The ESTEREL Language

FRÉDÉRIC BOUSSINOT AND ROBERT DE SIMONE

Invited Paper

We present the basics of the **ESTEREL** *reactive model of synchronous parallel systems. We illustrate the* **ESTEREL** *programming style, based on "instantaneous communications and decisions" through the example of a mouse handler. We briefly describe the* **ESTEREL** *formal semantics and show how programs can be compiled into finite states sequential machines for efficient execution. The up to date implementation is described together with the* **ESTEREL** *environment, including simulation, and verification and validation tools. Finally, we report on some* **ESTEREL** *uses in various contexts.*

I. Introduction

The **ESTEREL** language is a member of the new family of *synchronous* languages for *reactive programming*, which also counts languages as **LUSTRE** [13], **SIGNAL** [17], **SML** [18], and **STATECHARTS** [14].

ESTEREL originated from a joint INRIA-ENSMP project on the semantics of parallelism. Better understanding of real-time programming especially of temporal features such as "watchdogs," was the prime motivation. In the course of the **ESTEREL** design, the necessity of a formal approach became more and more clear: without mathematical semantics **ESTEREL** would not exist. As a benefit of the formal approach, **ESTEREL** programs can be compiled efficiently into finite states machines and can generate efficient code.

In this paper we show the basic principles of the **ESTEREL** reactive model and we give a flavor of the new programming style that comes with it. We also give an idea of the semantics and discuss how efficient code can be produced from **ESTEREL** programs.

The paper is organized as follows: In the second section we describe the **ESTEREL** basic assumptions and the underlying model. In the third section we give some examples of **ESTEREL** programs. In the fourth section we detail the most striking features of the **ESTEREL** formal semantics. In the fifth section we describe the compiling process and the **ESTEREL**v3 system. In the sixth section we present the **ESTEREL** existing environment including validation and simulation tools. In the seventh section we discuss future extensions to the present language. Finally, in the eighth section we briefly summarize nowadays programming experiences in **ESTEREL**.

Manuscript received September 30, 1990; revised March 2, 1991.
F. Boussinot is with ENSMP-CMA, F-06565-Valbonne, France.
R. de Simone is with INRIA-Sophia, F-06560-Valbonne, France.
IEEE Log Number 9102300.

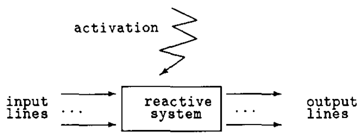


Fig. 1. A reactive system.

II. The Underlying Model

Motivations behind **ESTEREL** can be summed up by the following "equation":

ESTEREL = reactivity
+ atomicity of reactions
+ instantaneous broadcast
+ determinism

We are going to justify now each component of this equation.

A. Reactivity

The **ESTEREL** basic model is the *reactive* model in which one considers communicating systems that continuously interact with their environment [15]. When activated with an *input event*, a reactive system *reacts* by producing an *output event*. Reactive systems are seen as "black boxes" that must be activated from outside in order to react; they have input lines to receive input events from the outside, and output lines to produce output events. A reactive systems is shown in Fig. 1.

The life of a reactive system is divided into *instants* that are the moments where it reacts. Accordingly, one can speak of the first instant of a program, the second instant, and so on. We call *reactive* statements, statements that are defined by reference to instants. For example, in **ESTEREL**, "`await S`" stops execution until the first instant the signal `S` becomes present. The most basic **ESTEREL** reactive statement is the `watching` statement that implements a generalized watchdog. It will be described later on.

0018-9219/91/$01.00 © 1991 IEEE

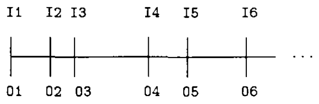


Fig. 2. Reactive system as history transducer.

To sum up, we can say that the reactive approach allows one to reason in a *logics of instants* and to program using reactive statements.

reactivity = instants + reactive statements

B. Atomicity of Reactions

The basic hypothesis of **ESTEREL** is called the *perfect synchrony hypothesis* [3]: it says that reactions are instantaneous so that activations and productions of output are synchronous, as if programs were executed on an infinitely fast machine. This idealized hypothesis can be more practically expressed by saying that reactions are *atomic*, that is, a particular reaction do not interfere with the others reactions. In other words, program reactions cannot overlap: there is no possibility to activate a system while it is still reacting to the current activation. This hypothesis simplifies the reasoning about reactive systems as concurrency between reactions is not to be considered. Without the atomicity assumption, this concurrency would be a source of nondeterminism.

$$\text{synchrony hypothesis} \equiv \text{atomicity of reactions.}$$

Atomicity of reactions allows to speak of the basic *clock of activations*. In other words, it allows to consider reactive programs as *history transducers*: a reactive program is a function that produces a sequence of output events from a sequence of input events. Figure 2 shows an output history `O1,...,On,...` produced from an input history `I1,...,In,...`.

