Science of Computer Programming 16 (1991) 103-149
Elsevier

Synchronous programming with events and relations: the SIGNAL language and its semantics

Albert Benveniste, Paul Le Guernic
IRISA/INRIA, Campus de Rennes Beaulieu, 35042 Rennes Cedex, France

Christian Jacquemot
CNET PAA/OGE/SML, 3 Av. de la Republique, 92131 Issy-les-Moulineaux, France

Communicated by G. Berry
Received February 1989
Revised January 1991

Abstract

Benveniste, A., P. Le Guernic and C. Jacquemot, Synchronous programming with events and relations: the SIGNAL language and its semantics, Science of Computer Programming 16 (1991) 103-149.

In this paper, systems which interact permanently with their environment are considered. Such systems are encountered, for instance, in real-time control or signal processing systems, C^3-systems, man-machine interfaces, to mention just a few. The design and implementation of such systems require a concurrent programming language which can be used to verify and synthesize the synchronization mechanisms, and to perform transformations of the concurrent source code to match a particular target architecture. Synchronous languages are convenient tools for such a purpose: they rely on the assumption that: (1) internal actions of synchronous systems are instantaneous, and (2) communication with the environment is performed via instantaneous flashes involving some external stimuli. In this paper, we present a synchronous programming language: SIGNAL. A SIGNAL program specifies dynamical relations between (internal and external) signal flows. The SIGNAL compiler checks deadlock and determinism of the program, and produces an intermediate level code equivalent to a nested family of concurrent automata. The compilation algorithm is supported by: (1) a behavioural semantics of SIGNAL programs in terms of conditional rewriting rules, (2) the coding of this semantics into the skew product of a dynamical system over the field of integers modulo 3 and directed graphs, (3) an algebraic algorithm to transform the above coding into an equivalent executable one, which provides by the way an execution semantics of the language. We briefly discuss the implementation aspects, and explain the capabilities and limitations of the current version of the SIGNAL compiler.

Chapter 1. Introduction

Reactive systems[1], i.e. systems which interact permanently with their environment are considered in this paper. Such systems are encountered, for instance, in real-time

[1] This name was introduced in [22], and extensively used in [10].

0167-6423/91/$03.50 © 1991—Elsevier Science Publishers B.V.

control or signal processing systems, C^3-systems, man-machine interfaces, to mention just a few. It is usually recognized that a reliable design of such systems should be supported by a *concurrent* programming style. On the other hand, the highly demanding nature of these applications forces to consider as well the requirement of highly efficient and reliable implementation, in both cases of sequential or distributed implementation. Unfortunately, the modular structure of the source application may be different from the modular structure of the implementation. To summarize, the design and implementation of reactive systems requires a concurrent programming language which can be used to verify and synthesize the synchronization mechanisms, and to perform transformations of the concurrent source code to match a particular target architecture.

We shall not discuss here the drawbacks and merits of current tools in programming reactive systems (finite state machines, Petri Nets, concurrent programming languages such as ADA or OCCAM); the interested reader is referred to the excellent discussion in [10] on this subject. We shall merely concentrate on the discussion of the *synchronous approach* we follow in this paper.

1.1. The basic synchronicity hypotheses

While classical (i.e. asynchronous) concurrent languages do implicitly or explicitly refer to some external and universal time reference, the notion of 'time' is completely different in synchronous reactive systems. To be more explicit, synchronous reactive systems differ from asynchronous ones in the following aspects:

(1) *The internal mechanisms of the system*: every action (computation or internal communication) is instantaneous, i.e. has a zero duration;

(2) *The communications with the external world*: the set of the possible input channels is fixed and known in advance, and the flows carried by these channels are specified through both

- the values they carry,
- a total ordering of the 'instants' at which these values are available at the external ports.

Of course, this last requirement is the fundamental feature which characterizes the way synchronous reactive systems communicate with the external world, compared to asynchronous ones. Let us illustrate this point using a simple example. Consider a reactive system with two inputs:

(1) a data input carrying an ordered file of data named x,
(2) an interrupt input port named s.

Then, the specification of an input history according to the synchronous point of view must be of the form

$$\begin{array}{lllll} x_1 & x_2 & x_3 & \perp & \\ \perp, & s_2, & \perp, & s_4, & \text{etc.} \end{array}$$

(as usual, $\perp$ denotes the absence of data), i.e. both the values and their global interleaving must be specified: the integer index $t = 1, 2, \ldots$ is used for this purpose. *This index 't' has to be considered as the proper notion of time in synchronous systems.* Another fundamental consequence is that the notion of time is *local to a given subsystem*: there is no universal time reference, as we shall see later when communications will be studied.

In other words, the essentially non-deterministic character of the communications with the external world in reactive systems is concentrated inside some (ignored) external mechanism which *decides* this global ordering. Hence, the advantage of the synchronous point of view is that *the non-determinism of external communications is strictly concentrated on this mechanism, and it is not propagated inside the body of the system itself.* This is the fundamental reason of the power of the synchronous approach, as far as deep program transforms is concerned. Among languages relying on this synchronicity assumption are the imperative language ESTEREL [20, 10], the declarative and functional language LUSTRE [17, 31], and the declarative and relational language SIGNAL we present in this paper; related to the same formalism is also the approach of *statecharts* in [21].

1.2. On the semantics of SIGNAL

To support the above program transforms, SIGNAL must rely on a mathematical abstract model; such a model and the language were developed simultaneously. In fact, two models of different styles were introduced.

1.2.1. A denotational semantics of SIGNAL

To our knowledge, the pioneering work relevant to the denotational style of semantics is the *Dynamic Network Processes* model introduced in [24]. DNP's are functions mapping input histories into output histories; their denotational semantics has been studied in detail in [16]. Kahn's model has been used with suitable extensions and modifications in [31] to cope with the synchronicity assumption. However, this approach cannot be used for SIGNAL, due to its relational nature. In [2, 4], a denotational *relational* model has been introduced for the SIGNAL language. According to this model, processes specify restrictions on the set of all possible interleavings of the involved signals (or data-flows); such restrictions may involve values (e.g. in a relation such as x = y + z which means that $\forall t$: $x_t = y_t + z_t$), or synchronization (e.g. in an instruction such as y = x ***when*** b). This allows an elegant definition of the notion of *communication* within this model; it is shown in [2] that observational equivalence is a congruence within this model, a surprising result compared to most models of concurrent systems [11]. Using this model, it is shown in [2] that SIGNAL is 'complete', i.e. provides a set of constructs which is sufficient to build any reactive system. It is not our purpose to discuss this model any further, the interested reader is referred to the above mentioned references.

1.2.2. An operational semantics

Inspired by [9] we give in this paper a semantics for SIGNAL in terms of conditional rewriting rules à la Plotkin [32]. Let us first discuss the consequences of the synchronicity assumption on the kind of process algebra we get; we shall refer to the classical notations of SCCS [28] for this purpose. We consider transitions of the form $E \xrightarrow{a} E'$, where E and E' range over expressions of the process algebra, and a ranges over the abelian group of *actions* $\{Act, \times, 1, \bar{a}\}$ where $\bar{a}$ denotes the inverse of a within the group; the derivation rules of SCCS are given in [28, p. 276, Table 1]. Referring to this table, the sum $E+F$ represents a non-determinate choice among the actions E or F can perform. The *sum* is known to be the cause of non-determinism and other features such as the distinction between observational equivalence and bisimulation; such a composition operator is not in agreement with the synchronicity assumption, and will not be used here. We shall only use a refinement of the *product* of SCCS we shall outline now. Think of E performing any one of the following actions: $!x$, $!y$, $!x!y$ to be interpreted as the delivery of x alone, y alone, or x and y simultaneously; on the other hand, consider E' as performing $?x?y$ (read x and y simultaneously) as only action, for instance in order to compute their sum $x+y$. Assume that $\overline{!x} = ?x$. Then the SCCS product $E \times E'$ is allowed to perform any one of the actions $!x$, $!y$, $?x?y$, $!x?x?y = ?y$, $!y?x?y = ?x$, $!x!y?x?y = 1$, i.e. *any product* of the original actions of E and E'. The synchronicity assumption will be reflected in the fact that $!x!y?x?y$ will be the only acceptable action the composition of E and E' can perform. In other words, the set of actions is not an abelian group any more, for the product ab of two actions a and b is not always defined; $c = ab$ will merely be considered as a *relation* on actions. An immediate consequence is that combining expressions via such a modified composition operator immediately yields actions which have the form of *an implicit system of equations*; solving such equations provides equivalent *explicit* actions which can be executed. This is exactly the task performed by the compilers of all synchronous languages (ESTEREL, LUSTRE, SIGNAL). This short discussion enlightens the difference between models of synchronous languages and others.

1.3. Organization of the paper

Chapter 2 is devoted to an introduction to SIGNAL and its illustration via a programming example. Our purpose is to show how the relational features of SIGNAL can be used for a 'self-proved' style of programming. The reader interested in a complete description of SIGNAL is referred to [19, 27], and other programming examples relevant to signal processing can be found in [4]. In Chapter 3, the mathematical semantics of SIGNAL is presented; an algebra of processes specified by Plotkin-like transition rules is introduced for this purpose; projecting this algebra into a smaller one provides us with transition rules which summarize synchronization, logic, and dependencies. A coding of this subalgebra is presented, using the field $\mathbb{F}_3$ of integers modulo 3. Chapter 4 is the core of the paper: the execution semantics

of SIGNAL is presented and it is shown how the compiler can check properties such as deadlock or determinism. Finally, related results and the current status of the SIGNAL compiler are presented in the conclusion.

Chapter 2. The SIGNAL programming language; some examples

To be concise, we shall introduce only the primitives of the SIGNAL language,[2] and drop any reference to typing and various declarations; the interested reader is referred to [27].

2.1. SIGNAL-kernel

SIGNAL handles (possibly infinite) sequences of data with time implicit: such sequences will be referred to as *signals*. For example, x denotes the infinite sequence $\{x_t\}_{t\geq 1}$ where the integer time index t is attached to this signal; signals possessing the same time index are said to have the same *clock*, so that clocks are equivalence classes of simultaneous signals (a formal definition will be discussed later). Instructions of SIGNAL are intended to relate clocks as well as values of the various signals involved in a given reactive system. We shall term a system of such relations *program*; programs will be used as modules.

A basic principle in SIGNAL is that a single name is assigned to every signal, so that in the sequel (and unless explicitly stated), identical names refer to identical signals. The kernel-language SIGNAL possesses 5 instructions, the first of them being a generic one.

(i) p(x1, ..., xn)
(ii) y := x$ ***init*** x_0
(iii) y := x ***when*** b
(iv) y := u ***default*** v
(v) P|Q

The intuitive meaning of these instructions is as follows:

(i) Direct extension of instantaneous relations into relations acting on flows:

$$\mathsf{p(x1, \ldots, xn)} \quad \Leftrightarrow \quad \forall t\colon p(x1_t, \ldots, xn_t) \tag{2.1}$$

For example, functions such as z := x+y ($\forall t$: $z_t = x_t + y_t$) or statements such as (a ***and*** b) ***or*** c := *true* ($\forall t$: (a_t and b_t) or c_t = *true*). A byproduct of this instruction is that *all referred signals must have the same time index, i.e. they must be present simultaneously.* This is a generic instruction, i.e. we assume a family of relations is available. If one chooses an instantaneous relation accepting any n-uple, the resulting SIGNAL instruction only constrains the involved signals to have the same clock: the so-obtained instruction written ***synchro*** x, y, ... only forces the two signals x, y, ... to have the same clock.

[2] SIGNAL is a joint trademark of CNET and INRIA.

(ii) Shift register:[3]

$$y := x\$ \ \textit{init}\ x_0 \quad \Leftrightarrow \quad \forall t > 1: y_t = x_{t-1}, y_1 = x_0.$$

Again this instruction forces the input and output signals to have the same time index, i.e. to be present simultaneously.

(iii) Condition (b is boolean): y equals x when the signal x and the boolean b are available and b is true; otherwise, y is not emitted; the result is an event-based downsampling of signals. Here follows an example of behaviour of this instruction (as mentioned before, $\perp$ denotes the absence of data):

b:	$\perp$	$\perp$	*ff*	*ff*	*tt*	$\perp$	*tt*	$\perp$	*ff*
x:	x_1	$\perp$	$\perp$	x_2	x_3	x_4	$\perp$	$\perp$	x_5
y:	$\perp$	$\perp$	$\perp$	$\perp$	x_3	$\perp$	$\perp$	$\perp$	$\perp$

When this instruction alone is being observed, $\perp$ meaning the absence of data can be deleted, so that events with triple $\perp$ are removed by the way:

b:		*ff*	*ff*	*tt*		*tt*	*ff*
x:	x_1		x_2	x_3	x_4		x_5
y:				x_3			

(iv) y merges u and v, with priority to u when both signals are simultaneously present; this instruction is the key to oversampling as we shall see later. Here follows an example of behaviour of this instruction:

u:	u_1		u_2	u_3		u_4	
v:		v_1	v_2		v_3	v_4	v_5
y:	u_1	v_1	u_2	u_3	v_3	u_4	v_5

The instructions (i)-(iv) specify the elementary programs, which we call *generators*. The objects named x, y, u, v, b will be termed *signals.*

(v) Communication of already defined programs: P and Q communicate through their signals with common names; for example

$$y := zy + a \mid \quad zy := y\$ \ \textit{init}\ x_0$$

denotes the system of recurrent equations for $t \geq 1$

$$y_t = zy_t + a_t,$$

$$zy_t = y_{t-1}, \quad zy_1 = x_0$$

which is equivalent to $y_t = y_{t-1} + a_t$, $y_0 = x_0$.

