Programming Reactive Systems using Dataflow

by

Naganjaneya Sarma Vempati

M.Sc., Andhra University, Waltair, India, 1975
Ph.D., Indian Institute of Technology, Knapur, India, 1981

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE 1990-07-26

We accept this thesis as conforming
to the required standard

[redacted]
Dr. John A. Plaice

[redacted]
Dr. William W. Wadge

[redacted]
Dr. Gary G. Miller

[redacted]
Dr. Hubert Garavel

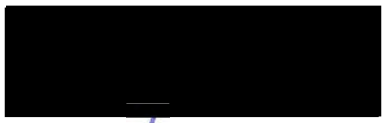
Supervisor: Dr. John A. Plaice

Abstract

RLUCID, presented in this thesis as a programming language for real-time applications, is an extension of LUCID, a dataflow language. The notion of time enters the language through the *timestamped streams*: LUCID streams of values together with a corresponding stream of timestamps. Two additional operators, `Which` and || (read 'synchro') are introduced to exploit the timing information without developing uncausal relations.

RLUCID is sufficiently expressive. It is employed in providing an alternative semantics of LUSTRE, a real-time dataflow language, more intuitively.

Further, a solution to an important problem of defining an interface to a LUSTRE program, is presented. Traditionally written in 'C', this interface maps the inherently asynchronous environment to the synchronous kernel generated from a LUSTRE program. Proposed here is a new language, APRIL, to concisely define such interfaces in a synchronous manner. The primitive operators of APRIL are defined by RLUCID functions.



Dr. John A. Plaice

Dr. William W. Wadge

Dr. Gary G. Miller

Dr. Hubert Garavel

Preface

It is not an overstatement that advances in computer technology are revolutionizing our lives. There is a computer, it seems, playing a part in every story. The speed and scope, the familiar objects gain by using computers, is at times overwhelming.

Computer controlled applications surround us. They are improving the quality of our life and free us from drudgery. A television set can be turned on or off without getting up from the chair; a video recorder can be programmed in advance to record events in advance.

The fact that they are working as we want them is taken as a sufficient guarantee of their reliability. When an automatic pilot crashes the plane or a train derails, due to an error on the part of the controlling computer, and lives are lost, do we blame ourselves for not having having checked thoroughly the computer programs. However, we should remember that 'to err is human'; excessive self criticism does not help.

But, if we could detect the reason for such failures in advance, we could perhaps avoid the mishaps. A great many of these are attributed to slovenliness in the production languages.

The computer scientists are neither the first, nor alone, in their efforts in trying to clean up languages. The linguistic philosophers, particularly the logical

positivists, [1] have been active since the turn of the century.

According to Rudolph Carnap, if one could invent a formalized language which is so constructed that unverifiable statements cannot be formulated within it, the adoption of such a language would meet all the requirements of the positivists. And so does it for us.

The design philosophy of languages like LUCID, LUSTRE and ESTEREL certainly concurs with Carnap's view. Since the meaning of all statements is *a priori* known, there is no danger of the unexpected. If the world were to be destroyed by a computer (program), it certainly would know its doom.

This thesis is concerned with the development of formal real-time languages, RLUCID, LUSTRE and APRIL. The last is an interface language, aimed at programming interfaces to other computer programs.

Acknowledgements

In doing this work, I have received help from many friends.

I am grateful to John Plaice, for introducing me to this interesting and important field of study; also for patiently going through all the revisions and effecting a great improvement in contents and quality of the thesis.

I would like to thank Bill Wadge, for introducing me to the dataflow programming, and for the many suggestions during the course of the work.

My thanks to Du and Tao, in building a subset of RLUCID as a class project

[1] Logical positivism is based on: 1) there is nothing we can know beyond experience; 2) the meaning of a proposition is its method of verification, also known as the principle of verifiability. For a popular account, see [24].

using POPSHOP'; to Mehmet and Du, for suggestions on the initial drafts of the thesis; and to Anand, for his interest in the work.

I am indebted to Dr. Hubert Garavel, VERILOG Company, Grenoble, France, for his invaluable suggestions in improving the thesis.

I would like to thank Mike Levy for his support and help as a (good) graduate advisor.

Help from the friendly staff of the department of Computer Science, is acknowledged.

Constant support and encouragement from my daughter Bhanu Sakti and my wife, Kamala, during my studies and in the preparation of the thesis is much appreciated.

I take pleasure in thanking all those who have directly or indirectly helped me in smaller or greater measures and apologize for not mentioning them by name.

June, 1990.
Victoria, B.C. Sarma Vempati.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the decrease in the cost of hardware, computers are being used in an ever increasing number of applications, ranging from domestic appliances to automatic pilots, communications, industrial processes and nuclear reactors.

Advances in high-level programming languages enable one to effectively produce the applications programs in all but the most time-critical systems.

Therefore, a short introduction to these (reactive) applications is given. Since the main thrust of the thesis is on the programming languages for such computer systems, various issues concerning such languages are then presented. The chapter concludes with an overview of the thesis.

1.1 Reactive Systems

The computer systems used in these applications all interact *continually* with their environments. These are called *reactive systems*, as they react to the input stimuli provided by their environments. The *reactive program* is the brain of a reactive system controlling or coordinating various operations in the system.

Reactive programs produce no spontaneous output; output is generated *only*

in response to, and at the pace set, by external stimuli. Further, not all input produces output. Programs are expected to run until they are stopped or until a fatal error occurs. They take a finite time to generate proper responses.

In reactive applications, a reactive program is considered to have failed if it ceases to perform the activity for which it is designed. Operating systems, process control systems, air traffic control systems, to name a few, literally run continuously.

Given the nature of some of the applications, a fatal error could have dire consequences. To guarantee proper functioning, the correctness of the reactive program must be proven. If such a proof cannot be furnished, then the designers must determine: how the program may fail; what precautions should be taken to avoid a possible failure; what is the likelihood of such a failure; and what measures will be taken in the event of such failure.[1]

We study an important subclass of reactive systems, e.g., control systems. There are other reactive systems, e.g., operating systems, communication protocols etc. which we are not addressed in this thesis.

The basic components that constitute such reactive systems are: 1) sensors, 2) input interface 3) computer 4) output interface and 5) actuators.[2] Figure 1.1 shows the connection of the various components and the progression of signals through the system.

Signals provided by sensors, after suitable treatment, are converted into computer compatible codes in an input interface. Then they are sent to the

[1]Proper software and hardware does not guarantee safety and security; many failures of existing systems are due to human error (e.g., Chernobyl) or to unexpected malfunctioning of mechanical, hydraulic, etc., components.

[2]A sensor is a device for converting a physical signal to electrical signal. An actuator converts the electrical signal to mechanical movement.

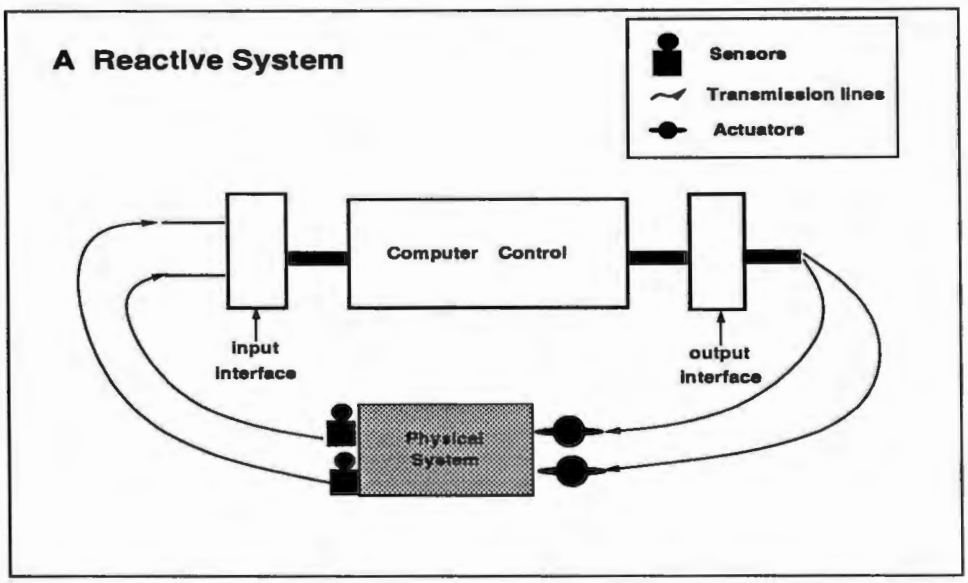


Figure 1.1: Structure of a reactive system.

computer.[3] The computer, acting on instructions contained in its program, processes the input data and generates output data which are converted in an output interface to suitable signals. In case they are being used to drive the actuators, they may further be subjected to treatment in signal conditioners.

The activities performed by the system shown in Figure 1.1 may also be understood as taking place in a simple network shown in Figure 1.2. Sensors are connected to the computer through channels (transmission lines, buses, wires or cables) and carry information to the controlling computer. Reactions (output) generated by the controlling computer are similarly conveyed on the output channels to the actuators.

[3]Signals, provided by the sensors such as potentiometers, are often treated before their entering the computer interfaces in signal conditioners. Signal conditioners suitably treat the signals to match them to the requirements of the interfaces. Similar conditioners may also be required between output interfaces and actuators.

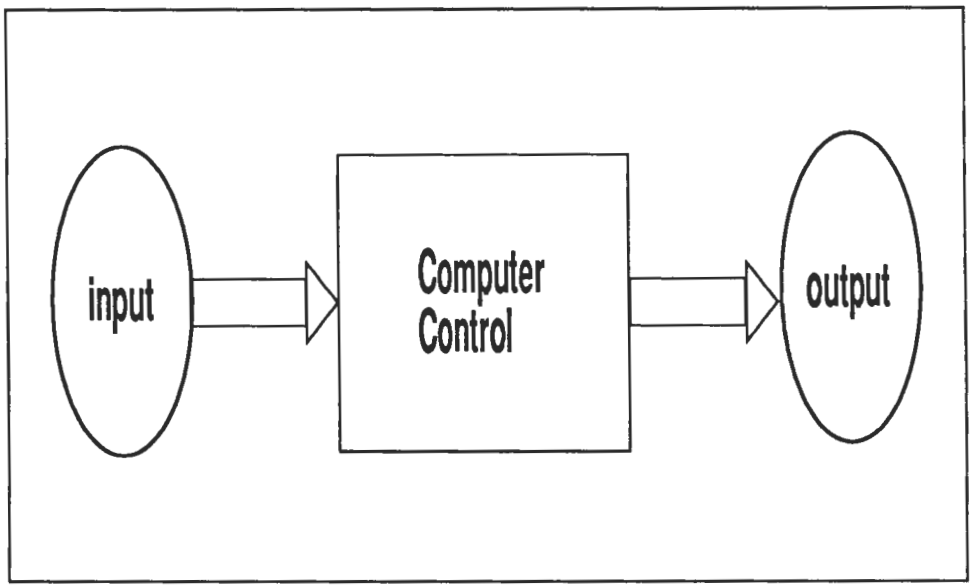


Figure 1.2: The real-time system as a network

1.2 Real-Time Systems

Reactive systems include *real-time systems*, whose reactions are subject to timing constraints — such as sampling frequency and response time — dictated by their environment.

The correctness of a conventional computer system is entirely determined by the result, if it occurs. On the other hand, for real-time systems, the time of response is as important as the response itself. That is, the program's correctness not only depends on *what* it does but also *when* it does it.

Timing constraints are sometimes created by the necessity of avoiding catastrophic situations. For example, it might be determined empirically that if a reactor has not been shut off within 5 seconds after reaching the critical temperature, the reactor will melt.

Timing constraints do exist in day-to-day life. A traveler taking a train at 10 A.M. must be at the station before 10 A.M. A person intending to answer a

telephone must pick it up before it stops ringing. To observe a lunar or a solar eclipse one must be at the right place at the right time. In a data acquisition environment, sensor readings must be recorded before new values arrive, or data will be lost. Thus timing constraints must be satisfied to achieve the desired goal or the proper behavior of a system.

Timing constraints are also found in the interaction between the environment and the reactive system. The state of the environment will change with time. Ideally, the reactive program would perceive the environment exactly as it is. Thus it is necessary that the stimuli sent by the environment are processed *expeditiously.* However, slight differences between the two may exist as long as the proper functioning of the system is not jeopardized. The timing constraints express these tolerance limits.

Two trains heading towards each other on the same track may end up in a collision, if the computer controlling the switches does not respond in time to the signals from the track. The designer might introduce timing constraints stating that those signals must be responded to within a certain time.

For example, should a railway switch not be set, because of a failure to process the signal announcing an arriving train, that train may end up in a collision. For instance, if two sensors exist, providing two signals announcing the train, one of the two may be missed, but not both. Or, if the computer fails to instruct the robot on the assembly line (to stop or turn) in time, the robot might collide with another object.

1.3 Comparison between Reactive and Real-Time Systems

In real-time systems, time-critical tasks (with hard timing constraints) coexist with those which are not so time-critical. Ideally, the computer should execute tasks in such a fashion that the timing constraints are met for the time-critical tasks, and the average response time is minimum for the others. An automatic banking machine is a good example of these mixed constraints. Accepting the keyed-in requests or the deposits occurs in real time. The updates to accounts and other transactions (when "wait, your request is in progress" is seen on the screen) are handled in a timesharing manner. For these the computer strives to provide a good response and to minimize the waiting. The need to meet the timing constraints makes the design of real-time systems a challenging problem.

