The synchronous data-flow language Lustre

Nicolas Halbwachs
Verimag-CNRS
Grenoble – France

1 Introduction

The synchronous language Lustre was designed in the eighties, and resulted in the industrial software development tool Scade\(^1\), which is now in use in many major companies developing embedded software (avionics, transportation, energy, . . . ). [Hal05] tells the story of Lustre and Scade.

Lustre is based on the synchronous paradigm [IEE91, Hal93, BCE\(^+\)03]: the behaviour of a program is a sequence of reactions, each reaction consisting of reading the current inputs, computing the current outputs, and updating the internal state. So, a program typically implements an automaton: the states are the valuations of the memory, and each reaction corresponds to a transition of the automaton. Such a transition may involve many computations, which, from the automaton point of view, are considered atomic (i.e., input changes are only taken into account between two reactions). This is the essence of the synchronous paradigm, where such a reaction is often said to take no time. An atomic reaction is called an instant (logical time), and all the events occurring during such a reaction are considered simultaneous. The way this logical time scale is defined, i.e., the way these reactions are triggered, is left to the environment: a program can be either event-triggered or time-triggered.

Synchronous languages aim at providing high level, modular, constructs, to make the design of such an automaton easier. The basic construct that all these languages provide, is a notion of synchronous concurrency, inspired by Milner’s synchronous product [Mil81, Mil83]: in the sampling scheme, when automata are composed in parallel, a transition of the product is made of “simultaneous” transitions of all of them. When participating in such a compound transition, each automaton considers the outputs of others as being part of its own inputs. This “instantaneous” communication is called the synchronous broadcast [BCG88]. The important point is that, in contrast with the asynchronous concurrency considered in asynchronous languages like Ada, this synchronous product can preserve determinism, a highly desirable feature in reactive systems design.

Examples of other synchronous languages are Esterel [BG92, BS91], Signal [LGLL91, LTL03], ReactiveC [Bou91], SL [BdS96], Synchronous Lucid [Pou06], and ReactiveML [MP05].

\(^1\)see www.esterel-technologies.com/products/scade-suite/
Beside being synchronous, Lustre is also data-flow. The goal is to adhere to the common formalisms of control engineers, which are often data-flow synchronous formalisms, inherited from earlier analog technology: differential or finite-difference equations, block-diagrams, analog networks. Interpreted in a discrete world, these models can be formalized using the data-flow paradigm [Kah74, AW85].

2 An overview of Lustre

The initial descriptions of Lustre appeared in [CPHP87, HCRP91]. We briefly recall here the general principles of the language: A Lustre program operates on flows of values. Any variable (or expression) \( x \) represents a flow, i.e., an infinite sequence \((x_0, x_1, \ldots, x_n, \ldots)\) of values, \( x_n \) being the value of \( x \) at the \( n \)th reaction of the program. A program computes output flows from input flows. Output (and possibly local) flows are defined by means of equations (in the mathematical sense), an equation "\( x = e \)" meaning "\( \forall n, x_n = e_n \)". So, an equation can be understood as a temporal invariant. Lustre operators operate globally on flows: for instance, "\( x + y \)" is the flow \((x_0 + y_0, x_1 + y_1, \ldots, x_n + y_n, \ldots)\). In addition to usual arithmetic, Boolean, conditional operators — extended pointwise to flows as just shown — we will consider only two temporal operators:

- the operator "\( \text{pre} \)" ("previous") gives access to the previous value of its argument: "\( \text{pre}(x) \)" is the flow \((\text{nil}, x_0, \ldots, x_{n-1}, \ldots)\), where the very first value "\( \text{nil} \)" is an undefined ("non initialized") value.

- the operator "\( \rightarrow \)" ("followed by") is used to define initial values: "\( x \rightarrow y \)" is the flow \((x_0, y_1, \ldots, y_n, \ldots)\), initially equal to \( x \), and then equal to \( y \) forever.

As a very simple and classical example, the program shown below is a counter of "events": It takes as inputs two Boolean flows "\( \text{evt} \)" (true whenever the counted "event" occurs), and "\( \text{reset} \)" (true whenever the counter should be reinitialized), and returns the number of occurrences of "events" which occurred since the last "reset".

```plaintext
node Count(evt, reset: bool) returns (count: int);
let
    count = if (true \rightarrow reset) then 0
              else if evt then pre(count) + 1
              else pre(count);
tel
```

Intuitively, "\( \text{true} \rightarrow \text{reset} \)" is a Boolean flow, which is true at the initial instant and whenever "\( \text{reset} \)" is true; when it is true, the value of "\( \text{count} \)" is 0; otherwise, when "\( \text{event} \)" is true, "\( \text{count} \)" is incremented; otherwise it keeps its previous value.

Once declared, such a "node" can be used anywhere in a program, as a user-defined operator. For instance, our counter can be used to generate an event "\( \text{minute} \)" every 60
“second”, by counting “second” modulo 60:

\[
\text{mod60} = \text{Count(second, minute)}; \\
\text{minute} = \text{second and pre(mod60)=59};
\]

Here, “mod60” is the output of a “Count” node, counting “second”, and reset each “minute”, while “minute” is true whenever the “second” occurs when the previous value of “mod60” is 59.

So, through the notion of node, Lustre naturally offers hierarchical description and component reuse. Data traveling along the “wires” of an operator network can be complex, structured informations.

From a temporal point of view, industrial applications show that several processing chains, evolving at different rates, can appear in a single system. Lustre offers a notion of boolean clock, allowing the activation of nodes at different rates.

Finally, one can express some knowledge about the input of a program using assertions. These assertions are taken into account in verification (the desired property is only intended to hold when the inputs satisfy the assertion), for automatic testing (only input scenarios satisfying the assertion are generated), and sometimes for code optimization.

The graphical counterpart of Lustre textual syntax is obvious; for instance, Fig. 1 is a Scade view of the “minute detector” described before.

3 Available tools and papers

Some significant examples of Lustre programs have been published in [Hol94, CD96]. Apart from industrial tools provided by the Scade toolbox, academic tools (see www-verimag.imag.fr/SYNCHRONE/) consist of the compiler V4, the model-checker Lesar [HLR92], the automatic testing tool Lurette [RWNH98, JRB04], and translators from Simulink and Stateflow to Lustre [CCM+03]. Extensions of the language concern arrays [Mor02, MM04] and the combination with explicit automata [JLRM94, MR98]. Lucid-synchrone [CP95, CP96, Pou06] is a higher-order extension of Lustre. A lot of work has been devoted to the compilation to distributed or multi-thread code [GC92,
CMR01, CS00, SC04]. [BCDPV99] presents a methodology for proving Lustre programs with PVS [ORS92].

References


