CS:5810 Formal Methods in Software Engineering

Reactive Systems and the Lustre Language

Part 2

Adrien Champion    Cesare Tinelli
Design of reactive systems:

- run in an infinite loop, and
- produce an output every \( n \) milliseconds
Lustre: a synchronous dataflow language

Design of **reactive** systems:

- run in an infinite loop, and
- produce an output every \( n \) milliseconds

<table>
<thead>
<tr>
<th></th>
<th>( in_0 )</th>
<th>( in_1 )</th>
<th>( in_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lustre: a synchronous dataflow language

Design of reactive systems:
- run in an infinite loop, and
- produce an output every $n$ milliseconds
Exercises

Model a switch with two buttons, Set and Reset.

```plaintext
node Switch( Set, Reset, Init : bool ) returns ( State : bool );
```

such that:

- pressing Set turns the switch on;
- pressing Reset turns the switch off;
- the initial position of the switch is determined by a third signal Init if Set and Reset are initially both unpressed.
Exercises

Model a switch with two buttons, Set and Reset.

```node
Switch( Set, Reset, Init : bool ) returns ( State : bool );
```

such that:

- pressing Set turns the switch on;
- pressing Reset turns the switch off;
- the initial position of the switch is determined by a third signal Init if Set and Reset are initially both unpressed.

```node
Switch( Set, Reset, Init : bool )
returns ( X : bool );
let
X = if Set then true
else if Reset then false
else (Init -> pre X);
tel
```
Model a switch with two buttons, Set and Reset.

```
node Switch( Set, Reset, Init : bool ) returns ( State : bool );
```

such that:

- pressing `Set` turns the switch on;
- pressing `Reset` turns the switch off;
- the initial position of the switch is determined by a third signal `Init` if `Set` and `Reset` are initially both unpressed.

Equivalently:

```
node Switch( Set, Reset, Init : bool )
returns ( X : bool );
let
    X = Set or ( not Reset and ( Init -> pre X) );
```

Exercises

node ??? (r, b: bool) returns (out: int);

let

  out = if r then 0
    else if b then (0 -> pre out) + 1
    else (0 -> pre out);

tel
node ??? (r,b: bool) returns (out: int);
let

\[
\begin{align*}
\text{out} &= \quad \text{if } r \text{ then } 0 \\
& \quad \text{else if } b \text{ then } (0 \rightarrow \text{pre out}) + 1 \\
& \quad \text{else } \quad (0 \rightarrow \text{pre out}); \\
\end{align*}
\]

tel
node ??? (r, b: bool) returns (out: int);
let

\[
\begin{align*}
\text{out} &= \quad \text{if } r \text{ then } 0 \\
&\quad \text{else if } b \text{ then } (0 \rightarrow \text{pre out}) + 1 \\
&\quad \text{else } \quad (0 \rightarrow \text{pre out});
\end{align*}
\]

\text{tel}
Counter with reset:

```plaintext
define Node ??? (r, b : bool) returns (out : int);
let
    out = if r then 0
        else if b then (0 -> pre out) + 1
        else (0 -> pre out);
end
```

<table>
<thead>
<tr>
<th>r0</th>
<th>b0</th>
<th>r1</th>
<th>b1</th>
<th>r2</th>
<th>b2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{out at 0} = 1
\]

\[
\text{out at 1} = 2
\]

\[
\text{out at 2} = 0
\]
Counter with reset:

```plaintext
def node cnt (r, b: bool) returns (out: int);
    var pre_out: int;
    let pre_out = 0 -> pre_out;
    out = if r then 0
             else if b then pre_out + 1
             else pre_out;
    tel
```

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>b</td>
<td>r</td>
<td>b</td>
<td>r</td>
<td>b</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out at 0</td>
<td>out at 1</td>
<td>out at 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Counter with reset:

\[
\text{node cnt (r, b: bool) returns (out: int);}
\]

\[
\text{var pre_out: int;}
\]

\[
\text{let pre_out = 0 -> pre out;}
\]

\[
\text{out = if r then 0}
\]

\[
\text{else if b then pre_out + 1}
\]

\[
\text{else pre_out;}
\]

\text{tel}
Modularity

Once defined, a node can be used as a basic operator
Once defined, a node can be used as a basic operator

What does A look like?

\[ X = \text{true} \rightarrow (\text{pre } A = 3) \]
\[ A = \text{cnt}(X, \text{true}); \]
Once defined, a node can be used as a basic operator

What does $A$ look like?