Reactive systems may be classified according to the speed at which they are required to generate responses. Systems with response times at the low-speed end merge into conventional systems and are sometimes called soft reactive systems. Here a result must eventually be produced. At the high-speed end, the systems merge into hard wired electronics as the software is simply not fast enough. Programming at both ends is simple: either satisfying the constraints is trivial or there is no time for processing anything, and hence the timing constraints are trivially satisfied. Between the two ends lie a large class of complex problems which many of the real-time programming languages are designed to address.

1.4 Languages for Real-Time Applications

Despite the evolution of reactive applications from the use of analog machines and relay circuits to the use of microprocessors and computers, current pro-

gramming techniques are often low-level and specific. Often, such applications are developed using assembly language programming or by hand coding of automata. However, there is optimism that the practices change quickly and that high-level languages which are formal, portable and machine independent will become available for these applications.

A language suitable for programming reactive systems should provide constructs for defining relations between inputs and outputs, and for expressing timing constraints. In addition, it should provide facilities required of any good programming language such as modularity and error handling.

1.4.1 Extended Sequential Languages

A common practice is to extend the framework of sequential languages with primitives to deal with concurrency and time. Languages such as Ada, Occam, Modula, and Concurrent Pascal provide specific mechanisms — signals, delays, watchdogs, semaphores, rendezvous, etc. — for concurrency and timing in the programs.

Although these languages provide features for good software development — in terms of readability, maintainability, portability and efficiency — they *lack formal semantics*. Temporal statements do not have well defined semantics and depend on the implementation.

The programs using signals and semaphores have been shown in practice to be insecure and error prone [27]. By avoiding constructs dependent on execution time, one gains the advantages of deriving the program's validity statically.

Some of these issues in the context of reactive programming are examined in the following.

Most problems in reactive programming in imperative languages have been

attributed to asynchronous communication, and non-determinism in timing constructs and in parallel composition.

In Ada, time is interpreted in terms of the computer clock corresponding to the notion of a global clock. Because of the asynchronous character of the communications, delays and watchdogs may have trouble in correctly perceiving the time (i.e., on the `calendar` package and the underlying scheduler). Consider the following example:

```
task countMinutes;

task body countMinutes is
begin
   loop
      delay 60.0;
      clock.aMinute;
   end loop;
end countMinutes;


task clock;
   entry aMinute;
end clock;

task body clock is
begin
   . . .
   . . .
```

```
        accept aMinute do
           minute := minute + 1;
           . . .
        end aMinute
        . . .
    end clock;
```

where aMinute is an entry of clock.

The actual meaning of the program is not what it appears to be. Since the communication is asynchronous, there is no systematic way of determining the duration of any statement. For the task clock to receive aMinute, three events must occur simultaneously: exactly 60 seconds have been counted by countMinutes; clock must be actively listening to countMinutes; and the rendezvous must be completed. The time taken by any of these steps is not exactly predictable. If aMinute is also to be sent to another task digitalWatch, countMinutes must call the entry digitalWatch.aMinute. Since aMinute is not broadcast but sent by rendezvous only one thing is sure: that is clock and digitalWatch.aMinute never receive aMinute at the same time and hence never show the same time. In an asynchronous language like Ada, real time constraints can never be ensured.

For details on further problems in communication, synchronization (rendezvous), exception handling and priorities and in handling them within a comprehensive semantics, see [5] or [3].

Thus it is clear that the current practices of developing software for reactive systems could benefit from better methods of expressibility and reliability.

1.4.2 Languages Based on Finite Automata

Reactive systems have been successfully modelled after finite state machines using asynchronous languages such as ESTELLE[15]. The system's response is completely determined by its current state and the input to the system. Transitions are triggered on the arrival of stimuli.

In extending to real-time systems, a finite state machine is supplemented with a timer feature[8]. A timer is started with an assigned time as soon as a transition is triggered. Either a stimulus is received in the new state, or the alarm signal goes off and default response is generated. Thus, timing constraints are enforced by the timer alarm, to dictate the behavior of the reactive system.

As an example, let us suppose that a maximum of 30 seconds be allowed between dialing two digits on a telephone. As soon as the first digit is dialed, the FSM enters a new state simultaneously starting a timer with 30 seconds. Either the second digit is dialed (a corresponding signal is received) or the alarm signal is received; in the later case the dial tone is suspended.

This formalism has all the advantages of finite state automata: clear semantics and established verification methods.

Other languages based on finite state machines have been proposed in a more high-level and structured manner. This is the notion of *process algebras* such as CSP[13], CCS[22], LOTOS[14]. These are asynchronous languages and we do not discuss them any more, since we are mainly interested in a different class of synchronous languages.

1.5 Synchronous Hypothesis

A different approach can also be adopted to specify real-time. Real-time systems are programmed under the assumption of *synchrony*.

A large class of reactive systems can be programmed with the assumption that *no new input events occur until the current reaction is completed.* As hardware gets faster and compilation techniques advance, the class of problems that can be programmed under this assumption only expands.

If the maximum reaction time is shorter than the minimum time interval between two external (input) events, a reactive system can be wholly abstracted from the implementation through the *synchronous hypothesis*: `it reacts` *instantaneously* `to its external stimuli`. Equivalently, the computers which control the applications are infinitely fast or, reactions do not take time.

1.6 Dataflow Model

Thus the successive readings or values sent by a sensor on a channel(*cf.* §1.1) form a sequence in course of time. Or one can visualize an endless stream of values communicated on the input channels, arcs labeled *input*, by the sensors.

If a variable represents a sensor (in the reactive program), the sequence of values assigned to it forms its history. These values pass through and are transformed by the 'computer program' or 'black box' representing it. They are sent to the outside world on the line labeled *output.* Similarly, one can imagine an endless stream of output values and a history of the output variables (actuators). The boxes through which data flow and by which they are transformed are sometimes called 'nodes' or 'filters' in analogy with electronics.[4] Filters represent not

[4]Individual values of data are referred to as *datons*, continuing in the same fashion.

only the whole programs, but also the subunits of a program. In the latter case, a program consists of a number of filters connected together as specified by the program.

The behavior of the system is then described by a set of functional and temporal relations between its input and output sequences. We can thus model the system as a history transformer. Dataflow, based on the general principle of processing the data while they are in motion, seems to be natural to model the reactive systems, from the points of view of both architecture and mathematical model.

The data pass through the filters one by one. An input or output variable, in its history, assumes the most recent value sent by the environment (from a sensor) or the most recent transformation affected by the computer, respectively. In course of its lifetime, a reactive program accepts a sequence of sensor readings or other inputs

$$< s_0,\ s_1,\ s_2,\ \ldots >$$

This forms its total input history. Similarly, a whole sequence,

$$< o_0,\ o_1,\ o_2,\ \ldots >$$

forms the total output of the filter which is the output history of the filter. Thus we can think of the filter as performing a mathematical function. It takes elements one by one from the sequence

$$< s_0,\ s_1,\ s_2,\ \ldots >$$

as its argument and returns the entire sequence

$$< o_0,\ o_1,\ o_2,\ \ldots >$$

one by one, in that order, as its result.

The input to a program that runs indefinitely is the history of the values 'fed into' it; the output is the history of all values produced.

The development of dataflow languages, in the remainder of the thesis, is based on a particular formalism: the functional model of Caspi and Halbwachs [6]. In this model, a sequence of values assigned to variables, together with the times at which these assignments take place, represent histories of variables. Further, as shown by them some simple mathematical tools such as precedence relation, time translation, subsequencing and filtering, then enable one completely to define the temporal relationships between those histories.

An *event* may be understood as a state transition in a system or in its environment. Assignment of a value to a variable is an instance of an event. Reception of stimuli and emission of signals are also examples of events. The same variable may be assigned a value more than once during its life time. Each of these assignments is then considered as an *occurrence* of the event 'assignment to that variable'. Since our main interest is in handling histories of variables, we characterize an event e by an increasing sequence of its *occurrence dates* (increasing time sequences), that is, with the occurrences numbered in chronological order. Then the evolution of a variable is described by the sequence of values taken by it, and an event by the sequence of assignments of values to the variable. The event e that the variable `X` takes the sequence of values

$$< x_1,\ x_2,\ x_3,\ \ldots >$$

is characterized by the sequence of dates

$$< t_1,\ t_2,\ t_3,\ \ldots >$$

where t_i is the date when `X` takes the value x_i.

1.7 Outline

Specifying and programming reactive systems in LUCID with minor changes is the subject of discussion in the remainder of the thesis.

An overview of the language LUCID along with a flavor for writing programs in it, is presented in Chapter 2. Ordinary streams of LUCID are replaced by *timestamped streams* as fundamental objects in a new language RLUCID.[5] Timestamps, consistent with the model introduced in the previous section §1.6, are added to all datons in RLUCID to provide timing information. An alternative approach using hiatonic streams was outlined by Wadge and Ashcroft [26]. Two new operators are also introduced to suit the language for programming reactive systems. The reactive systems are programmed under the assumption of synchronous hypothesis, which may be restated in moderate terms as: *no response takes longer than the least interval between two input events.* The fear that LUCID may 'inherently' be unsuitable for real-time systems will be shown to be unfounded by giving examples of reactive systems in RLUCID. The limitations and the suggestions for improvement are also discussed.

In Chapter 3, a synchronous dataflow language LUSTRE, based on the notion of abstract clocks, is introduced. The method of generating efficient code by placing certain restrictions on the form and availability of input is particularly noteworthy. In addition to the synchronous hypothesis, *all input to a* LUSTRE *node is assumed to be available at the same time.* The latter condition is the result of placing all operands of a node on the same (LUSTRE) clock. If this condition is not satisfied, a clock inconsistency error occurs. This has an important consequence that only the programs which satisfy the 'clock consistency rules',

[5] The short form for 'real-time LUCID. If the word *reactive dataflow* will be coined to model the reactive systems, then RLUCID is hoped to be the language for reactive flow language.

will have meaning in LUSTRE.

The semantics of LUSTRE are given in RLUCID to show the expressive power and sufficiency of RLUCID in describing the reactive systems.

It is very unlikely that input as demanded by LUSTRE programs occurs in the synchronous form. This is accomplished by an interface written in 'C' or such other imperative languages. RLUCID can also be used to develop these interfaces. But, as shown in Chapter 4, the dull tedium cannot be escaped.

APRIL, presented in Chapter 4, is a special purpose language designed for programming interfaces such as the one required for LUSTRE programs. The interface specifications, in APRIL, are refreshingly concise and clear. Its operational semantics given in RLUCID show once again its power to express the complex interfaces concisely and clearly. Were the present LUCID (RLUCID) compiler fast, APRIL could have been implemented in RLUCID.

Further, a special purpose language such as APRIL could be specified and written in RLUCID. Its clear, formal semantics enable one to reason about programs with efficiency and efficacy. Thus we see RLUCID *as the way of programming real-time systems* in a functional dataflow manner.

Chapter 2

Programming in rLucid

In recent years, a number of formalisms based on the synchronous hypothesis have been developed for reactive systems. These include the imperative language ESTEREL [4], the dataflow languages LUSTRE [7, 23] and SIGNAL [19], and a hierarchical description of automata, STATECHARTS [12].

The model of Caspi and Halbwachs, presented in §1.6 can be used to describe a large class of reactive systems. The possibility of using LUCID as a basis for a programming language for reactive systems is investigated. To this end, we introduce the notion of timestamped streams and extend the language with two additional primitives.

2.1 Dataflow Networks

LUCID is a functional dataflow language; LUCID programs consist of sets of equations defining streams and filters, and identities between streams. The order of appearance of definitions or equations has no effect on the computed result.

Every program written in LUCID defines a dataflow network. For each node in the network the entire output history is a function of the entire input history.

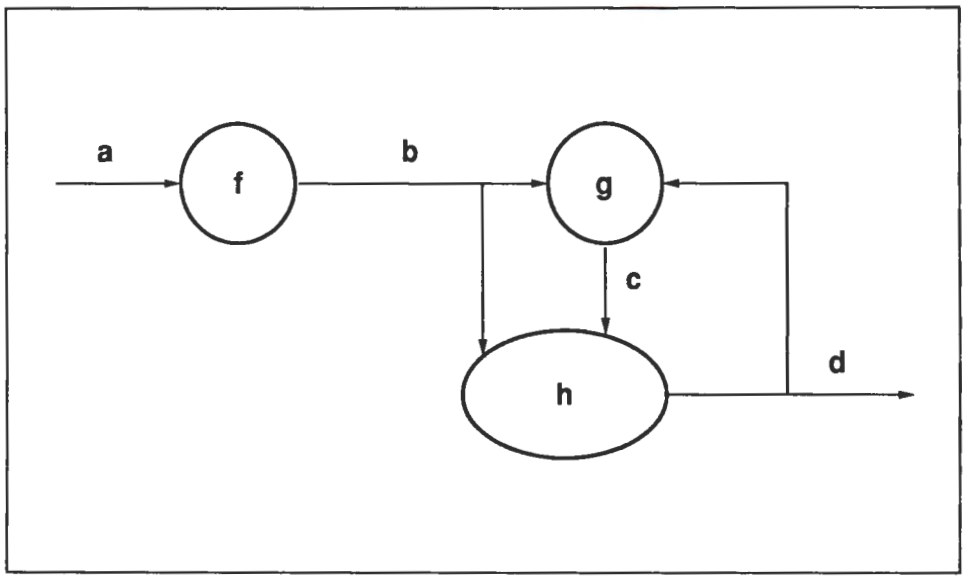


Figure 2.1: A network with three filters, `f`, `g`, `h`.

