secs <= 0;
endrule: fromAllRed

rule fromGreenNS (state == GreenNS && secs + 1 >= ns_green_delay);
   state <= AmberNS;
   secs <= 0;
endrule: fromGreenNS

rule fromAmberNS (state == AmberNS && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
   next_green <= GreenE;
endrule: fromAmberNS

rule fromGreenE (state == GreenE && secs + 1 >= ew_green_delay);
   state <= AmberE;
   secs <= 0;
endrule: fromGreenE

rule fromAmberE (state == AmberE && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
   next_green <= GreenW;
endrule: fromAmberE

rule fromGreenW (state == GreenW && secs + 1 >= ew_green_delay);
   state <= AmberW;
   secs <= 0;
endrule: fromGreenW

rule fromAmberW (state == AmberW && secs + 1 >= amberDelay);
   state <= AllRed;
   secs <= 0;
   next_green <= GreenNS;
endrule: fromAmberW
```

```
        endrules);

    addRules(rJoinPreempts(high_priority_rules, low_priority_rule));

    method lampRedNS() = (!(state == GreenNS || state == AmberNS));
    method lampAmberNS() = (state == AmberNS);
    method lampGreenNS() = (state == GreenNS);
    method lampRedE() = (!(state == GreenE || state == AmberE));
    method lampAmberE() = (state == AmberE);
    method lampGreenE() = (state == GreenE);
    method lampRedW() = (!(state == GreenW || state == AmberW));
    method lampAmberW() = (state == AmberW);
    method lampGreenW() = (state == GreenW);
endmodule: sysTL

endpackage
```

A.6 The source file TL2.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL2;

// Simple model of a traffic light

// Version 2: Add pedestrian request button and outputs for pedestrian lamps
//     When pedestrian request button is pressed,
//     wait for next transition out of Red
//     goto GreenPed      for pedGreenDelay  secs
//     goto AmberPed      for pedAmberDelay  secs
//     goto AllRed        for allRedDelay    secs
//     goto next green    (i.e., don't break the cycle)

(* always_ready *)
interface TL;
    method Action ped_button_push();

    method Bool lampRedNS();
    method Bool lampAmberNS();
    method Bool lampGreenNS();

    method Bool lampRedE();
    method Bool lampAmberE();
    method Bool lampGreenE();

    method Bool lampRedW();
    method Bool lampAmberW();
    method Bool lampGreenW();

    method Bool lampRedPed();
    method Bool lampAmberPed();
    method Bool lampGreenPed();
endinterface: TL

typedef enum {
    AllRed,
    GreenNS, AmberNS,
    GreenE, AmberE,
    GreenW, AmberW,
    GreenPed, AmberPed} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;
typedef UInt#(20) CtrSize;
```

```
(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
   Reg#(Time32) secs <- mkReg(0);
   Reg#(CtrSize) cycle_ctr <- mkReg(0);
   Reg#(Bool) ped_button_pushed <- mkReg(False);

   CtrSize clocks_per_sec = 100;

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;
   Time32 pedGreenDelay = 10;
   Time32 pedAmberDelay = 6;

   rule dec_cycle_ctr (cycle_ctr != 0);
      cycle_ctr <= cycle_ctr - 1;
   endrule

   // The default rule, which fires (every second) only if no other can:
   rule inc_sec (cycle_ctr == 0);
      secs <= secs + 1;
      cycle_ctr <= clocks_per_sec;
   endrule: inc_sec

   (* preempts = "fromAllRed, inc_sec" *)
   rule fromAllRed (state == AllRed && secs >= allRedDelay);
      state <= (ped_button_pushed ? GreenPed : next_green);
      secs <= 0;
      ped_button_pushed <= False;
   endrule: fromAllRed

   (* preempts = "fromGreenPed, inc_sec" *)
   rule fromGreenPed (state == GreenPed && secs >= pedGreenDelay);
      state <= AmberPed;
      secs <= 0;
   endrule: fromGreenPed

   (* preempts = "fromAmberPed, inc_sec" *)
   rule fromAmberPed (state == AmberPed && secs >= pedAmberDelay);
      state <= AllRed;
      secs <= 0;
```

```
endrule: fromAmberPed

(* preempts = "fromGreenNS, inc_sec" *)
rule fromGreenNS (state == GreenNS && secs >= ns_green_delay);
   state <= AmberNS;
   secs <= 0;
endrule: fromGreenNS

(* preempts = "fromAmberNS, inc_sec" *)
rule fromAmberNS (state == AmberNS && secs >= amberDelay);
   state <= AllRed;
   next_green <= GreenE;
   secs <= 0;
endrule: fromAmberNS

(* preempts = "fromGreenE, inc_sec" *)
rule fromGreenE (state == GreenE && secs >= ew_green_delay);
   state <= AmberE;
   secs <= 0;
endrule: fromGreenE

(* preempts = "fromAmberE, inc_sec" *)
rule fromAmberE (state == AmberE && secs >= amberDelay);
   state <= AllRed;
   next_green <= GreenW;
   secs <= 0;
endrule: fromAmberE

(* preempts = "fromGreenW, inc_sec" *)
rule fromGreenW (state == GreenW && secs >= ew_green_delay);
   state <= AmberW;
   secs <= 0;
endrule: fromGreenW

(* preempts = "fromAmberW, inc_sec" *)
rule fromAmberW (state == AmberW && secs >= amberDelay);
   state <= AllRed;
   next_green <= GreenNS;
   secs <= 0;
endrule: fromAmberW

method Action ped_button_push();
   ped_button_pushed <= True;
endmethod: ped_button_push
```

```
   method lampRedNS() = (!(state == GreenNS || state == AmberNS));
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = (!(state == GreenE || state == AmberE));
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = (!(state == GreenW || state == AmberW));
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
   method lampRedPed() = (!(state == GreenPed || state == AmberPed));
   method lampAmberPed() = (state == AmberPed);
   method lampGreenPed() = (state == GreenPed);
endmodule: sysTL
endpackage
```

