Demand-driven real-time computing

Jean-Raymond Gagné
Département d’informatique, Université Laval,
Québec (Québec) Canada G1K 7P4
gagne@ift.ulaval.ca

John Plaice
School of Computer Science and Engineering,
The University of New South Wales, Sydney 2052 Australia
plaice@cse.unsw.edu.au

2000

Abstract

We present the RLucid language, which is Lucid with a synchronous time semantics, along with an extra operator, before, to allow for reactivity. We translate the index-based semantics into a timestamp-based semantics, and, by using demand-driven techniques, show how the language can be implemented efficiently.

1 Introduction

The RLucid language [4] is a timestamped extension of Lucid. [1, 5] Simply stated, RLucid is unidimensional Lucid with a timestamped semantics, along with a single new operator, before. This binary Boolean operator incorporates the essence of reactive systems: the ability to make choices according to the order of arrival of inputs.

Like in Lucid, RLucid streams are indexed by the natural numbers. In addition, each daton is timestamped by a real number. Hence, two dataflows $X$ and $Y$ can be considered to be streams of pairs $[value, timestamp]$:

$$X = ([x_0, s_0], [x_1, s_1], \ldots, [x_i, s_i], \ldots)$$

and

$$Y = ([y_0, t_0], [y_1, t_1], \ldots, [y_i, t_i], \ldots)$$

As in Lucid, data operations are applied pointwise (over the indices) to their arguments. Since RLucid is a synchronous language, the application of, say, $+$ to two streams is applied as soon as the operands are available, as in:

$$X + Y = ([x_0 + y_0, \max(s_0, t_0)], \ldots, [x_i + y_i, \max(s_i, t_i)], \ldots)$$

Constants are considered to be infinite streams that are available at “the beginning of time”, say 0. So, constant $k$ actually means

$$([k, 0], [k, 0], \ldots, [k, 0], \ldots)$$
Lucid is essentially ISWIM with two additional operators, next and fby ("followed-by"), that allow manipulation of streams. Suppose that $X$ and $Y$ are as above. Then

$$\text{next } X = ([x_1, s_1], [x_2, s_2], \ldots, [x_{i+1}, s_{i+1}], \ldots)$$

and

$$X \text{ fby } Y = ([x_0, s_0], [y_0, t_0], \ldots, [y_{i-1}, t_{i-1}], \ldots)$$

RLucid's new operator is the Boolean before: If $X$ and $Y$ are as above, then

$$X \text{ before } Y = ([s_0 \leq t_0, \min(s_0, t_0)], \ldots, [s_i \leq t_i, \min(s_i, t_i)], \ldots)$$

Section 2 will show that by using this operator in recursive functions, many interesting operators can be defined.

A couple of notes on the semantics should be made. First, the above presentation allows situations where timestamps are not totally ordered. Although there are situations where this approach is acceptable, the presentation in this paper will disallow it, as our streams have a real-time interpretation: we want to ensure that timestamps are fully ordered.

Second, it is possible to define streams that have infinite bursts of datons, all with the same timestamp. This is clearly the case for constants, but also for expressions such as:

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

which defines the flow

$$([0, 0], [1, 0], \ldots, [i, 0], \ldots)$$

Problems like these have been resolved in the chapter on RLucid that appears in the Ph.D. thesis of Jean-Raymond Gagné.[2] This article summarizes the steps undertaken to ensure that RLucid can be compiled into ordinary programming languages.

### 2 Examples

There are several heavily-used RLucid constructs that can be derived from the base language. Many of these constructs are defined as recursive functions.

- **first $X$** generates a constant stream, whose value is the first value of $X$:
  $$\text{first } X = X \text{ fby first } X$$

- **$X \text{ wvr } Y$** generates the corresponding value of $X$ every time that $Y$ is true:
  $$X \text{ wvr } Y = \begin{cases} \text{if first } Y \\ \text{then } X \text{ fby } (\text{next } X \text{ wvr next } Y) \\ \text{else } (\text{next } X \text{ wvr next } Y) \end{cases}$$

- **event $X$** generates a void (empty type) stream which is synchronous with $X$. It will be used to define other operations that do not care about the value of $X$. The || is a tuple-building data operation and select extracts the $n$-th component from a tuple.
  $$\text{event } X = \text{select}(1, \text{ void || } X)$$

Here, void || $X$ builds a tuple, and select(1, void || $X$) extracts the void value (which is the first tuple element) along with $X$’s timestamps.