[3] Time is money.

2.2. The shared track example and its SIGNAL programming

This example is borrowed from [30].

2.2.1. Informal description of the example

Consider a plant which consists of two diesel trains which share a common section of railway track. On the shared track, there is a diesel pump for refueling the trains. There is an automatic mechanism which allows a controller to sample the level of diesel in the tank of the train, and a facility exists for the controller to command the pump to deliver any amount of fuel to the train. Since we wish to prevent the disastrous situation of two trains simultaneously occupying the shared track, two traffic lights have been installed. Each train is allocated a traffic light at its entrance to the shared track. A train waits before entering the shared track until it receives the signal to move. We shall now discuss the SIGNAL programming of this example.

2.2.2. Some macros

To allow for an easy description of complex objects, we shall build a toolbox of macros to be used later as standard instructions. When presenting the macros, to avoid the need for explicit typing, we shall use the following generic notations:

- u, x, y, z, ... : signal of any type,
- a, b, c, ... : boolean signals,
- h, k, l: signals of type *event*, i.e. boolean signals which take only the value *true*; type *event* is naturally embedded into type *boolean*, this will be used in the sequel.

(1) Access to the clock of a signal:

h := ***event***(x)

stands for

h := (x = x).

The pure clock h is delivered when x is present (since x = x always holds).

(2) Extraction of the occurrences *true* of a boolean signal:

h := ***when***(b)

stands for

h := b ***when*** b.

(3) Requiring that two events never happen at the same instant:

disjoint h, k

stands for

```
l := when((not(h when k)) default h)
synchro l, h.
```

Notice that the event l occurs when h occurs but not k. The 'synchro' instruction forces l and h to be simultaneous.

(4) A synchronized memory:

y := x ***cell*** b ***init*** y_0

stands for

```
zy := y$ init y0
y := x default zy
synchro y, (x default when(b)).
```

The output y returns either the present value of x (when x is received), or the last received value of x when b is present and true.

2.2.3. An outline of the complete SIGNAL language

It is not our purpose here to fully present SIGNAL, the interested reader is referred to [27]. The basic construct in SIGNAL is the PROGRAM; programs are used as modules or 'black-boxes'. Here follows the corresponding notation:

```
NEW_PROGRAM (list of parameters) {list of visible signals}
      =
   | OLD_PROGRAM_1 list of a_1:b_1
   | ...
   | OLD_PROGRAM_n list of a_n:b_n
where
   OLD_PROGRAM_1 ...
   ...
   OLD_PROGRAM_n ...
end
```

In this notation, a_i : b_i means that the signal a_i of OLD_PROGRAM_i is renamed to b_i; this mechanism is the basis for program interconnection, since visible signals

denoted by identical names must be identical. Signals which appear in the body of the program but are not listed in its interface are local signals, i.e. they are invisible from outside. To facilitate the reading, it will be useful to mark some of the interfaces with ? or ! to mean that the corresponding signal interfaces are *inputs* or *outputs* respectively; this will be used in the sequel.

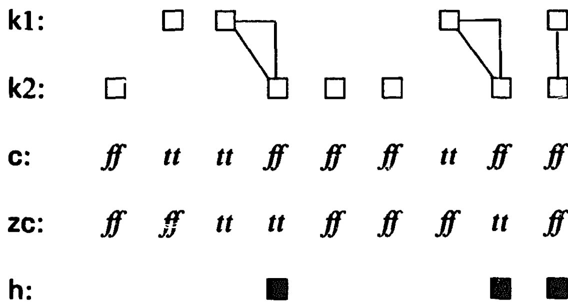
2.2.4. *Some basic mechanisms*

2.2.4.1. *A guard on interleaved signals* ('followed by')

The event h is emitted when k2 occurs simultaneously with or immediately after k1 when the latter is present alone. Here follows the program:

```
FBY {? event k1, k2   ! event h}
        =
   | c := (not k2) default k1
   | zc := c$ init false
   | h := k2 when (k1 default zc)
end
```

where we have used the fact that type ***event*** is embedded into type ***bool***; the internal signals c,zc are boolean built from events. Here is a picture of the corresponding temporal behaviour; the internal signals of the SIGNAL program below are also depicted:



The diagram above shows how the various signals are interleaved; symbols appearing in the same column are delivered simultaneously. This picture intuitively shows that ■ should occur exactly when triangles linking occurrences of □ can be inserted without intersecting other symbols. Notice that the space between the columns have no interpretation in terms of some 'regular' physical time, this diagram only specifies the global interleaving of the various signal flows.

2.2.4.2. *A guarded decreasing counter with reset*

The purpose of this counter is to model the internal consumption of time, or any other amount of events. Typically, the 'speed' of such counters are not visible

externally.

```
GUARD_COUNT (integer level) { ? event reset   ! bool empty}
        =
   | n := (level when r) default (zn − 1)
   | zn := n$ init 0
   | synchro (when(zn = 0)), r
   | e := when(n = 0) default (not r)
r: reset, e: empty
end
```

The parameter level is assumed to be >0. The first three instructions define a counter modulo level + 1 with a reset signal. When the event reset is received, the counter restarts decreasing from level, until the counter reaches 0. The boolean empty delivers *false* when the counter is reset, and *true* when the counter gets empty. The ***synchro*** instruction refuses any new reset before the counter gets empty. This is an example of an *active* program, i.e. of a program that *acts* on the timing of its inputs; this is a special feature of SIGNAL among all synchronous languages. Notice that the combination of the two basic constructs, ***default*** and $, allows us to generate internal clocks that are *faster* than the clocks of the interfaces, namely the clocks of the counters. This is a very powerful data dependent upsampling mechanism which is a byproduct of the relational style of SIGNAL. This program will be used repeatedly to represent the consumption of fuel in the tank of the train, or the delivery of fuel by the pump on the shared track.

2.2.5. Programming the shared track example

In this section, we shall also introduce an outline of SIGNAL modular programming.

2.2.5.1. The shared track acting as a critical section

When free, the track accepts any *single* train, and delivers the fuel. This program is a prototype of critical section. Here follows the program.

```
SHARED_TRACK (integer level) { ? event enter1, enter2   ! bool free}
        =
   | disjoint enter1, enter2
   | enter := enter1 default enter2
   | GUARD_COUNT(level) reset: enter, empty: free
where
      GUARD_COUNT (integer level)
        { ? event reset  ! bool empty} %already seen%
      end
end
```

where we show an outline of the SIGNAL syntax for modular programming. The critical section is the result of (1) GUARD_COUNT which refuses any new train to

proceed while the track is occupied, (2) the first instruction which prevents two trains to enter simultaneously. Notice that no particular priority among the two trains is specified here, so that the resulting program will be non-deterministic; such a non-determinism could be easily removed by assigning a different priority to each train. The critical section does not take care of the *request/acknowledgement* mechanism: the latter task will be devoted to each particular train; this choice corresponds to a decentralized style of control.

2.2.5.2. A train passing the shared track

Here we present the *request/acknowledgement* mechanism which is typical from critical sections.

```
TRAIN_ON_TRACK { ? bool request, free   ! event enter, travelstart}
      =
    | REQ_ACK acknowledge: enter
    | k2 := when(free)
    | FBY k1: enter, h: travelstart
where
    FBY { ? event k1, k2   ! event h} %already seen%
    end
    REQ_ACK { ? bool request, free   ! event acknowledge}
          =
        | cr := r when (f cell r init true)
        | cf := f when (r cell f init false)
        | a := when(cr) default when(cf)
    r: request, f: free, a: acknowledge
    end
end
```

The program REQ_ACK specifies the *request/acknowledgement* mechanism: acknowledgement is given when either a request occurs while the section being free, or the section gets free while a request has been sent. When allowed, the train proceeds on the track; then, it leaves the track when the pump gets empty; this last action is specified by the FBY program.

2.2.5.3. A train

```
TRAIN (integer level) { ? bool free   ! event enter}
      =
    | GUARD_COUNT(level) reset: travelstart, empty: request
    | TRAIN_ON_TRACK
where
    GUARD_COUNT (integer level)
      { ? event reset   ! bool empty} %already seen%
    TRAIN_ON_TRACK { ? bool request, free   ! event enter, travelstart}...
end
```

Here the program GUARD_COUNT figures the fuel consumption during the travel. The whole life of a particular train is described in this program.

2.2.5.4. The whole program

Here follows the program SNCF.[4]

```
SNCF (integer level) { ? %no input%   ! %what you want to observe%}
        =
    | SHARED_TRACK(level)
    | TRAIN(level) enter: enter1
    | TRAIN(level) enter: enter2
where %etc...%
end
```

The label i (i = 1, 2) refers to the particular train. Recall that this program is non-deterministic, but that it could be easily made deterministic if different priorities are assigned to each train.

This programming example reveals several interesting features of SIGNAL.

- Systems which permanently interact with their environment can be specified. The programs FBY and REQ_ACT are typical examples of *passive* programs, i.e. of programs accepting anything the environment proposes. On the other hand, the program GUARD_COUNT *acts* on its environment: inputs are accepted only when some *internal* condition is satisfied during the running of the program. Hence the generic word *reactive* we have used in the introduction should be accepted in a very wide sense: *reactions* can involve a complex mixture of *passive* and *active* interactions with the environment.
- *Closed* dynamical systems (i.e. with no input) can be specified as well. For example, the whole program SNCF simulates the interaction between the shared track and the trains.
- *Non-deterministic* programs can be composed to yield a *deterministic* one. For instance, the program SNCF without priority rule among trains is non-deterministic, while it can be made deterministic via composition with another program which specifies such a priority rule. Simpler examples of this kind are very frequently encountered in SIGNAL programming. A short practice of SIGNAL programming reveals how important such a facility is in providing user-friendly programming style.
- Finally, thanks to the relational nature of the language, *constraints* on synchronisation and logic can be stated within SIGNAL. Examples were repeatedly found in the critical section mechanism of the program SHARED_TRACK. Hence the SIGNAL language can be considered as a step towards *direct synthesis of reactive systems from their specifications.*

[4] Société Nationale des Chemins de Fer Français, a trademark from French government.

- A limitation of SIGNAL is also revealed here, namely trains cannot be *dynamically created*, they must be defined statically. This is a feature common to all synchronous languages, it might be inconvenient in some applications, but this is the price to pay for getting a language with powerful formal reasoning ability.

All these features characterize SIGNAL compared to the other synchronous languages ESTEREL and LUSTRE. As a counterpart, compiling SIGNAL programs will be a hard task for the following reasons:

- A SIGNAL program specifies relations between signals and their clocks via a system of equations. Hence two natural questions arise:
  (1) Does such a system of equations possess a *non-trivial solution* (i.e. a solution allowing events to occur), in other words *is the corresponding program deadlock-free*?
  (2) Is such a non-trivial solution unique (a question related to non-determinism)?
- The task of the compiler is to solve this system of equations in a sense that we formally discuss later; we could informally state that this specification has to be transformed into an effective machine which can produce the desired behaviours; such a machine should be a function mapping sequences of input stimuli (input histories) into sequences of output stimuli (output histories). One of the main steps of the compilation is the *synthesis* of the global synchronisation of the program from the relations between clocks specified within this program: this will be the purpose of the *clock calculus*.

Chapter 3. The mathematical semantics of SIGNAL

We shall first describe the mathematical semantics of SIGNAL using a suitable algebra of processes specified via transition systems following [32]. Then we shall introduce a subalgebra where reasoning can be performed about synchronization, logic, and data dependencies, and we shall present an algebraic coding of this subalgebra. This will completely define the behavioural semantics of SIGNAL, i.e. what SIGNAL programs should perform.

3.1. Notations: processes and transition systems

Definition 1. A *process* is a triple of the form

$$\Pi = \{\Xi, \mathsf{A}, \rightarrow\} \tag{3.1}$$

where

Ξ is the set of *states*: states are programs;
A is the set of *events*, events are denoted by

$$\mathsf{x1}\,(x_1)\ldots\mathsf{xn}\,(x_n) \tag{3.2}$$

or α for short; events are functions mapping a given set of *ports* (here x1, ..., xn) into a corresponding set of *values*, (here $x_1, \ldots, x_n$) which are said to be *carried* by these ports during this event; among the set of values is the distinguished value $\perp$ which has to be interpreted as the absence of data. Events where all ports carry the value $\perp$ are said to be *trivial*. From now on, italic letters such as '$x, y, \ldots$' will refer to effective values, as opposed to $\perp$ which will be always explicitly mentioned. If α is an event, we shall denote by $D(\alpha)$ its domain (i.e. the set of its ports). It will be useful to consider the event NIL with empty domain.

