An Introduction to Lustre

Monday Oct 06, 2014

Philipp Rümmer
Uppsala University
Philipp.Ruemmer@it.uu.se
ES Programming languages

• Which language to write embedded software in?
  - Traditional: low-level languages, C
  - Trends: high-level, declarative, model-based, component-based languages
Lustre, synchronous prog.

- **Lustre**, Esterel, Signal
- Execution governed by a global clock, static scheduling
- Determinism is guaranteed (despite concurrency)
- Same language used for specification, modelling, prototyping, implementation
Lecture outline

- History of Lustre
- Overview of syntax + semantics
- Tutorial; Lustre by example

Next lecture: verification

- Borrowed some material from slides by Pascal Raymond, Nicolas Halbwachs, Cesare Tinelli
History of Lustre

- Invented in 1980’s at Verimag (Fr)
- Continuously developed since then
- Currently:
  - Academic versions + compilers (Lustre V4, Lustre V6)
  - Commercial version (SCADE, Esterel Technologies)
Early applications

• 1979-1984: first versions of Lustre
• 1986: tool Saga (based on Lustre) to develop the monitoring and emergency stop system of a nuclear plant
• At the same time, a similar tool (SAO) was used to develop the fly-by-wire and flight control of the Airbus A320
• Both approaches were later combined by company Verilog → SCADE
History of Lustre/SCADE

• Nowadays, one of the standard languages for safety-critical systems
  • Avionics, automotive, etc.
  • Certified tools
    (SCADE compiler was one of the first commercial certified compilers)
• E.g., significant portion of A380 code is written in SCADE
Ideas that led to Lustre

- Embedded software replaces previous technologies
  - analogue systems, switching networks, hardware...
- Most embedded software is not developed by computer scientists, but rather by control engineers used with previous technologies (and this is still true!)
Ideas that led to Lustre (2)

- These people are used to specific formalisms:
  - differential or finite-difference equations, analogue diagrams, "block-diagrams"...

- These data-flow formalisms enjoy some nice properties:
  - simple formal (functional and temporal) semantics, implicit parallelism
Ideas that led to Lustre (3)

• Idea: specialize our formalism into a programming language
  • (discrete time, executable semantics)
  → Lustre

• First versions of Simulink were developed at the same time
  → similar concepts
Lustre paradigms

- **Dataflow** language
  - similar to Simulink, but textual + time-discrete
  - changes force propagation

- **Synchronous**
  - program can have concurrent tasks, but all tasks run on the same clock; static scheduling (similar to synchronous hardware circuits)
  - good for quick reactions to environment
Lustre paradigms (2)

- **Declarative**
  - similar to functional languages
  - definitions instead of assignments

- Simple + modular language
Synchronous language family

- **Lustre**
  - Synchronous + dataflow

- **Esterel**
  - Synchronous + imperative

- **Signal**
  - “Polychronous” → multiple top-level clocks possible
Tool chains

- Lustre programs can be compiled to different target languages
  - C
  - VHDL → hardware
  - ...
- Good V&V support
  - automatic testing
  - static verification, model checking
Main concepts

- **Nodes**
  - programs or sub-programs
  - collections of flow definitions

- **Flows/streams**
  - infinite sequence of values
    → e.g. stream of inputs or outputs
  - represented using variables
  - defined equationally (acyclic)

- Ignored here: **Clocks**
Node syntax

```
node name(parameters) returns(vals);
[var local_variable_list;]
let
  flow definition;
  flow definition;
  ...
let
```

Order is not important!
Basic types

- **bool**
  - true, false, and, or, not, xor, =>
  - if ... then ... else ...

