Data-flow Synchronous Languages

A. Benveniste, P. Le Guernic * P. Caspi, N. Halbwachs †‡

June 1993

Abstract

In this paper, we present a theory of synchronous data-flow languages. Our theory is supported by both some heuristic analysis of applications and some theoretical investigation of the data-flow paradigm. Our model covers both behavioural and operational aspects, and allows both synchronous and asynchronous styles of implementation for synchronous programs. This model served as a basis to establish the GC *common format for synchronous data-flow languages.*

Keywords: data-flow, concurrency, reactive, real-time, synchrony vs. asynchrony

1 Introduction and Motivations

In this paper we consider systems operating in real-time and handling events as well as "continuous" computations. As an example, let us discuss the case of an aircraft control system. Measurements are received from sensors and processed by the control loops to produce commands as outputs for the actuators: this involves various kinds of numerical computations. Switching from one operating mode to another one can be performed automatically or by the pilot: in both cases, events are received that control the various computations in some discrete event mode. For safety purposes, on-line failure detection and reconfiguration is performed by taking advantage of the redundancies in the aircraft system: actuators and sensor failure detection procedures are numerical computations that produce alarms and various detections which in turn result in reconfiguring the operating mode.

In such real-time complex systems, different kinds of components are involved, namely: computers running suitable application software, electronics, mechanical and other devices, and human operators. Parallelism, concurrency, and reactive aspects [18, 17, 3, 7], should be able to encompass those diverse items within a single framework. This makes the requirements for a related theory of concurrency quite different from those for a theory of concurrency concentrating only on issues of *computing* [27]. In this prospective, we are rather seeking for a

*INRIA-IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, name@irisa.fr

†Laboratoire VERIMAG, Miniparc – ZIRST, 38330 Montbonnot St Martin, France, name@imag.fr

‡This work has been supported, through several contracts in the framework of C^2A–group, by Ministère de la Recherche et de l'Espace, Ministère de l'Industrie (SERICS), and Ministère de la Défense (DRET).

theory of concurrency for *engineering.* Such a theory should concentrate on the description of *systems,* combining both software, and the various plants which interact with this software.

In the past years, the *synchronous* approach has been proposed as a candidate framework for reactive and real-time systems development. It is not our purpose here to further advocate this approach from the user's point of view, we better refer the reader to the special issue [3] and references therein. In this paper, our purpose is to present in a fairly new way the theory supporting the *data-flow* members of the family of synchronous languages (two instances are LUSTRE [16] and SIGNAL [24]).

The paper is organised as follows. In Section 2, our basic requirements for a synchronous data-flow language are derived from the heuristic analysis of two simple examples. Then Section 3 builds on the previous one and proposes a tiny synchronous data-flow language we call μGC. Section 4 is the core of this paper. It is devoted to a thorough theoretical analysis of the several issues that are raised by the data-flow approach in general. In particular, it is argued that some obvious requirements originating from real-time systems (for instance bounded memory and response time) necessary leads the data-flow approach to follow the synchronous point of view. Based on the analysis of this section, our final proposal for a synchronous data-flow model is presented in Section 5 and is used to give the mathematical semantics of our tiny μGC language. Finally, in our conclusion we briefly relate μGC to the actual GC *synchronous data-flow common format,* see [31], which is now shared in particular as an internal format by the two above mentioned LUSTRE and SIGNAL languages, and additional information on GC can be found in [31].

2 Why should Real-Time and Concurrency be Synchronous ? Practical issues

2.1 The simplest example you can imagine

The following equation :

$$\forall n \in \mathbf{N}^+ \ : \ Y_n = X_{n-1} + U_n \ , \ X_o \text{ given initial condition} \tag{1}$$

relates the two input signals (U_n) and (X_n) to the output signal (Y_n), note the one-sample delay on signal X. The following graphical description of this formula, also known as a "signal flow graph", is typically used in control or signal processing communities (`pre` denotes the delay operator):

$$\begin{array}{cccccc} X & \rightarrow & \boxed{\texttt{pre}} & & & \\ & & & \searrow & & \\ & & & & \boxed{+} & \rightarrow \ Y \\ & & & \nearrow & & \\ & & U & & & \end{array} \tag{2}$$

Representations (1) and (2) put emphazis on different aspects. Representation (1) is concerned with values of signals, i.e., "behaviours", and relations that are specified on behaviours via equations. Representations of this form can be easily combined into *systems* of equations in the usual mathematical setting, and this is the way control or signal processing people handle complex models of systems or algorithms for both their description and analysis. Modularity is for free and raises no particular problem since it happens trivially that systems of systems of equations are just systems of equations. On the other hand, representation (1) is not concerned with operational issues : executing (1) is trivial in this simple case, but this no longer holds for systems of such equations where some signals may occur both as input (right handside) of some equations and output (left handside) of others. Equation (1) shall be simply written (we call it "`behaviours`")

```
Y = pre(X) init Xo + U
```

i.e., we abstract over the dummy index "n" in equation (1) , and we introduce the delay operator $X_n \mapsto X_{n-1}$ we denote by " `pre` " [1].

In contrast, representation (2), with its arrows, figures the data flows and their direction of moving, and this clearly has an operational flavour. Such "signal flow graph" representations are also typically combined into "block-diagrams" where some of the signal flow graphs are abstracted as black-boxes with input and output "ports" as only visible items, and arrows link some output to some input ports, thus figuring the data flows between the different boxes. This block-diagram can also be considered as a specification of the possible *schedulings* for executing this computation. But an other interesting issue concerning representation (2) and its operational flavour is that very diverse execution mechanisms can be associated with it, ranging from the rigid "synchronous" one where some physical clock is broadcasted which makes diagram (2) to implement (1) with its rigid timing specification (U_n and Y_n simultaneously available at "instant" n), to the very asynchronous one where token based dataflow mechanisms are used. And this simple remark is indeed wellknown in the signal processing community, see for instance [22] [23]. But such a reasoning is better explained by unfolding diagram (2) into the following infinite directed graph :

$$\begin{array}{ccccccccccccc}
U & : & u_1 & \rightarrow & u_2 & \rightarrow & u_3 & \rightarrow & u_4 & \rightarrow & u_5 & \ldots \\
 & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
Y & : & y_1 & \rightarrow & y_2 & \rightarrow & y_3 & \rightarrow & y_4 & \rightarrow & y_5 & \ldots \\
 & & & \nearrow & & \nearrow & & \nearrow & & \nearrow & & \\
X & : & x_1 & \rightarrow & x_2 & \rightarrow & x_3 & \rightarrow & x_4 & \rightarrow & x_5 & \ldots
\end{array} \tag{3}$$

Now, if this infinite graph is viewed as specifying a *partial order* on its vertices, then the discussion above becomes clear : the common feature of the above two different execution styles (strongly synchronous, and asynchronous) is that they both match this partial order, i.e., they implement some extension (reinforcement) of it. For the moment, diagrams (2) or (3) will be simply denoted by the following two "`dependencies`"

```
     U  --> Y
pre(X) --> Y
```

to mean that, for every n, $u_n \rightarrow y_n$ and $x_{n-1} \rightarrow y_n$ in diagram (3). Note that both this program and the diagram (2) abstract over the time index n. Note also that dependencies of the form $u_{n-1} \rightarrow u_n$ need not to be explicited, since they just express the flow of time.

[1]Control people would have used the notation $z^{-1}X$ instead of `pre(X)` .

Our next idea is to consider that the "`behaviours`" and "`dependencies`" we have introduced are two instances of a single more general object we call *equation* in the sequel. Hence we shall take the liberty to write

```
( Y = pre(X) init Xo + U
|      U  --> Y
| pre(X) --> Y )
```

where "`|`" separates successive equations, and expresses that the whole system of equations (1) and (2) (or, equivalently, (3)) is viewed as a single object. By definition, this system of equations is a specification of all signals `U,X,Y` satisfying jointly these three equations (as customary in mathematics, a system of equations can have no solution, a unique solution, or many solutions). By doing so, we can encode both the behaviours and the schedulings to execute the computations.

At this point, what we have introduced is more than familiar to engineering people, apart from blending equations and block-diagrams within a single framework. Indeed, many industrial tools are available and used, that simulate systems described by diagrams of the form (2), and sometimes even produce executable code out of them. Problems occur, however, for larger applications such as the above discussed aircraft control system : for complex applications, *being bound to a single time index "n" is a severe limitation,* for it makes the effort of handling events, interruptions, and modularity, rather prohibitive. Thus we shall devote the second part of this heuristic discussion to see how such extremely simple ideas can be extended to handle arbitrarily complex systems involving several different time indexes we shall call *clocks* in the sequel.

2.2 Generalizing our equational approach

Clocks. ¿From now on, we shall handle several time indexes simultaneously. Time indexes will be called *clocks. Each signal shall possess its own clock,* i.e., clocks are local rather than global objects as in the preceding subsection. What is specific to the synchronous approach is that different clocks can be related and share some of their instants[2]. The following operations on clocks are introduced :

syntax	mathematical formula	meaning
`L = H default K`	$L = H \cup K$	union of instants
`L = H when K`	$L = H \cap K$	intersection of instants
`L = H whennot K`	$L = H \ominus K$	instants of H that are not instants of K

These operations are illustrated on the following chronogram, where the symbol $\top$ and $\perp$ denote presence and absence respectively ; symbols lying on the same column belong to the

[2]This is in contrast to the more usual model of concurrency via interleaving [25], where different signals interleave but never occur simultaneously.

same event (i.e., are simultaneous):

$$
\begin{array}{rcllllllll}
H & : & \top & \perp & \perp & \top & \top & \top & \perp & \dots \\
K & : & \perp & \top & \top & \top & \perp & \top & \top & \dots \\
H \cup K & : & \top & \top & \top & \top & \top & \top & \top & \dots \\
H \cap K & : & \perp & \perp & \perp & \top & \perp & \top & \perp & \dots \\
L = H \ominus K & : & \top & \perp & \perp & \perp & \top & \perp & \perp & \dots
\end{array}
$$

Using these operators, relations on clocks can be stated. For instance

```
K = K default H
```

expresses that every instant of `H` is an instant of `K`. Then,

```
clock(X)
```

shall denote the clock of the signal `X`. Finally, new clocks are created using the operator

```
true(B)
```

where `B` is a boolean signal, by keeping those instants of `clock(B)` where `B` is true, this is illustrated on the following diagram:

$$
\begin{array}{rcllllllll}
B & : & \mathit{false} & \perp & \perp & \mathit{true} & \mathit{true} & \mathit{false} & \perp & \dots \\
\texttt{clock}(B) & : & \top & \perp & \perp & \top & \top & \top & \perp & \dots \\
\texttt{true}(B) & : & \perp & \perp & \perp & \top & \top & \perp & \perp & \dots
\end{array}
$$

Thus the following properties hold:

```
true(B) default true (not B) = clock(B)
true(B) when    true (not B) = emptyclock
```

where " `emptyclock` " denotes the clock with no instant.

Equations. Clocks shall be used to specify *when* equations such as (1) hold. For instance,

```
C = A+B  at H
```

where `H` is a clock, states that, at every instant of `H`, equation `C = A+B` should be satisfied. This is illustrated on the following chronogram:

$$
\begin{array}{cclllllllll}
H & : & \perp & \perp & \perp & \top & \perp & \perp & \top & \perp & \dots \\
C & : & & & & c_? = a_2 + b_3 & & & c_? = a_4 + b_6 & & \dots \\
A & : & a_1 & \perp & \perp & a_2 & \perp & a_3 & a_4 & a_5 & \dots \\
B & : & \perp & b_1 & b_2 & b_3 & b_4 & b_5 & b_6 & \perp & \dots
\end{array}
$$

Note that we do not include $\perp$ in the domain of the addition "+", i.e., for any x, "$x + \perp$" is not defined. Consequently this statement requires that *both* `A` *and* `B` *be present when* `H` *occurs.* Also `C` is not completely defined via this equation unless it is furthermore stated that `clock(C) = H`. This is illustrated, on the one hand, by the "?" subscript indicating that other `C` samples may have occurred, and on the other hand, by the fact that no "$\perp$" symbol has been made explicit in this chronogram.

The delay "pre". Since clocks are local to signals, so are " pre " operators: pre(X) delays X by one time unit, as measured by the clock of X itself:

$$\begin{array}{rcccccccccc} Y & : & \perp & y_1 & y_2 & \perp & y_3 & y_4 & y_5 & \perp & \dots \\ X & : & x_1 & \perp & x_2 & x_3 & x_4 & \perp & \perp & x_5 & \dots \\ \texttt{pre}(X)\ \texttt{init}\ x_o & : & x_o & \perp & x_1 & x_2 & x_3 & \perp & \perp & x_4 & \dots \end{array}$$

The above diagram shows X and pre(X) together with another signal Y possessing a different clock. This example illustrates that *stuttering invariance* (see [21] and [25], pp. 260–261) is automatically guaranteed, i.e., delaying X does not depend on the environment of X . See also [5] for a more extensive discussion of this.

Dependencies and scheduling. Clocks shall also be used to specify *when* a dependency holds. For instance,

```
U --> Y  at H
```

specifies that Y cannot be produced before U at each instant of clock H . Unlike for the case of equation C = A+B at H , *we do not require that* U *and* Y *be present when clock* H *is present.*

2.3 Discussing an example

The following program,

```
( clock(X) = clock(U)
| clock(Y) = clock(X) default clock(V)
| Y = pre(X) init Xo + U  at (clock(X) whennot clock(V))
| Y = V  at clock(V) )
```

describes the combination of equation (1) with a resetting of Y by V when V occurs, thus we call it RESET . Note that the occurrence of V is preemptive. The following diagram shows a behaviour of this program:

$$\begin{array}{rcccccccc} V & : & \perp & \perp & v_1 & \perp & v_2 & \perp & \dots \\ Y & : & y_1 = x_o + u_1 & y_2 = x_1 + u_2 & y_3 = v_1 & y_4 = x_2 + u_3 & y_5 = v_2 & y_6 = x_4 + u_5 & \dots \\ U & : & u_1 & u_2 & \perp & u_3 & u_4 & u_5 & \dots \\ X & : & x_1 & x_2 & \perp & x_3 & x_4 & x_5 & \dots \\ \texttt{pre}(X)\ \texttt{init}\ x_o & : & x_o & x_1 & \perp & x_2 & x_3 & x_4 & \dots \end{array}$$

Note that the clocks of X and V are not related, i.e., any interleaving of these signals may occur. Dependencies are naturally associated with the last two equations of this program, namely

```
( pre(X) --> Y  at (clock(X) whennot clock(V))
| U      --> Y  at (clock(X) whennot clock(V))
| V      --> Y  at clock(V) )
```

and the resulting behaviour and scheduling is illustrated in the following diagram :

$$
\begin{array}{rccccccccc}
V & : & \perp & \perp & v_1 & \perp & v_2 & \perp & \ldots \\
 & & & & \downarrow & & \downarrow & & \ldots \\
Y & : & y_1 = x_o + u_1 & y_2 = x_1 + u_2 & y_3 = v_1 & y_4 = x_2 + u_3 & y_5 = v_2 & y_6 = x_4 + u_5 & \ldots \\
 & & \uparrow & \uparrow & & \uparrow & & \uparrow & \ldots \\
U & : & u_1 & u_2 & \perp & u_3 & u_4 & u_5 & \ldots \\
X & : & x_1 & x_2 & \perp & x_3 & x_4 & x_5 & \ldots \\
\texttt{pre}(X)\texttt{init}\ x_o & : & x_o & x_1 & \perp & x_2 & x_3 & x_4 & \ldots \\
 & & \downarrow & \downarrow & & \downarrow & & \downarrow & \ldots \\
Y & : & y_1 = x_o + u_1 & y_2 = x_1 + u_2 & y_3 = v_1 & y_4 = x_2 + u_3 & y_5 = v_2 & y_6 = x_4 + u_5 & \ldots
\end{array}
$$

The horizontal arrows, which figure causality due to the flow of time for each signal in diagram (3), are not mentioned here ; such dependencies are implicit. Note that there is no oblique arrow relating `X` to `pre(X)` , compare with diagram (3), this point will be further discussed later on.