The symbol $\rightarrow$ denotes a *transition*; $\rightarrow$ is a *relation* defined on $\Xi \times \mathsf{A} \times \Xi$. This transition is defined by a set of *rules* of the form

$$\frac{C}{\mathsf{P} \xrightarrow{\alpha} \mathsf{P}'} \tag{3.3}$$

where P, P′ are programs (i.e. states), α is the considered event, and

$$C = \textit{statement}\ (\mathsf{P}, \alpha, \mathsf{P}') \tag{3.4}$$

is a statement involving the mentioned arguments; the meaning is "P can perform α and yield P′ provided that C holds".

Hence it is clear that firing a transition generally requires to solve an implicit equation since the precondition C can depend on the event and on the resulting new state. The successive firing of a sequence of transitions

$$\mathsf{P}_0 \xrightarrow{\alpha_0} \mathsf{P}_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{n-1}} \mathsf{P}_n \cdots \tag{3.5}$$

where all α_i's are non-trivial, defines a *provable run* of the program P_0.

We shall denote by $\gamma_{/D}$ the restriction of an event γ to a subset D of its domain. Given two events α and β belonging to (possibly) different processes, we introduce the predicate

$$\alpha \cap \beta \quad \Leftrightarrow \quad \alpha_{/D(\alpha) \cap D(\beta)} = \beta_{/D(\alpha) \cap D(\beta)} \tag{3.6}$$

to indicate that the corresponding maps agree on the intersection of their respective domains (this means that ports with identical names carry the same value). When this predicate holds, the two events are said to be *compatible*, and their union is defined as usually for maps, keeping only a single copy of the common ports in the resulting notation. Union is denoted by $\alpha \cup \beta$.

Processes form a commutative monoid endowed with the composition operation defined by

$$\Pi | \Pi' :: \frac{\mathsf{P} \xrightarrow{\alpha} \mathsf{P}'; \quad \mathsf{Q} \xrightarrow{\beta} \mathsf{Q}'; \quad \alpha \cap \beta}{\mathsf{P} | \mathsf{Q} \xrightarrow{\alpha \cup \beta} \mathsf{P}' | \mathsf{Q}'} \tag{3.7}$$

The identity element of this semi-group is the process NOTHING characterized by the property A = NIL. Notice that, when P and Q do not have any ports with common names, any interleaving of events from P and/or Q is valid.

3.2. Encoding SIGNAL into processes

The rules of the instruction (i)

$$\frac{p(x_1, \ldots, x_n)}{\mathsf{p}(\mathsf{x1}, \ldots, \mathsf{xn}) \xrightarrow{\mathsf{x1}(x_1)\ldots\mathsf{xn}(x_n)} \mathsf{p}(\mathsf{x1}, \ldots, \mathsf{xn})} \tag{3.8}$$

where $p(\ldots)$ denotes a relation, cf. (2.1). Here, the states of the transition are just the text of the instruction (there is no memory, hence the state is unchanged). The predicate requires that the values to be presented to the ports satisfy the relation p; if this holds, then these values can be accepted by the transition, and this acceptance is the even performed by the transition. For example consider the relation z := x + y, here the event has to be interpreted as the computation of the sum of the values carried by the ports x and y and its delivery at the port z. In the case of the ***synchro*** generator, no precondition is required: its only effect is then to force the clocks of all signals to be identical. For the encoding of the composition, we shall also need to consider the following trivial rule, which represents the empty event (in which case the program does not change):

$$\mathsf{p}(\mathsf{x1}, \ldots, \mathsf{xn}) \xrightarrow{\mathsf{x1}(\perp)\ldots\mathsf{xn}(\perp)} \mathsf{p}(\mathsf{x1}, \ldots, \mathsf{xn}) \tag{3.9}$$

All the instructions (ii)-(iv) should also be provided with a corresponding trivial transition, which will thus be omitted.

The rules of the instruction (ii)

$$\mathsf{y} := \mathsf{x}\$\ \textbf{\textit{init}}\ u \xrightarrow{\mathsf{x}(v)\ \ \mathsf{y}(u)} \mathsf{y} := \mathsf{x}\$\ \textbf{\textit{init}}\ v \tag{3.10}$$

where u is the content of the memory: it is delivered at the output y, while the value v received at the input port x is fed into the memory. Notice that the state has been modified via the modification of the parameter involved in the program.

The rules of the instruction (iii)

$$\begin{aligned}
&\text{(i)} && \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \ \mathsf{b}(\perp)\ \ \mathsf{y}(\perp)} \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b}\\
&\text{(ii)} && \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b} \xrightarrow{\mathsf{b}(b)\ \ \mathsf{x}(\perp)\ \ \mathsf{y}(\perp)} \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b}\\
&\text{(iii)} && \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \ \mathsf{b}(\mathit{true})\ \ \mathsf{y}(x)} \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b}\\
&\text{(iv)} && \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b} \xrightarrow{\mathsf{x}(x)\ \ \mathsf{b}(\mathit{false})\ \ \mathsf{y}(\perp)} \mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{b}
\end{aligned} \tag{3.11}$$

These four rules exactly encode the intuitive meaning of the instruction ***when***. No change in the state occurs.

The rules of the instruction (iv)

$$\text{(i)}\quad \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v} \xrightarrow{\mathsf{u}(u)\ \ \mathsf{y}(u)\ \ \mathsf{v}(\perp)} \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v}$$

$$\text{(ii)}\quad \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v} \xrightarrow{\mathsf{u}(u)\ \ \mathsf{v}(v)\ \ \mathsf{y}(u)} \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v}$$

$$\text{(iii)}\quad \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v} \xrightarrow{\mathsf{v}(v)\ \ \mathsf{y}(v)\ \ \mathsf{u}(\perp)} \mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v} \tag{3.12}$$

This is the exact coding of the intuitive meaning of this instruction. Again, no change in the state occurs, and no precondition is required.

The rules of the instruction (v)

This is the major step, since this instruction is the key to modular construction. In fact the corresponding coding has already been given in (3.7).

An example. We want to encode the program

P(n0) =
| n := reset ***default*** xn
| xn := zn − 1
| zn := n$ ***init*** n0

The rules of the three instructions, together with the trivial rules (3.9), are as follows:

first instruction:

$$\text{(i)}\quad \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn} \xrightarrow{\mathsf{reset}(u)\ \ \mathsf{n}(u)\ \ \mathsf{xn}(\perp)} \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn}$$

$$\text{(ii)}\quad \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn} \xrightarrow{\mathsf{reset}(u)\ \ \mathsf{xn}(v)\ \ \mathsf{n}(u)} \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn}$$

$$\text{(iii)}\quad \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn} \xrightarrow{\mathsf{xn}(v)\ \ \mathsf{n}(v)\ \ \mathsf{reset}(\perp)} \mathsf{n} := \mathsf{reset}\ \textbf{\textit{default}}\ \mathsf{xn}$$

second instruction:

$$\frac{z = w - 1}{\mathsf{xn} := \mathsf{zn} - 1 \xrightarrow{\mathsf{zn}(w)\ \ \mathsf{xn}(z)} \mathsf{xn} := \mathsf{zn} - 1}$$

third instruction:

$$\mathsf{zn} := \mathsf{n\$}\ \textbf{\textit{init}}\ n0 \xrightarrow{\mathsf{n}(x)\ \ \mathsf{zn}(n0)} \mathsf{zn} := \mathsf{n\$}\ \textbf{\textit{init}}\ x$$

Combining these rules according to (3.7) yields the set of rules to encode the program P. Combining (i) with the non-trivial transitions of the other instructions requires the precondition

$$[\mathsf{reset}(u)\ \mathsf{n}(u)\ \mathsf{xn}(\perp)] \cap [\mathsf{zn}(w)\ \mathsf{xn}(z)] \cap [\mathsf{n}(x)\ \mathsf{zn}(n0)]$$

which is never true, since by convention z denotes an effective value $\neq\perp$. On the other hand, combining (i) with the trivial rules for the other instructions yields the precondition

$$[\mathsf{reset}(u)\ \mathsf{n}(u)\ \mathsf{xn}(\perp)] \cap [\mathsf{zn}(\perp)\ \mathsf{xn}(\perp)] \cap [\mathsf{n}(\perp)\ \mathsf{zn}(\perp)]$$

which is also false. Hence rule (i) cannot be used. The only valid rules are

(ii) $\mathsf{P}(n0) \xrightarrow{\mathsf{reset}(u)\quad \mathsf{xn}(n0-1)\quad \mathsf{n}(u)\quad \mathsf{zn}(n0)} \mathsf{P}(u)$

(iii) $\mathsf{P}(n0) \xrightarrow{\mathsf{reset}(\perp)\quad \mathsf{xn}(n0-1)\quad \mathsf{n}(n0-1)\quad \mathsf{zn}(n0)} \mathsf{P}(n0-1)$

where allowed substitutions have been performed. The program $\mathsf{P}(n0)$ possesses reset as the only input; on the other hand, the two rules above show that this program, considered as an input/output transform, is non-deterministic, i.e. different provable runs can be produced which accept the same input sequence.

We have already shown how the algebra of SIGNAL programs can be mapped into the semi-group of processes. On the other hand, it should be clear from (3.4) that *performing events requires solving systems of equations*: *their resolution is needed.* Now, the following problem remains, which is the crux of the theory: *transform any process into a machine which can execute it.* This sort of job usually requires a convenient algebraic calculus to be at hand, and it should be clear that there is no hope for us to have this sort of algebra, since our model of processes is too general.

Hence we need a reduction technique: use a convenient homomorphism from the semi-group of processes into itself such that its range is

- small enough to provide some suitable algebraic calculus,
- rich enough to still provide us with a convenient solution to our problem.

We address this issue in the following section.

3.3. A calculus of synchronization

3.3.1. The synchronisation rules of processes

As we have shown above, preconditions that arise in rules are in one of the three following forms:

(1) already defined transitions,
(2) matching events,
(3) constraints on values caused by instructions of type (i).

The latter kind of precondition cannot be handled finitely, and has to be 'reduced' in some way as we shall indicate now. Among the constraints on values created by

instructions of type (i), we shall distinguish those that arise from boolean relations (generated by {=, ***and***, ***or***, ***not***} and the constants *true*, *false*), and we shall call the rest *non-boolean relations.*

From now on we shall require that non-boolean relations are *functions*, generically denoted by $y=f(x_1, \ldots, x_n)$ in this section. Notice that, according to this definition, the instruction b := (x < y) (where x, y denote, say, reals) produces a boolean output, but is considered as a non-boolean function. The function $y=x$, where y, x are non-booleans, will be handled via the same procedure, although a value identifier could have been substituted as well. The only resolution method we shall use for non-boolean functions is term rewriting. Chains of such rewritings are encoded as usual via dependence graphs. Within this framework, solving systems of non-boolean equations is performed by checking whether the associated dependence graph is circuit-free. We now introduce the following rules that we term *synchronisation rules* since they summarize the properties of the original rules that are relevant to synchronization, logic, and dependencies.

In the sequel, the notation π will denote a (possibly empty) set of ports; such sets will specify the predecessors of a port in a given dependency relation, so they will be called *predecessors.* Synchronisation rules will be obtained from the original ones by the following maps ψ_e and ψ_c, mapping respectively previous events and preconditions into new ones which will define the new synchronisation rules.

The domain of ψ_e is the set of events, and this map is defined by

$$\psi_e : \mathrm{x}(x) \rightarrow \mathrm{x}(x, \pi). \tag{3.13}$$

The domain of ψ_c is the set of non-boolean preconditions, and this map is defined by: if

$$\psi_e : \mathrm{x}_i(x_i) \rightarrow \mathrm{x}_i(x_i, \pi_i) \quad \text{and} \quad \mathrm{y}(y) \rightarrow \mathrm{y}(y, \pi) \tag{3.14}$$

then

$$\psi_c : y=f(x_1, \ldots, x_n) \rightarrow \mathrm{y} \notin \bigcup_{1 \leq i \leq n} (\pi_i \cup \{\mathrm{x}_i\}); \quad \pi \supset \bigcup_{1 \leq i \leq n} (\pi_i \cup \{\mathrm{x}_i\}). \tag{3.15}$$

Hence ports carry pairs of the form (x, π) where x is a value as before, and π is a predecessor. The component x will be useful for preconditions arising from boolean relations, but will play no role for non-boolean functions. Conversely, the second component π will be used for non-boolean functions to encode dependencies, but will have no significance for boolean relations. Both components have to be kept since boolean signals can be involved in both non-boolean functions and boolean relations as shown in the example b:= (x<y). Substitution (3.15) expresses that non-boolean functions are encoded as their dependence graphs, and that such dependence graphs should be circuit free.

The synchronisation rule of the instruction (i)

According to (3.13), the synchronisation rules of boolean relations are

$$\frac{p(x_1, \ldots, x_n)}{\mathsf{p(x1, \ldots, xn)} \xrightarrow{\mathsf{x1}(x_1, \pi_1) \ldots \mathsf{xn}(x_n, \pi_n)} \mathsf{p(x1, \ldots, xn)}} \tag{3.16}$$

Here, the bounds of boolean values are kept, and predecessors are free. Such trivial modifications of the original rules will not be explicitly mentioned in the sequel, they will be understood when 'rule unchanged' will be written; also by convention, for $(\perp, \pi)$ to be carried by a port, we must have $\pi = \emptyset$, i.e. absent values have empty predecessors.

