Synchronous Languages—Lecture 18

Prof. Dr. Reinhard von Hanxleden

Christian-Albrechts Universität Kiel
Department of Computer Science
Real-Time Systems and Embedded Systems Group

4 Feb. 2019

Last compiled: February 4, 2019, 10:07 hrs

Lustre
Overview

A Short Tour

Examples

Clock Consistency

Arrays and Recursive Nodes
Lustre

- A synchronous data flow language
- Developed since 1984 at IMAG, Grenoble [HCRP91]
- Also graphical design entry available (SAGA)
- Moreover, the basis for SCADE, a tool used in software development for avionics and automotive industries
- Translatable to FSMs with finitely many control states
- Same advantages as Esterel for hardware and software design
Lustre Modules

General form:

```plaintext
node f(x₁:α₁, ..., xₙ:αₙ) returns (y₁:β₁,...,yₘ:βₘ)
var z₁:γ₁,...,zₖ:γₖ;
let
  z₁ = τ₁; ...; zₖ = τₖ;
  y₁ = π₁; ...; yₘ = πₖ;
assert ϕ₁; ...; assert ϕₗ;
tel
```

where

- **f** is the name of the module
- **Inputs** \(x_i\), outputs \(y_i\), and local variables \(z_j\)
- **Assertions** \(φ_i\) (boolean expressions)
Lustre Programs

- Lustre programs are a list of modules that are called **nodes**
- All nodes work synchronously, *i.e.* at the same speed
- Nodes communicate only via inputs and outputs
- No broadcasting of signals, no side effects
- **Equations** $z_i = \tau_i$ and $y_i = \pi_i$ are **not assignments**
- Equations must have solutions in the mathematical sense
Lustre Programs

- As $z_i = \tau_i$ and $y_i = \pi_i$ are equations, we have the Substitution Principle:
  The definitions $z_i = \tau_i$ and $y_i = \pi_i$ of a Lustre node allow one to replace $z_i$ by $\tau_i$ and $y_i$ by $\pi_i$.

- Behavior of $z_i$ and $y_i$ completely given by equations $z_i = \tau_i$ and $y_i = \pi_i$.
Assertions

- Assertions `assert ϕ` do not influence the behavior of the system.
- `assert ϕ` means that during execution, ϕ must invariantly hold.
- Equation `X = E` equivalent to assertion `assert(X = E)`.
- Assertions can be used to optimize the code generation.
- Assertions can be used for simulation and verification.
Data Streams

- All variables, constants, and all expressions are streams
- Streams can be composed to new streams
- Example: given \( x = (0, 1, 2, 3, 4, \ldots) \) and \( y = (0, 2, 4, 6, 8, \ldots) \), then \( x + y \) is the stream \( (0, 3, 6, 9, 12, \ldots) \)
- However, *streams may refer to different clocks*

\[\sim\] Each stream has a corresponding clock
Data Types

- Primitive data types: bool, int, real
  - Semantics is clear?
- Imported data types: type $\alpha$
  - Similar to Esterel
  - Data type is implemented in host language
- Tuples of types: $\alpha_1 \times \ldots \times \alpha_n$ is a type
  - Semantics is Cartesian product
Expressions (Streams)

- Every declared variable $x$ is an expression
- Boolean expressions:
  - $\tau_1$ and $\tau_2$, $\tau_1$ or $\tau_2$, not $\tau_1$
- Numeric expressions:
  - $\tau_1 + \tau_2$ and $\tau_1 - \tau_2$, $\tau_1 \times \tau_2$ and $\tau_1 / \tau_2$, $\tau_1 \text{ div } \tau_2$ and $\tau_1 \text{ mod } \tau_2$
- Relational expressions:
  - $\tau_1 = \tau_2$, $\tau_1 < \tau_2$, $\tau_1 \leq \tau_2$, $\tau_1 > \tau_2$, $\tau_1 \geq \tau_2$
- Conditional expressions:
  - if $b$ then $\tau_1$ else $\tau_2$ for all types
Node Expansion

▶ Assume implementation of a node $f$ with inputs $x_1 : \alpha_1, \ldots, x_n : \alpha_n$ and outputs $y_1 : \beta_1, \ldots, y_m : \beta_m$

▶ Then, $f$ can be used to create new stream expressions, e.g., $f(\tau_1, \ldots, \tau_n)$ is an expression
  ▶ Of type $\beta_1 \times \ldots \times \beta_m$
  ▶ If $(\tau_1, \ldots, \tau_n)$ has type $\alpha_1 \times \ldots \times \alpha_n$
Vector Notation of Nodes