- **$A \text{ on } C$** generates a stream whose values are those of $A$ but whose time-stamps are those of $C$ if $A$ is faster than $C$:
A on C = select(1, A\|\!C)

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

\[
\begin{align*}
\text{count on (void fby event A)} \\
\text{where} \\
\text{count} = 0 \text{ fby count + 1} \\
\end{align*}
\]

`count` counts the natural numbers infinitely fast. The `on` slows the counter to the rate of $A$.

- **choose X1:E1 X2:E2 ... Xn:En end** will execute the $E_i$ corresponding to the first $X_i$ which has arrived. It is defined as follows:

\[
\begin{align*}
\text{choose X1:E1 X2:E2 ... Xn:En end} = \\
\text{if first (X1 before X2) and ...} \\
\text{and first (X1 before Xn) then E1} \\
\text{else if first (X2 before X3) and ...} \\
\text{and first (X2 before Xn) then E2} \\
\text{...} \\
\text{else En}
\end{align*}
\]

- **last(A,B)** outputs the last value held by $A$ every time that $B$ occurs. It is defined as follows:

\[
\begin{align*}
\text{last(A,B) = current(A, B, nil)} \\
\text{where} \\
\text{current(A, B, C) =} \\
\text{choose} \\
\text{A: current(next A, B, first A)} \\
\text{B: C fby current(A, next B, first C)} \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{last (occurs, event B)} \\
\text{where} \\
\text{occurs} = \text{count on (void fby event A)} \\
\text{count} = 0 \text{ fby count + 1}
\end{align*}
\]

- **MergeLeft** will generate a single stream from two input streams. Should the two send values at the same time, then the left one takes precedence:

\[
\begin{align*}
\text{MergeLeft(A,B) =} \\
\text{choose} \\
\text{A: A fby MergeLeft(next A, B)} \\
\text{B: B fby MergeLeft(A, next B)}
\end{align*}
\]

- **SignalLeftFair** will generate a single stream from two input streams. However, the elements will be pairs of Booleans, showing which stream generated the pair. Only one true value will be sent at a time:
SignalLeftFair(A,B) =
choose
A||B: (true||false) fby (false||true)
      fby SignalLeftFair(next A, next B)
A: (true||false) fby SignalLeftFair(next A, B)
B: (false||true) fby SignalLeftFair(A, next B)
end

• SignalBoth will send two true values at a time, if necessary:

SignalBoth(A,B) =
choose
A||B: (true||true) fby SignalBoth(next A, next B)
A: (true||false) fby SignalBoth(next A, B)
B: (false||true) fby SignalBoth(A, next B)
end

• Finally, skip is a generalized next, where the rate of advance is not restricted to 1. It is assumed that the argument N is non-negative. Note that next X is equivalent to X skip 1 and that first X is equivalent to X skip 0.

X skip N = if first N = 0
            then X fby (X skip (next N))
            else (next X) skip ((N - 1) fby (next N))

3 Indexical semantics

In this section, we present a summary of RLUCID’s semantics. We begin by defining two auxiliary functions.

Definition 1 Let x be a Boolean indexical dataflow. Then count(x,n) is the number of true datons with a rank less than or equal to n ≥ 0 in x:

count(x,n) = # \{m : m ≤ n ∧ Fx(m) = true\}

and rank(x,n) is the rank of the n-th true daton in x:

rank(x,n) = inf \{m : count(x,m) ≥ n\} − 1

Now we can define indexical dataflows.

Definition 2 An indexical RLUCID dataflow x of type D is a pair x = (Fx, Hx), where Fx : N → D is a sequence of values, and Hx : N → N ∪ {∞} is a sequence of timestamps.

Definition 3 Let x, y, x1,...,xm be indexical RLUCID dataflows. The (original) indexical semantics of the RLUCID operators is given in Table 1.

Table 1 includes definitions for two functions, wvr and skip, which are not primitive: they are both defined in Section 2. That section presents a number of recursive functions of great utility. It turns out that all of these functions can be transformed into non-recursive functions that use the functions wvr and skip.[3]

We give an intuition for wvr and skip. Therefore, if X is a dataflow

X = ([x0,s0],[x1,s1],...,[xi,si],...)