In pure pipeline dataflow, the behavior of the network is exactly described by the *least fixed point* of the corresponding LUCID program. Such a relationship was first exhibited by Kahn [16].

The arcs of a network correspond to variables in a program and processing nodes (filters) correspond to operations or functions. Only one output line is allowed for each node in the network. T-junctions indicate copy nodes: datons flowing through are duplicated, and one copy goes to each output. The network shown in Figure 2.1 corresponds to the following set of equations:

```
b = f(a);
c = g(b,d);
d = h(b,c);
```

Each of the variables `b`, `c` and `d` in the above network corresponds to the output of nodes `f`, `g` and `h` respectively. Feedback loops in the network correspond to recursively defined variables.

There is a one-to-one correspondence between the dataflow model envisaged for reactive systems and the dataflow model of computation in LUCID. Infinitary programming in LUCID is consistent with the notion of nonterminating reactive programs. Further, unending histories of input and output from nodes correspond to values continuously arriving from sensors (input) and signals continuously transmitted to actuators (reactions). Simple and formal semantics, referential transparency and a functional style of computation provide a very good reason to consider LUCID for reactive systems programming.

2.2 Lucid

RLUCID is a functional dataflow language whose syntax is identical to LUCID's, except for one extension. We introduce LUCID briefly, referring the reader to the book, *Lucid, the dataflow programming language* [26], for further details.

A program is an *expression* and the evaluation of the expression constitutes its execution. The following simple expression:

```
X + Y
```

is a complete program in LUCID.

2.2.1 Streams and Operators

The fundamental objects of LUCID are *streams*. A stream is a (possibly) infinite sequence of values; it is denoted by:

$$< v_0, v_1, v_2, \ldots >$$

A *variable* is a stream. It is defined by exactly one *equation*. A free variable is not defined by any equation and its values are assumed to be input as required.

An equation `X=E` defines the variable `X`, giving it the same value as the expression `E`.

A *constant* in an expression defines an infinite sequence of values of the constant. The constant `1` is the sequence:

$$< 1, 1, 1, \ldots >$$

The prefix operator `first` produces the constant stream defined by the equation:

$$\texttt{first}(< x_0, x_1, x_2, \ldots >) = < x_0, x_0, x_0, \ldots >$$

The prefix operator `next` produces the stream defined by the equation:

$$\texttt{next}(< x_0, x_1, x_2, \ldots >) = < x_1, x_2, x_3 \ldots >$$

For two streams `X` and `Y`, the infix operator `fby` (followed by) is defined as:

$$\texttt{X fby Y} = < x_0, y_0, y_1, \ldots >$$

Also note that

```
X = first X fby (next X)
```

In LUCID, expressions are structured by the `where` clause; expressions are built up from variables, constants and operators.

2.2.2 Variables and Expressions

The arithmetic, boolean and conditional operators, are applied pointwise to the sequences. For example, if `X` and `Y` are two variables, then the expression

```
X + Y
```

denotes the sequence whose n^{th} term is the sum of the n^{th} terms of X and Y, i.e., by the following sequence:

$$< x_0 + y_0,\ x_1 + y_1,\ \ldots >$$

If the variable Z is defined by

```
Z = X + Y;
```

then it takes the following sequence of values:

$$< z_0 = x_0 + y_0,\ z_1 = x_1 + y_1,\ \ldots >$$

An example which generates the natural numbers is:

```
nat = X
  where
    X = 0 fby X + 1;
  end
```

The process network corresponding to this filter is shown in Figure 2.2.

The sequence of squares of natural numbers is generated by the filter **square**.

```
square where
   square = 0 fby square + 2*N + 1;
   N      = 0 fby N + 1;
end
```

It should be recalled that the order of appearance of equations has no effect on the computed result.

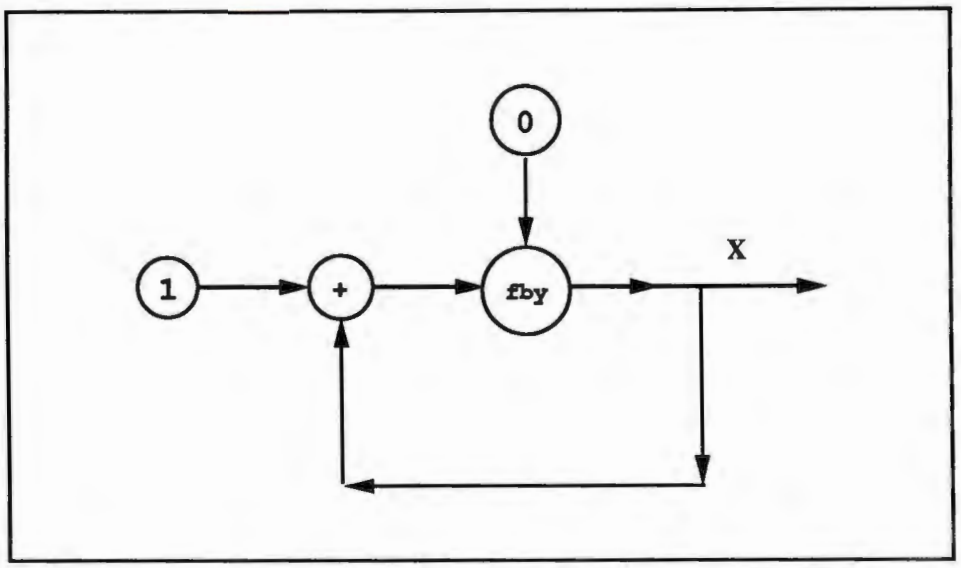


Figure 2.2: Network for the equation, `X = 0 fby X + 1`.

2.2.3 User Defined Functions

Recursive and non-recursive pointwise[1] functions (or filters) can be defined as whole programs or in the **where** clauses of another function. The following `factorial(N)` function requires one value of `N` to be input for each output value.

```
factorial(N) where
   factorial(N) = if N eq 0 then 1
                  else N*factorial(N-1)
end
```

Some of the more commonly used functions are built in the language as operators. The operator **wvr** (whenever) takes a variable `X` and a boolean `P`. It can be defined in terms of the previously defined operators as:

[1]The n^{th} value of a pointwise function or operation depends only on the n^{th} values of its arguments. The functions work in a manner similar to the data operations such as '+' described above.

```
X wvr P = if first P then X fby Z else Z
   where
      Z = next X wvr next P
   end
```

wvr works as an *sampling* operator. The output of the following program is a stream of odd numbers.

```
n wvr odd(n)
   where
      n = 0 fby n + 1;
      odd(n) = n mod 2 eq 0;
   end
```

Suitably defining isPrime(n) and using it in place of odd(n) in the above program will enable us to filter out all but the prime numbers.

The operator upon is a dual to wvr in the sense that it selectively stretches its input by repeating some of the values that it receives. It may also be thought of as a *delay* operator. The recursive definition of upon is as follows:

```
X upon P = X fby
           if first P
           then next X upon next P
           else X upon next P
```

Defining the streams X and P as

```
X = 0 fby X+1
P = false fby true fby P
```

The output of `X wvr P` selects only the odd numbers and that of `X upon P` contains each number twice. An alternative interpretation may be that **wvr** is synonymous with *skip, advance* while that of **upon** is *delay, repeat*.

2.3 Lucid for Real-Time: rLucid

In developing RLUCID from LUCID, firstly timing information is added to the semantics of LUCID. In addition, two new operators are introduced to make use of the timing information.

2.3.1 Timestamps

LUCID nodes and operations are mathematical functions which map input histories to output histories. The rate and order of arrival of inputs has no influence on the result of a program. But, in a reactive system, the absolute rate of arrival of inputs and their relative order may affect the output.

Timing information is introduced into the histories through timestamps. A timestamp corresponds to an occurrence date of an event (*cf.* §1.6).

2.3.2 Synchronous Model

When reactive systems are programmed using the synchronous hypothesis, the operations are assumed to be performed instantaneously. This implies that nodes or filters produce their output as soon as all input is received. Data flow infinitely fast through the network. Only when the input is incomplete, does a node wait. A node fires when all its operands are available. As a consequence, the runnable actions in a program are determined by the data dependencies.

A further consequence of the synchronous hypothesis is that event occur-

rences have no duration.

2.3.3 Timestamped Streams

The fundamental objects of RLUCID are *timestamped streams*. A timestamped stream is a pair (V,T) where V is a sequence of values and T, a sequence of dates. If the sequence of values is

$$< v_1,\ v_2,\ v_3, \ldots >$$

and the sequence of timestamps is

$$< t_1,\ t_2,\ t_3, \ldots >$$

then the desired interpretation is that the n^{th} value (daton) of V occurs at a time given by the n^{th} timestamp of T. Every daton has a timestamp, and the daton is available at the time given by its timestamp. The stream can be identified with the time sequence introduced in §1.6.

Anachronistic Streams

In LUCID, it is not required that all values of input stream be available or that the output values be computed in strict chronological order. Such *anachronistic* behavior may arise whenever `next` is used.

The program `lookAhead` computes how far ahead is a value of '0' from the present (input) value (*cf.* pg. 77 [26]). This program is an example of the anachronistic behavior of `next`.

```
lookAhead
    where
      lookAhead = if x eq 0 then 0
```

```
                    else 1 + next lookAhead
        end
```

One should carefully avoid writing such programs in real-time systems.

Uncausality, the dependence of the output of a filter at any instant on the future input, arising from the use of **next**, could lead to deadlocks.

```
X where X = next X end
```

In our model, the anachronistic behavior is disallowed, for it leads to *uncausality* and deadlocks in programs.[2] This behavior is curtailed in RLUCID. The timestamp of a future daton is never less than that of the present daton, consistent with the increasing time sequences in §2.3.3.

Timestamps are not limited to integers. In fact they may be made to correspond to real time from a global clock; in this case the reactive program acquires the notion of a universal time.

A virtual clock may be defined in terms of the dates (timestamps) corresponding to a variable. Then the sequence T defines the private (local) clock associated with the stream V. However, there are no clocks in RLUCID. Only they may be interpreted.

2.4 New Operators

Introducing timestamps does not in itself change the power of the language. The semantics change, but the programmer has no access to the timestamps.

Therefore, we first consider an operator which allows direct access to the timestamps.

[2]The dependence of the present value of a node on some future input is known as uncausality. Uncausality may be allowed in a distributed system, if the 'time inversion' is not too large. Further details may be found in Lamport [17].

2.4.1 The ck Operator

Suppose that a new operator, ck, is introduced.[3] (ck E) would generate a stream whose values are the timestamps of E. The timestamps of (ck E) are the same as those of E. This operator would allow the comparison of the timestamps. With ck, the programmer can determine whether one event actually occurred before another. Further, suppose that we have a stream T, of timestamps produced by the system clock.

Consider a program for merging two input streams in a time-wise manner. This may be accomplished by the following program.

```
Merge(A,B,T)
where
 Merge(A,B,t) =
    if first ck A <= first ck B
    then first A fby Merge(next A, B, next t)
    else first B fby Merge(A, next B, next t)
```

It will work when the two streams are not empty, at any given instant. Should only one of them have a daton, it will still work, if it is assumed that the timestamp of the daton, yet unavailable, is greater than the current time T.[4] However, this reasoning will fail if no stream has a daton. To handle this situation, we compare the timestamp of each daton with the current time. Should no datons exist, then the program waits until one appears.

The program, with these changes made, becomes:

```
Merge(A,B,T)
```

[3]Short form of clock.

[4]Formally, the stamp attached to a daton which never arrives is ∞. But, one may safely enquire if the stamp is $\geq$ a finite value, such as the stamp of a daton that has arrived.

```
where
  Merge(A,B,t) =
    if exists A or exists B then
       if first ck B >= first ck A
         then out1
         else out2
    else noout
    where
      out1 = first A fby Merge(next A, B, next t)
      out2 = first B fby Merge(A, next B, next t)
      noout = Merge(A, B, next t)
      exists(strm) = first ck strm <= first t
    end
end
```

The program is not yet correct. The program produces output in correct order. But the timestamps are not right. Without loss of generality, let us consider integer time and that `T = 1 fby T + 1`. Let `A` and `B` be the two streams:

$$\texttt{A} =< (a_0, 1),\ (a_1, 2),\ (a_2, 3),\ \ldots >$$

$$\texttt{B} =< (b_0, 2),\ (b_1, 3),\ (b_2, 4),\ \ldots >$$

The correct (expected) output is:

$$< (a_0, 1),\ (a_1, 2),\ (b_0, 2),\ (a_2, 3),\ (b_1, 3),\ (b_2, 4),\ \ldots >$$

According to the semantics of RLUCID expressions (see §2.6), the comparison '`>=`' carries a timestamp equal to the `max(date A, date B)`. Further, the timestamps carried by the output from the filter `Merge` are equal to `max(date A,`

`date B, date A,t)` or `max(date A, date B, date strm2,t)`. The output is not produced at the correct time, but as shown below.

$$< (a_0, 2),\ (a_1, 2),\ (b_0, 3),\ (a_2, 4),\ (b_1, 5),\ (b_2, 6),\ \ldots >$$

It appears that the program works by asking "is there some input now? ... is there some input now?" This notion is not consistent with the idea of data flowing through the network.

