

# Some thoughts on dataflow real-time programming languages

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

April 19, 1991

## Abstract

We propose a new language, RLUCID, which is LUCID with one new operator. RLUCID is intended for real time (or, more precisely, reactive) programming and is based on a simple synchronous dataflow model of computation. RLUCID is therefore similar to the LUSTRE language but is more general in that it does not require that operators be strict: in the RLUCID model a filter may respond when input arrives on only some of its input channels. The new operator before allows the programmer to make decisions based on the order of arrival of input; it is especially useful for programming interfaces. A simple least fixedpoint semantics of RLUCID is given using a domain of *time-stamped streams*. We discuss the possibility of implementing RLUCID.

## 1 Introduction

Reactive programs are programs which interact *continually* with their environments. They are at the core of many computerized systems, from switchyard controllers to electronic gadgets. Reactive programs include real-time programs, which are subject to “hard” timing constraints imposed by sampling frequency and desired response times.

Reactive programs produce no spontaneous output; output is generated *only* in response to external stimuli, though not necessarily to all. Programs are normally expected to run without terminating, unless they are deliberately stopped.

Many reactive systems are used in situations where reliability is crucial, for an error could lead to a major disaster. To increase reliability, all software should be specified and programmed using languages with clear and formal semantics. When time plays a *direct* role in the behavior of a system, this aspect should be expressible in the programming language. It should be clear not only *what* a program does, but *when* it does it. Furthermore, constructs are needed to allow decisions to be made according to the order of arrival of the input.

Since timing constraints can be critical, it would be desirable to have a language whose programs could be run with a predictable execution time and memory use. An efficient implementation would also be desirable.

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 *directly accessible to the user*. 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 those 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 behaviors, depending on the implementation architecture; this is a serious deficiency in the design of portable code.

One formal model which is capable of describing almost any existing timed system was developed by Caspi and Halbwachs [1]. In their model, every timed system has a set of basic events, such as "assignment to variable X" or "emission of signal S". There will be many *dated* occurrences of each event, and the sets of histories of these occurrences form the description of the system.

To this basic model, they added various standard mathematical tools to functionally express precedence relations, histories of variables and filtering. They successfully proved the proper functioning of a bus arbiter.

The set of values taken by a variable forms a sequence, and the  $n^{\text{th}}$  value is taken at the  $n^{\text{th}}$  assignment to it. Since this notion is commonly found in dataflow languages, there have been several attempts to design dataflow languages which could describe reactive systems.

This paper presents RLucid, a dataflow language whose semantics is given using time-stamped streams. We begin by presenting the operators of Ashcroft and Wadge's Lucid [6]. Although Lucid is elegant, it is not sufficient for reactive programming. We then present Lustre [2], which was expressly designed for programming reactive systems. Although very useful, Lustre suffers from the inability to describe the interaction between a reactive program and its environment. Finally, we present RLucid, which grew out of Lucid and is more powerful than Lustre.

## 2 LUCID

Lucid programs are functions between streams. A stream is an infinite sequence of values, written as

$$(e_0, e_1, \dots, e_n, \dots)$$

The body of a Lucid program is an expression. Local variables are defined by sets of mutually recursive equations. Free variables are input to the program.

A trivial Lucid program is

```
(x*y)/2
  where
    y=x+1;
  end;
```

If  $x$  is the stream

$$(x_0, x_1, \dots, x_n, \dots)$$

then the output of that program is

$$((x_0 * (x_0 + 1))/2, (x_1 * (x_1 + 1))/2, \dots, (x_n * (x_n + 1))/2, \dots)$$

Expressions are built up from variables, constants, data operators, temporal operators and local functions. The latter are defined in the same manner as programs.