B is a Boolean dataflow

B = ([b0,t0],[b1,t1],...,[bi,ti],...)

4
Table 1: The original semantics for RLUCID

<table>
<thead>
<tr>
<th>RLUCID operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = k )</td>
<td>( H_z n = 0 )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = k )</td>
</tr>
<tr>
<td>( z = op(x_1, \ldots, x_m) )</td>
<td>( H_z n = \max(H_{x_1}, \ldots, H_{x_m} n) )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = op(F_{x_1}, \ldots, F_{x_m} n) )</td>
</tr>
<tr>
<td>( z = next x )</td>
<td>( H_z n = H_x (n + 1) )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = F_x (n + 1) )</td>
</tr>
<tr>
<td>( z = x \ fby \ y )</td>
<td>( H_z n = \begin{cases} H_x 0 &amp; \text{if } n = 0 \ H_y (n - 1) &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = \begin{cases} F_x 0 &amp; \text{if } n = 0 \ F_y (n - 1) &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>( z = x \ before \ y )</td>
<td>( H_z n = \inf(H_x n, H_y n) )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = \begin{cases} \text{true} &amp; \text{if } H_x n \leq H_y n \ \text{false} &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>( z = x \ wvr \ y )</td>
<td>( H_z n = \max(H_x \rank(y, n), H_y \rank(y, n)) )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = F_x \rank(y, n) )</td>
</tr>
<tr>
<td>( z = x \ skip \ s )</td>
<td>( H_z n = \max(H_x n, H_x \sum_{i=0..n} F_s i) )</td>
</tr>
<tr>
<td></td>
<td>( F_z n = F_x \sum_{i=0..n} F_s i )</td>
</tr>
</tbody>
</table>

and \( N \) is a non-negative integer dataflow

\[
N = ([n_0, u_0], [n_1, u_1], \ldots, [n_i, u_i], \ldots)
\]

then

\[
\begin{align*}
X \ wvr \ B &= ([x_{\rank(b, 0)}, \max(s_{\rank(b, 0)}, t_{\rank(b, 0)})], \ldots, \\
&\quad [x_{\rank(b, i)}, \max(s_{\rank(b, i)}, t_{\rank(b, i)})], \ldots) \\
X \ skip \ N &= ([x_{n_0}, \max(s_{n_0}, u_{n_0})], \ldots, [x_{n_0 + \ldots + n_i}, \max(s_{n_0 + \ldots + n_i}, u_{n_i})], \ldots)
\end{align*}
\]

4 Bursts

Before RLUCID programs can be translated into equivalent BLIZZARD programs, some decisions must be taken with regards to the semantics of some of RLUCID’s operators.

First, we wish to be able to deal with real timestamps, i.e. we want the timestamps of datons to correspond to real time. We must therefore introduce the constraint that the timestamps of a dataflow be ordered, i.e. \( H_x n \leq H_x (n + 1) \).

Second, we have not addressed the problem of bursts, where more than one daton, possibly an infinite set, can appear at any given instant \( t \). Bursts are one of the reasons that no implementation of RLUCID has ever been made.

Bursts are produced by constants and RLUCID expressions such as

\[
X = 0 \ fby \ X + 1
\]

in this case an infinite set of integers, all sharing the same timestamp. The problem with bursts appears when they are combined with the RLUCID before operator.


As an example, consider stream \( Y \) in the expression:

\[
Y = X \text{ merge } X \\
\text{where } W \text{ merge } Z = \begin{cases} 
\text{if first } (W \text{ before } Z) \\
\text{then } W \text{ fby } ((\text{next } W) \text{ merge } Z) \\
\text{else } Z \text{ fby } (W \text{ merge } (\text{next } Z))
\end{cases}
\]

The intuition for \( Y \) is that it merges — according to the timestamps — two copies of stream \( X \), i.e. a fair semantics for \text{merge} would result in the stream \((0, 0, 1, 1, 2, 2, ...)\) for \( Y \): datons would be alternately taken from each of the two copies of \( X \). However, since in RLUCID there is no way to distinguish two datons in a burst through their timestamps, stream \( Y \) is in fact the sequence \((0, 1, 2, 3, ...)\). In the translation to BLIZZARD given below, we will give a subtly different semantics to RLUCID that will ensure a fair semantics for \text{merge}.

