Overview

A Short Tour

Examples

Clock Consistency

Arrays and Recursive Nodes

Part of this lecture is based on material kindly provided by Klaus Schneider.
http://rsg.informatik.uni-kl.de/people/schneider/
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:

```
node f(x₁:α₁, ..., xₙ:αₙ) returns (y₁:β₁, ..., yₘ:βₘ)
var z₁:γ₁, ..., zₖ:γₖ;
let
  z₁ = τ₁; ...; zₖ = τₖ;
  y₁ = π₁; ...; yₘ = πₘ;
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 \( \varphi_i \) (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 \( z_i = τ_i \) and \( y_i = π_i \) are not assignments
- Equations must have solutions in the mathematical sense

As \( z_i = τ_i \) and \( y_i = π_i \) are equations, we have the Substitution Principle:

The definitions \( z_i = τ_i \) and \( y_i = π_i \) of a Lustre node allow one to replace \( z_i \) by \( τ_i \) and \( y_i \) by \( π_i \).

Behavior of \( z_i \) and \( y_i \) completely given by equations \( z_i = τ_i \) and \( y_i = π_i \).
Assertions

- Assertions assert $\varphi$ do not influence the behavior of the system
- $\text{assert } \varphi$ means that during execution, $\varphi$ must invariantly hold
- Equation $X = E$ equivalent to assertion $\text{assert}(X = E)$
- Assertions can be used to optimize the code generation
- Assertions can be used for simulation and verification

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

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

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 $\tau_1 / \tau_2$, $\tau_1 \div \tau_2$ 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 $x_1 : \alpha_1$, ..., $x_n : \alpha_n$ and outputs $y_1 : \beta_1$, ..., $y_m : \beta_m$
- 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 $\alpha_1 \times \ldots \times \alpha_n$

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:
  - $\text{pre } \tau$ for every stream $\tau$
  - $\tau_1 \rightarrow \tau_2$, (pronounced “followed by”) where $\tau_1$ and $\tau_2$ have the same type
  - $\tau_1 \text{ when } \tau_2$ where $\tau_2$ has boolean type (downsampling)
  - $\text{current } \tau$ (upsampling)

Vector Notation of Nodes

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

