The Lustre language

Pascal Raymond
Verimag-CNRS

MOSIG - Embedded Systems

Data-flow approach

- A program = a network of operators connected by wires
- Rather classical (control theory, circuits)

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

- Synchronous: discrete time = \(\mathbb{N}\)
  \(\forall t \in \mathbb{N} \ A_t = (X_t + Y_t)/2\)
- Full parallelism: nodes are running concurrently
Another version

```plaintext
node Average(X, Y : int)
returns (A : int);
var S : int;
let
    A = S / 2;
    S = X + Y;
tel
```

- declarative: set of equations (≠ sequence of assignments)
- a single equation for each output and local variable
- 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:
  \[ X \equiv x_0, x_1, x_2, x_3\ldots \]
  \[ Y \equiv y_0, y_1, y_2, y_3\ldots \]
  \[ X + Y \equiv x_0 + y_0, x_1 + y_1, x_2 + y_2, x_3 + y_3\ldots \]

- All classical operators are provided

- Operator if-then-else

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

Warning: functional “if then else” (≠ control statement)
Memory programs

Delay operator

- Previous operator: $\text{pre}$

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

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

- Initialization: $\rightarrow$

\[
\begin{array}{c}
X \quad x_0 \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad \ldots \\
Y \quad y_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots \\
X \rightarrow Y \quad x_0 \quad y_1 \quad y_2 \quad y_3 \quad y_4 \quad \ldots \\
\end{array}
\]

$\rightarrow$ i.e. $(X \rightarrow Y)_0 = X_0$ and $\forall i \neq 0 (X \rightarrow Y)_i = Y_i$

Nodes with memory

- Boolean example: raising edge

  \begin{verbatim}
node Edge (X : bool) returns (E : bool);
let
  E = false $\rightarrow$ X and not pre X;
end
\end{verbatim}

- Numerical example: min and max of a sequence

  \begin{verbatim}
node MinMax(X : int)
returns (min, max : int); -- several outputs
let
  min = X $\rightarrow$ if (X < pre min) then X else pre min;
  max = X $\rightarrow$ if (X > pre max) then X else pre max;
end
\end{verbatim}
Recursive definition

Examples

- $N = 0 \rightarrow \text{pre } N + 1$
  
  $N = 0, 1, 2, 3, \cdots$

- $A = \text{false} \rightarrow \text{not pre } A$
  
  $A = \text{false, true, false, true, } \cdots$

- Correct $\Rightarrow$ 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: $X = Y = \text{if } C \text{ then } B \text{ else } A$
    is the unique solution

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 $\text{Switch}(\text{on}, \text{off}: \text{bool}) \ \text{returns} \ (s: \text{bool})$; such that:
  $\rightarrow s$ raises (from false to true) if $\text{on}$
  $\rightarrow s$ falls (from true to false) if $\text{off}$
  $\rightarrow s$ is false at the origin
  $\rightarrow$ must work properly even if $\text{off}$ and $\text{on}$ are both true

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

Solutions

• Fibonacci:
  $f = 1 \rightarrow \text{pre}(f + (0 \rightarrow \text{pre} f));$

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

• Counter:
  $\text{node Count}(\text{reset}, x: \text{bool}) \ \text{returns} \ (c: \text{int});$
  let
  $c = \text{if} \ \text{reset} \ \text{then} \ 0$
  else if $x$ then $(0\rightarrow \text{pre} c) + 1$
  else $(0\rightarrow \text{pre} c);$
  tel
Modularity

Reuse

- Once defined, a user node can be used as a basic operator
- Instantiation is functional-like
- Example (exercise: what is the value?)
  \[
  A = \text{Count} (\text{true} \rightarrow (\text{pre} A = 3), \text{true})
  \]
- Several outputs:
  ```
  node MinMaxAverage(x: int) 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)
- Find local variables (and how they are computed):
  - `running`: bool, a `Switch` instance
  - `frozen`: bool, a `Switch` instance
  - `cpt`: int, a `Count` instance
node Stopwatch(on_off, reset, freeze: bool) returns (time: int);
var running, frozen: bool; cpt: int;
let
  running = Switch(on_off, on_off);
  frozen = Switch(freeze and running, freeze or on_off);
  cpt = Count(reset and not running, running);
  time = if frozen then (0 -> pre time) else cpt;
tel

Clocks

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)

Sampling: when operator

<table>
<thead>
<tr>
<th>X</th>
<th>4</th>
<th>1</th>
<th>-3</th>
<th>0</th>
<th>2</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>X when C</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- whenever C is false, X when C does not exist
Projection: \textbf{current} operator

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

\begin{align*}
X & \quad \begin{array}{cccccc}
4 & 1 & -3 & 0 & 2 & 7 & 8
\end{array} \\
C & \quad \begin{array}{ccccccc}
\text{true} & \text{false} & \text{false} & \text{true} & \text{true} & \text{false} & \text{true}
\end{array} \\
Y = X \text{ when } C & \quad \begin{array}{cccc}
4 & 0 & 2 & 8
\end{array} \\
Z = \text{current}(Y) & \quad \begin{array}{cccccc}
4 & 4 & 4 & 0 & 2 & 2 & 8
\end{array}
\end{align*}

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

\begin{align*}
C & \quad \begin{array}{ccccccc}
\text{true} & \text{true} & \text{false} & \text{false} & \text{true} & \text{false} & \text{true}
\end{array} \\
\text{Count}((r, \text{true}) \text{ when } C) & \quad \begin{array}{ccc}
1 & 2 & 3 & 4
\end{array} \\
\text{Count}(r, \text{true}) \text{ when } C & \quad \begin{array}{ccc}
1 & 2 & 5 & 7
\end{array}
\end{align*}
Example: stopwatch with clocks

node Stopwatch(on_off, reset, freeze: bool)
returns (time: int);
var running, frozen: bool;
cpt_ena, tim_ena : bool;
(cpt: int) when cpt_ena;
let
  running = Switch(on_off, on_off);
frozen = 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 frozen;
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)
- \( clk(exp \ when \ C) = C \iff clk(exp) = clk(C) \)
- \( clk(current \ exp) = clk(clk(exp)) \)
- For any other \( op \):
  \( clk(e_1 \ op \ e_2) = C \iff clk(e_1) = clk(e_2) = C \)
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: activation condition
  - not an operator, rather a macro
  - \[ X = \text{CONDACT}(\text{OP}, \text{clk}, \text{args}, \text{dflt}) \text{ equivalent to:} \]
    \[ X = \text{if } \text{clk} \text{ then current}(\text{OP(args when clk)}) \]
    \[ \quad \text{else } (\text{dflt } \rightarrow \text{pre } X) \]
  - Provided by Scade (industrial)

Is that all there is?

Dedicated vs general purpose languages

- Synchronous languages are dedicated to reactive kernel
- Not (really) convenient 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)