By using tuple types for inputs, outputs, and local streams, we may consider just nodes like

\[
\text{node } f(x:\alpha) \text{ returns } (y:\beta) \\
\text{var } z:\gamma; \\
\text{let} \\
\quad z = \tau; \\
\quad y = \pi; \\
\quad \text{assert } \phi; \\
\text{tel}
\]
Clock-Operators

- All expressions are streams
- **Clock-operators** modify the temporal arrangement of streams
- Again, their results are streams
- The following clock operators are available:
  - `pre \( \tau \)` for every stream \( \tau \)
  - \( \tau_1 \rightarrow \tau_2 \), (pronounced “followed by”) where \( \tau_1 \) and \( \tau_2 \) have the same type
  - \( \tau_1 \) **when** \( \tau_2 \) where \( \tau_2 \) has boolean type (**downsampling**)
  - **current** \( \tau \) (**upsampling**)
Clock-Hierarchy

- As already mentioned, streams may refer to different clocks
- We associate with every expression a list of clocks
- A clock is thereby a stream $\varphi$ of boolean type
Clock-Hierarchy

- $\text{clocks}(\tau) := []$ for expressions without clock operators
- $\text{clocks}(\text{pre}(\tau)) := \text{clocks}(\tau)$
- $\text{clocks}(\tau_1 \rightarrow \tau_2) := \text{clocks}(\tau_1)$, where clocks($\tau_1$) = clocks($\tau_2$) is required
- $\text{clocks}(\tau \text{ when } \varphi) := [\varphi, c_1, \ldots, c_n]$, where clocks($\varphi$) = clocks($\tau$) = [c₁, ..., cₙ]
- $\text{clocks}(\text{current}(\tau)) := [c_2, \ldots, c_n]$, where clocks($\tau$) = [c₁, ..., cₙ]
Semantics of Clock-Operators

- $\llbracket \text{pre}(\tau) \rrbracket := (\bot, \tau_0, \tau_1, \ldots)$, provided that $\llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots)$

- $\llbracket \tau \rightarrow \pi \rrbracket := (\tau_0, \pi_1, \pi_2, \ldots)$, provided that $\llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots)$ and $\llbracket \pi \rrbracket = (\pi_0, \pi_1, \ldots)$

- $\llbracket \tau \; \text{when} \; \varphi \rrbracket = (\tau_{t_0}, \tau_{t_1}, \tau_{t_2}, \ldots)$, provided that
  - $\llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots)$
  - $\{t_0, t_1, \ldots\}$ is the set of points in time where $\llbracket \varphi \rrbracket$ holds

- $\llbracket \text{current}(\tau) \rrbracket = (\bot, \ldots, \bot, \tau_{t_0}, \ldots, \tau_{t_0}, \tau_{t_1}, \ldots, \tau_{t_1}, \tau_{t_2}, \ldots)$, provided that
  - $\llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots)$
  - $\{t_0, t_1, \ldots\}$ is the set of points in time where the highest clock of $\text{current}(\tau)$ holds
### Example for Semantics of Clock-Operators

<table>
<thead>
<tr>
<th>$\varphi$</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau$</td>
<td>$\tau_0$</td>
<td>$\tau_1$</td>
<td>$\tau_2$</td>
<td>$\tau_3$</td>
<td>$\tau_4$</td>
<td>$\tau_5$</td>
<td>$\tau_6$</td>
</tr>
</tbody>
</table>

| pre($\tau$) | $\bot$ | $\tau_0$ | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ | $\tau_5$ |
| $\tau \rightarrow$ pre($\tau$) | $\tau_0$ | $\tau_0$ | $\tau_1$ | $\tau_2$ | $\tau_3$ | $\tau_4$ | $\tau_5$ |
| $\tau$ when $\varphi$ | $\tau_1$ | $\tau_3$ | $\tau_6$ | |
| current($\tau$ when $\varphi$) | $\bot$ | $\tau_1$ | $\tau_1$ | $\tau_3$ | $\tau_3$ | $\tau_3$ | $\tau_6$ |

- Note: $[\tau \text{ when } \varphi] = (\tau_1, \tau_3, \tau_6, \ldots)$, i.e., gaps are not filled!
- This is done by current($\tau$ when $\varphi$)
Example for Semantics of Clock-Operators