Further, the sampling time (system time `T`) and the data arrival time could be different. This violates the assumption of *synchronous hypothesis*. How fine an interval must one choose to sample data in order that the stamps on datons correspond to the times they are actually sent out?

2.4.2 Operators `Which` and `Synchro('||')`

One may add new operators to overcome these problems (eg. a special operator for comparing timestamps). They may bring new problems in their wake and the solution is not elegant at any rate.

We now consider two operators that can be added to RLUCID without bringing a fresh crop of problems, and yet take advantage of the timestamps.

A new operator `Which` when applied to two streams, returns a number depending on which of them arrived before the other, or if they arrived together.

Another operator "||" (read "Synchro")is also provided in the language to handle concurrency constraints adequately. The semantics for the new operators are given in the next section.

2.5 Examples

A couple of examples follow to illustrate the use of new operators `Which` and `Synchro`. Next in another example, a complete reactive program is given.

2.5.1 Merge and Weld

`Merge` filter given previously (*cf.*§2.4.1) can be written using `Which` as follows:

```
Merge(A,B)
where
  flag = Which(A, B)
  Merge(A,B) =
    if flag eq 1 then first A fby Merge(next A, B)
      else if flag eq 2 then first B fby Merge(A, next B)
      else first A fby first B fby Merge(next A, next B)
```

All problems resulting from the comparison of the timestamps are quite elegantly solved.

As an example for `Synchro`, consider a robot arm welding each of a stream of pieces `A` onto each of another stream of pieces `B`, both moving on an assembly line. Suppose that the signals `a` `b` are sent when `A` and `B` reaches their correct position. If the signal `ready` is sent to the robot arm whenever the two pieces are in their correct position, the pieces could be weld together. The following program can be used in this scenario.

```
weld(A,B) = ready wvr Synchro(a,b)
```

If `A` and `B` do not arrive together, which is more likely, one will wait for the other.

2.5.2 A Stopwatch

In the following a program for a simple stopwatch is presented to illustrate the use of the language, which has been adopted into RLUCID from [7].

The stopwatch receives three signals. Initially the stopwatch is idle (not running) and `onOff` makes the watch toggle between the running and idle states. Signals emitted from a `quartz` oscillator (possibly at 1/100 sec) are used in counting the time. While the stopwatch is `running`, `reset` sets the watch back to 0, or a signal from the `quartz` oscillator increments the `counter` and the display is updated accordingly.

The complete program of the stopwatch is as follows:

```
stopwatch(onOff, reset, quartz) = time
   where
      running = computestate(false, onOff, reset);
      CK = (quartz and running) or reset;
      time = counter(0,1,reset) upon CK;
      counter(...) = ...
      computestate(...) = ...
   end
```

The output of the stopwatch, `time`, is computed by the function `counter`:

```
counter(init,incr,reset) = y
   where
      y = init fby if reset then init
                   else y + incr
   end
```

onOff= <	*ff*	*tt*	*ff*	*ff*	*ff*	*ff*	...>
quartz= <	*tt*	*tt*	*tt*	*tt*	*tt*	*tt*	...>
running= <	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*	...>
CK= <	*ff*	*tt*	*tt*	*tt*	*tt*	*ff*	...>
reset= <	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	...>
time= <	0	0	1	2	3	4	...>
timestamps:	0	1	2	3	4	5	⟶

Table 2.1: A Sample Run of the Stopwatch Program.

The following definition for **computestate** takes an initial value and two signals **start** and **stop** to determine the present state, 'running' or 'idle' of the stopwatch.

```
computestate(init, start, stop) = state
   where
      state = init fby
              if (start and not state) then true
              else if (stop and state) then false
              else state
   end
```

The **counter** function with parameters 0, 1 and **reset** sampled according to a suitable clock **CK** will compute the time. Whenever a signal from **quartz** is received while the stopwatch is **running** or the **reset** signal is received, the **counter** function performs a cycle.

A sample run is shown in Table 2.1. Until the toggle switch **onOff** is activated, no time is shown on display. However, it keeps receiving the **quartz** signals. Although the timestamps are shown starting from '0', they correspond

to the general situation '$t + 0$'. It should be remembered that until a signal appears, the corresponding variable is `false`.

For a more realistic stopwatch, the reader is referred to [7].

2.6 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^{th} daton of V occurs at the time given by the n^{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 chronon streams.

2.6.1 Abstract Syntax

The syntax of the two is identical, except for the introduction of `Which` and "||". Therefore, we capitalize the first letter of RLUCID operators. Operators in lower case, are those of LUCID.

The basic abstract syntax of stream expressions is given in Table 2.2.

In the grammar given in Table 2.2 for RLUCID, 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.

2.6.2 Fixed Point Semantics

Although the semantics is separated into two parts — daton streams and chronon streams — one should note that the two are interdependent. The first, which deals with the values, is the same as for LUCID operators except for `Which` and

$$
\begin{array}{rcl}
E & ::= & X \\
& | & K \\
& | & E \ \texttt{Op} \ E \\
& | & \texttt{If} \ E \ \texttt{Then} \ E \ \texttt{Else} \ E \\
& | & \texttt{First} \ E \\
& | & \texttt{Next} \ E \\
& | & E \ \texttt{Fby} \ E \\
& | & \texttt{Which} \ (E, E) \\
& | & (E \parallel E) \\
& | & (E, E) \\
& | & X \ E \\
& | & E \ \texttt{Where} \ Q \ \texttt{End} \\
Q & ::= & X = E \mid X(X, \ldots, X) = E \mid Q; Q
\end{array}
$$

Table 2.2: Abstract Syntax of RLUCID

Synchro. This is given in Table 2.3.

In the following, X is a stream (V,T) and K is a constant (k,0).

The second part deals with the timestamps. Since Fby and If-Then-Else are non-strict operators more care should be taken in expressing their semantics. The non-strictness could lead to nonmonotonic and not strictly increasing timestamped streams. For a constant K, timestamps are assumed to be equal to '0', as the value of a constant is known *a priori* at all times. See Table 2.4.

Let P and Q be the two streams:

$$\mathtt{P} = < (p_0, 3),\ (p_1, 4),\ (p_2, 5),\ \ldots >$$

$$\mathtt{Q} = < (q_0, 1),\ (q_1, 2),\ (q_2, 3),\ \ldots >$$

If we were to define

$$\mathcal{D}(E_1\ \texttt{Fby}\ E_2) = \texttt{first}\ \mathcal{D}E_1\ \texttt{fby}\ \mathcal{D}E_2$$

then we would have the following stream for P Fby Q:

$$< (p_0, 3),\ (q_0, 1),\ (q_1, 2),\ (q_2, 3),\ \ldots >$$

Another pathological example would be P Fby K.

Similarly, defining $\mathcal{D}$ If-Then-Else as

$$\texttt{if}\ \mathcal{V}E_1\ \texttt{then}\ \texttt{max}(\mathcal{D}E_1, \mathcal{D}E_2)\ \texttt{else}\ \texttt{max}(\mathcal{D}E_1, \mathcal{D}E_3)$$

would cause problems. Let B be a boolean variable defined by:

$$\mathtt{B} = < (true, 1),\ (false, 2),\ (false, 3),\ \ldots >$$

Further, let Z = If B Then P Else Q. Then

$$\mathtt{Z} = < (p_0, 3),\ (q_1, 2),\ (q_2, 3),\ \ldots >$$

which is again a nonmonotonic stream.

The definitions given in the Table 2.4 for the dates of Fby and If-Then-Else ensure that uncausal behavior would not occur. The timestamps never decrease.

$$
\begin{aligned}
\mathcal{V}X &= x_\nu \\
\mathcal{V}K &= k \\
\mathcal{V}(E_1 \texttt{ Op } E_2) &= \mathcal{V}E_1 \texttt{ op } \mathcal{V}E_2 \\
\mathcal{V}(\texttt{If } E_1 \texttt{ Then } E_2 \texttt{ Else } E_3) &= \texttt{if } \mathcal{V}E_1 \texttt{ then } \mathcal{V}E_2 \texttt{ else } \mathcal{V}E_3 \\
\mathcal{V}(\texttt{First } E) &= \texttt{first } \mathcal{V}E \\
\mathcal{V}(\texttt{Next } E) &= \texttt{next } \mathcal{V}E \\
\mathcal{V}(E_1 \texttt{ Fby } E_2) &= \mathcal{V}E_1 \texttt{ fby } \mathcal{V}E_2 \\
\mathcal{V} \texttt{ Which}(E_1, E_2) &= \texttt{if } \mathcal{D}E_1 < \mathcal{D}E_2 \\
&\quad \texttt{then 1 fby } \mathcal{V} \texttt{ Which(next } E_1, E_2) \\
&\quad \texttt{else if } \mathcal{D}E_2 < \mathcal{D}E_1 \\
&\quad \texttt{then 2 fby } \mathcal{V} \texttt{ Which } (E_1, \texttt{next } E_2) \\
&\quad \texttt{else 3 fby } \mathcal{V} \texttt{ Which(next } E_1, \texttt{next } E_2) \\
\mathcal{V}(E_1 \parallel E_2) &= (\mathcal{V}E_1, \mathcal{V}E_2) \\
\mathcal{V}(X\ E) &= x_\nu(\mathcal{V}E) \\
\mathcal{V}(E \texttt{ Where } Q \texttt{ End}) &= (\mathcal{V}E) \texttt{ where } (\mathcal{V}Q) \texttt{ end} \\
\mathcal{V}(X = E) &= x_\nu = (\mathcal{V}E) \\
\mathcal{V}(X(X_1, \ldots, X_n) = E) &= x_\nu(x_{1_\nu}, \ldots, x_{n_\nu}) = (\mathcal{V}E) \\
\mathcal{V}(Q_1; Q_2) &= (\mathcal{V}Q_1); (\mathcal{V}Q_2)
\end{aligned}
$$

Table 2.3: Semantics of Values of RLUCID Expressions

$$
\begin{aligned}
\mathcal{D}X &= x_{\mathcal{D}} \\
\mathcal{D}K &= 0 \\
\mathcal{D}(E_1 \texttt{ Op } E_2) &= \texttt{max}(\mathcal{D} \texttt{ first } E_1, \mathcal{D} \texttt{ first } E_2, \\
&\qquad 0 \texttt{ fby } \mathcal{D}(\texttt{next } E_1 \texttt{ Op next } E_2)) \\
\mathcal{D}(\texttt{If } E_1 \texttt{ Then } E_2 \texttt{ Else } E_3) &= \texttt{max}(\texttt{if val } E_1 \texttt{then max}(\mathcal{D}E_1, \mathcal{D}E_2) \\
&\qquad \texttt{else max}(\mathcal{D}E_1, \mathcal{D}E_3), \\
&\qquad 0 \texttt{ fby } \mathcal{D}(\texttt{If } E_1 \texttt{ Then } E_2 \texttt{ Else } E_3)) \\
\mathcal{D}(\texttt{First } E) &= \texttt{first } \mathcal{D}E \\
\mathcal{D}(\texttt{Next } E) &= \texttt{next } \mathcal{D}E \\
\mathcal{D}(E_1 \texttt{ Fby } E_2) &= \texttt{max}(\texttt{first } \mathcal{D}E_1, \mathcal{D}E_1 \texttt{ fby } \mathcal{D}E_2) \\
\mathcal{D}(E_1 \parallel E_2) &= \texttt{max}(\mathcal{D} \texttt{ first } E_1, \mathcal{D} \texttt{ first } E_2, \\
&\qquad 0 \texttt{ fby } \mathcal{D}(\texttt{next } E_1 \parallel \texttt{next } E_2)) \\
\mathcal{D} \texttt{ Which}(E_1, E_2) &= \texttt{if } \mathcal{D}E_1 < \mathcal{D}E_2 \\
&\qquad \texttt{then } \mathcal{D}E_1 \texttt{ fby } \mathcal{D} \texttt{ Which}(\texttt{next } E_1, E_2) \\
&\qquad \texttt{else if } \mathcal{D}E_2 < \mathcal{D}E_1 \\
&\qquad \texttt{then } \mathcal{D}E_2 \texttt{ fby } \mathcal{D} \texttt{ Which}(E_1, \texttt{next } E_2) \\
&\qquad \texttt{else } \mathcal{D}E_1 \texttt{ fby } \mathcal{D} \texttt{ Which}(\texttt{next } E_1, \texttt{next } E_2) \\
\mathcal{D}(X\ E) &= x_{\mathcal{D}}(\mathcal{D}E) \\
\mathcal{D}(E \texttt{ Where } Q \texttt{ End}) &= (\mathcal{D}E) \texttt{ where } (\mathcal{D}Q) \texttt{ end} \\
\mathcal{D}(X = E) &= x_{\mathcal{D}} = (\mathcal{D}E) \\
\mathcal{D}(X(X_1, \ldots, X_n) = E) &= x_{\mathcal{D}}(x_{1_{\mathcal{D}}}, \ldots, x_{n_{\mathcal{D}}}) = (\mathcal{D}E) \\
\mathcal{D}(Q_1; Q_2) &= (\mathcal{D}Q_1); (\mathcal{D}Q_2)
\end{aligned}
$$

Table 2.4: Semantics of Dates of RLUCID Expressions

2.7 Limitations and suggestions for improvement

Dataflow languages have one tremendous advantage over their counterparts: regardless of the parallelism used in the evaluation, the result will be completely repeatable. It is not possible to design a program, either by choice or by accident, whose result depends upon a parallel process winning some "race conditions".

