The Lustre Language

Synchronous Programming

Pascal Raymond, Nicolas Halbwachs
Verimag-CNRS
Data-flow approach

- A program = a network of operators connected by wires

- Rather classical (control theory, circuits)

\[
\begin{align*}
\text{node } & \text{Average}(X, Y : \text{int}) \\
\text{returns } & (A : \text{int}); \\
\text{let } & \ A = (X + Y) / 2 ; \\
\text{tel}
\end{align*}
\]

- Synchronous: discrete time \( \mathbb{N} \)

\[
\forall t \in \mathbb{N} \quad A_t = (X_t + Y_t) / 2
\]

- Full parallelism: nodes are running concurrently
Another version

node Average(X, Y : int)
returns (A : int);

var S : int;          – local variable
let
    A = S / 2;        – equations
    S = X + Y;       – (order does not matter)
tel

● declarative: set of equations

● a single equation for each output/local

● variables are infinite sequences of values
Lustre (textual) and Scade (graphical)
Combinational programs

- Basic types: bool, int, real

- Constants:
  \[ 2 \equiv 2, 2, 2, \ldots \]
  \[ \text{true} \equiv \text{true, true, true, } \ldots \]

- Pointwise operators:
  \[ \mathbf{X} \equiv x_0, x_1, x_2, x_3\ldots \]
  \[ \mathbf{Y} \equiv y_0, y_1, y_2, y_3\ldots \]
  \[ \mathbf{X} + \mathbf{Y} \equiv x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3\ldots \]

- All classical operators are provided
• if operator

node Max(A,B: real) returns (M: real);
let
    M = if (A >= B) then A else B;
tel

Warning: functional “if then else”, not statement

Combinational programs
Lustre (textual) and Scade (graphical)
Memory programs

Delay operator

- Previous operator: \texttt{pre}

\[
\begin{array}{cccccc}
X & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
\text{pre} X & \text{nil} & x_0 & x_1 & x_2 & x_3 & \ldots \\
\end{array}
\]
Memory programs

Delay operator

- Previous operator: \texttt{pre}

\[
\begin{array}{cccccc}
X & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
\text{pre}X & \text{nil} & x_0 & x_1 & x_2 & x_3 & \ldots \\
\end{array}
\]

i.e. \((\text{pre}X)_0\) undefined and \(\forall i \neq 0\) \((\text{pre}X)_i = X_{i-1}\)
Memory programs

Delay operator

- Previous operator: `pre`

  \[
  X \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \ldots
  \]

  \[
  \text{pre} \ X \quad \text{nil} \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad \ldots
  \]

  i.e. \((\text{pre} X)_0\) undefined and \(\forall i \neq 0 \ (\text{pre} X)_i = X_{i-1}\)

- Initialization: `->`

  \[
  X \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \ldots
  \]

  \[
  Y \quad y_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots
  \]

  \[
  X \rightarrow Y \quad x_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots
  \]
Memory programs

Delay operator

- **Previous operator:** \( \text{pre} \)

\[
\begin{align*}
X & \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \ldots \\
\text{pre} \ X & \quad \text{nil} \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad \ldots \\
\text{i.e.} \quad \text{(pre} \ X\text{)}_0 & \quad \text{undefined and } \forall i \neq 0 \quad \text{(pre} \ X\text{)}_i = X_{i-1}
\end{align*}
\]

- **Initialization:** \( \rightarrow \)

\[
\begin{align*}
X & \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \ldots \\
Y & \quad y_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots \\
X \rightarrow Y & \quad x_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots \\
\text{i.e.} \quad \text{(}X \rightarrow Y\text{)}_0 & = X_0 \text{ and } \forall i \neq 0 \quad \text{(}X \rightarrow Y\text{)}_i = Y_i
\end{align*}
\]
Nodes with memory

- **Boolean example: raising edge**

  node Edge (X : bool) returns (E : bool);
  let
    E = false -> X and not pre X;
  tel

- **Numerical example: min and max of a sequence**

  node MinMax(X : int)
  returns (min, max : int);  // several outputs
  let
    min = X -> if (X < pre min) then X else pre min;
    max = X -> if (X > pre max) then X else pre max;
  tel
Recursive definition

