

# The Esterel synchronous programming language and its mathematical semantics

Gérard Berry, L. Cosserat

▶ **To cite this version:**

Gérard Berry, L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. [Research Report] RR-0327, INRIA. 1984. inria-00076230

**HAL Id: inria-00076230**
**https://inria.hal.science/inria-00076230**

Submitted on 24 May 2006

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

CENTRE
SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tel. (3) 954 90 20

Rapports de Recherche

N° 327

# THE ESTEREL SYNCHRONOUS PROGRAMMING LANGUAGE AND ITS MATHEMATICAL SEMANTICS

Gérard BERRY
Laurent COSSERAT

Septembre 1984

# The ESTEREL Synchronous Programming Language and its Mathematical Semantics

**Gérard Berry**
**Laurent Cosserat**

Ecole Nationale Supérieure des Mines de Paris (ENSMP)
Centre de Mathématiques Appliquées
Sophia-Antipolis
06565 VALBONNE
FRANCE

**Résumé**

Nous présentons le langage de programmation temps-réel ESTEREL et sa sémantique mathématique. Contrairement aux langages classiques de type CSP, ESTEREL est un langage synchrone, déterministe et fondé sur une notion de temps multiforme. La sémantique mathématique est du type opérationnel, et est donnée sous forme de règles de réécriture conditionnelle à la Plotkin. Elle permet de définir complètement le comportement temporel des programmes et aussi de les compiler efficacement sous forme d'automates à contrôle fini.

Le projet ESTEREL est financé par un contrat ADI (Convention de recherche n° 82/625).

**Abstract**

We present the real-time programming language ESTEREL and its mathematical semantics. Contrarily to CSP-like languages, ESTEREL is a synchronous deterministic language based on a multiform notion of time. The mathematical semantics is a structural operational semantics given by conditional rewrite rules in Plotkin's style. It characterizes completely the temporal behavior of programs, and allows us to compile efficiently an ESTEREL program into a finite control automaton.

The ESTEREL project is supported by an ADI contract (Convention de recherche n° 82/685).

PAPIER RECUPERE ET RECYCLE

# The ESTEREL Synchronous Programming Language and its Mathematical Semantics

**Gérard Berry**
**Laurent Cosserat**

Ecole Nationale Supérieure des Mines de Paris (ENSMP)
Centre de Mathématiques Appliquées
Sophia-Antipolis
06565 VALBONNE
FRANCE

## 1. INTRODUCTION.

The goal of the ESTEREL project is to develop a real-time language based on a *rigorous formal model*, and actually to develop simultaneously the language, its semantics and its implementation. The present paper presents a reasonably stabilized version of ESTEREL together with its formal structural operational semantics. The language is rather unclassical since it is purely *synchronous*, *deterministic*, and based on a *multiform* notion of time, while all parallel and "real-time" languages such as ADA [3], CSP [19], LTR [1], OCCAM [4], RTL/2 [6], are asynchronous, nondeterministic, and consider only one notion of "absolute" time for their temporal primitives. The mathematical semantics leads naturally to a compiler producing efficient finite automata from ESTEREL programs, with possibility of temporal analysis by systems such as Clarke's EMC [14].

Let us first analyze the notion of time in ESTEREL and the synchrony hypothesis. In most "real-time" languages one can write delay instruction with classical temporal units (typically seconds), Hence one can write something like

```
delay 2 s; delay 3 s
```

How does one define what that statement *means*? Is it *equivalent* to the statement "delay 5 s"? The answer to the first question is often not given, and the answer to the second question may very well be negative since usual languages are essentially *asynchronous*. Take for example the OCCAM language, where an external event such as "second" is treated just as an ordinary message (and hence receives a clear semantics). Then the mentioned equivalence has no reason to be true, since messages are treated in a purely asynchronous way: one may expect the equivalence to be "true" if the implementation is "fast enough", but one has no real control on what will really happen. For actual real-time applications that approach is not sufficient, since one wants a program

to respond to externally generated input stimuli within a "controllable" delay (see Young's definition or real time in [29]). Although the term "controllable" is vague enough, it is in no way a synonym of "arbitrary" and asynchrony can do little for us here. Temporal statement should be semantically well-defined at least at a "conceptual level" suited to reason about programs, and the validity of their implementation should be really checkable (i.e. one should have a reasonable idea of the actual temporal behavior of the compiled code),

Our goal in ESTEREL is to treat completely these two problems in the rational case, i.e. in the case where there is no dynamic creation of processes. We have no other choice than building a *synchronous* language, in which a delay statements terminates *exactly* when its ending event occurs (at least in the formal semantics). With respect to the equivalence problem, we shall take a rather drastic choice and say that the sequencing operator ";" *takes no time at all*, in other words that our execution machine is infinitely fast. This will generalize to all other ESTEREL constructs: all control transmission inside a program and all simple operations (assignment, addition etc) will "take no time". The only instruction which may take time are the ones *explicitely required to do so*, for example a "delay" statement. A program has no "internal clock", and it simply reacts instantaneously to external stimuli producing itself some stimuli to its output lines, and does *nothing* in absence of external stimuli. The conjunction of the synchrony hypothesis and of the "control takes no time" hypothesis will be called the *strong synchrony hypothesis*. In a strongly synchronous framework, it is obvious that "delay 3 s; delay 2s" is indeed the same as "delay 5 s".

The strong synchrony hypothesis may seem totally unrealistic at first glance, since real machines are of course not infinitely fast. However we shall see that it is in fact extremely useful and has surprisingly good consequences: it really *simplifies* many problems, and for example allows us to reduce the number of primitives of the language (the hypothesis being necessary to ensure that many useful derived constructs work correctly). It suppresses the need for nondeterminism, which is usually necessary for handling communication between parallel processes but is not so natural in most real-time application. More surprisingly it leads to a truly *efficient implementation* of ESTEREL programs, where the *exact timing* of operation can be measured and where the assumption of having an infinitely fast machine is not at all unrealistic in most practical applications. We shall come back on this point in a moment.

We turn now to the study of the notion of *time*. A first remark is that manipulating only a notion of "absolute time" is not enough for most real-time applications. In fact many control algorithms use much more varied notion of time, if one considers a "time unit" as being just a repetitive event of some kind. For example a mile runner knows four different natural time units: the second of course, but also the step, the meter run and the lap elapsed. All these units are of the same temporal nature, and there is no reason not to write statements such as "delay 3 meter" or "at 3 laps do <action>". Therefore a

typical ESTEREL program will look like

```
during 2 LAP do
   every LAP do
      during 100 METER do
         RUN-SLOWLY
      end;
      during 100 STEPS and at most 50 SECOND do
         every STEP do
            [ JUMP-HIGH || BREATHE-OUT ]
         end
      end;
      WALK-SLOWLY
   end
end
```

The role of the "during" construct (which is actually not really primitive and will be derived from two "upto" and "uptonext" constructs) is to define the *temporal extent* of its body, measured versus the appropriate time unit, the "every" construct being simply a loop over a "during" construct. We leave to the reader to write the above program in a classical language and to discover how complex are the synchronizations it contains. As another example, here is a natural way of specifying an *exact* speed measure:

```
var SPEED : int in
   loop
      every 10 seconds do
         SPEED:=0;
         every METER do
            SPEED:=SPEED+1
         end
      end;
      emit SPEED-MEASURE (SPEED)
   end
end
```

With the strong synchrony hypothesis one can *guarantee* that an exact speed measure will be emitted exactly every 10 seconds. As we shall see in various examples, ESTEREL allows us to write small and elegant programs and to analyze them rigorously.

The next question is obvious: admitting that our strong synchrony hypothesis is reasonable for a specification language, how good is it for a real programming language, how far will our specification be from a real program? We have several answers. The first one is that there are many application in which the difficulty is *not* related to the speed of computation, but only to the complexity of the interaction between signals: this is the case when one wants to

control *slow processes*; that case turns out to be quite frequent in practice (an object like a train has intrinsically high time constants; the problem for controlling it is not to write fast programs but *correct* programs). The second answer concerns the way programs are executed. The inefficiency of many parallel languages is due to the fact that programs are executed just as they are written, with an important overhead due to the process handling. In ESTEREL we shall overcome this difficulty by *compiling* a program into a simple and efficient input-output automaton, just as a grammar is translated into an efficient automaton by a parser generator. Then the execution times of apparently complex tasks may really be very low and reasonably close to the theoretical zero of ESTEREL, since all the internal control transmission and process communication corresponding to the treatment of an external event have been compiled into a single state transition. The strong synchrony hypothesis will also prove very useful for avoiding the explosion in the number of states which is the rule in asynchronous formalisms, and is due to the fact that internal control of the program generates states (problem which does not exist in ESTEREL). One could even go further and realize functions specified by ESTEREL programs by hardware components, for which the notion of synchrony makes real sense. In any case we think that the advantages of having a clean synchronous language overcome the possible inconvenients.

Let us now analyze the relation between the language and its semantics. The process of developing a language together with its semantics is now widely admitted and actually used for classical sequential languages. But the situation is not so advanced for parallel or real time languages: on one hand the asynchronous languages we mentioned above were developed mainly on pragmatic grounds, often by adding various synchronization and communication primitives to rather classical sequential languages; on the other hand, many models of parallel computations have been developed in a quite independent way: denotational semantics based on powerdomain constructions [23], Petri Nets [22, 2], process algebras such as CCS, SCCS or MEIJE [20, 21, 9], temporal logics [26]. The relation between models and languages are still not very clear: the models tend to depart more and more from the classical sequential languages models and to become more and more elegant, but they are not so easy to use for giving natural semantics of actual languages or to compare the respective power of possible communication primitives. (although there are for instance some relations between rendez-vous in CSP and CCS [18, 5]). These problems were particularly clear during the development of ESTEREL, since we made the choice of introducing primitives only when their formal semantics were clear, but at the same time lacked good tools for giving the semantics. We actually developed several intermediate versions of the language and semantics [7, 11, 8], which all contained imprecisions or even mistakes revealed only by a careful analysis of the proposed semantics (and which should now be considered as obsolete). At that time we used the only existing synchronous models, namely Milner's SCCS [21] or Boudol's MEIJE [9] calculi of processes. Our main difficulty was to give natural translation of ESTEREL constructs into SCCS or

MEIJE ones. The final solution was obtained when we decided to forget about such translations and to use a *direct structural operational semantics* à la Plotkin [25, 24]. We shall in fact propose three semantics of ESTEREL :

(i) A *static semantics*, the role of which is mainly to check that a program raises no "temporal paradoxes", of the same kind as short-circuits in electricity.

(ii) A *behavioral semantics*, which defines completely the temporal behavior of a program which is correct versus the static semantics. This semantics is suited for studying program behaviors or program equivalences, but not for computing what a program does, in other words for executing the program : it relies on the fact that some fixpoint equations have a unique solution, but does not really give ways of computing the solution.

(iii) A *computational semantics*, which is more complex than the behavioral semantics, but also more effective in the sense that it really allows us to compute what a program does.

A remarkable fact is that the computational semantics we give is really *executable*: once its was fully understood, we were able to write a prototype interpreter of ESTEREL within a few days (using the LELISP/CEYX environment [13]). An even more remarkable fact is that the computational semantics leads directly to a *compiler* from ESTEREL programs to finite states machines, where the states are just program texts and the transitions program execution steps. Once a machine is constructed, one may forget about the programs associated with its states, and replace them simply by state numbers. The finite state machine is then trivially implementable in any classical language. It realizes a sequential implementation of an originally parallel synchronous program.

In section 2 we present the kernel language and its "naive" semantics: we define the notions of signal, event and control, and give the basic instruction set. In section 3 we propose natural extended instructions useful in real programs; the way we define these instructions gives some insight into the ESTEREL programming style. In section 4 we give two examples of ESTEREL programs. Sections 5, 6 and 7 are devoted to the three semantics, namely the static, behavioral and computational semantics. In section 8 we indicate briefly how ESTEREL symbolic evaluators and compilers can be derived from the computational semantics (they will be presented in more details in another paper). The last section indicates the futures research directions of the ESTEREL project.

## 2. The kernel language.

### 2.1. Types, variables,expressions and declarations

The treatment of variables and values is quite classical and not very much relevant to what follows, so that we shall give little detail here. We assume given some basic types, say integer, string and boolean, with adequate denotations for their values (the user will be able to introduce its own types in full ESTEREL

versions). We may also use *type constructors* such as records or arrays. In the present paper we shall use a polymorphic *list* constructor valid for any type, with the usual functions head, tail, cons, append, null. List constants will be written with brackets as in "[1,2,3]".

The variable declarations have the form

var X: *type*

The expressions are constructed in a completely classical way. They may contain calls to externally defined functions (on data only, not on events).

**2.2. Signal and events.**

The basic communication unit in ESTEREL is the *signal*. A signal has a *name* and possibly a *type*, which is the type of the values it may convey.

An *event* is an *instantaneous and broadcasted flash of information*, where an "information" is formed of possibly many simultaneous signals conveying values. An event is not remanent, so that its information is simply lost in the air if nobody listens to it.