C. Instantaneous Broadcast

As other concurrent languages (Occam, for example), **ESTEREL** has a parallelism operator, written ||. With it, one directly program parallel entities. In Ada these entities that are called tasks, communicate and synchronize using a "hand shaking" mechanism. This mechanism (sometimes called "rendezvous") is one-to-one: it only allows one entity to communicate with another one, at a time. On the contrary, *broadcast* is the unique communication mechanism in **ESTEREL**. Broadcast, in opposition to "hand shaking," can be seen as "hand raising": when one wants to communicate, one raises its hand, so everybody can see it. It is analogous to radio communication where there are many receptors that *all receive the same information* (this is a consequence of the synchronous characteristics of the parallel operator that we will describe later on).

In **ESTEREL**, communication is done using *signals* that can be emitted, tested for presence, and that can have a value with. Internal communication (between **ESTEREL** subprograms) and external communication (between **ESTEREL** programs and the external world) are unified in the same framework of signals: input and output lines are signals and input and output events are sets of signals. Figure 3 shows more precisely the architecture of an **ESTEREL** program made of several subprograms, called *modules*. In it, the doted lines denote signals, and the broken lines indicate that the basic clock of activations is shared by all submodules.

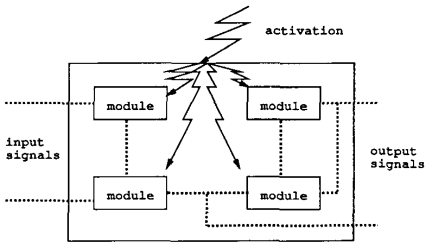


Fig. 3. An ESTEREL module.

Broadcast is limited to instants: the emission of a signal (using an `emit` statement) lasts for the current instant and the emitted signal is seen as present (for example, using a `present` statement) by all the receptors, during this instant. For example, consider:

```
   present S then emit T end
||
   emit S
||
   present S then emit U end
```

The signal `S` is emitted by the `emit S` instruction. It is seen as present by both `present` instructions and both `then` branches are executed. So `S`, `T`, and `U` are all emitted during the same instant: they are *synchronous*.

Notice that this program has one unique solution: we do not have to consider the case where `S` would be considered as absent by any of the receptors. In this respect, broadcast reduces the number of distinct possible communications.

As can be seen on the previous example, **ESTEREL** broadcast has an important characteristic: emissions and receptions do no terminate the current instant. In other words, there can be several signal emissions and receptions in sequence within the same instant. This characteristic allows to program so called "*instantaneous decisions*" that are specific to **ESTEREL**. Emission of a query and reception of the answer during the same instant is an example of instantaneous decision. Later we shall show an instantaneous decision in the program example of a mouse handler:

$$\text{instantaneous broadcast} \Rightarrow \text{instantaneous decision.}$$

D. Determinism

Nondeterminism is completely thrown out of **ESTEREL**. With traditional approaches, one has to choose between parallelism and nondeterminism on one hand, or determinism but sequentiality on the other hand. On the contrary, **ESTEREL** parallelism is deterministic.

Determinism is another way to simplify reactive programming. With deterministic programs, behaviors are reproducible, that simplifies program tests and validations (a faulty behavior can be replayed at will):

ESTEREL= parallelism + determinism.

III. ESTEREL PROGRAMMING

We are not going to describe precisely the language (see [10] for a complete presentation) but instead give the flavor of the **ESTEREL** programming style through some small examples. Actually, we advocate how **ESTEREL** programs can be seen in a way as very close to specifications:

ESTERELprogram $\approx$ specification.

We first give some examples to illustrate the use of signals in **ESTEREL** and the `watching` statement; then we describe a small mouse handler that exhibits an instantaneous decision.

A. The Programming Style

The `watching` statement is the basic **ESTEREL** reactive statement. `watching` statements have the syntax: "`do` *body* watching *signal*". The semantics is that the body is "killed" as soon as the signal watched for becomes present.

To illustrate the use of the `watching` statement, consider the following specification *Spec1*: within a delay of one second, do an action once a button is pushed. In **ESTEREL** one may write:

```
do
   await BUTTON;
   emit ACTION
watching SECOND
```

If SECOND is present before BUTTON or at the same instant, then ACTION is not emitted and the `watching` terminates. On the contrary, if BUTTON is present before SECOND, then ACTION is immediately emitted and the `watching` terminates. Otherwise, if neither SECOND nor BUTTON are present, nothing is done and the waiting for BUTTON and SECOND is postponed to the next instant. There is a special case at the first instant where in all cases, nothing is done. To extend to the first instant the previous behavior, one must use the "immediate" variant of the `watching` and `await` statements:

```
do
  await immediate BUTTON;
  emit ACTION
watching immediate SECOND
```

Now consider the "opposite" specification *Spec2*: within a delay of one second, raise an alarm if a button is not pushed. One may simply write:

```
do
  await SECOND;
  emit ALARM;
watching BUTTON
```

Suppose now that we want to extend *Spec1* with the following: an alarm must be emitted when the button is not pressed within the one second delay. Notice that this extended specification *Spec3* is partial (as also are *Spec1* and *Spec2*): it does no tell what to do when the button is pushed in exactly one second.

