Nested clocks: The LUSTRE synchronous dataflow language

John A. Plaice
Dept of Computer Science
University of Ottawa
150 Louis Pasteur
Ottawa, Ontario
CANADA K1N 6N5

1988
Abstract

LUSTRE is a dataflow language designed for the programming of reactive systems. After a brief introduction to reactive systems, and the notion of synchronous system, the constructs of the language are presented. LUSTRE programs manipulate clocked streams, and the notion of nested clock is the principal interest of LUSTRE. The denotational semantics of the language is in the style of Kahn’s networks.
1 Introduction

The LUSTRE dataflow programming language is a synchronous language; other synchronous formalisms include the Esterel [6] imperative language, the SIGNAL [15] dataflow programming language and the Statecharts visual formalism [13]. These formalisms were conceived for the design of reactive systems, which include real-time process controllers and human-machine interfaces. In this paper, we present LUSTRE, based on the notion of nested clocks.

1.1 Reactive programs and systems

Reactive programs are computer programs which interact continuously with their environments, normally by reacting to the stimuli sent by the environments; reactive systems are applications whose main component is a reactive program (cf. Harel and Pnueli [13]). Typical examples of reactive systems are automatic control systems such as nuclear reactors and railway switchyards, electronic gadgets such as digital watches and compact disk players, and signal processing units. Reactive systems include real-time systems, which are also subject to “hard” timing constraints, such as sampling frequency and response time.

Many reactive systems are used in situations where security is important. To increase reliability, their software should be specified and programmed using languages with a clear and formal semantics, and be well adapted to the domain; in this case, as time plays a rôle in the functioning of a program, it should be clear not only what a program does, but when it does it.

To solve the “real-time problem”, existing imperative programming languages have been extended with timing primitives, such as watchdogs and priority systems, which refer to a notion of universal time. But there are two fundamental problems with this approach. First, their semantics is unclear: if we were to set a watchdog on task A with a time limit of three seconds, and if the three seconds elapsed, could A be interrupted immediately if it were atomic? This sort of question is left unanswered in language definitions, so there is no way of guaranteeing that the timing constraints are going to be met. Second, a program can have very different behaviours, depending on the implementation architecture; this is a serious deficiency in the design of portable code.

One might imagine a system in which these problems were suitably addressed, perhaps through the use of astronomical time or a computer’s physical time. But this model may not even be appropriate, unless either of those two notions of time actually affects the behaviour of a system.

An example from Wlad Turski [21] is of cooking the perfect soft-boiled egg, with a firm white and a slightly runny yolk. Such an egg is normally referred to as a three-minute egg, as an average egg at room temperature takes about three minutes to cook after being immersed in a pot of boiling water. But, of course, if we wished our method to work for all eggs, cold or warm, large or small, fresh or stale, then we would have to devise a series of differential equations which would relate a cooking egg’s consistency to its size, shape, age and original temperature, the water’s temperature, the altitude, and the time elapsed since the egg was immersed. A much simpler solution would be to use an egg-consistency meter (with ultrasound?), and to cook until the desired consistency is reached.

It appears therefore that the current methods for programming reactive systems leave something to be desired, in terms of the reliability and of the expressability of the software.

1.2 The synchronous hypothesis

The problem can be approached from another angle. There exists a class of reactive systems which can be programmed in a much simpler manner, where it is assumed that reactions are independent, i.e., that no new input events occur until the current reaction is finished. A surprising number of reactive systems can be programmed with this assumption, and as hardware becomes faster and as compilation techniques improve, this number can only increase.

With this approach, guaranteeing that timing constraints are going to met consists solely of checking that no reaction can take longer than the minimum time interval between two input
events. If such is the case, then it doesn’t matter whether there is an extra attosecond every cycle or an extra terasecond. In fact, we can abstract ourselves completely from the implementation of these systems through the synchronous hypothesis: we pretend that they react \textit{instantaneously} to their external stimuli, or equivalently that the computers they run on are \textit{infinitely fast}.

1.3 Models of time

Unlike in other scientific disciplines, it is only recently that time has been studied in computer science. In fact, one might argue that time has been \textit{ignored}, as in the asynchronous model underlying languages such as Ada.

Real-time processes are subject to timing constraints defined in terms of universal time; however, a process only “sees” local events, such as emissions and receptions of messages with other processes. If a process $P$ must respond to an input event $I$ with an output event $O$ within 3 seconds, the events $I$ and $O$ are local to $P$, but the 3 seconds are global. The constraints are therefore global statements about local events. This dichotomy is reminiscent of Russell’s notions of \textit{private} and \textit{public} times. No model of time for timed systems would be complete without allowing for the different levels of time.

The synchronous process algebrae of Milner [17] and Austry and Boudol [1] were devised to better understand the notion of communication between processes, and in particular the notion of synchronization. It is clear that synchronization plays a key role in timed systems, for without it one can not devise any kind of reaction to an input event. However, there is no global clock, so it would be difficult to express timing constraints.

The interval temporal logics of Moszkowski [18] and Schwartz, Melliar-Smith and Vogt [20] subsume a global time scale, and actions are considered to take a closed time interval to execute, with an explicit beginning and end. Here, expressing timing constraints is easy, but it is difficult to differentiate between the different levels of time.

The model of events of Caspi and Halbwachs [8] can be considered to have the advantages of the two previous sets of formalisms. There does exist a global time frame, but it is the actual events that define “time”. A system is defined in terms of a set of events, such as emissions and receptions of signals or assignments to variables; the system’s behaviour is completely determined by the sequences of dated occurrences of its events. With this model, any timed behaviour can be described, even if time is continuous.

In this model, the dates in the sequence of occurrences of an event $e$ define a \textit{clock} $c$: the $n$-th occurrence of $e$ coincides with the $n$-th date defined by $c$. Time becomes \textit{multi-form}, i.e., can be defined according to any scale; the physical “clock” of the computer is just one more event, with its own clock.

1.4 Lustre

If we wish to devise a programming language based on this notion, it is natural to use the dataflow model: this is what is done in \textsc{Lustre} [9]. The dated sequences become \textit{clocked streams}, the desired interpretation being that at the $n$-th instant defined by its clock, a stream takes its $n$-th value. Each clock is itself a clocked stream or is the global clock, which is the finest clock; it can be considered to be the union of all the other clocks; one tick of the global clock defines an “instant”.

We also use the synchronous hypothesis, so data operations can be applied term by term on operands of the same clock, yielding a result of the same clock. There are operations to change a clock of one stream to a clock of another stream.

\textsc{Lustre} belongs to the family of synchronous languages, along with \textsc{Esterel} [6] and \textsc{Signal} [2]. Arguments for a similar programming style have also been made by Turski [21].

\textsc{Lustre} has been used to program automatic control examples [5], systolic algorithms [12] and hardware circuits [11]. The rationale behind the design of the language was presented in Bergerand’s dissertation [3]. The semantics and the prototype compiler were presented in the author’s dissertation [19].
This paper discusses the Lustre language, with particular focus on the use of clocks. It presents the language and its basic operators.

2 The Lustre language

To present Lustre, we will use the following problem: suppose that two 12-hour wall clocks are to be installed in a train station; the first will show the time using hands, and the second will be digital. The two wall clocks will be controlled by a single program WATCH, which receives two inputs, a signal s from a quartz occurring once a second, and a reset signal which states that the wall clocks must be reset to 12 o’clock. Clearly, a set of clocks defined in this manner would be totally unsatisfactory, because the clocks could only be set twice a day. However, they are sufficiently complex to present Lustre.

The clock with hands is receptive to four signals: start puts all the hands at the twelve o’clock position, and second, minute and hour each move the appropriate hand by one position. The digital clock has a display which must be replaced ten times a second; it is receptive to the signal time.

2.1 Clocked streams and nodes

The basic entity in Lustre is a clocked stream, consisting of a sequence of values and of a clock, which defines the sequence of instants at which those values appear. If the sequence of values is 
\[(e_1, e_2, \ldots, e_n, \ldots)\],

then the stream takes the value \(e_n\) at the \(n\)-th instant of its clock.

A clock is either the base clock of the program (intuitively, the sequence of instants at which it is active), or a boolean stream. A boolean stream \(b\), considered as a clock, defines the sequence of instants where its value is true, as is shown in the example of Table 1.

Clocks allow a system to have several component subsystems evolving at different rates. In fact, in a single system, the same subsystem may be used several times, each time at a different rate. In our example, we will find that the same counter is used to count seconds, minutes, and hours, each at a different rate.

With clocks, a variable does not have to be calculated when it is not required. This possibility can be used for generating more efficient code, but it is far more useful in expressing relationships which must hold between variables. In our example, the hour hand may not move if the minute hand does not also move.

