TD Lustre

December 2010

1 Introduction to Lustre Programming

1.1 Lustre Programs

Lustre programs are usually set in a file suffixed by **".lus"**. For instance, let us consider the **Edge** program :

```
node    Egde (X : bool) returns (Y : bool)
let
        Y = false -> X and not pre(X) ;
tel
```

First, we can write this program in a file **edge.lus**. To simulate its behavior, we rely on the **Luciole** simulator.

1.2 Luciole Simulator

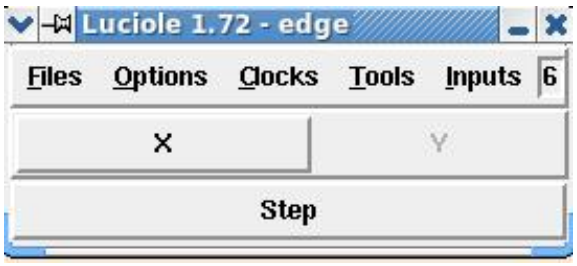

FIG. 1 – Edge node simulation

To test a Lustre program, we call luciole : **luciole** edge.lus Edge.

This command opens a simulation window (see figure1) with a button for **X** and a "lamp" for **Y** (it is highlighted red when **Y** is true and remains gray otherwise). Clicking on **X** button sets it true in the environment of the current instant. To compute the result with **X** false , click on `Step` button. Hence, you can see the behavior of **Edge** node.

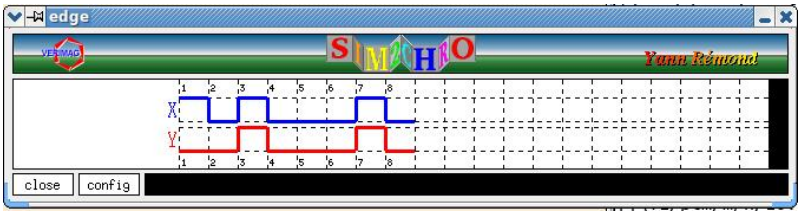

FIG. 2 – Waveform

Waveform

The command : *Tools* → *sim2chro* opens a window where the evolution of **X** and **Y** variables is shown in a "waveform" shape (see figure 2).

2 Exercises

2.1 Programming in Lustre

1. design a node *osc* which computes the flow : {true, false, true, false, true, false,....} ;
2. design a node *osc2* which computes the flow : {true, true, false, false, true, true, ...} using *pre* and → operators ;
3. design a node *count* whose input argument is a Boolean flow *reset* and which computes an integer flow N. At each instant N is increased by one if *reset* is not present. When *reset* is true, N is set to zero.

2.2 Verification with Lesar

To illustrate *Lesar* application, we consider the following node traim (already detailed in the course) :

```
node train (sec,bea: bool) returns (ontime,late, early: bool);
var diff:
let
  diff = (0 -> pre diff) +
         (if bea then 1 else 0) +
         (if sec then -1 else 0);
  early = (true -> pre ontime) and (diff > 3) or
          (false -> pre early) and (diff > 1);
  late = (true -> pre ontime) and (diff < -3) or
         (false -> pre late) and (diff < -1);
  ontime = not (early or late);
tel
```

To prove that "it is impossible to remain late only one instant", we define the following verification node :

```
node train_verif (sec, bea: bool) returns (property: bool);

var ontime, late, early: bool; Plate, PPlate:bool;

let
  (ontime, late, early) = beacon(sec, bea);
  Plate = false-> pre late;
  PPlate = false -> pre Plate;
  property = not (not late and Plate and not PPlate);
tel
```

Then , we verify that the variable property is always true using the Lustre model checker :

lesar train.lus train_verif -diag

Perform the verification and correct the program if the property is falsifiable.

3 Traffic Light

A crossroads with two orthogonal roads (east-west and north-south) is controlled by two traffic lights. Each traffic light works as follows : at each instant, the traffic light manages three Boolean outputs : `red`, `orange`, `green`. These three outputs are exclusive and they are true only following the sequence : `red`, `orange`, `green`, `red`, ...

We consider that the duration of each light is the duration of the clock of the traffic light. We suggest to first, write a program managing a single traffic light and then compose two instances to get the program which manage the crossroads.