Next, combine the above program with the following equations :

```
( clock(B) = clock(X) = clock(U) = clock(Y)
| X = Y   at clock(Y)
| U = -1  at clock(U)
| B = ((pre(X) init Xo) <= 0)  at clock(X)
| clock(V) = true(B) )
```

This makes `Y` to be the output of a decreasing counter with reset `V` . But the reset signal `V` itself is synchronised on some condition involving `X` (or `Y` equivalently). We can rewrite the resulting program equivalently as follows, call it `MUX` :

```
( clock(X) = clock(B)
| clock(V) = true(B)
| X = (pre(X) init Xo) - 1  at (clock(X) whennot clock(V))
| X = V  at clock(V)
| B = ((pre(X) init Xo) <= 0)  at clock(X) )
```

where `<=` means "less than or equal to". A sample legal behaviour of this program is depicted next :

$$
\begin{array}{rccccccccc}
\texttt{V} & : & 3 & \perp & \perp & \perp & 0 & 2 & \perp & \perp & \ldots \\
\texttt{X} & : & 3 & 2 & 1 & 0 & 0 & 2 & 1 & 0 & \ldots \\
\texttt{pre(X)} & : & 0 & 3 & 2 & 1 & 0 & 0 & 2 & 1 & \ldots \\
\texttt{pre(X)} \leq \texttt{0} & : & t & f & f & f & t & t & f & f & \ldots \\
\texttt{clock(V)} & : & \top & \perp & \perp & \perp & \top & \top & \perp & \perp & \ldots
\end{array}
$$

This program performs "upsampling" at a variable rate. Each time `V` is read (a nonnegative integer), `V` additional "ticks" are delivered before the next present occurrence of `V` is read. This `MUX` program is the basic building block for variable rate transmultiplexing : assuming `V` data are received at once (i.e., via, say, spatial multiplexing), the `MUX` program provides the adequate temporal multiplexing to rearrange the data such that a single one is emitted at once.

How dependencies should be introduced is now less obvious, we shall discuss this later. It is however clear that the two programs `RESET` and `MUX` very much differ in their control structure, namely :

- Inputs of `RESET` program are
  1. the clocks of `X` and `V` , and
  2. the values of `X,U,V` .

  Such a control structure makes this program fully *passive,* i.e., the control is driven by inputs and the program accepts all inputs and reacts to them.

- Inputs of `MUX` program are
  1. the clock of `X` (note that the values of `X` are the output of the program), and
  2. the values of `V` .

  Such a control structure makes this program *active,* i.e., running according to a demand driven mode.

Obviously, programs of mixed passive/active mode can be easily written. This also illustrates how combining a program with another one can drastically change its control and scheduling, this makes separate compilation and distributed code generation a subtle task, see [24] for a discussion of related issues.

Remark : **how to associate dependencies with the delay " `pre` "?** As depicted in diagram (3), the adequate dependencies associated with the `pre` operator are the following :

$$\begin{array}{rccccccccccc}
X : & x_1 & & x_2 & & x_3 & & x_4 & & x_5 & \dots \\
 & & \searrow & & \searrow & & \searrow & & \searrow & & \\
Y = \mathtt{pre}(X)\ \mathtt{init}\ x_o : & y_1 = x_0 & & y_2 = x_1 & & y_3 = x_2 & & y_4 = x_3 & & y_5 = x_4 & \dots
\end{array} \tag{4}$$

i.e., they involve "oblique" arrows. But "oblique" arrows do not belong to the class of equations we have introduced so far, since expressions of the form `X --> Y at H` specify only "vertical" arrows, i.e., dependencies that are local to a given instant. The idea to overcome this is very simple : replace $\searrow$ by $\downarrow\!\longrightarrow\!\downarrow$, i.e., we replace diagram (4) above by the following one, where columns again collect occurrences belonging to the same instant :

$$\begin{array}{rccccccccccc}
X : & x_1 & & x_2 & & x_3 & & x_4 & & x_5 & \dots \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
\xi : & x_o, x_1 & \rightarrow & x_1, x_2 & \rightarrow & x_2, x_3 & \rightarrow & x_3, x_4 & \rightarrow & x_4, x_5 & \dots \\
 & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & \\
Y = \mathtt{pre}(X)\ \mathtt{init}\ x_o : & y_1 = x_0 & & y_2 = x_1 & & y_3 = x_2 & & y_4 = x_3 & & y_5 = x_4 & \dots
\end{array}$$

where the new notion of a *state* ξ has been introduced. Sates are characterised by the following property (see Section 5 for a formal definition). At each instant n, state ξ has two occurrences, namely $\xi_{n,1} = x_{n-1}$ and $\xi_{n,2} = x_n$. Thus, in contrast to signals, states can have several totally ordered occurrences per instant. Also, it is a property of states that

$$\xi_{n,\text{first}} = \xi_{n-1,\text{last}}$$

must hold. By doing so, only vertical arrows remain, and the same dependencies can be used to specify them as before.

DISCUSSION. At this point we have provided some practical motivations for our dataflow synchronous approach to concurrency. Its main features are summarized now :

- We view synchronous systems as "systems of dynamical equations", i.e., constraints on the behaviours of signal flows. Different time indexes or *clocks* may occur within the considered synchronous system, in fact clocks are local to signals. It is part of the constraints on behaviours that different clocks be related, and this is the particular feature of the synchronous approach. This approach has several advantages, among them the main one is a clean approach to concurrency and modularity (our theory will not need a fine taxonomy of different kinds of equivalence nor a deep study of bisimulation concepts).
- Operational semantics would be provided based on partial orders or dependencies. Dependencies specify legal schedulings. In doing so, we do not preclude from using such and such particular implementation consistent with the specified partial order.
- We want to handle behavioural and operational semantics separately or to mix them, as our convenience. Equivalently, from the engineer point of view, we want to mix equations and signal flow graphs or block-diagrams. In doing so, we shall be able to combine properties or specifications, and executable programs within a single framework.

In the section to follow we shall build on our heuristic analysis and requirements of this section and propose a practical "Data-flow synchronous" language. Then we further investigate from the theoretical point of view why data-flow should be synchronous, and explain how a theoretical analysis naturally leads us to the same conclusions as our previous heuristic analysis. By the way, our proposed language will be supported by both practical and theoretical considerations. We now move to the first item.

3 A proposal for a practical Data-flow Synchronous language

In this section, we build on our heuristic analysis and requirements of the previous section. We introduce a tiny language we call μGC, which will provide a practical answer to the questions and requirements we raised before. Also, as will be explained later, it happens μGC is a direct concrete instanciation of our mathematical model. These features make μGC very clean from the theoretical point of view, but at the same time this makes it slightly verbose. Based on μGC, the GC language [31] has been defined. GC is essentially a way of making μGC less verbose. The GC language is now taken as the *common format for data-flow synchronous languages,* see [31], and is currently implemented. Theoretical bases for μGC will be provided in the subsequent sections.

3.1 The tiny language μGC

¿From our analysis of the previous section, we deduce the following features that our tiny language should possess:

Clocks are used to handle several time index sets and to relate them. Clocks are subsequences of "events" or "instants". Also, relations between clocks will be necessary. In what follows, " `base` " shall denote a clock faster than any other one, see section 5.5 for a precise definition.

Systems of equations should be described, where relations on signals would hold at particular *clocks.* Primitive relations will take the form " `relation at H` " where `relation` denotes any relation linking signals, and `H` is the clock at which the referred relation holds.

Block-diagrams or signal-flow graphs will be specified as *systems of dependencies* (in fact we shall make use of *preorders* instead) which may also hold at particular clocks. Primitive orders will take the form " `X --> Y at H` " to express that `Y` *cannot be produced before* `X` *at those instants of clock* `H` [3].

States are introduced to handle properly dependencies that across instants. States will be used to encode the `pre` delay operator, which will not be primitive any more. Also, states are useful to insert in a mathematically sound way imperative programs within dataflow, this is extensively used in the GC common format for synchronous languages, see [31].

Based on these remarks, here is an informal description of what μGC consists of:

```
H = K when    L            %
H = K whennot L            % relating clocks
H = K default L            %

H = clock(X)               %
H = true(B)                % creating clocks

Y = f(A,B) at H            % relating flows

state   : Xi               % declaration of local state Xi
IN-ACT  : Xi = X  at H     % writing value of signal X in state Xi
OUT-ACT : Y  = Xi at H     % emitting value of state Xi on signal Y

FOO --> FOO' at H          % preorder specification

PROG_1 | PROG_2            % composing equations or programs
```

In this informal description, keywords of the language are written in lower cases.

[3] This is the reason to refer to preorders instead of partial orders, since partial orders would correspond to a different interpretation, namely `X` *must be provided before* `Y` *at those instants of clock* `H`. And we shall explain later in the theoretical analysis why the former solution — i.e., relying on preorders — is preferred to this one.

- The “ `when, default, whennot` ” operators on *clocks* respectively correspond to the infimum of clocks (intersection of instants of presence), supremum of clocks (union of instants of presence), and complement of `L` in `K` (instants where `K` is present and `L` absent).

- The expression `clock(X)` denotes the clock of signal `X`, i.e., the subset of instants where signal `X` is present. Then, for `B` a boolean signal, `true(B)` is the clock of the *true* occurrences of `B`, i.e., `true(B)` is present if and only if `B` is present and takes the value *true*.

- The expression `Y = f(A,B) at H` , where `H` is a clock, relates *signals* `A,B,Y` through `Y = f(A,B)` at those instants belonging to clock `H` . Here `f` is a “usual” function and $\perp$ is not incuded in its domain. Hence `A` and `B` must be present when `H` is.

- *States* can be declared, that are local to each program. States can be written into or read from signals ; writings and readings are referred to as “*actions*” that have names (here `IN-ACT` and `OUT-ACT` are examples of such names). Different actions sharing the same state may occur within a single instant, hence states may have multiple occurrences within a single instant and this is in contrast with signals where single occurrence per instant is required. When multiple occurrence of some state occurs, its semantics is that of nondeterministic interleaving, unless ordering is explicitly specified as indicated next. Between two successive writings, states keep constant. Hence, states are reminiscent of classical (local) variables. They will be provided later on with a synchronous data-flow semantics, however.

- The *preorder* `FOO --> FOO' at H` specifies that, at each instant of clock `H` , `FOO'` cannot occur before `FOO` , where `FOO` and `FOO'` denote signals or actions. In constrast to relations on flows, it may happen that `FOO` or `FOO'` be absent when `H` is present. In particular, ordering actions is the way to remove nondeterminism in the case of multiply occurring states. Again, preorders will be provided later on with a synchronous data-flow semantics.

The operational semantics of a μGC program, i.e., how it should be executed, entirely relies on the preorders. Other instructions of μGC specify only constraints on behaviours, i.e., they play no role in operational issues. This feature of μGC makes it extremely clean, but at the same time quite verbose as we shall see. For the formalists, a syntax for μGC is given in the appendix in Table 2.

EXAMPLE. We claim that the μGC program of figure 1 corresponds to the `pre` operator used in (2), where `IN` denotes the given signal, and `OUT` its delayed version. In order to discuss this program informally, we first need to clarify what we mean by “actions”, since actions are non standard players in our game. Take the above μGC program. There is a single state, namely `Xi` , and this state is shared by the two actions `REFRESH_STATE, OUTPUT` , that have the same clock `H` . Hence, at each instant n of `H` , *two* totally ordered occurrences $\xi_{n,1}, \xi_{n,2}$ of the state `Xi` are created in our model. Since the ordering `OUTPUT --> REFRESH_STATE at H` is

```
metagc PRE                                  % program name

  act REFRESH_STATE, OUTPUT                 % declaration of actions
  state Xi init V                           % declaration of states

    ( ( clock(IN)            = H            % clock equations
      | clock(OUT)           = H
      | clock(REFRESH_STATE) = H
      | clock(OUTPUT)        = H )

    | ( OUTPUT --> OUT           at H       % preorders
      | OUTPUT --> REFRESH_STATE at H
      | IN     --> REFRESH_STATE at H
      | H      --> IN            at base
      | H      --> OUT           at base )

    | ( REFRESH_STATE :  Xi = IN at H       % actions
      | OUTPUT        : OUT = Xi at H ) )

end PRE
```

Figure 1: μGC *program of the* `pre` *operator*

specified, occurrences $\xi_{n,1}$ and $\xi_{n,2}$ are attached to actions `OUTPUT` and `REFRESH_STATE` respectively : the value of $\xi_{n,1}$ is delivered on signal `OUT` , and then $\xi_{n,2}$ takes the value provided by signal `IN` . Let us briefly check that we really recover the `pre` operator used in (2). Dependencies and relations on values for this program are informally depicted in the following diagram, where x_n, y_n, and $\xi_{n,1}, \xi_{n,2}$ denote the successive values of input `IN` , output `OUT` , and state `X` respectively ($\xi_{1,1}$ is a given initial condition) :

$$
\begin{array}{lcccccccc}
\texttt{IN :} & & & x_1 & \rightarrow & \rightarrow & \rightarrow & x_2 & \rightarrow & \rightarrow & \ldots \\
 & & & \downarrow & & & & \downarrow & & & \ldots \\
\texttt{Xi :} & \xi_{1,1} & \rightarrow & \xi_{1,2} = x_1 & \rightarrow & \xi_{2,1} = \xi_{1,2} & \rightarrow & \xi_{2,2} = x_2 & \rightarrow & \xi_{3,1} = \xi_{2,2} & \ldots \\
 & \downarrow & & & & \downarrow & & & & \downarrow & \ldots \\
\texttt{OUT :} & y_1 = \xi_{1,1} & \rightarrow & \rightarrow & \rightarrow & y_2 = \xi_{2,1} & \rightarrow & \rightarrow & \rightarrow & y_3 = \xi_{3,1} & \ldots
\end{array}
$$

Erasing ξ in this diagram yields

$$
\begin{array}{lcccccccc}
\texttt{IN :} & x_1 & \rightarrow & x_2 & \rightarrow & x_3 & \rightarrow & \ldots \\
 & & \searrow & & \searrow & & \searrow & \ldots \\
\texttt{OUT :} & y_1 = \xi_{1,1} & \rightarrow & y_2 = x_1 & \rightarrow & y_3 = x_2 & \rightarrow & \ldots
\end{array}
$$

as desired for the `pre` operator used in (2). This yields an *operational* semantics for the `pre` . If a *behavioural* semantics only is wanted, just remove in the μGC program `PRE` above the following equations :

```
    | ( OUTPUT --> OUT           at H            % preorders
```

μGC equation		induced preorder
	(i)	`clock(X) --> X at base`
`H = K clockop L`	(ii)	`(K,L) --> H at base`
`X --> Y at H`	(iii)	`H --> Y at base`
`H = true(B)` `K = clock(B)`	(iv)	`B --> H at K`
`Y = f(A,B) at H`	(v)	`(A,B) --> Y at H`
`IN-ACT: Xi = X at H`	(vi)	`X --> IN-ACT at H`
`OUT-ACT: X = exp at H`	(vii)	`OUT-ACT --> X at H`

Table 1: *Rules for inducing preorders in μGC*

```
| IN     --> REFRESH_STATE at H
| H      --> IN            at base
| H      --> OUT           at base )
```

It is important to note, however, that the following preorder should *not* be removed

```
| OUTPUT --> REFRESH_STATE at H
```

since the two different occurrences of state ξ must be ordered (read from before write in memory), so it is really part of behavioural semantics, and this is a particular feature of states as opposed to signals. In Section 5.5 we shall provide a rigorous mathematical semantics for the tiny language μGC.