But there exist situations in operating systems, and in dealing with input/output, where some kind of race is just what is wanted. The behavior of an operating system depends upon the terminating order of the processes.

Abramsky and Sykes [1] have built operating systems by introducing a special operator, `merge`. The stream returned by `merge`($\mathbf{s}_1, \mathbf{s}_2$) contains the elements of the streams $\mathbf{s}_1$ and $\mathbf{s}_2$ in the order in which they are computed. This merge operation has been shown to be sufficient to encode all the operating systems applications. And it is not difficult to implement. Unfortunately, large programs that use `merge` become so complex that they are very difficult to reason about.

The limitations to reactive programming in RLUCID may come from *deadlocks* or *unbounded memory* as shown in the following.

Unbounded memory may be required in evaluating the above program. Or, it could happen in a program which may not require all the values it computes. Such programs may even cause deadlocks as the extra computations may fail to terminate.

Unbounded memory is required if the basic filters themselves require an unbounded memory; or an unbounded number of operations are created dynamically; or when the values computed get accumulated in the arcs connecting the nodes.

A classic example is the calculation of Fibonacci numbers using recursive definition of `fib`:

```
fib(n)= if n eq 0 then 1
            else if n eq 1 then 1
                 else fib(n-1) + fib(n-2)
```

If the previously computed results are not stored, or if the storage management scheme is not efficient, the computations could degrade the performance.

2.8 Conclusions

Reactive systems can programmed by extending LUCID with timing (stamps) and concurrent primitives. Assumption of *synchrony* (*cf.* §1.5) in RLUCID enables us to apply data operations pointwisely. The operator `ck` is more permissive and may lead to incorrect semantics as has been shown in the text. Hence, this operator is not included in the set of primitives. Being an extension to a general dataflow language LUCID, uncausal relations may develop with the use of operators like `next`, even in apparently innocuous programs. It should be possible to detect anachronistic behavior at run-time(?).

The two new primitives, `Which` and ||, are sufficient to specify synchronous systems. They enable the programmer to make use of timestamps without directly accessing them.

Further, the language may benefit further from an efficient implementation. Lack of one does not in any way limit its use as a powerful specification language. This will become clear in the following chapters. In the next chapter about the synchronous language LUSTRE, we see how efficiency can be gained by imposing synchronization requirements on input.

Chapter 3

Lustre

3.1 Introduction

It was maintained in Chapters 1 and 2 that dataflow languages are suitable to a particular class of real-time programs; the declarative style of programming is natural to control systems, where differential equations, operator networks and such, are used. Mathematical soundness and referential transparency and the ease of program transformation make them even more attractive. However, the need for generating efficient code, is nowhere more important than in real-time programming. Unbounded memory is another serious problem. Additional difficulties arise because of the absence of control structures. Further, good compilers are at a premium.

Efficient code generation became possible in ESTEREL [4] and LUSTRE [7] by imposing restrictions: every operator receives all of its inputs at the same time, and all the operators synchronously respond to their inputs. These languages transform a program into an automaton that can be minimized. Usually, an interface (written in 'C') bridges the gap between the asynchronous world and

the synchronous LUSTRE program (kernel). The problems related to this area of interfaces have not been fully investigated. The ground for this topic will be covered in Chapter 4.

Other synchronous formalisms include the dataflow programming languages, the SIGNAL [19] and the STATECHARTS visual formalism [12]. All these languages were designed for developing reactive systems, which include real-time process controllers and human-machine interfaces.

In this chapter, after presenting LUSTRE as in [23, 7], its operational semantics is given in RLUCID. For this purpose, LUSTRE extended with additional primitives, but having the same 'extensional meaning' as the original lustre, is chosen. The correspondence between LUSTRE and RLUCID is then exhibited through the operational semantics of the first in terms of the second.

3.1.1 Notion of time

One major difference between LUCID and LUSTRE programs is in how they view time. LUCID programs strictly adhere to the formalism of Caspi and Halbwachs (*cf.*§1.6), and deal with time in terms of the timestamps. In contrast, there is no direct correspondence to their model in LUSTRE programs. The notion of time is incorporated through abstract clocks, which are associated with all streams.

In LUSTRE, a *clocked stream* takes its n^{th} value at the n^{th} instant of its clock. Every clock, except for the global clock, is itself a clocked stream. The global clock, the finest, defines the instants at which a program runs. One tick of the global clock defines an "instant".

Reactive systems are programmed with the assumption of the synchronous hypothesis. As a consequence, the data operations can be performed pointwise.

LUSTRE has been used to program automatic control examples [2], systolic

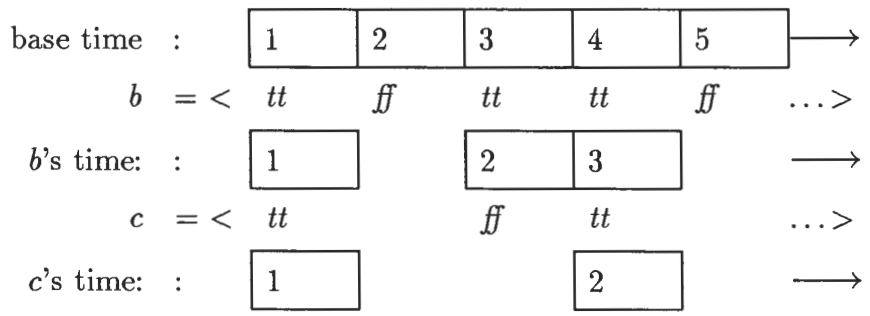


Table 3.1: Clocks in Lustre programs.

algorithms [11] and hardware circuits [10]. The semantics and the prototype compiler were presented in Plaice's dissertation [23].

3.2 The Language

The language LUSTRE is very similar to LUCID. Since the fundamental objects in both languages are streams, the basic operations are also similar. Whenever possible, a comparison is made between them in the following.

3.2.1 Streams and Nodes

If the sequence of values

$$< e_0, e_1, \ldots, e_n, \ldots >,$$

define a clocked stream, then the clocked stream takes the value e_n at the n^{th} instant of its clock.

A clock is defined by either the base clock of the program[1], or a boolean stream. In the latter case, the sequence of instants at which the boolean stream

[1] Intuitively, the sequence of instants at which the program is active.

is true defines a clock, In Table 3.1, the clock of a boolean stream b is the base clock, while the clock c is that defined by b. Thus clocks can be defined in a nested manner.

Clocks allow subsystems to evolve at different rates. In fact, the same subsystem may be used at different rates at different times. In a `watch`, the same counter may be used in counting `seconds`, `minutes`, and `hours`, each at a different rate.

With clocks, the value of a variable need not be calculated until it is required. This fact is useful for generating efficient code, but it is far more useful in expressing relationships that hold between variables. In our example, the `hour` hand may not move if the `minute` hand does not also move. In this example, `hour` is a subclock of the `minute`, whose clock is the base clock.

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; further, it may be *totally* independent of any notion of physical time.

Clocked streams are manipulated by nodes, which are functions between streams. A node can be predefined or user-defined, in the latter case with a set of equations. They correspond to LUCID functions and user defined functions. The node in the following example, returns the input stream as its output:

```
node IDENTITY (x:*) returns (y:*)
let y = x tel
```

3.2.2 Variables and Equations

The building blocks of *expressions* are: *variables, constants, data operators* and *temporal operators.* A variable is defined by an *equation*, and its value is a

clocked stream. Nodes are defined through a set of equations using the *let ... tel* constructs, similar to the *where* clause of LUCID. If `X` is a variable and `E` is an expression, `X=E` defines `X`. The clock and the sequence of values of `X` and `E` are identical.

If `E`'s clock is h and its sequence of values is

$$< e_0, e_1, \ldots, e_n, \ldots >,$$

then `X`'s clock is h and its sequence is

$$< x_0 = e_0, x_1 = e_1, \ldots, x_n = e_n, \ldots > .$$

An expression `E` is deemed to have a clock, either the `base` clock or a boolean expression `B`. If the clock is `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`.

Streams are possibly infinite sequences of values.

Each variable which is not an input is defined by exactly one equation. The order of a set of equations has no significance.

3.2.3 Constants

A constant appearing in an expression means the infinite constant sequence on the appropriate clock. For example, `1 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.

3.2.4 Data Operators

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

To avoid the need for an unbounded memory, all operands of every operator have *exactly* the same notion of time, hence the same clock. This 'clock consistency check' is made by the compiler.

If, two variables `X` and `Y`, are on the same clock `H`, the expression

```
if X>Y then X else Y
```

denotes the sequence whose n^{th} term at the n^{th} instant of the clock `H`, is the larger of the n^{th} terms of `X` and `Y`.

If a variable `Z` defined by the above equation then its values at every instant of the clock are equal to the larger of `X` and `Y`.

3.2.5 Synchronous Operators

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

The `pre` ("previous") operator returns the value of its argument at the previous instant. If

$$\texttt{X} =< x_0, x_1, \ldots, x_n, \ldots >,$$

then

$$\texttt{pre X} =< \texttt{nil}, x_0, x_1, \ldots, x_{n-1}, \ldots >,$$

where `nil` is an *undefined* value, similar to the value of an uninitialized variable in imperative languages. The clocks of `X` and `pre X` are the same.

All data operators are strict with respect to `nil`, i.e., return the value `nil` each time that one of their operands is the value `nil`. But, if the condition is true, the `else` branch of `if-then-else` is not considered; if the condition is false, the `then` branch is not considered. For example,

```
X = if true then 1 else pre X
```

defines `X` as the constant sequence

$$< 1, 1, \ldots, 1, \ldots >$$

on the base clock.

If

$$\texttt{X} =< x_0, x_1, \ldots, x_n, \ldots >$$

and

$$\texttt{Y} =< y_0, y_1, \ldots, y_n, \ldots >$$

are two expressions of the same type and of the same clock, then `X->Y`[2] is defined by:

$$\texttt{X->Y} =< x_0, y_1, y_2, \ldots, y_n, \ldots >,$$

It is always equal to `Y` except at the initial instant.

The example

```
X = 0 -> pre X + 1
```

specifies that `X` is initially 0 and is incremented by 1 at subsequent instants. If `X` is therefore the sequence of the natural numbers on the base clock. It is a counter of instants.

[2]Read '->' as 'followed by'.

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

$$y_0 = init$$
$$y_n = ay_{n-1} + bx_n, \quad n > 0$$

is written in LUSTRE as:

```
Y = init -> a * pre Y + b*X
```

As far as the sequence of values are concerned, `X = 0 -> pre X + 1` is identical to `X = 0 fby X + 1` of LUCID. Further, `pre` is the right inverse of `next` in LUCID.[3] The linear filter in LUCID is

```
Y = init fby (a*Y + b* next X)
```

The essential difference between the two is that there is an implicit `pre` before every variable on the right hand side in LUCID.

3.2.6 Clock-changing Operators

When a stream is sampled or projected, we get streams whose clocks are coarser or finer grained respectively.

For a boolean expression `B` and an expression `E` on the same clock, the sequence of values taken by `E when B` are those of `E` corresponding to the 'true' values taken by `B`. The clock of this expression is different from that of `E` and `B` as in Table 3.2.

We say that `B` is the sampling clock of `X`. `X` understands only the sequence of instants at which `B` is true. It makes no sense to talk of the value of `X` at an instant when `B` is false. Additional examples of `when` may be found in §3.2.8.

[3] While `pre next X = X`, `next pre X = nil fby next X` but, $\neq$ `X`.

E = <	e_1	e_2	e_3	e_4	e_5	...>
B = <	*tt*	*ff*	*tt*	*tt*	*ff*	...>
X=E when B = <	$x_1 = e_1$		$x_2 = e_3$	$x_3 = e_4$		...>
time of E, B:	1	2	3	4	5	⟶

Table 3.2: Oversampling operator: `when`.

When applying an operator to expressions having different clocks, such as adding `X` and `E` in the example in Table 3.2, larger and larger memory may be required to save the old values of `E`. This has been avoided in the original version of LUSTRE [23] by placing the arguments on the same clock, either by sampling `E`, or by projecting `X`, using the `current` operator.

For an expression `E` on clock `B` and a sampling clock `B`, the clock of `current E` is that of `B`. Its value at each instant is that taken by `E` the last time, upto and including the current instant, that `B` was true. Table 3.3 illustrates the combined effect of the `when` and `current` operators.

The clocks of a node form a tree. The `base` is at the root. Its subclocks are the children in the tree. A branch in the tree is created by using the operator `when`. Using the operator `current` moves the clock towards the root.

Finally, the clock of an expression can be accessed with the `ck` operator: `ck(E)` yields the clock of expression `E`.