Examples

\( N = 0 \rightarrow \text{pre } N + 1 \)
Recursive definition

Examples

- $N = 0 \rightarrow \text{pre } N + 1 \quad N = 0, 1, 2, 3, \ldots$
Recursive definition

Examples

- $N = 0 \rightarrow \text{pre } N + 1 \quad N = 0, 1, 2, 3, \ldots$
- $A = \text{false} \rightarrow \text{not pre } A$
Recursive definition

Examples

• \( N = 0 \rightarrow \text{pre} \ N + 1 \) \( N = 0, 1, 2, 3, \ldots \)

• \( A = \text{false} \rightarrow \text{not pre} \ A \) \( A = \text{false, true, false, true,} \ldots \)
Recursive definition

Examples

- $N = 0 \rightarrow \text{pre } N + 1 \quad N = 0, 1, 2, 3, \cdots$
- $A = \text{false} \rightarrow \text{not pre } A \quad A = \text{false, true, false, true, } \cdots$
- Correct $\implies$ the sequence can be computed step by step

Counter-example

- $x = 1/(2-x)$
- unique (integer) solution: “$x=1$”
- but not computable step by step

Sufficient condition: forbid combinational loops

How to detect combinational loops?
Syntactic vs semantic loop

- **Example:**
  
  \[
  X = \text{if } C \text{ then } Y \text{ else } A; \\
  Y = \text{if } C \text{ then } B \text{ else } X;
  \]

- **Syntactic loop**

- **But not semantic:** \( X = Y = \text{if } C \text{ then } B \text{ else } A \)

**Correct definitions in Lustre**

- **Choice:** syntactic loops are rejected
  
  (even if they are “false” loops)
Exercices

• A flow $F = 1, 1, 2, 3, 5, 8, \cdots$?

• A node $\text{Switch}(\text{on, off: bool})$ returns $(s: \text{bool})$;
  such that:
  ★ $s$ raises ($false$ to $true$) if $\text{on}$, and falls ($true$ to $false$) if $\text{off}$
  ★ everything behaves as if $s$ was $false$ at the origin
  ★ must work properly even if $\text{off}$ and $\text{on}$ are the same

• A node $\text{Count}(\text{reset, x: bool})$ returns $(c: \text{int})$;
  such that:
  ★ $c$ is reset to 0 if $\text{reset}$, otherwise it is incremented if $x$,
  ★ everything behaves as if $c$ was 0 at the origin
Solutions

- **Fibonacci:**
  
  \[ f = 1 \rightarrow \text{pre}( f + (0 \rightarrow \text{pre } f)) \];

- **Bistable:**
  
  node Switch(on, off: bool) returns (s: bool);
  let s = if(false \rightarrow \text{pre } s) then not off else on; tel

- **Counter:**
  
  node Count(reset, x: bool) returns (c: int);
  let
  
  \[ c = \begin{cases} 
  0 & \text{if reset} \\
  (0 \rightarrow \text{pre } c) + 1 & \text{if } x \\
  (0 \rightarrow \text{pre } c) & \text{else}
  \end{cases} \]

  tel
Modularity

Reuse

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

• Instanciation is functional-like

• Example (exercise: what is the value?)
  \[ A = \text{Count}(\text{true} \to (\text{pre} \ A = 3), \text{true}) \]

• Several outputs:

  ```
  node MinMaxAverage(x: bool) returns (a: int);
  var min, max: int;
  let
      a = average(min, max);
  min, max = MinMax(x);
  tel
  ```
A complete example: stopwatch

- 1 integer output: displayed time

- 3 input buttons: on_off, reset, freeze
  - on_off starts and stops the stopwatch
  - reset resets the stopwatch (if not running)
  - freeze freezes the displayed time (if running)
A complete example: stopwatch

- 1 integer output: displayed **time**

- 3 input buttons: **on/off**, **reset**, **freeze**
  - **on/off** starts and stops the stopwatch
  - **reset** resets the stopwatch (if not running)
  - **freeze** freezes the displayed time (if running)