According to (3.15), the synchronisation rule of a non-boolean function is

$$\frac{\mathsf{y} \notin \bigcup_{1 \le i \le n} (\pi_i \cup \{\mathsf{x}_i\}); \quad \pi \supset \bigcup_{1 \le i \le n} (\pi_i \cup \{\mathsf{x}_i\})}{\mathsf{y:=f(x1, \ldots, xn)} \xrightarrow{\mathsf{x1}(x_1, \pi_1) \ldots \mathsf{xn}(x_n, \pi_n) \quad \mathsf{y}(y, \pi)} \mathsf{y:= f(x1, \ldots, xn)}} \tag{3.17}$$

The values y, x_i are not constrained. The precondition expresses that at least y is added to the dependence graph, and that, by doing so, no circuit should be created.

The synchronisation rule of the instruction (ii)

The rule of the boolean delay is unchanged. For the non-boolean case, we get

$$\mathsf{y:= x\$}\boldsymbol{init}\cdot \xrightarrow{\mathsf{x}(x, \pi) \quad \mathsf{y}(y, \pi')} \mathsf{y:= x\$}\boldsymbol{init}\cdot \tag{3.18}$$

The predecessor of y is unconstrained: shift registers do not create dependencies. On the other hand, the bound on the value carried by the ports via the memory is lost: nonboolean delays are just mapped into pure synchronisation instructions. The memory plays no role any more, so it has been cancelled.

The synchronisation rule of the instruction (iii)

The instruction ***when*** with boolean output is unchanged. For the non boolean case, we get

(i) $\mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x, \pi_x) \quad \mathsf{b}(\perp, \emptyset) \quad \mathsf{y}(\perp, \emptyset)} \mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b}$

(ii) $\mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(\perp, \emptyset) \quad \mathsf{b}(b, \pi_b) \quad \mathsf{y}(\perp, \emptyset)} \mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b}$

(iii) $\dfrac{\mathsf{y} \notin (\pi_x \cup \{\mathsf{x}\}); \quad \pi_y \supset (\pi_x \cup \{\mathsf{x}\})}{\mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x, \pi_x) \quad \mathsf{b}(\mathit{true}, \pi_b) \quad \mathsf{y}(y, \pi_y)} \mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b}}$

(iv) $\mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b} \xrightarrow{\mathsf{x}(x, \pi_x) \quad \mathsf{b}(\mathit{false}, \pi_b) \quad \mathsf{y}(\perp, \emptyset)} \mathsf{y:=x}\ \boldsymbol{when}\ \mathsf{b}$ (3.19)

Only the rule (iii) modifies the dependence graph: the relation $y = x$ has been replaced by the corresponding dependency, according to (3.15).

The synchronisation rule of the instruction (iv)

Again, there is no change for the boolean case; and for the nonboolean case, we have

$$\text{(i)}\quad \frac{y \notin (\pi_u \cup \{u\});\quad \pi_y \supset (\pi_u \cup \{u\})}{y:=u\ \textbf{\textit{default}}\ v \xrightarrow{u(u,\,\pi_u)\ \ v(\perp,\,\emptyset)\ \ y(y,\,\pi_y)} y:=u\ \textbf{\textit{default}}\ v}$$

$$\text{(ii)}\quad \frac{y \notin (\pi_u \cup \{u\});\quad \pi_y \supset (\pi_u \cup \{u\})}{y:=u\ \textbf{\textit{default}}\ v \xrightarrow{u(u,\,\pi_u)\ \ v(v,\,\pi_v)\ \ y(y,\,\pi_y)} y:=u\ \textbf{\textit{default}}\ v}$$

$$\text{(iii)}\quad \frac{y \notin (\pi_v \cup \{v\});\quad \pi_y \supset (\pi_v \cup \{v\})}{y:=u\ \textbf{\textit{default}}\ v \xrightarrow{u(\perp,\,\emptyset)\ \ v(v,\,\pi_v)\ \ y(y,\,\pi_y)} y:=u\ \textbf{\textit{default}}\ v} \tag{3.20}$$

where the relations $y=u$, $y=v$ are replaced by their respective dependencies.

The synchronisation rule of the instruction (v)

Finally, the image of the instruction (v) has to be defined for the resulting map on processes to be a semi-group homomorphism; hence, denoting this map by Ψ, we inductively define

$$\Psi(\Pi)\,|\,\Psi(\Pi') :: \frac{P \xrightarrow{\psi(\alpha)} P';\quad Q \xrightarrow{\psi(\beta)} Q';\quad \psi(\alpha) \cap \psi(\beta)}{P\,|\,Q \xrightarrow{\psi(\alpha)\,\cup\,\psi(\beta)} P'\,|\,Q'} \tag{3.21}$$

where the events $\psi(\alpha)$ have been defined for the image by Ψ of the generators, and are defined by induction via the formula (3.21). This definition guarantees that Ψ is a semi-group homomorphism, i.e.

$$\Psi(\Pi\,|\,\Pi') = \Psi(\Pi)\,|\,\Psi(\Pi'). \tag{3.22}$$

The range of Ψ will be called the semi-group of *synchro-processes*, since they summarize the logic, synchronization, and dependency structure of the process.

3.3.2. Algebraic representation of synchro-processes

The purpose of this section is to introduce the coding we shall use for the compilation. The idea behind this coding is the following. There are two basic tools for transforming and analysing programs before execution on a given architecture. The first tool is the *directed graph* showing data dependencies; this may be the only one for very regular algorithms such as encountered in systolic architectures where retiming is of interest. The second tool is the automaton describing the control of the program. In the previous section, we have prepared the introduction of a single

framework to handle both tools simultaneously. In this section, we go further by presenting an efficient co-'ng for this purpose: the control of the program will be encoded into a dynamical system (shown to be equivalent to an automaton), and we shall show how to handle *dynamically evolving* data dependence graphs.

3.3.2.1. *Dynamical systems over finite fields*

Synchro-processes are defined via rules involving $\perp$, booleans, and dependencies. We shall first provide an algebra with a convenient calculus where the pairs $\{\perp, \text{booleans}\}$ can be represented. All we need to encode are the following status: *absent, present, true, false.* These are encoded onto the finite field $\mathbb{F}_3 = \mathbb{Z}/3\mathbb{Z}$ of integers modulo 3 as follows

true: $+1$
false: -1
absent: 0
present: ± 1

where ± 1 denotes a non-determinate choice of $+1$ or -1; i.e. we handle labels and boolean of non-determinate value in the same way. Let us now define how the control of SIGNAL programs is encoded, namely using the algebra of *dynamical systems* over $\mathbb{F}_3^n$.

A dynamical system over $\mathbb{F}_3^n$ is specified by

(1) a submanifold of the product space $\mathbb{F}_3^n \times \mathbb{F}_3^n$;
(2) an initial condition in $\mathbb{F}_3^n$.

Indeed, denoting the generic point of $\mathbb{F}_3^n$ by ξ, such a submanifold is specified via a system of polynomial equations

$$P_1(\xi', \xi) = 0, \ldots, P_k(\xi', \xi) = 0 \tag{3.23}$$

where $\xi = (x_1, \ldots, x_n)$, and the x_i's are variables in $\mathbb{F}_3$. Then, *the dynamical system* Δ *is the subset of the trajectories on* $\mathbb{F}_3^n$ *satisfying*

$$P_1(\xi_t, \xi_{t-1}) = 0, \ldots, P_k(\xi_t, \xi_{t-1}) = 0 \tag{3.24}$$

where ξ_0 equals the given initial condition.

3.3.2.2. *Dynamical graphs*

A *dynamical graph* is a triple $\{\Delta, \Gamma, \gamma\}$ where

- Δ is a dynamical system over $\mathbb{F}_3^n$,
- Γ is a directed graph,
- γ is a function mapping $\mathbb{F}_3^n$ into the set of the subgraphs of Γ.

Notice that the map γ is equally well defined by specifying for each branch $x \longrightarrow y \in \Gamma$ the subsets of $\mathbb{F}_3^n$ of the points ξ such that $\gamma(\xi)$ contains the considered branch.

This will be denoted by

$$V: x \longrightarrow y \quad \text{or} \quad x \stackrel{V}{\longrightarrow} y \tag{3.25}$$

where V is the considered subset of $\mathbb{F}_3^n$.

Hence dynamical graphs are skew products of dynamical systems and graphs. The dynamical system is intended to encode the underlying control involved in a program, while the directed graphs will encode the way dependencies evolve during an execution: the dependencies at a given event will only depend on the set of signals that are present in this event.

3.3.2.3. The algebraic representation

Using these notions, the algebraic coding of the synchronisation rules is derived as follows. First, add the distinguished value $\perp$ to the domain of the value identifiers $x, y, \ldots$, and explicitly mention the additional constraints $x = \perp, y \neq \perp, \ldots$ whenever needed in the preconditions. Second, introduce the following map χ; the domain of χ is the set of preconditions, and its codomain is the set of dynamical graphs. This map assigns, to each original constraint involving presence/absence, a dynamical system on $\mathbb{F}_3^n$ involving the same value identifiers. More precisely, this map is defined as follows.

$$\begin{cases} \chi : x = \perp \rightarrow x^2 = 0, \\ \chi : x \neq \perp \rightarrow x^2 = 1 \end{cases} \tag{3.26}$$

where the map encodes the presence/absence of the values carried by ports within the actions; only squares appear since the value of booleans plays no role here. Other formulas concern boolean values:

$$\begin{aligned} &\chi : b = \textit{true} \rightarrow b = 1, \\ &\chi : b = \textit{false} \rightarrow b = -1, \\ &\chi : b = \textbf{not } a \rightarrow b = -a, \\ &c = a \textbf{ and } b \rightarrow c = 1 - (ab + a + b). \end{aligned} \tag{3.27}$$

Only the last part needs to be verified by checking all the combinations of ± 1 values for a and b. The following formula indicates how dependency constraints are mapped:

$$\chi : x \in \pi_u \rightarrow x \longrightarrow y. \tag{3.28}$$

In other words, the coding uses a graphic notation to describe predecessors. Checking that $y \notin \pi_x \cup \{x\}$ is then equivalent to verifying whether adding the branch $x \longrightarrow y$

to the dependence graph does or does not create circuits. This will be always assumed in the sequel.

Finally, as before, the map χ induces a semi-group homomorphism denoted by X on synchro-processes by proceeding as in (3.21), so that we get

$$\mathrm{X}(\Pi \mid \Pi') = \mathrm{X}(\Pi) \mid \mathrm{X}(\Pi'). \tag{3.29}$$

Let us illustrate how the coding works on the synchronisation rules of the instruction ***when*** in the non boolean case. Starting from the rules (3.19), the new preconditions are

$$\begin{aligned}
&\text{(i)} && x^2=1, \quad b^2=0, \quad y^2=0,\\
&\text{(ii)} && b^2=1, \quad x^2=0, \quad y^2=0,\\
&\text{(iii)} && b=1, \quad x^2=1, \quad y^2=1, \quad \mathrm{x} \longrightarrow \mathrm{y},\\
&\text{(iv)} && b=-1, \quad x^2=1, \quad y^2=0.
\end{aligned} \tag{3.30}$$

On the other hand, it should be clear that *everything relevant is encoded in these preconditions*, i.e. events as well as states of the conditional rewriting rules provide no further information than just the syntax of the instruction. Hence we shall keep (3.30) as the only relevant part of the process X[Ψ(y:= x ***when*** b)]. On the other hand, since any one of the rules (i)-(iv) can be applied, these preconditions can be summarized as the double coding

$$\mathrm{X}[\Psi(\mathrm{y} := \mathrm{x}\ \textbf{\textit{when}}\ \mathrm{b})] :: \left(\frac{y^2 = x^2(-b-b^2)}{y^2 : \mathrm{x} \longrightarrow \mathrm{y}} \right) \tag{3.31}$$

In the second field of this coding 'y^2:' is a shorthand to indicate that the dependency holds exactly when $y^2 = 1$. A systematic application of this method yields the algebraic coding synchro-processes that we present in the next subsection.

3.3.3. *Encoding SIGNAL programs*

The following notation will be used to present this coding:

$$\Sigma(\text{program}) :: \left(\frac{\textit{clock calculus}}{\textit{conditional dependence graph}} \right) \tag{3.32}$$

where

- program denotes the program to be encoded, and $\Sigma = \mathrm{X} \circ \Psi$ is the encoding map, i.e. the composition of the maps Ψ and X;

- *clock calculus* denotes the set of algebraic equations encoding the constraints on synchronisation or logic as we discussed above; these equations define dynamical systems on $\mathbb{F}_3^n$;
- *conditional dependence graph* denotes the set of possibly occurring dependencies together with the clocks where these dependencies are in force.

3.3.3.1. Instruction (i): *relation or function*

Boolean relation

The coding of all boolean relations is easily derived from the coding of the following instructions and the coding of the communication we shall see below:

$$\Sigma(\mathsf{a} := \mathit{true}) :: \left(\frac{a^2 - a = 0}{\emptyset} \right)$$

$$\Sigma(\mathsf{b} := \textbf{\textit{not}}\ \mathsf{a}) :: \left(\frac{b = -a}{\emptyset} \right)$$

$$\Sigma(\mathsf{c} := \mathsf{a}\ \textbf{\textit{and}}\ \mathsf{b}) :: \left(\frac{\begin{matrix} a^2 = b^2 \\ c = a^2 - (ab + a + b) \end{matrix}}{\emptyset} \right) \tag{3.33}$$

The algebraic equation of the first formula possesses $a = 1$, $a = 0$ as the only solutions, which means that a is either absent or true. The second equation is obvious. To derive the last one, note that its first component encodes the fact that both signals a and b must have the same clock (they are either both present or absent, which is encoded as $a^2 = 1$ or $a^2 = 0$); then it is straightforward to verify that the last equation maps the pairs $(0, 0)$, $(1, 1)$, $(-1, 1)$, $(1, -1)$, $(-1, -1)$ onto 0, 1, -1, -1, -1 respectively. Since only booleans are involved, no coding of dependencies is required hence the symbol $\emptyset$ in the second field.