The *Spec3* specification can be coded by adding a `timeout` part to the `watching` statement:

```
do
  await BUTTON;
  emit ACTION
watching SECOND
timeout emit ALARM
```

The `timeout` part is executed only when the watched signal becomes present and the body is not yet terminated. So, the signal ALARM is emitted if SECOND is present before BUTTON. Notice the deterministic behavior in case SECOND and BUTTON are both present together: according to the semantics of `watching`, the body is killed, so ACTION is not emitted, and the `timeout` part is executed, so ALARM is emitted.

The following program also satisfies the partial *Spec3* specification:

```
do
  await SECOND;
  emit ALARM
watching BUTTON
timeout emit ACTION
```

Now, when SECOND and BUTTON are both present at the same instant only ACTION is emitted.

Finally, suppose one wants ACTION and ALARM to be both emitted when SECOND and BUTTON occur simultaneously. One may write:

```
trap END in
  await SECOND;
  emit ALARM;
  exit END
 ||
  await BUTTON;
  emit ACTION;
  exit END
end
```

The `trap END` statement defines a block that is instantly exited when "`exit END`" is executed. So, the "`trap END`" block is terminated as soon as SECOND or BUT-

TON or both, are present. Moreover, when SECOND and BUTTON are simultaneously present, ACTION and ALARM are both emitted as a consequence of the parallel operator semantics.

B. A Mouse Handler

We consider a mouse handler with two inputs:

1) CLICK: a push button,
2) TOP: a clock signal.

The mouse handler has to figure the number of CLICK's (none, one, or more than one) performed within a delay of five TOP's. Accordingly, it outputs NONE, SINGLE, or MANY.

We first define an auxiliary module Counter that counts occurrences of CLICK. When a signal RST is present, Counter emits a valued signal VAL whose value is the number of occurrences of CLICK. The **ESTEREL** program is the following:

```
module Counter:
input RST, CLICK;
output VAL(integer);

var v : integer in
   do
      v := 0;
      every immediate CLICK do
         v := v+1;
      end
   watching RST;
   emit VAL(v)
end
```

The "input RST, CLICK;" declaration introduces two input signals; these signals are *"pure"*: they carry no value and only their presence or absence is significant. The "output VAL(integer);" declaration introduces an output signal with an integer value. The statement "emit VAL(v)" emits the signal VAL with value v. Notice that the module Counter terminates at the instant RST becomes present (because of the watching semantics).

Second, we define an auxiliary module Emission that processes the value of the valued signal VAL. Accordingly, it outputs NONE, SINGLE, or MANY. Emission begins to wait for VAL and then uses its value denoted by ?VAL.

```
module Emission:
input VAL(integer);
output NONE, SINGLE, MANY;

await VAL;
if ?VAL = 0 then
   emit NONE
else
   if ?VAL = 1 then
      emit SINGLE
   else
      emit MANY
   end
end
```

Finally, in the main module Mouse, within a global loop, one puts in parallel a copy of Counter and a copy of Emission (using the copymodule statement), and a statement that emits RST when five TOP's have been received.

```
module Mouse:
input CLICK, TOP;
output NONE, SINGLE, MANY;

signal RST, VAL(integer) in
  loop
     copymodule Counter
    ||
     await 5 TOP;
     emit RST;
    ||
     copymodule Emission
  end
end
```

The signal keyword introduce the declaration of two local signals that correspond to those of Counter. The parallel statement terminates when its three branches terminate. It is the case when the fifth TOP is present, because then RST is emitted which forces Counter and Emission to terminate. The loop statement lets the behavior of Mouse cycle: the parallel statement is restarted immediately at the same instant it terminates.

Notice the instantaneous communication and decision inside the parallel statement: the second branch emits the signal RST that is received by Counter in the first branch. Then at the same instant, Counter emits Val that is received and processed by Emission.

IV. ESTEREL SEMANTICS

The **ESTEREL** reactive approach gives rise to several specific problems, leading to undesirable programs. Some of them, involving signals, are called *causality problems*. They are akin to short-circuits in electronics and appear in all powerful synchronous languages. There is a need for a formal semantics to solve these problems. In this section we first describe the new class of problems and then we give some indications on the semantical approach that has been used to tackle them.

A. Specific Problems

Instantaneous Loops: *Instantaneous loops* are loops whose body terminate at the instant they are executed for the first time. Such loops must be rejected as they do not allow to agree on completion of the instant. An example of instantaneous loop using a variable **x** is:

```
loop x:=x+1 end
```

Causality Problems: **ESTEREL** programs communicate via signals which are instantaneously broadcast. **ESTEREL** communication is powerful but, we have to pay for it: there can be so-called "causality problems" [4]. There are two kinds of causality problems. First, consider the following program **Causal0**:

```
signal S in
   present S else emit S end
end
```

This program has no solution: if one supposes that the local signal S is absent, then it is emitted; on the other hand, we cannot suppose it is present as then, it would be not emitted. In other words, **Causal0** is incoherent, as it violates the **ESTEREL** broadcast communication mechanism.