A.7 The source file TL2a.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL2a;

// Simple model of a traffic light

// Version 2a: Use a "next_state" local definition in the module
//             for improved readability

(* always_ready *)
interface TL;
    method Action ped_button_push();

    method Bool lampRedNS();
    method Bool lampAmberNS();
    method Bool lampGreenNS();

    method Bool lampRedE();
    method Bool lampAmberE();
    method Bool lampGreenE();

    method Bool lampRedW();
    method Bool lampAmberW();
    method Bool lampGreenW();

    method Bool lampRedPed();
    method Bool lampAmberPed();
    method Bool lampGreenPed();
endinterface: TL

typedef enum {
    AllRed,
    GreenNS, AmberNS,
    GreenE, AmberE,
    GreenW, AmberW,
    GreenPed, AmberPed} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;
typedef UInt#(20) CtrSize;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
```

```
  Reg#(Time32) secs <- mkReg(0);
  Reg#(CtrSize) cycle_ctr <- mkReg(0);
  Reg#(Bool) ped_button_pushed <- mkReg(False);

  CtrSize clocks_per_sec = 100;

  Time32 allRedDelay = 2;
  Time32 amberDelay = 4;
  Time32 ns_green_delay = 20;
  Time32 ew_green_delay = 10;
  Time32 pedGreenDelay = 10;
  Time32 pedAmberDelay = 6;

  function Action next_state(TLstates ns);
     action
state <= ns;
secs <= 0;
     endaction
  endfunction: next_state

  rule dec_cycle_ctr (cycle_ctr != 0);
     cycle_ctr <= cycle_ctr - 1;
  endrule

  // The default rule, which fires (every second) only if no other can:
  rule inc_sec (cycle_ctr == 0);
     secs <= secs + 1;
     cycle_ctr <= clocks_per_sec;
  endrule: inc_sec

  (* preempts = "fromAllRed, inc_sec" *)
  rule fromAllRed (state == AllRed && secs >= allRedDelay);
     if (ped_button_pushed)
action
   ped_button_pushed <= False;
   next_state(GreenPed);
endaction
     else next_state(next_green);
  endrule: fromAllRed

  (* preempts = "fromGreenPed, inc_sec" *)
  rule fromGreenPed (state == GreenPed && secs >= pedGreenDelay);
     next_state(AmberPed);
  endrule: fromGreenPed
```