When a number of clocks are used, time becomes multiform. Any boolean signal to a program can be used as a clock; the base clock becomes the union of all the clocks, and can therefore be totally independent of any notion of physical time. In our example, the s and reset input signals both define clocks; the base clock is their union.

Furthermore, clocks can be defined in a nested manner; in Table 1, the clock of \(c\) is \(b\), and its clock is the base clock. For our problem, the hour output, which itself is a clock, is a subclock of the minute output, whose clock is the base clock.

\[1\] Do not confuse Lustre clocks with the wall clocks!
Finally, one of the biggest problems with implementing dataflow languages has been trying to avoid situations where an unbounded amount of memory is required. The clock calculus of Lustre will ensure that this sort of thing can never occur in a correct Lustre program.

Clocked streams are manipulated by nodes, effectively functions between streams. A node can be predefined or user-defined, in the latter case with a set of equations. An example node, which outputs the stream which it receives as input, is:

\[
\text{node IDENTITY (x:* returns (y:*)}
\]
\[
\text{let y = x tel}
\]

2.2 Variables and Equations

Nodes are defined through a set of equations. A variable is defined by an equation, and its value is a clocked stream. If \(X\) is a variable and \(E\) is an expression, \(X=E\) equates \(X\) and \(E\). Whatever the input to the node being defined, the clock and the sequence of values of \(X\) and \(E\) will be identical.

If, in a particular node instantiation, \(E\)’s clock is \(h\) and its sequence is \((e_1,e_2,\ldots,e_n,\ldots)\), then \(X\)’s clock is \(h\) and its sequence is \((x_1 = e_1, x_2 = e_2, \ldots, x_n = e_n, \ldots)\).

An expression \(E\) is deemed to have a clock, which is itself an expression, either \texttt{base} or a boolean expression \(B\). If the clock is \texttt{base}, then, in an instantiation, the clock of the stream associated with \(E\) will be the clock of the actual input parameters of the instantiation. Otherwise, the clock of the stream will be the stream associated with \(B\).

Actually, the semantics of the language is for expressions of the form \(E \texttt{ on } B\), where \(B\) is \(E\)’s clock. Normally, a programmer will not use the \texttt{on} form, as the compiler can infer the clock.

To ensure referential transparency, we insist upon two notions:

- **Substitution principle** An equation \(X=E\) specifies the variable \(X\) and the expression \(E\) to be completely synonymous. In any context, \(X\) may be replaced by \(E\), and vice versa.

- **Definition principle** The context in which an expression \(E\) is used can have no influence on the behaviour of that expression. In particular, no information may be inferred about the input.

Streams are possibly infinite sequences of values, but this by no means implies that one has to imagine infinite sequences to write a Lustre program. More typical is to determine what actually happens at a particular instant, i.e., to simply consider a variable as an object which takes on a value each time that its clock is true.

Each variable which is not an input is defined by exactly one equation. The order of a set of equations is no significance. Expressions are constructed with variables, constants, data operators and temporal operators. The latter, which effectively manipulate sequences, are either synchronous or clock-changing.

2.3 Constants

A constant appearing in an expression means the infinite constant sequence on the appropriate clock. For example, \(1 \texttt{ on } B\) defines a constant sequence of 1’s, occurring when \(B\) is true. If the clock is not specified, then the compiler assumes that the clock is the base clock.

2.4 Data Operators

Data operators are the usual operators (arithmetic, boolean, conditional); they are supposed to operate term by term on the sequences.

The simplest manner to avoid the need for an unbounded memory is to ensure that the operands of every operator have exactly the same notion of time, hence the same clock.

For example, if \(X\) and \(Y\) are two variables on the same clock \(H\), the expression
if \( X > Y \) then \( X - Y \) else \( Y - X \)
denotes the sequence whose \( n \)-th term, i.e., the term at the \( n \)-th instant of the clock \( H \), is the absolute value of the difference of the \( n \)-th terms of \( X \) and \( Y \).

### 2.5 Synchronous Operators

The synchronous operators directly manipulate sequences, but without introducing new clocks.

The \( \text{pre} \) ("previous") operator returns the value of its argument at the previous instant. If

\[
X = (x_1, x_2, \ldots, x_n, \ldots),
\]

then

\[
\text{pre} \ X = (\text{nil}, x_1, x_2, \ldots, x_{n-1}, \ldots),
\]

where \text{nil} is an \textit{undefined} value, similar to the value of an uninitialized variable in imperative languages. Apart from the conditional operator, all data operators are strict with respect to \text{nil}, i.e., return the value \text{nil} each time that one of their operands is the value \text{nil}. For the conditional, if the condition is false, the \text{then} branch is not considered; if the condition is true, the \text{else} branch is not considered. For example,

\[
X = \text{if true then } 1 \text{ else pre } X
\]

defines \( X \) as the constant sequence

\[
(1,1,\ldots,1,\ldots)
\]
on the base clock.

The \( \rightarrow \) operator is used to initialize variables. If

\[
X = (x_1, x_2, \ldots, x_n, \ldots)
\]

and

\[
Y = (y_1, y_2, \ldots, y_n, \ldots)
\]

are two expressions of the same type and of the same clock, then

\[
X \rightarrow Y = (x_1, y_2, y_3, \ldots, y_n, \ldots),
\]

so \( X \rightarrow Y \) is always equal to \( Y \) except at the initial instant.

The example

\[
X = 0 \rightarrow \text{pre } X + 1
\]
specifies that \( X \) is initially 0 and is incremented by 1 in subsequent instants. \( X \) is therefore the sequence of the natural numbers on the base clock. It is a counter of instants.

A second example is a first order linear filter. The filter

\[
\begin{align*}
y_1 &= \text{init} \\
y_{n+1} &= ay_n + bx_{n+1}, & n \geq 1
\end{align*}
\]
is written in \textsc{Lustre} as:

\[
Y = \text{init} \rightarrow a \ast \text{pre } Y + b \ast X
\]

5
\[ E = (e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad \ldots) \]
\[ C = (tt \quad ff \quad tt \quad tt \quad ff \quad \ldots) \]

\[ X = E \text{ when } C = (x_1 = e_1 \quad x_2 = e_3 \quad x_3 = e_4 \quad \ldots) \]

Table 2: Oversampling operator

\[ E = (e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad \ldots) \]
\[ C = (tt \quad ff \quad tt \quad tt \quad ff \quad \ldots) \]

\[ X = E \text{ when } C = (x_1 = e_1 \quad x_2 = e_3 \quad x_3 = e_4 \quad \ldots) \]

\[ Y = 1 \text{ when } C \rightarrow X = (1 \quad e_3 \quad e_4 \quad \ldots) \]

\[ Z = \text{current } Y = (1 \quad 1 \quad e_3 \quad e_4 \quad \ldots) \]

Table 3: Sampling and Projection

### 2.6 Clock-changing Operators

To change clocks, one can sample a stream to form a stream whose clock is coarser grained, or project a stream onto a clock of finer grain.

If \( E \) is an expression and \( C \) is a boolean expression, and if their clocks are the same, \( E \text{ when } C \) is an expression whose data sequence is extracted from the sequence of \( E \) by only choosing the values taken by \( E \) when \( C \) is true.

The expression \( E \text{ when } C \) does not “have the same notion of time” that \( E \) and \( C \) do, as is shown by the example in Table 2. In this example, we say that \( X \) is calculated from the clock \( C \) and that \( C \) is a sampling clock. The only notion of time “known” by \( X \) is the sequence of instants where \( C \) is true. So, the question, “What is the value of \( X \) when \( C \) is false?”, makes no more sense than does the question, “What is the value of a variable between two whole instants?” More complex examples of the use of \( \text{when} \) will be given in §2.8.

Suppose that one wishes to apply an operator to expressions having different clocks, e.g., adding \( X \) and \( E \) in the example in Table 2. Given that the two expressions evolve at different rates, more and more memory is necessary to save the old values of \( E \). To avoid this problem, we insist that the arguments be put on the same clock, either by sampling \( E \), or by projecting \( X \), using the \( \text{current} \) operator.

If \( E \) is an expression on clock \( C \) and if \( C \) is a sampling clock, \( \text{current } E \) is an expression whose clock is that of \( C \) and whose value at each instant is that taken by \( E \) the last time that \( C \) was true.

Table 3 illustrates the combined effect of the \( \text{when} \) and \( \text{current} \) operators.

The clocks of a node form a tree: the root is \( \text{base} \); if \( C \) is a clock in the tree, its children in the tree are its subclocks. \( \text{when} \) and \( \text{current} \) allow one to move up and down the tree. However, it is sometimes necessary to move up or down several branches of the tree. To do this, we introduce generalized sampling and projection operators which are defined in terms of \( \text{when} \) and \( \text{current} \).