Second, consider the following program **Causal2**:

```
signal S1, S2 in
   present S1 else emit S2 end
||
   present S2 else emit S1 end
end
```

This program has two solutions: in the first solution S1 is absent and S2 is present; conversely, in the second solution S2 is absent and S1 is present. So, **Causal2** is nondeterministic and the **ESTEREL** determinism hypothesis is violated.

Valued Signals: Some problems are specific to valued signals. For example, consider:

```
emit S(?S + 1)
```

This is a kind of "positive feedback" effect: the value of S denoted by ?S, must verify: ?S = ?S + 1, which clearly has no solution.

It is important to notice that instantaneous loops and causality problems are detected by **ESTEREL** compilers. Causality problems are the synchronous counterpart of asynchronous deadlocks. More precisely, causality problems can be seen as *instantaneous deadlocks*. For example, in **Causal2**, one needs to assume the presence status of S1 to decide of the emission of S2, and vice-versa. The difference with general deadlocks is that causality problems can be detected at compile time instead of being undetected at run-time.

ESTEREL ⇒ static detection of instantaneous deadlocks.

Of course, noninstantaneous deadlocks exist in **ESTEREL** and are not statically detected. It is the case for example, in:

```
signal S1, S2 in
   await S1; emit S2
||
   await S2; emit S1
end
```

This statement is correct in **ESTEREL**, but it never terminates and does nothing at all.

B. Mathematical Semantics

The goal of formal semantics is to describe without ambiguity how programs behave. The formal approach is the guide line for compiler or validation tools designers.

There exist several distinct levels of description of **ESTEREL** formal semantics, expressed either denotationally or operationally. They are presented and connected in [12].

The most fruitful of these semantics is the *behavioral* semantics, presented in the so called "*Structural Operational Semantics*" framework [22]. It provides an abstract vision of the reactive structure of programs and allows simple reasoning. It accepts more programs than any other semantics: only programs without solution, as **Causal0**, are rejected (but nondeterministic programs, such as **Causal2**, are accepted).

The Behavioral Semantics: The behavioral semantics we are going to describe now, manipulates operational *transitions* of the following general format :[1]

$$\mathit{Program} \xrightarrow[\mathit{Terminated}]{\mathit{Input/Output}} \mathit{NewProgram}$$

This transition is read as follows: at the first instant and with $Input$ as input event, the program $Program$ reacts by producing the output event $Output$. $NewProgram$ is the *residual* program to be executed at the next instant. Moreover, the boolean $Terminated$ is true if and only if $Program$ has terminated its execution. For example, the semantics of the **emit** statement is given by

$$\texttt{emit}\ S \xrightarrow[\textbf{true}]{I/\{S\}} \texttt{nothing}$$

The Semantics of compound statements are constructed out from the semantics of their components, using *inference rules*. To give an idea, the semantics of the full **ESTEREL** language needs approximatively thirty such rules. The semantics of watching is given by the two following rules:

$$\frac{p \xrightarrow[\texttt{true}]{I/O} q}{\texttt{do}\ p\ \texttt{watching}\ S \xrightarrow[\texttt{true}]{I/O} \texttt{nothing}}$$

[1] Slightly simplified, as we do not take trap and exit constructs into account.

$$\frac{p \xrightarrow[\textbf{false}]{I/O} q}{\texttt{do}\ p\ \texttt{watching}\ S \xrightarrow[\textbf{false}]{I/O} \texttt{present}\ S\ \texttt{else do}\ q\ \texttt{watching}\ S\ \texttt{end}}$$

The first. rule is read as following: one can conclude that, under the hypothesis that execution of the body p terminates, the `watching` statement terminates too, with `nothing` as residual program (q, the residual of p disappears in the conclusion, as execution is terminated). Moreover, emitted signals are those emitted by p. The second rule holds in case p does not terminate. Then, S will be tested for presence next instant, deciding upon whether to execute the body.

The semantics of the parallel operator is given by the unique rule:

$$\frac{p_1 \xrightarrow[b_1]{I/O} q_1 \qquad p_2 \xrightarrow[b_2]{I/O} q_2}{p_1 \parallel p_2 \xrightarrow[b_1\ \textbf{and}\ b_2]{I/O_1 \cup O_2} q_1 \parallel q_2}$$

The parallel operator is synchronous: its two branches are working at each instant. It terminates as soon as both its branches terminate (because of the **and** connective).

These rules give only a flavor of the behavioral semantics. Notice that in the rule of the parallel operator, both subterms p_1 and p_2 share the same input I. The more complex semantics of local signals declarations has to verify the coherency between signals emissions and receptions. Actually, in the behavioral semantics, this coherency must be verified after making hypothesis about local signals presence or absence. Sequential implementations of the behavioral semantics necessary need some kind of backtracking.

The Execution Semantics: The purpose of the *execution* semantics is twofold:

- First, to reject nondeterministic programs as well as programs without solution.
- Second, to give an efficient implementation, avoiding hypothesis to be possibly negated later on and leading to backtracks.