Non-boolean function

$$\Sigma(\mathsf{y} := \mathsf{f}(\mathsf{x}_1, \ldots, \mathsf{x}_n)) :: \left(\frac{y^2 = x_1^2 = \cdots = x_n^2}{y^2 : \mathsf{x}_1 \longrightarrow \mathsf{y} \cdots \mathsf{x}_n \longrightarrow \mathsf{y}} \right) \tag{3.34}$$

The first field encodes the constraints on clocks (equality), while the second one encodes the data dependencies. The second field means "the listed dependencies hold when $y^2 = 1$". Notice that $\mathsf{a} := (\mathsf{u} < \mathsf{v})$ produces a boolean, but it is a non-boolean function.

3.3.3.2. *Instruction* (ii): *the register*

Boolean register

This is the key case where dynamical systems in $\mathbb{F}_3$ come out.

$$\Sigma(\mathsf{b} := \mathsf{a\$}\ \textbf{\textit{init}}\ u) :: \left(\frac{\begin{array}{c} \xi' = (1-a^2)\xi + a; \quad \textit{initial cond} = u \\ b = a^2\xi \end{array}}{\emptyset} \right) \tag{3.35}$$

where ξ' is the current state of the dynamical system, ξ its previous state, and u its initial condition (± 1 valued). The corresponding explicit form of this dynamical system is

$$\xi_t = (1-a_t^2)\xi_{t-1} + a_t; \quad \xi_0 = u,$$

$$b_t = a_t^2 \xi_{t-1}$$

where t is any time index fast enough to capture every presence of signal. Notice that the state takes $+1$ or -1 as only values, i.e. states are persistent. The state is modified when a new input is received, and at the same instant the old state is delivered at the output. Again no dependence graph is necessary.

Non-boolean register

$$\Sigma(\mathsf{y} := \mathsf{x\$}\ \textbf{\textit{init}}\ u) :: \left(\frac{y^2 = x^2}{\emptyset} \right) \tag{3.36}$$

The first field expresses that clocks must be identical; the second field is empty even if we consider non-boolean types, since the current value of y does not depend on the current value of x, but on the content of the memory (which has been lost in the coding via Σ).

3.3.3.3. *Instruction* (iii): *the* ***when***

***when** with boolean output*

$$\Sigma(\mathsf{c} := \mathsf{b}\ \textbf{\textit{when}}\ \mathsf{a}) :: \left(\frac{c = b(-a-a^2)}{\emptyset} \right) \tag{3.37}$$

***when** with non-boolean output*

$$\Sigma(\mathsf{y} := \mathsf{x}\ \textbf{\textit{when}}\ \mathsf{a}) :: \left(\frac{y^2 = x^2(-a-a^2)}{y^2 : x \longrightarrow y} \right) \tag{3.38}$$

The second field expresses that x influences y when y is produced.

3.3.3.4. Instruction (iv): *the merge*

default *with boolean output*

$$\Sigma(\mathsf{c} := \mathsf{a}\ \textbf{\textit{default}}\ \mathsf{b}) :: \left(\frac{c = a + b(1 - a^2)}{\emptyset} \right) \tag{3.39}$$

default *with non-boolean output*

$$\Sigma(\mathsf{y} := \mathsf{u}\ \textbf{\textit{default}}\ \mathsf{v}) :: \left(\frac{y^2 = u^2 + v^2(1 - u^2)}{\begin{array}{c} u^2 : \mathsf{u} \longrightarrow \mathsf{y} \\ v^2(1 - u^2) : \mathsf{v} \longrightarrow \mathsf{y} \end{array}} \right) \tag{3.40}$$

The second field expresses the fact that u influences y when it is present, while v influences y when it is present and u is absent.

3.3.3.5. Instruction (v): *the communication*

$$\Sigma(\mathsf{P} \mid \mathsf{Q}) = \Sigma(\mathsf{P}) \mid \Sigma(\mathsf{Q}) \tag{3.41}$$

where the symbol | on the right-hand side simply means that the conjunction of the clock calculi (resp. conditional dependence graph) of P and Q is taken to produce the two corresponding fields of Σ(P|Q).

For general SIGNAL programs, the coding above has the following generic form

$$\Sigma(\mathsf{P}(u)) :: \left(\frac{\begin{array}{c} \xi' = Q(\xi, X);\ \textit{initial cond} = u \\ R(X, \xi) = 0 \end{array}}{\textit{for } i \in I, \quad h_i^2 : \mathsf{x}_i \longrightarrow \mathsf{y}_i} \right) \tag{3.42}$$

where

- ξ' (resp. ξ) is the vector of new (resp. old) boolean memories; the first equation of the clock calculus summarizes the evolution of the boolean memories; the vector u summarizes the initial values of the boolean memories;
- X is the vector of the other variables of the clock calculus; the second equation of the clock calculus summarizes the static constraints on clocks;
- the conditional dependence graph is a list of arcs labelled by clocks as written in the second field.

Conversely, all synchronisation rules of a given program can be recovered from the coding (3.42) as follows: partition the set I as $I = I_0 \cup I_1$ and consider the rule

$$\frac{v = Q(u, X); \quad R(X, u) = 0; \quad i \in I_0 : h_i^2 = 0; \quad i \in I_1 : h_i^2 = 1 \quad \text{and} \quad \mathsf{x}_i \longrightarrow \mathsf{y}_i}{\mathsf{P}(u) \xrightarrow{\mathsf{X}(X)} \mathsf{P}(v)} \tag{3.43}$$

Reject this rule if the dependencies for $i \in I_1$ create circuits; the remaining rules are the synchronisation rules of $\mathsf{P}(u)$.

3.3.4. Examples

To allow drawing of graphs, conditional dependence graphs will be depicted using the following notation:

$$\mathsf{x} \xrightarrow{h} \mathsf{y} \quad \text{instead of} \quad h\colon \mathsf{x} \longrightarrow \mathsf{y}$$

3.3.4.1. The macro *disjoint*

Recall that, for events h, k,

disjoint h, k

stands for

```
| l := when ((not(h when k)) default h)
|
| synchro l, h.
```

Using the property $h = h^2$ if h is an event, the clock calculus is easily derived:

$$h = l = -[-hk + (1 + hk)h] - [-hk + (1 - hk)h]^2$$

which implies

$$h = -h[1 + k] - h^2[1 + k]^2 = h - hk$$

and yields the desired result, namely $hk = 0$.

3.3.4.2. The macro *cell*.

This instruction has been used as a macro in the 'shared track' example. Recall the corresponding program:

y := x ***cell*** b ***init*** y_0

stands for

```
| zy := y$ init y0
|
| y := x default zy
|
| synchro y, (x default when (b))
```

We shall make a distinction between two cases: x boolean, and x non-boolean.

Encoding the boolean ***cell*** *into its clock calculus*

$$\xi' = (1 - y^2)\xi + y, \ \textbf{\textit{init}} = y_0,$$

$$zy = y^2\xi,$$

$$y = x + zy(1 - x^2),$$

$$y^2 = x^2 + (-b - b^2)(1 - x^2). \qquad (3.44)$$

Eliminating zy yields

$$\xi' = (1-x^2)\xi + x, \quad \textbf{\textit{init}} = y_0,$$
$$y = x + (-b-b^2)(1-x^2)\xi$$

which reflects exactly the meaning of the instruction ***cell*:** the memory is refreshed when x is received, and y delivers the current or last value of x when x is received or b is received and true.

Encoding the non-boolean cell into its clock calculus and conditional dependence graph

Clock calculus:

$$zy^2 = y^2 = x^2 + (1-x^2)zy^2,$$
$$y^2 = x^2 + (1-x^2)(-b-b^2)$$

which yields

$$zy^2 = y^2 = x^2 + (1-x^2)(-b-b^2).$$

Conditional dependence graph:

$$\mathsf{zy} \xrightarrow{y^2(1-x^2)} \mathsf{y}, \qquad \mathsf{x} \xrightarrow{x^2} \mathsf{y}$$

where the dynamics has been lost; the clock calculus expresses only how the clocks of the signals are related.

3.3.4.3. *The program* GUARD_COUNT

Recall the program:

```
GUARD_COUNT (integer level)  {? even reset  ! bool empty}

      =

  | n := (level when r) default (zn - 1)
  | zn := n$ init 0
  | synchro (when (zn = 0)), r
  | e := when (n = 0) default (not r)

r: reset, e: empty
end
```

Clock calculus and dependence graph will be written with the short signal names.

Clock calculus:

$$n^2 = r + zn^2(1-r) = zn^2,$$
$$e = (-[n=0] - [n=0]^2) - r(1 + [n=0] + [n=0]^2),$$
$$-[zn=0] - [zn=0]^2 = r. \tag{3.45}$$

Introduce the notations l = level, $\alpha = [n = 0]$, $\beta = [zn = 0]$. The clock calculus can be rewritten as

$$zn^2 = n^2 = \alpha^2 = \beta^2,$$
$$e = (-\alpha - \alpha^2) + (\beta + \beta^2)(1 + \alpha + \alpha^2),$$
$$r = -\beta - \beta^2. \tag{3.46}$$

Conditional dependence graph:

$$\mathsf{l} \xrightarrow{r} \mathsf{n}, \qquad \mathsf{zn} \xrightarrow{(1-r)n^2} \mathsf{n}, \qquad \mathsf{n} \xrightarrow{n^2} \alpha. \tag{3.47}$$

Discussion (observability). Consider the first two equations of (3.45), which correspond to the first two instructions of the program. They can be rewritten in the equivalent *explicit* form

$$n^2 = zn^2 = r^2 + \Phi^2(1 - r^2)$$

where Φ is a *free* variable of $\mathbb{F}_3$; this additional variable, which we shall call a *phantom*, reflects the fact that the two first instructions of the program are *not observable* by the input reset alone. The clock of the outputs zn and n of this subprogram is not entirely constrained by the clock of the input reset, which is a cause of the non-determinism of this subprogram (see [12] for a discussion of how non-determinism can result from the interconnection of deterministic processes).

However, considering the whole clock calculus yields a different result. This clock calculus is an algebraic variety which is entirely parametrized by the free parameters $\{\alpha, \beta\alpha^2\}$. On the other hand, since the conditional dependence graph has reset as the only source node (except from the delay output zn), it is expected that the whole program is observable by the triple $\{$reset, $\alpha, \beta\alpha^2\}$. A systematic study of this kind of observability notion will be presented later.

3.3.4.4. An example of deadlock

Consider the following example.

```
| x := u when (u<v)
| y := x+v
```

The meaning of this program is "add u to (v *when* u < v)"; this program should be rejected, since the clocks are inconsistent. Writing β for short instead of (u < v), the conditional dependence graph of this program is

$$\mathsf{u} \xrightarrow{u^2} \beta, \quad \mathsf{v} \xrightarrow{v^2} \beta, \quad \mathsf{u} \xrightarrow{x^2} \mathsf{x}, \quad \mathsf{u} \xrightarrow{x^2} \mathsf{y}, \quad \mathsf{v} \xrightarrow{v^2} \mathsf{y}.$$

The clock calculus is

(i) $u^2 = v^2 = \beta^2$,

(ii) $x^2 = u^2(-\beta - \beta^2)$,

(iii) $y^2 = x^2 = v^2$.

which obviously enforces $\beta = 1$. However β is not free, but it is the result of the evaluation of the inputs u and v; however neither our clock calculus nor our conditional dependence graph can reason about non-boolean values, therefore the actual value (true or false) of β cannot be predicted within our calculus, and this value should not be constrained. This is taken into account by adding to (ii) the constraint obtained by the symmetry $\beta \to -\beta$, thus resulting in the new constraint

(ii′) $0 = \beta u^2$

instead of (ii), thus yielding finally

$$any^2 = 0$$

i.e. the whole program starvates from the beginning. This illustrates informally how deadlocks can be detected by taking into account clocks and data dependencies. Again this will be formalized later.

3.4. Conclusion of the chapter

We have presented a behavioural semantics of the SIGNAL language using processes encoded via transition systems. We have shown how general processes can be mapped into the subalgebra of synchro-processes systems. Finally we have exhibited an algebraic coding of these processes via pairs {clock calculus, conditional dependence graph}. What remains to be done is to provide an algorithm to *solve* such processes, i.e. to compile them into executable machines, and this will be the goal of the next chapter.

Chapter 4. Execution semantics

In the preceding section, we showed how the pair {*clock calculus, conditional dependence graph*} (referred to as *synchro-process* in the sequel) can be used to encode and analyze a SIGNAL program. The purpose of this section is to investigate the following questions about synchro-processes:

- what is observability and how to check this property;
- what is deadlock and how to detect and isolate it;
- construct a machine which can execute synchro-processes.