3.2 Issues of operational semantics

As we have seen in discussing the `PRE` example, operational semantics in μGC is tightly related to preorder specification. The idea is that, in order to get the operational form of a μGC program, it is only required to extend the preorders. Obviously this must be performed in some appropriate way, and the basis for doing so is Table 1 where " `clockop` " denotes one of the clock operators `when, whennot, default` . Basically, what Table 1 performs is adding to each μGC equation some adequate preorder intended to specify an appropriate execution scheme for it :

(i) `X` cannot be produced before its clock is known ;

(ii) to know how `K` and `L` are combined to form `H` , we need to know an environment where the absence of each of these clocks can be referred to; this is the reason for using the special clock " `base` ", which is more frequent than any clock;

(iii) in order to enforce this dependency properly, it is needed to know when `H` is present or absent, hence the reference to " `base` ";

(iv) it is enough to know `B` in order to produce its *true* occurrences;

(v) selfexplanatory;

(vi) it is needed to know `X` before performing the considered action;

(vii) the considered action must be performed prior knowing `X` .

Based on these rules, the following method can be followed to derive a suitable operational form for a μGC program.

1. Start with the μGC program encoding the desired *behaviours*, we call it `SPEC` .
2. Apply iteratively rules of Table 1 until a fixpoint is reached. Note that the rules of this table are bound to the *syntax* of the `SPEC` program, not to its behavioural semantics. This yields a new μGC program with the same behaviour and additional preorders. It is a candidate *operational semantics* associated with `SPEC` , we call it `EXEC` .
3. Check whether the resulting program `EXEC` is indeed *executable*; what this statement means and how this can be achieved is clarified later.
4. If `EXEC` was not executable, return to `SPEC` , and modify its syntactic form while keeping its behavioural semantics unchanged, until the new associated `EXEC` program becomes executable.

The rules of Table 1 are justified by the following criterion for executability. In the following statement, we call " `empty_clock` " the clock with no present occurrence.

Definition 1 (executable μGC program) *A μGC program* `P` *is called* **executable** *if it satisfies the following conditions:*

1. `P` *is not modified by applying any rule of Table 1 (i.e., all induced preorders have yet been added to the program.)*
2. *For any circuit in* `P` *, of the form*

```
( X_0     --> X_1 at H_1
| X_1     --> X_2 at H_2
| .......
| X_(p-1) --> X_0 at H_0 )
```

we have

```
H_1 when H_2 when ... when H_0  =  empty_clock
```

i.e., `P` *is "circuit free".*

3. *Clocks are uniquely defined, i.e.,* `P` *does not involve double definitions of clocks of the form*

   ```
   ( H = exp_1
   | H = exp_2 )
   ```

   with `exp_1` $\neq$ `exp_2` *.*

4. *For any double definition of signals in* `P` *, of the form*

   ```
   ( X = exp_1 at H_1
   | X = exp_2 at H_2 )
   ```

   we have

   ```
   H_1 when H_2 =  empty_clock
   ```

   i.e., signals are uniquely defined.

5. *Consider the reading and writing actions sharing a given state. Then, at each instant, each present occurrence of a considered reading action possesses, among the above mentioned writing actions, a nearest predecessor which is uniquely specified. In other words, no nondeterminism result from actions sharing the same state.*

If program `P` *is executable, its inputs are found as follows:*

- *source nodes of the "* `-->` *" graph yield* **input clocks** *;*
- *signals not appearing on the left hand side of flow equations yield* **input values.**

See the illustration of this on the detailed discussion of the `MUX` example to follow. It can be shown [31] that executable programs in the above sense are *deterministic input/output functions* for some suitable input/output partition of the signals. This justifies the name of "executable" for programs satisfying the conditions of Definition 1, and this also justifies the rules of Table 1.

Obviously, the conditions of Definition 1 provide only a sufficient condition for input/output functionality. As a matter of fact, these conditions are undecidable in general. The reason for this is the use of operator " `true(B)` " which maps boolean signals into clocks, since in turn boolean `B` itself may be the evaluation of some predicate on signals of any (hence possibly infinite) type. Consequently, only abstractions of these conditions are implemented in practice, here are the most typical ones:

- Conditions 3 and 4 of Definition 1 are reinforced as follows: there is neither circuit nor double definition.

```
metagc BEHAV_MUX

  action REFRESH_STATE, OUTPUT
  state Xi init 0

    ( ( clock(X) = clock(Y) = clock(B) = H
      | clock(REFRESH_STATE) = clock(OUTPUT) = H
      | K = true(B)
      |
      | clock(U) = K
      | L = H whennot K )

    | ( OUTPUT --> REFRESH_STATE at H )

    | ( REFRESH_STATE :  Xi = X at H
      | OUTPUT        :  Y  = Xi at H ) )

    | ( B = (Y <= 0) at H
      | X = U at K
      | X = (Y-1) at L ) )

end BEHAV_MUX
```

Figure 2: *The* `MUX` *program: behaviours*

- Conditions 3 and 4 of Definition 1 are kept, but we use the abstraction that the present values of any boolean signal `B` resulting from the evaluation of a predicate on non-boolean types is taken as an input (i.e., the two values *true* and *false* are equally possible as soon as `B` is present).

3.3 Analysing the `MUX` program

To have a complete analysis of the `MUX` program, we first rewrite it in μGC. Thus the `pre` operator is expanded, states and actions are introduced instead. The behaviours of `MUX` are specified by the program depicted in figure 2. Applying the rules of Table 1 until a fixpoint is reached yields the operational form of the `MUX` program. This is shown in figure 3. It is easily verified that this program is executable since their is no cycle and no double definition, and actions are properly ordered. Also, inputs are easily found :

input clocks are source nodes of the dependency graph, i.e., symbols appearing only on the left hand side of the dependencies `* --> * at *` ; in the considered example, only clock `H` has this property ;

input values are obtained by selecting the signals that occur only on the right hand side of the equations `* = * at *` ; in the considered example, only signal `U` has this property.

```
metagc OPER_MUX

  action REFRESH_STATE, OUTPUT
  state Xi init 0

    ( ( clock(X) = clock(Y) = clock(B) = H
      | clock(REFRESH_STATE) = clock(OUTPUT) = H
      | K = true(B)
      | clock(U) = K
      | L = H whennot K )

    | ( H --> (X,Y,B) at base        <== input clock
      | K --> U       at base
      | (H,K) --> L   at base
      | B --> K       at H
      | Y --> B at H
      | U --> X at K
      | Y --> X at L
      | X --> REFRESH_STATE at H
      | OUTPUT --> Y at H
      | OUTPUT --> REFRESH_STATE at H
      | H --> K at base
      | K --> X at base
      | L --> X at base )

    | ( REFRESH_STATE  :  Xi = X at H
      | OUTPUT :  Y  = Xi at H ) )

    | ( B = (Y <= 0) at H
      | X = U at K                   <== input value
      | X = (Y-1) at L ) )

end OPER_MUX
```

Figure 3: *The* `MUX` *program: operational form*

This operational form of program `MUX` makes a deep use of its "synchronous" nature, i.e., of its ability to refer explicitly to *instants* where some signals are *present* while other ones are *absent.* Now, if no explicit reference to "absence" were needed, we would be in the same situation as depicted in diagram (3), namely an asynchronous interpretation is also well suited (this is fully clarified in the theoretical section to follow). How can we remove from `OPER_MUX` any explicit use of "absence"? Just:

1. Compute the transitive closure of the directed graph specified by the dependencies `-->`,
2. Avoid using those "`clockop`" operator on clocks (cf. table 1) for which explicit reference to absence is needed. Thus the only acceptable operator on clocks is `true(.)`. In particular we can implement equivalently the `whennot` operator by producing the boolean signal `C = not B`, and then by noting in particular that `L = true(C)`.
3. In the "`FOO --> FOO' at H`" resulting expressions, remove from `H` those instants where `FOO` is absent.

This yields the program shown in figure 4. Thus we have provided an *asynchronous executable form* of our GC program `MUX`. Note that `base` is not used any more. In addition, in the preorders, the left hand side of "`... --> ... at H`" is never absent when `H` is present: this characterizes desynchronised executable programs. Obviously, it is not needed to remove all explicit references to absence, partial removal is also possible. This technique will be mathematically justified in the sections to follow.

DISCUSSION. Note the following:

- Properties of the clocks and booleans have been used to modify the syntax of our μGC programs while keeping their behaviour identical.
- Additional preorders that are required for the operational semantics have been included *at the final stage,* i.e., after submodules have been composed and simplified. In particular, note that, in program `MUX` of Figure 3, preorders point from `X` toward the clock `K` of `U`. This is in contrast to the scheduling of the `RESET` example, where dependencies are in the opposite direction — recall that `MUX` is just obtained by setting `RESET` in a proper environment specified by additional equations. Thus producing adequate preorders depends on the environment, and this obviously makes separate compilation a difficult topic [4].
- Partial or total desynchronisation has been performed after proper operational semantics is constructed.
- The notion of an "executable program" has been used, see Definition 1.

Based on the μGC tiny language, the GC common format for synchronous languages [31] has been defined as a way to make μGC less verbose. In particular, it is not needed in GC to

[4]Solutions exist, however, but presenting them is beyond the topic of this article, see [24].

```
metagc ASYNCH_MUX

  action REFRESH_STATE, OUTPUT
  state Xi init 0

    ( ( clock(X) = clock(Y) = clock(B) = clock(C) = H
      | clock(REFRESH_STATE) = clock(OUTPUT) = H
      | K = true(B)
      | L = true(C)
      | clock(U) = K )

    | ( H --> (X,Y,B) at H           <== input clock
      | K --> U at H
      | B --> K at H
      | C --> L at H
      | Y --> B at H
      | U --> X at K
      | Y --> X at L
      | X --> REFRESH_STATE at H
      | OUTPUT --> Y at H
      | OUTPUT --> REFRESH_STATE at H
      | H --> K at H
      | K --> X at H
      | L --> X at H )

    | ( REFRESH_STATE :  Xi = X at H
      | OUTPUT :  Y = Xi at H ) )

    | ( B = (Y <= 0) at H
      | C = not B at H
      | X = U at K                   <== input value
      | X = (Y-1) at L ) )

end ASYNCH_MUX
```

Figure 4: *The* `MUX` *program: desynchronised form.*

explicitly apply the rules of Table 1 to get the operational form of a program: different fields (*definitions of signals,* and *properties of signals*) are introduced in the body of programs to indicate whether rules of Table 1 apply to a considered equation or not. On the other hand, a richer set of (redundant) primitives are used in GC. We now move to the promised theoretical analysis.

4 Why should Data-flow be Synchronous ? Theoretical issues

In this section, we investigate the data-flow approach from the theoretical point of view. As we have widely discussed, the data-flow paradigm provides a natural framework for control and signal processing, and more generally, for engineering people. Many attempts have been made to use a data-flow approach in Computer Science [20, 1, 13], yet not successfully [5], for two major reasons:

- the composition of bounded memory data flow systems may not be bounded memory, see [11],
- the composition of equivalent non deterministic data flow systems may not yield equivalent systems, see [10] [6].

So far, several solutions have been brought to both problems, but it does not seem that an effective implementation of these solutions have been provided as actual practical programming tools. Now Control and Signal processing systems are reactive ones, and intail a need for bounded memory and reaction time. Our purpose here is to investigate theoretically why removing the above listed drawbacks and answering the request of bounded memory and reaction time naturally leads to the synchronous approach. We first formally state our requirements.

4.1 Problem statement

Notation. In the sequel of the paper, we shall handle different monoids and consider associated regular expressions. The following notations will be generically used. The composition of the considered monoid is written multiplicatively, and is denoted by a dot ".", and we denote by ε its neutral element. The set theoretic union is written additively. Finally, for A a subset of this monoid, $A^\star =_{\text{def}} \{\varepsilon\} + A + A.A + \dots$ is the usual star operation.

Requirements

Assume we want to design a programming language which allows one to define:

- Flow names or "*Signals*" `x` and types $D_{\mathtt{x}}$;

[5] except for SISAL [13] whose determinism and special control structures lead to avoid both problems.
[6] Though this is not particular to data flow programmig: see for instance CCS [26]

- "Equations", i.e., primitive relations over flows [7]

$$\mathcal{S}_{\mathrm{primitive}}(\mathbf{X}) \subseteq \prod_{\mathtt{x}\in\mathbf{X}} D^{\star}_{\mathtt{x}},$$

where $\mathbf{X}$ is a finite set of signals and $[\ldots]^{\star}$ is the usual star operation ; note that $\prod_{\mathtt{x}\in\mathbf{X}} D^{\star}_{\mathtt{x}}$ involves t-uples of sequences of data, not sequences of t-uples of data.

- "Systems of equations", properly formalised as a parallel composition construct :

$$\mathcal{S}_1(\mathbf{X_1}) \mid \mathcal{S}_2(\mathbf{X_2}) \quad = \quad \Phi^{-1}_{\mathbf{X_1}\cup\mathbf{X_2}\mapsto\mathbf{X_1}}(\mathcal{S}_1(\mathbf{X_1})) \bigcap \Phi^{-1}_{\mathbf{X_1}\cup\mathbf{X_2}\mapsto\mathbf{X_2}}(\mathcal{S}_2(\mathbf{X_2})) \qquad (5)$$

where

$$\mathcal{S}_i(\mathbf{X_i}) \subseteq \prod_{\mathtt{x}\in\mathbf{X_i}} D^{\star}_{\mathtt{x}}$$

and $\Phi_{\mathbf{X}\mapsto\mathbf{X}'}$, for $\mathbf{X}' \subseteq \mathbf{X}$, is the canonical projection of $\prod_{\mathtt{x}\in\mathbf{X}} D^{\star}_{\mathtt{x}}$ onto $\prod_{\mathtt{x}\in\mathbf{X}'} D^{\star}_{\mathtt{x}}$. This mechanism forces communication through the identity of flows of those signals that are shared by $\mathbf{X_1}$ and $\mathbf{X_2}$ (this provides the adequate formalisation of the notion of a "system of equations" we have informally used in the preceding sections). Obviously, " | " is commutative and associative.