```
(* preempts = "fromAmberPed, inc_sec" *)
rule fromAmberPed (state == AmberPed && secs >= pedAmberDelay);
   next_state(AllRed);
endrule: fromAmberPed

(* preempts = "fromGreenNS, inc_sec" *)
rule fromGreenNS (state == GreenNS && secs >= ns_green_delay);
   next_state(AmberNS);
endrule: fromGreenNS

(* preempts = "fromAmberNS, inc_sec" *)
rule fromAmberNS (state == AmberNS && secs >= amberDelay);
   next_state(AllRed);
   next_green <= GreenE;
endrule: fromAmberNS

(* preempts = "fromGreenE, inc_sec" *)
rule fromGreenE (state == GreenE && secs >= ew_green_delay);
   next_state(AmberE);
endrule: fromGreenE

(* preempts = "fromAmberE, inc_sec" *)
rule fromAmberE (state == AmberE && secs >= amberDelay);
   next_state(AllRed);
   next_green <= GreenW;
endrule: fromAmberE

(* preempts = "fromGreenW, inc_sec" *)
rule fromGreenW (state == GreenW && secs >= ew_green_delay);
   next_state(AmberW);
endrule: fromGreenW

(* preempts = "fromAmberW, inc_sec" *)
rule fromAmberW (state == AmberW && secs >= amberDelay);
   next_state(AllRed);
   next_green <= GreenNS;
endrule: fromAmberW

method Action ped_button_push();
   ped_button_pushed <= True;
endmethod: ped_button_push

method lampRedNS() = (!(state == GreenNS || state == AmberNS));
method lampAmberNS() = (state == AmberNS);
method lampGreenNS() = (state == GreenNS);
```