A constant in an expression represents the infinite sequence, all of whose elements (or *datons*) are that constant. So 1 means

$$(1, 1, \dots, 1, \dots)$$



Data operators are all the usual arithmetic, boolean and relational operators, or even ones supplied somehow by the user. These operators work in a pointwise manner. The  $n^{\text{th}}$  daton of the result is computed from the  $n^{\text{th}}$  datons of the operands. For example,  $X+Y$  means

$$(x_0 + y_0, x_1 + y_1, \dots, x_n + y_n, \dots)$$

It is the temporal operators that allow the user to do something interesting. If  $E$  and  $F$  are respectively  $(e_0, e_1, \dots, e_n, \dots)$  and  $(f_0, f_1, \dots, f_n, \dots)$  then

$$\text{first } E = (e_0, e_0, \dots, e_0, \dots)$$

$$\text{next } E = (e_1, e_2, \dots, e_{n+1}, \dots)$$

$$E \text{ fby } F = (e_0, f_0, f_1, \dots, f_{n-1}, \dots)$$

For example,

$$X = 0 \text{ fby } X+1$$

counts the natural numbers.

Similarly,

$$Y = \text{init fby } (a*Y + b * \text{next } X)$$

would define the first order linear filter

$$y_0 = \text{init}$$

$$y_{n+1} = ay_n + bx_{n+1}, \quad n \geq 0$$

LUCID is a member of the ISWIM family. In other words, it is just syntactically sugared lambda calculus with a least-fixedpoint semantics for recursive definitions.

To specify the semantics of LUCID we need only specify the domain of streams (with an ordering), and the basic operations over these streams. The semantics of LUCID is then just an instance of the “generic” semantics of the ISWIM family.

We assume that we have an underlying algebra of simple ‘atomic’ data types — integers, characters, booleans and the like — together with the usual arithmetic and logical operations.

The normal semantics of LUCID is given through a tagged dataflow mechanism. All streams of datons are infinite; some of the datons may be undefined ( $\perp$ ). The LUCID domain is the set of all such streams along with the following order: stream  $A$  is more defined than stream  $B$  if each daton of  $B$  is either the corresponding one of  $A$  or is  $\perp$ . The basic operations over the streams are defined above.

LUCID’s semantics are elegant, but contain no notion of time. There is no way to determine *when* a program will do something, if ever. This problem will be solved in RLUCID by introducing timestamps.

LUCID also has no efficient implementation. In fact, right now, there is no way to estimate the required memory and execution time to run a program. These problems will be addressed by LUSTRE.

### 3 LUSTRE

LUSTRE (Synchronous Real-Time Lucid — in French) was designed as an efficiently implementable subset of LUCID suitable for reactive programming. This was done by introducing draconian restrictions on the input (they must be highly synchronized) and by assuming the synchronous hypothesis.

	E =	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>4</sub>	e <sub>5</sub>	...
	C =	tt	ff	tt	tt	ff	...
	X = E when C =	e <sub>1</sub>		e <sub>3</sub>	e <sub>4</sub>		...
Y = 1 when C ->	X =	1		e <sub>3</sub>	e <sub>4</sub>		...
Z = current Y =		1	1	e <sub>3</sub>	e <sub>4</sub>	e <sub>4</sub>	...

Table 1: Sampling and projection

If two streams are to be joined somehow in a pointwise manner, and of a bounded memory is required, then there must be a bound on how far out of sync those two streams are. In *LUSTRE*, the task is simple: the streams must be perfectly synchronous.

This requirement forced a change in the temporal operators. The two main ones in *LUSTRE* are *pre* and *->*.

$$\begin{aligned} \text{pre } E &= (\text{nil}, e_0, e_1, \dots, e_{n-1}, \dots) \\ E \rightarrow F &= (e_0, f_1, f_2, \dots, f_n, \dots) \end{aligned}$$

*LUCID*'s

$$X = 0 \text{ fby } X+1$$

becomes

$$X = 0 \rightarrow \text{pre } X + 1$$

These two operators are sufficient to describe perfectly synchronous systems. But there are situations when it makes no sense to have a value in a stream. To allow for this, an operator to sample streams according to a boolean expression was introduced. *E when C* produces the stream consisting of values of *E* when the corresponding value of *C* is true.