The execution semantics is based on a so-called*potential function* [12], that syntactically forecast which signals may or may not be further emitted inside the instant. These informations are used to order the processing of signals: one execute presence tests on a given signal only when the signal is not in the potential anymore. When one cannot find any order to process signals, one detects a causality problem. For example, the previous **Causal2** program is rejected by the execution semantics: no good order exists as both signals `S1` and `S2` are in the potential of the parallel statement. Unfortunately, as the potential function is syntactically defined, the execution semantics is only an approximation of the semantics where exactly programs with an unique solution are accepted: there exist programs that have an unique transition for each input, with the behavioral semantics, but that are rejected by the execution's.

V. Esterel Implementations

In this section we describe the **Esterel** compiling technics, code production and the current **Esterel**v3 implementation.

A. Compiling into Automata

Perhaps the most striking characteristics of **Esterel** is the ability toproduce a finite state machine, also called *automaton.* The construction follows the execution semantics in a symbolic evaluation. The execution rules provide a transition from a state, for each input event. The automaton is constructed by merely gathering those transitions, identifying states with successive residuals. The key point here is that there are only finitely many such residual terms for an **Esterel** program.

For example, consider the "`do halt watching S`" statement, where `halt` is the statement that never terminates. The semantics of `halt` is given by the transition:

$$\texttt{halt} \xrightarrow[\textbf{false}]{I/O} \texttt{halt}$$

We denote by T the program

$$T \equiv \texttt{present}\ S\ \texttt{else do halt watching}\ S\ \texttt{end}$$

Accordingly to the semantics of `watching`, for all input I, one has:

$$\texttt{do halt watching}\ S \xrightarrow[\textbf{false}]{I/O} T$$

Consider now, the residual program T. For all input I such that $S \notin I$, one shows that:

$$T \xrightarrow[\textbf{false}]{I/O} T$$

The proof is given by the following *proof tree*:

$$\frac{S \notin I \quad \dfrac{\texttt{halt} \xrightarrow[\textbf{false}]{I/O} \texttt{halt}}{\texttt{do halt watching}\ s \xrightarrow[\textbf{false}]{I/O} \texttt{T}}}{T \xrightarrow[\textbf{false}]{I/O} T}$$

Conversely, for all input I such that $S \in I$, one has:

$$T \xrightarrow[\textbf{true}]{I/O} \texttt{nothing}$$

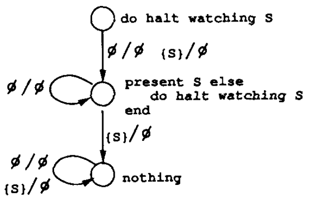


Fig. 4. Automaton associated to an ESTEREL program.

Finally, the semantics of **nothing** is of course, given by:

$$\texttt{nothing} \xrightarrow[\textbf{true}]{I/O} \texttt{nothing}$$

Now, remark that the signal S is the only significant signal in all the programs we consider. Identifying states with residual programs, we obtain the three states automaton drawn in Fig. 4. This example shows that "`await S`" and "`do halt watching S`" are equivalent as their semantics produce the same automaton.

In the resulting automaton, both parallelism and local signal communications have disappeared: they are compiled away and sequentialized inside each instant transition. For example, the automaton produced from the module `Mouse` has two states and correspond to the following sequential textual description:[2]

```
State 0
   V3 := 5; v := 0;
   if CLICK then
      v := v+1;
      goto 1
   end;
   goto 1

State 1
   if TOP then
      if V3 then
         V2 := v;
         if V2=0 then
            V3 := 5; v := 0;
            if CLICK then
            v := (v+1); emit NONE;
            goto 1
         end;
         emit NONE;
         goto 1
      end;
      if V2=1 then
         V3 := 5; v := 0;
         if CLICK then
            v := (v+1); emit SINGLE;
            goto 1
            end;
            emit SINGLE;
            goto 1
         end;
         V3 := 5; v := 0;
         if CLICK then
            v := (v+1); emit MANY;
            goto 1
         end;
         emit MANY;
         goto 1
      end;
      if CLICK then
         v := (v+1);
         goto 1
      end;
      goto 1
   end;
   if CLICK then
      v := (v+1);
      goto 1
   end;
   goto 1
```

In this text, `V2` is the value of the signal `VAL` and `V3` is used to count five `TOP`'s. Notice that in state 1, `CLICK` is always processed even in the case `TOP` is also present.

The automaton produced is deterministic; parallelism and communications that appear in the **ESTEREL** source code have been compiled to produce sequential code. In particular, the instantaneous communication in the module `Mouse` has completely disappeared. In a sense, this justifies the synchrony hypothesis: the communication is infinitely fast since there is nothing to execute as *no code* is generated!

Compiling **ESTEREL** programs into automata offers many advantages:

Efficiency: Automata can be efficiently executed. As no parallelism appear in automata any more, there is no time overhead resulting from run-time tasks or processes management, or from run-time communication or synchronization. On the contrary, suppose we have to implement the mouse system in Ada using a task `Counter`. Then we have to pay for task management and communication. This is often why only the sequential part of Ada is used for real-time programming.

Predictability: The maximum transition time of an automaton is predictable. This is especially important when one want to code real-time systems.