```
   method lampRedE() = (!(state == GreenE || state == AmberE));
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = (!(state == GreenW || state == AmberW));
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
   method lampRedPed() = (!(state == GreenPed || state == AmberPed));
   method lampAmberPed() = (state == AmberPed);
   method lampGreenPed() = (state == GreenPed);
endmodule: sysTL
endpackage
```

A.8 The source file TL3.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL3;

// Simple model of a traffic light

// Version 3: Add magnetic car-detection sensors and short-cuts
//            to next demanded green.  If there is no demanded
//            green, stays in AllRed

(* always_ready *)
interface TL;
    method Action ped_button_push();

   (* always_enabled *)
    method Action set_car_state_N(Bool x);
   (* always_enabled *)
    method Action set_car_state_S(Bool x);
   (* always_enabled *)
    method Action set_car_state_E(Bool x);
   (* always_enabled *)
    method Action set_car_state_W(Bool x);

    method Bool lampRedNS();
    method Bool lampAmberNS();
    method Bool lampGreenNS();

    method Bool lampRedE();
    method Bool lampAmberE();
    method Bool lampGreenE();

    method Bool lampRedW();
    method Bool lampAmberW();
    method Bool lampGreenW();

    method Bool lampRedPed();
    method Bool lampAmberPed();
    method Bool lampGreenPed();
endinterface: TL

typedef enum {
    AllRed,
    GreenNS, AmberNS,
    GreenE, AmberE,
```

```
    GreenW, AmberW,
    GreenPed, AmberPed} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;
typedef UInt#(20) CtrSize;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
   Reg#(Time32) secs <- mkReg(0);
   Reg#(CtrSize) sec_ctr <- mkReg(0);

   Reg#(Bool) ped_button_pushed <- mkReg(False);

   Reg#(Bool) car_present_N <- mkReg(True);
   Reg#(Bool) car_present_S <- mkReg(True);
   Reg#(Bool) car_present_E <- mkReg(True);
   Reg#(Bool) car_present_W <- mkReg(True);

   Bool car_present_NS = car_present_N || car_present_S; // dynamically varying

   CtrSize clocks_per_sec = 100;

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;
   Time32 pedGreenDelay = 10;
   Time32 pedAmberDelay = 6;

   function Action next_state(TLstates ns);
      action
 state <= ns;
 secs <= 0;
      endaction
   endfunction

   function TLstates green_seq(TLstates x);
      case (x)
 GreenNS: return (GreenE);
 GreenE:  return (GreenW);
 GreenW:  return (GreenNS);
      endcase
   endfunction
```

```
  function Bool car_present(TLstates x);
     case (x)
GreenNS: return (car_present_NS);
GreenE:  return (car_present_E);
GreenW:  return (car_present_W);
     endcase
  endfunction

  rule dec_sec_ctr (sec_ctr != 0);
     sec_ctr <= sec_ctr - 1;
  endrule

  // The default rule, which fires (every second) only if no other can:
  rule inc_sec (sec_ctr == 0);
     secs <= secs + 1;
     sec_ctr <= clocks_per_sec;
  endrule: inc_sec

  (* preempts = "fromAllRed, inc_sec" *)
  rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
     if (ped_button_pushed)
action
   ped_button_pushed <= False;
   next_state(GreenPed);
endaction
     else if (car_present(next_green))
next_state(next_green);
     else if (car_present(green_seq(next_green)))
next_state(green_seq(next_green));
     else if (car_present(green_seq(green_seq(next_green))))
next_state(green_seq(green_seq(next_green)));
     else
noAction;
  endrule: fromAllRed

  (* preempts = "fromGreenPed, inc_sec" *)
  rule fromGreenPed (state == GreenPed && secs + 1 >= pedGreenDelay);
     next_state(AmberPed);
  endrule: fromGreenPed

  (* preempts = "fromAmberPed, inc_sec" *)
  rule fromAmberPed (state == AmberPed && secs + 1 >= pedAmberDelay);
     next_state(AllRed);
  endrule: fromAmberPed
```

```
(* preempts = "fromGreenNS, inc_sec" *)
rule fromGreenNS (state == GreenNS && secs + 1 >= ns_green_delay);
   next_state(AmberNS);
endrule: fromGreenNS

(* preempts = "fromAmberNS, inc_sec" *)
rule fromAmberNS (state == AmberNS && secs + 1 >= amberDelay);
   next_state(AllRed);
   next_green <= GreenE;
endrule: fromAmberNS

(* preempts = "fromGreenE, inc_sec" *)
rule fromGreenE (state == GreenE && secs + 1 >= ew_green_delay);
   next_state(AmberE);
endrule: fromGreenE

(* preempts = "fromAmberE, inc_sec" *)
rule fromAmberE (state == AmberE && secs + 1 >= amberDelay);
   next_state(AllRed);
   next_green <= GreenW;
endrule: fromAmberE

(* preempts = "fromGreenW, inc_sec" *)
rule fromGreenW (state == GreenW && secs + 1 >= ew_green_delay);
   next_state(AmberW);
endrule: fromGreenW

(* preempts = "fromAmberW, inc_sec" *)
rule fromAmberW (state == AmberW && secs + 1 >= amberDelay);
   next_state(AllRed);
   next_green <= GreenNS;
endrule: fromAmberW

method Action ped_button_push();
   ped_button_pushed <= True;
endmethod: ped_button_push

method Action set_car_state_N(b) ;
   car_present_N <= b;
endmethod: set_car_state_N

method Action set_car_state_S(b) ;
   car_present_S <= b;
endmethod: set_car_state_S
```

```
   method Action set_car_state_E(b) ;
      car_present_E <= b;
   endmethod: set_car_state_E

   method Action set_car_state_W(b) ;
      car_present_W <= b;
   endmethod: set_car_state_W

   method lampRedNS() = !(state == GreenNS || state == AmberNS);
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = !(state == GreenE || state == AmberE);
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = !(state == GreenW || state == AmberW);
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
   method lampRedPed() = !(state == GreenPed || state == AmberPed);
   method lampAmberPed() = (state == AmberPed);
   method lampGreenPed() = (state == GreenPed);
endmodule: sysTL
endpackage
```

A.9 The source file TL3a.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL3a;

// Simple model of a traffic light

// Version 3a: Add early termination of Green when no more cars
//             present for that flow

(* always_ready *)
interface TL;
    method Action ped_button_push();