Suppose we wish to sample an expression \( E \) according to a clock \( C \), where the clock of \( C \) is \( B \), and the clocks of \( B \) and \( E \) are the same. We could write \( (E \text{ when } B) \text{ when } C \), but that would be cumbersome. If we had many levels to go up, then the resulting expressions would be truly unwieldy. To avoid this situation, we write \( \text{filter}(E, C) \), which is ultimately translated into the appropriate cascade of \( \text{when} \)’s.

There is a similar analogue with \( \text{current} \): \( \text{project}(E, C) \), projects \( E \) until its clock is \( C \). Our wall clock example will use \( \text{filter} \) and \( \text{project} \).

Finally, the clock of an expression can be accessed with the \( \text{ck} \) operator: \( \text{ck}(E) \) yields the clock of expression \( E \).

### 2.7 Nodes and networks

Any \textsc{Lustre} program can be considered to be a network of operators connected by wires. For example, the equation
A node is a Lustre subprogram. It receives streams as input, and returns streams as output, possibly calculating local streams, by means of a system of equations. For example, a generalized counter can be defined by:

```
node COUNT (init:int, incr:int, reset:bool)
  returns (n:int)
  let
    n = init -> if reset then init else pre n + incr
  tel
```

Node instantiation is done in a functional manner: if $N$ is the name of a node, declared with the header

```
node N (i_1 : \tau_1; \ldots; i_p : \tau_p) returns (i_{p+1} : \tau_{p+1}; \ldots; i_q : \tau_q);
```

and if $E_1, \ldots, E_p$ are expressions of type $\tau_1, \ldots, \tau_p$, then the instantiation $N(E_1, \ldots, E_p)$ is an expression of type $(\tau_{p+1}, \ldots, \tau_q)$. To come back to the example of the counter, one can write

```
pair = COUNT(0, 2, false)
mod5 = COUNT(0, 1, pre(mod5=4))
```

which defines `even` to be the sequence of even numbers, and `mod5` to be the cyclic sequence of naturals modulo 5.

### 2.8 Nodes and clocks

The base clock of a node is determined by the clock of its input parameters. For example, the call

```
COUNT(0 when C, 1 when C, false when C)
```

This is not quite true. The network in Figure 1 gives the impression that constants are stream generators. A constant is really a function which takes as input the base clock.

---

**Figure 1: Operator network**

![Operator network diagram](image)

**Table 4: Sampling of input, sampling of output**

<table>
<thead>
<tr>
<th>C = (</th>
<th>tt</th>
<th>ff</th>
<th>tt</th>
<th>ff</th>
<th>tt</th>
<th>... )</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(0 when C, 1 when C, false when C) = (</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>... )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT(0,1,false) when C = (</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>... )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
counts each time that \( C \) is true. Now, \texttt{when} does not commute with the temporal operators, and it is in general different to filter a node’s input than to filter its output (cf. Table 4).

A node can take as input streams of different clocks, but when a parameter’s clock is not the basic clock, that clock must itself be passed as a parameter. To do this, the \texttt{on} form of the expression must be given. For example, the header of a node could be

\[
\text{node } N \ (\text{millisecond:bool on base; second:bool on millisecond}) \\
\quad \text{returns} \ldots
\]

which states that \texttt{millisecond} is on the base clock and \texttt{second}'s clock is \texttt{millisecond}, i.e., \texttt{second} only exists when \texttt{millisecond} is true.

A node can also return output on different clocks, assuming that the clocks are all visible from outside the node. For example,

\[
\text{node } \text{ONE_OUT_OF_TWO} \ (x:\*) \text{ returns } \text{clock:bool; y:}* \\
\quad \text{let} \\
\quad \quad \text{clock} = \text{true} \rightarrow \text{not pre clock} \\
\quad \quad y = x \text{ when clock} \\
\quad \text{tel}
\]

removes every second input.

2.9 The \texttt{WATCH} example

To program our wall clocks, we will use a counter which increments by one:

\[
\text{node } \text{COUNT1} \ (\text{init:int; reset:bool}) \text{ returns } \text{count:int} \\
\quad \text{let} \ 	ext{count} = \text{COUNT}(\text{init}, 1, \text{reset}) \text{ tel}
\]

as well a node which detects a rising boolean value:

\[
\text{node } \text{EDGE} \ (b:bool) \text{ returns } \text{edge:bool} \\
\quad \text{let} \ 	ext{edge} = b \rightarrow b \text{ and not pre } b \text{ tel}
\]

The \texttt{WATCH} node is therefore:

\[
\text{node } \text{WATCH} \ (s:bool; \text{reset:bool}) \\
\quad \text{returns } \text{start:bool; min:bool; hour:bool on min; time:[int,int]} \\
\quad \text{var} \ 	ext{no_s:int on s; no_min:int on min; no_hour:int on hour} \\
\quad \text{assert } s \rightarrow \text{true}, #(s, \text{reset}) \\
\quad \text{let} \\
\quad \quad \text{no_s} = \text{COUNT1}(0 \text{ on s, reset or pre no_s = 59 on s}) \\
\quad \quad \text{min} = \text{no_s} = 0 \text{ on s} \\
\quad \quad \text{no_min} = \text{COUNT1}(0 \text{ on min, filter(reset, min) or pre no_min = 59 on min}) \\
\quad \quad \text{hour} = \text{no_min} = 0 \text{ on min} \\
\quad \quad \text{no_hour} = \text{COUNT1}(12 \text{ on hour, filter(reset, hour) or pre no_hour = 12 on hour}) \\
\quad \quad \text{time} = [\text{project(no_min, base), project(no_hour, base)}] \\
\quad \quad \text{start} = \text{project(EDGE(no_hour = 12 on hour), base)} \\
\quad \text{tel}
\]

2.10 Tuples

When nodes have more than input or output stream, they are manipulating tuple streams, i.e., a tuple of streams. If \( E_1 \) and \( E_2 \) are expressions of types \( \tau_1 \) and \( \tau_2 \), respectively, then \( (E_1, E_2) \) is a tuple of type \( (\tau_1, \tau_2) \); for example, \( (1, \text{true}) \) is of type \( \text{(int, bool)} \). Tupling implies the
juxtaposition of two streams, so we can consider that they are flattened: there is no difference between \((1, (2, 3))\), \(((1, 2), 3)\), and \((1, 2, 3)\).

The clock of a tuple is the tuple of the clocks of its elements. Since tupling just implies the juxtaposition of streams, a tuple can have elements whose clocks are different. There is therefore a significant difference between a record stream, which is a stream whose elements are all records, hence only one clock, and a tuple stream, which is a list of streams.

There will be times when a tuple is returned by a node and that only one of the elements is of interest. To provide for such situations, there is the select operator: \texttt{select}(i,E) selects the \(i\)-th stream in the tuple stream \(E\).

3 The denotational semantics

The first version of the denotational semantics was given in the original article presenting Lustre [4] and was refined in the thesis of J.-L. Bergerand [3]. Based on Kahn’s model [14], it considers a stream to be a sequence; clocks are only static notions. But a clock is a dynamic entity, and the semantics should reflect that fact. Furthermore, it would be simpler to give a semantics which considers that all operators are nodes.

In Kahn’s semantics, a stream of type \(\tau\) is considered to be a form of pipeline which carries values of type \(\tau\). If an observer is put on that line, it would see a possibly infinite sequence of objects of type \(\tau\). In Lustre, the notion of stream is more complicated: a stream is a pair consisting of a sequence and a clock.

The semantics of a set of equations is given by supposing an environment of variables which associates a stream with each variable. The semantics of a node declaration is then the least fixed point of the semantics of its system of equations. As for Kahn, and unlike for Lucid, the order on sequences is prefix.

In the introduction, we presented the causality and bounded memory properties; formally, causality corresponds to the monotony of the semantics. We prove that all Lustre programs have this property.

The semantics given here is simpler than that given in the author’s dissertation [19] as we have finally understood the difference between record streams and tuple streams.

3.1 The abstract syntax

3.1.1 Expressions

We do not give the semantics for ordinary expressions, but rather for clocked expressions \(ce\):

\[
ce ::= e \text{ on } h
\]

which means the expression \(e\) on clock \(h\).

The clock of an expression is either the base clock of the node, base, a clocked expression, or a list of clocks:

\[
h ::= \text{base} | ce | (h,h)
\]

The domain of identifiers is \(Id \ (\ni \ id, \id_1, \id_2, \ldots)\). \(k\) denotes a constant.