Difficulties

Clearly, this basic framework can be extended so as to provide modular constructs etc., and does not raise particular problems. Problems begin to appear when it is wanted to distinguish between input and output flows, and to have practical algorithms for answering some questions of practical importance, for instance :

- *Functionality :* Given a partition of $\mathbf{X}$ into $\mathbf{X} = \mathbf{I} + \mathbf{O}$ (i.e., inputs plus local or outputs), is $\mathcal{S}(\mathbf{X})$ a function from $\prod_{\mathtt{i}\in\mathbf{I}} D^{\star}_{\mathtt{i}}$ (inputs) to $\prod_{\mathtt{o}\in\mathbf{O}} D^{\star}_{\mathtt{o}}$ (locals or outputs), i.e., does the following property hold ?

$$\forall \sigma, \sigma' \in \mathcal{S}(\mathbf{X}) \;:\; \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma) = \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma') \text{ implies } \sigma = \sigma' \qquad (6)$$

- *Sequentiality :* Can behaviours be constructed stepwisely, i.e., does the following property hold ?

$$\exists k \,,\, \forall \sigma, \sigma' \in \mathcal{S}(\mathbf{X}) \;:\; \sigma < \sigma' \text{ implies } \exists \sigma'' \in \mathcal{S}(\mathbf{X}) \,,\, \sigma \leq \sigma'' < \sigma' \text{ and } |\sigma''^{-1}\sigma'| \leq k$$

where

  - $\leq$ is the prefix order of the product monoid $\prod_{\mathtt{x}\in\mathbf{X}} D^{\star}_{\mathtt{x}}$, i.e., $\sigma_1 \leq \sigma_2$ if and only if $\exists \sigma_3 \,,\, \sigma_2 = \sigma_1.\sigma_3$. Then we define $\sigma_1^{-1}\sigma_2$ as being σ_3 ;

[7] The symbol "$\mathcal{S}$" should be reminiscent of "specification", this is because sets of behaviours are described, but not how to actually generate them.

– $|\ldots|$ is the standard length function over $D^\star$, and

$$\forall \sigma \in \prod_{\mathtt{x}\in\mathbf{X}} D^\star_{\mathtt{x}}\ ,\ |\sigma| = \sum_{\mathtt{x}\in\mathbf{X}} |\Phi_{\mathbf{X}\mapsto\mathtt{x}}(\sigma)|$$

This property is essential for applications to reactive or real-time systems.

- *Causality:* Does the future of inputs only affects the future of outputs, i.e., does the following property hold ?

$$\forall \sigma, \sigma' \in \mathcal{S}(\mathbf{X})\ :\quad \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma) \leq \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma')\ \text{ implies }\ \sigma \leq \sigma'$$

Again is causality an essential property for deterministic reactive or real-time systems, as it is easy to see that causality implies functionality.

Discussing possible solutions

One idea for getting those answers would consist in restricting ourselves to finite types, (maybe by defining finite abstractions of infinite types), and then in restricting ourselves to *regular* relations. According to [12] a regular relation $\mathcal{S}(\mathbf{X})$ can be represented by a finite generalised automaton — i.e., a finite automaton whose transitions are labelled with elements of $\prod_{\mathtt{x}\in\mathbf{X}} D^\star_{\mathtt{x}}$ — and the above properties can be finitely checked on this automaton. This certainly requires choosing *regular primitive relations,* but this is not sufficient, since product monoids of the form $\prod_{\mathtt{x}\in\mathbf{X}} D^\star_{\mathtt{x}}$ are *not* free monoids, and intersection does not preserve regularity, see [12]. Thus, the same holds for our parallel composition, as illustrated by the following example.

Example. Consider

- a "downsampler" denoted by the equation " `y = x when c` ", and defined by the regular expression:

$$\left[\sum_{u\in D_{\mathtt{x}}} (u^{\mathtt{x}}, \mathrm{true}^{\mathtt{c}}, u^{\mathtt{y}}) + (u^{\mathtt{x}}, \mathrm{false}^{\mathtt{c}}, \varepsilon^{\mathtt{y}})\right]^\star$$

where ε is the empty string of any type,

- a logical inverter denoted by " `y = not x` ", and defined by the regular expression:

$$[(\mathrm{false}^{\mathtt{x}}, \mathrm{true}^{\mathtt{y}}) + (\mathrm{true}^{\mathtt{x}}, \mathrm{false}^{\mathtt{y}})]^\star$$

- a logical "and" gate denoted by " `z = x and y` ", and defined by the regular expression:

$$\left[\sum_{u\in B_{\mathtt{x}}, v\in B_{\mathtt{y}}} (u^{\mathtt{x}}, v^{\mathtt{y}}, [u \text{ and } v]^{\mathtt{z}})\right]^\star$$

where $B_{\mathtt{x}} = B_{\mathtt{y}} = \{\text{true, false}\}$.

and the program:

```
( t = y and z
| y = x when c
| z = x when d
| d = not c )
```

Since the composition `|` forces both `y` and `z` strings to have equal lengths, any admissible behaviour of the program should have a `c` string with an equal number of "true" and "false" values, which is clearly not regular.

Another even more radical possibility would consist in restricting to recognisable relations, which are known to be preserved by intersection. But the resulting expressive power would be very poor : for instance, the relation $(a,b)^\star$ which emits a "b" each time an "a" is received, is not recognisable over $\{a\}^\star \times \{b\}^\star$. Let us show now how a synchronous solution solves the problem.

4.2 The synchronous solution

The problem solves trivially if we restrict ourselves to length preserving relations, i.e., we move from $\prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^\star$ to free monoids like $[\prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}]^\star$. Then regular relations are identical to recognisable ones. Furthermore, inverse morphism and intersection preserve recognisability. Thus, our parallel construct preserves it. However, length preserving relations have a restricted modeling power [8]. The idea is then to add to each type an "absent" element $\perp$, i.e., $D_{\mathtt{x}}^\perp = D_{\mathtt{x}} \cup \{\perp\}$ and work over monoids of the form $[\prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^\perp]^\star$. This is exactly what we did when introducing informally our notion of a clock in Section 2.2. In doing so, we have in mind checking most properties on this structure, and then eventually desynchronising behaviours by deleting those $\perp$ values, in the very same way our informal discussion of example (1) proceeded in the preceding section.

Example. Returning to our previous example, and considering the following modified primitive relations :

$$\texttt{y = x when c} \quad : \quad [\textstyle\sum_{u\in D_{\mathtt{x}}} (u^{\mathtt{x}}, \mathrm{true}^{\mathtt{c}}, u^{\mathtt{y}}) + (u^{\mathtt{x}}, \mathrm{false}^{\mathtt{c}}, \perp^{\mathtt{y}}) + (\perp^{\mathtt{x}}, \perp^{\mathtt{c}}, \perp^{\mathtt{y}})]^\star$$

$$\texttt{y = not x} \quad : \quad [(\mathrm{false}^{\mathtt{x}}, \mathrm{true}^{\mathtt{y}}) + (\mathrm{true}^{\mathtt{x}}, \mathrm{false}^{\mathtt{y}}) + (\perp^{\mathtt{x}}, \perp^{\mathtt{y}})]^\star$$

$$\texttt{z = x and y} \quad : \quad \left[\textstyle\sum_{u\in B_{\mathtt{x}}, v\in B_{\mathtt{y}}} (u^{\mathtt{x}}, v^{\mathtt{y}}, [u \text{ and } v]^{\mathtt{z}}) + (\perp^{\mathtt{x}}, \perp^{\mathtt{y}}, \perp^{\mathtt{z}})\right]^\star$$

the program :

```
( t = y and z
| y = x when c
| z = x when d
| d = not c )
```

now yields the only "silent" behaviour $[\perp^{\mathtt{c}}, \perp^{\mathtt{d}}, \perp^{\mathtt{x}}, \perp^{\mathtt{y}}, \perp^{\mathtt{z}}, \perp^{\mathtt{t}}]^\star$, which is certainly regular.

[8] they correspond to the "single clocked systems" of our informal discussion in the introduction.

Checking properties

More formally, we choose some appropriate recognisable prefix closed primitive relations:

$$\mathcal{S}_{\text{primitive}}(\mathbf{X}) \subseteq \left[\prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^{\perp}\right]^{\star} .$$

Then every relation $\mathcal{S}(\mathbf{X})$ constructible by our parallel composition, is recognisable and prefix closed on $[\prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^{\perp}]^{\star}$. Then there exists a deterministic finite automaton $\mathrm{Autom}\mathcal{S}(\mathbf{X})$ whose transitions are labelled over the alphabet

$$\mathrm{Event}(\mathbf{X}) =_{\mathrm{def}} \prod_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^{\perp},$$

i.e., t-uples of values of each type, and such that every state is terminal (to guarantee prefix closedness). Let

$$\mathrm{Autom}\mathcal{S}(\mathbf{X}) = \{Q\ ,\ q_0 \in Q\ ,\ T \in Q \times \mathrm{Event}(\mathbf{X}) \mapsto Q\}$$

be such an automaton, with as usual Q a finite non empty set of states, q_0 an initial state, and T a transition function. Checking our properties on this automaton is then obvious:

- *Functionality:* In any state, there cannot be two distinct transitions whose labels yield the same projection on inputs, i.e., using the notations of (6):

$$\begin{array}{c} \forall q \in Q\ ,\ a \in \mathrm{Event}(\mathbf{X})\ : \\ \exists T(q,a) \ \text{ and } \ \exists T(q,a') \ \text{ and } \ \Phi_{\mathbf{X}\mapsto\mathbf{I}}(a) = \Phi_{\mathbf{X}\mapsto\mathbf{I}}(a') \ \text{ implies } \ a = a' \end{array}$$

- *Sequentiality:* There is nothing to check since prefix closedness yields the desired property with $k = |\mathbf{X}|$ where $|\mathbf{X}|$ is the cardinal of the set $\mathbf{X}$.

- *Causality:* In general, causality implies functionality, and, in our approach, it is easy to show that functionality and prefix clodsedness implies causality. Let us verify this. Prefix closedness can be expressed as:

$$\forall \sigma \in \mathcal{S}(\mathbf{X}), \sigma' \in [\mathrm{Event}(\mathbf{X})]^{\star}\ ,\ \sigma' \leq \sigma \ \text{ implies } \sigma' \in \mathcal{S}(\mathbf{X})$$

  Now assume $\sigma, \sigma' \in \mathcal{S}(\mathbf{X})$, with $\Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma) \leq \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma')$. Then, there exists $\sigma'' \in [\mathrm{Event}(\mathbf{X})]^{\star}$ such that $\Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma) = \Phi_{\mathbf{X}\mapsto\mathbf{I}}(\sigma'')$ and $\sigma'' \leq \sigma'$. But then prefix closedness yields $\sigma'' \in \mathcal{S}(\mathbf{X})$ and, by functionality, $\sigma = \sigma''$. Thus $\sigma = \sigma'' \leq \sigma'$.

- *Model checking:* More generally, so called "Model checking" techniques can be applied on this automaton, so as to verify any desirable property.

Issues of operational semantics, and synchronous partial orders

Referring to the classical dataflow framework, let us investigate what would result from relying on Kahnian operational semantics [20] for our synchronous systems. Consider again our simple example.

EXAMPLE. Let us return to our "and" gate `z = x and y` . Recall its definition :

$$\left[\sum_{u \in B_{\mathtt{x}}, v \in B_{\mathtt{y}}} (u^{\mathtt{x}}, v^{\mathtt{y}}, [u \text{ and } v]^{\mathtt{z}}) + (\perp^{\mathtt{x}}, \perp^{\mathtt{y}}, \perp^{\mathtt{z}})\right]^{\star}$$

Assume now that we want to connect the output `z` to the input `y` . A possible way of achieving this connection is to define the relation `y = z` :

$$\left[\sum_{u \in D_{\mathtt{z}}^{\perp}} (u^{\mathtt{z}}, u^{\mathtt{y}})\right]^{\star}$$

and consider the composition:

```
( z = x and y
| y = z )
```

This system of equations has many solutions, namely

$$\left[\sum_{u,v \;:\; v=u \text{ and } v} (u^{\mathtt{x}}, v^{\mathtt{y}}, v^{\mathtt{z}}) + (\perp^{\mathtt{x}}, \perp^{\mathtt{y}}, \perp^{\mathtt{z}})\right]^{\star} \tag{7}$$

On the other hand, operationally in the Kahnian sense, the connection from `z` to `y` yields a short circuit which we expect to yield the unique solution $(\perp^{\mathtt{x}}, \perp^{\mathtt{y}}, \perp^{\mathtt{z}})$, see [20]. Clearly, this is an anomaly of the same nature as that discussed by Brock and Ackerman in [10], namely a discrepancy between behavioural and operational semantics. Yet, several solutions have been brought to the problem [10, 9, 28], for instance. Among these, the one which seems to best fit our approach is the original one of Brock and Ackerman, better formalised by several authors, [14, 30], which consists of moving from strings to partial orders. This is what we do next.