ESTEREL =compilation of parallelism
and output as automaton

B. The ESTERELv3 *System*

The **ESTEREL**v3 system is based on a very efficient compiling algorithm described in [12]. It rejects nondeterministic programs and programs with causality problems or instantaneous loops. As a benchmark, it takes about four

[2] This text is actually generated by the ESTERELv3 compiler with the pretty printing `-debug` option.

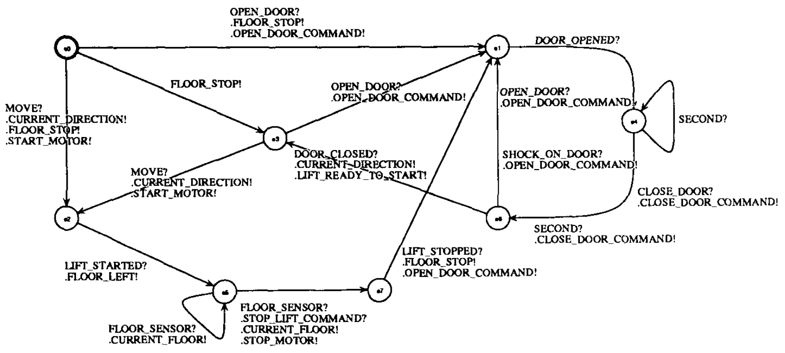


Fig. 5. The lift automaton shown by **Autograph**.

seconds to compile on a Sun3 the wristwatch program of [2] that generates a 41 states automaton.

The **ESTEREL**v3 system is written in C++ [24],runs on several machines,[3] and produces automata in an output code format common with the LUSTRE project [7]. This format can be translated into several target languages: C, Ada,[4] LeLisp, etc. The C language is the default target: when one types `esterel fich.strl` the **ESTEREL**v3 compiler generates a `fich.c` C file. The C executable code produced from the mouse program by the **ESTEREL** compiler, is a direct implementation of the sequential pseudo-code described in Section V-A.

ESTEREL = efficient implementation

VI. The Esterel Environment

The **ESTEREL** environment tools are divided into two groups:

Verification and Validation Tools: Those tools consider the internal structure of the compiled automata. They allow comparisons with specifications, as well as total or partial visualization of states and transitions.

Simulation and Development Tools: Those tools consider **ESTEREL** programs as reactive boxes. They allow the user to activate them interactively, and record the corresponding reactions.

A. Verification

Down the **ESTEREL** programming lines, one produces, simulates, executes and validates automata. This approach complies with the Berry's WYPIWYE ("*What You Prove Is What You Execute*") principle [3]. Automata can be analyzed using several systems, in particular **Auto**[25] developed in the same INRIA-ENSMP project as **ESTEREL**.

Verification is a broad topic. Classically, it consists in confronting a program with a set of specifications, where each of these specification takes into account only partial aspects of the whole system. Specifications can be provided either operationally, in an automaton fashion, or logically, using so-called temporal logics.

Auto is an automatic verification tool dedicated to analysis of finite automata, and therefore focuses on the first approach. Its primary method is *abstraction*, which reduces a large global automaton with respect to those behaviors which are considered relevant. The reduced automaton can be more readily checked for validity. Reductions implemented in **Auto** are semantically grounded, based on process calculi theory notions such as *bisimulations* [19] and *observation criteria* [6].

In most cases, one can even do without an explicit specification, by just inspecting the reduced automaton. This requires visualization of automata, before or after reduction. The **Autograph** [23] graphical interface of **Auto** allows exactly this. It allows also to directly draw the operational specifications as automata.

As an illustration, we shall consider a simple lift program, slightly more complicated than the mouse handler. The full automaton produced by **ESTEREL** compilation is shown in Fig. 5 in its **Autograph** postscript output. Labeling actions are compound and each one represents a full reaction: input signals are suffixed by "?" and output signals by "!". For example, the action on the arrow connecting state $e0$ to $e1$ indicates that FLOOR_STOP and OPEN_DOOR_COMMAND are both emitted on reception of OPEN_DOOR.

We now want to establish that the lift may not travel with the door open. But already for this simple program, the result is rather big, so that this property is not obvious to check. We shall thus abstract the automaton and only consider the actions OPEN_DOOR, DOOR_CLOSED, LIFT_STARTED, and LIFT_STOPPED. Indeed, the lift is supposed to be in motion in between LIFT_STARTED and LIFT_STOPPED, while the door is open in between OPEN_DOOR and DOOR_CLOSED. The other signals are irrelevant to the considered property.

Using **Auto**, we perform both the hiding of irrelevant signals and the reduction (with respect to bisimulation) of the lift in Fig. 5. The result is displayed in Fig. 6. It is smaller, more manageable and more directly related to the property. Indeed, OPEN_DOOR cannot be performed in

[3] Vax, Sun3, Sun4, Hp9000, GouldPN9000 at the date of writing.

[4] Of course, only the sequential part of Ada is used.