The domain of clocked expressions is \(Exp \ (\ni \ ce, \ce_1, \ce_2, \ldots)\).

\[
\begin{align*}
ce & ::= e \text{ on } h \\
e & ::= k | id | (ce, ce) | d ce
\end{align*}
\]

The last case involves a node instantiation. It includes all of the basic nodes of Lustre, i.e., \(\text{pre}\), \(\Rightarrow\), \(\text{when}\), \(\text{current}\), \(\text{select}\) and \(\text{ck}\), and all of the data operators, predefined or not. We note a clocked identifier by \(cid\).
3.1.2 Equations
The domain of equations is \( \text{Eq} (\ni eq) \). Through the use of the \texttt{select} operator, we can consider that each equation assigns to one variable.

\[
eq : = cid = ce \mid eq_1 eq_2
\]

3.1.3 Node declarations
The domain of node declarations is \( \text{Node} (\ni d) \). A node consists of its name, a list of clocked variables and an equation:

\[
d : = id p q r cid_1, \ldots, cid_r eq
\]
where \( p \) is the number of input parameters, \( q - p \) the number of output parameters and \( r - q \) the number of local variables.

Depending on the concept which is being presented, some information can be left out. For example, in the section on clocks, we will not use the type.

3.1.4 Node signatures
The abstract syntax of a node should allow abstraction from the node, i.e., definition of the information which is necessary for the use of a node, yet without knowing how the node was defined. We call this information the \texttt{signature} of the node.

The signature includes the information of the node, but without referring to the local variables or to the equation:

\[
s : = id p q cid_1, \ldots, cid_q
\]

3.1.5 The syntax of a program
Since recursive instantiations are not allowed in \texttt{Lustre}, we can suppose that a \texttt{Lustre} program is a sequence of node declarations, arranged in an order such that if a node \( B \) calls a node \( A \), the \( A \) precedes \( B \). This topolgical sort is done by the type checker.

3.2 Semantic Domains
3.2.1 Sequences
\( \tau \) will denote a type or the set of values associated with it. \( \tau^\infty \) denotes the finite and infinite sequences of \( \tau \). The \textit{domain of sequences} of type \( \tau \) is the set

\[
S_{\tau} = (\tau \cup \{\text{nil}\})^\infty \ni \nu
\]
along with the prefix order. The length of sequence \( \nu \) is denoted by \( \lg(\nu) \).

3.2.2 The global base clock
In the previous section, we said that a stream is a pair consisting of a sequence of values and a clock; this clock can either be a boolean stream, or the base clock of the node. If the node is called by another node, then the base clock will be instantiated with another stream; on the other hand, if it is the main node, the base clock must be instantiated with the \texttt{global base clock}, which is supplied by the environment. To simplify the semantics which follows, we consider that the global base clock is a boolean sequence whose values are always \( tt \).
3.2.3 Simple streams

A simple stream of type $\tau$ is a pair

$$\phi = (\upsilon, \chi),$$

where $\upsilon$ is a sequence of type $\tau$ and where $\chi$ is a clock. A clock is either the global base clock, or a simple boolean stream.

The domain of simple streams of type $\tau$ is therefore defined by:

$$FS_\tau = S_\tau \times H$$

$$H = S_{bool} + FS_{bool}$$

The function $\Upsilon$ selects the sequence of a stream and $X$ is used for the clock:

$$\Upsilon(\upsilon, \chi) = \upsilon$$

$$X(\upsilon, \chi) = \chi$$

If $\upsilon$ should be the global base clock, we define

$$\Upsilon(\upsilon) = \upsilon.$$  

Let $\chi$ be a clock. For each $n \geq 0$, $\text{count}(\chi)n$ is the number of true terms in the sequence of values of $\chi$ whose rank is less than or equal to $n$.

$$\text{count} = \lambda \chi \cdot \lambda n \cdot \text{card}\{m \leq n \mid \Upsilon \chi m = \text{tt}\}$$

For each $n \geq 0$, $\text{rank}(\chi)n$ is the rank of the $n$-th true term is the sequence of values of $\chi$, but only if there are no $\text{nil}$ in the sequence (it is strict).

$$\text{rank} = \lambda \chi \cdot \lambda n \cdot \inf\{m \mid \text{count}(\chi)m \geq n\} \circ \lambda \chi \cdot \lambda n \cdot \text{si } \Upsilon \chi n \neq \text{nil} \text{ alors } \Upsilon \chi n$$

The length of a simple stream is not necessarily the length of its sequence. Consider the stream $$(\lambda n \cdot k, <, \text{tt}, \text{tt}>).$$ The sequence is infinite, but since the length of the clock is only 2, only the first two values of the sequence should be considered. The length of a stream $(\upsilon, \chi)$ is therefore:

$$L(\upsilon, \chi) = \min(\text{lg}(\upsilon), \text{count}(\chi)(L\chi))$$

$$L(\upsilon) = \text{lg}(\upsilon)$$

The preorder on simple streams is defined by

$$(\upsilon, \chi) \preceq (\upsilon', \chi') \iff \begin{align*}
\chi &\preceq \chi' \\
L(v, \chi) &\leq L(v', \chi') \\
\left(\forall i \in [1..L(v, \chi)]\right) v(i) &= v'(i)
\end{align*}$$

The smallest element of $FS_\tau$ is $<(>, <>)$. From now on, we will denote the induced order by $\preceq$. The induced equality is denoted by $=.$ $\inf\{\phi, \phi'\}$ denotes the greatest lower bound of $\phi$ and of $\phi'$.

3.2.4 Synchronous streams

A simple stream $(v, \chi)$ is said to be synchronous if:

$$\text{lg}(v) = \text{count}(\chi)(L\chi),$$

i.e., if the sequence has exactly the same number of elements that its clock would allow one to suppose.

Two simple streams $(v, \chi)$ and $(v', \chi')$ are mutually synchronous is each is synchronous and if their clocks are identical, i.e., $\chi = \chi'$. 

11
3.2.5 Tuple streams

A tuple stream is a tuple of streams: if $\phi_1$ and $\phi_2$ are streams, then $\phi_1 \cdot \phi_2$ is a tuple stream.

The order of tuple streams is the order induced by the order on simple streams. The domain $F$ of all streams is therefore:

$$F = \sum_{\tau} F_{\tau} + \sum_{\tau} F_{\tau} \times F$$

One must not confuse a simple stream whose elements are records with a tuple stream. For example,

$$(\lambda n \cdot (k_1, k_2), <tt, tt>)$$

is not the same thing as

$$(\lambda n \cdot k_1, <tt, tt>) \cdot (\lambda n \cdot k_2, <tt, tt>).$$

We now define the bullet operation; here, tuple streams are noted $L\phi$. In the rest of the paper, they will be noted as $\phi$:

$$\begin{align*}
\phi_1 \cdot \phi_2 &= \text{list}(\phi_1, \phi_2) \\
\phi_1 \cdot L\phi_2 &= \text{cons}(\phi_1, L\phi_2) \\
L\phi_1 \cdot \phi_2 &= \text{append}(L\phi_1, \phi_2) \\
L\phi_1 \cdot L\phi_2 &= \text{append}(L\phi_1, L\phi_2)
\end{align*}$$

We also define the projection operator $\pi_i$: $\pi_i(\phi)$ returns the $i$-th stream in the tuple stream $\phi$.

3.2.6 Derived domains

The domain of environments of variables is denoted by $\text{VEnv}$ ($\zeta, \zeta' \in \text{VEnv}$) which associates a stream with each variable:

$$\text{VEnv} = \text{Id} \rightarrow F$$

The semantics of a node declaration is a function which computes an output stream from an input stream:

$$\text{Node} = F \rightarrow F$$

The semantics of an expression is a function between streams, given a variable environment:

$$\text{Exp} = \text{VEnv} \rightarrow F$$

An equation extends a variable environment:

$$\text{Equ} = \text{VEnv} \rightarrow \text{VEnv}$$

3.3 Semantic functions

3.3.1 Expressions ($\text{Exp} \rightarrow \text{Exp}$)

$$\begin{align*}
[\text{base}]\zeta &= \zeta(\text{base}) \\
[k \text{ on } h]\zeta &= (\lambda n \cdot k, [h]\zeta)
\end{align*}$$
To check the coherence of the clock of a variable, the actual clock of the variable must be compared to the clock it should have according to the formal parameter:

\[
[id \text{ on } h] \zeta = \left( \Upsilon(\zeta(id)), \inf \left\{ X(\zeta(id)), [h]\zeta \right\} \right)
\]

The semantics of a tuple is the tuple of the semantics of its elements:

\[
[(ce_1, ce_2) \text{ on } h] \zeta = [ce_1]\zeta \bullet [ce_2]\zeta
\]

\[
[d \text{ ce on } h] \zeta = [d][\text{ce}]\zeta
\]

### 3.3.2 Equations (Equ → Equ)

An equation extends the environment; as for variables appearing in expressions, the coherence of the clocks must be checked:

\[
[id \text{ on } h = ce] \zeta = \zeta \left( \left( \Upsilon([Te]\zeta), \inf \left\{ X([Te]\zeta), [h]\zeta \right\} \right) / id \right)
\]

\[
[eq_1 eq_2] \zeta = [eq_1][\text{eq}_2]\zeta
\]

### 3.3.3 Node declarations (Node → NEnv → NEnv)

We now give the semantics for a node declaration:

\[
d = id p q r \mid [id_i : \tau_i \text{ on } h_i \text{ depends } lid_i]_{i=1..p} \text{ eq}
\]

A node is a function between streams which is defined as the least fixed point of the semantics of its system of equations and of its initial environment.