   (* always_enabled *)
    method Action set_car_state_N(Bool x);
   (* always_enabled *)
    method Action set_car_state_S(Bool x);
   (* always_enabled *)
    method Action set_car_state_E(Bool x);
   (* always_enabled *)
    method Action set_car_state_W(Bool x);

    method Bool lampRedNS();
    method Bool lampAmberNS();
    method Bool lampGreenNS();

    method Bool lampRedE();
    method Bool lampAmberE();
    method Bool lampGreenE();

    method Bool lampRedW();
    method Bool lampAmberW();
    method Bool lampGreenW();

    method Bool lampRedPed();
    method Bool lampAmberPed();
    method Bool lampGreenPed();
endinterface: TL

typedef enum {
    AllRed,
    GreenNS, AmberNS,
    GreenE, AmberE,
    GreenW, AmberW,
```

```
    GreenPed, AmberPed} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;
typedef UInt#(20) CtrSize;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);
   Reg#(Time32) secs <- mkReg(0);
   Reg#(CtrSize) sec_ctr <- mkReg(0);

   Reg#(Bool) ped_button_pushed <- mkReg(False);

   Reg#(Bool) car_present_N <- mkReg(True);
   Reg#(Bool) car_present_S <- mkReg(True);
   Reg#(Bool) car_present_E <- mkReg(True);
   Reg#(Bool) car_present_W <- mkReg(True);

   Bool car_present_NS = car_present_N || car_present_S; // dynamically varying

   CtrSize clocks_per_sec = 100;

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;
   Time32 pedGreenDelay = 10;
   Time32 pedAmberDelay = 6;

   function Action next_state(TLstates ns);
      action
 state <= ns;
 secs <= 0;
      endaction
   endfunction

   function TLstates green_seq(TLstates x);
      case (x)
 GreenNS: return (GreenE);
 GreenE:  return (GreenW);
 GreenW:  return (GreenNS);
      endcase
   endfunction
```

```
  function Bool car_present(TLstates x);
     case (x)
GreenNS: return (car_present_NS);
GreenE:  return (car_present_E);
GreenW:  return (car_present_W);
     endcase
  endfunction

  rule dec_sec_ctr (sec_ctr != 0);
     sec_ctr <= sec_ctr - 1;
  endrule

  // The default rule, which fires (every second) only if no other can:
  rule inc_sec (sec_ctr == 0);
     secs <= secs + 1;
     sec_ctr <= clocks_per_sec;
  endrule: inc_sec

  (* preempts = "fromAllRed, inc_sec" *)
  rule fromAllRed (state == AllRed && secs >= allRedDelay + 1);
     if (ped_button_pushed)
action
   ped_button_pushed <= False;
   next_state(GreenPed);
endaction
     else if (car_present(next_green))
next_state(next_green);
     else if (car_present(green_seq(next_green)))
next_state(green_seq(next_green));
     else if (car_present(green_seq(green_seq(next_green))))
next_state(green_seq(green_seq(next_green)));
     else
//  next_state(AllRed);
noAction;
  endrule: fromAllRed

  (* preempts = "fromGreenPed, inc_sec" *)
  rule fromGreenPed (state == GreenPed && secs >= pedGreenDelay + 1);
     next_state(AmberPed);
  endrule: fromGreenPed

  (* preempts = "fromAmberPed, inc_sec" *)
  rule fromAmberPed (state == AmberPed && secs >= pedAmberDelay + 1);
     next_state(AllRed);
  endrule: fromAmberPed
```

```
(* preempts = "fromGreenNS, inc_sec" *)
rule fromGreenNS (state == GreenNS && (secs >= ns_green_delay + 1 || !car_present_NS));
   next_state(AmberNS);
endrule: fromGreenNS

(* preempts = "fromAmberNS, inc_sec" *)
rule fromAmberNS (state == AmberNS && secs >= amberDelay + 1);
   next_state(AllRed);
   next_green <= GreenE;
endrule: fromAmberNS

(* preempts = "fromGreenE, inc_sec" *)
rule fromGreenE (state == GreenE && (secs >= ew_green_delay + 1 || !car_present_E));
   next_state(AmberE);
endrule: fromGreenE