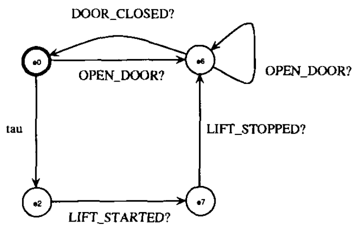


Fig. 6. The reduced lift automaton.

the state labeled by `e7`. Notice the remaining hidden `tau` action. It indicates an internal change of state to which bisimulation is sensible:

ESTEREL = possibilities of proofs and validations
and WYPIWYE.

B. Simulation

Simulation tools execute the ESTEREL program instants by instants, most often by running on its compiled automaton. There exists textual as well as graphical simulators.

The Basic Simulator: It is obtained with the `-simul` option of the `esterel` command. An instrumented C code is generated. It can be linked with a C standard library, which allows interactive simulation on the keyboard. This simulator prompts the user to type signal names which build the next input event, and then lets the program react. As an answer, names and carried values of output signals are printed.

The X-Window Graphical Simulator: It uses the same instrumented code as the basic simulator, but when linked with a X-window graphical library, it provides the user a signals menu to build input events. When an output signal is emitted back in reaction, an icon is lit up.

The Sahara Environment: With the **Sahara** environment [11], one can easily and quickly build graphical control panels to interface with ESTEREL programs. **Sahara** contains a full blown language for describing these control panels which are structured and organized at the user's will. Execution can be semiautomatic: some signals can be automatically and periodically raised by the executing computer itself. Figure 7 shows a control panel for the lift program built with **Sahara**.

ESTEREL = graphical simulation

VII. ESTEREL USE

There are several ways to design systems made of ESTEREL reactive parts. The simplest way is to use only one such part embedded into a larger system. One has to define how the automaton corresponding to the reactive part is interfaced with the overall system. This calls for defining how input and output signals are processed and how and when the automaton is executed. This leads to the architecture of Fig. 8.

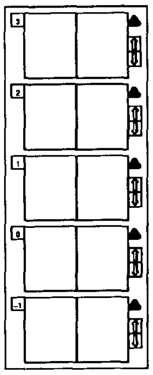


Fig. 7. Lift panel control obtained with **Sahara**.

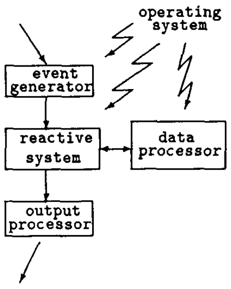


Fig. 8. A simple architecture.

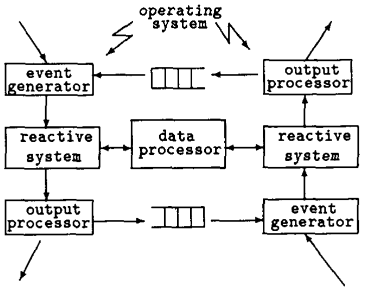


Fig. 9. A more complex architecture.

On the other hand, one can put several reactive systems together and make them communicate through asynchronous channels (first-in/first-out files, for example). It is the case for example, when one has to code distributed systems that synchronously communicate and synchronize. In these cases, ESTEREL is useful to code individual reactive parts of the overall system and both synchronous and asynchronous approaches must be used together. For example, the Fig. 9 shows two reactive systems put together in a larger system and communicating through two channels.

Moreover, there are situations where, while staying in a reactive and synchronous framework, a single automaton

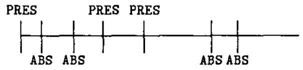


Fig. 10. Presently unimplementable specification.

would be too big an object. Notice that big automata are easy to produce. For example, the **ESTEREL** statement:

```
await S1 || await S2 || ... || await Sn
```

produces a 2^n states automaton. The *cascade* approach consists in producing several automata and giving a way to execute them as if there were an unique automaton. The **ESTEREL**v3 system offers a possibility of automatized cascading in a restricted case where automata can be executed in a fixed order independent of the instants.

Several systems have been programmed in **ESTEREL**. We can cite:

- A digital watch [2]. This program can be automatically cascaded with the **ESTEREL**v3 system. The fixed automata execution order is natural: first, the button handler, second the watch and finally the display handler.
- A "minitel" modem [16]. The produced code has been plugged into a real-time environment.
- Several communication protocols, in particular, an HDLC protocol [5], a terminal call protocol [20] and a local area network protocol [18].
- A car Antilock Braking System, that has needed a manual cascade of automata.
- A robotics application [9]. **ESTEREL** is used to ensure control sequencing for robots. The **exec** primitive described in the next paragraph, has been widely used in this application.
- Other applications, in avionics, in hardware drivers, in process controllers [1], for example.

VIII. New Extensions For Esterel

In this section we describe three new extensions for **ESTEREL**, which will be incorporated in the next version of the **ESTEREL** system. The extensions concern the abilities to program by referring to the basic instants clock and to code boolean conditions on signals, on one hand; on the other hand, it concerns a restricted form of asynchrony, based on a new primitive called `exec` [21].