Synchronous partial orders. Moving our synchronous solution from strings to partial orders is a rather technical but easy exercise. Let $\mathrm{Pomset}(\mathbf{X})$ the set of finite pomsets labelled over

$$\mathrm{Domain}(\mathbf{X}) =_{\mathrm{def}} \mathbf{X} + \sum_{\mathtt{x} \in \mathbf{X}} D_{\mathtt{x}}^{\perp}$$

where $+$ denotes disjoint union, and labels in $\mathbf{X}$, i.e., signals (or flow names) will serve to define concatenation. For $\mathbf{X}' \subseteq \mathbf{X}$, the canonical projection $\Phi_{\mathbf{X} \mapsto \mathbf{X}'}(o)$, $o \in \mathrm{Pomset}(\mathbf{X})$, is the order obtained by deleting in o every element not labelled over $\mathrm{Domain}(\mathbf{X}')$. Then our alphabet will now be the set $\mathrm{PomEvent}(\mathbf{X}) \subseteq \mathrm{Pomset}(\mathbf{X})$ such that $o \in \mathrm{PomEvent}(\mathbf{X})$ if and only if the following condition holds :

Condition 1 (definition of $\mathrm{PomEvent}(\mathbf{X})$**)** *For any* $\mathtt{x} \in \mathbf{X}$*, there exists a unique element* $u \in D_{\mathtt{x}}^{\perp}$ *such that the projection* $\Phi_{\mathbf{X} \mapsto \mathtt{x}}(o)$ *has the form* $\mathtt{x} \to u^{\mathtt{x}} \to \mathtt{x}$*, where* $\to$ *is the order relation; furthermore, the two extremal elements of* $\Phi_{\mathbf{X} \mapsto \mathtt{x}}(o)$ *are also extremal in* o*.*

The concatenation $o.o'$ is is the order obtained by merging together maximal elements of o and minimal elements of o' sharing the same labels, and then by erasing them. For instance:

$$\begin{pmatrix} a & \rightarrow & \mathtt{x} \\ & \nearrow & \\ b & \rightarrow & \mathtt{y} \end{pmatrix} \cdot \begin{pmatrix} \mathtt{x} & \rightarrow & c \\ & \nearrow & \\ \mathtt{y} & \rightarrow & d \end{pmatrix} = \begin{pmatrix} a & \rightarrow & c \\ & \nearrow & \\ b & \rightarrow & d \end{pmatrix}$$

This operation is clearly associative, with the empty order $\emptyset$ as a neutral element. Concatenation extends to sets of orders, and together with set theoretic union, allows us to define a $[\ldots]^\star$ operation. Clearly also, $[\mathrm{PomEvent}(\mathbf{X})]^\star$ is a monoid, its elements will be generically denoted by the symbol π. Also, for $\pi = o_1.o_2.\ldots \in [\mathrm{PomEvent}(\mathbf{X})]^\star$, and for $\mathbf{X}' \subseteq \mathbf{X}$, the canonicalprojection $\Phi_{\mathbf{X}\mapsto\mathbf{X}'}(\pi)$ is equal to $\Phi_{\mathbf{X}\mapsto\mathbf{X}'}(o_1).\Phi_{\mathbf{X}\mapsto\mathbf{X}'}(o_2).\ldots$, i.e., the projection acts componentwise with respect to concatenation.

Accordingly we select our primitive relations, $\Pi_{\mathrm{primitive}}(\mathbf{X})$, as being some appropriate recognisable prefix closed subsets of $[\mathrm{PomEvent}(\mathbf{X})]^\star$ [9].

Finally, the parallel composition remains the same as before, namely

$$\Pi_1(\mathbf{X_1}) \mid \Pi_2(\mathbf{X_2}) \quad = \quad \Phi^{-1}_{\mathbf{X_1}\cup\mathbf{X_2}\mapsto\mathbf{X_1}}(\Pi_1(\mathbf{X_1})) \bigcap \Phi^{-1}_{\mathbf{X_1}\cup\mathbf{X_2}\mapsto\mathbf{X_2}}(\Pi_2(\mathbf{X_2})) \tag{8}$$

where

$$\Pi_i(\mathbf{X_i}) \subseteq [\mathrm{PomEvent}(\mathbf{X_i})]^\star,$$

and the $\Phi_{\mathbf{X}\mapsto\mathbf{X}'}$ are restricted to operate on $[\mathrm{PomEvent}(\mathbf{X})]^\star$ instead of $[\mathrm{Pomset}(\mathbf{X})]^\star$. Then recognisability is preserved by the inverse morphisms Φ^{-1} and intersection, and this is also trivially true for $\mathrm{PomEvent}(\mathbf{X})$-prefix closedness.

EXAMPLE. We now redefine `z = x and y` as follows:

$$\left[\sum_{u\in B_{\mathtt{x}}, v\in B_{\mathtt{y}}} \begin{pmatrix} \mathtt{x} & \rightarrow & u^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{z} & \rightarrow & [u \text{ and } v]^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \uparrow & & \\ \mathtt{y} & \rightarrow & v^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{pmatrix} + \begin{pmatrix} \mathtt{x} & \rightarrow & \perp^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{z} & \rightarrow & \perp^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \uparrow & & \\ \mathtt{y} & \rightarrow & \perp^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{pmatrix}\right]^\star \tag{9}$$

and `y = x` as

$$\left[\sum_{u\in D^{\perp}_{\mathtt{x}}} \begin{pmatrix} \mathtt{x} & \rightarrow & u^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{y} & \rightarrow & u^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{pmatrix}\right]^\star \tag{10}$$

Clearly, the composition

[9] IMPORTANT REMARK: *let us emphazise that both notions are intimately bound to the monoid they appear in. In fact we better write here* $[\mathrm{PomEvent}(\mathbf{X})]^\star$*-recognisable and* $[\mathrm{PomEvent}(\mathbf{X})]^\star$*-prefix closed. Also, the symbol "*Π*" should be reminiscent of "program", since operational semantics is considered here.*

```
( z = x and y
| y = z )
```

yields the unique solution $\emptyset$! On the other hand,

```
( z = x and y
| z = y )
```

yields the many solutions corresponding to (7). If it is wished to "close the circuit" as in the first case, but in a correct way, a delay has to be introduced. For this purpose, we define the new primitive " `y = pre x init` v_o " as follows:

$$S^\star + S^\star.Y(v_o).\left[\sum_{u\in D_{\mathtt{x}}} X(u).S^\star.Y(u)\right]^\star . \sum_{u\in D_{\mathtt{x}}} X(u).S^\star \tag{11}$$

where

$$X(u) = (\mathtt{x} \to u^{\mathtt{x}} \to \mathtt{x})\ ,\ Y(u) = \left(\begin{array}{ccccc} \mathtt{x} & \to & \to & \to & \mathtt{x} \\ & \searrow & & & \\ \mathtt{y} & \to & u^{\mathtt{y}} & \to & \mathtt{y} \end{array}\right)\ ,\ S = \left(\begin{array}{ccccc} \mathtt{x} & \to & \perp^{\mathtt{x}} & \to & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{y} & \to & \perp^{\mathtt{y}} & \to & \mathtt{y} \end{array}\right)$$

(S is the silent event). This allows us to construct a correct feedback from outputs to inputs by writing:

```
( z = x and y
| y = pre z init true)
```

Note here that we have defined the `pre` relation by a regular expression whose atoms do not belong to PomEvent($\{\mathtt{x},\mathtt{y}\}$). We are allowed to do this provided we check that it truly defines a $[\text{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$-recognisable, $[\text{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$-prefix closed set. We briefly verify this for the sake of completeness.

1. " `y = pre x init` v_0" $\subseteq [\text{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$. This results from the fact that, although the atoms $Y(u)$ and $X(v)$ do not belong to PomEvent($\{\mathtt{x},\mathtt{y}\}$), they only appear jointly within expressions of the following form

$$Y(u).X(v) = \left(\begin{array}{ccccc} \mathtt{x} & \to & v^{\mathtt{x}} & \to & \mathtt{x} \\ & \searrow & & & \\ \mathtt{y} & \to & u^{\mathtt{y}} & \to & \mathtt{y} \end{array}\right)$$

   which truly belong to PomEvent($\{\mathtt{x},\mathtt{y}\}$).

2. $[\text{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$–prefix closedness is obvious since any $[\text{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$–prefix of

$$S^\star.Y(v).\prod_{i=1,n}\left[X(u_i).S^\star.Y(u_i)\right].X(u_{n+1}).S^\star$$

   has the form

$$S^\star.Y(v).\prod_{i=1,m}\left[X(u_i).S^\star.Y(u_i)\right].X(u_{m+1}).S^\star\ ,$$

   where $m \leq n$.

3. finally, $[\mathrm{PomEvent}(\{\mathtt{x},\mathtt{y}\})]^\star$–recognisability comes from the fact that the set of suffixes of

$$S^\star.Y(v).\prod_{i=1,n}[X(u_i).S^\star.Y(u_i)].X(u_{n+1}).S^\star$$

in `y = pre x init v` is

$$S^\star + S^\star.Y(u_{n+1}).\left[\sum_{u\in D_{\mathtt{x}}} X(u).S^\star.Y(u)\right]^\star.\sum_{u\in D_{\mathtt{x}}} X(u).S^\star$$

which equals " `y = pre x init` u_{n+1}". Since we assume all types to be finite, there is only a finite number of such sets and this proves recognisability.

4.3 Towards asynchronous implementation

Finally, assume now that we have obtained a satisfactory program, i.e. a $[\mathrm{PomEvent}(\mathbf{X})]^\star$-recognisable, $[\mathrm{PomEvent}(\mathbf{X})]^\star$-prefix closed set, over which we have checked for functionality, for instance. Then we can implement it *synchronously,* by changing (reinforcing) each label $o \in \mathrm{PomEvent}(\mathbf{X})$ of each transition, into a total order that extends it. We thus obtain a $[\sum_{\mathtt{x}\in\mathbf{X}} D^\perp_{\mathtt{x}}]^\star$–automaton whose transitions involve finite memory, finite time reactions.

Erasing $\perp$'s can also be performed on this automaton, since it can be finitely checked whether this preserves functionality, or not. Since erasing $\perp$'s clearly preserves sequentiality (the k bound can only decrease!), the total order extension can be performed once some $\perp$'s have been erased, thus yielding optimised *desynchronised* programs.

Yet, truly asynchronous implementations arise when taking fully into account the partial order nature of our behaviours. However, this yields now the problem that our solution is $[\mathrm{PomEvent}(\mathbf{X})]^\star$-recognisable, but this doesn't imply $\mathrm{Pomset}(\mathbf{X})$–recognisability, which is the required property for having bounded memory, bounded reaction time, asynchronous implementations. Let us check this on an example.

Example. Consider again the program `z = x and y` which was defined in (9). Erasing all $\perp$'s preserves functionality, and yields

$$\left[\sum_{u\in B_{\mathtt{x}},v\in B_{\mathtt{y}}}\left(\begin{array}{ccccc} \mathtt{x} & \rightarrow & u^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{z} & \rightarrow & [u \text{ and } v]^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \uparrow & & \\ \mathtt{y} & \rightarrow & v^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{array}\right)\right]^\star$$

The asynchronous behaviours allowed by this partial order specification are all total order extensions of its behaviours. These do not build a $[\sum_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}]^\star$–regular set, where $\mathbf{X} = \{\mathtt{x},\mathtt{y},\mathtt{z}\}$. To check this, denote by $\Pi($`z = x and y`$)$ this set, and consider its projection on $[D_{\mathtt{x}} + D_{\mathtt{y}}]^\star$. It contains all strings having an equal number of symbols in $D_{\mathtt{x}}$ and in $D_{\mathtt{y}}$, and thus is not $[D_{\mathtt{x}} + D_{\mathtt{y}}]^\star$-regular. Clearly, the reason is that this specification is "too asynchronous", so it has to be constrained in some way. For instance,

$$\left[\sum_{u\in B_{\mathtt{x}},v\in B_{\mathtt{y}}}\left(\begin{array}{ccccc} \mathtt{x} & \rightarrow & u^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & \nearrow & \\ \mathtt{z} & \rightarrow & [u \text{ and } v]^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \uparrow & \searrow & \\ \mathtt{y} & \rightarrow & v^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{array}\right)\right]^{\star}$$

is Pomset($\mathbf{X}$)–recognisable, and corresponds to a weakly synchronous solution: by itself, it yields the synchronous solution we already described, but composed with other relations, it provides concurrency and pipelining. Also, bounded overlapping of successive "instants" would preserve recognisability.

4.4 Summary and discussion

What we have built at this stage of our theoretical investigation is the following:

1. Given that our informal requirement was to model "systems of equations", relations over flows emerged as a natural framework, with (5) as only appropriate composition construct.

2. We have shown that such an approach was not practical, however, since regularity was not nicely preserved by our composition; but regularity is the proper mathematical formalisation of bounded memory – bounded reaction time, so it is really an important issue. Our *synchronous* approach was proposed as an alternative to overcome this difficulty. And we have shown that the use of our "$\perp$" additional symbol nicely fits our informal requirements for multiple clocked systems we expressed in the former section. This yields an appropriate technique for behavioural semantics, and it fits our informal requirements.

3. Then we have shown how relying on partial orders allows us to extend naturally our composition construct (5) to the form (8) in order to handle operational semantics. In doing so, desirable properties related to regularity and recognisability are guaranteed. Again, this composition rule just formalises the informal notions of signal flow graphs, block-diagrams, and their interconnections, as we have discussed in the previous section.

4. Finally we have shown how to properly "desynchronise" operational semantics, essentially by carefully erasing $\perp$'s when they are not required to ensure functionality; to keep regularity, reinforcing the partial orders is sometimes necessary. A key point was that desynchronised programs *should not be composed* since regularity would be immediately lost, exactly as in item 1 above.

To conclude, this analysis strongly calls for our synchronous approach to data flow. Should we be happy now? Not yet, since some unpleasant features still remain, namely:

1. Composing together regular expressions of the forms (10) and (11) via our " $\mid$ " composition construct cannot be easily written in some simple closed form formula, i.e., our

syntactic form for the regular expressions is not preserved via the " | " composition. We shall propose a practical solution to overcome this.

2. Return to example discussed at equations (9,10), this program is certainly not alive, and the corresponding PomEvent($\mathbf{X}$)-automaton has a sink state, as mentioned in Subsection 4.3. However, combining this program with another alive one with no shared signal would leave this program deadlocked and the other one alive. And more complicated situations can occur. So it is desirable to *isolate* in some way deadlocked components of a given program. But, what our partial order technique basically does is the following: when pomsets are composed which do not match together (i.e., intuitively, short circuits are created), then our composition construct, based on intersection, just removes them. Hence, roughly speaking, short circuits that would result from composition are not created, but associated behaviours are erased instead. So incorrect parts of a program are erased, we would prefer them to be enlighten for reporting to the user. To overcome this, we shall move from *partial orders* to *preorders,* since preorders will allow us to keep short circuits visible in our semantics.

3. Each of our two semantics (behavioural and operational) are "monolithic", i.e., they build different worlds. In particular, we have no composition construct combining a set of legal behaviours with a set of legal partial orders. But we have expressed in the former section the informal requirement that behavioural and operational semantics can be handled separately or combined as our convenience. Thus we should provide an adequate glue allowing us to mix the two styles of semantics in a flexible way. This is what we shall do in the sequel.

With these three improvements, we shall be very close to a satisfactory proposal for a synchronous dataflow programming language, both from theoretical and practical point of view.

5 Modelling: our final proposal

In this section, in each subsection, we stepwisely answer points 1, 2, and 3 of our previous discussion.

5.1 Introducing states

The reason for expressions (10) and (11) not being easily combined is the tricky form of expression $\sum_{u \in D_{\mathbf{X}}} X(u).S^{\star}.Y(u)$ inside brackets in (11). In this expression, the value u is shared by input X and output Y, but at *different* instants (note that $Y(v)$ alone lies before this bracket, which causes different instants to be involved): this is the reason why expression (11) is not easily combined with (10), where the star operation involves a single instant. What we shall do in essence is to modify our expression for the delay so that only a single instant be involved in the star operation. This theoretical reason for introducing states compares with the informal one we have provided in Section 2.3.