The semantics for RLUCID’s \text{fby} operator must also be addressed. In the expression \( Z = X \text{ fby } Y \), the first daton of \( Z \) is the first daton of \( X \), and the subsequent datons are those of \( Y \) shifted from the present to the future by one place in \( Y \)’s timestamp sequence. A question arises: What happens to the \( Y \) datons that occur before the occurrence of \( X \)’s first daton?

There are two possibilities: either they are lost (and do not appear in the resulting stream) or they are memorized until they are output in a burst. Given the semantics of RLUCID, it would be inappropriate to lose the datons, so some method must be found to deal with the potential resulting bursts. This could be handled by building queues corresponding to sets of unused datons on a given line, however this would lead to unbounded memory situations, which we want to avoid.

A solution to the burst problem appears if we use the non-standard domain for the timestamps. We can then redefine the notion of constant as well as the semantics of the \text{fby} operator.

**Definition 4** A burst-free indexical RLUCID dataflow \( x \) of type \( D \) is a pair \( x = (F, H) \), where \( F : \mathbb{N} \rightarrow D \) is a sequence of values, and \( H : \mathbb{N} \rightarrow \mathbb{R} \times \mathbb{Z} \cup \{\infty\} \) is a sequence of timestamps.

**Definition 5** Let \( x, y, s, b, x_1, \ldots, x_m \) be burst-free indexical RLUCID dataflows. The burst-free indexical semantics of the RLUCID operators is given in Table 2.

Constants generate datons with timestamps from successive micro-instants within the same macro-instant.

The operator \text{fby} can no longer generate bursts, as it takes into account the timestamp of the previously generated daton \( H_x(n - 1) + (0, 1) \) while yielding a new one.

Note that we have replaced the \text{next} operator by the \text{skip} operator.

## 5 Timestamped semantics

We present here a semantics for RLUCID using only timestamps and prove the equivalence with the previous semantics.

**Definition 6** For each RLUCID dataflow \( x_R = (H, F) \), define a BLIZZARD dataflow \( x_B = (C, V) \):

\[
C_x = \{ H_x n : n \in \mathbb{N} \} \\
V_x t = F_x(I_x t) \\
I_x t = #\{ t' \in C_x : t' \leq t \} - 1
\]

In order to define the semantics of the RLUCID operators, we will first define an auxiliary function.
Table 2: Burst-free semantics of RLucid

<table>
<thead>
<tr>
<th>RLucid operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$z = k$</td>
<td>$H_z n = (0, n)$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = k$</td>
</tr>
<tr>
<td>$z = \text{op}(x_1, \ldots, x_m)$</td>
<td>$H_z n = \max(H_{x_1 n}, \ldots, H_{x_m n})$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = \text{op}(F_{x_1 n}, \ldots, F_{x_m n})$</td>
</tr>
<tr>
<td>$z = x \text{skip } s$</td>
<td>$H_z n = \max(H_{s n}, H_{x(n+\sum_{i=0}^{n} F_s i)})$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = F_x(\sum_{i=0}^{n} F_s i)$</td>
</tr>
<tr>
<td>$z = x \text{fby } y$</td>
<td>$H_z n = \begin{cases} H_x 0 &amp; \text{if } n = 0 \ \max(H_y(n-1), H_z(n-1) + (0, 1)) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = \begin{cases} F_x 0 &amp; \text{if } n = 0 \ F_y(n-1) &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$z = x \text{before } y$</td>
<td>$H_z n = \inf(H_{x n}, H_{y n})$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = \begin{cases} \text{true} &amp; \text{if } H_{x n} \leq H_{y n} \ \text{false} &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>$z = x \text{wvr } b$</td>
<td>$H_z n = \max(H_x(\text{rank}(b, n)), H_b(\text{rank}(b, n)))$</td>
</tr>
<tr>
<td></td>
<td>$F_z n = F_x(\text{rank}(b, n))$</td>
</tr>
</tbody>
</table>

**Definition 7** The index function $R$ computes the sum of the data whose timestamps are less than the argument $t$.

$$R_x t = \sum_{t' \in C_x, t' \leq t} V_x t'$$

We can now define the timestamp-based semantics for the operators.