Events may come from the outside of a program (e.g. interrupts) or be internally generated by the "emit *signal*" instruction, see below. We make no difference between these two cases.

We allow to have several simultaneous emissions of the same signal with possibly different values. A typical case is resource allocation, where several consumers may want simultaneously some resource: they send an inquiry message containing their name, the manager of the resource sends an acknowledgement to one of them (i.e. broadcasts a message containing the name of one of them). Another well-known case is local network protocols.

We have then to determine what values the receivers of such an event will then receive. The idea is taken from Milner [21]. With any signal of type $t$ we associate an associative and commutative operation $*$ of type $t \times t \rightarrow t$ . If there are $n$ emitters ($n>0$) emitting $n$ values $v_1, v_2, \ldots, v_n$ then the receivers will receive the value $v_1 * v_2 \cdots * v_n$.

As natural examples, one may send *pure signals* by considering a type $t$ with only one value 1, setting $1 * 1 = 1$. One may instead emit and receive *sorted lists* of values, the product operation "sortappend" appending and sorting the lists. This amount to gathering all emitters into a list, and is perfectly suitable in the above resource allocation example. One may also say that the product of two objects is a special value "error", so that one will dynamically detect that there was more than on emitter (typical example : local networks).

A signal declaration has the form

signal *name* (*type* , *product*)

where *name*, *type* and *product* are the name of the signal, its type, and the name of its $*$ operation. Signals will usually be named s, s1 etc.

The events will be called $E, E', F$. They have the following form :

$$s_1^{v_1}.s_2^{v_2}. \cdots .s_n^{v_n}$$

where $v_n$ is the value conveyed by $s_n$ The operations $*$ naturally induce a commutative monoid structure on the set of events : the neutral element 1 contains no signal, and one derives the $*$ product of two events in an obvious way by setting

$$s^{v_1} * s^{v_2} = s^{v_1 * v_2}$$

We say that a signal s *belongs* to an event $E$ and write $s \in E$ if s is present in $E$ with some exponent $v$. We then write $E(s)=v$.

### 2.3. The discrete time model

The strong synchrony hypothesis naturally lead to a discrete time model: we shall deal with sequences of events, that we shall call *histories* $H=E_1,E_2, \ldots, E_n, \ldots$. The pair $(n,E_n)$ will be called the n-th *instant* of $H$. An *input history* for a program will be an history such that only $E_1$ may contain no signal, i.e. be equal to 1: the idea is that a program will do something only on reception of an event from its environment, except when it is started (in that case it may decide on its own to send signals to its environment). A program P produces an *output history* $H'=F_1,F_2, \cdots, F_n \cdots$. Conceptually the event $F_n$ is formed by the signals emitted by P at instant $(n,E_n)$ i.e. upon reception of the input event $E_n$. Since reception of $E_n$ and emission of $F_n$ are synchronous, an external observer will observe simply the composite event $E'_n=E_n * F_n$.

One should always remember that a program does nothing "between" input instants.

### 2.4. The basic instructions and their naive semantics.

We describe informally the basic primitives used for forming the ESTEREL instructions (or terms, or statements). We use an implicit notion of "control flow", and say that an instruction "passes the control" when it terminates and allows other instructions in sequence to proceed. Passing the control always takes no time, in the sense that it takes place in the current instant. If the control reaches the end of a program, i.e. if the whole instruction constituting the program passes the control, then the program does nothing anymore. We say that an instruction "takes no time" if it passes the control at he same instant it receives it, and that it "takes time" otherwise.

We use the letters $i, i', i_1$ for denoting instructions. We use brackets "[" and "]" to parenthesize instructions in the concrete syntax with usual priority rules, so that ";" has precedence over "||". Comments start with "%" and end at newline. we use the metasymbols *exp* and *boolexp* to range over expressions. For

variables, tag labels and signals, we write simply X (or Y), T, s.

We do not give a precise definition of the way a program transforms histories for the moment, this will be done only when we will give the behavioral semantics.

Here is the list of the ESTEREL instructions, which we shall always treat in that order in the naive or in the formal semantics :

nothing
X:=*exp*
emit s(*exp*)
do $i$ upto s(*exp*)
do $i$ uptonext s(*exp*)
$i_1$; $i_2$
loop $i$ end
if *boolexp* then $i_1$ else $i_2$ fi
$i_1 \parallel i_2$
tag T in $i$ end
exit T
var X : *type* in $i$ end

### 2.4.1. The nothing instruction.

This instruction does nothing in no time. In particular the output history associated with a program containing only the nothing instruction is a sequence of 1's for any input history.

### 2.4.2. The assignment instruction.

An assignment has the form

X:=*exp*

The computation of the expression and the assignment itself take no time, so that the assignment statement passes the control in the same instant it gets it. (In practice it may be unreasonable to assume that the computation of an expression takes no time - for example if it is a matrix inversion. Then one has to create signals for sending the data to an external program and for receiving the output of that program, not synchronously.)

### 2.4.3. The emit instruction.

It has the form

emit s(*exp*)

where *exp* is an expression having the required type (given at the signal declaration). An emit instruction always passes the control in the same instant.

For emitting two signals s1 and s2 simultaneously, one just writes

emit s1($exp_1$) || emit s2($exp_2$)

or indifferently

emit s1($exp_1$); emit s2($exp_2$)

according to the semantics of the operators "||" and ";" described below.

**2.4.4. The upto statement.**

It has the form

do $i$ upto s(X)

where X is a variable of the appropriate type. The idea is as follows: the upto instruction defines the "end of the world" for its body $i$, which is executed until the first occurrence of s. At the first instant where an s occurs, nothing of $i$ is executed, X is bound to the current value conveyed by s, the upto instruction terminates and passes the control. Three facts need to be noticed :

(i) If the signal s is present when the upto instruction receives the control, then the body $i$ is *not executed at all.* In any case when s occurs $i$ is *instantaneously killed* without receiving the control in the corresponding instant.

(ii) The termination of $i$ does *not* provoke the termination of the upto instruction, which simply waits for s.

(iii) There is a "temporal paradox" if $i$ is "emit s($exp$)", we shall come back on that point in section 5.

**2.4.5. The uptonext instruction.**

Notice that waiting twice in a row for an event with an upto instruction is useless; in the instruction

do $i_1$ upto s(X); do $i_2$ upto s(X)

the instruction $i_2$ will never be executed: $i_1$ will be killed at the first occurrence of s, but since the sequencing operator ";" takes no time the instruction "do $i_2$ upto s(X)" will receive the same event and hence also the signal s. It will terminate instantly. Hence we need a way to skip the current instant. This is the purpose of the uptonext instruction, which has the form :

do $i$ uptonext s(x)

It behaves just as the upto instruction, except that an occurrence of s at the instant where it receives the control is *ignored*, so that $i$ is always executed in the instant where the uptonext instruction receives the control. Notice that an

uptonext instruction will always "take time", (unless it contains exits, see below). It is actually the only instruction having this property.

The upto and uptonext instructions are used both for synchronization and value communication.

### 2.4.6. The sequencing instruction.

Sequencing is written as usual :

$i_1$; $i_2$

The control is passed instantaneously to $i_1$ when received. It is then passed instantaneously to $i_2$ when $i_1$ terminates (if ever).

### 2.4.7. The loop instruction.

There is a single loop instruction, written

loop $i$ end

Its semantics is obvious : it behaves just as the infinite sequence

$i$; $i$; ... ; $i$; ...

However the body $i$ is not allowed to pass the control instantaneously! (We shall show that this is statically checkable.) Consider what should do instantaneous instructions such as

```
X:=0; loop X:=X+1 end
loop emit s(X) end
```

### 2.4.8. The conditional instruction.

The conditional is also the usual one :

if *boolexp* then $i_1$ else $i_2$ fi

The evaluation of the boolean expression *boolexp* is assumed to be instantaneous, so that either $i_1$ or $i_2$ receives the control instantaneously.

### 2.4.9. The parallel instruction.

It is written as follows :

$i_1 \parallel i_2$

The control is passed instantaneously to $i_1$ and $i_2$. The parallel construct terminates exactly when both $i_1$ and $i_2$ have terminated. Since the signals are broadcasted, $i_1$ receives all the signals emitted by itself and by $i_2$, and conversely. Hence $i_1$ and $i_2$ work in the same event environment. We shall freely

use the "||" operator as a n-ary one, assuming associativity to the left, and use often brackets for the parallel constructs in order to improve readability.

### 2.4.10. tag and exit.

The tag-exit mechanism is a central one. It is to comparable to the catch-throw mechanism in LISP or the failure mechanism in ML [16]. The syntax is as follows for the two instructions tag and exit :

```
tag T in i end
exit T
```

where an exit can only appear within a tag of the same name (the usual scope rules apply). Again the transmission of control is instantaneous, so that an exit provokes instantaneously the termination of its tag. Of course the exit instruction itself does *not* pass the control in sequence. A tag also terminates when its body terminates. Exits are used for exiting loops, upto or uptonext instructions. They will be of great use for deriving new convenient construct from the kernel ones.

Since we have also a parallel construct, we must be careful to understand how an exit works inside a parallel branch. Here are to examples :

```
tag T in
   [
      emit s1; exit T; emit s2
   ||
      emit s3
   ]
end
```

Here s1 is obviously emitted while s2 is obviously not emitted, and s3 is also emitted : the exits in a parallel branch are treated only "at the end" of the treatment of the whole parallel construct in the current instant.

```
tag T1 in
   tag T2 in
      exit T1 || exit T2
   end;
   emit s
end
```

We have two simultaneous exits. Then the outermost one has precedence, so that the whole program terminates instantaneously, s being not emitted.

### 2.4.11. Local bindings.

If *decl* is a variable declaration, then

*decl* in *i* end

is an instruction. The variable bound is local to the instruction, with scope rules as usual. Variables may not be shared. More precisely, a variable which is assigned (by an assignment instruction, an upto or an uptonext instruction) within one branch of a parallel construct must be local to that branch; this is not necessary for a variable which is just read. The following situation is not allowed :

```
var X: int in
   [
      X:=0
   ||
      emit s(X)
   ]
end
```

But the following situation is allowed:

```
var X: int in
   X:=3;
   [
      emit s(X)
   ||
      var X: int
         do nothing upto s(X);
         X:=X+1;
         emit s1(X)
      end
   ]
end
```

Because of the ordinary scope rules, there are two distinct variables named X.

## 2.5. The ESTEREL programs.

A program is formed of a *sort definition* which defines the signals being input, output, input and output, or local to the program and their types, (keywords "input", "output" or "input output" followed by signal declarations), followed by a closed term (i.e. a term without free variables or free exits). The local events are internal to the program, they will not be emitted to its environment. Example of programs are given in section 4.

The type-checking of expression is classical and omitted here. We shall always assume that programs are well-typed. We shall also assume that a

variable is always initialized before being used (this is very easy to check).

## 3. The extended language.

The kernel language presented in the previous section is not really a language we would like to program in, but rather a set of programming *primitives*. We now present *derived constructs* making life easier. These constructs are just *macros*, so that there will be no need to define their semantics. They make extensive use of the tag-exit mechanism, and work correctly only under the strong synchrony hypothesis. We hope that the way we define extensions will train the reader to the quite unusual style of ESTEREL programs.

The extended ESTEREL language being still in development, several aspects will not be treated completely (in particular modularity).

### 3.1. Some trivial extensions.

We give the possibility of assigning a value to a variable when it is declared :

```
var X:=exp : type in i end
```

stands for

```
var X : type in X:=exp; i end
```

And we give also the possibility of defining constants :

```
const TWO=2 in i end
```

We finally give the possibility of not binding a variable by an upto or uptonext instruction :

```
do i upto(next) s
```

stands for

```
var X : type in
    do i upto(next) s(X) end
end
```

where *type* is the type of s and X is a new variable not contained in $i$.

### 3.2. Pure and single signals.

General signals being a bit heavy to handle, we introduce two special cases as follows:

```
pure signal s
single signal s(type)
```

Pure signals convey no value, while single signals are restricted to have at most one emitter at a time; hence their receivers will receive single values, and one may forget about the * operation. The reader should be convinced that the pure and single signal constructs represent only convenient syntactic sugar : pure signals were shown to be a general case of general signals in section 2, and for single signals the operation * needs not even to be defined since the fact that there is no more than one emitter at a time will be shown to be statically checkable in section 5.

### 3.3. Local signals.

In the kernel language, we did not give the full power of local declarations for signals as we did for variables, since we allowed local signals to be declared only at the level of a program (the effect being that the signal will not be output to the outside). The reason is that we would have to manipulate a much heavier formalism for the static and dynamic semantics given in the present paper, since we shall arrange the signals of a program into a graph.