These questions will be addressed by constructing the execution semantics of SIGNAL programs. A particular difficulty arises from the relational nature of SIGNAL, namely: *what are inputs, what are outputs* in a given process? This is a key issue, since deadlock and observability (or determinism) are usually defined with respect to a prespecified set of input stimuli. Unfortunately, in our case, the body of SIGNAL programs specifies relations between signals, but does not indicate completely what the input signals are; for instance, in case of a program involving

only synchronization and logic, no (non-boolean) function takes place, the conditional dependence graph is empty, so that any subset of signals could be selected as the desired inputs provided these are free and determine the other signals. Unfortunately, automatic selection of the input signals by the compiler is hard to perform since several choices are possible in general, hence the programmer is expected to play an active role in such a selection. Consequently, we chose to consider that the declarations of input/output signals given in the interface of a program provide *specifications of the desired inputs*, and investigate the above questions given *these* input signals.

4.1. Solving clock calculi: a toolbox

As we have discussed before, producting events in SIGNAL programs requires solving implicit systems of equations. These implicit systems involve only signals at a given instant. For this reason, *static clock calculi* that we shall introduce now play a crucial role in the execution semantics of SIGNAL. Static clock calculi are clock calculi which do not involve boolean states (or memories), which means that the following rule has been used to encode the boolean delays instead of the rule (3.25):

$$\textbf{\textit{synch}}(\mathsf{b} := \mathsf{a}\$\textbf{\textit{init}}\ u) :: \left(\frac{b = a^2 \varsigma}{\emptyset}\right) \tag{4.1}$$

hence boolean registers are encoded as the single output equation. Obviously, the algebraic variety defined by the static clock calculus is just the projection of the clock calculus (more precisely the projection of the orbits of the dynamical system defined by the clock calculus) along the time axis. Static clock calculi are of the generic form

$$P_1(x_1, \ldots, x_n) = 0, \ldots, P_K(x_1, \ldots, x_n) = 0$$

where the x_i's are the variables of the static clock calculus, and the P_k's are polynomials of $\mathbb{F}_3[x_1, \ldots, x_n]$ of degree at most 2 with respect to each variable.

Definition 2. A static clock calculus is said to be *pre-solved* if it is composed of equations of one of the following forms only:

$$\begin{aligned} &\text{(i)} \quad y = Ax^2 + Bx + C, \\ &\text{(ii)} \quad y^2 = Ax^2 + Bx + C \end{aligned} \tag{4.2}$$

where A, B, C are polynomials which are free from the variable x. Moreover every variable must be

- either absent from the left-hand side of all equations,
- or appearing once at the left-hand side in only one of the two forms (i) and (ii) above.

The form (i) means that the value of y is bound, while the form (ii) means that only the clock of the variable y is bound, on the other hand its actual value (+1 or −1) might be free (this is the case for boolean variables which are produced by non-boolean functions such as b:= (x<y) or for vertices of the conditional dependence graph). Notice that cycles of mutually defined variables can exist in pre-solved clock calculi: for instance $x^2 = u^2y^2$, $y^2 = x^2 + v^2(1-x^2)$ is pre-solved, but $u^2x^2 + v^2y^2 = 0$ is not.

As in computational algebraic geometry [14, 15], a pre-solved form is obtained via elimination techniques. The basic lemma for elimination is

Lemma 1.

$$ax^2 + c = 0 \quad \Leftrightarrow \quad \begin{cases} c(a+c) = 0, \\ x^2 = \Phi^2(1-a^2) + c^2. \end{cases} \tag{4.3}$$

$$ax^2 + bx + c = 0$$

$$\Leftrightarrow \begin{cases} c[(a+c)^2 - b^2] = 0, \\ x = \Phi\left[\prod_{y=a,b,c} (1-y^2)\right] - (1-a^2)bc + a[b + (1+\Phi^2)(b^2 - ac)]. \end{cases} \tag{4.4}$$

In both equations, Φ denotes a phantom, i.e. an additional free variable.

Comment. The first rule is convenient to solve for clocks, while the second one has to be used for boolean relations. Both rules have the form

$$\text{equation} \Leftrightarrow \begin{cases} \text{adding constraints on the remaining variables when } x \text{ is eliminated} \\ \text{defining } x \text{ or } x^2 \text{ in terms of the other variables.} \end{cases}$$

Proof. The proof relies on the following formulas that are useful and immediate

$$\begin{aligned} \{p = 0 \text{ and } q = 0\} \quad &\Leftrightarrow \quad \{p^2 + q^2 = 0\}, \\ \{p = 0 \Rightarrow q = 0\} \quad &\Leftrightarrow \quad \{q(1-p^2) = 0\}. \end{aligned} \tag{4.5}$$

We prove only (4.4), since (4.3) is easier and follows the same lines. For the equation (4.4) to have a solution, the following constraints must be satisfied by the triple $\{a, b, c\}$:

$$\begin{aligned} \{a = b = 0\} \quad &\Rightarrow \quad \{c = 0\}, \\ \{a \neq 0\} \quad &\Rightarrow \quad \{\Delta^2 = b^2 - ac \neq -1\}. \end{aligned}$$

Notice that Δ is nothing but the discriminant of the equation. Combining these constraints using (4.5) yields the constraint in (4.4). Then, the definition of x follows easily as in college algebra. Notice that in this second equation, $1 + \Phi^2$ is a writing of ±. The same phantom can be used in the two terms of the definition of x since at most one of these two terms is different from 0 depending upon the value of a^2.

Using Lemma 1 allows us to perform elimination when an ordering of the equations and variables has been chosen; a detailed algorithm will be presented later. Synchro-processes with pre-solved clock calculi will be called *pre-solved synchro-processes.* In the rest of this chapter, we shall write *clock calculus* for short to refer to the *static* clock calculus.

4.2. The graph of a pre-solved synchro-process

The main difficulty in solving synchro-processes is due to the presence of two different kinds of ordering, namely

- the ordering of the variables and of the equations required for the elimination to be performed in the clock calculus,
- the ordering resulting from the conditional dependence graph.

Both orderings interact. In fact, elimination in the clock calculus must be performed by taking into account the conditional dependence graph, and moreover *it is not possible to know a clock, which depends on the value of a boolean signal resulting from a non-boolean function, prior to evaluating this function.* On the other hand, evaluating such functions require the knowledge of their clock. The purpose of this paragraph is to introduce the main tool to handle this interaction. This tool plays a role similar to the 'potentials' introduced independently by Gonthier [20, 10].

Definition 3. The *graph* G of a synchro-process is the labelled directed graph obtained by considering branches of the form

$$x \xrightarrow{h} y \quad \text{or} \quad \mathsf{x} \xrightarrow{h} \mathsf{y} \tag{4.6}$$

where h is a clock encoded by its polynomial expression in $\mathbb{F}_3$, and x and y are variables of the clock calculus (resp. x and y are vertices of the conditional dependence graph).

The intuitive meaning is: "*x may influence y when $h = 1$*". We make use of the following conventions to simplify graphs: if a label is known to be zero, the corresponding branch can be removed. On the other hand, labels known to be equal to 1 are not written.

The graph is built according to the rules below. Here, A denotes the set of the vertices of the conditional dependence graph, x^2 will denote the clock of x, if b is a boolean output of a non-boolean function its value in the clock calculus is b, CDG denotes the conditional dependence graph introduced before, G denotes the graph of the synchro-process we shall build now, and CLOCK denotes the static clock calculus of the considered process.

Rule GRAPH_1

$$\frac{a \in A}{a^2 \xrightarrow{a^2} a} \tag{4.7}$$

To have access to the value of a non-boolean signal, we need to know whether or not it is present at the considered instant, i.e. we need to know its clock.

Rule GRAPH_2

$$\frac{h : x \longrightarrow y}{x \xrightarrow{h} y, \quad h \xrightarrow{h} y} \tag{4.8}$$

The first part of the rule is the exact translation of the contribution of CDG; the second part expresses that to evaluate y, we must know when the considered dependency holds, i.e. we must know the actual value of h.

Rule GRAPH_3

$$\frac{y = Ax^2 + Bx + C}{x \xrightarrow{x^2B^2} y, \quad x^2 \xrightarrow{x^2A^2(1-B^2)} y} \tag{4.9}$$

x influences y when $B \neq 0$, while only x^2 influences y when $B = 0$ and $A \neq 0$.

Rule GRAPH_4

This rule will be used for synchro-processes which are not pre-solved; for these processes the rule GRAPH_3 does not cover all the cases. For any clock equation to which rule GRAPH_3 does not apply (it is not of the form y *or* $y^2 = \cdots$, or more than a single expression is available on the right-hand side of y *or* $y^2 = \cdots$)

$$\frac{Ax^2 + Bx + C = 0, \quad A, B, C \text{ free from } x}{x \xrightarrow{x^2B^2} any, \quad x^2 \xrightarrow{x^2A^2(1-B^2)} any} \tag{4.10}$$

where *any* refers to any variable of the considered equation except x.

Notice that, in rules GRAPH_3 and GRAPH_4, branches denote *potential* influence between vertices.

Example. The instruction y := x ***when*** b in the boolean case.

The rule GRAPH_3 gives

$$x \xrightarrow{x^2(-b-b^2)} y, \qquad b \xrightarrow{x^2b^2} y.$$

There is a clear intuitive meaning for this graph: x influences y when b permits to use x to produce y, and similarly for the other dependency. On the other hand, the rule GRAPH_4 gives the following additional branches

$$x \xrightarrow{x^2(-b-b^2)} b, \quad b \xrightarrow{b^2x^2} x, \quad y \xrightarrow{y^2} x, \quad y \xrightarrow{y^2} b.$$

This graph is much larger and more difficult to interpret, but it should be kept in mind that this is the rule to be used if it is not known in advance which signal will be considered as the input.

Warning. In the following, we shall omit for short the clock of the source node of any branch.

Example. The program GUARD_COUNT

Using the synchro-process (3.46, 3.47) encoding this program, and setting

$$\begin{aligned} &h = n^2, \\ &\alpha_* = (1+\alpha+\alpha^2)^2 = 1+\alpha+\alpha^2, \\ &\beta_* = (1-\beta-\beta^2)^2 = 1 \end{aligned} \tag{4.11}$$

we get, by rule GRAPH_1

$$h \longrightarrow \alpha, \quad h \longrightarrow \beta, \quad h \longrightarrow \mathsf{n}, \quad h \longrightarrow \mathsf{zn};$$

by rule GRAPH_2

$$\mathsf{I} \xrightarrow{r} \mathsf{n}, \quad \mathsf{zn} \xrightarrow{(1-r)h} \mathsf{n}, \quad \mathsf{n} \xrightarrow{h} \alpha;$$

by rule GRAPH_3, and taking into account (4.11)

$$\alpha \longrightarrow e, \quad \beta \xrightarrow{\alpha_*} e, \quad \beta \longrightarrow r.$$

Important Remark. It should be clear from the rules GRAPH_3 and GRAPH_4 that the graph of a synchro-process is by no means invariant under transformations of the clock calculus which preserve the underlying algebraic variety: two isomorphic synchro-processes can have different associated graphs. This property will be exploited in the sequel.

We are now ready to present the algorithm EXEC for the execution of synchro-processes. This algorithm is used at run-time.

4.3. Synchro-process execution: the algorithm EXEC

The purpose of this algorithm is *to decompose any transition rule* (3.43) *into a chain of elementary transitions that can be explicitly performed.* Such a method (also used in [20]) guarantees that the execution semantics performs only runs which meet the specifications of the behavioural semantics.

4.3.1. Introducing the notations for EXEC

4.3.1.1. States

STATES of such transition systems will be partitions on the pair {G, CLOCK} = {graph, static clock calculus}. The static clock calculus is partitioned according to

$$\text{CLOCK} = \textbf{\textit{val}}(\text{CLOCK}) \cup \textbf{\textit{?}}(\text{CLOCK}),$$

i.e. variables which value is known, and the others. Similarly the nodes of G are partitioned according to

$$\text{nodeG} = \textbf{\textit{val}}(\text{nodeG}) \cup \textbf{\textit{?}}(\text{nodeG})$$

where ***val***(...) refers to nodes which have been evaluated (they may be absent). The notation

$$\text{G}: y \in \textbf{\textit{val}} \tag{4.12}$$

means that the partition on G is modified by transferring the node y into ***val***(nodeG); this notation will be modified in obvious ways to cover all possible modifications of the states of EXEC. The initial condition of algorithm EXEC is exactly the pair {G, CLOCK} encoding the considered program: initial states of EXEC correspond to programs in the original transition rules of SIGNAL.

4.3.1.2. Actions

ACTIONS of these transition systems will be elements of the following list

(i) CLOCK: list of "$x \leftarrow \textbf{\textit{val}}(x)$"

(ii) G: list of "$x \leftarrow \textbf{\textit{val}}(x)$"

The first rule means that "the algebraic variable x is substituted by its value in CLOCK", then all variables of CLOCK which depend upon x are recursively evaluated. Finally all the so evaluated clock variables are substituted by their values on the branches and nodes of G. If this actual value is 0, the considered branch is said to be *broken.*

The second rule means that "the vertex x is substituted by its value in G". Notice that this action concerns the evaluation of non-boolean functions only. Then if x turns out to be boolean (remember $\text{x} = (\text{u} < \text{v})!$)), its value is substituted for the corresponding variable in CLOCK and in the clock nodes of G. In the forthcoming rules, we shall omit, for brevity, to mention "CLOCK:" or "G:" in the actions (i) and (ii): this is understood according to the type of evaluation being performed.