If the input stream is \(\phi\), the initial environment is defined by

\[
\zeta_0 = \inf \left\{ \pi_i \phi \mid i \in [1..p] \land h_i = \text{base} \right\} / \text{base}, \pi_1 / id_1, \ldots, \pi_p / id_p, [\langle\rangle, \langle\rangle] / id_j]_{j=p+1..q}
\]

The semantics of \(d\) calculates the smallest environment \(\zeta\) such that

\[
\zeta_0 \preceq \zeta \land \zeta = [eq]\zeta
\]

Hence,

\[
[d] = \lambda \phi \cdot \left[ \mu \zeta \cdot \zeta_0([eq]\zeta) \right]_{i=p+1..q}
\]

where

\[
[\zeta]_{i=j..k} = \zeta(id_j) \bullet \cdots \bullet \zeta(id_k).
\]

and where \(\mu\) is the least fixed point operator.

### 3.4 Semantics of the basic operators

Here we define \([d]\) for each basic node. Except for the selection operator, these nodes all return simple streams; we can consider their semantics as pairs of functions \((\Upsilon, X)\) such that if \(op\) is an operator of arity \(p\), then

\[
N(op) = \lambda \phi \cdot (\Upsilon(op)(\pi_1 \phi, \ldots, \pi_p \phi), X(op)(\pi_1 \phi, \ldots, \pi_p \phi))
\]

For the selection operator,

\[
[\text{select}(i, \phi)] = \pi_i \phi
\]
3.4.1 Strict data operators

We give the semantics for an operator \( op \) of arity \( p \): If \( op \) is the associated mathematical operator, \( op \) represents the application, in a term by term manner, of the natural extensions of \( op \) with respect to \( \text{nil} \):

\[
\Upsilon(op) = \lambda \phi_1, \ldots, \phi_p \cdot \lambda n \cdot \text{si} \ (\forall i \in [1..m]) \Upsilon \phi_i n \neq \text{nil} \\
\text{alors} \ op(\Upsilon \phi_1 n, \ldots, \Upsilon \phi_p n) \\
\text{sinon} \ \text{nil}
\]

\[
X(op) = \lambda \phi_1, \ldots, \phi_p \cdot \lambda n \cdot \inf \{X \phi_1, \ldots, X \phi_p\}
\]

3.4.2 The conditional operator

The conditional is not a strict operator: if the condition is true, the \text{else} branch is ignored; if the condition is false, the \text{then} branch is ignored.

\[
\Upsilon(\text{if}) = \lambda \phi_1, \phi_2, \phi_3 \cdot \lambda n \cdot \text{si} \ \Upsilon \phi_1 n = \text{nil} \text{ alors} \ \text{nil} \\
\text{sinon si} \ \Upsilon \phi_1 n \text{ alors} \ \Upsilon \phi_2 n \\
\text{sinon} \ \Upsilon \phi_3 n
\]

\[
X(\text{if}) = \lambda \phi_1, \phi_2, \phi_3 \cdot \inf \{X \phi_1, X \phi_2, X \phi_3\}
\]

3.4.3 Synchronous operators

\[
\Upsilon(\text{pre}) = \lambda \phi \cdot \lambda n \cdot \text{si} n = 1 \text{ alors} \ \text{nil} \text{ sinon} \ \Upsilon \phi (n - 1)
\]

\[
X(\text{pre}) = \lambda \phi \cdot X \phi
\]

\[
\Upsilon(\rightarrow) = \lambda \phi_1, \phi_2 \cdot \lambda n \cdot \text{si} n = 1 \text{ alors} \ \Upsilon \phi_1 n \text{ sinon} \ \Upsilon \phi_2 n
\]

\[
X(\rightarrow) = \lambda \phi_1, \phi_2 \cdot \inf \{X \phi_1, X \phi_2\}
\]

3.4.4 Clock-changing operators

The semantics for \( \text{ck} \) is trivial:

\[
\Upsilon(\text{ck}) = \lambda \phi \cdot X \phi
\]

\[
X(\text{ck}) = \lambda \phi \cdot X X \phi
\]

The semantics of the \( \text{when} \) and \( \text{current} \) operators uses the functions \( \text{count} \) and \( \text{rank} \):

\[
\Upsilon(\text{when}) = \lambda \phi_1, \phi_2 \cdot \Upsilon \phi_1 \circ \text{rank}(\phi_2)
\]

\[
X(\text{when}) = \lambda \phi_1, \phi_2 \cdot (\Upsilon \phi_2, \inf \{X \phi_1, X \phi_2\})
\]

\[
\Upsilon(\text{current}) = \lambda \phi \cdot \lambda n \cdot \text{si} \ \text{count}(X \phi)n = 0 \text{ alors} \ \text{nil} \\
\text{sinon} \ \Upsilon \phi(\text{count}(X \phi)n)
\]

\[
X(\text{current}) = \lambda \phi \cdot X(X \phi)
\]

3.5 Causality

A node \( d \) is causal if at each instant \( n \), the future of the input stream has no effect on the past or the present of the output stream. In other terms, if

\[
[d] \phi_0 = \phi_0',
\]

and if one adds terms to the end of \( \phi_0 \) to create a stream \( \phi_1 \), then \( \phi_0' \) is not put into question, i.e.,

\[
\phi_0' \preceq [d] \phi_1.
\]

Hence, causality is identical to monotony. This result is due to Paul Caspi [7].
Proposition 1 Every Lustre node is causal.

The semantics of a system of equations is defined from an environment variable, constant functions, the functions $\text{inf}$, $\text{count}$ and $\text{rank}$ and of the conditional function. Each of these function is monotone; since any functional defined from the composition of monotone functions and of a functional variable $F$ is continuous (cf. Manna [16]), the semantics of this system is a continuous (and hence monotone) functional. Since the semantics is continuous, according to Kleene, there must exist a least fixed point, which is itself continuous. Hence, any Lustre node is causal.

3.6 Bounded memory

A node evaluates itself with bounded memory if there exists a finite bound such that at each instant, the number of memory slots of fixed size necessary for the calculation of the present and future output remains smaller than this bound.

In an operator network, there are three ways that the memory required to do calculations is unbounded:

- the basic operators themselves require an unbounded memory;
- an unbounded number of basic operators can be created in a dynamic manner;
- values can accumulate in wires connecting the operators.

In Lustre, there is nothing that can be done about the first point except to hope that the programmer will not use operators requiring unbounded memory. The second case is not allowed as a node can not be instantiated in a recursive manner. Finally, the clock checking done at the instantiation of a node and while using variables inside a system of equations ensures that the basic operators do not do any calculations if the operands are on different clocks. This implies that if calculations are done in a synchronous manner, the operands of every basic node evolve at exactly the same rate, and that no past values need be memorized to handle streams which evolve at different rates.

If the arguments are synchronous, the only past values that must be memorised are the arguments of the memory operators, whose number is finite, as recursive instantiations are not allowed. To put a bound on the number of memory slots needed, it suffices to count the number of $\text{pre}$ and $\text{current}$ which appear in the node, counting as well the memory operators which appear in the nodes which that node calls.

If $d$ is a node, $\#\text{pre}_d$ is the number of times that the operator $\text{pre}$ appears in the equations of $d$. Similarly for $\#\text{current}_d$. The function $M$ computes an upper bound:

$$M(d) = \#\text{pre}_d + \#\text{current}_d + \sum_{d_i \text{ appears } n_i \text{ times}} n_i M(d_i)$$

We therefore have the proposition:

Proposition 2 Consider a node $d$ and a stream $\phi_1 \cdot \cdots \cdot \phi_p$. If each $\phi_i$ is synchronous, and if all of the $\phi_i$ which are on the base clock are mutually synchronous, then

$$d(\phi_1 \cdot \cdots \cdot \phi_p)$$

can be evaluated with a memory of size $M(d)$.