A way to introduce local signal declarations at any point in the program is to rename the signal with a new name and to push its declaration at the top of the program; this would not be feasible in a natural way for variables, because we must guarantee that variables are not shared, that constraint making no sense for signals. We shall therefore use freely the following syntax for declaring local signals :

```
signal s(type) in
    i
end
```

### 3.4. Waiting for signals.

The following extended statements are obvious:

await s(X) ≡ do nothing upto s(X)
awaitnext s(X) ≡ do nothing uptonext s(X)

There is a sharp difference between await and awaitnext : two await statements in a row behave just as one, since the signal is already present when the second wait statement is activated: Hence "await s; await s" is equivalent to "await s". Consider also the two programs :

emit s(5); await s(X)
emit s(5); awaitnext s(X)

The first program immediately terminates with X bound to 5. The second program does not terminate, since the signal s emitted by the emit statement is ignored by the awaitnext statement. One has always to remember that an await statement may take no time, while an awaitnext statement always takes time.

As a side remark, we notice that an await statement may be used for exporting the value of a variable which is local to a branch of a parallel construct to the end of that construct :

```
single signal s(int) in
   tag T in
      [
        ...
      ||
         var X : int in
            ...
            emit s(X);
            exit T
         end
      ]
   end;
   await s(Y)
end
```

Since the emission of s(X) is synchronous with the "exit T" statement, the "await s(Y)" statement terminates immediately and binds Y to the last value of the local variable X.

### 3.5. Temporal loops.

We introduce several forms of temporal loops :

```
do i uptoeach s(X)
```

stands for

```
do i upto s(X);
loop
  do i uptonext s(X)
end
```

similarly

```
do i uptoeachnext s(X)
```

stands for

```
loop
  do i uptonext s(X)
end
```

while

```
every s(X) do i end
```

stands for

```
await s(X);
loop
   do i uptonext s(X)
end
```

and also

```
everynext s(X) do i end
```

stands for

```
awaitnext s(X);
loop
   do i uptonext s(X)
end
```

To understand why the signal and its variable appear last in the first cases and first in the last cases, just look at when $i$ starts and at which value X is bound.

### 3.6. Testing for the presence or absence of a signal.

This construct really shows the power of our primitives. Assume that we want to do $i_1$ only if the signal s is present in the current instant. The instruction is

```
inpresence s(X) do i end
```

Its implementation is

```
tag T in
   do exit T upto s(X);
   i
end
```

where T is not free in $i$. It works as follows: if s is present, then the upto terminates immediately without executing its body "exit T", and $i$ starts immediately with X bound to the value of s in the current instant. The whole construct then terminates when $i$ terminates. Otherwise the exit statement is indeed executed, and the whole construct immediately terminates. The strong synchrony hypothesis is absolutely essential here. Conversely, one can test for the absence of a signal :

```
inabsence s do i end
```

stands for

```
tag T1 in
   tag T2 in
      do exit T2 upto s;
      exit T1
   end;
   i
end
```

where T1 is not free in $i$. We need two exits: T1 is exited if s is present, which provokes the termination of the whole construct; T2 is exited if s is absent and this provokes the execution of $i$.

### 3.7. Watchdogs.

A very common problem in real-time programming is to check whether a task has been completed within a fixed amount of time, to terminate normally in that case, and to provoke some exception treatment in the abnormal case. This may be programmed as follows :

```
do i watching s(X) abnormal i' end
```

standing for

```
tag T in
   do
      i; exit T
   upto s(X);
   i'
end
```

### 3.8. Failures

The tag-exit mechanism is obviously close to a classical failure mechanism. However we have given no way of telling why an exit was raised, i.e. of naming exceptions. This is very easily done by using local signals. Here is a syntax for failure handling :

```
trapfailure
      i
   failure F1 do i1
   failure F2 do i2
   ...
   failure Fn do in
end
```

where $i$ may contain statements "failwith Fi". It translates into

```
pure signal F1, F2, ... , Fn in
   tag OK in
      tag FAIL in
         i;
         exit OK
      end;
      [
         inpresence F1 do i1 end
      ||
         inpresence F2 do i2 end
      ||
         ...
      ||
         inpresence Fn do in end
      ]
   end
end
```

all the "failwith Fi" statements in $i$ being replaced by "emit Fi; exit FAIL". Notice that several failures can be handled simultaneously.

### 3.9. Operations on signals.

#### 3.9.1. Division of signals.

For the moment we are able only to reference the current or the next occurrence of a signal. But we should also be able to reference any occurrence in the future. Hence we extend the upto and uptonext statements in the following way, s having type $t$ and *exp* type int :

do $i$ uptonext *exp* s(X)

stands for (the value of *exp* being supposed greater than 1)

```
pure signal f, var COUNT:=0 in
   [
      tag T in
         var Y:t in
            everynext s(Y) do
               COUNT:=COUNT+1;
               if COUNT=exp then emit f; exit T fi
            end
         end;
   ||
      do i upto f
   ];
   await s(X)
end
```

The local signal f is sent for killing $i$ at the right instant. Because of the strong synchrony hypothesis, one is sure that f is synchronous with s, and that the last "await s(X)" will indeed terminate instantaneously and bind X to the right value.

Now we can define a clock division, producing for example a clock beating the second from a clock beating the millisecond, and being sure that a second is indeed *synchronous* with one millisecond over 1000 :

```
loop
   awaitnext 1000 MS;
   emit S
end
```

And one sees at once how to program arbitrary long delays (actually the meaning of the word "delay" is not quite clear in a synchronous framework : when does a delay start, right now or on the next signal?).

### 3.9.2. Conjunction or disjunction of pure signals.

It is quite common to wait for some instantaneous conjunction or disjunction of signals. This naturally leads to the introduction of two constructs

```
s1 and s2
s1 or s2
```

We introduce these constructs only for pure signals here, The program

```
do i upto(next) s1 and s2
```

translates as

```
pure signal s1-and-s2 in
   [
      every s1 do
         inpresence s2 do emit s1-and-s2 end
      end
   ||
      do i upto(next) s1-and-s2
   ]
end
```

which terminates when s1 and s2 occur simultaneously, and similarly the program

```
do i upto(next) s1 or s2
```

expands into

```
pure signal s1-or-s2 in
   [
      tag OK in
         [
            await(next) s1; exit OK
         ||
            await(next) s2; exit OK
         ]
      end;
      emit s1-or-s2
   ||
      do i upto(next) s1-or-s2
   ]
end
```

which terminates at the first occurrence of s1 or s2.

### 3.9.3. Absolute time.

It is often useful to count signals and define the "absolute time" associated with a signal, for writing instructions such as

```
at 1000 s do i end
```

For this we introduce an absolute time counter in parallel with the whole program. It uses a variable s-COUNT declared globally and initialized to 0. The value of that variable may then be consulted.

```
every s do
   s-COUNT:=s-COUNT+1;
   every s-TIME do
      emit s-TIME-IS(s-COUNT)
   end
end
```

The above "at" instruction translates as

```
var TIME: int in
   tag END in
      every s do
         emit s-TIME;
         await s-TIME-IS(TIME);
         if TIME=1000 then
            i; exit END
         fi
      end
   end
end
```

and the absolute time may indeed be consulted at any instant by

```
emit s-TIME;
await s-TIME-IS(s-count)
```

The syntax for this kind of extended construct has clearly to be polished further on. Notice however that we used two synchronous messages s-TIME and s-TIME-IS to model in fact an asynchronous communication.

### 3.10. Modules.

It is of course necessary to have nice module structures in really useful programming languages. The modular structure we propose for ESTEREL is quite simple. A *module* has formal parameters, which may be constants, variables or signals. The syntax of a module declaration is

```
module name(par1, ... , parn) =
   i
in
   i'
end
```

A parameter declaration may be a classical variable declaration, a constant declaration, or a signal declaration with input/output indication, as in the program sort definition :

```
module FOO (var X:int list, const TWO=2, input output pure signal s) =
    ...
end
```

A module invocation has the form

$$name(actual_1, \ldots, actual_n)$$

and it works just as a textual copy of the body of the module with syntactic replacement of the formal parameters by the actual ones (the types must match, of course). Hence module invocation may not be recursive, and modules add nothing to the power of the language. They will not be considered in the formal semantics. Of course there are some problems with polymorphic modules etc., but these problems are in no way specific to ESTEREL and they will not be treated here.

## 4. Some programming examples.

In the examples we make free use of the extended constructs presented in the last section.

### 4.1. A reflex game.

We want to program a reflex game, working as follows : a game starts by pressing a RESET button, and it will be composed of 10 reflex measures. Each measure starts when the player presses a button A; then after a random time a green lamp lights on, and the player must press as fast as possible a button B. Then the green lamp is turned off and the reflex time is displayed. A new measure starts when the player presses A again. When the cycle of 10 measures is completed, the average reflex time is displayed after a pause of 3 seconds. There are many exception cases; some of them are simple mistakes and make a bell ring. Some other represent true cheating tentatives or abandons of the game, they turn on a red light and stop completely the game, which waits for a new RESET :

- The player presses B instead of A to start a measure. The bell rings.
- The player presses A during a measure. The bell rings.
- The player presses B too early during a measure, i.e. before the green lamp turns on or just at the same time it turns on. Then the red light turns on and the game is ended (the player cheats!).
- The player does not press A or B within 10 seconds when it is supposed to do so. Then the machines stops the game and turns the red light on (case of abandon).

A last rule is that a new game is started from fresh if the player presses RESET at any time.

We use an external timer which receives a signal START–TIMER and sends back a signal TIME after a random (absolute) time.

```
input pure signal RESET, A, B, MS, TIME
output pure signal GREEN-OFF, GREEN-ON, RED-OFF, RED-ON,
                   RING-BELL, START-TIMER
output single signal DISPLAY(int) in
   every RESET do
      emit RED-OFF;
      trapfailure
         var AVERAGE:=0 : INT in
            % measure loop
            repeat 10 times
               % waiting for A
               do
                  every B do emit RING-BELL upto A
               watching 10000 MS abnormal failwith END-GAME end;
               % delay and waiting for B; A rings the bell
               [
                  every A do emit RING-BELL end
               ||
                  %random delay
                  do
                     emit START-TIMER;
                     await TIME
                  watching B abnormal failwith END-GAME end;
                  emit GREEN-ON;
                  % waiting for B and displaying the result.
                  var TIME:= 0 : int in
                     do
                        do
                           every MS do TIME:=TIME+1 end
                        upto B
                     watching 10000 MS abnormal failwith END-GAME end;
                     emit GREEN-OFF;
                     emit DISPLAY(TIME);
                     AVERAGE:=AVERAGE+TIME
                  end
               ]
            end;
            % final display of the average time
            await 3000 MS;
            emit DISPLAY (AVERAGE/10)
         end
      failure END-GAME do emit RED-ON end
   end
end
```

**4.2. A prompting machine.**

This quite different example is actually used in the user interface of the prototype ESTEREL compiler. The problem is to send prompts to a terminal at the right moment. The machine is interfaced with a lexical analyzer and a reader; the lexical analyzer sends signals PEEK–CHAR and READ–CHAR to the reader, which gives back a character by sending a signal CHAR(char), after a while (i.e. not synchronously). Our machine uses the fact that signals are broadcasted, and listens to the discussion between the analyzer and the reader.

The prompting mechanism may be armed by a signal PROMPT–ON for reading from a terminal, and disarmed by a signal PROMPT–OFF, for reading from a file. In addition one may emit two prompts PS1 and PS2, as in UNIX: PS1 is emitted at the first line, PS2 is emitted at the following lines, until a signal BACK–TO–PS1 is received (from some other machine driving the lexical analyzer). A prompt should be sent when the program is started and at each PEEK–CHAR or READ–CHAR request following the reading of a carriage return (a carriage return is read each time the reader answers a request READ–CHAR by an answer CHAR(CR), where CR is the appropriate character value). The program is made of three parallel submachines exchanging local signals; the first machine detects when a carriage-return is read, the second machine decides when a prompt should be emitted, the third machine decides which prompt should be emitted.

```
input pure signal PROMPTON, PROMPTOFF, BACK-TO-PS1,
                  READ-CHAR, PEEK-CHAR,
output pure signal PS1, PS2 in
   pure signal CR-READ, SEND-PROMPT in
      [
         % detects when a READ-CHAR is followed by a CHAR(CR)
         var X: char in
            every READ-CHAR do
               await CHAR(X);
               if X=CR then emit CR-READ fi
            end
         end
      ||
         % decides when to emit a prompt
         emit SEND-PROMPT;
         loop
            do % upto PROMPTOFF
               every CR-READ do
                  awaitnext READCN or PEEKCN;
                  emit SEND-PROMPT
               end
            upto PROMPTOFF;
            awaitnext PROMPTON
         end
      ||
         % decides to emit PS1 or PS2
         do
            await SEND-PROMPT; emit PS1;
            everynext SEND-PROMPT do emit PS2 end
         uptoeach BACK-TO-PS1
      ]
   end
end
```

## 5. Signal dependency and the static semantics.

### 5.1. Signal dependency.

Let us come back on the naive semantics of the upto instruction :

```
do i upto s(X)
```

One terminates instantaneously or one executes $i$ according to the presence or absence of the signal s in the current instant. Hence one must know whether s is present or not *before* executing the upto statement; moreover if s is indeed present, one must know which values were emitted in order to bind properly X. This forbids writing instructions such as

```
do emit s upto s
```

where the signal s should be emitted if not present and not emitted if present, which is clearly a nonsense, or instructions such as

```
    emit s([0])
||
    var X: int list in
        await s(X); emit s(X+1)
    end
```

where the signal s is assumed to have type int list with * operation "sortappend", see section 2, the "+1" operation being assumed to distribute over lists : the "emit s(X+1)" statement if synchronous with the "emit s([0])" statement, according to the strong synchrony hypothesis. It is then impossible to bind sensibly X: if one binds X to the list [0], then one should also bind X to the list [0,1], hence to the list [0,1,2] and so on,

The problem is comparable to what happens in electronics, where it is also not really advisable to plug the output of an amplifier directly into its input line. We want our flow of control to keep a precise "past-to-future" direction even inside an instant, so that we should not be allowed to emit a signal if we have already received it or if we have decided that we shall not receive it. We may have similar short-circuit situations involving more than one signal :

```
    var X1: int list in
        await s1(X1); emit s2(X1+1)
    end
||
    var X2: int list in
        await s2(X2); emit s1(X2+1)
    end
||
    emit s1([0])
||
    emit s2([0])
```

and also

```
    do emit s2 upto s1
||
    await s1; emit s2
```

To avoid short circuits of this kind, we introduce a notion of *dependency* between signals. We say that s2 depends on s1 in two cases

(i) An "emit s2" instruction appearing inside the body of a "do *i* upto s1" statement may be executed in the current instant. In that case we say that s1 *inhibits* s2.

