Assignment Synchronous Programming

Frédéric MALLET
Frederic.Mallet@univ-cotedazur.fr

August 2021

Note : this assignment is designed for you to practice what you have learned. Do whatever exercises you deem best suited for you. If you are confident that you can do some without difficulties feel free to skip them.

1 Observer

We have seen that we can compare the equivalence of two programs with tools like `Lesar`. To do that, you must build a synchronous observer, i.e., a node that returns a Boolean (cf. obs1 and obs2).

```
node obs1(a : bool)
returns (o : bool);
let
  o = a or not a;
tel

node obs2(a,b,c : bool)
returns (o : bool);
let
  o = a or (b and c);
tel
```

Use the following command to test it with the two examples above : `lesar <fic>.lus <node> -v -dbg -diag`.

Exercise 1: BDD obs1

Focus on *obs*1. Build a BDD of the output o using the composition rules seen during the lecture. What can you conclude?

Exercise 2: BDD obs2

Focus now on *obs*2. Build a BDD of the output o using the composition rules seen during the lecture. What can you conclude?

2 1-bit addder

2.1 Half adder

The following `Lustre` node is called a half adder. It adds two inputs of 1-bit and produces two outputs, one for the sum, the other one for the carry.

```
node half_adder(a,b : bool) returns (sum,cy: bool);
let
  sum = a xor b;
  cy  = a and b;
tel
```

Exercise 3: BDD half adder

Build the ROBDDs of the two outputs *sum* and *cy*.

2.2 Full adder

A full adder adds three inputs of one-bit and sums them up to produce two outputs, the sum and the carry. We can build a full adder by combining two half adders (hence the name). The following lustre nodes is a full adder (or is it ?).

```
include "half_adder.lus"

node full_adder(a, b, cin : bool) returns (sum, cy : bool);
var s,c1,c2 : bool;
let
  s,c1 = half_adder(a,b);
  sum,c2 = half_adder(s, cin);
  cy = c1 and c2;
tel
```

Exercise 4: BDD full adder 1

Build two ROBDDs for each output *sum* and *cy* by composing the ones you have built at exercise 3 and by using the ITE rules seen during the lecture.

We can also attempt to build a full adder by writing equations supposed to realize our understanding of what a full adder is :

```
node add1 (a, b, cin : bool)
returns   (s,cout    : bool);
let
  s    = a xor b xor cin;
  cout = if a then b or cin
              else b and cin;
tel
```

Exercise 5: BDD full adder 2

Build two ROBDDs for each output *s* and *cout*.

Exercise 6: Equivalence

Use the ROBDDs of both exercises 4 and 5 to show that the two programs are equivalent. If they are not equivalent, can you find the difference and find the bug? If you do not find it by building the ROBDDs yourself, you can ask `Lustre` and `Lesar` to do it for you.

3 Co-factors

We consider the two following Boolean functions :
— $F(a, b, c) = (a \wedge b) \vee c$
— $G(a, b, c) = (a \vee b) \wedge (b \vee \neg c)$

Exercise 7: ROBDDs and cofactors

1. Build the RO-BDDs of functions F and G;
2. Build the co-factors F_a, F_b, F_c, $F_{\bar{a}}$, $F_{\bar{b}}$, $F_{\bar{c}}$;
3. Build the co-factors G_a, G_b, G_c, $G_{\bar{a}}$, $G_{\bar{b}}$, $G_{\bar{c}}$.

4 ITE

We consider the following Boolean functions :
— $F(a, c) = \neg(a \wedge c)$
— $G(b, c) = b \wedge c$
— $H1(a, b, c) = F(a, c) \vee G(b, c)$
— $H2(a, b, c) = F(a, c) \wedge G(b, c)$
— $H3(a, b, c) = F(a, c) \wedge \neg G(b, c)$

Exercise 8: ITE

Use the ITE algorithm of the lecture to build RO-BDDs for F, G, $H1$, $H2$, $H3$.

5 Rising Edge

The following node `RisingEdge` detects a rising edge on its input.

```
node RisingEdge (b : bool )
  returns ( edge : bool ) ;
let
  edge = false -> b and not pre b ;
tel
```

Exercise 9: BDD edge

— Build a state machine of this node.
— Deduce from the state machine a truthtable for the output *edge*.
— Deduce from the truthtable a RO-BDD for *edge*.

Note : States can be encoded with Boolean variables, 2 states require one Boolean variable, 4 states require 2 Boolean variables and so on. Encoding a state machine (a Mealy machine) amounts to encoding a Boolean function F that represents the outputs and another Boolean function G that represents the next state.

— $O = F(I, Q)$, the outputs is a Boolean function of the inputs I and the state Q.
— $Q^+ = G(I, Q)$, the next state is a Boolean function of the inputs I and the current state Q.