3.7 Deadlock

Consider a node $d$. A variable $id$ in $d$ is deadlocked at instant $n$ on the input stream $\phi$ if no extension (addition of a suffix) of $\phi$ allows $id$ to extend the stream associated with it.

A stream can be blocked for several reasons:
• The clock of the stream, no matter what happens, will never be true again. For example, the stream defined by

\[ 1 \text{ when false} \]

is empty. This kind of deadlock is allowed in Lustre as it does not conflict with the synchronous behaviour of the program.

• Clocks in a program can be inconsistent. This is an error. A static clock calculus will be introduced to check the consistency of the clocks.

• A clock can take the value nil. This is an error. The code generator detects the use of uninitialised streams and ensures that a clock or an output parameter never take the value nil.

• The computation of a stream can not be done without knowing the future of a stream. Since all Lustre nodes are causal, this possibility does not exist.

• The computation of the value of a stream at a particular instant can not be done without already knowing that value. This is an error. In the static analysis, the dependencies between variables are calculated, and the compiler must ensure that there is no possible deadlock.

4 Clock calculus

The denotational semantics of Lustre defines the clock coherence in a dynamic manner. However, we have seen that it is possible for a stream to deadlock because of an inconsistency among the clocks.

Since Lustre was designed for areas where security is primordial, it would be useful to be able to statically determine that a Lustre program has no clock inconsistencies. To do this, we shall reduce the clock problem to a typing problem.

Two more or less disjoint problems must be addressed. The first is clock inference, which is reminiscent of type inference in ML, although the principle of definition must be respected.

The second is clock checking, which consists of determining statically that two boolean expressions actually define the same stream, independently of the input. In a language containing the integers, this problem is undecidable. Two options remain: either constrain the language in which clocks are defined until the equality problem becomes decidable, or introduce an equivalence between clocks, which is stronger than strict equality and which is statically evaluable.

The solution which was retained was to consider that two clocks are equivalent if the expressions defining them are syntactically equal, with respect to substitution. This equivalence is similar to structural equivalence in a recursive type system; it led us to define a syntactic normal form for a node.

4.1 Clock inference

Clock inference for a node has two objectives:

• determine for each expression the clock that it must have, while detecting the constraints which will have to verified later;

• introduce the clocks in the concrete syntax to end up with the abstract syntax.
4.1.1 Clock environments

The abstract syntax considers that the clock of an expression is either the base clock `base`, a boolean expression, or a list of clocks:

\[ h ::= \text{base} | e | (h, h) \]

It is important to understand that `base` represents a different clock for each node. It is a parameter of the node.

A clock environment associates a clock to each variable of the node. It can be written as:

\[ \omega = [id_i \text{ on } h_j] \]

If a variable `id` should not have a clock, \( \omega(id) = \bot \).

The abstract syntax for a node considers that expressions always carry their type, of which the clock is part. Consider the node

\[
\text{node } N \ (\text{in1, in2, in3 when in1}) \ \text{returns (out1, out2);}
\]

\[
\text{let}
\]

\[
\text{out1 = in1 -> in2;}
\]

\[
\text{out2 = (current in3) when in2;}
\]

\[
\text{tel;}
\]

The abstract syntax for this node is

\[
N 3 5 5
\]

\[
\text{in1 on base, in2 on base, in3 on (in1 on base),}
\]

\[
\text{out1 on base, out2 on (in2 on base)}
\]

\[
\text{out1 on base = (in1 on base -> in2 on base) on base}
\]

\[
\text{out2 on (in2 on base) =}
\]

\[
((\text{current (in3 on (in1 on base)) on base})
\]

\[
\text{when (in2 on base)) on (in2 on base)}
\]

which is not very readable, but corresponds exactly to the information required by the compiler (and as we have seen, by the denotational semantics).

For a node instantiation \( d \), all that is needed is the node’s signature \( \Omega(d) \). For \( N \) above, we get:

\[
N 3 5 5
\]

\[
\text{in1 on base, in2 on base, in3 on (in1 on base),}
\]

\[
\text{out1 on base, out2 on (in2 on base)}
\]

The clock of a parameter may have one of three forms:

- `base` means that the corresponding parameter must be on the base clock of the node.

- `cid_i` means that the corresponding parameter must have as clock the \( i \)-th parameter.

- `ck(base)` means that the corresponding parameter must be the clock of the base clock of the node; only the output of `current` (or `last`) uses this possibility.

Here are the signatures of the predefined nodes:

\[
\Omega(\text{op}_p) = \text{op } p p+1 \text{ in}_1 \text{ on base, } \ldots, \text{in}_p \text{ on base, out}_1 \text{ on base}
\]

\[
\Omega(\text{if}) = \text{op } 3 4 \text{ in}_1 \text{ on base, } \ldots, \text{in}_3 \text{ on base, out}_1 \text{ on base}
\]

\[
\Omega(\text{pre}) = \text{op } 1 2 \text{ in}_1 \text{ on base, out}_1 \text{ on base}
\]

\[
\Omega(\text{->}) = \text{op } 2 3 \text{ in}_1 \text{ on base, in}_2 \text{ on base, out}_1 \text{ on base}
\]

\[
\Omega(\text{when}) = \text{op } 2 3 \text{ in}_1 \text{ on base, in}_2 \text{ on base, out}_1 \text{ on in}_2
\]

\[
\Omega(\text{current}) = \text{op } 1 2 \text{ in}_1 \text{ on base, out}_1 \text{ on ck(base)}
\]
4.1.2 Node consistency

To define clock consistency, we use the predicate

\[ \omega \vdash eq : \omega', \varrho \]

which means that from the equation \( eq \), the environment \( \omega \) is augmented to yield \( \omega' \); during this phase, constraints are gathered up into \( \varrho \). Furthermore, the equation \( eq \) is rewritten into \( eq' \).

Each constraint is of the form \( e_1 = e_2 \), and its meaning is that in any variable environment, the expression \( e_1 \) is always equivalent to \( e_2 \). If \( \bot = e \) or \( e = \bot \) belong to a set of constraints, they are ignored.

The clock calculus associated with a node consists of defining the clock environment from the header, augmenting this environment through the equations, and then checking that the clocks of the parameters are themselves parameters. Finally, the clock consistency must be checked (cf. §4.2).

\[\omega = [id_i \text{ on } h_i],\]
\[\omega \vdash eq : \omega', \varrho\]
\[(\forall i \in [1..r]) \ h'_i \neq \bot\]
\[(\forall i \in [1..q]) \ h'_i = \text{base} \lor h'_i = \text{cid}_j', \ j \in [1..q]\]
\[\text{eq} \vdash \varrho\]
\[\omega \vdash \text{id} = e : \text{id} = e', \omega[h/id], \varrho\]

4.1.3 Consistency of a set of equations

To give the semantics of a system of equations, we need the predicate

\[ \omega \vdash e : e', h, \varrho \]

which means that in the environment \( \omega \), if the constraints \( \varrho \) are true, the clock of \( e \) is \( h \), and \( e \) is rewritten into \( e' \). It will be defined in the following subsection.

The definition principle has a great influence on the clock calculus. A first attempt at an inference rule, similar to that for the type calculus, would be:

\[\omega \vdash \text{id} = e : \text{id} = e', \omega[h/id], \varrho\]

But in Lustre, there are recursive equations, as in:

\[X = 0 \to \text{pre } X + 1;\]

One way or another, it is necessary to suppose that \( X \) has a clock before its consistency can be checked.

A natural solution would be that used for type inference in ML, i.e., by using unification:

\[\omega[h/id] \vdash e : e', h, \varrho\]

\[\omega \vdash \text{id} = e : \text{id} = e', \omega[h/id], \varrho\]

However, the following example shows that this rule violates the definition principle: if \( f \) and \( g \) are two nodes preserving clocks, from the system

\[X = f(Y);\]
\[Y = g(X);\]

it can be inferred that the clocks of \( X \) and \( Y \) are equal, but it cannot be determined what that clock is. However if we add an equation
\[ X = f(Y); \]
\[ Y = f(X); \]
\[ Z = X + 1; \]

the rule gives that the clocks of \( X \), \( Y \) and \( Z \) are all \textbf{base}. The context in which \( X \) was used would have determined its clock, which violates the definition principle.

One might ask if that principle is not too strict; after all, an automatic system could check that there is no inconsistency. This is true, but we are interested in programming systems where security is a high priority. If a programmer writes a node with many equations, it is more likely that she not make any errors if she can immediately know the clock of an equation simply by looking at it.