(ii) An "emit s2" statement follows in the same instant the termination of a "do *i* upto s1" statement (which of course has terminated because of the presence of s1 in the current instant). We then say that s1 *causes* s2 (notice that the emission of s2 is synchronous with the reception of s1. For example, when one constructs a clock beating the second by dividing a clock beating the millisecond, the second is caused by the millisecond).

In the dependency analysis, inhibition and causality behave exactly in the same way. Our requirement will be as follows:

*The dependency relation between signals must have no cycle*

In the first examples, the dependency graph has a cycle containing only s (s inhibits itself in the first case, s causes itself in the second case). In the two last examples, there are cycles through s1 and s2 (s1 causes s2 and s2 causes s1 in the first case, s1 inhibits s2 and s2 causes s1 in the second case).

Short-circuits problems appear only "inside an instant". There is no problem with an instruction like

```
await s1; awaitnext s2; emit s1
```

Here we are certain that the "await s1" and "emit s1" instructions will never appear in the same instant.

**5.2. The formalism used for the static semantics.**

We now construct the dependency graph on signals in a formal way : this is the main purpose of the *static semantics*, which gives an analysis of all possible behaviors of a program *in the first instant of any of its executions.* The complete causality analysis of a program in all possible input histories will require to use also the dynamic semantics presented in the next sections. We shall not require the dependency graph to be declared by the programmer, and not even to be constant for a given program (i.e. the graphs corresponding to two different instants need not to have any kind of relation with eachother). In practical systems, we shall however allow the user to specify himself dependency relations which must hold at every instant (e.g. that the second is always caused by the millisecond).

As all our semantics, we shall present our static semantics by a set of *structural conditional rewrite rules* à la Plotkin, defining a transition system with transitions of the form

$$<i, D> \xrightarrow{t, L} <C, D'>$$

where :

(i) $i$ is an ESTEREL instruction.

(ii) $D$ is a set of signals, on which the execution of $i$ will depend. That is $D$ is the set of signals of which we must know the status before being able to execute $i$.

(iii) The termination status $t$ is a boolean indicating whether $i$ may terminate and pass the control in sequence (values *tt* or *ff*). The boolean conjunction and disjunction are written $\cap$ and $\cup$.

(iv) $L$ is a set of tag labels (free in the current instruction), which represents the set of exits that $i$ may execute in the first instant.

(v) $C$ is the dependency graph, represented as a set of pairs (s1,s2) where s2 depends on s1.

(vi) $D'$ is the set of signals on which the execution will depend after executing $i$. The set $D'$ will always contain the set $D$.

The dependency graph of a program P with body $i$ (removing the event declarations which are irrelevant here) will be obtained by computing a transition of the form

$$<i, \phi> \xrightarrow{t, L} <C, D'>$$

The static semantics will have another role: testing that the body of a loop statement cannot terminate instantaneously. The static semantics will then be undefined (and the rules will show how to output an appropriate error message, not done here).

## 5.3. The rules of the static semantics.

### 5.3.1. Axiom of nothing.

A nothing statement terminates instantaneously and does just that :

$$\text{<nothing, } D\text{>} \xrightarrow{\text{tt}, \phi} \text{<}\phi, D\text{>}$$

### 5.3.2. Axiom of assignment.

As far as signal dependencies are concerned, an assignment behaves just like nothing :

$$\text{<X:=}exp\text{ , } D\text{>} \xrightarrow{\text{tt}, \phi} \text{<}\phi, D\text{>}$$

### 5.3.3. Axiom of emission.

We record in the graph all generated dependencies (this is the only rule which increments $G$) :

$$\text{<emit s(}exp\text{), } D\text{>} \xrightarrow{\text{tt}, \phi} \text{<\{(d,s), d}\in D\text{\}, } D\text{>}$$

### 5.3.4. Rule of upto.

The body $i$ of an upto statement is analyzed by adding the signal s to the given dependency set $D$, since all emissions it may contain are potentially inhibited by s. This yields as result a graph G and a new dependency set $D'$. One outputs the facts that an upto statement may always terminate (if the signal is present), that it exits the same tags as its body, that it gives graph G and dependency set $D'$ :

$$\frac{\text{<}i, D \cup \{s\}\text{>} \xrightarrow{t, L} \text{<}G, D'\text{>}}{\text{<do } i \text{ upto } s(X), D\text{>} \xrightarrow{\text{tt}, L} \text{<}G, D'\text{>}}$$

Notice that $D'$ will always contain $D$ and s, so that the dependency on s is transmitted to the instructions which follow in sequence; the rule for upto is the only one to increment $D$.

### 5.3.5. Rule of uptonext.

An uptonext does not pass the control in sequence. Besides that, it behaves just as its body.

$$\frac{\text{<}i, D\text{>} \xrightarrow{t, L} \text{<}G, D'\text{>}}{\text{<do } i \text{ uptonext } s(X), D\text{>} \xrightarrow{\text{ff}, L} \text{<}G, D'\text{>}}$$

**5.3.6. Rules of sequence.**

There are two cases, according to the fact that the first instruction may pass the control or not. In the first case one analyzes the second instruction with the dependencies generated by the first one, and one outputs as termination status the status then obtained, as tag set the union of the two tag sets, as graph the union of the two generated graphs, as new dependencies those generated by the second instruction, which cumulate those generated by the first one :

$$\frac{\langle i_1, D\rangle \xrightarrow{t_1, L_1} \langle G_1, D'\rangle \quad \langle i_2, D'\rangle \xrightarrow{t_2, L_2} \langle G_2, D''\rangle}{\langle i_1; i_2, D\rangle \xrightarrow{t_2, L_1 \cup L_2} \langle G_1 \cup G_2, D''\rangle}$$

In the second case one outputs simply what was generated by the analysis of the first instruction :

$$\frac{\langle i_1, D\rangle \xrightarrow{\not t, L_1} \langle G_1, D'\rangle}{\langle i_1; i_2, D\rangle \xrightarrow{\not t, L_1} \langle G_1, D'\rangle}$$

**5.3.7. Rule of loop.**

We give a rule only if the body of the loop may not terminate instantaneously. The loop statement then behaves as its body :

$$\frac{\langle i, D\rangle \xrightarrow{\not t, L} \langle G, D'\rangle}{\langle \text{loop } i \text{ end}, D\rangle \xrightarrow{\not t, L} \langle G, D'\rangle}$$

**5.3.8. Rule of conditional.**

To be really accurate, the conditional cannot be treated in a simple way. One should study separatedly all possible control paths it generates. However we shall make some approximation here in order to simplify the presentation: we shall union the graphs generated by the two arms of the conditional. This makes incorrect a program such as

```
if boolexp then
     do emit s1 upto s2
else
     do emit s2 upto s1
fi
```

which could perfectly be considered as correct, since there is obviously no communication between the two branches of a conditional. We shall say that a conditional may terminate iff one of its arms may, and output the union of the exits, graphs and generated dependencies :

$$\frac{<i_1, D> \xrightarrow{t_1, L_1} <G_1, D'_1> \quad <i_2, D> \xrightarrow{t_2, L_2} <G_2, D'_2>}{<\text{if } boolexp \text{ then } i_1 \text{ else } i_2 \text{ fi}, D> \xrightarrow{t_1 \cup t_2, L_1 \cup L_2} <G_1 \cup G_2, D'_1 \cup D'_2>}$$

**5.3.9. Rule of parallel.**

A parallel statement may terminate iff both its arms may. The tags, graphs, and resulting dependencies are obtained by unioning those of the arms.

$$\frac{<i_1, D> \xrightarrow{t_1, L_1} <G_1, D'_1> \quad <i_2, D> \xrightarrow{t_2, L_2} <G_2, D'_2>}{<i_1 \parallel i_2, D> \xrightarrow{t_1 \cap t_2, L_1 \cup L_2} <G_1 \cup G_2, D'_1 \cup D'_2>}$$

**5.3.10. Rule of tag.**

A tag may terminate iff its body may terminate or generate a corresponding exit. The other exit labels generated by the body are output, the graph and dependency output are those of the body :

$$\frac{<i, D> \xrightarrow{t, L} <G, D'>}{<\text{tag T in } i \text{ end}, D> \xrightarrow{t \cup (T \in L), L - \{T\}} <G, D'>}$$

**5.3.11. Axiom of exit.**

An exit does not pass the control, and generate as exit set $L$ the corresponding singleton :

$$<\text{exit T}, D> \xrightarrow{ff, \{T\}} <\phi, D>$$

**5.3.12. Rules of binding.**

A variable binding has no effect w.r.t. the static semantics :

$$\frac{<i, D> \xrightarrow{t, L} <G, D'>}{<\text{var X in } i \text{ end}, D> \xrightarrow{t, L} <G, D'>}$$

**5.4. Static correctness.**

**Definition:** We say that a program P is *statically correct (w.r.t. all possible first instants)* if and only if there exists a provable transition of the form

$$<P, \phi> \xrightarrow{t, L} <G, D'>$$

with G acyclic.

From the form of our rules, one sees easily that the a proof can be

computed in a pure top-down way, with at most one rule to apply at any subterm. Therefore there is at most one possible transition from <P,$\phi$>. There is no transition if and only if the body of a loop statement may terminate instantaneously. In fact we could have defined the static semantics as a partial function from terms and dependency sets to termination status, tag label set, graph and new dependency set. But we think that the rewrite rule formalism is easier to read anyway.

Our rules are in fact a bit too strong, and we reject also the following program which could be considered as correct:

```
tag T1 in
    tag T2 in
        exit T1 || exit T2 || do emit s1 upto s2
    end;
    do emit s2 upto s1
end
```

Here the graph is {(s1,s2),(s2,s1)} and has a cycle. But the "do emit s2 upto s1" instruction will never be executed because of the presence of "exit T1". There is however not too much trouble to reject programs in which some instruction will never be activated.

Notice finally that we are able to count how many "emit s" instructions may receive the control in the first instant. Hence it is possible to check that a signal declared to be single has indeed at most one emitter.

