Synchronous Languages—Lecture 16

Prof. Dr. Reinhard von Hanxleden

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

23 June 2020

Last compiled: June 30, 2020, 10:57 hrs

Lustre

A Short Tour
Examples
Clock Consistency
Arrays and Recursive Nodes

The 5-Minute Review Session

1. In sequential constructiveness, what is the *iur-protocol*?
2. When are threads *statically concurrent*?
3. What is a characteristic of the causality handling and compilation in the Blech language?
4. In addition to event-triggered execution, which other execution models do you know?
5. What is the idea of *dynamic ticks*?

---

1. For run-time concurrent variable accesses, for each variable, must first initialize (confluent absolute writes), then update (confluent relative writes), then read
2. Threads are statically concurrent when they are descendants of distinct threads sharing a common fork node (the least-common-ancestor fork, or lca fork). Alternatively: if their least common ancestor in thread tree is a fork node.
3. In Blech, causality issues are handled locally. Sub programs are black boxes, can be compiled separately.
4. Time-triggered, time-event-triggered, eager.
5. The tick function computes when—at the latest—the next call to the tick function should occur.

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

$\leadsto$ 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 = 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

A Short Tour | Lustre
Examples | Data Streams
Clock Consistency | Node Expansion
Arrays and Recursive Nodes | Clock Operators

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

A Short Tour | Lustre
Examples | Data Streams
Clock Consistency | Node Expansion
Arrays and Recursive Nodes | Clock Operators

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

Data Streams

- All variables, constants, and all expressions are streams, *i. e.*, sequences of values of a certain type
- 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**

$\leadsto$ Each stream has a corresponding clock, which filters out elements whenever the clock is false

- Per default, streams run on the base clock, which is always true

Data Types

- Primitive data types: `bool, int, real`
  - Semantics is clear?
- Imported data types: `type` α
  - 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:
  - τ_1 `and` τ_2, τ_1 `or` τ_2, `not` τ_1
- Numeric expressions:
  - $\tau_1 + \tau_2$ and $\tau_1 - \tau_2$, $\tau_1 * \tau_2$ and τ_1 / τ_2, τ_1 `div` τ_2 and τ_1 `mod` τ_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` τ_1 `else` τ_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
  - τ_1 `->` τ_2, (initialization) where τ_1 and τ_2 have the same type
  - τ_1 `when` τ_2 where τ_2 has boolean type (downsampling)
  - `current` τ (upsampling)

A Short Tour | Lustre
Examples | Data Streams
Clock Consistency | Node Expansion
Arrays and Recursive Nodes | Clock Operators

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 φ of boolean type

A Short Tour | Lustre
Examples | Data Streams
Clock Consistency | Node Expansion
Arrays and Recursive Nodes | Clock Operators

Clock-Hierarchy

- clocks(τ) := [] for expressions without clock operators
- clocks(`pre`(τ)) := clocks(τ)
- clocks(τ_1 `->` τ_2) := clocks(τ_1),
  where clocks(τ_1) = clocks(τ_2) is required
- clocks(τ `when` φ) := $[\varphi, c_1, \ldots, c_n]$,
  where clocks(φ) = clocks(τ) = $[c_1, \ldots, c_n]$
- clocks(`current`(τ)) := $[c_2, \ldots, c_n]$,
  where clocks(τ) = $[c_1, \ldots, c_n]$

A Short Tour | Lustre
Examples | Data Streams
Clock Consistency | Node Expansion
Arrays and Recursive Nodes | Clock Operators

