Synchronous Languages:
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 (now marketed by Esterel Technologies), 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

Thanks to Klaus Schneider
(http://rsg.informatik.uni-kl.de/people/schneider/) for providing part of the following material
Lustre Modules

General form:

```plaintext
node f(x_1:α_1, ..., x_n:α_n) returns (y_1:β_1, ..., y_m:β_m)
var z_1:γ_1, ..., z_k:γ_k;
let
  z_1 = τ_1; ...; z_k = τ_k;
  y_1 = π_1; ...; y_m = π_k;
  assert φ_1; ...; assert φ_ℓ;
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 \varphi` do not influence the behavior of the system.
- `assert \varphi` means that during execution, \varphi 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
- Each stream has a corresponding clock
Data Types

- Primitive data types: `bool`, `int`, `real`
- Imported data types: type `α`
  - Similar to Esterel
  - Data type is implemented in host language
- Tuples of types: `α₁ × ... × αₙ` 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 \( \tau_1 \mod \tau_2 \)
- 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

```
node f(x:α) returns (y:β)
var z:γ;
let
  z = τ;
  y = π;
  assert ϕ;
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 τ` for every stream `τ`
  - `τ₁ → τ₂`, (pronounced “followed by”) where `τ₁` and `τ₂` have the same type
  - `τ₁ when τ₂` where `τ₂` has boolean type (**downsampling**)
  - `current τ` (**upsampling**)

▶ All expressions are streams
▶ **Clock-operators** modify the temporal arrangement of streams
▶ Again, their results are streams
▶ The following clock operators are available:
  ▶ `pre τ` for every stream `τ`
  ▶ `τ₁ → τ₂`, (pronounced “followed by”) where `τ₁` and `τ₂` have the same type
  ▶ `τ₁ when τ₂` where `τ₂` has boolean type (**downsampling**)
  ▶ `current τ` (**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
- Whenever this stream $\varphi$ is true (considered at its clock), a point in time is selected that belongs to the new clock hierarchy
Clock-Hierarchy

- clocks(\tau) := [] if expression \tau does not contain any clock operator
- clocks(pre(\tau)) := clocks(\tau)
- clocks(\tau_1 \rightarrow \tau_2) := clocks(\tau_1),
  where clocks(\tau_1) = clocks(\tau_2) is required
- clocks(\tau \text{ when } \varphi) := [\varphi, c_1, \ldots, c_n],
  where clocks(\varphi) = clocks(\tau) = [c_1, \ldots, c_n]
- clocks(current(\tau)) := [c_2, \ldots, c_n],
  where clocks(\tau) = [c_1, \ldots, c_n]
Semantics of Clock-Operators

- \[ \llbracket \text{pre}(\tau) \rrbracket := (\bot, \tau_0, \tau_1, \ldots), \text{ provided that } \llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots) \]
- \[ \llbracket \tau \rightarrow \pi \rrbracket := (\tau_0, \pi_1, \pi_2, \ldots), \text{ provided that } \llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots) \text{ and } \llbracket \pi \rrbracket = (\pi_0, \pi_1, \ldots) \]
- \[ \llbracket \tau \text{ when } \varphi \rrbracket = (\tau_{t_0}, \tau_{t_1}, \tau_{t_2}, \ldots), \text{ provided that} \]
  - \[ \llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots) \]
  - \{t_0, t_1, \ldots\} is the set of points in time where \[ \llbracket \varphi \rrbracket \] holds
- \[ \llbracket \text{current}(\tau) \rrbracket = (\bot, \ldots, \bot, \tau_{t_0}, \ldots, \tau_{t_0}, \tau_{t_1}, \ldots, \tau_{t_1}, \tau_{t_2}, \ldots), \text{ provided that} \]
  - \[ \llbracket \tau \rrbracket = (\tau_0, \tau_1, \ldots) \]
  - \{t_0, t_1, \ldots\} is the set of points in time where the highest clock of \text{current}(\tau) \text{ holds}
Example for Semantics of Clock-Operators

Note: \([\tau \text{ when } \varphi] = (\tau_1, \tau_3, \tau_6, \ldots)\), i.e., gaps are not filled!

This is done by \(\text{current}(\tau \text{ when } \varphi)\)
**Example for Semantics of Clock-Operators**

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 0 \to \text{pre}(n)+1 )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>( d2 = (n \div 2) \times 2 = n )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( n2 = n \text{ when } d2 )</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( d3 = (n \div 3) \times 3 = n )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( n3 = n \text{ when } d3 )</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( d3' = d3 \text{ when } d2 )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n6 = n2 \text{ when } d3' )</td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( c3 = \text{current}(n2 \text{ when } d3') )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: Counter

node Counter(x0, d:int; r:bool) returns (n:int)
let
  n = x0 -> if r then x0 else pre(n) + d
tel

- Initial value of $n$ is $x_0$
- If no reset $r$ then increment by $d$
- If reset by $r$, then initialize with $x_0$
- $Counter$ can be used in other equations, e.g.
  - $even = Counter(0, 2, 0)$ yields the even numbers
  - $mod_5 = Counter(0, 1, \text{pre}(mod_5) = 4)$ 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

\[
\begin{align*}
    y &= \text{if } c \text{ then } y_f \text{ else } y_g; \\
    y_f &= f(x_f); \\
    y_g &= g(x_g); \\
    x_f &= \text{if } c \text{ then } y_g \text{ else } x; \\
    x_g &= \text{if } c \text{ then } x \text{ else } y_f;
\end{align*}
\]

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

- **Impossible without cycles**

Sharad Malik.

*Analysis of cyclic combinatorial circuits.*

Clock Consistency

Consider the following equations:

\[ b = 0 \rightarrow \text{not } \text{pre}(b); \]
\[ y = x + (x \text{ when } b) \]

We obtain the following:

<table>
<thead>
<tr>
<th></th>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</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 \text{ when } b )</td>
<td>( x_1 )</td>
<td>( x_3 )</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x + (x \text{ when } b) )</td>
<td>( x_0 + x_1 )</td>
<td>( x_1 + x_3 )</td>
<td>( x_2 + x_5 )</td>
<td>( x_3 + x_7 )</td>
<td>( x_4 + x_9 )</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
```

- `false^(d)` denotes the boolean array of length $d$, which entries are all false.
- Observe that `pre` and `->` 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 $Y_n = false$ 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

```
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:

```
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