### 5.5. An example.

Here is an example of a formal derivation, presented in a bottom-up way. Consider the following program :

```
input pure signal s1,
output pure signal s2, s3,
local pure signal s in
   [
      do nothing upto s1;
      emit s
   ||
      tag T in
         [
            do
               do emit s3 upto s;
               emit s2
            uptonext s3;
         ||
            exit T
         ]
      end
   ]
end
```

We shall give names to the transitions and indicate for any transition the name of its premisses, replacing the horizontal bar we used in rules by a turnstile $\vdash$. Here is a derivation :

a) First branch of the outermost || :

$\vdash T_1 \equiv$ <nothing, {s1}> $\xrightarrow{\#,\phi}$ <$\phi$, {s1}>

$T_1 \vdash T_2 \equiv$ <do nothing upto s1, $\phi$> $\xrightarrow{\#,\phi}$ <$\phi$, {s1}>

$\vdash T_3 \equiv$ <emit s, {s1}> $\xrightarrow{\#,\phi}$ <{(s1,s)}, {s1}>

$T_2, T_3 \vdash T_4 \equiv$ <do nothing upto s 1;emit s, $\phi$> $\xrightarrow{\#,\phi}$ <{(s 1,s)}, {s1}>

b) Second branch of the outermost || :

b.1) First branch of the innermost || :

$\vdash T_5 \equiv$ <emit s3, {s}> $\xrightarrow{\#,\phi}$ <{(s,s3)}, {s}>

$T_5 \vdash T_6 \equiv$ <do emit s3 upto s, $\phi$> $\xrightarrow{\#,\phi}$ <{(s,s3)}, {s}>

$\vdash T_7 \equiv$ <emit s2, {s}> $\xrightarrow{\#,\phi}$ <{(s,s2)}, {s}>

$T_6, T_7 \vdash T_8 \equiv$ <do emit s3 upto s;emit s2, $\phi$> $\xrightarrow{\#,\phi}$ <{(s,s3),(s,s2)}, {s}>

$T_8 \vdash T_9 \equiv$ <do $\cdots$ uptonext s3, $\phi$> $\xrightarrow{\#,\phi}$ <{(s,s3),(s,s2)}, {s}>

b.2) Second branch of the innermost || :

$$\vdash\ T_{10} \equiv \langle \text{exit T}, \phi \rangle \xrightarrow{\mathit{ff},\{T\}} \langle \phi, \phi \rangle$$

c) The innermost || and its enclosing tag :

$$T_9, T_{10} \vdash\ T_{11} \equiv \langle [\ \text{do} \cdots \text{uptonext s3} \parallel \text{exit T}\ ], \phi \rangle \xrightarrow{\mathit{ff},\{T\}} \langle \{(s,s3),(s,s2)\}, \{s\} \rangle$$

$$T_{11} \vdash\ T_{12} \equiv \langle \text{tag T in} \cdots \text{end}, \phi \rangle \xrightarrow{\mathit{tt},\phi} \langle \{(s,s3),(s,s2)\}, \{s\} \rangle$$

d) The body of the program :

$$T_4, T_{12} \vdash\ T_{13} \equiv \langle [\ \cdots \parallel \text{tag T in} \cdots \text{end}\ ], \phi \rangle \xrightarrow{\mathit{tt},\phi} \langle \{(s1,s),(s,s3),(s,s2)\}, \{s1,s\} \rangle$$

The resulting graph is {(s1,s),(s,s3),(s,s2)}, and the program is statically correct since it has no cycle.

## 6. The behavioral semantics.

### 6.1. The formalism of the behavioral semantics.

The behavioral semantics defines the history transformation associated with a program. The key idea is to treat each event separatedly: let P be a program, let $E$ be some input event for P, which defines the signals coming from the outside world and the values they convey; the execution of P on $E$ will provoke the emission of signals forming an output event $F$ (intented to be used by the outside world) and yield *another program* $P_1$ which represents the program to be executed at the next instant. This is written

$$P \overset{F}{\underset{E}{\Longrightarrow}} P_1$$

Then it is extremely easy to define the behavior of a program P on a given history $H = E_1, E_2, \cdots E_n, \ldots$, considering the sequence

$$P \overset{F_1}{\underset{E_1}{\Longrightarrow}} P_1 \overset{F_2}{\underset{E_2}{\Longrightarrow}} P_2 \Longrightarrow \cdots \overset{F_n}{\underset{E_n}{\Longrightarrow}} P_n \Longrightarrow \cdots$$

The $F_n$ are the events output from input events $E_n$.

For example consider the following program P, which "shifts" a signal s1 : a signal s2 is output synchronously with the reception of any occurrence of s1 but the first one, and the value output with s2 is the previous value input with s1 :

```
input single signal s1(int),
output single signal s2(int) in
    var X,Y : int in
        await s1(X);
        everynext s1(Y) do
            emit s2(X);
            X:=Y
        end
    end
end
```

Assume the first input event is $s1^3$. Then one has a transition

$$P \xrightarrow[s1^3]{1} P_1$$

with $P_1$ as follows :

```
input single signal s1(int),
output single signal s2(int) in
    var X:=3, Y : int in
        every s1(Y) do
            emit s2(X);
            X:=Y
        end
    end
end
```

If the second input event is $s1^5$, then one has a transition :

$$P_1 \xrightarrow[s1^5]{s2^3} P_2$$

with P2 as follows :

```
input single signal s1(int),
output single signal s2(int) in
    var X:=5, Y:=5 : int in
        every s1(Y) do
            emit s2(X);
            X:=Y
        end
    end
end
```

Notice that in this setting the value taken by the variables are recorded in the program themselves. We shall also use a different setting and work with a global

memory $\sigma$ which associates values with variables, having then transitions of the form

$$<P_1, \sigma_1> \overset{\substack{F_1 \\ \Longrightarrow \\ E_1}}{} <P_2, \sigma_2> \Longrightarrow \cdots$$

In this new setting the program $P_1$ above must be replaced by the pair $<P'_1, \sigma'_1>$ where $P'_1$ is the program

```
input single signal s1(int),
output single signal s2(int) in
   var X, Y : int in
      every s1(Y) do
         emit s2(X);
         X:=Y
      end
   end
end
```

and where $\sigma'_1$ associates the value 3 with X. Both techniques have their advantages and their drawbacks : the first technique is more intrinsic and gives a simpler semantics. The second technique is more classical and will be used anyway when we shall compile programs in section 8. Our formalism will actually treat equally well the two possibilities, with only two rules changing from the first one to the second one : the rule of variable declaration, the rule of the parallel construct.

Now we have to describe how an input event transforms a program into another program. This will be done as usual by a set of structural rewrite rules, which will define transitions of the form

$$<i, \sigma, E> \xrightarrow{t,L} <i', \sigma', E'>$$

where

(i) $i$ is the current instruction to treat.

(ii) $\sigma$ is a *memory state*, defining the current association of values with variables.

(iii) $E$ is the input event.

(iv) $t$ is the *termination status* of $i$. It is a boolean set to $tt$ if and only if $i$ terminates and passes the control in the current instant (cf. the static semantics).

(v) $L$ is the set of labels of tags actually exited by $i$ (only the labels which are free in $i$ appear in $L$).

(vi) $i'$ is the *reconfiguration* of $i$. It represents the new instruction to execute at the next instant.

(vii) $\sigma'$ is the memory state after the execution of $i$ in the instant $E$.

(viii) $E'$ is the event output by $i$ (and in particular is the event 1 if $i$ emits no signal). Notice that $E'$ may contain signals which are local to the program, event if $i$ is the body of the program. They will have to be removed in order to get the event $F$ in the $\Longrightarrow$ relation.

The form is more complex than with the $\Longrightarrow$ relation, but this is necessary for the structural induction to work correctly. The position of $E$ and $E'$ is also different this will be justified in the next section.

We shall only treat general signals in the semantics, single and pure signals being just particular cases.

### 6.2. Two techniques for dealing with the memory.

Before giving the rules, we say how to deal with the memory $\sigma$. We shall use two different techniques : the de Bruijn stack technique [10] for the case where we want the relation $\Longrightarrow$ to work with programs only, and the allocation technique, where we want the relation $\Longrightarrow$ to work with program-memory pairs. Both techniques will treat correctly name conflicts, but both will require a preliminary labeling of the occurrences of the variables of a program by integers. In both cases we use the same notations : given a memory $\sigma$ and a (labeled) variable $X^i$, the value of $\sigma$ on $X^i$ is written $\sigma(X^i)$ and the memory obtained by changing the value of $\sigma$ on $X^i$ to $v$ is written $\sigma[X^i \leftarrow v]$.

#### 6.2.1. The stack technique.

The index of an occurrence of a variable represents its *binding depth*, i.e. the number of enclosing variable declarations upto its own declaration. Here is a correct indexing :

```
var X in
    var Y in
        Y¹:=X²;
        var X in
            Y²:=X¹
        end
    end;
    X¹ := 1
end
```

A memory is simply a stack of cells, with usual push and pop operations. A new cell will be pushed at each new "var" declaration (with initial value $\perp$, the undefined value), it will be popped when the corresponding instruction is treated. The cell associated with a variable in a stack is just given by the index of the variable: $X^1$ is the top of the stack, $Y^2$ the cell just before the top and so

on. The name of the variable is irrelevant.

Let us call ε the empty stack. Then we shall produce for a program P and an input instant $E_1$ a transition of the form

$$<P, \varepsilon, E_1> \xrightarrow{t, \phi} <P_1, \varepsilon, E'_1>$$

where the output event $E'_1$ still contains signals which are local to P. Let $F_1$ be obtained from $E'_1$ by removing them. Then the corresponding $\Longrightarrow$ transition will be :

$$P \underset{E_1}{\overset{F_1}{\Longrightarrow}} P_1$$

### 6.2.2. The global allocation technique.

Given the initial program P we want to study, we allocate a cell for every variable of P, taking care of name conflicts so that different X's correspond to different cells. The index is just the "address" of the cell. The above program could be labeled as follows, allocating the three cells in a natural order :

```
var X in
    var Y in
        Y²:=X¹;
        var X in
            Y²:=X³
        end
    end;
    X¹ := 1
end
```

A cell is then read and written in the obvious way, by indexing the global memory at the right position. The relation $\Longrightarrow$ is then obtained as follows : start initially with a memory *undef* which associates undefined values with variables. Produce the transition

$$<P, undef, E_1> \xrightarrow{t, \phi} <P_1, \sigma_1, E'_1>$$

Remove from $E'_1$ the local signals, obtaining $F_1$. Then the relation $\Longrightarrow$ is

$$<P, undef> \underset{E_1}{\overset{F_1}{\Longrightarrow}} <P_1, \sigma_1>$$

and the second step now starts from $<P_1, \sigma_1, E_2>$, producing

$$<P_1, \sigma_1, E_2> \xrightarrow{t_2, \phi} <P_2, \sigma_2, E'_2>$$

Removing the local events from $E'_2$ to get $F_2$ this yields

$$<P_1, \sigma_1> \overset{F_2}{\underset{E_2}{\Longrightarrow}} <P_2, \sigma_2>$$

and so on.

### 6.2.3. Computing expressions.

We assume given a semantics $[\![\ ]\!]$ for simple expressions, which for any expression *exp* and memory state $\sigma$ defines the value $[\![exp]\!]\sigma$ of *exp* in $\sigma$ (see for example [25]). Since the test of proper variable initialization was assumed to be made for all programs, we shall not have to treat the case of expression containing undefined variables.

## 6.3. The rules of the behavioral semantics.

### 6.3.1. Axiom of nothing.

$$<\text{nothing}, \sigma, E> \xrightarrow{\#, \phi} <\text{nothing}, \sigma, 1>$$

### 6.3.2. Axiom of assignment.

An assignment modifies the memory in the usual way :

$$<X^i := exp, \sigma, E> \xrightarrow{\#, \phi} <\text{nothing}, \sigma[X^i \leftarrow [\![exp]\!]\sigma], 1>$$

### 6.3.3. Axiom of emission.

The rule is obvious, using the notation of 2.2:

$$<\text{emit } s(exp), \sigma, E> \xrightarrow{\#, \phi} <\text{nothing}, \sigma, s^{[\![exp]\!]\sigma}>$$

### 6.3.4. Rules of upto.

There are two cases, according to the presence of the signal in the current instant. If the signal is present one binds the variable and terminates :

$$\frac{s \in E,\ E(s)=v}{<\text{do } i \text{ upto } s(X^i), \sigma, E> \xrightarrow{\#, \phi} <\text{nothing}, \sigma[X^i \leftarrow v], 1>}$$

If the signal is not present, we execute $i$ without passing the control. The exits possibly generated by $i$ are transmitted :

$$\frac{s \notin E, \quad <i, \sigma, E> \xrightarrow{t, L} <i', \sigma', E'>}{<\text{do } i \text{ upto } s(X^i), \sigma, E> \xrightarrow{ff, L} <\text{do } i' \text{ upto } s(X^i), \sigma', E'>}$$

### 6.3.5. Rule of uptonext.

An uptonext does not pass the control, and besides that behaves as its body. The reconfiguration becomes an upto statement, according to the naive semantics :

$$\frac{<i, \sigma, E> \xrightarrow{t, L} <i', \sigma', E'>}{<\text{do } i \text{ uptonext } s(X^i), \sigma, E> \xrightarrow{ff, L} <\text{do } i' \text{ upto } s(X^i), \sigma', E'>}$$

### 6.3.6. Rules of sequencing.

The are two cases. In the first one, the first instruction passes the control; this may happen only if it provokes no exit. The second instruction receives the signal emitted by the first one. The event produced is the product of the of events yield by both the two instructions.

$$\frac{<i_1, \sigma, E> \xrightarrow{tt, \phi} <i'_1, \sigma'_1, E'_1> \quad <i_2, \sigma'_1, E * E'_1> \xrightarrow{t_2, L_2} <i'_2, \sigma'_2, E'_2>}{<i_1; i_2, \sigma, E> \xrightarrow{t_2, L_2} <i'_2, \sigma'_2, E * E'_1 * E'_2>}$$

In the second case the first instruction does not pass the control. The rule is obvious :

$$\frac{<i_1, \sigma, E> \xrightarrow{ff, L_1} <i'_1, \sigma'_1, E'_1>}{<i_1; i_2, \sigma, E> \xrightarrow{ff, L_1} <i'_1; i_2, \sigma'_1, E'_1>}$$