Since streams are no longer necessarily synchronous, but the data operators are strictly synchronous, the notion of stream is changed to that of a *clocked stream*. A clocked stream consists of a sequence of values and a clock, which defines the sequence of instants at which those values appear. If the sequence of values is

$$(e_0, e_1, \dots, e_n, \dots)$$

the stream takes the value  $e_n$  at the  $n^{\text{th}}$  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.

So, if *E* and *C* are on the same clock, then *E when C* has clock *C*. Data operators are allowed to operate over streams of the same clock. With *when*, one can create cascades of ever coarser clocks, thereby forming a tree with the base clock at root.

The projection operator *current* creates a finer stream whose clock is the clock of the clock of the argument. We illustrate the two operators in Table 1.

*LUSTRE*'s semantics are much more complex than are *LUCID*'s, for two reasons:

- The *pre* operator produces and the *current* operators can produce occurrences of the undefined *nil*. If a boolean stream is used as a clock, and it contains an occurrence of *nil*, what does that mean? Should *nil* be assimilated to false, or should the program deadlock? The latter choice was made.



- The clock of a constant is difficult to define. The synchronous hypothesis tells us that calculations are done infinitely fast; for constants, that means that their value should be made available instantaneously at the beginning of the execution of the program. But the inputs to a strict operator must be on the same clock. A choice was therefore made to consider the clock of a constant to be the clock of the input of the node in which it appears.

For example, consider the node

```
node ADD (x:int) returns (y:int);
let
  y = x+1;
tel;
```

The clock of the 1 is perceived to be the same as the clock of x.

This choice has the unfortunate implication that LUSTRE cannot be considered a member of the iswim family. For example, consider the following node:

```
node ADD_WHEN_C (x:int; c:bool) returns (y:int);
let
  y = ADD(x when c);
tel;
```

The expression `ADD(x when c)` cannot simply be replaced by `x when c + 1`. For the semantics to work, the expression must become `x when c + 1 when c`.

In both cases, problems arise because of the use in the semantics of concepts which are not directly accessible from the language itself, namely `nil` and clocks.

Nevertheless, LUSTRE does have a very efficient implementation. The static semantics includes a clock calculus, which determines which LUSTRE programs have no clock inconsistencies. Furthermore, by symbolically executing the boolean expressions in a program, that program can be compiled into a finite state automaton. The resulting code is much more efficient than one would write by hand. Furthermore, this technique yields a formal verification system on the boolean subset of the language.

If we return to the question of suitability for reactive programming, LUSTRE has been used to program the kernel of such systems; however, because of its insistence on strict synchrony between its input, it can rarely be used directly with its environment. A buffer program has to be written to handle the interface between the kernel and its environment. It is this problem which will be addressed by RLUCID, through the introduction of a new operator.

## 4 RLUCID

RLUCID is LUCID with two new operators, `ck` and `before`, and with a timed pipeline dataflow semantics. The basic entity in RLUCID is a *timestamped stream*. The stream

$$([x_0, s_0], [x_1, s_1], \dots, [x_n, s_n], \dots)$$

means that the daton  $x_n$  occurs at time  $t_n$ .

Timestamps form a totally ordered set, with a minimal element, which we call 0. The `ck` operator generates a time-stamped stream whose value is the date itself. So the semantics for `ck X` is

$$([s_0, s_0], [s_1, s_1], \dots, [s_n, s_n], \dots).$$

The semantics of timestamps is based on the notion of an operator network running infinitely fast. So the semantics for `X+Y` is

$$([x_0 + y_0, \max(s_0, t_0)], [x_1 + y_1, \max(s_1, t_1)], \dots, [x_n + y_n, \max(s_n, t_n)], \dots)$$

It is assumed here that the timestamps in a sequence are nondecreasing.

