CS:5810 Formal Methods in Software Engineering

Reactive Systems and the Lustre Language

Adrien Champion  Cesare Tinelli
Embedded systems development

Pivot language between design and code should have clear and precise semantics, and be consistent with design / prototype formats and target platforms.
Embedded systems development

Pivot language between design and code should

- have clear and precise semantics, and
Embedded systems development

Pivot language between design and code should

- have clear and precise semantics, and
- be consistent with design / prototype formats and target platforms
Lustre: a synchronous dataflow language

- **Synchronous:**
  - a base clock regulates computations;
  - computations are inherently parallel

- **Dataflow:**
  - inputs, outputs, variables, constants ... are endless streams of values
Lustre: a synchronous dataflow language

- **Synchronous:**
  - a base clock regulates computations;
  - computations are inherently parallel

- **Dataflow:**
  - inputs, outputs, variables, constants . . . are endless streams of values

- **Declarative:**
  - set of equations, no statements
Lustre: a synchronous dataflow language

- **Synchronous:**
  a base clock regulates computations; computations are inherently parallel

- **Dataflow:**
  inputs, outputs, variables, constants ... are endless streams of values

- **Declarative:**
  set of equations, no statements

- **Reactive systems:**
  Lustre programs run forever
  At each clock tick they
  - compute outputs from their inputs
  - before the next clock tick
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
  out = (x + y) / 2.0;
tel
```
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
end
```

Circuit view:

![Circuit diagram]
A simple example

\textbf{node average (x, y: real) returns (out: real);}
let
  \textbf{out} = (x + y) / 2.0;
\textbf{tel}

Mathematical view:

\[ \forall i \in \mathbb{N}, \text{out}_i = \frac{x_i + y_i}{2} \]
A simple example

\[
\text{node average (} x, y : \text{real}) \text{ returns (} \text{out} : \text{real}); \\
\text{let} \\
\quad \text{out} = (x + y) / 2.0; \\
\text{tel}
\]

Transition system unrolled view:

\[
\begin{array}{ccccccc}
\text{clock ticks} & 0 & 1 & 2 & 3 & \ldots \\
\end{array}
\]
A simple example

define average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
end

Transition system unrolled view:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>y0</td>
<td>x1</td>
<td>y1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x0+y0</td>
<td></td>
<td>x1+y1</td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td></td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>out0</td>
<td></td>
<td>out1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x2+y2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3+y3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

clock ticks: 0 1 2 3 ...
A simple example

```plaintext
node average (x, y: real) returns (out: real);
let
    out = (x + y) / 2.0;
```

Transition system unrolled view:

```
<table>
<thead>
<tr>
<th></th>
<th>4.0</th>
<th>6.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5.0</td>
<td>6.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0.0</th>
<th>7.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.5</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>7.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>
```

clock ticks  0  1  2  3  ...
Combinational programs

- Basic types: bool, int, real

- Constants (i.e., constant streams):
  
<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>...</td>
</tr>
</tbody>
</table>

All classical operators are provided
Combinational programs

- Basic types: bool, int, real

- Constants (i.e., constant streams):

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>true</strong></td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>...</td>
</tr>
</tbody>
</table>

- Pointwise operators:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>x₀</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>y₀</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x + y</td>
<td>x₀ + y₀</td>
<td>x₁ + y₁</td>
<td>x₂ + y₂</td>
<td>x₃ + y₃</td>
<td>x₄ + y₄</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- All classical operators are provided
Conditional expressions:

```plaintext
node max (n1, n2: real) returns (out: real);
let
    out = if (n1 >= n2) then n1 else n2;
end
```

- Functional "if ... then ... else ...
- It is an expression, **not a statement**
Combinational programs

Conditional expressions:

```plaintext
node max (n1,n2: real) returns (out: real);
let
    out = if (n1 >= n2) then n1 else n2;
tel
```

- Functional “if ... then ... else ...”
- It is an expression, **not a statement**
  ```plaintext
  -- This does not compile
  if (a >= b) then m = a else m = b;
  ```
Combinational programs

Local variables:

```plaintext
def max (a, b: real) returns (out: real):
    var
        condition: bool;
    let
        out = if condition then a else b;
        condition = a >= b;
    end
```

Order does not matter
Set of equations not sequence of statements
Causality is resolved syntactically
Combinational programs

Local variables:

```
node max (a,b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
tel
```

- Order does not matter
- Set of equations not sequence of statements
Combinational programs