Furthermore, with the principle of definition, any part of a \textsc{Lustre} system is deterministic. Relaxing this principle would introduce nondeterminism, and would lead to the static analysis problems of \textsc{Signal} or \textsc{Esterel}.

We therefore use the first rule, but we use several iterations, i.e., we explicitly calculate a least fixed point:

\[
\frac{\omega \vdash eq : eq', \omega', g' \quad \omega \neq \omega'}{\omega \vdash eq : eq'', \omega'', g''}
\]

\[
\frac{\omega \vdash eq : eq', \omega', g' \quad \omega = \omega'}{\omega \vdash eq : eq', \omega', g''}
\]

In theory, if a clock must be given to each of \( n \) variables, \( n \) iterations might be required. In practice, few iterations are required, typically 2 or 3.

4.1.4 One step

The equations are analysed in order using the following rule:

\[
\frac{\omega \vdash eq : eq', \omega', g' \quad \omega' \vdash eqs' : eqs', \omega'', g''}{\omega \vdash eq \cdot eqs : eq \cdot eqs', \omega', g' \cup g''}
\]

The semantics of an equation is:

\[
\frac{\omega \vdash e : e', h', g}{\omega \vdash id \text{ on } h = e : id \text{ on } h' = e', \omega[h'/id], g \cup \{ \omega(id) = h' \}}
\]

4.1.5 Expression coherence

A constant whose clock has not been attributed by the programmer is on the base clock:

\[
\omega \vdash k \text{ on } \bot : k \text{ on base, base, } \emptyset
\]

Otherwise, the clock that the programmer gave is used:

\[
\omega \vdash k \text{ on } h : k \text{ on } h, h, \emptyset
\]

For variables, it suffices to look in the environment:

\[
\omega \vdash id \text{ on } h : id \text{ on } h', h', \{ h = h' \}
\]

The clock of a tuple is formed from the clocks of its elements:

\[
\omega \vdash e_1 : e'_1, h_1, g_1 \quad \omega \vdash e_2 : e'_2, h_2, g_2
\]
\[
\omega \vdash (e_1, e_2) : (e'_1, e'_2), (h_1, h_2), g_1 \cup g_2
\]
4.1.6 Semantics of a node instantiation

Before we define the semantics of an instantiation, we introduce two new predicates:

The predicate
\[ \text{newb}, e \vdash h, \text{in} : \text{newb}', g' \]
is used to calculate the effective base clock of a node instantiation. The partial value upto the present is \( \text{newb} \), the new value, calculated from the expression \( e \) and the clock \( h \), is \( \text{newb}' \). The accumulated constraints are \( g' \).

The predicate
\[ \text{newb}, d, e \vdash \text{out} : h \]
means that given the effective base clock \( \text{newb} \) and the expression \( d, e \), the formal clock \( \text{out} \) becomes the effective clock \( h \).

The instantiation \( d, e \) is given a clock by calculating the effective base clock defined by the expression \( e \) and by calculating the clocks of the outputs:

\[
\begin{align*}
\Omega(d) &= (\text{id} p q \omega_d) \\
\omega &\vdash e : e', h, g \\
\bot, e' &\vdash h, [\omega_d]_{1..p} : \text{newb}, g' \\
\text{newb}, d, e' &\vdash [\omega_d]_{p+1..q} : h' \\
\omega &\vdash d e : d e', h', g' \cup g'
\end{align*}
\]

We now define the predicate \text{input}. On a list of parameters, we iterate in a standard manner:

\[ \text{newb}, e \vdash [], [] : \text{newb}, \emptyset \]
\[ \text{newb}, e \vdash h, \text{in} : \text{newb}', g' \]
\[ \text{newb}', e \vdash h \cdot \text{hs}, \text{ins} : \text{newb}'', g'' \]
\[ \text{newb}, e \vdash h \cdot \text{hs}, \text{in} \cdot \text{ins} : \text{newb}'', g' \cup g'' \]

The case for \text{base} is treated differently if it already has a value:

\[ \bot, e \vdash h, \text{id on base} : h, \emptyset \]
\[ \text{newb} \neq \bot \]
\[ \text{newb}, e \vdash h, \text{id on base} : \text{newb}, \{h = \text{newb}\} \]

For the case of an input clock, we generate a constraint:
\[ \text{newb}, e \vdash h, \text{id on id} : \text{newb}, \{h = \text{select}, e\} \]

We now define the predicate \text{output}. On a list of parameters, we iterate in a standard manner:

\[ \text{newb}, d, e \vdash [], [] \]
\[ \text{newb}, d, e \vdash \text{out} : h \]
\[ \text{newb}, d, e \vdash \text{outs} : \text{hs} \]
\[ \text{newb}, d, e \vdash \text{out} \cdot \text{outs} : h \cdot \text{hs} \]

The formal base clock becomes the effective base clock:
\[ \text{newb}, d, e \vdash \text{id on base} : \text{newb} \]
The formal clock $ck(\text{base})$ becomes the clock of $\text{newb}$:

\[
\text{newb, d, e} \vdash id \ on \ ck(\text{base}) : ck(\text{newb})
\]

The others are obvious:

\[
i \leq p \quad \Omega(d) = (id \ p \ q \ \omega_d)
\]

\[
\text{newb, d, e} \vdash id \ on \ id_i : \text{select}_i \ e
\]

\[
i > p \quad \Omega(d) = (id \ p \ q \ \omega_d)
\]

\[
\text{newb, d, e} \vdash id \ on \ id_i : \text{select}_i \ d \ e
\]

### 4.2 Constraint consistency checking

The rules of the previous section construct a set of constraints for each node; we must ensure that these constraints are satisfied before declaring that the node is coherent with respect to the clocks.

Statically checking that a constraint between clocks is satisfied is similar to statically determining the equivalence between two subprograms, which is undecidable: the clock language is too complex. We are therefore forced to consider a more constrained clock language which is decidable, or to find a clock equivalence relation which is stronger than simple equality. We choose the latter option: the consequence will be that some constraints will be declared to be unsatisfied when in fact they could be accepted.

The simplest equivalence would be syntactic equality. According to that rule, the equation

\[
C = A \ \text{when} \ (X=Y) + B \ \text{when} \ (X=Y);
\]

would be consistent, unlike the following system

\[
C = A \ \text{when} \ (X=Y) + B \ \text{when} \ (N(X)=Y);
\]

where node $N$ is defined as

\[
\text{node } N(x:*) \ \text{returns} \ (y:*) ; \ \text{let} \ y = x ; \ \text{tel};
\]

So, that rule would violate the substitution principle.

Mutually recursive equations introduce new problems. Consider the following system:

\[
C = A \ \text{when} \ X + B \ \text{when} \ Y;
X = \text{true} \rightarrow \text{not pre} \ Y;
Y = \text{true} \rightarrow \text{not pre} \ X;
\]

By substituting the definition for $X$ in that for $Y$, and vice versa, we get the following system

\[
C = A \ \text{when} \ X + B \ \text{when} \ Y;
X = \text{true} \rightarrow \text{not pre} \ (\text{true} \rightarrow \text{not pre} \ X);
Y = \text{true} \rightarrow \text{not pre} \ (\text{true} \rightarrow \text{not pre} \ Y);
\]

and the clocks are obviously equivalent.

The equivalence relation that we choose is therefore syntactic equivalence, with respect to renaming and substitution; in the parlance of type-checking, we consider structural equivalence between recursive types.

Constraint checking consists of several steps:

1. The abstract syntax tree is manipulated so that all clocks are variables. Because of the substitution principle, this causes no problem.

2. Node instantiations are replaced by equivalence sets of equations.

3. We verify the constraints, using the new equations. One way to do this is to put the equations in a canonical form. This problem was addressed in a general manner by Courcelle, Kahn and Vuillemin [10]. Because the recursion in LUSTRE is more constrained, we can find a more efficient algorithm.
4.2.1 Abstract syntax tree manipulations

The abstract syntax tree of the equations is manipulated so that every clock can be associated with an identifier and so that the effective input and output parameters of a node instantiation are also identifiers.

For example, the equation

\[ X = E \text{ when } (A \text{ or } B); \]

becomes

\[ X = E \text{ when } T_1; \]
\[ T_1 = A \text{ or } B; \]

where \( T_1 \) is a new variable; the equation

\[ X = 5 + N(Y+Z); \]

becomes

\[ X = 5 + T_1; \]
\[ T_1 = N(T_2); \]
\[ T_2 = Y+Z; \]

where \( T_1 \) and \( T_2 \) are new variables.