**Definition 8** Let $x, y, b, s, x_1, \ldots, x_m$ be timestamped RLucid dataflows. The timestamped semantics of the RLucid operators is given in Table 3. We write $C_x n$ for the $n$-th occurrence of clock $C_x$.

Now we can prove the equivalence of the two definitions.

**Proposition 9** Let $x_1, \ldots, x_m$ be burst-free RLucid indexical dataflows. Let $p$ be a RLucid operator, where $z = p(x_1, \ldots, x_m)$ is defined in Definition 5. We rewrite the equivalent semantics in Definition 8 as $z = p(x_1, \ldots, x_m)$. If for all $i = 1..m$, $x_i = x_{iBR}$, then $z = z_{BR}$.

**Proof** We must show, for each $p$, that $\{H_z n : n \in \mathbb{N}\} = C_z$ and also that $F_z(I_z t) = V_z t$. The details can be found in [2], pages 118–23.

**6 Infinite accumulation**

The semantics of Table 3 cannot be implemented since

- constants continue to generate an infinite set of datons in the first macro-instant;
- the functions $I$ and $R$ are used in expressions such as $V_x(C_x(I_z t))$, thereby accessing a timestamp in a clock $C_x$, which is in turn used to access a value in $V_x$. In general, this cannot be done without having to memorize an unbounded set of datons.
Table 3: Timestamped semantics for RLucid

<table>
<thead>
<tr>
<th>RLucid operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>( z = k )</td>
<td>( C_z = {0} \times \mathbb{N} )</td>
</tr>
<tr>
<td>( z = op(x_1, \ldots, x_m) )</td>
<td>( C_z = { t : (\exists i \leq m : t \in C_{x_i} ) ) ( \land (\forall j \leq m : I_{x_j} t \leq I_{x_i} t) }) )</td>
</tr>
<tr>
<td>( z = x \text{ skip } s )</td>
<td>( C_z = { t : t \in C_s \land \mathcal{R}_s t \leq I_s t ) ( \lor t \in C_s \land \mathcal{R}_s t \leq I_s t } )</td>
</tr>
<tr>
<td>( z = x \text{ fby } y )</td>
<td>( C_z = { t : t = C_y 0 \lor t &gt; C_y 0 \land (t \in C_y \land \pi_r(t) = \pi_r(C_y 0) \land \pi_z(t - C_y 0) - 1) } )</td>
</tr>
<tr>
<td>( z = x \text{ before } y )</td>
<td>( C_z = { t : t \in C_z \land \mathcal{I}_y t \leq I_y t ) ( \lor t \in C_y \land \mathcal{I}_y t \leq I_y t } )</td>
</tr>
<tr>
<td>( z = x \text{ wvr } b )</td>
<td>( C_z = { t : t \in C_b \land \mathcal{V}_b t = \text{true} \land \mathcal{I}_b t \leq I_b t ) ( \lor t \in C_y \land \mathcal{I}_y t \leq I_y t \land ) ( \mathcal{V}_b(C_y(I_z t)) = \text{true} } )</td>
</tr>
</tbody>
</table>

We can simplify the above semantics by using constraints on clocks, in effect restraining them so that datons are generated only when they are needed. In fact, datons may be retained so long as their relative arrival order is kept unchanged, so that the \( \text{before} \) operator yields the same results. A clock \( C \) may be transformed into a retarded clock \( C' \) if we change the function \( \mathcal{V} \) into a new function \( \mathcal{V}' \) so that:

\[
\forall n \in \mathbb{N} : \mathcal{V}'(C' n) = \mathcal{V}(C n)
\]

Table 4 summarizes the new semantics with constraints on clocks. Note that if clock retarding is used as needed, then the same (denotational) dataflow might have to be replicated several times into several (operational) dataflows, for different uses at different rates (on different clocks).

**Proposition 10** The replication of one (denotational) dataflow into several (operational) dataflows need only be done a finitely many times.

**Proof** See [2], pages 126–7.

Let us note that the system’s inputs must be memorized if they are to be used at different speeds. This is a limitation of the current semantics. It can duplicate its internal dataflows but not its inputs. So memorization of the inputs could lead to unbounded memory situations.