Partial orders with states

Thus we enlarge our domain of labels by setting

$$\mathrm{Domain}(\mathbf{X},\Xi) =_{\mathrm{def}} \ \mathbf{X} + \sum_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^{\perp} + \sum_{\xi\in\Xi} D_{\xi} \tag{12}$$

where we now introduce an additional set Ξ of *states* and associated domains. Note that D_ξ is *not* enlarged with the symbol $\perp$, i.e., we do not need to refer explicitly about the absence of a state, the reason for doing so will be given soon. Then we denote by $\mathrm{Pomset}(\mathbf{X},\Xi)$ the set of finite pomsets labelled over $\mathrm{Domain}(\mathbf{X},\Xi)$. Canonical projections are defined as before. Our new alphabet $\mathrm{PomEvent}(\mathbf{X},\Xi)$ is the subset composed of elements $o \in \mathrm{Pomset}(\mathbf{X},\Xi)$ satisfying the following two conditions (compare with Condition 1):

Condition 2 (definition of $\mathrm{PomEvent}(\mathbf{X},\Xi)$**)**

1. *For any signal* $\mathtt{x} \in \mathbf{X}$*, there exists a unique element* $u \in D_{\mathtt{x}}^{\perp}$ *such that the projection* $\Phi_{(\mathbf{X},\Xi)\mapsto\mathtt{x}}(o)$ *has the form* $\mathtt{x} \to u^{\mathtt{x}} \to \mathtt{x}$*, where* $\to$ *is the order relation; furthermore, the two extremal elements of* $\Phi_{(\mathbf{X},\Xi)\mapsto\mathtt{x}}(o)$ *are also extremal in* o*.*
2. *For any state* $\xi \in \Xi$*, there exists a finite integer* $k \geq 0$ *and there exist* $u_0,\ldots,u_{k+1} \in D_\xi$ *such that the projection* $\Phi_{(\mathbf{X},\Xi)\mapsto\xi}(o)$ *has the form* $u_0^{\xi} \to \ldots \to u_{k+1}^{\xi}$*, where* $\to$ *is the order relation; furthermore, the two extremal elements of* $\Phi_{(\mathbf{X},\Xi)\mapsto\xi}(o)$ *are also extremal in* o*.*

The concatenation $o.o'$ is now slightly modified to account for the fact that labels of extremal states are not names any more, but are rather values, hence they may not match in general. Thus $\mathrm{Pomset}(\mathbf{X},\Xi)$ is enlarged with a zero element we call NIL, which satisfies the following properties:

$$\begin{aligned}
\forall o \in \mathrm{Pomset}(\mathbf{X},\Xi) &\ : \ \mathrm{NIL}.o = o.\mathrm{NIL} = \mathrm{NIL} \\
\forall o,o' \in \mathrm{Pomset}(\mathbf{X},\Xi) &\ : \ \max\{o\} \neq \min\{o'\} \ \Rightarrow \ o.o' = \mathrm{NIL} \\
\forall o \in \mathrm{Pomset}(\mathbf{X},\Xi) &\ : \ \mathrm{NIL} + o = o + \mathrm{NIL} = o
\end{aligned}$$

where the expression "$\max\{o\} \neq \min\{o'\}$" (resp. "$\max\{o\} = \min\{o'\}$") means that the set of labels of maximal elements of o and that of minimal elements of o' are different (resp. identical)[10]. Thus, NIL is absorbing for the concatenation, is the neutral element for the union $+$, and it has to be interpreted as the "absence of behaviour". Now if $\max\{o\} = \min\{o'\}$ holds, then $o.o'$ is the order obtained by merging together maximal elements of o and minimal elements of o' sharing the same labels, and then by erasing them. Hence a mismatch between maximal and minimal elements of adjacent items in a chain of the form $o_1, o_2, o_3, \ldots$ causes this chain to collapse into the absorbing element NIL. This particular definition of concatenation plays a central role in "propagating" values through instants, as explained in example (15) to follow. Again, concatenation extends to sets of orders, and together with set theoretic

[10]Note that the third property involves the "+" operation, which is not related to concatenation ; this property is essential for the desired properties of our regular expressions, however.

union, allows us to define a $[\ldots]^\star$, and $[\text{PomEvent}(\mathbf{X},\Xi)]^\star$ is a monoid, whose elements are still generically denoted by the symbol π. Finally, the parallel construct remains the same :

$$\Pi_1(\mathbf{Z_1}) \mid \Pi_2(\mathbf{Z_2}) \quad = \quad \Phi^{-1}_{\mathbf{Z_1}\cup\mathbf{Z_2}\mapsto\mathbf{Z_1}}(\Pi_1(\mathbf{Z_1})) \bigcap \Phi^{-1}_{\mathbf{Z_1}\cup\mathbf{Z_2}\mapsto\mathbf{Z_2}}(\Pi_2(\mathbf{Z_2})) \tag{13}$$

where $\mathbf{Z}$ stands for $(\mathbf{X},\Xi)$,

$$\Pi_i(\mathbf{Z_i}) \subseteq [\text{PomEvent}(\mathbf{Z_i})]^\star,$$

and the $\Phi_{\mathbf{Z}\mapsto\mathbf{Z'}}$ are restricted to operate on $[\text{PomEvent}(\mathbf{Z})]^\star$ instead of $[\text{Pomset}(\mathbf{Z})]^\star$. Accordingly, recognisability is still preserved by this construct. An important property of states in our Π programs is that *states are local to each program, so that they play no role in our communication construct,* and this is the reason for not using $\perp$ symbols for states (explicit synchronisation via a rigid notion of "instant" is only needed for signals, since only signals play a role in communication).

Example. We reconsider the primitive " `y = pre x init` v_0" defined in (11). Using our new notion of state, we can redefine it as follows :

$$S^\star(v_o) + S^\star(v_o).\left[\sum_{u\in D_\mathtt{x}} pre(v_o,u)\right].\left[\sum_{v,u\in D_\mathtt{x}} pre(v,u) + \sum_{v\in D_\mathtt{x}} S(v)\right]^\star \tag{14}$$

where the new silent event $S(v)$ and $pre(v,u)$ are given by

$$S(v) = \left(\begin{array}{ccccc} v^\xi & \to & \to & \to & v^\xi \\ & & & & \\ \mathtt{x} & \to & \perp^\mathtt{x} & \to & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{y} & \to & \perp^\mathtt{y} & \to & \mathtt{y} \end{array}\right) \ , \ pre(v,u) = \left(\begin{array}{ccccccc} \mathtt{x} & \to & \to & \to & u^\mathtt{x} & \to & \mathtt{x} \\ & & & & \downarrow & & \\ v^\xi & \to & v^\xi & \to & u^\xi & \to & u^\xi \\ & & \downarrow & & & & \\ \mathtt{y} & \to & v^\mathtt{y} & \to & \to & \to & \mathtt{y} \end{array}\right)$$

where ξ is the (local) state. Note that, due to our rule for concatenation, $pre(v_1,u_1).pre(v_2,u_2)$ is defined if and only if $v_2 = u_1$, and equals in this case

$$\left(\begin{array}{ccccccccccc} \mathtt{x} & \to & \to & \to & u_1^\mathtt{x} & \to & \to & \to & u_2^\mathtt{x} & \to & \mathtt{x} \\ & & & & \downarrow & & & & \downarrow & & \\ v_1^\xi & \to & v_1^\xi & \to & u_1^\xi & \to & u_1^\xi & \to & u_2^\xi & \to & u_2^\xi \\ & & \downarrow & & & & \downarrow & & & & \\ \mathtt{y} & \to & v_1^\mathtt{y} & \to & \to & \to & u_1^\mathtt{y} & \to & \to & \to & \mathtt{y} \end{array}\right)$$

Consequently,

$$\begin{aligned} pre(v_1,u_1).\sum_{v_2\in D_\mathtt{x}} pre(v_2,u_2) \quad &= \quad \left(\begin{array}{ccccccccccc} \mathtt{x} & \to & \to & \to & u_1^\mathtt{x} & \to & \to & \to & u_2^\mathtt{x} & \to & \mathtt{x} \\ & & & & \downarrow & & & & \downarrow & & \\ v_1^\xi & \to & v_1^\xi & \to & u_1^\xi & \to & u_1^\xi & \to & u_2^\xi & \to & u_2^\xi \\ & & \downarrow & & & & \downarrow & & & & \\ \mathtt{y} & \to & v_1^\mathtt{y} & \to & \to & \to & u_1^\mathtt{y} & \to & \to & \to & \mathtt{y} \end{array}\right) \\ &= \quad pre(v_1,u_1).pre(u_1,u_2) \end{aligned} \tag{15}$$

hence it is easily checked that formulae (11) and (14) are equivalent when projection on the input/output pair `{x,y}` is considered.

A fundamental formula for parallel composition

The following formula is very important since it justifies the use of our new notion of domain, as enlarged with states:

$$\left[\sum_{P_1 \in \mathbf{P_1}} P_1\right]^\star \mid \left[\sum_{P_2 \in \mathbf{P_2}} P_2\right]^\star = \left[\sum_{(P_1,P_2)\ \in\ \mathbf{P_1} \times \mathbf{P_2}} (P_1 \mid P_2)\right]^\star \tag{16}$$

where, for $i = 1, 2$, $\mathbf{P_i}$ is some subset of Pomset$(\mathbf{X_i}, \Xi_i)$.

Example. Applying formula (16) to compute the composition of (14) and (9) yields the following explicit formula for the program

```
( x = z and y
| y =  pre x init true )
```

namely

$$[S(v_o) \mid S']^\star + [S(v_o) \mid S']^\star . \Pi(v_o) . \left[\sum_{v \in D_\mathbf{X}} \Pi(v)\right]^\star \tag{17}$$

where

$$\Pi(v) = \sum_{u,w',v' \in D_\mathbf{X}} [\ \mathit{pre}(v,u) \mid \mathit{and}(w',v')\ ]\ ,$$

$v_o =$ `true`, $S(v)$ and $\mathit{pre}(v,u)$ have been defined just after (14), and

$$S' = \left(\begin{array}{ccccc} \mathtt{z} & \rightarrow & \perp^\mathtt{z} & \rightarrow & \mathtt{z} \\ & & \downarrow & & \\ \mathtt{x} & \rightarrow & \perp^\mathtt{x} & \rightarrow & \mathtt{x} \\ & & \uparrow & & \\ \mathtt{y} & \rightarrow & \perp^\mathtt{y} & \rightarrow & \mathtt{y} \end{array}\right)\ ,\ \mathit{and}(w',v') = \left(\begin{array}{ccccc} \mathtt{z} & \rightarrow & w'^\mathtt{z} & \rightarrow & \mathtt{z} \\ & & \downarrow & & \\ \mathtt{x} & \rightarrow & [w' \text{ and } v']^\mathtt{x} & \rightarrow & \mathtt{x} \\ & & \uparrow & & \\ \mathtt{y} & \rightarrow & v'^\mathtt{y} & \rightarrow & \mathtt{y} \end{array}\right)$$

Note that

$$\begin{aligned} \sum_{v \in D_\mathbf{X}} \Pi(v) &= \sum_{v,u,w',v' \in D_\mathbf{X}} [\ \mathit{pre}(v,u)\ \mid \mathit{and}(w',v')\ ] \\ &= \sum_{v,w' \in D_\mathbf{X}} [\ \mathit{pre}(v, u = w' \text{ and } v) \mid \mathit{and}(w',v)\ ] \end{aligned}$$

so that formula (17) yields the expected operational semantics. Thus the introduction of states solves both the above raised theoretical issue and the practical one that was discussed in Section 2.3.

5.2 Using prossets

To keep "short circuits" in programs visible, we move from pomsets to prossets (i.e., "preorder specification sets"), as suggested in [14]. There is very little to change in our alphabet. Indeed, we replace Pomset$(\mathbf{X}, \Xi)$ by Prosset$(\mathbf{X}, \Xi)$, i.e., $\rightarrow$ is now a *preorder,* i.e., a transitive and reflexive relation. The associated equivalence relation is denoted by $\leftrightarrow$. Hence, $\mathtt{x} \rightarrow \mathtt{z}$ should read : `z` *cannot be computed before* `x`. Then the definition of $\mathrm{ProsEvent}(\mathbf{X}, \Xi)$ follows accordingly using again Condition 2. Also, the definition of the concatenation remains the same as in Subsection 5.1.

There is something to be changed, however, namely the parallel composition construct `|` for the reason we explain now. Let us first revise our requirements for this composition, namely :

♡ the parallel composition construct `|` should properly formalise both notions of systems of equations and block-diagrams, i.e.,

1. as far as behavioral semantics is concerned, it should correspond to the *intersection* of behaviours of shared signals,
2. as far as operational semantics is concerned, it should correspond to the *least common extension* of preorders of shared signals.

Consequently, defining the parallel composition as being the intersection of sets of preorders is not suited, since it would not match requirement 2 above. To satisfy this love "♡", it will be useful to introduce the following partial order $\leq_\pi$ on $[\mathrm{ProsEvent}(\mathbf{X}, \Xi)]^\star$. Consider two elements $\pi_1 = \{S_1, \rightarrow_1, v_1\}$ and $\pi_2 = \{S_2, \rightarrow_2, v_2\}$ of $[\mathrm{ProsEvent}(\mathbf{X}, \Xi)]^\star$, where, for $i = 1, 2$, S_i denotes the set of vertices of the preorder $\rightarrow_i$, and v_i is the labelling function. We say that

$$\pi_1 \ \leq_\pi \ \pi_2 \tag{18}$$

if $S_1 = S_2$, $v_1 = v_2$, and if $\rightarrow_2$ is an extension of $\rightarrow_1$. Hence the two prossets have the same labels, and the preorder of the greatest one is an extension of that of the least one.

Now we are ready to redefine our composition. If $(\mathbf{X}', \Xi') \subseteq (\mathbf{X}, \Xi)$, the canonical projection

$$\Phi_{(\mathbf{X},\Xi)\mapsto(\mathbf{X}',\Xi')}(\pi) \ , \ \pi \in [\mathrm{ProsEvent}(\mathbf{X}, \Xi)]^\star$$

is the prosset obtained by deleting in π every element not labelled over $\mathrm{ProsEvent}(\mathbf{X}', \Xi')$. For $i = 1, 2$, let $\Pi_i \subseteq [\mathrm{ProsEvent}(\mathbf{X_i}, \Xi_i)]^\star$ two programs, and define $\mathbf{X} = \mathbf{X_1} \cup \mathbf{X_2}$, $\Xi = \Xi_1 \cup \Xi_2$. Then [11]

$$\begin{array}{l} \Pi_1 \mid \Pi_2 \text{ is the set of } \textit{minimal } \pi \in [\mathrm{ProsEvent}(\mathbf{X}, \Xi)]^\star \text{ such that,} \\ \text{for } i = 1, 2 \ , \ \exists \pi_i \in \Pi_i(\mathbf{X_i}, \Xi_i) \ : \ \pi_i \ \leq_\pi \ \Phi_{(\mathbf{X},\Xi)\mapsto(\mathbf{X_i},\Xi_i)}(\pi) \end{array} \tag{19}$$

The composition " `|` " is associative, commutative, and idempotent (i.e., $\Pi | \Pi = \Pi$).