Among the basic set of operators is the synchronizing pairing operator `||`. The semantics for `X || Y` is

$$([(x_0, y_0), \max(s_0, t_0)], [(x_1, y_1), \max(s_1, t_1)], \dots, [(x_n, y_n), \max(s_n, t_n)], \dots).$$

`fst` and `scnd` are used to access the first and second elements of a pair.

For non-strict operators such as `fby`, we must ensure that the timestamps remain that way. So the semantics of `X fby Y` are

$$([x_0, s_0], [y_0, \max(s_0, t_0)], [y_1, \max(s_1, t_1)], \dots, [y_{n-1}, \max(s_0, t_{n-1})], \dots)$$

By changing the semantics and introducing the `ck` operator, we have not increased the power of the language. This is done by introducing `before`. `A before B` will return true if the first daton of `A` arrives before the first daton of `B`. We can now make decisions based on the relative order of arrival of the input.

New operators can now be defined with the use of these two operators. We give two examples:

- `A on C` generates a stream whose values are those of `A` but whose time-stamps are those of `C` if `A` is faster than `C`:

$$A \text{ on } C = \text{fst}(A || C);$$

Suppose we wish to count the occurrences of an event `A`, exactly when they occur. This would be done as follows:

```
count on (0 fby ck A)
where
  count = 0 fby count + 1;
end;
```

To generate the equivalent expression in `LUSTRE` is really difficult.

- `A onto C` outputs the last value held by `A` every time that `C` occurs. It is defined as follows:

```
A onto C = current(A, C, first A)
where
  current(A, B, C) =
    if A before B
    then C fby current(A, next B, C)
    else which(next A, B, first A);
end;
```



Suppose that we wish to take the count of occurrences of A from above and use it whenever B turns up. The result would be:

```
(0 fby occurs) onto ck B
where
  occurs = count on (0 fby ck A);
  count = 0 fby count + 1;
end;
```

This expression would also be difficult to write in LUSTRE.

## 4.1 Small examples

The before operation now allows us to merge two streams which are running completely independently. MergeLeft merges two streams, favoring the left one if they occur at the same moment:

```
MergeLeft(A,B) =
  if B before A
  then B fby MergeLeft(A, next B)
  else A fby MergeLeft(next A, B);
```

Interfaces for LUSTRE programs can also be written in RLUCID. Suppose that two pure signals arrive, more or less independently. For a LUSTRE program to run, then two boolean streams would have to be created, each true when the appropriate signal arrived. Furthermore, those two streams would have to be simultaneous. Finally, it is typical to simplify the programming by assuming that only one of the input can be true at a given moment. The interface becomes:

```
SignalLeft(A,B) =
  if B before A
  then (false||true) fby SignalLeft(A, next B)
  else (true||false) fby SignalLeft(next A, B);
```

In his master's thesis [5], Vempati describes a language for interfaces called APRIL. APRIL consists of a number of functions written in RLUCID.

## 4.2 Semantics

A timestamped stream in RLUCID is a pair  $(V, T)$ , where  $V$  is the stream of datons (values) and  $T$  is the stream of timestamps (dates). The correct interpretation is that the  $n^{\text{th}}$  daton of  $V$  occurs at the time given by the  $n^{\text{th}}$  date in  $T$ . The semantics for an RLUCID expression,  $E$ , is two LUCID programs,  $\mathcal{V}(E)$  and  $\mathcal{D}(E)$ .  $\mathcal{V}(E)$  gives the daton streams and  $\mathcal{D}(E)$  gives the timestamp streams.