Semantics of Clock-Operators

- $[\![\texttt{pre}(\tau)]\!] := (\bot, \tau_0, \tau_1, \ldots)$, provided that $[\![\tau]\!] = (\tau_0, \tau_1, \ldots)$
- $[\![\tau\ \texttt{->}\ \pi]\!] := (\tau_0, \pi_1, \pi_2, \ldots)$,
  provided that $[\![\tau]\!] = (\tau_0, \tau_1, \ldots)$ and $[\![\pi]\!] = (\pi_0, \pi_1, \ldots)$
- $[\![\tau\ \texttt{when}\ \varphi]\!] = (\tau_{t_0}, \tau_{t_1}, \tau_{t_2}, \ldots)$, provided that
  - $[\![\tau]\!] = (\tau_0, \tau_1, \ldots)$
  - $\{t_0, t_1, \ldots\}$ is the set of points in time where $[\![\varphi]\!]$ holds
- $[\![\texttt{current}(\tau)]\!] = (\bot, \ldots, \bot, \tau_0, \ldots, \tau_0, \tau_1, \ldots, \tau_1, \tau_2, \ldots)$,
  provided that
  - $[\![\tau]\!] = (\tau_0, \tau_1, \ldots)$
  - Stream holds value of τ from last tick of *clock of* clock of τ

Example for Semantics of Clock-Operators

φ	0	1	0	1	0	0	1
τ	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
pre(τ)	$\perp$	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5
τ -> pre(τ)	τ_0	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5
τ when φ		τ_1		τ_3			τ_6
current(τ when φ)	$\perp$	τ_1	τ_1	τ_3	τ_3	τ_3	τ_6

- Note: $[\![\tau \texttt{ when } \varphi]\!] = (\tau_1, \tau_3, \tau_6, \ldots)$, *i. e.*, **gaps are not filled!**
- This is done by current(τ when φ)

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

0	0	0	0	0	0	0	...
1	1	1	1	1	1	1	...
n = (0 -> pre(n)+1)	0	1	2	3	4	5	...
e = (1 -> not pre(e))	1	0	1	0	1	0	...
n when e	0		2		4		...
current(n when e)	0	0	2	2	4	4	...
current (n when e) div 2	0	0	1	1	2	2	...

Example for Semantics of Clock-Operators

n = 0 -> pre(n)+1	0	1	2	3	4	5	6	7	8	9	10	11
d2 = (n div 2)*2 = n	1	0	1	0	1	0	1	0	1	0	1	0
n2 = n when d2	0		2		4		6		8		10	
d3 = (n div 3)*3 = n	1	0	0	1	0	0	1	0	0	1	0	0
n3 = n when d3	0			3			6			9		
d3' = d3 when d2	1		0		0		1		0		0	
n6 = n2 when d3'	0						6					
c3 = current(n2 when d3')	0		0		0		6		6		6	

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 $x0$
- 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.
  - $ex1 = Counter(0, 2, 0)$ yields the even numbers
  - $ex2 = Counter(0, 1, \texttt{pre}(ex2) = 4)$ 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 τ or does only occur as subterm $\texttt{pre}(x)$ in τ
- Examples:
  - `a = a and pre(a)` is cyclic
  - `a = b and 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

```
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.
in IEEE Transactions on Computer-Aided Design, 1994

Clock Consistency

Consider the following equations:

```
b = 0→not pre(b);
y = x + (x when b)
```

- We obtain the following:

x	x_0	x_1	x_2	x_3	x_4	...
b	0	1	0	1	0	...
x when b		x_1		x_3		...
$x + (x$ when $b)$	$x_0 + x_1$	$x_1 + x_3$	$x_2 + x_5$	$x_3 + x_7$	$x_4 + x_9$	...

- 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 $x + (x \texttt{ when } b)$ 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:

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

- x and u have the same clock
- x and v do not have the same clock

Arrays

- Given type α, α^n defines an array with n entries of type α
- Example: `x: 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))-> 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 $Y_n = \mathit{false}$ 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:

```
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

- Nicolas Halbwachs and Pascal Raymond, A Tutorial of Lustre, 2002 `http://www-verimag.imag.fr/~halbwach/lustre-tutorial.html`
- Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud,The Synchronous Data-Flow Programming Language Lustre, In Proceedings of the IEEE, 79:9, September 1991, `http://www-verimag.imag.fr/~halbwach/lustre:ieee.html`