4.3.1.3. Preconditions

PRECONDITIONS of these transition systems are of the form:

(i) $x \in \textbf{\textit{val}}(\text{nodeG})$,

(ii) $h \in \textbf{\textit{val}}(\text{CLOCK}), \quad h = 0, 1, -1,$

(iii) list of $x \xrightarrow{h} y$.

(i, ii) have already been defined; (iii) refers to the list of *all* predecessors of y in G. When no confusion can occur or when this information is unnecessary, the mention (nodeG) or (CLOCK) will be omitted in the preconditions (i) and (ii).

4.3.2. The rules of EXEC

It is assumed that *the environment provides any information* (clocks and/or values) *whenever needed*, i.e. it matches the synchronisation constraints required by the program. Notice that this is not a trivial assumption, this point is further discussed in Section 5.1. The following rules describe the algorithm. The notation (4.12) is used to indicate the modifications of the states.

Rule EXEC_0

$$\frac{x \in \text{source(G)}}{\{\text{G, CLOCK}\} \xrightarrow{x \leftarrow \textit{val}(x)} \{\text{G, CLOCK}: x \in \textbf{\textit{val}}\}}$$

The source nodes of G can be immediately evaluated at the beginning of any instant, and their values are substituted for the corresponding variables in CLOCK; source nodes are always elements of CLOCK.

Rule EXEC_1

$$\frac{h \longrightarrow \text{x} \quad \text{and} \quad h = 1}{\{\text{G, CLOCK}\} \xrightarrow{\text{x} \leftarrow \textit{val}(\text{x})} \{\text{G: x} \in \textbf{\textit{val}}, \text{CLOCK}\}}$$

When an input non-bolean signal is known, it can be evaluated.

Rules EXEC_2

$$\frac{\forall \text{x: x} \xrightarrow{0} \text{y}}{\{\text{G, CLOCK}\} \xrightarrow{\text{y} \leftarrow \perp} \{\text{G: y} \in \textbf{\textit{val}}, \text{CLOCK}\}}$$

$$\frac{\{\exists \text{x: x} \xrightarrow{1} \text{y}\} \quad \text{and} \quad \{\text{x} \xrightarrow{1} \text{y} \Rightarrow \text{x} \in \textbf{\textit{val}}(\text{G})\}}{\{\text{G, CLOCK}\} \xrightarrow{\text{y} \leftarrow \textit{val}(\text{y})} \{\text{G: y} \in \textbf{\textit{val}}, \text{CLOCK}\}}$$

If all incoming branches are labelled with a zero clock, then y is known to be absent, and thus considered as evaluated. If y is such that the origin of every (present) incoming branch of G has been evaluated, then y can also be evaluated. No change results in the clock calculus from the use of these rules.

Rule EXEC_3

$$\frac{x \in \textbf{val}(\mathrm{G}) \quad \text{and} \quad x \in ?(\mathrm{CLOCK})}{\{\mathrm{G}, \mathrm{CLOCK}\} \xrightarrow{\mathrm{CLOCK}: x \leftarrow \textbf{val}(x)} \{\mathrm{G}, \mathrm{CLOCK}: x \in \textbf{val}\}}$$

When a new boolean has been evaluated as the result of a non boolean function, then its value is substituted in the clock calculus and all the clocks which can now be evaluated are evaluated. This rule does not modify the graph.

Comments. (1) In the next section we shall give a sufficient condition that guarantees the correct termination of EXEC, namely with a state satisfying

$$?(\mathrm{CLOCK}) = \emptyset, \qquad ?(\mathrm{nodeG}) = \emptyset. \tag{4.13}$$

(2) The technique of substitutions in the clock calculus and graph we have used leads to the following result:

Theorem 1. *Provided that the algorithm terminates correctly in the sense of* (4.13), *then the combination of the transitions of* EXEC *produces the event* α *of some transition* (*cf.* (3.5)) *of the considered program.*

Hence, provable runs of the original program can be obtained via an infinite loop of EXEC (provided that values in memory are properly handled). We cannot guarantee that all provable runs of the original transition system can be realized in this way. Nevertheless, the method we shall give later to transform any process in its pre-solved form is expected to allow the realization of the largest possible set of provable runs.

Example. The program GUARD_COUNT.

Let us show how the algorithm runs on the program GUARD_COUNT.

Step 0. The clock of the counter is known to be the fastest one, therefore we can assume $h = 1$; first, rule EXEC_0 is applied.

$$zn^2 = n^2 = \alpha^2 = \beta^2 = 1,$$

$$e = (-\alpha - 1) + (\beta + 1)(\alpha - 1),$$

$$r = -\beta - 1,$$

$$h \longrightarrow \alpha, \qquad h \longrightarrow \beta, \qquad h \longrightarrow \mathrm{n}, \qquad h \longrightarrow \mathrm{zn},$$

$$\mathrm{I} \xrightarrow{r} \mathrm{n}, \qquad \mathrm{zn} \xrightarrow{1-r} \mathrm{n}, \qquad \mathrm{n} \longrightarrow \alpha,$$

$$\alpha \longrightarrow e, \qquad \beta \xrightarrow{\alpha - 1} e,$$

$$\beta \longrightarrow r.$$

Step 1. Rule EXEC_1 yields $\mathrm{zn} \in \textbf{val}(\mathrm{G})$.

Step 2. By the rule EXEC_2, β can be evaluated, and we immediately apply EXEC_3 in order to substitute β by its value in CLOCK. Finally, for simplicity, we delete the nodes that will not be used any more. In this example, we consider the case when $\beta = -1$.

$$zn^2 = n^2 = \alpha^2 = \beta^2 = 1,$$

$$e = -\alpha - 1,$$

$$r = 0,$$

$$h \longrightarrow \alpha, \qquad h \longrightarrow \mathsf{n},$$

$$\mathsf{zn} \longrightarrow \mathsf{n}, \qquad \mathsf{n} \longrightarrow \alpha,$$

$$\alpha \longrightarrow e, \qquad \beta \xrightarrow{\alpha - 1} e.$$

Step 3. In this case, n is ready to be evaluated by the rule EXEC_2. Again we delete the branches that will not be used any more.

$$n^2 = \alpha^2 = \beta^2 = 1,$$

$$e = -\alpha - 1,$$

$$r = 0,$$

$$h \longrightarrow \alpha, \qquad \mathsf{n} \longrightarrow \alpha,$$

$$\alpha \longrightarrow e, \qquad \beta \xrightarrow{\alpha - 1} e.$$

Step 4. α is ready to be evaluated by the rule EXEC_2. We assume here that $\alpha = 1$ and use the rule EXEC_3 in order to replace α by its value in CLOCK, which terminates the execution:

$$e = 1.$$

4.4. Correct termination of EXEC: fundamental theorems

Two natural questions arise about the preceding algorithm, namely

(1) *Is this algorithm deterministic, i.e. does it exhibit a single provable run* (cf. (3.5))?
(2) *Does this algorithm terminate correctly, i.e. with all nodes and clocks being evaluated* (cf. (4.13))?

These are the questions we want to answer in this section.

4.4.1. Observability and determinism

Referring to (3.23, 3.24), recall that the clock calculus of a program P specifies a set of trajectories in $\mathbb{F}_3^n$ for some n; the algebraic variety specified by the static clock calculus is nothing but the projection on $\mathbb{F}_3^n$ of this set of trajectories along the time axis. We shall denote by

$$V(\mathsf{P}) \tag{4.14}$$

the algebraic variety specified by the static clock calculus. Now, consider a program P, and denote by A the set of the variables involved in its static clock calculus. Denote by B the subset of A composed by the boolean variables involved in a non-boolean relation, and by Ξ the set of the (old) boolean memories.

Definition 4. Consider $\Omega \subset A$ (Ω is intended to refer to a subset of some 'visible' ports of P). We shall say that P is *observable* via Ω if the following condition is satisfied by the static clock calculus of P:

$$\text{every point of } V(P) \text{ is entirely determined by its components in } \Omega \cup B \cup \Xi. \tag{4.15}$$

The following theorem holds, which justifies the definition:

Theorem 2. *If* P *is not observable via* Ω, *there exist at least two different provable runs of the synchro-process* (3.43) *associated with* P *that agree on* Ω.

Conversely, if P *is observable by* Ω, *and if* Ω *contains all the source nodes of the conditional dependence graph* CDG, *then there exists at most one provable run of* P *according to the original SIGNAL rules* (3.8, 3.10, 3.11, 3.12, 3.7), *which agree on a specified trajectory* $\{(\Omega_t, B_t)\}_{t>0}$ *of the components in* $\Omega \cup B$ *of the clock calculus.*

The proof is given in the report [5]. Let us explain this theorem. Roughly speaking, when Ω refers to 'input ports', the first assertion means that, if P is not observable via Ω, then the synchro-process associated with P is not deterministic. Unfortunately, we cannot derive from this property that the program P itself is non-deterministic since this would require to reason about any data type, which is something we cannot do. However, in the same situation, the second assertion means that observability does imply determinism of the original program. To summarize, this theorem provides a condition for determinism, which is sufficient and almost necessary. This criterion provides a formal framework to support the kind of argument we mentioned in the preceding examples for checking non-determinism in the SIGNAL programs. Checking property (4.15) relies on the presence of phantoms in the elimination procedure which is used to solve the static clock calculus (cf. the examples above, and the sketchy presentation of such an elimination procedure in the next section).

4.4.2. *The clock of a cycle of G: a tool for deadlock isolation*

Consider a node x in the graph G of the considered synchro-process, and assume that it satisfies the following property:

$$\{h : y \longrightarrow x\} \in \text{CDG} \quad \text{and} \quad x \in B, \quad \text{referred to as 'x free boolean'}. \tag{4.16}$$

In other words, x is a boolean which is the result of the evaluation of a non-boolean function (for instance $x := (u < v)$), hence the name 'free' since our clock calculus cannot compute its actual value. The clock of a cycle of G is defined now.

Step 1. Consider a cycle

$$C = x_1 \xrightarrow{h'_1} x_2, \ldots, x_n \xrightarrow{h'_n} x_1$$

of G, and denote by B the (possibly empty) subset of the x_i's which are *free booleans.*

Step 2. Each equation $h'_i = 0$ defines an algebraic variety V'_i. Denote by V_i the smallest variety containing V'_i which is *invariant* by the group of symmetries $x \rightarrow -x \ \forall x \in B$, and denote by h_i the polynomial such that $h_i = 0$ defines the variety V_i.

Step 3. The clock of the considered cycle is defined as

$$\textbf{clock}(C) = \prod_{i=1}^{n} h_i \tag{4.17}$$

In other words, inside a cycle of G, we extend the dependencies to the least frequent clock which is

1. more frequent than the product of the original clocks of the branches,
2. independent of the actual values of the free booleans which will be evaluated within the considered cycle.

Obviously, if no free boolean belongs to the nodes of this cycle, we just get the product of the original clocks.

Important Remark. Since the graph of a synchro-process is not invariant via isomorphisms, the cycles depend on the particular form of the clock calculus. Hence it is expected that some cycles can be broken[5] via suitable manipulations of the clock calculus. This idea will be exploited further. Cycles which cannot be broken by manipulations of the clock calculus will generally cause *deadlocked subprocesses* to appear in the considered process, since the equation ***clock***$(C) = 0$ must be included anyway as an additional constraint as the Fundamental Theorem 3 below will show. Hence the notion of cycle-clock will be a basic tool to check and isolate deadlocks; this will be illustrated in the examples below. Of course, deadlocks can also result from contradictory statements on boolean signals (such as, for instance, b ***and not*** b = *true*).

4.4.3. The fundamental theorem about correct termination of EXEC

Fundamental Theorem 3. *The conditions* (i, ii, iii) *below ensure that* EXEC *terminates correctly for the process* P, *i.e. that all nodes and clocks have either been evaluated or proved to be absent from the considered instant* (*condition* (4.13)):

(i) *The clock calculus is pre-solved.*

(ii) P *is observable with respect to the observer composed by its source nodes and free booleans.*

[5] See the definition of the actions of EXEC.

(iii) *All cycles of its graph* G *have zero clock.*
Furthermore, if these conditions are satisfied, for each history of the source nodes of its graph G, *and each sequence of effective* (*i.e.* $\neq \perp$) *values of the source nodes of its conditional dependence graph* CDG, *there exists exactly one provable run of* P, *and this run can be realized by a repeated use of the algorithm* EXEC.

In other words, processes which are observable by their inputs as well as cycle free in the sense of (iii) are *deadlock free and deterministic*, and their runs can be realized by the executable code produced by the SIGNAL compiler. The proof of the first statement is given in [4], and the proof of the second one is given in the report [5].

4.5. Solving static clock calculi

Our purpose here is to investigate how to transform the clock calculus of any synchro-process to get the form mentioned in the Fundamental Theorem 3, i.e. a form suitable to a correct termination of EXEC. To help the resolution, we need to handle graphs for synchro-processes which are not in the pre-solved form: we shall use the rule GRAPH_4 for this purpose.