To differentiate between RLUCID and LUCID, we write the RLUCID operators in majuscules. The abstract syntax of RLUCID is therefore:

$$\begin{aligned}
 E &::= X \mid K \mid E \text{ Op } E \mid \text{If } E \text{ Then } E \text{ Else } E \mid \text{Next } E \mid E \text{ Fby } E \mid E \text{ Before } E \mid \\
 &\quad \text{Ck } E \mid (E \parallel E) \mid (E, E) \mid \text{Fst } E \mid \text{Scnd } E \mid X E \mid E \text{ Where } Q \text{ End} \\
 Q &::= X = E \mid X X = E \mid Q; Q
 \end{aligned}$$

where  $K$  is a constant,  $X$  is an identifier,  $Op$  is a data operator, such as the arithmetic, boolean and relational operators.  $Q$  is an equation or a group of them, such as those found in the body of a where clause.

Since R LUCID is a member of the ISWIM family, we only give the definitions of the basic operators. To give the values of the Before operation, we need to know the timestamps. Similarly, to give the timestamps of the If-Then-Else operation, we need to know the values taken by the condition.

$VX = x_V$	$DX = x_D$
$VK = k$	$DK = 0$
$V(E_1 \text{ Op } E_2) = VE_1 \text{ op } VE_2$	$D(E_1 \text{ Op } E_2) = \max(DE_1, DE_2)$
$V(\text{If } E_1 = \text{if } VE_1$	$D(\text{If } E_1 = \max(\text{if } VE_1$
Then $E_2$ then $VE_2$	then $\max(DE_1, DE_2)$
Else $E_3$ else $VE_3$	else $\max(DE_1, DE_3),$
	0 fby $D(\text{If } E_1 \text{ Then } E_2 \text{ Else } E_3))$
$V(\text{Next } E) = \text{next } VE$	$D(\text{Next } E) = \text{next } DE$
$V(E_1 \text{ Fby } E_2) = VE_1 \text{ fby } VE_2$	$D(E_1 \text{ Fby } E_2) = \max(\text{first } DE_1, DE_1 \text{ fby } DE_2)$
$V(E_1 \text{ Before } E_2) = \text{first } DE_1 < \text{first } DE_2$	$D(E_1 \text{ Before } E_2) = \min(\text{first } DE_1, \text{first } DE_2)$
$V(\text{Ck } E) = DE$	$D(\text{Ck } E) = DE$
$V(E_1 \parallel E_2) = (VE_1, VE_2)$	$D(E_1 \parallel E_2) = \max(DE_1, DE_2)$
$V(E_1, E_2) = (VE_1, VE_2)$	$D(E_1, E_2) = (DE_1, DE_2)$
$V(\text{Fst } E) = \text{fst } VE$	$D(\text{Fst } E) = \text{fst } DE$
$V(\text{Scnd } E) = \text{scnd } VE$	$D(\text{Scnd } E) = \text{scnd } DE$

### 4.3 Implementation

For the moment, R LUCID can be used as an elegant specification language. There is no implementation, except in a very awkward manner through the existing LUCID interpreters. For R LUCID to be truly usable in a real-time environment, then a compiler with the following properties would have to be designed:

- It is possible to determine statically if a program cannot put itself in a position of deadlock.
- It is possible to determine statically if a program will not attempt to generate an infinite amount of output in a single instant.
- It is possible to determine statically if a program can be run with a determined, bounded memory.
- Good quality code can be generated for a program.

The first three properties are clearly undecidable. However, a static semantics can be devised in such a way that all programs which successfully pass the static analysis fulfill the first three properties. The question which must then be asked is, how restrictive will this static semantics be? Once the first three properties are fulfilled, then there should be no problem generating good quality code.

We now consider each property separately.

#### 4.3.1 Deadlocks

Deadlocks can occur in dataflow networks because there is a short-circuit in the computation. One of the inputs to a node is the output of a second node, and vice versa. To avoid such a situation, a dependency graph can be calculated from the equations. A variable  $A$  is said to depend on a variable  $B$  if, in any possible flow of control, the calculation of  $A$ 's value depends on the knowledge of  $B$ 's value. Clearly, if there is no cycle in the dependency graph of a program, then there are no deadlocks in the program. This is a conservative choice, as this dependence might only be at one particular instant. Nevertheless, it works.