Here is a first implementation, test it with the simulator and correct it if necessary.

```
node   TrafficLight(rinit, oinit, ginit : bool)
       returns (red, orange, green : bool)
let
       red = rinit -> pre(orange) ;
       orange = oinit -> pre(green) ;
       green = ginit -> pre(red) ;
tel
node   Crossroads (reset : bool)
       returns (redEW, orangeEW, greenEW, redNS, orangeNS, greenNS : bool)
let
       (redEW, orangeEW, greenEW) = TrafficLight(reset, false, not(reset)) ;
       (redNS, orangeNS, greenNS) = TrafficLight(not(reset),false,reset) ;
tel
```

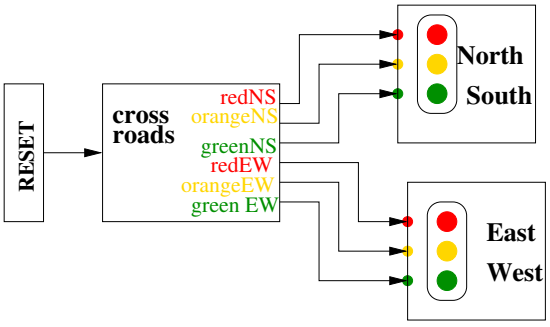

FIG. 3 – Crossroads design

4 Application to WComp

4.0.1 Designing the crossroads in WComp

First, we recall how generate C code for Lustre programs.

4.1 From Lustre to C

Lustre programs compilation follows several steps :

1. a first phase translates a Lustre program into kernel Lustre code. The command : **lus2ec edge.lus Edge** generates a file : **Edge.ec** which contains the kernel code for Edge node.
2. Then this kernel code is compiled into a C-ansi program : **ec2c edge.ec -v** (the -v option is a verbose mode). The result is a file : **Edge.c** which contains the automaton model of the program encoded in C. The file : **Edge.h** contains the external declarations (types, functions, etc...) needed to run the automaton (when it is necessary).
3. This compilation generates only the automaton in C code. The user must provide the main C function which acquires the inputs, shows the outputs and launch the automaton execution. However, the command : **ec2c edge.ec -loop -v** generates : **edge.c**, **edge.h**, and a file **edge_loop.c** defining a "standard" *main* function. Then, it is sufficient to adapt this latter to cope with the wanted main function.

4.1.1 Designing the crossroads in WComp

In WComp, we want to design an assembly to manage a crossroads with the specification already described in 3. This design in shown in figure 3. As it is a critical component, we don't design it in WComp but we generate it as a synchronous monitor. Thus, you must follows four steps :

1. design the component behavior in Lustre (use the Lustre program you have defined in section 3
2. validate the component with Lesar : you must prove that :

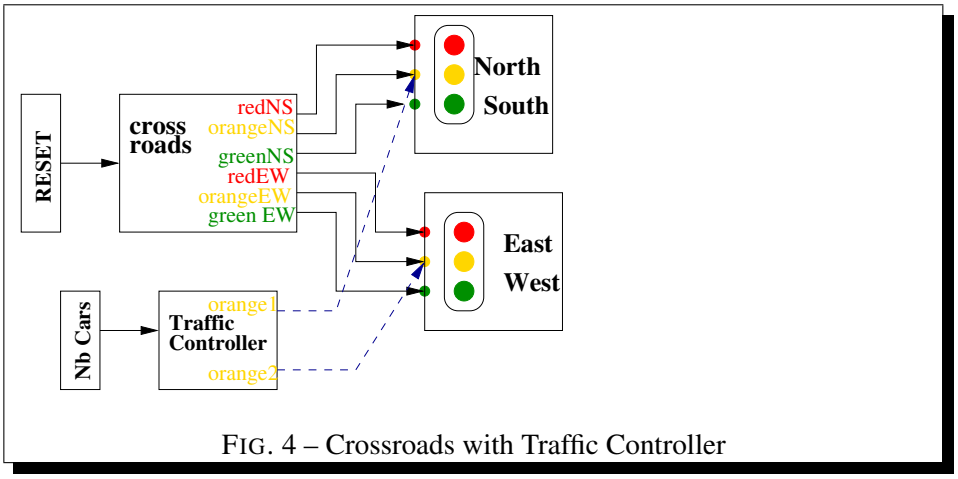

FIG. 4 – Crossroads with Traffic Controller

  - `greenEW` and `orangeNS` or `greenNS` are not true in the same instant and `greenNS` and `orangeEW` or `greenEW` also ;
  - `greenEW` implies `redNS` and the opposite.
3. generate the C code (see section 4.1)
4. generate the C# code to integrate the validated monitor in your WComp design.

4.1.2 Synchronous Monitor Composition

To illustrate the composition mechanism, we add a Traffic Controller component to the design to take into account the traffic density. This component listen the number of cars on the two roads and has two outputs `orange1` and `orange2`(see figure 4 . When the maximal number of cars is less than a given N, its two outputs are true. After specifying the Traffic Controller in Lustre, you must compose it with the `crossroads` component to get an only new component in the WComp design. This composition rely on the definition of a constraint function (see the course) to define correctly what happens when `orange1` and `orange2` are true. Indeed, in such a case, only the respective orange lights of North South traffic light and East West traffic light are highlighted and the others are not.