(* preempts = "fromAmberE, inc_sec" *)
rule fromAmberE (state == AmberE && secs >= amberDelay + 1);
   next_state(AllRed);
   next_green <= GreenW;
endrule: fromAmberE

(* preempts = "fromGreenW, inc_sec" *)
rule fromGreenW (state == GreenW && (secs >= ew_green_delay + 1 || !car_present_W));
   next_state(AmberW);
endrule: fromGreenW

(* preempts = "fromAmberW, inc_sec" *)
rule fromAmberW (state == AmberW && secs >= amberDelay + 1);
   next_state(AllRed);
   next_green <= GreenNS;
endrule: fromAmberW

method Action ped_button_push();
   ped_button_pushed <= True;
endmethod: ped_button_push

method Action set_car_state_N(b) ;
   car_present_N <= b;
endmethod: set_car_state_N

method Action set_car_state_S(b) ;
   car_present_S <= b;
endmethod: set_car_state_S
```

```
   method Action set_car_state_E(b) ;
      car_present_E <= b;
   endmethod: set_car_state_E

   method Action set_car_state_W(b) ;
      car_present_W <= b;
   endmethod: set_car_state_W

   method lampRedNS() = !(state == GreenNS || state == AmberNS);
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = !(state == GreenE || state == AmberE);
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = !(state == GreenW || state == AmberW);
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
   method lampRedPed() = !(state == GreenPed || state == AmberPed);
   method lampAmberPed() = (state == AmberPed);
   method lampGreenPed() = (state == GreenPed);
endmodule: sysTL
endpackage
```

A.10 The source file TL4.bsv

```
// Copyright 2000--2006 Bluespec, Inc.  All rights reserved.

package TL4;

// Simple model of a traffic light

// Version 4: Clean up, factoring out some common rule-building into
//            function calls

import PrimArray::*;

(* always_ready *)
interface TL;
   method Action ped_button_push();

   (* always_enabled *)
   method Action set_car_state_N(Bool x);
   (* always_enabled *)
   method Action set_car_state_S(Bool x);
   (* always_enabled *)
   method Action set_car_state_E(Bool x);
   (* always_enabled *)
   method Action set_car_state_W(Bool x);

   method Bool lampRedNS();
   method Bool lampAmberNS();
   method Bool lampGreenNS();

   method Bool lampRedE();
   method Bool lampAmberE();
   method Bool lampGreenE();

   method Bool lampRedW();
   method Bool lampAmberW();
   method Bool lampGreenW();

   method Bool lampRedPed();
   method Bool lampAmberPed();
   method Bool lampGreenPed();
endinterface: TL

typedef enum {
   AllRed,
   GreenNS, AmberNS,
```

```
   GreenE, AmberE,
   GreenW, AmberW,
   GreenPed, AmberPed} TLstates deriving (Eq, Bits);

typedef UInt#(5) Time32;
typedef UInt#(20) CtrSize;

(* synthesize *)
module sysTL(TL);
   Reg#(TLstates) state <- mkReg(AllRed);
   Reg#(TLstates) next_green <- mkReg(GreenNS);

   Reg#(Time32) secs <- mkReg(0);
   Reg#(CtrSize) sec_ctr <- mkReg(0);

   Reg#(Bool) ped_button_pushed <- mkReg(False);

   Reg#(Bool) car_present_N <- mkReg(True);
   Reg#(Bool) car_present_S <- mkReg(True);
   Reg#(Bool) car_present_E <- mkReg(True);
   Reg#(Bool) car_present_W <- mkReg(True);

   // The following value is dynamically varying:
   Bool car_present_NS = car_present_N || car_present_S;

   CtrSize clocks_per_sec = 100;

   Time32 allRedDelay = 2;
   Time32 amberDelay = 4;
   Time32 ns_green_delay = 20;
   Time32 ew_green_delay = 10;
   Time32 pedGreenDelay = 10;
   Time32 pedAmberDelay = 6;

   function Action next_state(TLstates ns);
      action
 state <= ns;
 secs <= 0;
      endaction
   endfunction: next_state

   function TLstates green_seq(TLstates x);
      case (x)
 GreenNS: return (GreenE);
 GreenE:  return (GreenW);
```