Example. It is interesting to revisit example

```
( z = x and y
| y = z )
```

[11] in the following formula, "minimal" refers to the order $\leq_\pi$ on prossets.

we discussed just after formulae (9,10). Its semantics now yields

$$\left[\sum_{u,v\ :\ v=u \text{ and } v} \begin{pmatrix} \mathtt{x} & \rightarrow & u^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{z} & \rightarrow & [u \text{ and } v]^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \updownarrow & & \\ \mathtt{y} & \rightarrow & v^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{pmatrix} + \begin{pmatrix} \mathtt{x} & \rightarrow & \perp^{\mathtt{x}} & \rightarrow & \mathtt{x} \\ & & \downarrow & & \\ \mathtt{z} & \rightarrow & \perp^{\mathtt{z}} & \rightarrow & \mathtt{z} \\ & & \updownarrow & & \\ \mathtt{y} & \rightarrow & \perp^{\mathtt{y}} & \rightarrow & \mathtt{y} \end{pmatrix} \right]^{\star} \tag{20}$$

where $\mathtt{z} \leftrightarrow \mathtt{y}$ stands for $\{\ \mathtt{z} \leftarrow \mathtt{y}$ and $\mathtt{z} \rightarrow \mathtt{y}\ \}$. Hence short circuits are exhibited, but removing all arrows exactly yields the behavioural semantics (7), which is what we wanted.

5.3 Some useful abstractions of programs, mixing behavioural and operational semantics

Given a program $\Pi(\mathbf{X}, \Xi)$, the following abstractions will be of interest.

Behavioural abstraction of $\Pi(\mathbf{X}, \Xi)$ is defined as follows. For each $\pi \in \Pi(\mathbf{X}, \Xi)$, we maximally weaken the preorder subject to Condition 2, i.e., we keep only those "straight" preorders of the form $\ldots \rightarrow u^{\mathtt{x}} \rightarrow \ldots$ or $\ldots \rightarrow u_i^{\xi} \rightarrow \ldots$ and remove the "oblique" ones, involving different signals or states. This yields a new preorder we write $\sigma(\pi)$ and we call the *behaviour* of π. Indeed, $\sigma(\pi)$ involves only the ordering of time, locally for each signal or state, in its preorder part, so that it truly reflects only behaviours. When π ranges over $\Pi(\mathbf{X}, \Xi)$, its behaviour $\sigma(\pi)$ ranges over a set we denote by $\mathcal{S}(\Pi(\mathbf{X}, \Xi))$ and we call the *behavioural abstraction* of program $\Pi(\mathbf{X}, \Xi)$. The map $\Pi(\mathbf{X}, \Xi) \mapsto \mathcal{S}(\Pi(\mathbf{X}, \Xi))$ is a morphism for the composition operator $\mid$. The behavioural abstraction provides the adequate answer to our requirement of flexible mixing of behavioural and operational semantics.

Synchronisation abstraction of $\Pi(\mathbf{X}, \Xi)$ is obtained by mapping, for each signal $\mathtt{x}$, the type $D_{\mathtt{x}}$ onto the single value $\top$, where $\top$ is another distinguished symbol referring to the status "present". This yields, for each signal $\mathtt{x}$, the (universal) *synchronisation type*

$$H_{\mathtt{x}}^{\perp} = \{\perp, \top\}$$

encoding the status {absent, present}. Taking the image of the behavioural abstraction via this map yields the *synchronisation abstraction* we denote by $\mathcal{H}(\Pi(\mathbf{X}, \Xi))$. The map $\Pi(\mathbf{X}, \Xi) \mapsto \mathcal{H}(\Pi(\mathbf{X}, \Xi))$ is a morphism for the composition operator $\mid$. Signals $\mathtt{h}$ possessing the synchronisation type $\{\perp, \top\}$ are called *clocks*. We equip the set $\{\perp, \top\}$ with the order $\perp < \top$ and we consider the associated $\cup$ (supremum), $\cap$ (infimum), and $\ominus$ (complement, defined by $k = h \ominus g$ if and only if $h = g \cup k$ and $g \cap k = \perp$). This provides us with the following operators on clocks:

$\mathtt{h} \cup \mathtt{g}$	supremum (i.e., the union of instants)
$\mathtt{h} \cap \mathtt{g}$	infimum (i.e., the intersection of instants)
$\mathtt{h} \ominus \mathtt{g}$	complement of $\mathtt{g}$ in $\mathtt{h}$

Asynchronous abstraction is possible *only when preorders are in fact partial orders* (i.e., no short circuit occurs), and is obtained as in the previous section by erasing those occurrences that are labelled with $\perp$. We do not define the $\mid$ composition operator on asynchronous abstractions, since taking asynchronous abstraction would not result in a morphism as the following example shows.

EXAMPLE. Consider the following two executions :

$$
\begin{array}{lcccccccccccccccccc}
\mathtt{H}: & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \dots \\
 & & & \downarrow & & & & \downarrow & & & & \downarrow & & & & \downarrow & & \dots \\
\mathtt{X}: & 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5 & \rightarrow & 6 & \rightarrow & 7 & \rightarrow & 8 & \rightarrow & \dots
\end{array}
$$

$$
\begin{array}{lcccccccccccccccccc}
\mathtt{H}: & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \top & \rightarrow & \perp & \rightarrow & \dots \\
 & & \searrow & & & & \searrow & & & & \searrow & & & & \searrow & & & \dots \\
\mathtt{X}: & 1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5 & \rightarrow & 6 & \rightarrow & 7 & \rightarrow & 8 & \rightarrow & \dots
\end{array}
$$

where `X,H` are the signals, the occurrences are figured by the vertices of the graphs, and the values written at these vertices are the second component of the label of these occurrences. These two executions are different, but they induce the same asynchronous abstraction. Hence, if Π_1 and Π_2 are the programs having each of these executions as respective single legal execution, then $\Pi_1|\Pi_2 = \emptyset$, so its asynchronous abstraction is also the empty program. On the other hand, since both Π_1 and Π_2 have the same asynchronous abstraction, composing them via a $\mid$ operator would yield the same (nontrivial) asynchronous abstraction, since $\mid$ is idempotent.

5.4 Summary

Since we introduced it stepwisely, it is useful at this stage to sumarize our model. We first provide a discussion of our model, and then we summarize it.

Discussion

The features of our final model are summarized now :

1. This model properly formalises both notions of systems of equations and block-diagrams, i.e., its parallel composition construct corresponds to

   – the *intersection* of behaviours of shared signals,

   – the *least common extension* of preorders of shared signals.

   Furthermore, thanks to the introduction of states, we have provided an explicit formula for the composition of our programs. Also, programs where only *partial orders* occur (i.e., short circuits are not encountered) can be desynchronised as indicated before.

2. This model allows us to mix the two different worlds of systems of equations and of block-diagrams, this is achieved by embedding the framework of behavioural semantics into that of operational semantics: behaviours are just obtained by removing preorders involving different signals or states.

3. It (fortunately) happens that both our theoretical analysis and resulting model and our previous informal analysis of Section 3 and resulting μGC language converge to the same solution. This claim will be supported in the next section by using our resulting model as a mathematical semantics for our tiny μGC language.

Summary of the model

Definition of PomEvent$(\mathbf{X}, \Xi)$. We consider prossets that are labelled over the domain

$$\text{Domain}(\mathbf{X}, \Xi) =_{\text{def}} \ \mathbf{X} + \sum_{\mathtt{x}\in\mathbf{X}} D_{\mathtt{x}}^{\perp} + \sum_{\xi\in\Xi} D_{\xi}$$

where the elements of $\mathbf{X}$ are the signals those of Ξ are the states. The preorder relation is written $\rightarrow$ and the associated equivalence relation is written $\leftrightarrow$. We restrict ourselves to those prossets satisfying Condition 2 which we recall for the sake of completeness:

1. For any signal $\mathtt{x} \in \mathbf{X}$, there exists a unique element $u \in D_{\mathtt{x}}^{\perp}$ such that the projection $\Phi_{(\mathbf{X},\Xi)\mapsto\mathtt{x}}(o)$ has the form $\mathtt{x} \rightarrow u^{\mathtt{x}} \rightarrow \mathtt{x}$, where $\rightarrow$ is the preorder relation; furthermore, the two extremal elements of $\Phi_{(\mathbf{X},\Xi)\mapsto\mathtt{x}}(o)$ are also extremal in o.

2. For any state $\xi \in \Xi$, there exists a finite integer $k \geq 0$ and there exist $u_0, \ldots, u_{k+1} \in D_\xi$ such that the projection $\Phi_{(\mathbf{X},\Xi)\mapsto\xi}(o)$ has the form $u_0^\xi \rightarrow \ldots \rightarrow u_{k+1}^\xi$, where $\rightarrow$ is the preorder relation; furthermore, the two extremal elements of $\Phi_{(\mathbf{X},\Xi)\mapsto\xi}(o)$ are also extremal in o.

Concatenation. The concatenation $o.o'$ is defined next. First, Pomset$(\mathbf{X}, \Xi)$ is enlarged with a zero element we call NIL, which satisfies the following properties:

$$\begin{aligned}
\forall o \in \text{Pomset}(\mathbf{X}, \Xi) &\ : \ \text{NIL}.o = o.\text{NIL} = \text{NIL} \\
\forall o, o' \in \text{Pomset}(\mathbf{X}, \Xi) &\ : \ \max\{o\} \neq \min\{o'\} \ \Rightarrow \ o.o' = \text{NIL} \\
\forall o \in \text{Pomset}(\mathbf{X}, \Xi) &\ : \ \text{NIL} + o = o + \text{NIL} = o
\end{aligned}$$

where the expression "$\max\{o\} \neq \min\{o'\}$" (resp. "$\max\{o\} = \min\{o'\}$") means that the set of labels of maximal elements of o and that of minimal elements of o' are different (resp. identical)[12]. Thus, NIL is absorbing for the concatenation, is the neutral element for the union $+$, and it has to be interpreted as the "absence of behaviour". Now if $\max\{o\} = \min\{o'\}$ holds, then $o.o'$ is the order obtained by merging together maximal elements of o and minimal elements of o' sharing the same labels, and then by erasing them. Hence a mismatch between maximal and minimal elements of adjacent items in a chain of the form $o_1, o_2, o_3, \ldots$ causes this chain

[12] Note that the third property involves the "+" operation, which is not related to concatenation; this property is essential for the desired properties of our regular expressions, however.

to collapse into the absorbing element NIL. This particular definition of concatenation plays a central role in "propagating" values through instants. Concatenation extends to sets of orders, and together with set theoretic union, allows us to define a $[\ldots]^\star$, and $[\mathrm{PomEvent}(\mathbf{X},\Xi)]^\star$ is a monoid, whose elements are still generically denoted by the symbol π.

Projections and parallel composition. If $(\mathbf{X}',\Xi') \subseteq (\mathbf{X},\Xi)$, the canonical projection

$$\Phi_{(\mathbf{X},\Xi)\mapsto(\mathbf{X}',\Xi')}(\pi)\ ,\ \pi \in [\mathrm{ProsEvent}(\mathbf{X},\Xi)]^\star$$

is the prosset obtained by deleting in π every element not labelled over $\mathrm{ProsEvent}(\mathbf{X}',\Xi')$. To define the parallel composition, we introduce the following partial order $\leq_\pi$ on $[\mathrm{ProsEvent}(\mathbf{X},\Xi)]^\star$. Consider two elements $\pi_1 = \{S_1, \rightarrow_1, v_1\}$ and $\pi_2 = \{S_2, \rightarrow_2, v_2\}$ of $[\mathrm{ProsEvent}(\mathbf{X},\Xi)]^\star$, where, for $i = 1,2$, S_i denotes the set of vertices of the preorder $\rightarrow_i$, and v_i is the labelling function. We say that

$$\pi_1\ \leq_\pi\ \pi_2$$

if $S_1 = S_2$, $v_1 = v_2$, and if $\rightarrow_2$ is an extension of $\rightarrow_1$. Hence the two prossets have the same labels, and the preorder of the greatest one is an extension of that of the least one. Then, for $i = 1,2$, let $\Pi_i \subseteq [\mathrm{ProsEvent}(\mathbf{X_i},\Xi_i)]^\star$ two programs, and define $\mathbf{X} = \mathbf{X_1} \cup \mathbf{X_2}$, $\Xi = \Xi_1 \cup \Xi_2$. Then [13]

$$\begin{array}{l}\Pi_1 \mid \Pi_2 \text{ is the set of } \textit{minimal}\ \pi \in [\mathrm{ProsEvent}(\mathbf{X},\Xi)]^\star \text{ such that,}\\ \quad \text{for } i = 1,2\ ,\ \exists \pi_i \in \Pi_i(\mathbf{X_i},\Xi_i)\ :\ \pi_i\ \leq_\pi\ \Phi_{(\mathbf{X},\Xi)\mapsto(\mathbf{X_i},\Xi_i)}(\pi)\end{array}$$

The composition " | " is associative, commutative, and idempotent (i.e., $\Pi|\Pi = \Pi$). We are ready to use this model for the mathematical semantics of μGC.

5.5 The mathematical semantics of μGC

In this subsection, we use our model to establish the mathematical semantics of μGC. To this purpose, reference is made to Table 2 of the appendix.

Declaration of actions and states.

- The declaration

  ```
  metagc PROG

    act ACT_1,...,ACT_p;...
    state Xi init Xi_o
  ```

  where `ACT_1,...,ACT_p` is the family of actions sharing state `Xi` , creates a state ξ with at most p occurrences per event, i.e., associated chains in Condition 2 take the form $u_0^\xi \rightarrow \ldots \rightarrow u_{k+1}^\xi$ where $k \leq p$. The subset, say, of cardinality k, of those occurrences of actions `ACT_1,...,ACT_p` , that are present at a given instant, is mapped onto the set of labels $\{u_1^\xi, \ldots, u_k^\xi\}$ in a nondeterminate way. Also, `Xi_o` is the initial condition u_0^ξ of the head element in the star operation.

[13] in the following formula, "minimal" refers to the order $\leq_\pi$ on prossets.