### 4.3.2 Infinitely fast programs

It is not difficult to generate an infinitely fast R L U C I D program. For example, the programs

```
0
and
  first X
```

run infinitely fast (if X ever arrives). Now, there is no problem with a program which has a subprogram which theoretically runs infinitely fast, so long as the output has been slowed down by operations such as on and onto. For example, the program

```
0 on X
```

where X is an input, is fine, as the 0 has been slowed down.

Infinitely fast subprograms can also be generated through the creation of recursive functions, where the input never advances. For example, one could define the first operation as follows:

```
first X = X fby first X;
```

Because the input never advances in the recursive call, once the first input daton arrives, an infinite stream is immediately output.

The static analysis phase must therefore be capable of determining if an entire network is capable of generating an infinite amount of output in a single instant.

Function calls create difficulties, because of the possibility of recursive calls. Essentially, a recursive call will be assumed to generate infinite output in a single instant, unless the input lines have not changed, and at least one of the lines used has advanced. This is the case for the current node defined above.

### 4.3.3 Bounded memory

There are two ways for a dataflow network to require an unbounded amount of memory.

First, through recursive instantiations, the network itself can grow with the creation of more and more new nodes. This possibility can be avoided by ensuring that all recursive calls are tail-recursive, i.e., every recursive call simply replaces the current computational node.

Second, the arcs between certain computational nodes can be forced to accept unlimited numbers of datons. For example, the following expression cannot be computed with a bounded memory:

```
X + X wvr half
where
  half = true fby false fby half;
end;
```

To solve this problem, a concept of "clock" must be devised. Strict operations such as + will only be allowed if the operands of the operation are "on the same clock." But how should a clock be defined?

The first possible definition would be that streams A and B would be considered to be on the same clock if, at any instant, the difference between the number of datons in A and that in B is less than a fixed number. To be useful, this definition would have to ensure that when two streams on the same clock pass through a filter, the result remains on the same clock. This is certainly not true, as one can devise filters which can generate radically different results.

The next possible definition would be that streams A and B would be considered to be on the same clock, if at one instant, either one or both of the streams has become infinitely fast, or the two streams occur at exactly the same instants.

#### 4.3.4 Efficient Code

Once the first three properties have been ensured, generating high quality code should not be too difficult. Techniques such as were used for LUSTRE are entirely applicable for RLUCID, with the added difficulty of dealing with recursive instantiations. Since those instantiations have been curtailed to tail-recursion, even they should be manageable.

## 5 Related Work

LUCID has been considered as a possible language for both programming and specifying real-time systems. In [3], Faustini and Lewis describe a LUCID with hiaton streams, where ordinary LUCID streams have 'hiatons' interspersed between the datons when data is not available. However, the semantics is unclear. In [4], Skillicorn and Glasgow describe the use of LUCID for real-time specification. Essentially, an existing LUCID program is analyzed to see if it can meet certain real-time constraints. However, the analysis cannot be done statically. This is a major goal of RLUCID.

## References

- [1] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–627, 1986.
- [2] P. Caspi, N. Halbwachs, D. Pilaud, and J. A. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 178–188, München, B.R.D., 1987.
- [3] A. Faustini and E. Lewis. Toward a real-time dataflow language. In J. Stankovic and K. Ramamrithan, editors, *Hard Real Time Systems*, pages 139–145. IEEE, 1989.
- [4] D. Skillicorn and J. Glasgow. Real Time specification using Lucid. *IEEE Transactions on Software Engineering*, 15(2):221–229, 1989.
- [5] N. S. Vempati. Programming reactive systems using dataflow. Master's thesis, University of Victoria, Victoria, B.C., Canada, 1990.
- [6] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.