The clock calculus is solved in the following way:

Step 1. Perform all possible substitutions of the left-hand side by the corresponding expression in the right-hand side of y *or* $y^2 = \cdots$ until implicit equations are encountered. When several equations of the form y *or* $y^2 = \cdots$ are encountered, select one as the definition equation of the left-hand side and form a constraint by expressing that the two right-hand sides must be equal (an elementary way of performing elimination). Definition equations of the form y *or* $y^2 = Q(\textit{freebool})$, where Q is any polynomial and *freebool* denotes any free boolean, are preferred in the case of selecting a definition equation among several ones.

Step 2. Build the graph G of the so obtained synchro-process. Define on the set of the vertices of G the following equivalence relation denoted by $x \leftrightarrow y$: $x \leftrightarrow x$ and $x \leftrightarrow y$ if x and y belong to the same strong connectivity class (i.e. there is a path from x to y and vice-versa). Then $G/\leftrightarrow$ is a circuit-free graph; denote by $\{G_i\}_{1 \leq i \leq n}$ the subgraphs of G which are mapped onto vertices of G/$\leftrightarrow$ where the index n is compatible with the partial order on these subgraphs. These subgraphs will be simply called *strong connectivity classes* in the sequel. Denote by

$$C_{k_1}, \ldots, C_{k_p}$$

the circuits of the G_i's which possess at least one branch of one of the forms

$x^2 \xrightarrow{h}$ vertex (originating from GRAPH_1),

vertex $\xrightarrow{h}$ vertex (originating from GRAPH_2).

Such cycles will be called *data-cycles*: they cannot be broken[6] by transformations of the clock calculus. Hence, for each data cycle, we must add the following rule which ensures the condition (iii) of the Fundamental Theorem 3:

$$\frac{C \text{ data-cycle}}{\text{add } \boldsymbol{clock}(C)=0 \text{ to the clock calculus}} \tag{4.18}$$

where the clock of C has been defined in (4.17). Notice that (4.18) in general modifies the graph G, *but does not add new data-cycles.* After Step 2, data-cycles are broken.

Step 3. Using the rules for elimination of Lemma 1, the remaining connectivity classes are broken successively, starting from the least one (according to the partial order induced by G). This is done as follows. Data-cycles are not modified since they have already been handled. Elimination within a connectivity class terminates with the variables of the class which are successors of nodes of G which do not belong to the class (the 'source nodes of the connectivity class').

Result. If this procedure terminates with no phantom, the assumptions of the Fundamental Theorem 3 are satisfied. The procedure we have presented informally is described in [4] via the technique of transition systems.

Discussion. As will be shown in the examples, this procedure isolates the subset of the ports that are deadlocked in the considered process. Hence this procedure is a fundamental tool for programming fault isolation. The resulting graph is also a convenient starting point to target the considered application on a multiprocessor architecture, see [18, 27].

4.6. Examples

Our purpose in this section is to show how the preceding procedure handles spurious programs, to detect and isolate deadlocks, or transform a program into an executable form. Hence some pathological examples will be reviewed.

4.6.1. A wrong synchronization

Recall the following example which has been introduced before:

```
| x := u when (u<v)
| y := x+v
```

Writing, for short, β instead of (u<v) yields the clock calculus

$$y^2=x^2=u^2=v^2=\beta^2=h,$$
$$x^2=u^2(-\beta-\beta^2)$$

[6] See the definition of the actions of EXEC.

Due to this clock calculus, the graph of this program exhibits a cycle, namely

$$C = h \xrightarrow{-\beta-\beta^2} h$$

Hence, the clock of this cycle has to be zero. But this clock is equal to

$$clock(C) = (-\beta-\beta^2)^2 + (\beta-\beta^2)^2 = \beta^2.$$

(We used the rule $\{p=0 \text{ and } q=0\} \Leftrightarrow \{p^2+q^2=0\}$.) Hence the constraint $\beta=0$ must hold, which means that the process stays in deadlock. In this example, the synchronisation was incorrect.

4.6.2. A data-cycle

The following example is due to G. Gonthier (private communication); its interpretation is

if z > 0 ***then*** z := a else z := b.

A program corresponding to this informal specification should be rejected. A corresponding SIGNAL program is

```
synchro a,b
β = (z > 0)
x := a when β
y := b when not β
z := x default y
```

The clock calculus is

$$a^2 = b^2 = h,$$

$$x^2 = h(-\beta-\beta^2),$$

$$y^2 = h(\beta-\beta^2),$$

$$\beta^2 = z^2 = x^2 + y^2(1-x^2) = h\beta^2.$$

The graph contains in particular the following branches:

$x^2 \xrightarrow{x^2} \text{x}$,	$y^2 \xrightarrow{y^2} \text{y}$		by GRAPH_1
$\text{z} \xrightarrow{\beta^2} \beta$,	$\text{x} \xrightarrow{x^2} \text{z}$,	$\text{y} \xrightarrow{y^2} \text{z}$	by GRAPH_2
$\beta \xrightarrow{\beta^2} x^2$,	$\beta \xrightarrow{\beta^2} y^2$		by GRAPH_3.

Two cycles are exhibited; their clocks are both equal to $h\beta^2$. Hence, these cycles add to the clock calculus the constraint $\beta=0$. As a result, this program accepts the

inputs (a, b), but refuses to produce any other signal. The isolated deadlock involves the signals x, y, z, β; such an isolation can be used for fault recovery.

Chapter 5. Conclusions

We have presented the kernel of the SIGNAL synchronous programming language. We illustrated the SIGNAL programming style on a typical example relevant to real-time control systems. We discussed in a fundamental way the mathematical semantics and execution schemes of this language. While the usefulness of synchronous languages in the area of reactive system is established, we hope to have shown how SIGNAL can be used to proceed towards automatic synthesis of executable programs from their specifications. The following key features should be mentioned:

- SIGNAL is a block-diagram oriented language. As such, it is provided with a graphical interface for program editing and execution, see [27] for further information.
- Since block-diagrams naturally specify constraints or relations between the involved signals, SIGNAL is a language of *equational* style. This has several important consequences we list now:
  - The programmer has only to specify *local* synchronization constraints involving few signals; synthesizing the whole synchronization is the task of the compiler.
  - SIGNAL is its *own proof system*: desired properties can be expressed as (possibly non-deterministic) SIGNAL programs, and processed by the compiler as additional equations. Checking for contradictions in the resulting program is the mechanism for proofs, see [27] for further information.
  - The behavior of a program P in a context C may be easily studied as a program C|P (proofs, simulation, ...).
- The conditional dependence graph associated with a program is the universal tool for proving, distributing, optimizing SIGNAL programs, see [27].

Finally, issues of executing (distributed) SIGNAL programs in an asynchronous environment are common to all synchronous languages, see [/] for a detailed discussion of this topic. To summarize, various services such as proof, compilation, distributed implementation, are all supported by the SIGNAL formal system. This releases the user from handling different formalisms and associated tools for these tasks.

SIGNAL is currently available under two different versions that were developed with different objectives. The INRIA H2 SIGNAL system provides a block-diagram interface presented in [27]. Its compiler implements a subset of the full clock calculus we have presented here: the reason for this is to provide a fast compilation method, by avoiding heavily computational steps of the procedure we have presented. A brief presentation of this implementation of the SIGNAL compiler is given in [27].

Sequential FORTRAN code is currently produced. Developments on distributed implementation are in progress based on this version. Tools for proving dynamical properties will be integrated in a short time.

The CNET-TNI V3 version is commercially available. A multiple windowing system of Macintosh style is provided for both program editing and on-line monitoring and supervision of the execution. Sequential C code is produced. Experiments have been performed based on this version to produce distributed OCCAM code for a multi-Transputer system.

The SIGNAL environment has been experimented on significant applications in the area of signal processing and control: a speech recognition system, a radar system, a digital watch, a rail road crossing were the major ones.

References

[1] C. Bechon, Thesis, University of Rennes I, 1988.

[2] A. Benveniste and P. Le Guernic, A denotational theory of synchronous communicating systems, INRIA Res. Rep. No. 685, 1987, to appear in *Information and Computation*, 1991.

[3] A. Benveniste and P. Le Guernic, Hybrid dynamical systems theory and nonlinear dynamical systems over finite fields, *Proc. 1988 IEEE Control and Decision Conference*, Austin, TX, 7-9 Dec. 1988.

[4] A. Benveniste, B. Le Goff and P. Le Guernic, Hybrid dynamical systems theory and the language SIGNAL, INRIA Res. Rep. No. 838, 1988.

[5] A. Benveniste, P. Le Guernic and C. Jacquemot, Synchronous programming with events and relations: the SIGNAL language and its semantics, IRISA Res. Rep., 1989.

[6] A. Benveniste and P. Le Guernic, Hybrid dynamical systems theory and the SIGNAL language, *IEEE Trans. Automatic Control* **35** (5) (1990) 535-546.

[7] A. Benveniste and G. Berry, Real-time systems design and programming, to appear in *Proc. IEEE*, special section on real-time programming, 1991.

[8] J.L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud and E. Pilaud, Outline of a real-time data-flow language, in: *Real Time Systems Symposium*, San Diego, CA, Dec. 1985.

[9] G. Berry and L. Cosserat, The ESTEREL programming language and its mathematical semantics, INRIA Res. Rep. No. 327, Rocquencourt, France, 1984.

[10] G. Berry and G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation, CMA Res. Rep., 1988 to appear in *Science of Computer Programming.*

[11] B. Bloom, S. Istrail and A.R. Meyer, Bisimulation can't be traced, *Proc. POPL'88.*

[12] J.D. Brock and W.B. Ackerman, Scenarios, a model of non-determinate computation, *Conf. Formal Definition of Programming Concepts*, Lecture Notes in Computer Science **107** (Springer, Berlin, 1981).

[13] S.D. Brookes, C.A.R. Hoare and A.W. Roscoe, A theory of communicating sequential processes, *J. ACM* **31** (3) (1984) 560-599.

[14] B. Buchberger, Ein algorithmisches Kriterium für die Lösbarkeit eines algebraisches Gleichungssystems, *Aequat. Math.* **4** (1970) 374-383.

[15] B. Buchberger, A criterion for detecting unnecessary reductions of Groebner bases, *Eurosam 79*, Lecture Notes in Computer Science **72** (Springer, Berlin, 1979) 3-21.

[16] A. De Bruin and W. Boehm, The denotational semantics of dynamic network of processes, *ACM Trans. Prog. Lang. Syst.* **7** (4) (1985) 656-679.

[17] P. Caspi, D. Pilaud, N. Halbwachs and J.A. Plaice, LUSTRE: a declarative language for programming synchronous systems, *Proc. 14th ACM Symp. Principles of Programming Languages,* 1987.

[18] C. Figueira, T. Gautier, B. Le Goff and P. Le Guernic, Towards multiprocessor implementation of real-time, data-flow programs, *The 1988 International Symposium on LUCID and Intentional Programming,* Victoria, Canada, April 6-8, 1988.

[19] T. Gautier, P. Le Guernic and L. Besnard, SIGNAL, a declarative language for synchronous programming of real-time systems, in: G. Kahn, Ed., *Proc. third Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **274** (Springer, Berlin, 1987).
[20] G. Gonthier, Thesis, Univ. de Nice and Ecole des Mines, 1988.
[21] D. Harel, Statecharts: a visual approach to complex systems, *Science of Computer Programming* **8** (3) (1987) 231-275.
[22] D. Harel and A. Pnueli, On the development of reactive systems: logic and models of concurrent systems, *Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*, NATO ASI Series F **13** (Springer, Berlin 1985), 477-498.
[23] C.A.R. Hoare, Communicating sequential processes, *Comm. ACM* **21** (8) (1978) 666-678.
[24] G. Kahn, The semantics of a simple language for parallel programming, in: J.L. Rosenfeld, Ed., *Proceedings IFIP 74* (North-Holland, Amsterdam 1974) 471-475.
[25] G. Kahn and D.B. MacQueen, Coroutines and network of parallel processes, in B. Gilchrist, Ed., *Proceedings IFIP 77* (North-Holland, Amsterdam, 1977) 993-998.
[26] P. Le Guernic, A. Benveniste, P. Bournai and T. Gautier, SIGNAL: a data-flow oriented language for signal processing, *IEEE Trans. Acoust. Speech Signal Process.* **34** (2) (1986) 362-374.
[27] P. Le Guernic, T. Gautier, M. Le Borgne and C. Le Maire, Programming real-tine applications with SIGNAL, to appear in *Proc. IEEE*, special section on real-time programming, Sept. 1991.
[28] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science **92** (Springer, Berlin, 1980.
[29] R. Milner, Calculi for synchronicity and asynchronicity, *Theoret. Comput. Sci.* **25** (3) (1983) 267-310.
[30] J.S. Ostroff, Real time computer control of discrete systems modelled by extended state machines: a temporal logic approach, Rep. No. 8618, Systems Control Group, Dept of Elec Eng., Univ of Toronto, 1986.
[31] J.A. Plaice, Semantique et compilation de LUSTRE, un langage declaratif synchrone, Thesis, Institut National Polytechnique de Grenoble, 1988.
[32] G.D. Plotkin, A structural approach to operational semantics, Lecture Notes, Aarhus Univ., 1981.