Note that we still have to deal with some kind of infinity, since the semantics of the \( \text{before} \) operator makes use of \( \mathcal{I} \) on two different dataflows \( x \) and \( y \), and the difference between \( \mathcal{I}_x t \) and \( \mathcal{I}_y t \) could potentially be unbounded.

The introduction of new temporal constraints into the definitions of the operators in Table 3 yields new definitions for these operators. We call the new semantics the **clocked** semantics.
Table 4: Clocked semantics for RLucid

<table>
<thead>
<tr>
<th>RLucid operator</th>
<th>Semantics</th>
</tr>
</thead>
</table>
| z = k           | $z \subseteq T_3$  
                | $\forall z_t = k$ |
| $z = op(x_1, \ldots, x_m)$ | $C_z = C_{x_z}$ with constraint $\forall i \in 1..m: C_z = C_{x_i}$  
                | $\forall z_t = op(\forall x_1 t, \ldots, \forall x_m t)$ |
| $z = x \text{ skip s}$ | $C_z = \{ t : t \in C_x \land I_xt = R_s t \}$ with constraint $\forall n \in N : C_z n \leq C_x (R_s (C_z n)) < C_z (n + 1)$  
                | $\forall z_t = \forall z_t$ |
| $z = x \text{ fby y}$ | $C_z = \{ t : t = C_x 0 \lor t \in C_y \}$ with constraint $C_y 0 > C_x 0$  
                | $\forall z_t = \{ \begin{cases} \forall z_t & \text{if } t = C_x 0 \\ \forall z_t & \text{otherwise} \end{cases}$ |
| $z = x \text{ before y}$ | $C_z = \{ t : t \in C_x \land I_xt \geq I_y t \lor t \in C_y \land I_y t \geq I_xt \}$  
                | $\forall z_t = \{ \begin{cases} \text{true} & \text{if } I_xt \geq I_y t \\ \text{false} & \text{otherwise} \end{cases}$ |
| $z = x \text{ wvr b}$ | $C_z = \{ t \in C_x : \forall b t = \text{true} \}$ with constraint $C_x = C_b$  
                | $\forall z_t = \forall z_t$ |

Definition 11 Let $x$, $y$, $b$, $s$, $x_1, \ldots, x_m$ be timestamped RLucid dataflows. The clocked semantics of the RLucid operators is given in Table 4.

Proposition 12 Let $x_1, \ldots, x_m$ be burst-free RLucid indexical dataflows. Let $p$ be a RLucid operator, where $z = p(x_1, \ldots, x_m)$ is defined in Definition 8 We rewrite the equivalent semantics in Definition 11 as $z = p(x_1, \ldots, x_m)$. If the clock constraints on the $x_i$ in Definition 11 are respected, then $z = z$.

Proof The only non-trivial new derivations of clocks are for fby and skip. The details can be found in [2], pages 126–7.

7 Chinook implementation

In Gagné’s thesis,[2] it is shown that the semantics given in Table 4 can be translated into the Chinook formalism. Chinook is a formalism in which, for each operator, one must express the functional dependencies between inputs and outputs; the clock synchronizations between variables in the system; and the functional behavior of the different operators. Chinook systems can be easily translated into any standard programming language.

8 Conclusion

We have resolved several difficulties in the RLucid language. First, RLucid allows recursively defined functions. We resolved this problem by restricting ourselves to tail-recursive functions and by giving a translation process in which RLucid tail-recursive programs can be translated into RLucid iterative programs. This allowed us to deal with the two remaining problems.
The second problem was the presence of bursts. It turned out that this can be resolved by slightly changing RLUCID’s semantics. The time domain of BLIZZARD gives to each element of a burst a distinct timestamp, corresponding to a different micro-instant within the same macro-instant: it is thus possible to do a finer analysis of a macro-instant. In general, BLIZZARD allows a better understanding of instantaneous interaction.

The last problem was related to the use of the same dataflows at different rates. This leads, in the worst case, to unbounded-memory RLUCID programs.

It turned out that this can be solved using the demand-driven model, assuming that inputs are not unsynchronized in an unbounded manner. With this approach, we need to instantiate each (denotational) dataflow for each different rates of its use. In the CHINOOK implementation of RLUCID one can see that clock demand signals are not simply passed to clock suppliers, but are indeed the subject of a more subtle approach. All this has allowed us to give the first working implementation of RLUCID.

References