Local variables:

```plaintext
node max (a,b: real) returns (out: real);
var
  condition: bool;
let
  out = if condition then a else b;
  condition = a >= b;
```

- Order does not matter
- Set of equations not sequence of statements
- Causality is resolved syntactically
Combinational programs

Combinational recursion is forbidden:

\[ x = \frac{1}{2 - x}; \]
Combinational programs

Combinational recursion is forbidden:

\[
x = \frac{1}{2 - x};
\]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step
Combinational programs

Combinational recursion is forbidden:

\[ x = 1 / (2 - x); \]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step

Syntactic loop:

\[ x = \text{if } c \text{ then } y \text{ else } 0; \]
\[ y = \text{if } c \text{ then } 1 \text{ else } x; \]
Combinational programs

Combinational recursion is forbidden:

\[ x = 1 / (2 - x); \]

- has a unique integer solution: \( x = 1 \),
- but is not computable step by step

Syntactic loop:

\[ x = \text{if } c \text{ then } y \text{ else } 0; \]
\[ y = \text{if } c \text{ then } 1 \text{ else } x; \]

- not a real (semantic) loop:
  \[ x = \text{if } c \text{ then } 1 \text{ else } 0; \]
  \[ y = x; \]
- but still forbidden by Lustre
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (\textit{nil})
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (\textit{nil})
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization “->”:
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (\text{nil})
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization “->”:
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)

- Examples:

\[
\begin{array}{c|cccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & & & & & & &
\end{array}
\]
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (\textit{nil})
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization “\(\rightarrow\)”:
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)

- Examples:

  \[
  \begin{array}{c|cccccccc}
  \text{x} & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & \text{nil} & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  \end{array}
  \]
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (nil)
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization “->”:
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)

- Examples:

```
x          | x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | ...
pre x      | nil | x_0 | x_1 | x_2 | x_3 | x_4 | ...
y          | y_0 | y_1 | y_2 | y_3 | y_4 | y_5 | ...
x \rightarrow y
```
Memory programs

- Previous operator “pre”:
  \[(\text{pre } x)_0\] is undefined (nil)
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization “->”:
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)

- Examples:

<table>
<thead>
<tr>
<th>x</th>
<th>x₀</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
<th>x₅</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre x</td>
<td>nil</td>
<td>x₀</td>
<td>x₁</td>
<td>x₂</td>
<td>x₃</td>
<td>x₄</td>
<td>x₅</td>
</tr>
<tr>
<td>y</td>
<td>y₀</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>y₅</td>
<td>...</td>
</tr>
<tr>
<td>x \rightarrow y</td>
<td>x₀</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>y₅</td>
<td>...</td>
</tr>
</tbody>
</table>
Memory programs

- Previous operator "pre":
  
  \[(\text{pre } x)_0\] is undefined (\text{nil})
  \[(\text{pre } x)_i = x_{i-1}\] for \(i > 0\)

- Initialization "\(\rightarrow\)"
  
  \[(x \rightarrow y)_0 = x_0\]
  \[(x \rightarrow y)_i = y_i\] for \(i > 0\)

- Examples:

<table>
<thead>
<tr>
<th>(x)</th>
<th>(x_0)</th>
<th>(x_1)</th>
<th>(x_2)</th>
<th>(x_3)</th>
<th>(x_4)</th>
<th>(x_5)</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{pre } x)</td>
<td>\text{nil})</td>
<td>(x_0)</td>
<td>(x_1)</td>
<td>(x_2)</td>
<td>(x_3)</td>
<td>(x_4)</td>
<td>(x_5)</td>
</tr>
<tr>
<td>(y)</td>
<td>(y_0)</td>
<td>(y_1)</td>
<td>(y_2)</td>
<td>(y_3)</td>
<td>(y_4)</td>
<td>(y_5)</td>
<td>(y_6)</td>
</tr>
<tr>
<td>(x \rightarrow y)</td>
<td>(x_0)</td>
<td>(y_1)</td>
<td>(y_2)</td>
<td>(y_3)</td>
<td>(y_4)</td>
<td>(y_5)</td>
<td>(y_6)</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>(2 \rightarrow (\text{pre } x))</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memory programs

- Previous operator “pre”:
  
  \[(\text{pre } x)_0\] is undefined (\textit{nil})

  \[(\text{pre } x)_i = x_{i-1} \text{ for } i > 0\]

- Initialization “->”:
  
  \[(x \rightarrow y)_0 = x_0\]

  \[(x \rightarrow y)_i = y_i \text{ for } i > 0\]