Clock equations. This part of the semantics corresponds to

$$
\begin{array}{rcl}
\text{Clockeq} & ::= & \boxed{\texttt{clock(}}\text{Name}\boxed{\texttt{)}}\ \boxed{\texttt{=}}\ \text{Clockexp} \\
& & \mid\ \text{Flow\_name}\ \boxed{\texttt{=}}\ \text{Clockexp} \\
\text{Clockexp} & ::= & \boxed{\texttt{base}}\mid\ \text{Flow\_name} \\
& & \mid\ \text{Clockexp Clockop Clockexp} \\
& & \mid\ \boxed{\texttt{true(}}\text{Flow\_name}\boxed{\texttt{)}} \\
\text{Clockop} & ::= & \boxed{\texttt{default}}\ \mid\ \boxed{\texttt{when}}\ \mid\ \boxed{\texttt{whennot}}
\end{array}
$$

- `H = K default/when/whennot L` corresponds to clock equations $H = K \cup / \cap / \ominus L$
- `clock(X) = H` corresponds to $H_X = H$
- `H = true(B)` has as a semantics

$$
\left[\begin{pmatrix} \mathtt{H} & \to & \top^{\mathtt{H}} & \to & \mathtt{H} \\ & & & & \\ \mathtt{B} & \to & [\text{true}]^{\mathtt{B}} & \to & \mathtt{B} \end{pmatrix} + \sum_{u \in B_{\mathtt{B}}^{\perp}\ ,\ u \neq \text{ true}} \begin{pmatrix} \mathtt{H} & \to & \perp^{\mathtt{H}} & \to & \mathtt{H} \\ & & & & \\ \mathtt{B} & \to & u^{\mathtt{B}} & \to & \mathtt{B} \end{pmatrix}\right]^{\star}
$$

Also we need to define the special clock `base` :

$$
\left[\begin{pmatrix} \mathtt{base} & \to & \top^{\mathtt{base}} & \to & \mathtt{base} \end{pmatrix}\right]^{\star}
$$

Signal equations. This corresponds to

$$
\text{Floweq} \quad ::= \quad \text{Flow\_Name}\ \boxed{\texttt{=}}\ \text{Flowexp}\ \boxed{\texttt{at}}\ \text{Clockexp}
$$

The semantics of `X = f(Y,Z) at H` is

$$
\left[\sum_{y \in B_{\mathtt{Y}}, z \in B_{\mathtt{Z}}} \begin{pmatrix} \mathtt{H} & \to & \top^{\mathtt{H}} & \to & \mathtt{H} \\ \mathtt{X} & \to & f(y,z)^{\mathtt{X}} & \to & \mathtt{X} \\ \mathtt{Y} & \to & y^{\mathtt{Y}} & \to & \mathtt{Y} \\ \mathtt{Z} & \to & z^{\mathtt{Z}} & \to & \mathtt{Z} \end{pmatrix} + \sum_{x \in B_{\mathtt{X}}^{\perp}, y \in B_{\mathtt{Y}}^{\perp}, z \in B_{\mathtt{Z}}^{\perp}} \begin{pmatrix} \mathtt{H} & \to & \perp^{\mathtt{H}} & \to & \mathtt{H} \\ \mathtt{X} & \to & x^{\mathtt{X}} & \to & \mathtt{X} \\ \mathtt{Y} & \to & y^{\mathtt{Y}} & \to & \mathtt{Y} \\ \mathtt{Z} & \to & z^{\mathtt{Z}} & \to & \mathtt{Z} \end{pmatrix}\right]^{\star}
$$

where we recall that $\top$ denotes the presence of a clock. Note that f is a "usual" function, i.e., $\perp$ does not belong to its domain. This is the reason for having $\perp$ excluded from the range of the first sum. In the other hand, $\perp$ does belong to the range of the second sum. For instance, we have $B_{\mathtt{Y}}$ in the first range while we have $B_{\mathtt{Y}}^{\perp}$ in the second one.

Imperative part. This corresponds to μGC :

Imper	::=	Act_Name $\boxed{\texttt{:}}$ Act
Act	::=	input | output
input	::=	State_Name $\boxed{\texttt{=}}$ Flow_Name $\boxed{\texttt{at}}$ Clockexp
output	::=	Flow_Name $\boxed{\texttt{=}}$State_Name $\boxed{\texttt{at}}$ Clockexp

We already have given the semantics of action declaration. What remains to be done is to give the semantics of "Act". Since states are local, we must provide the joint semantics of all actions sharing a given state and their associated preorders. We give only the semantics of a sample case, others are derived similarly. The program

```
( IN_ACT  : Xi = X     at H
| OUT_ACT : Y  = f(Xi) at H
| OUT_ACT --> IN_ACT   at H )
```

has as a semantics formula (14), where $S(v)$ and $pre(v,u)$ are suitably modified to account for `H` , i.e.,

$$S(v) = \left(\begin{array}{ccccc} \mathtt{H} & \to & \perp^{\mathtt{H}} & \to & \mathtt{H} \\ \\ v^{\xi} & \to & \to & \to & v^{\xi} \\ \\ \mathtt{X} & \to & \perp^{\mathtt{X}} & \to & \mathtt{X} \\ & & \downarrow & & \\ \mathtt{Y} & \to & \perp^{\mathtt{Y}} & \to & \mathtt{Y} \end{array}\right), \; pre(v,u) = \left(\begin{array}{ccccccc} \mathtt{H} & \to & \top^{\mathtt{H}} & \to & \to & \to & \mathtt{H} \\ \\ \mathtt{X} & \to & \to & \to & u^{\mathtt{X}} & \to & \mathtt{X} \\ & & & & \downarrow & & \\ v^{\xi} & \to & v^{\xi} & \to & u^{\xi} & \to & u^{\xi} \\ & & \downarrow & & & & \\ \mathtt{Y} & \to & v^{\mathtt{Y}} & \to & \to & \to & \mathtt{Y} \end{array}\right),$$

Preorders. We give the semantics of

Preceq	::=	Name $\boxed{\texttt{-->}}$ Name $\boxed{\texttt{at}}$ Clockexp
Name	::=	Flow_Name | Act_Name

For signals `X,Y` , the semantics of `X --> Y at H` is

$$\left[\sum_{x\in B^{\perp}_{\mathtt{X}},\, y\in B^{\perp}_{\mathtt{Y}}} \left(\begin{array}{ccccc} \mathtt{H} & \to & \top^{\mathtt{H}} & \to & \mathtt{H} \\ \\ \mathtt{X} & \to & x^{\mathtt{X}} & \to & \mathtt{X} \\ & & \downarrow & & \\ \mathtt{Y} & \to & y^{\mathtt{Y}} & \to & \mathtt{Y} \end{array}\right) + \sum_{x\in B^{\perp}_{\mathtt{X}},\, y\in B^{\perp}_{\mathtt{Y}}} \left(\begin{array}{ccccc} \mathtt{H} & \to & \perp^{\mathtt{H}} & \to & \mathtt{H} \\ \\ \mathtt{X} & \to & x^{\mathtt{X}} & \to & \mathtt{X} \\ \\ \mathtt{Y} & \to & y^{\mathtt{Y}} & \to & \mathtt{Y} \end{array}\right)\right]^{\star}$$

Other cases are defined similarly.

The parallel composition construct is exactly defined by (19).

This establishes the semantics of μGC in terms of our model. This does not justify why we have proposed an adequate set of primitives, however. This point is not the subject of our paper, and we refer the reader to [4] for a mathematical justification of the claim that our set of primitives provides maximum expressive power in terms of timing constructs.

6 Conclusion

In this paper we have presented a synchronous data-flow model with the following features:

1. This model properly formalises both notions of *systems of equations* and of signal flow graphs or *block-diagrams*. This is achieved in the following way. Systems of equations are modelled by sets of legal *behaviours* of data-flows. Then the operational flavour of blocks diagrams (as conveyed by the arrows linking ports of the various boxes) is modelled by *preorders* rather than partial orders, so that short circuits be part of our model (rather than being forbidden). Finally, the parallel composition construct of this model corresponds to

   - the *intersection* of behaviours of shared signals,
   - the *least common extension* of preorders of shared signals,

   which is what the intuition behind systems of equations and hierachies of block-diagrams suggests. Also, thanks to our synchronous approach, parallel composition preserves regularity, so that, in particular, bounded memory and response time is preserved. Finally, programs where only *partial orders* occur (i.e., short circuits are not encountered) can be desynchronised, so that synchronous programs can have asynchronous implementations.

2. This model allows us to mix the two different worlds of systems of equations and of block-diagrams, this is achieved by embedding the framework of behavioural semantics into that of operational semantics.

3. *Compilation,* i.e., moving from a behavioural specification to an operational form for execution, is characterised in a fairly simple way. Since an operational form has been automatically attached to each μGC program, we just have to check whether this operational form meets the criteria for executability. If this is not the case, then the problem reduces to that of finding a different synctactic form of our original program, such that 1/ behaviours remain identical, 2/ the associated operational form meets criteria for executability. Developing appropriate techniques for such program rewritings is the core of the compilation theory for data-flow synchronous languages, see [6, 16, 24].

It (fortunately) happens that both our informal analysis of Section 3 and theoretical analysis of Sections 4 and 5 converge to the same solution, namely the definition of the μGC language. In turn, the μGC language served as a basis for the definition of the GC common format [31] as explained above.

ACKNOWLEDGEMENT : *The authors gratefully acknowledge Oded Maler for fruitful criticism of an earlier version of this manuscript.*

References

[1] E. A. ASHCROFT AND W. W. WADGE , *Lucid, the data-flow programming language* , Academic Press, 1985

[2] A. BENVENISTE, G. BERRY, Eds., *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1268–1336.

[3] A. BENVENISTE, G. BERRY, "Real-Time systems design and programming", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1270–1282.

[4] A. BENVENISTE, P. LE GUERNIC, Y. SOREL, M. SORINE, "A denotational theory of synchronous communicating systems", *Information and Computation,* vol. 99 n^{o}2, August 1992, 192–230.

[5] A. BENVENISTE, P. LE GUERNIC, "Hybrid Dynamical Systems Theory and the SIGNAL Language", *IEEE transactions on Automatic Control*, 35(5), May 1990, pp. 535–546.

[6] A. BENVENISTE, P. LE GUERNIC, C. JACQUEMOT, "Synchronous programming with events and relations: the SIGNAL language and its semantics", *Science of Computer Programming,* 16 (1991) 103–149.

[7] G. BERRY, "Real Time Programming: Special Purpose Languages or General Purpose Languages", 11th IFIP Wold Congress 1989, San Francisco.

[8] F. BOUSSINOT, R. DE SIMONE, "The ESTEREL language", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1293–1304.

[9] F. BOUSSINOT, Proposition de sémantique dénotationnelle, pour des réseaux de processus avec mélange équitable, *Theoretical Computer Science*, Vol. 82, 173–206, 1982

[10] J.D. BROCK AND W.B. ACKERMAN, Scenarios, a model of non determinate computation, in Proc. Conf. on Formal Definition of Programming Concepts, LNCS 107, Springer, Berlin, 252–259, 1981.

[11] P. CASPI, Clocks in Dataflow languages, *Theoretical Computer Science*, Vol. 94, 125–140, 1992

[12] S. EILENBERG, *Automata languages and machines*, Vol. A, Academic Press, New York and London, 1974.

[13] J. T. FEO AND D. C. CANN AND R. R. OLDEHOEFT, A report on the Sisal language project, *J. Par. Distrib. Comp*, Vol. 10, 349–366, 1990

[14] H. Gaifman and V. Pratt , Partial order models of concurrency, and the computation of functions, in *Proc. of Symposium on Logic in Computer Science*, North Holland, Amsterdam, 1987, 72-85

[15] N. Ghezal, S. Matiatos, P. Piovesan, Y. Sorel et M. Sorine, Un environnement de programmation pour multiprocesseur de traitement du signal, Research Report INRIA–Rocquencourt, 1990, N^o 1236

[16] N. Halbwachs, P. Caspi, D. Pilaud, "The synchronous dataflow programming language Lustre", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1305–1320.

[17] D. Harel and A. Pnueli, "On the Development of Reactive Systems" in *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol. 13 (K. R. Apt, ed.), Springer-Verlag, New York, 1985, pp. 477-498.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot, "Statemate: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering* **16** (1990), 403–414.

[19] Inmos Ltd, *The* Occam *programming manual,* Prentice Hall, 1984.

[20] G. Kahn , The semantics of a simple language for parallel programming, in *Proc. of IFIP 74 Congress*, North Holland, Amsterdam, 1974

[21] L. Lamport, "What good is temporal logic", in *Proc. IFIP 9th World Congress,* R.E.A. Mason Ed., North Holland, 657-668, 1983.

[22] E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing", *IEEE Trans. on Computers,* Jan. 1987, C-36(2).

[23] E.A. Lee, "Consistency in Data–Flow graphs", Research Report UCB/ERL M89/125, Electronics Research Lab., College of Eng., U.C. Berkeley, 1989, to appear in *IEEE Trans. on Parallel and Distributed Systems.*

[24] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, "Programming real-time applications with Signal", *Another look at real-time programming*, special section of *Proc. of the IEEE,* vol. 9 n^o 9, September 1991, 1321–1336.

[25] Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems — Specification,* Springer Verlag, 1992.

[26] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Vol. 92, Springer Verlag, Berlin, 1980.

[27] R. Milner, "Turing award lecture : elements of interaction", CACM, Vol 36, N^{o}1, Jan. 1993, 78–89.

[28] J. Misra , Equational reasonning about nondeterministic processes, in *Formal Aspects of Computing*, 1990, 167–195

[29] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends", in *Current Trends in Concurrency* (de Bakker et al., eds.), Lecture Notes in Comput. Sci., Vol.224, Springer-Verlag, Berlin, 1986, pp.510–584.

[30] A. Rabinovitch, Pomset semantics is consistent with data flow semantics, *EATCS Bull.*, 107–117, 1987

[31] The C^2A group: *Projet SYNCHRONE, Les formats communs des langages synchrones,* INRIA Tech. Rep. n^o 157, June 1993.

Appendix : a syntax of μGC

prog	::=	$\boxed{\texttt{metagc}}$ Prog_Name { $\boxed{\texttt{act}}$ List(Act_Name)} { $\boxed{\texttt{state}}$ List(State) } Body $\boxed{\texttt{end}}$ Prog_Name
State	::=	State_Name $\boxed{\texttt{init}}$ Parameter
List(Item)	::=	Item | Item $\boxed{\texttt{,}}$ List(Item)
Body	::=	Clockeq | Preceq | Floweq | Imper | Prog_Name $\boxed{\texttt{[}}$ List(Renaming) $\boxed{\texttt{]}}$ | $\boxed{\texttt{(}}$ Body $\boxed{\texttt{)}}$ | Body $\boxed{\texttt{\|}}$ Body
Renaming	::=	Local_Name $\boxed{\texttt{:}}$ Name
Clockeq	::=	$\boxed{\texttt{clock(}}$ Name $\boxed{\texttt{)}}$ $\boxed{\texttt{=}}$ Clockexp | Flow_name $\boxed{\texttt{=}}$ Clockexp
Clockexp	::=	$\boxed{\texttt{base}}$ | Flow_name | Clockexp Clockop Clockexp | $\boxed{\texttt{true(}}$ Flow_name $\boxed{\texttt{)}}$
Clockop	::=	$\boxed{\texttt{default}}$ | $\boxed{\texttt{when}}$ | $\boxed{\texttt{whennot}}$
Preceq	::=	Name $\boxed{\texttt{-->}}$ Name $\boxed{\texttt{at}}$ Clockexp
Name	::=	Flow_Name | Act_Name
Floweq	::=	Flow_Name $\boxed{\texttt{=}}$ Flowexp $\boxed{\texttt{at}}$ Clockexp
Imper	::=	Act_Name $\boxed{\texttt{:}}$ Act
Act	::=	input | output
input	::=	State_Name $\boxed{\texttt{=}}$ Flow_Name $\boxed{\texttt{at}}$ Clockexp
output	::=	Flow_Name $\boxed{\texttt{=}}$ State_Name $\boxed{\texttt{at}}$ Clockexp

Table 2: *A syntax of μGC ; key symbols are inside boxes.*