<table>
<thead>
<tr>
<th>[ e = (1 \rightarrow \text{not } \text{pre}(e)) ]</th>
<th>0 1 0 1 0 1 0 ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ n = (0 \rightarrow \text{pre}(n)+1) ]</td>
<td>0 1 2 3 4 5 ...</td>
</tr>
<tr>
<td>[ \text{n when } e ]</td>
<td>0 2 4 ...</td>
</tr>
<tr>
<td>[ \text{current(n when e)} ]</td>
<td>0 0 2 2 4 4 ...</td>
</tr>
<tr>
<td>[ \text{current (n when e) div 2} ]</td>
<td>0 0 1 1 2 2 ...</td>
</tr>
</tbody>
</table>
### Example for Semantics of Clock-Operators

<table>
<thead>
<tr>
<th>expression</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 0 \rightarrow \text{pre}(n)+1 )</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
<tr>
<td>( d_2 = (n \div 2) \times 2 = n )</td>
<td>1 0 1 0 1 0 1 0 1 0 1 0</td>
</tr>
<tr>
<td>( n_2 = n \text{ when } d_2 )</td>
<td>0 2 4 6 8 10</td>
</tr>
<tr>
<td>( d_3 = (n \div 3) \times 3 = n )</td>
<td>1 0 0 1 0 0 1 0 0 1 0 0</td>
</tr>
<tr>
<td>( n_3 = n \text{ when } d_3 )</td>
<td>0 3 6 9</td>
</tr>
<tr>
<td>( d_3' = d_3 \text{ when } d_2 )</td>
<td>1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>( n_6 = n_2 \text{ when } d_3' )</td>
<td>0 6</td>
</tr>
<tr>
<td>( c_3 = \text{current}(n_2 \text{ when } d_3') )</td>
<td>0 0 0 0 6 6 6 6</td>
</tr>
</tbody>
</table>
Example: Counter

\[
\text{node } \text{Counter}(x_0, d: \text{int}; r: \text{bool}) \text{ returns } (n: \text{int}) \\
\begin{align*}
\text{let} & \quad n = x_0 \rightarrow \text{if } r \text{ then } x_0 \text{ else } \text{pre}(n) + d \\
\text{tel}
\end{align*}
\]

- Initial value of \( n \) is \( x_0 \)
- If no reset \( r \) then increment by \( d \)
- If reset by \( r \), then initialize with \( x_0 \)
- \( \text{Counter} \) can be used in other equations, e.g.
  - \( \text{ex1} = \text{Counter}(0, 2, 0) \) yields the even numbers
  - \( \text{ex2} = \text{Counter}(0, 1, \text{pre(ex2)} = 4) \) yields numbers mod 5
ABRO in Lustre

node EDGE(X:bool) returns (Y:bool);
let
  Y = false → X and not pre(X);
tel

node ABRO (A,B,R:bool) returns (O: bool);
  var seenA, seenB : bool;
let
  O = EDGE(seenA and seenB);
  seenA = false → not R and (A or pre(seenA));
  seenB = false → not R and (B or pre(seenB));
tel
Causality Problems in Lustre

- Synchronous languages have causality problems
- They arise if preconditions of actions are influenced by the actions
- Therefore they require to solve fixpoint equations
- Such equations may have none, one, or more than one solutions

〜 Analogous to Esterel, one may consider reactive, deterministic, logically correct, and constructive programs
Causality Problems in Lustre

- $x = \tau$ is acyclic, if $x$ does not occur in $\tau$ or does only occur as subterm $\text{pre}(x)$ in $\tau$

- **Examples:**
  - $a = a$ and $\text{pre}(a)$ is cyclic
  - $a = b$ and $\text{pre}(a)$ is acyclic

- Acyclic equations have a unique solution!

- Analyze cyclic equations to determine causality?

- **But:** Lustre only allows acyclic equation systems

- Sufficient for signal processing
Malik’s Example

- However, some interesting examples are cyclic

  ```plaintext
  y = if c then y_f else y_g;
  y_f = f(x_f);
  y_g = g(x_g);
  x_f = if c then y_g else x;
  x_g = if c then x else y_f;
  ```

- Implements if c then f(g(x)) else g(f(x)) with only one instance of f and g

- **Impossible without cycles**

---

Sharad Malik.  
*Analysis of cyclic combinatorial circuits.*  
Clock Consistency

Consider the following equations:

\[
\begin{align*}
  b &= 0 \rightarrow \text{not pre}(b); \\
  y &= x + (x \text{ when } b)
\end{align*}
\]