- Examples:

  \[
  \begin{array}{c|cccccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & \ldots \\
  \text{pre } x & \text{nil} & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  y & y_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  x \rightarrow y & x_0 & y_1 & y_2 & y_3 & y_4 & y_5 & \ldots \\
  2 & 2 & 2 & 2 & 2 & 2 & 2 & \ldots \\
  2 \rightarrow (\text{pre } x) & 2 & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  \end{array}
  \]
Memory programs

Recursive definition using \texttt{pre}:

\[
\begin{align*}
n &= 0 \rightarrow 1 + \texttt{pre } n; \\
a &= \texttt{false} \rightarrow \texttt{not} \texttt{pre } a;
\end{align*}
\]

\begin{tabular}{l|l}
\texttt{n} & 0 \\
\texttt{a} & \texttt{false}
\end{tabular}
Recursive definition using $\text{pre}$:

\[
\begin{align*}
n &= 0 \rightarrow 1 + \text{pre } n; \\
a &= \text{false} \rightarrow \text{not pre } a;
\end{align*}
\]

\[
\begin{array}{c|ccccc}
  n & 0 & 1 & 2 & 3 & \ldots \\
  a & \text{false} & & & & \\
\end{array}
\]
Memory programs

Recursive definition using \texttt{pre}:

\begin{align*}
n &= 0 \rightarrow 1 + \texttt{pre} \ n; \\
a &= \texttt{false} \rightarrow \texttt{not} \ \texttt{pre} \ a;
\end{align*}

\begin{tabular}{c|cccccc}
  n & 0 & 1 & 2 & 3 & \ldots \\
  a & \texttt{false} & \texttt{true} & \texttt{false} & \texttt{true} & \ldots \\
\end{tabular}
node guess (signal: bool) returns (e: bool);
let
    e = false -> signal and not pre signal;
tel

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>1 1 0 1 0 ...</td>
</tr>
</tbody>
</table>
node guess (signal: bool) returns (e: bool);
let
    e = false -> signal and not pre signal;
tel

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
</tr>
</tbody>
</table>
node guess (signal: bool) returns (e: bool);
let
  e = false \rightarrow signal \text{ and not pre signal};
tel

<table>
<thead>
<tr>
<th>signal</th>
<th>0 1 1 0 1 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0 1 0 0 1 0 ...</td>
</tr>
</tbody>
</table>
Memory programs: examples

Raising edge:

```
node guess (signal: bool) returns (e: bool);
let
  e = false -> signal and not pre signal;
tel
```

<table>
<thead>
<tr>
<th>signal</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td></td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4</td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
  out1 = n -> if (n < pre out1) then n else pre out1;
  out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4 2 3 0 3 7 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4 2 2 0 0 0 0 ...</td>
</tr>
</tbody>
</table>
node guess (n: int) returns (out1, out2: int);
let
    out1 = n -> if (n < pre out1) then n else pre out1;
    out2 = n -> if (n > pre out2) then n else pre out2;
tel

<table>
<thead>
<tr>
<th>n</th>
<th>4  2  3  0  3  7  …</th>
</tr>
</thead>
<tbody>
<tr>
<td>out1</td>
<td>4  2  2  0  0  0  0  …</td>
</tr>
<tr>
<td>out2</td>
<td>4  4  4  4  4  4  7  …</td>
</tr>
</tbody>
</table>
Memory programs: examples

Min and max of a sequence:

```plaintext
node guess (n: int) returns (out1, out2: int);
let
    out1 = n -> if (n < pre out1) then n else pre out1;
    out2 = n -> if (n > pre out2) then n else pre out2;
tel
```

```
n   4 2 3 0 3 7 ...
out1 4 2 2 0 0 0 ...
out2 4 4 4 4 4 7 ...
```
Design a node

\[
\text{node switch (on, off: bool) returns (state: bool)};
\]

such that:

- state raises (false to true) if on;
- state falls (true to false) if off;
Exercises

Design a node

```python
node switch (on, off: bool) returns (state: bool);
```

such that:

- state raises (false to true) if on;
- state falls (true to false) if off;
- everything behaves as if state was false at the origin;
- switch must work properly even if on and off are the same
Compute the sequence 1, 1, 2, 3, 5, 8 ...
Exercises

Compute the sequence 1, 1, 2, 3, 5, 8, 13, 21 …

Fibonacci sequence:

\[ u_0 = u_1 = 1 \]
\[ u_n = u_{n-1} + u_{n-2} \quad \text{for } n \geq 2 \]
These notes are based on the following lectures notes:

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