- **int, real**
  - machine integers, floating-point num.
  - +, -, *, /, div, mod, <>, <, <=, >, >=

- **Tuples**
  - Arbitrary combinations of bool, int, real, & tuple terms
  - Used to return multiple values
Variable declarations, comments

X : int;
A, B : bool;
C : bool; D : int;

-- Comments!
The Luke tool

- Command line simulator & verifier
- Fragment of Lustre (v4) language
  - does not support arrays, const, assert, #, when, current, real
  - allows non-standard structures: nodes with no inputs; =, <> can be used on type bool
- Outputs simulations & counterexamples to Javascript webpage
Examples ...

• Luke binaries:
  • Linux: http://bit.ly/1n79Bnc
  • Solaris: http://bit.ly/1EhG6Vc
Lustre is a declarative language!
Consequences of declarativeness

- Definitions of flows are **equations**, not assignments!
- Order is irrelevant:
  
  ```
  y = x + 1;
  z = y + 1;
  z = y + 1;
  y = x + 1;
  ```

  is the same as

  ```
  z = y + 1;
  y = x + 1;
  ```

- No side effects
Consequences of declarat. (2)

- **Cyclic** definitions are not allowed:
  
  ```
  y = x + 1;
  z = y + 1;
  x = z + 1;
  ```

  (this gives an error message during compilation/simulation)

- Also across multiple nodes!
Warning: functional if-then-else

• **Never** write something like this:

```plaintext
node Abs (x : int) returns (y : int);
let
  if x >= 0 then y = x else y = -x;
tel
```

• Correct version:

```plaintext
node Abs (x : int) returns (y : int);
let
  y = if x >= 0 then x else -x;
tel
```

Similar to `?:` in C
The $\text{pre}$ operator

- Access values of variables in the previous cycle:

$$X = (X_0, X_1, X_2, X_3, \ldots)$$

$$\text{pre } X = (\text{nil}, X_0, X_1, X_2, \ldots)$$
The followed-by operator $\rightarrow$

- Choose the initial element of a flow:

$$X = (X_0, X_1, X_2, X_3, \ldots)$$
$$Y = (Y_0, Y_1, Y_2, Y_3, \ldots)$$

$$X \rightarrow Y = (X_0, Y_1, Y_2, Y_3, \ldots)$$

- Typical use: $0 \rightarrow \text{pre (…)}$

- Be careful: $\rightarrow$ binds very weakly:

$X$ and false $\rightarrow$ pre $Y$

means

$$(X \text{ and false}) \rightarrow \text{pre } Y$$
Use of -> and pre

- -> and pre are commonly used to implement *iteration*
- The two operators replace loops
Examples ...
MultiStateSwitch

Switch is released

OFF

Switch is released

Switch pressed and held for $A$ ms

ON 2

held additional $B$ ms

ON 1
Traffic lights

System of two traffic lights, governing a junction of two (one-way) streets. In the default case, traffic light 1 is green, traffic light 2 is red. When a car is detected at traffic light 2 (the carSensor input), the system switches traffic light 1 to red, light 2 to green, waits some amount of time, and then switches back to the default situation.
Luke usage

- **Simulation:**
  
  ```
  luke --node top_node filename
  ```

- **Verification:**
  
  ```
  luke --node top_node --verify filename
  ```

  returns either “Valid. All checks succeeded. Maximal depth was n” or “Falsified output ‘X’ in node ‘Y’ at depth n” along with a counterexample.
Further Lustre features not supported in Luke

- Clocks
  - Used to delay sampling, execution
  - Operators: \texttt{when}, \texttt{current}
- assert, const, \#}
- Invocation of external functions
- Arrays, recursion, higher-order functions
SCADE features not supported in Luke

- case :: switching
- fby(x, n, i): n-fold followed-by + pre
  - Guarded delay
  - i -> pre (i -> pre ...)
- contact
  - Guarded clock change
Further reading


- A tutorial of Lustre: http://www-verimag.imag.fr/~halbwach/PS/tutorial.ps

- Slides by Pascal Raymond, Nicolas Halbwachs:
  http://pop-art.inrialpes.fr/~girault/Synchron06/Slides/halbwachs

Next lecture

- How to **specify** and **analyse** Lustre programs