### 6.3.7. Rule of loop.

The rule realizes a classical unfolding of the loop. As in the static semantics, it applies only if the body does not pass the control :

$$\frac{<i, \sigma, E> \xrightarrow{ff, L} <i', \sigma', E'>}{<\text{loop } i \text{ end}, \sigma, E> \xrightarrow{ff, L} <i'; \text{loop } i \text{ end}, \sigma', E'>}$$

### 6.3.8. Rules of conditional.

The rule are obvious :

$$\frac{[\![boolexp]\!]\sigma = tt \quad <i_1, \sigma, E> \xrightarrow{t_1, L_1} <i'_1, \sigma'_1, E'_1>}{<\text{if } boolexp \text{ then } i_1 \text{ else } i_2 \text{ end}, \sigma, E> \xrightarrow{t_1, L_1} <i'_1, \sigma'_1, E'_1>}$$

$$\frac{[\![boolexp]\!]\sigma = ff \quad <i_2, \sigma, E> \xrightarrow{t_2, L_2} <i'_2, \sigma'_2, E'_2>}{<\text{if } boolexp \text{ then } i_1 \text{ else } i_2 \text{ end}, \sigma, E> \xrightarrow{t_2, L_2} <i'_2, \sigma'_2, E'_2>}$$

### 6.3.9. Rule of parallel.

#### 6.3.9.1. Rule for the stack technique.

This is the most interesting rule. The main idea is that each component of the parallel construct receives also the signals emitted by the other component :

$$\frac{\langle i_1, \sigma, E * E'_2\rangle \xrightarrow{t_1, L_1} \langle i'_1, \sigma, E'_1\rangle \quad \langle i_2, \sigma, E * E'_1\rangle \xrightarrow{t_2, L_2} \langle i'_2, \sigma, E'_2\rangle}{\langle i_1 \parallel i_2, \sigma, E\rangle \xrightarrow{t_1 \cap t_2,\, L_1 \cup L_2} \langle i'_1 \parallel i'_2, \sigma, E * E'_1 * E'_2\rangle}$$

The termination is the boolean conjunction of the terminations of the components. The exits are obtained by unioning the exits raised by the components. The memory $\sigma$ does not change, since there are no shared variables and since a branch of a parallel statement in not allowed to modify its free variables. For signals, we do not say at all how to find $E'_1$ and $E'_2$, and it is not clear that $E'_1$ and $E'_2$ indeed exist or are unique. Hence our rule is in some sense "non-constructive" Our main result will say that we indeed have existence and uniqueness for statically correct programs. The whole purpose of the computational semantics of the next section is to make the rule constructive.

#### 6.3.9.2. Rule for the global allocation technique.

We have just to remember that the variables that $i_1$ and $i_2$ may modify are disjoint, Let $L(i)$ be the set of local variables to an instruction $i$. For two disjoints sets of variables $V_1$ and $V_2$, and for $\sigma_1,\sigma_2$ two memory states such that $\sigma_1(X^i)=\sigma_2(X^i)$ for $X^i \notin V_1 \cup V_2$ Then let $\sigma_1\, {}_{V_1}{+}_{V_2}\, \sigma_2$ be the memory $\sigma$ such that $\sigma(X^i)=\sigma_1(X^i)$ for $X^i \in V_1$, $\sigma(X^i)=\sigma_2(X^i)$ for $X^i \in V_2$, and $\sigma(X^i)=\sigma_1(X^i)=\sigma_2(X^i)$ for $X^i \notin V_1 \cup V_2$. Then the rule is

$$\frac{\langle i_1, \sigma, E * E'_2\rangle \xrightarrow{t_1, L_1} \langle i'_1, \sigma'_1, E'_1\rangle \quad \langle i_2, \sigma, E * E'_1\rangle \xrightarrow{t_2, L_2} \langle i'_2, \sigma'_2, E'_2\rangle}{\langle i_1 \parallel i_2, \sigma, E\rangle \xrightarrow{t_1 \cap t_2,\, L_1 \cup L_2} \langle i'_1 \parallel i'_2, \sigma'_1\, {}_{L(i_1)}{+}_{L(i_2)}\, \sigma'_2, E * E'_1 * E'_2\rangle}$$

### 6.3.10. Rules of tag.

A tag passes the control if its body does and or if the body executes an exit with the corresponding label and no enclosing exit :

$$\frac{\langle i, \sigma, E\rangle \xrightarrow{t, L} \langle i', \sigma', E'\rangle \quad t = \# \text{ or } L = \{T\}}{\langle \text{tag T in } i \text{ end}, \sigma, E\rangle \xrightarrow{\#, \phi} \langle \text{nothing}, \sigma', E'\rangle}$$

Otherwise, i.e. if the body does not pass the control while raising no exit or if the body exits enclosing tags, the tag instruction does not pass the control and removes its label from the tag label set $L$ (if present) :

$$\frac{<i, \sigma, E> \xrightarrow{\mathit{ff}, L} <i', \sigma', E'> \quad L \neq \{T\}}{<\text{tag T in } i \text{ end}, \sigma, E> \xrightarrow{\mathit{ff}, L-\{T\}} <\text{tag T in } i' \text{ end}, \sigma', E'>}$$

**6.3.11. Axiom of exit.**

An exit does not pass the control and raises the exit :

$$<\text{exit T}, \sigma, E> \xrightarrow{\mathit{ff}, \{T\}} <\text{nothing}, \sigma, 1>$$

**6.3.12. Rule of variable declaration.**

**6.3.12.1. Rule for the stack technique.**

One pushes a new cell (with initial value $\perp$) for analyzing the body. Then one pops the cell and one retains its value as the current value of the variable by an appropriate assignment (types omitted) :

$$\frac{<i, \mathit{push}(\perp, \sigma), E> \xrightarrow{t, L} <i', \sigma', E'>}{<\text{var X in } i \text{ end}, \sigma, E> \xrightarrow{t, L} <\text{var X in } X^1 := \mathit{top}(\sigma'); i', \mathit{pop}(\sigma'), E'>}$$

**6.3.12.2. Rule for the global allocation technique.**

There is just nothing to do :

$$\frac{<i, \sigma, E> \xrightarrow{t, L} <i', \sigma', E'>}{<\text{var X in } i \text{ end}, \sigma, E> \xrightarrow{t, L} <\text{var X in } i' \text{ end}, \sigma', E'>}$$

**6.4. Existence and uniqueness of the behavioral semantics.**

**Theorem**: Let P be statically correct for the first instant. Then the exists one and only one provable transition of the form

$$<P, \sigma, E> \xrightarrow{t, L} <P', \sigma', E'>$$

(where $\sigma = \sigma' = \varepsilon$ in the stack memory management and where $\sigma$ and $\sigma'$ are the memory states before and after execution with global memory allocation).

The proof is rather technical and omitted here, see [15]. The main difficulty is to show that there is always one and exactly one way of applying the rule of $\|$ . The main argument is an induction on the longest path of the graph $G$ produced by the static semantics. Notice that we never chain the arrows $\longrightarrow$ , unlike in [24]. An instant corresponds always to a single transition in our formalism. The flow of control may be quite complex inside an instant, and this is exactly reflected in the complexity of the corresponding transition proof.

Hence we can derive the relation $\Longrightarrow$ as in 6.2, provided that the program generated at each step is statically correct for the first instant. This means that

any computation step involves the computation of the static and dynamic semantics, in that order.

The fact that there is at most one reduction shows that an ESTEREL program is indeed deterministic: it produces exactly one output history per input history.

### 6.5. Examples.

In the examples we shall name the transitions we prove and indicate for any transition the names associated with its premisses, as already done in section 5. We show first why static correction is necessary. Consider the incorrect program P :

```
local pure signal s1, s2
[
    do nothing upto s1; emit s2
||
    do nothing upto s2; emit s1
]
```

Then there are two ways to apply the parallel rule to the body of P (we omit obvious deductions and the memory state which is unnecessary here) :

$T_1 \equiv$ <do nothing upto s1; emit s2, 1> $\xrightarrow{ff.\phi}$ <do nothing upto s1; emit s2, 1>

$T_2 \equiv$ <do nothing upto s2; emit s1, 1> $\xrightarrow{ff.\phi}$ <do nothing upto s2; emit s1, 1>

$T_1, T_2 \vdash$ <[.. || ..], 1> $\xrightarrow{ff.\phi}$ <[.. || ..], 1>

orelse :

$T_1 \equiv$ <do nothing upto s1; emit s2, s1> $\xrightarrow{tt.\phi}$ <nothing, s2>

$T_2 \equiv$ <do nothing upto s2; emit s1, s2> $\xrightarrow{tt.\phi}$ <nothing, s2>

$T_1, T_2 \vdash$ <[.. || ..], 1> $\xrightarrow{tt.\phi}$ <[nothing || nothing], s1 * s2>

Let us now turn to a correct program P :

```
output pure signal s3,
local pure signal s1, s2 in
   [
      [
         emit s1
      ||
         do nothing upto s2;
         emit s3
      ]
   ||
      do nothing upto s1;
      emit s2
   ]
end
```

Here s3 is emitted, but the proof is not immediate : one has to guess that the first branch of the outermost || receives s2, while its second branch receives s1 and s3, and that for the innermost || the first branch receives s2 * s3 while the second branch receives s1 * s3. Here is the derivation, which is unique up to some step permutation (we again omit the memory state) :

$\vdash T_1 \equiv \langle \text{emit s1, s2} * \text{s3} \rangle \xrightarrow{it,\phi} \langle \text{nothing, s1} \rangle$

$\vdash T_2 \equiv \langle \text{do nothing upto s2, s1} * \text{s2} \rangle \xrightarrow{it,\phi} \langle \text{nothing, 1} \rangle$

$\vdash T_3 \equiv \langle \text{emit s3, s1} * \text{s2} \rangle \xrightarrow{it,\phi} \langle \text{nothing, s3} \rangle$

$T_2, T_3 \vdash T_4 \equiv \langle \text{do nothing upto s2; emit s3, s1} * \text{s2} \rangle \xrightarrow{it,\phi} \langle \text{nothing, s3} \rangle$

$T_1, T_4 \vdash T_5 \equiv \langle [\cdots \| \cdots], \text{s2} \rangle \xrightarrow{it,\phi} \langle \text{nothing} \| \text{nothing, s2} \rangle$

$\vdash T_6 \equiv \langle \text{do nothing upto s1, s1} * \text{s3} \rangle \xrightarrow{it,\phi} \langle \text{nothing, } \rangle$

$\vdash T_7 \equiv \langle \text{emit s2, s1} * \text{s3} \rangle \xrightarrow{it,\phi} \langle \text{nothing, s2} \rangle$

$T_6, T_7 \vdash T_8 \equiv \langle \text{do nothing upto s1; emit s2, s1} * \text{s3} \rangle \xrightarrow{it,\phi} \langle \text{nothing, s2} \rangle$

$T_5, T_8 \vdash T_9 \equiv \langle [[\cdots \| \cdots] \| \ldots], 1 \rangle \xrightarrow{it,\phi} \langle [[\text{nothing} \| \text{nothing}] \| \text{nothing }], \text{s1} * \text{s2} * \text{s3} \rangle$

$P \overset{s3}{\underset{1}{\Longrightarrow}} P_1$

Hence we emit indeed s3. The resulting program is clearly equivalent to nothing (two programs being equivalent if they produce the same output history for any input history).[9, 21].

### 6.6. Derived rules for derived constructs.

When we introduced the derived constructs in section 3, we said that they should be considered as syntactic macros having no own semantical meaning. This means in particular that the semantics of the extended language is completely determined by the semantics of the kernel. But in practice for analyzing programs it is very cumbersome to always come down to the kernel level. A much better idea is to use the syntactic definition of a construct to *derive* its behavior. For example it is not hard to derive the following rules for the "inpresence" construct :

$$\frac{s \in E \quad E(s)=l \quad \langle i, \sigma[X^i \leftarrow l], E\rangle \xrightarrow{t,L} \langle i', \sigma', E'\rangle}{\langle \text{inpresence } s(X^i) \text{do } i \text{ end}, \sigma, E\rangle \xrightarrow{t,L} \langle i', \sigma', E'\rangle}$$

$$\frac{s \notin E}{\langle \text{inpresence } s(X^i) \text{do } i \text{ end}, \sigma, E\rangle \xrightarrow{\mathit{tt}, \phi} \langle \text{nothing}, \sigma, 1\rangle}$$

It would often be preferable to give directly a set of rules rather than an implementation for a derived construct. However the problem of knowing whether a construct defined by a rule is definable from our kernel has not yet been investigated, and has no reason to be simple. See [27] for an analysis of that problem for SCCS and MEIJE calculi.

## 7. The computational semantics.

### 7.1. The intuitive idea.

The behavioral semantics is enough for giving the dynamic semantics of any program, but it is not "effective" and therefore does not lead to a natural interpretation mechanism. The computational semantics will do that job. It is a refinment of the behavioral semantics, which uses the information given by the static semantics (and is naturally related to the proof of our main theorem, which we did not give here).