4.2.2 Node expansion

To present node expansion, consider the node declaration

\[
\begin{align*}
\text{node } N(X_1, \ldots, X_p) \text{ returns } (X_{p+1}, \ldots, X_q); \\
\text{var } X_{q+1}, \ldots, X_r; \\
\text{let } X_{p+1} = E_1; \ldots; X_r = E_{r-p}; \text{ tel;}
\end{align*}
\]

where there are only basic operators (noted \( OP \)) in the expressions \( E_i (i = 1..r-p) \).

Consider furthermore the instantiation

\[
W_{p+1}, \ldots, W_q = N(W_1, \ldots, W_p);
\]

where the \( W_i (i = 1..q) \) are variables, possibly created by a manipulation of the abstract syntax tree. We suppose the existence of variables \( W_i (i = q+1..r) \) with which we will rename the local variables of the node.

The node instantiation is translated into

\[
W_{p+1}, \ldots, W_r = (T(E_1), \ldots, T(E_{r-p}));
\]

where the transformer \( T \) is defined by:

\[
\begin{align*}
T(\text{base}) &= BASE \\
T(k \text{ on } h) &= k \text{ on } T(h) \\
T(X_i \text{ on } h) &= W_i \text{ on } T(h) \\
T(OP(e_1, \ldots, e_n)) &= OP(T(e_1), \ldots, T(e_n))
\end{align*}
\]

where \( BASE \) is the base clock of the instantiation (known by now). With the introduction of all those variables, the \text{select}_i become unnecessary.
4.2.3 A variable and its equation

If it is possible to determine the syntactic equivalence between two expressions, it becomes possible to find a syntactic normal form for a program. It suffices to ensure that an expression is only defined once and to eliminate all variables which are not outputs or memories.

Courcelle's, Kahn's and Vuillemin's algorithm can be applied in a general manner to any recursive system of equations. The system is decomposed so that there is one operator for each equation, the minimal set of equations is found and the system is reconstructed.

In Lustre, the occurrence of a variable in its definition (a recursive occurrence) can appear in three different manners:

- A variable \( id \) can depend on its own previous values, i.e., there are recursive occurrences of \( id \) appearing as operands of a `pre`. Two examples are:

  \[
  \begin{align*}
  X &= 0 \rightarrow \text{pre } X + 1; \\
  X &= 0 \rightarrow \text{pre } Y - 1; \\
  Y &= 1 \rightarrow \text{pre } X + 2;
  \end{align*}
  \]

  In the generated code, the argument of a `pre` must be memorised. A variable will be required for this step. Since variables in a Lustre program are going to become variables in the generated code, the abstract tree is manipulated so that the argument of each `pre` is an identifier.

- Two variables (or more) can depend upon each other, without there being deadlock. For example, in

  \[
  \begin{align*}
  Y &= \text{if } D \text{ then } Z \text{ else } B; \\
  Z &= \text{if } D \text{ then } C \text{ else } Y;
  \end{align*}
  \]

  at any given instant, depending on the value of D, the value of Y can depend on that of Z, or the reverse, but not both. The current compiler cannot handle such cases, but a more intelligent compiler might be able to. In fact, if the definitions of Y and Z are substituted into the equations, we get

  \[
  \begin{align*}
  Y &= \text{if } D \text{ then } (\text{if } D \text{ then } C \text{ else } Y) \text{ else } B; \\
  Z &= \text{if } D \text{ then } C \text{ else } (\text{if } D \text{ then } Z \text{ else } B);
  \end{align*}
  \]

  which is equivalent, although not syntactically, to

  \[
  \begin{align*}
  Y &= \text{if } D \text{ then } C \text{ else } B; \\
  Z &= \text{if } D \text{ then } C \text{ else } B;
  \end{align*}
  \]

- A variable depends directly upon itself. This is a deadlock, always an error. An example is:

  \[
  X = X + Y;
  \]

So, since the current code generator considers the last two cases to be similar, the only recursive occurrences which are allowed are those in a `pre`. Hence in Lustre, it is possible to know where the recursive variables are. This means that there is no need to decompose the system of equations.

Before the syntactic equivalence classes of variables can be calculated, the equation associated with a variable must be determined. But the programmer can introduce new variables in an arbitrary manner. For example, if A and B are inputs,
\[ Z = W + B; \]
\[ W = A; \]
is equivalent to
\[ Z = A + B; \]
\[ W = A; \]
and \( W \) can be eliminated if it is not an output. It would therefore be useful to replace with its definition each occurrence of a variable which is not an input.

Since the only permitted recursive occurrences are the arguments of the \texttt{pre}, we can with all impunity replace the occurrence of each variable by its definition, unless its occurrence is in a \texttt{pre}.

### 4.2.4 The equivalence classes

If recursivity is taken into account, determining if two variables are equivalent is similar to determining of two states in a finite state automaton are equivalent. Two variables are equivalent if the right hand sides of their “real” equations are similar, i.e., the only points where they differ is in the names of the memory or recursive variables, and the corresponding variables in those points are themselves equivalent.

The two expressions \( e \) et \( e' \) are similar if the predicate

\[ \text{sim} \vdash e, e' \]

is true. It is defined by

\[ \text{sim} \vdash k, k \]
\[ \text{sim} \vdash \text{id}, \text{id} \]
\[ \text{sim} \vdash \text{pre id}, \text{pre id} \]
\[ \text{sim} \vdash e_1, e'_1 ; \cdots ; e_n, e'_n \]
\[ \text{sim} \vdash \text{OP}(e_1, \ldots, e_n), \text{OP}(e'_1, \ldots, e'_n) \]

where \( \text{OP} \) is any operator other than \texttt{pre}.

This algorithm does the initial partition. Clearly, in the compiler, hashing the expressions means that the expressions are but rarely compared.

To refine the partitions, it is necessary to have a list of necessary variables in the definition of each variable. This is done with the following predicate:

\[ \text{find} \vdash e : \text{lid} \]

which is defined by:

\[ \text{find} \vdash k : [] \]
\[ \text{find} \vdash \text{id} : [] \]
\[ \text{find} \vdash \text{pre id} : [\text{id}] \]
\[ \text{find} \vdash e_1 : \text{lid}_1 \cdots \vdash e_n : \text{lid}_n \]
\[ \text{find} \vdash \text{OP}(e_1, \ldots, e_n) : \text{lid}_1 \cdots \text{lid}_n \]
Finally, the equivalence between two variables is formally defined: it is the greatest fixed point of the relation defined by the predicate

\[
\text{eq} \vdash \text{id}, \text{id}'
\]

where eq is the set of equations.

Its definition is:

\[
\begin{align*}
\text{eq}(\text{id}) &= e & \text{eq}(\text{id}') &= e' \\
\text{find} &\vdash e, e' \\
\text{find} &\vdash e : [\text{id}_1, \ldots, \text{id}_n] & \text{find} &\vdash e' : [\text{id}'_1, \ldots, \text{id}'_n] \\
\text{eq} &\vdash \text{id}_1, \text{id}'_1 & \cdots & \text{eq} &\vdash \text{id}_n, \text{id}'_n \\
\text{eq} &\vdash \text{id}, \text{id}'
\end{align*}
\]

4.2.5 The normal form

The normal syntactic form of a system of equations consists of keeping the definitions of the output parameters and the definition of a representative of the equivalence class of each memory or recursive variable. Since we do not compile nodes with deadlocks, we consider that all such nodes define the Error node. We have the following proposition:

**Proposition 3** If two nodes are syntactically equivalent with respect to substitution, their normal forms are identical, with respect to the names of variables and the order of the equations.

For nodes containing a deadlock, this result is trivial. If two systems \(Q_1\) and \(Q_2\) of equations are syntactically equivalent, their normal forms \(Q'_1\) and \(Q'_2\) are as well. Now, if \(id_1 = e_1 \in Q'_1\), then there must be a \(id_2\) such that \(id_2 = e_2 \in Q'_2\) and \(\vdash id_1, id_2\). Similarly the other way. But by the definition of the normal form, there can only be one such variable in each system and the proposition’s result follows.

Future Work

There is a prototype compiler for Lustre, which more or less corresponds to the semantics. What is interesting is the method of compilation, which synthesizes the control in the form of a finite state automaton; the generated code is very efficient. Since the compiler symbolically evaluates the program, it is in itself a formal verification system for Lustre. Furthermore, many of these notions could be extended to more general dataflow languages, such as Lucid.

Acknowledgements

I only joined the Lustre project in Grenoble once the basic language had been designed. I am indebted to my colleagues in Grenoble, Nicolas Halbwachs, Paul Caspi and Daniel Pilaud. The years of discussion taught me how to do research. I also wish to thank Gérard Berry for helping me focus on the problems of abstract syntax and semantics. The language is much cleaner as a result.

References