3.2.7 Nodes and Networks

Any LUSTRE program, just as a LUCID program, can be considered to be a network of operators connected by wires. The network corresponding to the equation

```
X = 0 -> pre X + 1
```

E= <	e_1	e_2	e_3	e_4	e_5	...>
B= <	tt	ff	tt	tt	ff	...>
X = E when B= <	e_1		e_3	e_4		...>
Y = current X= <	e_1	e_1	e_3	e_4	e_4	...>
Z = 1 when B= <	1		1	1		...>
W = Z -> X= <	1		e_3	e_4		...>
X=E when B= <	$x_1 = e_1$		$x_2 = e_3$	$x_3 = e_4$		...>
time of E, B:	1	2	3	4	5	$\longrightarrow$

Table 3.3: Sampling and Projection operators: when and current.

shown in Figure 3.1.[4]

A node, just as a filter or function in LUCID, is a LUSTRE subprogram. It takes streams as input, and returns streams as output. It may possibly use some local streams, in computing the output. A counter, similar to the one defined in §2.5.2, may be written as:

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

One may define even numbers and odd numbers by even and odd, and the cyclic sequence of naturals modulo 5 mod5 as:

```
even = COUNT(0, 2, false)
```

[4]The network in Figure 3.1 gives the impression that constants are stream generators. A constant is really a function which takes the base clock as input.

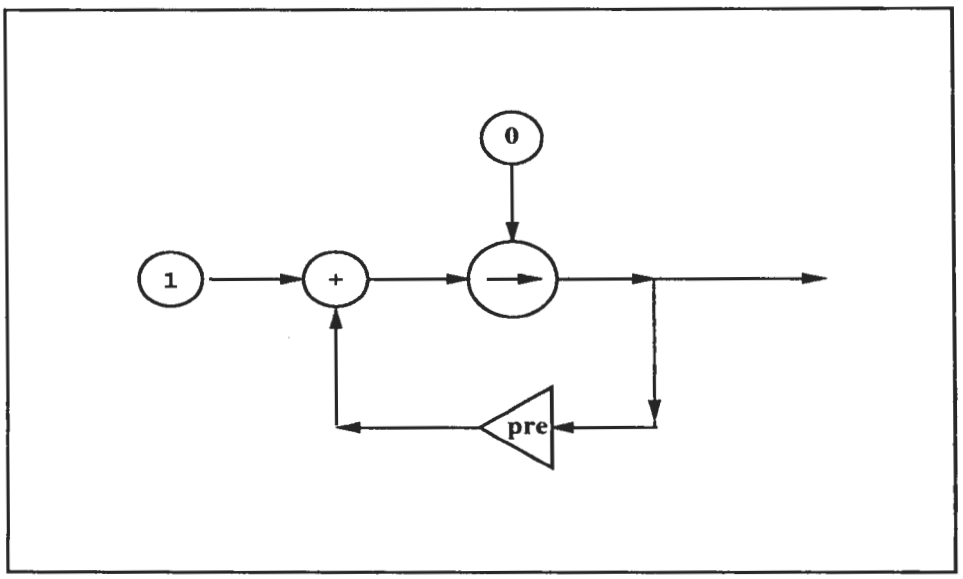


Figure 3.1: Network of `X = 0 -> pre X + 1`.

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

Node instantiation is done in a functional manner. If `N` is the name of a node, it is declared with the header

`node N` $(i_0 : \tau_0; \ldots; i_p : \tau_p)$ `returns` $(i_{p+1} : \tau_{p+1}; \ldots; i_q : \tau_q)$`;`

If $E_0, \ldots, E_p$ are expressions of type $\tau_0, \ldots, \tau_p$, then the node `N`$(E_0, \ldots, E_p)$ is an expression of type $(\tau_{p+1}, \ldots, \tau_q)$.

3.2.8 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)
```

`B`	= <*tt*	*ff*	*tt*	*ff*	*tt*	...>
`COUNT(0 when B,`						
`1 when B, false when B)`	= <0		1		2	...>
`COUNT(0,1,false) when B`	= <0		2		4	...>

Table 3.4: Sampling of input vs. sampling of output.

counts each time that `C` is true. `when` does not commute with the temporal operators. In general filtering a node's input is different from filtering its output (*cf.* Table 3.4).

Input to a node may consist of streams of different clocks, but when a parameter's clock is not the `base` clock, that clock must itself be passed as a parameter. The `on` form of the expression is used as illustrated in the following example.

```
node N (millisec:bool on base; second:bool on millisec)
       returns ...
```

Here, `millisec` is on the base clock and `second`'s clock is `millisec`, i.e., `second` only exists when `millisec` is true.

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

```
node ALTERNATE (x:*) returns (clock:bool; y:*) let
   clock = true -> not pre clock
   y = x when clock
tel
```

removes every second input.

3.3 A Stopwatch

The stopwatch described in §2.5.2 is used as an illustration in the following. We may write computeState as follows:

```
node computeState(init, start, reset: bool )
        returns (state: bool);
let
  state = init ->
              if start and not pre state then true
              else if reset and pre state then false
              else pre state;
tel;
```

The node counter previously defined, computes the time. The complete program for stopwatch is:

```
node stopwatch(onOff, reset, oscillator: bool)
        returns( time: int);
var clock, running: bool;
let
   time =  current counter((0, 1, reset) when clock);
   clock = true -> ( oscillator and running) or reset;
   running = computeState(false, onOff, onOff);
tel;
```

When the stopwatch is not idle, a signal from the oscillator updates the time; a reset signal resets the time to zero.

Since the program operates only in a *synchronous* environment, an interface to the outside world is required. This interface will be discussed in Chapter 4

3.4 Semantics

In the denotational semantics given in Plaice's dissertation [23], a clocked stream is a pair: a value stream and a clock. A clock, in turn, would be either the base clock or another stream.

The approach solved the problem associated with the `current` operator, namely that there is an implicit argument which is not readily visible. By maintaining a tree of clocks of §3.2.6, `current` acquired this clock.

A different approach is taken here. We consider an extension of LUSTRE, with two clock-changing operators, `on` and `onto`. `on` will deal with the clocks of constants and `onto` will implement both `current` and `when`.

Both `when` and `current` create new sequences of timestamps. For `when`, it is the sequence of timestamps at which the associated values are true. For `current`, it is the sequence of timestamps of its "clock's clock".

We therefore create a language for clocks, whereby one can create new clocks. The operator `truesOf(B)` generates a timestamped stream whose values are the timestamps when `B` is true. `E when B` therefore becomes `E onto truesOf B`.

`current E` becomes `E onto ck C`, where `C` is the "clock" of `E`. It is statically determined by the clock calculus from the program, as outlined in [7] and detailed in [23].

The examples in Table 3.5 illustrate these operators.

Another problem arises from the use of constants. If no clock is specified, then a constant `k` is assumed to run at the same rate as the base clock of the node. If we put this directly into the semantics, then LUSTRE loses its referential transparency. A better way would be to translate `k` into `k on ck(arg)`, where `arg` is an argument to the node that is to run on the base clock.

The syntax we consider is a modified version of LUSTRE, so that it better

E = <	e_1	e_2	e_3	e_4	e_5	...>
B = <	*tt*	*ff*	*tt*	*tt*	*ff*	...>
truesOf B = <	1		3	4		...>
E onto truesOf B = <	e_1		e_3	e_4		...>
X=E when B = <	e_1		e_3	e_4		...>
ck B = <	1	2	3	4	5	...>
X onto ck B = <	e_1	e_1	e_3	e_4	e_4	...>
current X = <	e_1	e_1	e_3	e_4	e_4	...>
time of E, B:	1	2	3	4	5	⟶

Table 3.5: The operators `onto` and `current`.

corresponds to RLUCID. The basic abstract syntax of a clocked stream expression is shown in the Table 3.6

In the above grammar, K is a constant, X is an identifier, H is a clock whose values are dates from a physical clock, `Op` is a data operator, such as the arithmetic, boolean and relational operators. The values of `truesOf E` is a sequence consisting only of the *true* values of `E`. Q is an equation or a group of them, such as those found in the body of a node instantiation, similar to the `where` clause of RLUCID.

The semantics are given in RLUCID. Therefore no explicit division into values and dates is made. The operators corresponding to LUSTRE are close enough to those of RLUCID. Hence we capitalize the first letter of all LUSTRE operators and use all lower case letters for those of RLUCID in Table 3.7. In this table, `fst`, applied to an ordered pair (x, y), yields x.

$$
\begin{array}{rcl}
E & ::= & X \\
& | & K \\
& | & E \texttt{ Op } E \\
& | & \texttt{If } E \texttt{ Then } E \texttt{ Else } E \\
& | & \texttt{Pre } E \\
& | & E \texttt{ -> } E \\
& | & E \texttt{ On } H \\
& | & E \texttt{ Onto } H \\
& | & (E, E) \\
& | & X\ E \\
& | & E \texttt{ Where } Q \texttt{ End} \\
Q & ::= & X = E \mid X(X, \ldots, X) = E \mid Q; Q \\
H & ::= & 0 \mid \texttt{ck } E \mid \texttt{truesOf } E \mid H \texttt{ fby } H
\end{array}
$$

Table 3.6: Abstract Syntax of extended LUSTRE.

$$
\begin{aligned}
E_1 \texttt{ Op } E_2 &= E_1 \texttt{ op } E_2 \\
\texttt{If } E_1 \texttt{ Then } E_2 \texttt{ Else } E_3 &= \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \\
\texttt{Pre } E &= \texttt{nil fby } E \\
E_1 \texttt{ -> } E_2 &= E_1 \texttt{ fby next } E_2 \\
\texttt{truesOf } E &= \texttt{ck } E \texttt{ wvr } E \\
E \texttt{ On } H &= \texttt{fst}(E \parallel H) \\
E \texttt{ Onto } H &= \texttt{select}(E, H, \texttt{nil})
\end{aligned}
$$

where