Consider the last program studied in section 6. The causality graph produced by the static semantics is easily seen to be {(s1,s2),(s2,s3)} : s1 causes s2 which in turn causes s3. In the computational semantics we shall use a "wavefront propagation" technique on that graph, resolving first s1, then s2 and finally s3, where by "resolving" a signal s we mean treating the active "upto s" statements. By construction of the graph, we know that there will be no emission of a signal after that signal is resolved, so that we shall correctly decide whether a signal is emitted or not and bind correctly variables to emitted values when the signal is actually emitted. Back to the example, at the first step we freeze all upto statements, so that the only possible instruction to execute is "emit s1". At the second step we can terminate the "do nothing upto s1" statement and hence emit s2. Finally at the third step we can terminate the "do nothing upto s2" statement and emit s3.

### 7.2. The formalism.

The formalism we use is quite close to the one we used for the behavioral semantics. We manipulate transitions of the form :

$$\langle i, \sigma, E \rangle \xrightarrow[G,n]{t,L} \langle i', \sigma', E' \rangle$$

where $i$, $\sigma$, $E$, $i'$, $\sigma'$, $E'$, $L$ are just as before, where $t$ has now three values, $tt$ and $ff$ as before and $\perp$ standing for unresolved, and where there are two additional inputs, the causality graph $G$ and the current propagation level $n$ (notice that all inputs are on the left of the arrow and under it, while all outputs are above the arrow of on its right).

We define the depth $depth(G)$ of an acyclic graph $G$ as being the length of its longest path, and the level $level_G(s)$ of s in $G$ as the longest path from s to a root of $G$.

### 7.3. The rules of the computational semantics.

We consider only the stack technique for memory, the global allocation technique being left to the reader (it works just as before).

#### 7.3.1. Axiom of nothing.

$$\langle \text{nothing}, \sigma, E \rangle \xrightarrow[G,n]{tt,\phi} \langle \text{nothing}, \sigma, 1 \rangle$$

#### 7.3.2. Axiom of assignment.

$$\langle X^i := exp, \sigma, E \rangle \xrightarrow[G,n]{tt,\phi} \langle \text{nothing}, \sigma[X^i \leftarrow [\![exp]\!]\sigma], 1 \rangle$$

#### 7.3.3. Axiom of emission.

$$\langle \text{emit } s(exp), \sigma, E \rangle \xrightarrow[G,n]{tt,\phi} \langle \text{nothing}, \sigma, s^{[\![exp]\!]\sigma} \rangle$$

#### 7.3.4. Rules of upto.

The two first rules apply when the signal can be properly treated. In the first one the signal is present :

$$\frac{level_G(s) \leq n \qquad s \in E \qquad E(s) = v}{\langle \text{do } i \text{ upto } s(X^i), \sigma, E \rangle \xrightarrow[G,n]{tt,\phi} \langle \text{nothing}, \sigma[X^i \leftarrow v], 1 \rangle}$$

In the second one the signal is not present. One has to take care of the fact that the body of the upto may not be resolved yet; this is the reason of the $\cap$ in the resulting termination flag

$$\frac{level_G(s) \leq n \qquad s \notin E \qquad <i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'>}{<\text{do } i \text{ upto } s(X^i), \sigma, E> \xrightarrow[G,n]{\text{if } t \neq tt, L} <\text{do } i' \text{ upto } s(X^i), \sigma', E'>}$$

The last rule applies when the upto cannot be resolved :

$$\frac{level_G(s) > n}{<\text{do } i \text{ upto } s(X^i), \sigma, E> \xrightarrow[G,n]{\perp, \phi} <\text{do } i \text{ upto } s(X^i), \sigma, 1>}$$

### 7.3.5. Rules of uptonext.

The first rule applies when the body is resolved :

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'> \qquad t \neq \perp}{<\text{do } i \text{ uptonext } s(X^i), \sigma, E> \xrightarrow[G,n]{\text{if } t, L} <\text{do } i' \text{ upto } s(X^i), \sigma', E'>}$$

And the second rule applies when the body is unresolved :

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{\perp, \phi} <i', \sigma', E'>}{<\text{do } i \text{ uptonext } s(X^i), \sigma, E> \xrightarrow[G,n]{\perp, \phi} <\text{do } i' \text{ uptonext } s(X^i), \sigma', E'>}$$

### 7.3.6. Rules of sequencing.

The rules are basically the same as for the behavioral semantics, the second one taking care of the case when $i_1$ is unresolved :

$$\frac{<i_1, \sigma, E> \xrightarrow[G,n]{tt, \phi} <i'_1, \sigma'_1, E'_1> \qquad <i_2, \sigma'_1, E * E'_1> \xrightarrow[G,n]{t_2, L_2} <i'_2, \sigma'_2, E'_2>}{<i_1; i_2, \sigma, E> \xrightarrow[G,n]{t_2, L_2} <i'_2, \sigma'_2, E'_1 * E'_2>}$$

$$\frac{<i_1, \sigma, E> \xrightarrow[G,n]{t_1, L_1} <i'_1, \sigma'_1, E'_1> \qquad t_1 \neq tt}{<i_1; i_2, \sigma, E> \xrightarrow[G,n]{t_1, L_1} <i'_1; i_2, \sigma'_1, E'_1>}$$

### 7.3.7. Rule of loop.

Same rule as in the behavioral semantics, but adding the case where the body is unresolved :

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'> \qquad t \neq tt}{<\text{loop } i \text{ end}, \sigma, E> \xrightarrow[G,n]{t,L} <i'; \text{loop } i \text{ end}, \sigma', E'>}$$

### 7.3.8. Rules of conditional.

The rule are again obvious :

$$\frac{[\![ boolexp ]\!]\sigma = tt \qquad \langle i_1, \sigma, E\rangle \xrightarrow[G,n]{t_1, L_1} \langle i'_1, \sigma'_1, E'_1\rangle}{\langle \text{if } boolexp \text{ then } i_1 \text{ else } i_2 \text{ end}, \sigma, E\rangle \xrightarrow[G,n]{t_1, L_1} \langle i'_1, \sigma'_1, E'_1\rangle}$$

$$\frac{[\![ boolexp ]\!]\sigma = ff \qquad \langle i_2, \sigma, E\rangle \xrightarrow[G,n]{t_2, L_2} \langle i'_2, \sigma'_2, E'_2\rangle}{\langle \text{if } boolexp \text{ then } i_1 \text{ else } i_2 \text{ end}, \sigma, E\rangle \xrightarrow[G,n]{t_2, L_2} \langle i'_2, \sigma'_2, E'_2\rangle}$$

### 7.3.9. Rules of parallel.

The first rule applies when both sides are simultaneously resolved or unresolved:

$$\frac{\langle i_1, \sigma, E\rangle \xrightarrow[G,n]{t_1, L_1} \langle i'_1, \sigma, E'_1\rangle \qquad \langle i_2, \sigma, E\rangle \xrightarrow[G,n]{t_2, L_2} \langle i'_2, \sigma, E'_2\rangle \qquad t_1 \cap t_2 \neq \perp \text{ or } t_1 = t_2 = \perp}{\langle i_1 \,\|\, i_2, \sigma, E\rangle \xrightarrow[G,n]{t_1 \cap t_2, L_1 \cup L_2} \langle i'_1 \,\|\, i'_2, \sigma, E'_1 * E'_2\rangle}$$

Notice that the above rule is very different from the corresponding behavioral rule, since $E'_1$ and $E'_2$ can be now computed by pure structural induction. This will be also true for the other rules for $\|$, which correspond to the case where one of the arms is resolved and the other arm is unresolved. We need to remember the output termination tag, exit and reconfiguration of the resolved arm. For this we introduce two new operators : ${}^{t_1, L_1}\|$ means that the left arm has been resolved with termination status $t_1$ and exits $L_1$, and conversely $\|^{t_2, L_2}$ means that the second arm has been resolved with status $t_2$ and exits $L_2$ (in an implementation, this amounts of course to keep appropriate attributes together with the parallel node - one must say that the implementation is obvious here while the rewrite rule formalism is quite heavy). This gives the two rules :

$$\frac{\langle i_1, \sigma, E\rangle \xrightarrow[G,n]{t_1, L_1} \langle i'_1, \sigma, E'_1\rangle \qquad \langle i_2, \sigma, E\rangle \xrightarrow[G,n]{\perp, \phi} \langle i'_2, \sigma, E'_2\rangle \qquad t_1 \neq \perp}{\langle i_1 \,\|\, i_2, \sigma, E\rangle \xrightarrow[G,n]{\perp, \phi} \langle i'_1 \,{}^{t_1, L_1}\!\|\, i'_2, \sigma, E'_1 * E'_2\rangle}$$

$$\frac{\langle i_1, \sigma, E\rangle \xrightarrow[G,n]{\perp, \phi} \langle i'_1, \sigma, E'_1\rangle \qquad \langle i_2, \sigma, E\rangle \xrightarrow[G,n]{t_2, L_2} \langle i'_2, \sigma, E'_2\rangle \qquad t_2 \neq \perp}{\langle i_1 \,\|\, i_2, \sigma, E\rangle \xrightarrow[G,n]{\perp, \phi} \langle i'_1 \,\|^{t_2, L_2}\, i'_2, \sigma, E'_1 * E'_2\rangle}$$

Now the rules for the new operators are immediate. Two rules apply when the arm still remains unresolved :

$$\frac{<i_2, \sigma, E> \xrightarrow[G,n]{\perp,\phi} <i'_2, \sigma, E'_2>}{<i_1\ {}^{t_1,L_1}\!\|\ i_2, \sigma, E> \xrightarrow[G,n]{\perp,\phi} <i_1\ {}^{t_1,L_1}\!\|\ i'_2, \sigma, E'_2>}$$

$$\frac{<i_1, \sigma, E> \xrightarrow[G,n]{\perp,\phi} <i'_1, \sigma, E'_1>}{<i_1\ \|^{t_2,L_2}\ i_2, \sigma, E> \xrightarrow[G,n]{\perp,\phi} <i'_1\ \|^{t_2,L_2}\ i_2, \sigma, E'_2>}$$

And the two last rules finally resolve the || operator :

$$\frac{<i_2, \sigma, E> \xrightarrow[G,n]{t_2,L_2} <i'_2, \sigma, E'_2> \quad t_2 \neq \perp}{<i_1\ {}^{t_1,L_1}\!\|\ i_2, \sigma, E> \xrightarrow[G,n]{t_1 \cap t_2, L_1 \cup L_2} <i_1 \| i'_2, \sigma, E'_2>}$$

$$\frac{<i_1, \sigma, E> \xrightarrow[G,n]{t1,L_1} <i'_1, \sigma, E'_1> \quad t_1 \neq \perp}{<i_1\ \|^{t_2,L_2}\ i_2, \sigma, E> \xrightarrow[G,n]{t_1 \cap t_2, L_1 \cup L_2} <i'_1 \| i_2, \sigma, E'_1>}$$

**7.3.10. Rules of tag.**

The rules are similar to the behavioral ones, but taking care of an unresolved body :

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'> \quad t=tt \text{ or } (t=ff \text{ and } L=\{T\})}{<\text{tag T in } i \text{ end}, \sigma, E> \xrightarrow[G,n]{tt,\phi} <\text{nothing}, \sigma', E'>}$$

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'> \quad (t=ff \text{ and } L\neq\{T\}) \text{ or } t=\perp}{<\text{tag T in } i \text{ end}, \sigma, E> \xrightarrow[G,n]{t,L-\{T\}} <\text{tag T in } i' \text{ end}, \sigma', E'>}$$

**7.3.11. Axiom of exit.**

$$<\text{exit T}, \sigma, E> \xrightarrow[G,n]{ff,\{T\}} <\text{nothing}, \sigma, 1>$$

**7.3.12. Rule of variable declaration.**

$$\frac{<i, \sigma, E> \xrightarrow[G,n]{t,L} <i', \sigma', E'>}{<\text{var X in } i \text{ end}, push(\perp,\sigma), E> \xrightarrow[G,n]{t,L} <\text{var X in } X^1{:=}top(\sigma'); i', pop(\sigma'), E'>}$$

### 7.4. Identity of the computational and behavioral semantics.

**Theorem**: Let P be a statically correct program with associated graph $G$. Then for any memory state $\sigma$ and input event $E$ there exists a unique provable sequence of transitions of the form

$$\langle P, \sigma, E\rangle \xrightarrow[G,0]{t_0,\phi} \langle P_0, \sigma_0, E_0\rangle$$

$$\langle P_0, \sigma_0, E * E_0\rangle \xrightarrow[G,1]{t_1,\phi} \langle P_1, \sigma_1, E_1\rangle$$

$$\langle P_{k-1}, \sigma_{k-1}, E * E_0 * E_1 * \cdots * E_{k-1}\rangle \xrightarrow[G,k]{t_k,\phi} \langle P_k, \sigma_k, E_k\rangle$$

with $k \leq m$, $t_0 = t_1 = \cdots = t_{k-1} = \perp$ and $t_k \neq \perp$. Moreover one has in the behavioral semantics

$$\langle P, \sigma, E\rangle \xrightarrow{t_k,\phi} \langle P_k, \sigma_k, E * E_0 * E_1 * \cdots * E_k\rangle$$

The proof is omitted.