```
GreenW:  return (GreenNS);
     endcase
  endfunction

  function Bool car_present(TLstates x);
     case (x)
GreenNS: return (car_present_NS);
GreenE:  return (car_present_E);
GreenW:  return (car_present_W);
     endcase
  endfunction


  function Rules make_from_green_rule(TLstates green_state,
      Time32 delay,
      Bool car_is_present,
      TLstates ns);

     return (rules
rule from_green (state == green_state &&
 (secs + 1 >= delay || !car_is_present));
   next_state(ns);
endrule
     endrules);
  endfunction: make_from_green_rule

  function Rules make_from_amber_rule(TLstates amber_state,
      TLstates ng);
     return (rules
rule from_amber (state == amber_state &&
 secs + 1 >= amberDelay);
   next_state(AllRed);
   next_green <= ng;
endrule
     endrules);
  endfunction: make_from_amber_rule

  rule dec_sec_ctr (sec_ctr != 0);
     sec_ctr <= sec_ctr - 1;
  endrule


  Rules low_priority_rule =
     (rules
 rule await (sec_ctr == 0);
```

```
     secs <= secs + 1;
     sec_ctr <= clocks_per_sec;
  endrule
       endrules);

   Rules hprs[7];
   hprs[0] =
      (rules
  rule fromAllRed (state == AllRed && secs + 1 >= allRedDelay);
     if (ped_button_pushed)
action
   ped_button_pushed <= False;
   next_state(GreenPed);
endaction
     else if (car_present(next_green))
next_state(next_green);
     else if (car_present(green_seq(next_green)))
next_state(green_seq(next_green));
     else if (car_present(green_seq(green_seq(next_green))))
next_state(green_seq(green_seq(next_green)));
     else
noAction;
  endrule: fromAllRed

  rule fromGreenPed (state == GreenPed && secs + 1 >= pedGreenDelay);
     next_state(AmberPed);
  endrule: fromGreenPed

  rule fromAmberPed (state == AmberPed && secs + 1 >= pedAmberDelay);
     next_state(AllRed);
  endrule: fromAmberPed
       endrules);
   hprs[1] = make_from_green_rule(GreenNS, ns_green_delay, car_present_NS, AmberNS);
   hprs[2] = make_from_amber_rule(AmberNS, GreenE);
   hprs[3] = make_from_green_rule(GreenE, ew_green_delay, car_present_E, AmberE);
   hprs[4] = make_from_amber_rule(AmberE, GreenW);
   hprs[5] = make_from_green_rule(GreenW, ew_green_delay, car_present_W, AmberW);
   hprs[6] = make_from_amber_rule(AmberW, GreenNS);

   Rules high_priority_rules = hprs[0];
   for (Integer i = 1; i<7; i=i+1)
      high_priority_rules = rJoin(hprs[i], high_priority_rules);

   addRules(preempts(high_priority_rules, low_priority_rule));
```

```
   method Action ped_button_push() ;
      ped_button_pushed <= True;
   endmethod: ped_button_push

   method Action set_car_state_N(b) ;
      car_present_N <= b;
   endmethod: set_car_state_N

   method Action set_car_state_S(b) ;
      car_present_S <= b;
   endmethod: set_car_state_S

   method Action set_car_state_E(b) ;
      car_present_E <= b;
   endmethod: set_car_state_E

   method Action set_car_state_W(b) ;
      car_present_W <= b;
   endmethod: set_car_state_W

   method lampRedNS() = !(state == GreenNS || state == AmberNS);
   method lampAmberNS() = (state == AmberNS);
   method lampGreenNS() = (state == GreenNS);
   method lampRedE() = !(state == GreenE || state == AmberE);
   method lampAmberE() = (state == AmberE);
   method lampGreenE() = (state == GreenE);
   method lampRedW() = !(state == GreenW || state == AmberW);
   method lampAmberW() = (state == AmberW);
   method lampGreenW() = (state == GreenW);
   method lampRedPed() = !(state == GreenPed || state == AmberPed);
   method lampAmberPed() = (state == AmberPed);
   method lampGreenPed() = (state == GreenPed);
endmodule: sysTL

endpackage
```