```
select(e,h,v) =
        if which(e,h) eq 2
           then v fby select(e, next h, v)
           else select(next e, h, first e)
```

Table 3.7: Alternative Semantics of LUSTRE.

3.5 Discussion

Clock consistency[5] was defined dynamically in the semantics of LUSTRE [23]. It is possible for streams to deadlock because of clock inconsistencies. To check the consistency of clocks, *clock inferencing* was introduced. Thus we have two steps to go through in determining the meaning of programs. A LUSTRE program which is correct according to the static semantics (clock checking) can be compiled.

Clocks are interpreted in terms of sequences of timestamps. A subclock of a clock (boolean stream) can be constructed by using the primitive `truesOf`.

The operators `onto` and `on` are more general and the language becomes more powerful than if it were to have only `current` and `when`. This is one of the areas that requires further research, to explore the possibility of enriching the language.

Semantics exists only if the clocks are consistent, i.e., if the timestamps of operands of an operator are all equal. In other words, only correct programs have semantics. An expression such as '`X + Y`' has a meaning only if at every cycle of the computation, the dates of `X` and `Y` are identical.[6]

This has important consequences. A node requires only a bounded memory.

When calculations are done in a synchronous manner, the operands of every basic node evolve at exactly the same rate. No past values need be memorized to handle streams which evolve at different rates.

When arguments are synchronous, only those of the memory operators need be saved. Since no recursive instantiations are allowed only a finite number of them exist. By counting the total number of memory operators, `pre` and

[5]It states that the operands of any operator must be on the same clock.

[6]All input and output from a node is *synchronous*.

`current`, one can determine the total size of the memory required.

Further, deadlock detection and efficient code generation are among the other advantages.

3.6 Summary and Conclusion

The alternative semantics of LUSTRE shows that LUSTRE is a subset of RLUCID, to which a clock calculus has been added, to ensure that there are never any clock inconsistencies.

Based on the same idea of dated streams, LUSTRE combines efficiency and efficacy of programming real-time systems by demanding the input in a specific manner. Synchronous interpretation of streams, especially is important for the generation of efficient code, although it places restrictions on the type of programs that can be written. The other advantages of the formalism are detection of deadlocks and bounded memory.[7]

As no restrictions of synchronization are applied to the input fed to RLUCID programs, they require no interfaces. On other hand, LUSTRE programs need an interface to synchronize the input, which forms the subject of next chapter.

[7]The stream `1 when false` is empty. This deadlock is allowed in LUSTRE as it does not conflict with the synchronous behavior of the program.

Chapter 4

Interface Programming

4.1 Introduction

LUSTRE is an elegant language for programming reactive systems, and LUSTRE programs can be compiled into very high quality code. However, this code can not stand alone. A reactive system must deal with the actual environment that may not adhere to the strict conditions expected by a LUSTRE program.

The LUSTRE semantics assumes that the environment is synchronous: if the arguments of a node have the same clock, then the values for any given instant must arrive at the node at *exactly* the same moment.

This assumption is very strong. It would be unreasonable to expect the outside world to meet such specifications. In fact, we would be surprised if it did.

The solution adopted here is to build an interface between the "real world" and the LUSTRE kernel. This interface takes the possibly non-synchronous environment and synchronizes the input for the kernel.

Traditionally, this interface is written in a standard imperative language,

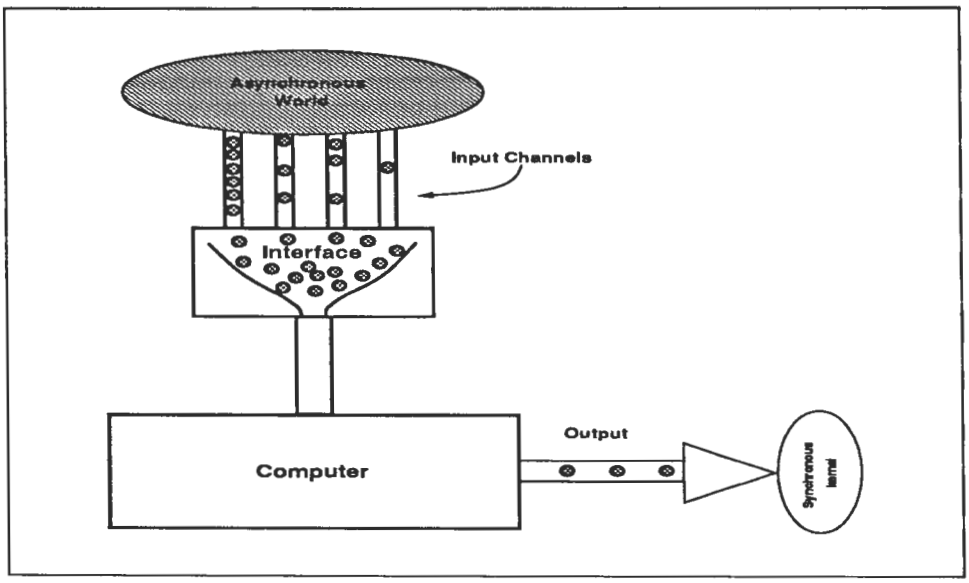


Figure 4.1: Relation of an Interface to a Reactive system.

such as C. In this chapter, we will show that RLUCID is sufficient to describe any interface that we might wish. Furthermore, we can define a simple language, APRIL,[1] which will allow the concise specification of most interfaces.

4.2 Interface

In the following, we assume the configuration shown in Figure 4.1 for reactive systems. A hardware multiplexer receives the stimuli from the environment. It adds timing information (timestamps) to each of them and transmits them in to the software interface.

It is assumed that the multiplexer can cope with the frequency of the input. If the multiplexer cannot deal with the input, then no software interface will be able to do so. Thus the multiplexer is counted on (somehow) to resolve two

[1]It stands for *A Programmable Interface Language*. Incidentally, it took its birth in April.

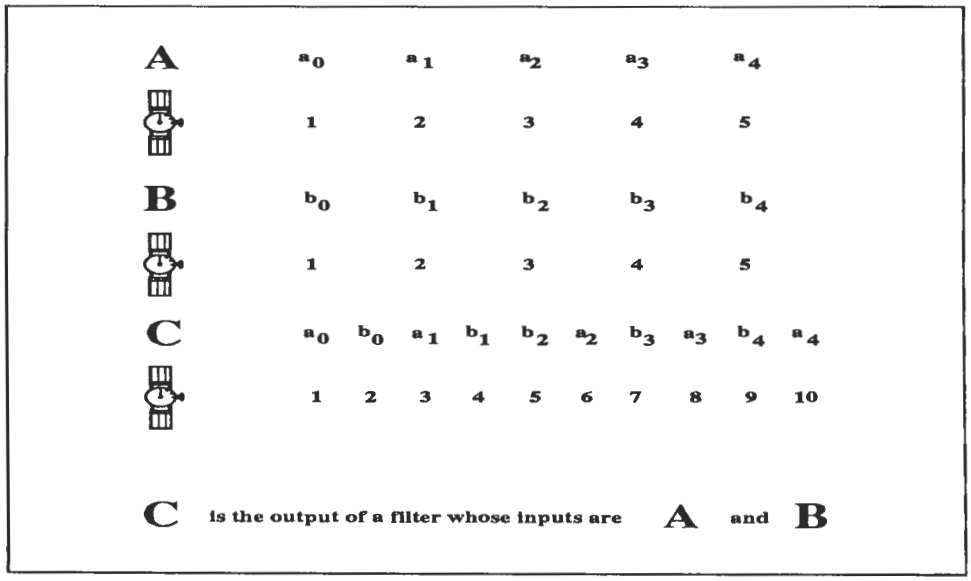


Figure 4.2: Clocks: Local and Global.

signals that occur at the same time.[2]

We envisage only a minimal processing of the signals by the interface, to satisfy data constraints and concurrency. But this processing should be kept to a bare minimum so that the interface does all its the work as fast as possible.

In the configuration presented in Figure 4.1, the interface drives the reactive kernel. This view is consistent with the synchronous hypothesis in that, the kernel having responded to the last input, waits for more input. It is only when there is a request from the interface that the kernel responds.

The LUSTRE clocks are independent of any notion of physical time. It is not meaningful to talk of the passage of time in between the 'ticks' of clock. Consider two variables A and B. The instants when variables receive their values, define the

[2]It may be argued that if the times of occurrences are measured to arbitrary precision, then two signals will never be simultaneous. At least, it would be impossible to show that two events took place at the same time without imposing certain restrictions on the precision to which time is measured.

ticks of the clocks associated with them as shown in figure 4.2. The clocks may be thought as associating an integer index with each occurrence of the values of a variable.

Suppose that we are interested in the global order of the subevents of A and B as seen by an external observer. The global clock is then the union of the clocks of A and B. Unless we know the order in which the events of A occurred relative to those of B as seen by C (in Figure 4.2), a global clock cannot be constructed. An interface which sequences the inputs from the environment in a timely order and passes them on to the kernel functions as a global clock.

4.3 Interfaces in Real-Time Lucid

The input to the interface consists of timestamped streams (*cf.* chapter 2). The interface functions like a filter.[3] Consider a reactive system which is to treat its stimuli on a first-come first-served basis. For the sake of simplicity, let us assume that two sensors in the environment send signals on the two channels connected to the reactive program as shown in Figure 4.3.

Channels 1 and 2 supply streams of values to the interface which passes them on to the kernel on a first-come first-served basis. This may be programmed in RLUCID as follows:

```
Int2(Ch1,Ch2)
where
  flag = Which(ch1, ch2)
```

[3]Hardware multiplexers resolve non-deterministically two or more signals received at the same time. A software multiplexer, which behaves like a filter or a function, fails to work in this case as it constitutes an essential singularity to a function, and such functions are not continuous.

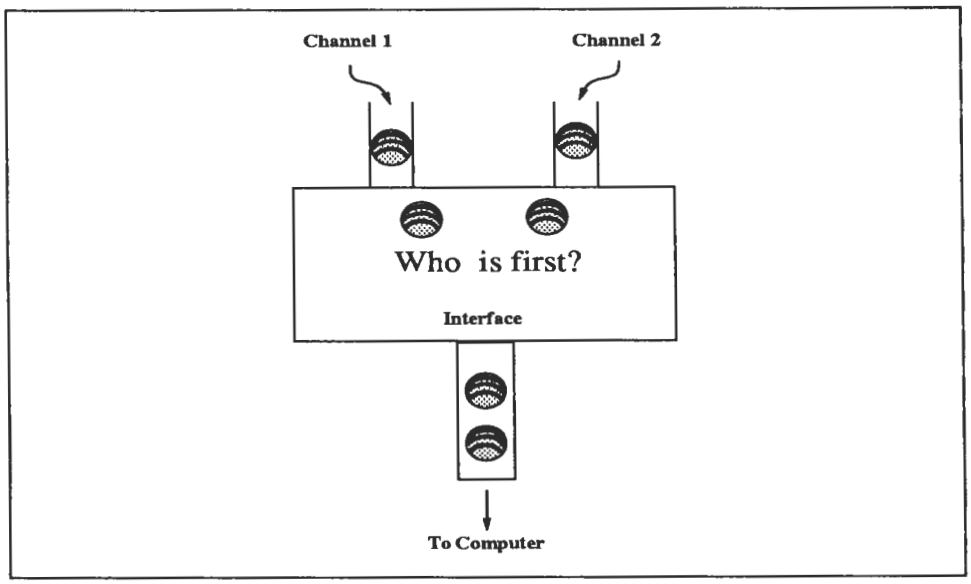


Figure 4.3: An interface with two input channels.

```
Int2(ch1,ch2) =
  if flag eq 1 then first ch1 fby Int2(next ch1, ch2)
  else if flag eq 2 then first ch2
                         fby Int2(ch1, next ch2)
  else if flag eq 3 first ch1 fby first ch2
                         fby Int2(next ch1, next ch2)
```

This agrees both in spirit and contents with the dataflow model assumed for the interfaces (reactive systems). Note that nothing ever happens unless there is input. As soon as there is a daton on either channel, it goes through the interface. However, if both of them come together, one has to decide what the output must be. Here we simply chose that of `Ch1` to precede the other.

It is easy to extend the interface for the case of three channels as in the following.

```
Int3(Ch1,Ch2,Ch3)
```

```
where
 Int3(ch1,ch2,ch3) =
    if flag eq 1
    then first ch1 fby Int3(next ch1, ch2, ch3)
    else if flag eq 2 then
        if flag1 eq 1
        then first ch2 fby Int3(ch1, next ch2, ch3)
        else if flag eq 2
        then first ch3 fby Int3(ch1,ch2, next ch3)
        else first ch2 fby first ch3 fby
             Int3(ch1, next ch2, next ch3)
        where flag1 = Which(ch1, ch2) end
    else first ch1 fby first ch2 fby
         first ch3 fby Int3(next ch1, next ch2, next ch3)
    where flag = Which(ch1, Int2(ch2,ch3)) end
end
```

In fact, this program can be extended to any number of arguments. Although the concept is simple, one can imagine the explosion of details which have to be carefully programmed. And right now, we are only dealing with a first-come first-served interface. It is left to the reader to imagine what might happen when several inputs must all arrive together, or when some input are to be selected only if a boolean input is true, as may be required by the LUSTRE kernel.

Although RLUCID is sufficient to write interfaces, it is by no means easy to do so. However, the general ideas can be retained for a special programming language, APRIL, whose semantics will be given in RLUCID.

4.4 APRIL

APRIL is a language which is designed to express *synchronous interfaces* in a dataflow manner. The language has six primitives, which are sufficient to write many interfaces.

4.4.1 Abstract Syntax

As in RLUCID, it is assumed that we are dealing with time-stamped streams, noted by a, but the timestamps are not visible to the user, only the relative order of arrival of stimuli is. Streams of pure signals will be referred to by b.

An interface is a filter which takes several streams, whose clocks may be quite disparate, and produces a single stream of packets which can be used by the reactive kernel. A specification for an interface is denoted by σ.

Streams of pure signals are converted into boolean streams through the use of β. The syntax for β is included in the Table 4.1.

The grammar for specifications, σ has the form given in the Table 4.1.

`SignalOne`(β_1, β_2) ensures that the first available pure signal will generate *true*, and the others will generate *false*. If, however, both arrive simultaneously, then β_2 generates *true* following β_1. `SignalBoth` differs only in the latter case in that both signals generate true together.

`SelectOne`(σ_1, σ_2) is to take the first available value from the streams σ_1 and σ_2. If both are available at the same time, then σ_1 is selected following which σ_2 is selected. `SelectBoth` sends the values from both channels together when they are available at the same time.

`Together`(σ_1, σ_2) means that a value from *each* of σ_1 and σ_2 must be taken.

`Wait`(σ_1, σ_2) means that if a value is available from σ_1, and if a value is available from σ_2, then the former value is sent on. That is σ_1 waits for σ_2. Note

$$
\begin{array}{rcl}
\sigma & ::= & a \\
& | & \beta \\
& | & \texttt{Wait}(\sigma, \sigma) \\
& | & \texttt{SelectOne}(\sigma, \sigma) \\
& | & \texttt{SelectBoth}(\sigma, \sigma) \\
& | & \texttt{Together}(\sigma, \sigma) \\
\beta & ::= & b \\
& | & \texttt{SignalOne}(\beta, \beta) \\
& | & \texttt{SignalBoth}(\beta, \beta)
\end{array}
$$

Table 4.1: Abstract Syntax for APRIL.

that `Wait` is the similar to on of LUSTRE.

Priorities are not necessary when working under the assumption of synchronous hypothesis (*cf.* 1.5), as the computers are assumed to run infinitely fast. It is also true in the case of the systems in which no reaction takes longer than the least interval between two inputs.

It may further be noted that all the above constructs except for **`Wait`**, can be easily extended to the case of *n-streams*. For example:

$$\texttt{SelectOne}(\sigma_1, \sigma_2, \sigma_3) = \texttt{SelectOne}(\sigma_1, \texttt{SelectOne}(\sigma_2, \sigma_3))$$

or:

$$\texttt{SelectOne}(\texttt{SelectOne}(\sigma_1, \sigma_2), \sigma_3)$$

4.4.2 Semantics

The specification σ of an interface actually produces a stream of synchronous *packets*. If, at any moment, no values are available, then no packet is generated.

The semantics is given in the following.