## 8. Compiling ESTEREL programs into automata.

### 8.1. Symbolic evaluation of ESTEREL programs.

Although the rules of the computational semantics are much more complicated than the rules of the behavioral semantics, they are very easy to implement, and we built rapidly from them a LISP symbolic evaluator for ESTEREL which fits in a few pages. The interpreter is not fast enough for running really in real-time (it needs typically some tenths of a second to evaluate the game program on any input). But it makes possible to make easily experiments and simulations on a program.

### 8.2. Compiling programs containing only pure signals to finite automata.

The main role of the interpreter is however not to make simulations, but to produce really "compiled code" from an ESTEREL program. We shall do it first for programs containing only pure signals. Let us state a definition and an essential result :

**Definition** : write $P \overset{*}{\Longrightarrow} P'$ if there exists an input history $E_1, E_2, \cdots, E_n$ and an output history $F_1, F_2, \cdots, F_n$ such that

$$P \xrightarrow[E_1]{F_1} P_1 \xrightarrow[E_2]{F_2} P_2 \Longrightarrow \cdots \xrightarrow[E_n]{F_n} P'$$

**Theorem** : Let P be a program containing only pure signals. Then there exists only finitely many P' such that $P \overset{*}{\Longrightarrow} P'$

As a consequence we can use the symbolic evaluator for producing *all* the possible P' such that $P \stackrel{\bullet}{\Longrightarrow} P'$, and all the possible transitions from those P', using the fact that there is always a finite number of possible input events. Now we can enumerate these programs : call them $P_0=P, P_1, \ldots, P_n$. The transitions $P_i \underset{E}{\overset{F}{\Longrightarrow}} P_j$ may be equally well seen as transitions $i \underset{E}{\overset{F}{\Longrightarrow}} j$ where $i$ and $j$ are now states of a *finite automaton*. The automaton works exactly as the original program : from state $i$ with input event $E$ it produces an output event $F$ and goes to state $j$ if and only if $P_i \underset{E}{\overset{F}{\Longrightarrow}} P_j$. The essential point is that once we have produced the automaton we can completely forget about the original program, and just execute the state-to-state transitions corresponding to given input events, with emission of the output event to the external world. The automaton may therefore be trivially implemented in any classical *sequential* language. The parallelism and internal communication of the original program have totally disappeared in the "compiling" process.

It is of course important to know whether the automata produced are small or big. Our present experiments tend to show that they are indeed quite small. This is due to the strong synchrony hypothesis, which implies that many things may happen at once in an instant, and in particular that adding many internal communications does not increase the size of the automaton, contrarily to what happens in usual asynchronous mechanisms. Consider for example the two following programs, which both emit s2 every time s1 is received twice :

```
input pure signal s1,
output pure signal s2 in
    await s1;
    loop
        awaitnext s1;
        emit s2;
        awaitnext s1;
    end
end
```

```
input pure signal s1,
output pure signal s2,
local pure signal s3,s4 in
   [
      await s1;
      awaitnext s1;
      emit s3
   ||
      await s3;
      emit s2
   ||
      await s1;
      awaitnext s1;
      awaitnext s1;
      awaitnext s1;
      emit s4
   ||
      await s4;
      loop
         emit s2;
         awaitnext s1;
         awaitnext s1
      end
   ]
end
```

It is obvious that the optimal automaton has two states. Our compiling process indeed produces that automaton directly from the first program. For the second (stupid) program, the automaton produced has 6 states, but the classical minimisation algorithm brings it back to the optimal one : the additional signals and control structure give no state explosion at all.

A more convincing use of internal events for improve the programming style was of course the prompting machine of section 4.

### 8.3. Compiling general programs.

The above theorem does not hold for general programs handling possibly infinite data types. However we can still compile any program into a finite automaton representing its *control behavior*, associating with each transition a sequential program manipulating a global memory. The sequential program is written with 5 instructions :

(i) assignment X:=*exp* as before.

## 9. Conclusion.

We intend to develop the present work in four directions :

(i) *Investigating extensions to the language.* The most important one is certainly dynamic creation of processes. There as far as behavioral and computational semantics are concerned, but we will certainly loose the possibility of compiling a program into a finite automata, which was the main idea for making the strong synchrony hypothesis realistic. Hence we have to find some limitations in order to keep fast execution mechanisms. We have also to see if some applications require non-determinism in a natural way.

(ii) *Producing really usable compilers and programming environments.* This work is in progress at the moment. An important issue is separate compilation, but it raises to our opinion no major problem. We should also investigate other means of producing automata than the one presented in section 8, and especially more structural ones.

(iii) *Understanding the proof theory of ESTEREL*, and more generally relating the ESTEREL imperative formalism to non-imperative ones, such as temporal logics or event algebras [28, 12, 17]. The translation to automata is of great help there.

(iv) Making real experiments with ESTEREL programs, by constructing real-time machines and experimenting complex ESTEREL programs on them. Running real programs in real environments should also serve as the main tool for understanding what should be the programming style in a synchronous language such as ESTEREL.

**Acknowledgements**: we want to thank J-P Marmorat and J-P Rigault who got the original ideas of ESTEREL and participated to the whole project, J. Camerini and B. Nguyen-Phuoc who wrote the first SCCS formal semantics, and F. Boussinot, S. Moisan and R. de Simone for their contributions to the present version.

**References.**


1. *LTR Manuel Officiel de Référence,* Ministère de la Défense, France (1978).
2. *Net Theory and Applications,* LNCS 84, Springer-Verlag (1979).
3. *Reference Manual for the ADA Programming Language,* CII Honeywell-Bull (1980).
4. "OCCAM Programming Manual," *INMOS Limited* (1983).
5. E. Astesiano and E. Zucca, "Semantics of CSP via Translation into CCS," in *Proc. MFCS 81*, Springer-Verlag, LNCS 116 (1981).
6. J.G.P. Barnes, *RTL/2 Design and Philosophy,* Heyden & Sons Ltd. (1976).
7. G. Berry, J. Camerini, B. Nguyen Phuoc, J.P. Marmorat, and J.P. Rigault, "Quelques Primitives pour la Progammation Temps Réel et leur Sémantique Mathématique," *Proc. Real Time Data Conference, INRIA,* (1982).
8. G. Berry, S. Moisan, and J.P. Rigault, "ESTEREL: Towards a Synchronous and Semantically Sound High Level Language for Real Time Applications," Proc. IEEE 1983 Real-Time Systems Symposium (1983).
9. G. Boudol and D. Austry, "Algèbre de Processus et Synchronisation," *Theoretical Computer Science* **30**, pp.91-131 (1984).
10. N.G. de Bruijn, "AUTOMATH, a Language for Mathematics," in *Lecture Notes prepared by B. Fawcett,* , Les Presses de l'Université de Montreal, Canada, (1973).
11. J. Camerini, "Sémantique Mathématique de Primitives Temps Réel," Thèse de Troisième Cycle, Université de Nice (1982).
12. P. Caspi and N. Halbwachs, "Algebra of Events: a Model for Parallel and Real-Time Systems," RR 285, IMAG, Grenoble (1982).
13. J. Chailloux, M. Devin, and J.M. Hullot, "LELISP: a Portable and Efficient LISP System," *1984 ACM Symposium on LISP and Functional Programming, Austin, Texas* (1984).
14. E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach," Department of Computer Science Report, Carnegie-Mellon University (Septermber 1983).
15. L. Cosserat, "Sémantique Opérationnelle du Langage Synchrone ESTEREL," Thèse de Docteur Ingénieur, Ecole des Mines de Paris (1984).
16. M. Gordon, R. Milner, and C. Wadsworth, "Edinburgh LCF," Lecture Notes in Computer Science 78, Springer-Verlag (1980).
17. N. Halbwachs, "Modélisation et Analyse du Comportement des Systèmes Informatiques Temporisés," Thèse de Doctorat d'Etat, Université de Grenoble (1984).

(ii) An instruction X:=*val*(*s*) assigning to X the values conveyed by an external signal in the current input event.

(iii) An instruction emit s(*exp*) for emitting a signal with some value (list)

(iv) The usual conditional.

(v) A "newstate $n$" instruction which indicates that the current processing is terminated and that the new state is $n$.

We shall give no more details here, the compiler will be described in a forthcoming paper. But we may mention that the game and prompting machines of section 3 lead to automata having respectively 4 and 11 states. We also give the example of a program which emits s2(X+Y) every time it has received two successive signals s1(X) and s2(Y), but only if X+Y is even :

```
input single signal s1(int),
output single signal s2(int) in
   var X, Y : int in
      loop
         await s1(X);
         awaitnext s1(Y);
         if even(X+Y) then emit s2(X+Y) fi;
         awaitnext s1
      end
   end
end
```

The automaton obtained looks as follows :

```
state 0 :
   1  --> newstate 0
   s1 --> X:=val(s1); newstate 1
state 1 :
   s1 --> Y:=val(s1);
          if even(X+Y) then
             emit s2(X+Y);
             newstate 0
          else
             newstate 0
          fi
```

Again that automaton is trivially implemented in any sequential language. Notice that the execution of a transition may be very fast, so that the strong synchrony hypothesis may be met in practice upto a really satisfactory point. Moreover if we have a model of the execution machine we can exactly compute the time taken by transitions, and know if the machine is fast enough for executing the program in a satisfactory way.

We are not forced to allocate globally *all* variables in a program. Remember the stack allocation technique for variables : we keeped the value of a variable *inside* the program by an appropriate assignment. If the data type on which the variable ranges is finite, we can still apply this technique and get a finite automaton without allocation of the variable, transforming so to speak data into control (there is indeed no problem in intermixing the two allocation techniques). A typical use of that method is for protocols: messages between senders and receivers convey generally two kinds of informations, the message itself which will naturally be stored in a globally allocated variable, and some control information which is often limited to a finite number of values. Then it is good practice not to allocate the variables which hold that information: it makes the analysis and minimization of the automaton produced both easier and more powerful.

If a program has $n$ input signals, the number of transitions to consider from any state is normally $2^n$, since we have to consider all possible input events. However there are often relations between the input signals : two signals may be always synchronous (say millisecond and second) or on the contrary incompatible. We give in the prototype compiler rather crude ways of specifying such facts, and this results in reducing dramatically the size of the automata (mainly in the number of transitions to consider). Much work remains to be done there, in particular for expressing such constraints in a general formalism (see in particular [28]), and for taking care of these additional constraints in the static semantics.

As far as automaton analysis is concerned, we have done very succesful experiments with the EMC temporal logic model checker [14]. This will also be detailed elsewhere.

### 8.4. Interfacing ESTEREL programs.

An ESTEREL program produces an automaton which should be interfaced to the real world. We have little to say here, since the problem is common to all real time languages. The only difference is that the ESTEREL program should *not* be considered as the "master" program as in usual languages, but rather as a "servant". Some master program should decide *when* events have really come from the outside, send them to the ESTEREL program for calculations and state transitions, and transmit to the outside world the emitted signals. This is particularly clear for programs which test for the instantaneous conjunction of two events (resetting a game by pressing two buttons together is a good example). The master must decide when to consider that two external events are simultaneous. Further investigation is obviously needed here.

18. M.C.B. Hennessy, W. Li, and G.D. Plotkin, "A First Attempt at Translating CSP into CCS," Research Report, Edinburgh University (1980).

19. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM* **21**(8), pp.666-678 (1978).

20. R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, LNCS 92 (1980).

21. R. Milner, "Calculi for Synchrony and Asynchrony," *Theoretical Computer Science* **25**(3), pp.267-310 (1983).

22. J.L. Peterson, "Petri Nets," *Computing Surveys* **9**(3) (1977).

23. G.D. Plotkin, "A Powerdomain Construction," *SIAM Journal on Computing* **5**(3), pp.452-487 (1976).

24. G.D. Plotkin, "An Operational Semantics for CSP," Research Report, Edinburgh University (1981).

25. G.D. Plotkin, "A Structural Approach to Operational Semantics," Lectures Notes, Aarhus University (1981).

26. A. Pnueli, "The Temporal semantics of concurrent Programs," *TCS* **13**, pp.45-60 (1981).

27. R. de Simone, "Calculabilité et Expressivité dans l'Algèbre des Processus Parallèles MEIJE," Thèse de Troisième Cycle, Université Paris VII (1984).

28. G. Winskel, "Events in Computations," PhD Thesis, Univ. of Edinburgh (1980).

29. S.J. Young, *Real-Time Languages : Design and Development*, Ellis Horwood Publishers (1982).

**Imprimé en France**

**par**

**l'Institut National de Recherche en Informatique et en Automatique**