$X = \text{true} \rightarrow (\text{pre } A = 3)$

$A = \text{cnt}(X, \text{true})$;

$A = 0,$
Once defined, a node can be used as a basic operator

What does A look like?

\[
X = \text{true} \rightarrow (\text{pre } A = 3)
\]

\[
A = \text{cnt}(X, \text{true});
\]

\[A = 0, 1,\]
Once defined, a node can be used as a basic operator

What does $A$ look like?

\[ X = \text{true} \rightarrow (\text{pre } A = 3) \]
\[ A = \text{cnt}(X, \text{true}); \]
\[ A = 0, 1, 2, \]
Modularity

Once defined, a node can be used as a basic operator

What does \( A \) look like?

\[
X = \text{true} \rightarrow (\text{pre } A = 3)
\]

\[
A = \text{cnt}(X, \text{true});
\]

\[
A = 0, 1, 2, 3,
\]
Modularity

Once defined, a node can be used as a basic operator

What does $A$ look like?

\[
X = \text{true} \implies (\text{pre } A = 3)
\]

\[
A = \text{cnt}(X, \text{true});
\]

\[
A = 0, 1, 2, 3, 0,
\]
Once defined, a node can be used as a basic operator

What does \( A \) look like?

\[
X = \text{true} \implies (\text{pre } A = 3)
\]

\[
A = \text{cnt}(X, \text{true});
\]

\[A = 0, 1, 2, 3, 0, 1, 2, 3, 0, 1\ldots\]
A node can have several outputs:

```plaintext
node MinMax( X : real ) returns ( Min, Max : real );
let
    Min = X -> if (X < pre Min) then X else pre Min ;
    Max = X -> if (X > pre Max) then X else pre Max ;
.tel

node minMaxAverage ( X : real ) returns ( Y : real ) ;
var Min, Max: real ;
let
    Min, Max = MinMax(X) ;
    Y = (Min + Max)/2.0 ;
.tel
```
Complete example: specification

Stopwatch:

- one integer output: \texttt{time} “to display”;
- three input buttons:
  - \texttt{on\_off} starts and stops the stopwatch,
  - \texttt{reset} resets the stopwatch \textbf{if not running},
  - \texttt{freeze} freezes the displayed time \textbf{if running}, cancelled if stopped
Complete example: available nodes

-- Bistable switch
node switch (on, off: bool) returns (state: bool);
let
  state =
    if (false -> pre state) then not off else on;
tel

-- Counts steps if inc is true, can be reset
node counter (reset, inc: bool) returns (out: int);
let
  out = if reset then 0
       else if inc then (0 -> pre_out) + 1
       else (0 -> pre_out);
tel

-- Detects raising edges of a signal
node edge (in: bool) returns (out: bool);
let
  out = false -> in and (not pre in);
tel
Complete example: solution(s)

Unsatisfactory solution not using edge:

```plaintext
node stopwatch (on_off, reset, freeze: bool)
returns (time: int);
var actual_time: int;
    running, frozen: bool;

let

    running = switch(on_off, on_off);
frozen = switch(
    freeze and running, freeze or on_off
);
actual_time = counter(reset and not running, running);
time = if frozen then (0 -> pre time) else actual_time;
tel
```
Satisfactory solution:

```
node stopwatch (on_off, reset, freeze: bool)
returns (time: int);
var actual_time: int;
    running, frozen,
    on_off_pressed, r_pressed, f_pressed: bool;
let
    on_off_pressed = edge(on_off);
    r_pressed = edge(reset);
    f_pressed = edge(freeze);
    running = switch(on_off_pressed, on_off_pressed);
    frozen = switch(
        f_pressed and running, f_pressed or on_off_pressed
    );
    actual_time = counter(r_pressed and not running, running);
    time = if frozen then (0 -> pre time) else actual_time;
tel
```
Part of these notes are based on the following lectures notes:

The Lustre Language — Synchronous Programming
by Pascal Raymond and Nicolas Halbwachs
Verimag-CNRS