Synchronous programming exercises
Compiling and Verifying

Pascal Raymond
Verimag-CNRS

MOSIG - Embedded Systems

Using compilation into automaton for reasoning about program

The autograph tool (atg)

- The command **`lus2atg file.lus node`**:
  - ↪ compile the node **`node`** form the file **`file.lus`** into a minimal automaton (as seen in the course), in a format called **`oc`**
  - ↪ extract graphical information from the automaton in a file **`node.atg`**
- Once created the atg file, type **`atg node.atg`** to start exploring graphically the automaton
- Hints: within the **`atg`** window, type **`x`** to enter the explore mode.

Explore some automata...

- the car light controller, the switch etc.
- Warning: the atg automaton is a *canonical* representation of the program behaviors, as long as only Boolean operation are involved
  this is not the case with programs using numerical values (try it)

Comparing two programs

- let **`myCarLights`** be your own implementation of the car light controller,
- ask a colleague for his/her version **`otherCarLights`**
  (the more ”different” seems the code, the more interesting will be the exercise),
- write a node **`compare`**:
  - ↪ the inputs are those of the controllers (**`TL, TR, LH)`**,
  - ↪ it has a single output **`ok`**,
  - ↪ it contains both a call of **`myCarLights`** and **`otherCarLights`**,
  - ↪ **`ok`** is defined as the conjunction of the pair-wise comparison between the outputs of **`myCarLights`** and those of **`otherCarLights`**

Comparing two programs (cntd)

- generate and explore the automaton of the node **`compare`**,
- can you deduce from this automaton whether the two controllers are equivalent or not ?
- if they are not equivalent, think about some assumption that would make then equivalent:
  - ↪ e.g. initial condition, exclusion of inputs etc.

Comparing two programs (cntd)

- about assertions in Lustre:
  - ↪ assumptions can be introduced in Lustre program with **assert <Boolean expresion>;**
  - ↪ e.g. **assert not (TR and TL);** assumes that it is impossible to turn both right and left at the same time
  - ↪ the "exclusion" operator of Lustre is often convenient: **assert #(X1,...,Xn);** assumes that at most one variable **Xi** is true at each instant
- When generating an automaton, the compiler *removes any transition* that violates the assertions,
- Try to write the suitable assertions in the **compare** node for making (and proving via the automaton) the equivalence of the 2 controllers.

Proving properties with xlesar

xlesar

- launch **xlesar**
- load the program+node to check:
  Browse button, right side of **Main Mode** line
- → the lists of inputs/outputs is listed
- create and edit a property:
  New button then select property and **Edit** button
- → by default, the the property is simply **true**, which is trivially an invariant !
  Check it by pressing **RUN PROOF** button

Find and check interesting properties

- try, for instance, to check that **`side`** and **`low`** are exclusive
- Hint: some hypothesis (called *assertion* in Lustre) are maybe necessary, you can add assumptions via **`Edit`** menu, **`New assertion`**.
- find and check other interseting properties ...

Binary arithmetics

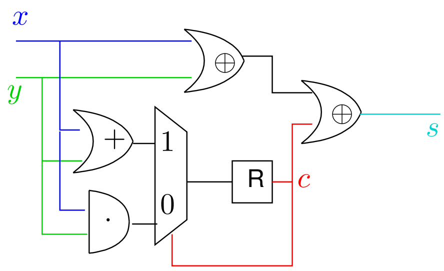
Binary numbers in Lustre

- Lustre Boolean sequences are encoding unbounded binary number, low significant bit first:
- Sequences ”ending” with an infinity of 0 are cearly classical natural numbers: e.g. **`false true true false true false false ...`** encodes $2 + 4 + 16 = 22$
- to simplify the notation, from now on, we write **`0`** and **`1`** for Lustre Boolean values: e.g. **`01010...`** for **`false true false true false ...`**
- Give the Lustre flow that encodes the natural 0 ? 1 ? 17 ?

Serial adder

- Lustre Boolean sequences are encoding unbounded binary number, low significant bit first:
  e.g. 0011010 encodes $4 + 8 + 32 = 44$
- A serial adder:
  - ↪ takes 2 Boolean flows **x** and **y** interpreted as binary numbers x and y,
  - ↪ produces *step by step* the bits of the sum $s = x + y$,
  - ↪ it uses an internal flow **c** indicating the current carry, whenever **c = 0** (no pending carry) the result is exact so far
- DON'T LOOK AT THE NEXT SLIDE: try to write the serial adder in Lustre
- If you are stucked, take a look at the next slide...

The serial adder circuit



- inputs x, y
- sum s, carry c
- classical circuit
- easy to translate into Lustre
- behavior:

time →

c	0	0	1	0	
x	0	1	0		(2)
y	1	1	0		(3)
s	1	0	1		(5)
					(4)

Proving basic arithmetic theorems

- If **x** encodes the numer x, how to compute the flow **m** that encodes *two times x* $(2x)$?
- Prove that, for any number, $(x = y) \Rightarrow (x + y = 2x)$
- Prove that the infinite flow **11111...** encodes the number -1

Light controller

The goal is to check your version of the car light controller.

- Find and formalize some expected properties.
- Try to prove them, find the necessary assumptions if necessary.

Generic observers

The goal is to write a set of generic observers for common and useful properties on logical events:

- never X between A and the following B;
- always X between A and the following B;
- at least one X between A and the following B.

To do:

- formalize these properties, for instance with an explicit automaton that recognize the correct sequences of A, B, X;
- write these observers in Lustre, test them, compile them in atg, check that they correspond to the expected automata.

Streetcar door controller

Goal:

- Given an already written program,
- Formalize the expected properties,
- Try to check them, and introduce (if necessary) a set of necessary assumptions (as few as possible)

Streetcar door controller (cntd)

The controller:

- Inspired by the (old) version of Grenoble streetcar: control a door + a ramp for wheelchairs.
- Here, a very simplified version.
- Inputs:
  - ↪ user requests: `ramp_req` et `door_req`
  - ↪ State of the train: `in_station`, `end_station`
- Outputs:
  - ↪ door/ramp state: `ramp_on`, `door_on`
  - ↪ departure acknowledgment: `door_and_pass_ok`

Streetcar door controller (cntd)

Expected properties (informally):

- never runs with door or ramp on
- never moves ramp while door is opened

To do:

- test/simulate the program with luciole
- formalize the properties
- fnd the (necessary) assumptions

Streetcar door controller (cntd)

Technical notes:

- download the code `streetcar.lus`, `utils.lus`
- in xlesar, use the menu ”import” to load any extra Lustre file: `utils.lus`, and, probably, your own extra code (generic observers for instance)