- We obtain the following:

<table>
<thead>
<tr>
<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>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>...</td>
</tr>
<tr>
<td>x when b</td>
<td>x₁</td>
<td>x₃</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x + (x when b)</td>
<td>x₀ + x₁</td>
<td>x₁ + x₃</td>
<td>x₂ + x₅</td>
<td>x₃ + x₇</td>
<td>x₄ + x₉</td>
<td>...</td>
</tr>
</tbody>
</table>

- To compute \( y_i := x_i + x_{2i+1} \), we have to store \( x_i, \ldots, x_{2i+1} \)

- Problem: not possible with finite memory
Clock Consistency

- Expressions like $x + (x \text{ when } b)$ are not allowed
- **Only streams at the same clock can be combined**
- What is the ‘same’ clock?
- Undecidable to prove this semantically
- Check syntactically
Clock Consistency

- Two streams have the same clock if their clock can be syntactically unified
- Example:

  \[ x = a \text{ when } (y > z); \]
  \[ y = b + c; \]
  \[ u = d \text{ when } (b + c > z); \]
  \[ v = e \text{ when } (z < y); \]

- \( x \) and \( u \) have the same clock
- \( x \) and \( v \) do not have the same clock
Arrays

- Given type $\alpha$, $\alpha^n$ defines an array with $n$ entries of type $\alpha$
- Example: $x: \text{bool}^n$
- The bounds of an array must be known at compile time, the compiler simply transforms an array of $n$ values into $n$ different variables.
- The $i$-th element of an array $X$ is accessed by $X[i]$.
- $X[i..j]$ with $i \leq j$ denotes the array made of elements $i$ to $j$ of $X$.
- Beside being syntactical sugar, arrays allow to combine variables for better hardware implementation.
Example for Arrays

```plaintext
node DELAY (const d: int; X: bool) returns (Y: bool);
    var A: bool^(d+1);
    let
        A[0] = X;
        A[1..d] = (false^(d)) → pre(A[0..d--1]);
        Y = A[d];
    tel
```

- \(\text{false}^{(d)}\) denotes the boolean array of length \(d\), which entries are all \text{false}
- Observe that \text{pre} and \(\rightarrow\) can take arrays as parameters
- Since \(d\) must be known at compile time, this node cannot be compiled in isolation
- The node outputs each input delayed by \(d\) steps.
- So \(Y_n = X_{n-d}\) with \(Y_n = \text{false}\) for \(n < d\)
Static Recursion

- Functional languages usually make use of recursively defined functions
- **Problem:** termination of recursion in general undecidable
- Primitive recursive functions guarantee termination
- **Problem:** still with primitive recursive functions, the reaction time depends heavily on the input data
- **Static recursion:** recursion only at compile time
- **Observe:** If the recursion is not bounded, the compilation will not stop.
Example for Static Recursion

- **Disjunction of boolean array**
  ```plaintext
  node BigOr(const n:int; x: bool^n) returns (y:bool)
  let
  y = with n=1 then x[0]
      else x[0] or BigOr(n--1,x[1..n--1]);
  tel
  ```

- Constant $n$ must be known at compile time
- Node is unrolled before further compilation
Example for Maximum Computation

Static recursion allows logarithmic circuits:

```plaintext
node Max(const n:int; x:int^n) returns (y:int)
    var y_1,y_2: int;
    let
        y_1 = with n=1 then x[0]
            else Max(n div 2,x[0..(n div 2)--1]);
        y_2 = with n=1 then x[0]
            else Max((n+1) div 2, x[(n div 2)..n--1]);
    y = if y_1 >= y_2 then y_1 else y_2;
    tel
```
Delay node with recursion

```plaintext
node REC_DELAY (const d: int; X: bool) returns (Y: bool);
let
    Y = with d=0 then X
    else false → pre(REC_DELAY(d--1, X));
tel
```

A call `REC_DELAY(3, X)` is compiled into something like:

```plaintext
Y = false → pre(Y2)
Y2 = false → pre(Y1)
Y1 = false → pre(Y0)
Y0 = X;
```
Summary

- Lustre is a synchronous dataflow language.
- The core Lustre language are boolean equations and clock operators `pre, ->, when, and current`.
- Additional datatypes for real and integer numbers are also implemented.
- User types can be defined as in Esterel.
- Lustre only allows acyclic programs.
- Clock consistency is checked syntactically.
- Lustre offers arrays and recursion, but both array-size and number of recursive calls must be known at compile time.
To Go Further