- Find local variables (and how they are computed):
  - **running**: bool, a *Switch* instance
  - **freezed**: bool, a *Switch* instance
  - **cpt**: int, a *Count* instance
node Stopwatch(on_off, reset, freeze: bool)
returns (time: int);
var running, freezed: bool; cpt: int;
let
    running = Switch(on_off, on_off);
freezed = Switch(
    freeze and running,
    freeze or on_off);
cpt = Count(reset and not running, running);
time = if freezed then (0 -> pre time) else cpt;
tel
Motivation

- Attempt to conciliate “control” with data-flow
- Express that some part of the program works less often
- ⇒ notion of data-flow clock (similar to clock-enabled in circuit)
Clocks

Motivation

- Attempt to conciliate “control” with data-flow
- Express that some part of the program works less often
- \( \Rightarrow \) notion of data-flow clock (similar to clock-enabled in circuit)

Sampling: \texttt{when} operator

\[
\begin{array}{cccccccc}
X & 4 & 1 & -3 & 0 & 2 & 7 & 8 \\
C & \text{true} & \text{false} & \text{false} & \text{true} & \text{true} & \text{false} & \text{true} \\
X \text{ when } C & 4 & 0 & 2 & 8 \\
\end{array}
\]

- whenever \( C \) is false, \( X \text{ when } C \) does not exist
Projection: \texttt{current} operator

- One can operate only on flows with the same clock
- \texttt{projection} on a common clock is (sometime) necessary

\[
\begin{array}{cccccccc}
X & 4 & 1 & -3 & 0 & 2 & 7 & 8 \\
C & true & false & false & true & true & false & true \\
Y = X \text{ when } C & 4 & 0 & 2 & 8 \\
Z = \texttt{current}(Y) & 4 & 4 & 4 & 0 & 2 & 2 & 8 \\
\end{array}
\]
Nodes and clocks

- Clock of a node instance = clock of its effective inputs
- Sampling inputs = enforce the whole node to run slower
- In particular, sampling inputs $\neq$ sampling outputs

<table>
<thead>
<tr>
<th>C</th>
<th>true</th>
<th>true</th>
<th>false</th>
<th>false</th>
<th>true</th>
<th>false</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count((r,true) when C)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Count(r,true) when C</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: stopwatch with clocks

node Stopwatch(on_off, reset, freeze: bool)
returns (time: int);
var running, freezed: bool;
    cpt_ena, tim_ena : bool;
    (cpt:int) when cpt_ena;
let
    running = Switch(on_off, on_off);
    freezed = Switch(
        freeze and running,
        freeze or on_off);
    cpt_ena = true -> reset or running;
    cpt = Count((not running, true) when cpt_ena);
    tim_ena = true -> not freezed;
    time = current(current(cpt) when tim_ena);
tel
Clock checking

- Similar to type checking
- Clocks must be named (clocks are equal iff they are the same var)
- The clock of each var must be declared (the default is the base clock)
- \( \text{clk}(\text{exp when } C) = C \Leftrightarrow \text{clk}(\text{exp}) = \text{clk}(C) \)
- \( \text{clk}(\text{current exp}) = \text{clk}(\text{clk}(\text{exp})) \)
- For any other \( \text{op} \):
  \[ \text{clk}(\text{e1 op e2}) = C \Leftrightarrow \text{clk}(\text{e1}) = \text{clk}(\text{e2}) = C \]
Programming with clocks

- Clocks are the right semantic solution

- However, using clocks is quite tricky (cf. stopwatch)

- Main problem: initialisation

  \[ \text{current}(X \text{ when } C) \text{ exists, but is undefined until } C \text{ becomes true for the first time} \]

- Solution: \textit{activation condition}

  - not an operator, rather a \textit{macro}

  \[ X = \text{CONDACT}(\text{OP, clk, args, dflt}) \text{ equivalent to:} \]

  \[ X = \text{if } \text{clk then current}(\text{OP(args when clk)}) \]

  \[ \text{else (dflt } \rightarrow \text{ pre } X) \]

  - Provided by Scade (industrial)
Is that all there is?

Dedicated vs general purpose languages

- Synchronous languages are dedicated to *reactive kernel*
- Not suitable for complex data types manipulation
- Abstract types and functions are *imported* from the host language (typically C)

However ...

- Statically sized arrays are provided
- Static recursion (Lustre V4, dedicated to circuit)
- Modules and templates (Lustre V6, dedicated to software)