```
SignalOne(β1, β2) =
   case which(β1, β2) of
   1: ( true || false ) fby
      SignalOne(next β1, β2)
   2: ( false || true ) fby
      SignalOne(β1, next β2)
   3: ( true || false ) fby
      ( false || true ) fby
      SignalOne(next β1, next β2)

SignalBoth(β1, β2) =
   case which(β1, β2) of
   1: ( true || false ) fby
      SignalBoth(next β1, β2)
   2: ( false || true ) fby
      SignalBoth(β1, next β2)
   3: ( true || true ) fby
      SignalBoth(next β1, next β2)

SelectOne(σ1, σ2) =
   case which(σ1, σ2) of
```

```
   1: ( first σ1 || zap(σ2) ) fby
      SelectOne(next σ1,σ2)
   2: ( zap( σ1 ) || first σ2) fby
      SelectOne(σ1, next σ2)
   3: ( first σ1 || zap(σ2) ) fby
      ( zap( σ1 ) || first σ2) fby
      SelectOne(next σ1, next σ2)

SelectBoth(σ1,σ2) =
   case which(σ1,σ2) of
   1: ( first σ1 || zap(σ2) ) fby
      SelectBoth(next σ1,σ2)
   2: ( zap( σ1 ) || first σ2) fby
      SelectBoth(σ1, next σ2)
   3: ( first σ1 || first σ2 ) fby
      SelectBoth(next σ1, next σ2)

Together(σ1,σ2) = σ1,|| σ2

Wait(σ1, σ2) =
   if (first σ2 eq first σ2) then first σ1
      fby Wait(next σ1, next σ2)

zap (a) = ⊥
zap SelectOne (a, b) = zap(a) || zap(b)
zap SelectBoth (a, b) = zap(a) || zap(b)
```

zap SignalOne (a, b) = zap(a) || zap(b)

zap SignalBoth (a, b) = zap(a) || zap(b)

zap Together (a, b) = zap(a) || zap(b)

zap Wait (a, b) = zap(a) || zap(b)

4.4.3 Examples

In building interfaces to LUSTRE programs, all defined constructs are not necessary.

The interface for the watch presented in Chapter 3 is amazingly simple in APRIL, as given below:

$$\Sigma = \texttt{SignalOne}(\texttt{SignalOne}(\mathit{quartz},\ \mathit{reset}), \mathit{OnOff})$$

The example shown in Figure 4.4 shows the output of the 4-channel interface with specification

$$\Sigma = \texttt{SelectOne}(\texttt{ch1}, \texttt{Together}(\texttt{ch2}, \texttt{ch3}), \texttt{ch4})$$

on sample input.

This four channel interface might be used, for example, in a power control system. Channels 2 and 3 send information about dependent variables such as, temperature and rate of coolant flow, while the remaining two may be used for other information, such as requests for increase or decrease in output power.

`Wait` is useful in delaying one signal for the appearance of another. As an example, consider boxing, packing or filling in the contents in a container, where the contents are delayed until the container is in correct position.

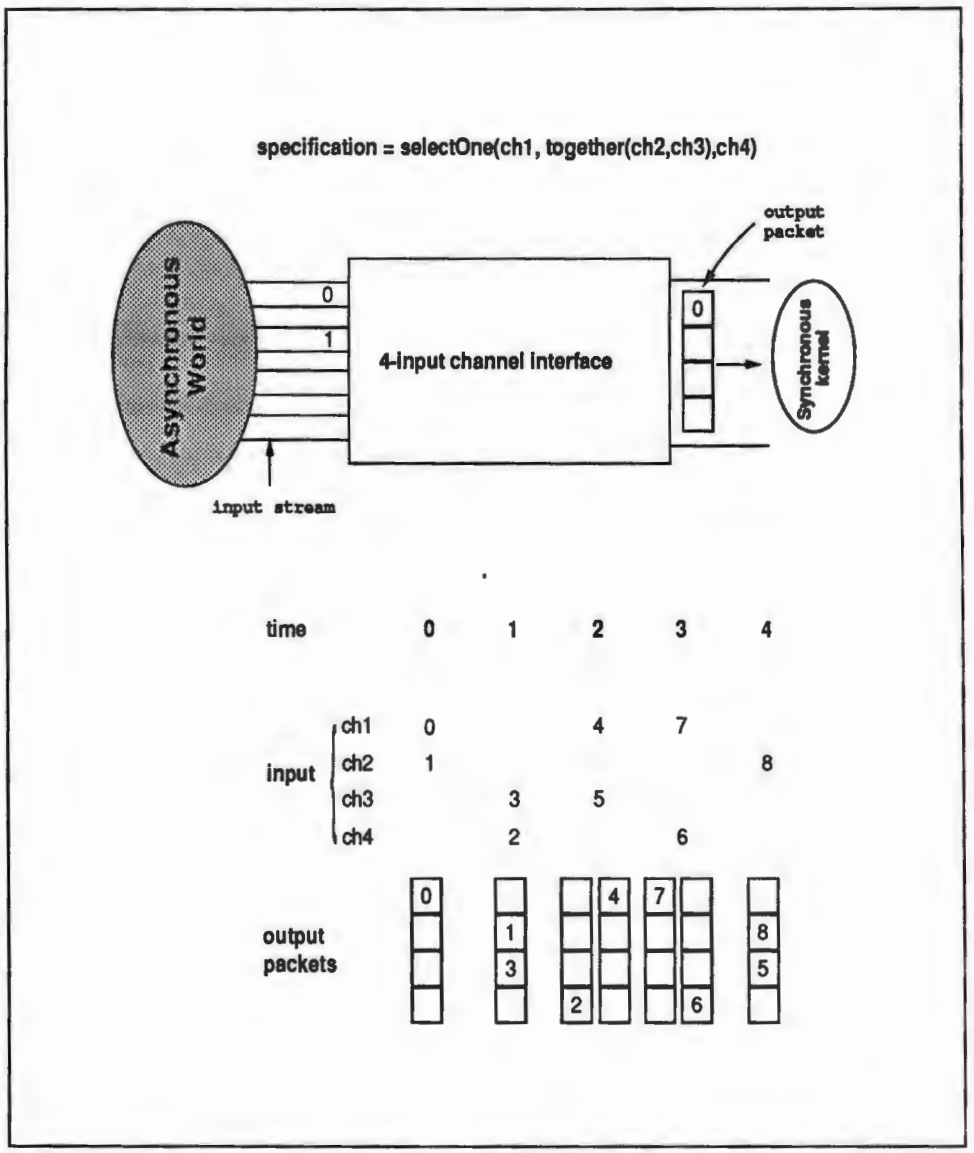


Figure 4.4: A 4-channel interface with sample input and output.

4.5 Summary and Conclusions

APRIL is just one of the many special purpose languages that may be designed for reactive systems from RLUCID. It has been designed keeping in mind, the various functions performed by an interface. In fact, other primitives could be added, depending on the application, as one sees fit.

Thus RLUCID forms the basis for developing a host of languages suitable for real-time systems programming. It is also clear that the salient features of many reactive systems can be described in LUSTRE and APRIL, hence in RLUCID. This is only to be expected as the underlying mathematical model [6] was shown to specify and describe the behavior of any reactive system.

Clear formal semantics of RLUCID prove to be an asset as the conditions under which safety properties hold good, can be laid down precisely. By placing restrictions, as in LUSTRE, one should be able to obtain an efficient implementation, quelling all doubts about the efficiency and hence the applicability of dataflow languages for reactive systems.

Chapter 5

Conclusions

It is customary for the final chapter to gain a broader perspective from the details presented in the earlier chapters, on what has been achieved and where the research is going.

The synchronous dataflow approach takes the idea of removing at every step, the imperative perception and non-determinism from reactive programming. Further, the philosophy that *the execution time is best hidden for safer and correct programs*, is taken to its limit.

There are alternative approaches. One of the most common is simply to employ a great deal of care and discipline in writing programs with explicit timing constraints. This approach may be rescued by developing simple mathematical systems of reasoning about non-deterministic programs.

Meanwhile, the problem of ensuring correctness of computer programs has become more and more acute. This is especially important in application areas such as reactive systems, where technology has made the computer a ubiquitous tool.

In this thesis the three languages, rLUCID, LUSTRE and APRIL,have been

presented. They are all amenable to formal reasoning and transformation. Although their semantics are based on timestamps, these languages do not allow direct access to individual timestamps. Further, concise languages, tailored to applications, could be developed from the basic language RLUCID.

In our effort to bring RLUCID and LUSTRE closer to each other, we have extended LUSTRE with two new operators: `onto` and `truesOf`. `onto` allows a more general language in that a stream can be placed on an arbitrary clock, unlike for `when`, which requires that the clocks of its arguments must be the same.

It is not clear whether the new set of primitives is more appropriate. It is not clear which style is more appropriate: the timestamped streams of RLUCID or the abstract clocks of LUSTRE.

Nevertheless, all of these languages share the synchronous hypothesis, unlike traditional imperative languages. Synchrony is an abstraction, useful in generating efficient code, and simplifies program verification.

We do not pretend that synchronous languages solve all problems. The automaton (kernel) produced responds to the input provided by the interface. As long as no input arrives while processing the current one, we foresee no problems. This implies two assumptions: 1) that all actions are atomic (uninterrupted) and 2) that the computer copes with the input frequency. Execution time can be decreased in some situations by taking advantage of hidden parallelism.

A further issue is the implementation of the languages. RLUCID could be implemented using the LUCID compiler, but the generated code would be too slow for real reactive programming; it would, however, be useful for prototyping. The compiler for LUSTRE generates efficient code, but a separate interface must be developed for every application or, for the same application on different machines. Once a language such as APRIL, is implemented, which we hope to

do shortly, portable interfaces can be developed.

The ideal would be to have one efficiently implementable language to develop complete applications.

Bibliography

[1] S. Abramsky and R. Sykes. SECD-M: A virtual Machine for Applicative Multiprogramming. In *Functional Programming and Computer Architecture*. Springer Verlag 1985. LNCS 201.

[2] J.-L. Bergerand, P. Caspi, N. Halbwachs, and J. A. Plaice. Automatic control systems programming using a real-time declarative language. In *Proc. 4th IFAC/IFIP Symposium (SOCOCO '86)*, Graz, Österreich, 1986.

[3] G. Berry. Real Time Programming: Special purpose or General purpose languages. In *Information Processing 89, Proceedings of the IFIP 11th World Computer Congress*, pp 11-17, 1989.

[4] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. *Seminar on Concurrency* Springer Verlag, 1985.

[5] A. Burns, A. M. Lister and A. J. Wellings. A Review of Ada Tasking. Lecture Notes in Computer Science, vol. 262, Springer-Verlag, 1987.

[6] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behavior of computing systems. *Acta Informatica*, 22:595–627, 1986.

[7] P. Caspi, D. Pilaud, N. Halbwachs, 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.

[8] B. Dasarathy. *Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them.* IEEE Trans. on Software Eng., vol. 11, 80-86, 1985.

[9] T. Gautier, P. Le Guernic and L. Bensnard, *SIGNAL: A Declarative language for Synchronous Programming of Real-Time Systems* Functional Programming Languages and Computer Architecture, September 1987, Gilles Kahn (Ed.), LNCS 274, pp257-277.

[10] N. Halbwachs, A. Longchampt, and D. Pilaud. Describing and designing circuits by means of a synchronous declarative language. In *Proc. IFIP Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, 1986.

[11] N. Halbwachs and D. Pilaud. Use of a real-time declarative Language for systolic array design and simulation. In *Proc. International Workshop on Systolic Arrays*, Oxford, U.K., 1986.

[12] D. Harel and A. Pnueli. On the development of reactive systems: logic and models of concurrent systems. In *Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, pages 477–498, Springer-Verlag, 1987.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* Communications of the ACM, vol. 21, 666-677, 1978.

[14] LOTOS. ISO International Standard, 8807, Geneva, 1989.

[15] ESTELLE. ISO International Standard, 9074, Geneva, 1989.

[16] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress 74*, pages 471–475, Elsevier North-Holland, 1974.

[17] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Communications of the ACM, vol. 21, 558-565, 1978.

[18] P. J. Landin. *The Next 700 Programming Languages.* Communications of the ACM, vol. 9, 157-166, 1966.

[19] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. Signal: a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.

[20] Z. Manna and J. Vuillemin. *Fixpoint Approach to the Theory of Computation.* Communications of the ACM, vol. 15, 528-536, 1972.

[21] Z. Manna. *Introduction to Mathematical Theory of Computation.* McGraw-Hill, 1973.

[22] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.

[23] J. A. Plaice. *Sémantique et compilation de LUSTRE, un langage déclaratif synchrone.* Thèse, Institut National Polytechnique de Grenoble, Grenoble, France, 1988.

[24] B. Russell. *Wisdom of the West.* Doubleday & Company, Garden City, New York, 1959.

[25] W. M. Turski. Time considered irrelevant for real-time systems. *BIT*, 28(3), 1988.

[26] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*, Academic Press, 1985.

[27] S. J. Young. Programming language requirements for process control. in *Real-Time Computer Control.* Peter Peregrinus Ltd., London, U.K., 1984.

Vita

Surname: Vempati | Given Names: Naganjaneya Sarma

Place of Birth: Vijayawada, India | Date of Birth: September 25, 1954

Educational Institutions Attended:

Andhra University, India	1970 to 1975
Indian Institute of Technology, Kanpur, India	1976 to 1981
University of Victoria, Canada	1988 to 1990

Degrees Awarded:

B.Sc.	1973	Andhra University
M.Sc.	1975	Andhra University
Ph.D.	1981	Indian Institute of Technology

Honours and Awards:

Mutyala Venkanna Naidu Gold Medal, 1973.
National Merit Scholarship, 1973-75.

Publications:

About 25 research papers published in
Journal of Optical Society of America,

Journal of Physics B, Atomic and Molecular Physics,
Canadian Journal of Physics,
Journal of Molecular Spectroscopy,
Physica B + C,
and presented in international symposia.

Partial Copyright License

Title of Thesis:

Programming Reactive Systems using Dataflow

Author: [redacted]
Naganjaneya Sarma Vempati
June 26, 1990