```plaintext
node f(x:α) returns (y:β)
var z:γ;
let
  z = τ;
  y = π;
assert ϕ;
tel
```
Clock-Hierarchy

- clocks(τ) := [] for expressions without clock operators
- clocks(pre(τ)) := clocks(τ)
- clocks(τ₁ -> τ₂) := clocks(τ₁), where clocks(τ₁) = clocks(τ₂) is required
- clocks(τ when ϕ) := [ϕ, c₁, ..., cₙ], where clocks(ϕ) = clocks(τ) = [c₁, ..., cₙ]
- clocks(current(τ)) := [c₂, ..., cₙ], where clocks(τ) = [c₁, ..., cₙ]

Example for Semantics of Clock-Operators

<table>
<thead>
<tr>
<th>τ</th>
<th>0 1 0 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre(τ)</td>
<td>0 0 1 0 1</td>
</tr>
<tr>
<td>τ -&gt; pre(τ)</td>
<td>0 0 1 0 1</td>
</tr>
<tr>
<td>τ when ϕ</td>
<td>1 0 1 0 1</td>
</tr>
<tr>
<td>current(τ when ϕ)</td>
<td>1 0 1 0 1</td>
</tr>
</tbody>
</table>

- Note: [τ when ϕ] = (τ₁, τ₃, τ₆, ...), i.e., gaps are not filled!
- This is done by current(τ when ϕ)

Semantics of Clock-Operators

- [pre(τ)] := (∅, τ₀, τ₁, ...), provided that [τ] = (τ₀, τ₁, ...)
- [τ -> π] := (τ₀, π₁, τ₂, ...), provided that [τ] = (τ₀, τ₁, ...) and [π] = (π₀, π₁, ...)
- [τ when ϕ] := (τ₀, τ₁, τ₂, ...), provided that
  - [τ] = (τ₀, τ₁, ...)
  - {t₀, t₁, ...} is the set of points in time where [ϕ] holds
- [current(τ)] := (⊥, ..., ⊥, τ₀, τ₁, ..., τ₀, τ₁, ..., τ₀, τ₁, ..., τ₀, τ₁, ..., τ₀, τ₁, ..., τ₀, τ₁, ...), provided that
  - [τ] = (τ₀, τ₁, ...)
  - {t₀, t₁, ...} is the set of points in time where the highest clock of current(τ) holds

When inputs run on different clocks than the basic clock of the node, these clocks must be explicit inputs. Outputs of a node may only run on different clocks, when these clocks are known at the outside.
Therefore, all externally visible variables must run on the basic clock, i.e., they must be masked using current.
Example for Semantics of Clock-Operators

\[
\begin{array}{cccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & \ldots \\
1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
n = (0 \to \text{pre}(n)+1) & 0 & 1 & 2 & 3 & 4 & 5 & \ldots \\
e = (1 \to \text{not pre}(e)) & 1 & 0 & 1 & 0 & 1 & 0 & \ldots \\
n \text{ when } e & 0 & 2 & 4 & \ldots \\
\text{current} (n \text{ when } e) & 0 & 0 & 2 & 2 & 4 & 4 & \ldots \\
\text{current} (n \text{ when } e) \text{ div } 2 & 0 & 0 & 1 & 1 & 2 & 2 & \ldots \\
\end{array}
\]

Example: Counter

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

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

Example for Semantics of Clock-Operators

\[
\begin{array}{cccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
n = (0 \to \text{pre}(n)+1) & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
d2 = (n \div 2) \times 2 = n & 0 & 2 & 4 & 6 & 8 & 10 \\
n2 = n \text{ when } d2 & 0 & 2 & 4 & 6 & 8 & 10 \\
n3 = n \text{ when } d3 & 0 & 3 & 6 & 9 \\
n3' = d3 \text{ when } d2 & 0 & 0 & 0 & 0 & 0 \\
n6 = n2 \text{ when } d3' & 0 & 0 & 0 & 0 \\
c3 = \text{current}(n2 \text{ when } d3') & 0 & 0 & 0 & 6 & 6 & 6 \\
\end{array}
\]

ABRO in Lustre

node EDGE\(X: \text{bool}\) returns \(Y: \text{bool}\)
\[
\begin{align*}
\text{let} \\
Y &= \text{false} \rightarrow X \text{ and not pre}(X) \\
tel
\end{align*}
\]

node ABRO \(A, B, R: \text{bool}\) returns \(O: \text{bool}\)
\[
\begin{align*}
\text{var seenA, seenB : bool;}
\text{let} \\
O &= \text{EDGE(seenA and seenB)}; \\
\text{seenA} &= \text{false} \rightarrow \text{not } R \text{ and } (A \text{ or pre(seenA)}); \\
\text{seenB} &= \text{false} \rightarrow \text{not } R \text{ and } (B \text{ or pre(seenB)}); \\
tel
\end{align*}
\]
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

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 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:

\[
\begin{align*}
  b &= 0 \rightarrow \neg \text{pre}(b); \\
  y &= x + (x \text{ when } b)
\end{align*}
\]

- We obtain the following:

\[
\begin{array}{cccccc}
  x & x_0 & x_1 & x_2 & x_3 & x_4 & \ldots \\
  b & 0 & 1 & 0 & 1 & 0 & \ldots \\
  x \text{ when } b & x_1 & x_3 & x_5 & x_7 & x_9 & \ldots \\
  x + (x \text{ when } b) & x_0 + x_1 & x_1 + x_3 & x_3 + x_5 & x_5 + x_7 & x_7 + x_9 & \ldots 
\end{array}
\]

- 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

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

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

- $\text{false}^d$ denotes the boolean array of length $d$, which entries are all 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
  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