A. The Next Instant

In the actual **ESTEREL** language there is no possibility to have direct control over instants: at least one of the input signals must be present at each reaction (except at the very first instant). There is no possibility to code the simple following specification: at every instant, the signal `ABS` is emitted if the signal `PRES` is not in the input event. This specification is shown in Fig. 10.

To have direct control over instants, a new input signal named `tick` is introduced. This signal is implicitly declared and it is always present[5] The signal `tick` defines the clock of step activations. Using `tick`, the previous specification can be coded by the following module:

```
module FUTURE:
input PRES;
output ABS;

every tick do
   present PRES else emit ABS end
end.
```

Note that we could equivalently introduce a new statement `stop` that stops execution for the current instant. For example, with this new statement, the specification is coded by:

```
module FUTURE:
input PRES;
output ABS;

loop
   present PRES else emit ABS end;
   stop
end
```

In fact, `stop` can be simulated by "`await tick`." Conversely, a signal `TICK` simulating `tick` can be generated by:

```
loop
    emit TICK;
    stop
end
```

B. Boolean Conditions on Signals

With the third extension, one will be able to directly use boolean conditions on signals. For example, to wait for `A` or `B` one will simply write "`await A or B`." Control over the next instant, is necessary to be able to express the boolean negation **not** corresponding to signal absence. For example, the previous `FUTURE` module could be written as:

```
module FUTURE:
input PRES;
output ABS;

every not PRES do emit ABS end
```

C. Asynchrony in Esterel

With the **exec** primitive, one can use asynchronous *tasks* in **ESTEREL**. A task is a sequential code that is not instantaneous: it does not terminate in the same instant it is started. Tasks introduce only a restricted form of

[5] It is analog to the constant "true" of **LUSTRE** [13].

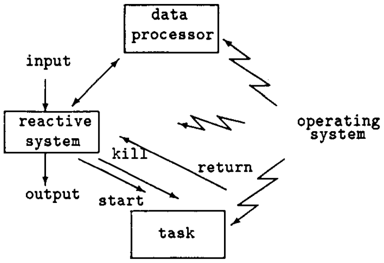


Fig. 11. Interfacing the exec primitive.

asynchrony: a task is allowed to synchronize only when it terminates its execution. Moreover, tasks are not allowed to communicate. A task can be started with some arguments. When it terminates, results are returned to the calling **ESTEREL** program. Tasks can also be killed, for example by a `watching` statement. Consider:

```
do
   exec Move (returnStatus)
            (initialPosition,
             goalPosition)
watching LIMIT_TIME
```

The task `Move` is started with `initialPosition` and `goalPosition` as arguments. It must terminate before `LIMIT_TIME` becomes present. If it is the case, the task sets the `returnStatus` result variable. Otherwise, the task `Move` is killed and `returnStatus` is not set. Note that a task can be killed and then started with new arguments in the same instant. The interface of the `exec` primitive is shown in Fig. 11.

IX. Conclusion

The **ESTEREL** language introduces a new programming style extremely natural for coding reactive systems. Separating a program into parallel components for better modularity and adding signals for synchronization incur no run-time overhead as they are compiled away.

ESTEREL is especially useful when an unique automaton is to be produced. In this case, we have a complete method: the **ESTEREL** high level program can be seen as the automaton specification; it can be graphically simulated and proved using verification systems; it can be translated into several sequential languages and executed with great efficiency. This is of special interest for real-time system *kernels*, when efficiency and proofs are required. Another use of **ESTEREL** that seems very promising is to produce code for electronic circuits.

However, **ESTEREL** can be useful in larger context when one has to produce several automata that must cooperate. It is the case for two main reasons: first, when an unique automaton would be too big an object. Second, when one has to mix synchronous and asynchronous approaches. The **ESTEREL** system gives a partial response in the first case with the `-cascade` option. However, work has to be done to extend the cascade method for more general cases (in other words, only a restricted form of separate compiling is available presently).

References

[1] C. André, L. Fancelli, "A mixed implementation of a real-time system," presented at Euromicro'90, Amsterdam, The Netherlands, 1990.

[2] G. Berry, *Programming a Digital Watch in Esterelv3*, Esterelv3 Programming Examples, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1989.

[3] ——, *Real time programming: special purpose or general purpose languages*, Information Processing 89, G.X. Ritter (Ed.), Elsevier Science Publishers B.V. (North Holland), 1989.

[4] G. Berry and G. Gonthier, "The **Esterel** synchronous programming language: Design, semantics, implementation," INRIA Rep. 842, to appear in Science of Computer Programming, 1988.

[5] ——, *Incremental Development of an HDLC Protocol in Esterelv3*, Esterelv3 Programming Examples, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1989.

[6] G. Boudol, V. Roy, R. de Simone, and D. Vergamini, "Process algebras and systems of communicating processes," in *Proc. of the Automatic Verification Methods for Finite State Systems,* LNCS 407, Springer-Verlag, 1990.

[7] P. Couronné, J. Plaice, and J-B. Saint, "The Esterel—Lustre Oc Portable Format," Tech. Rep., Ecole des Mines / INRIA, Sophia-Antipolis, 1988.

[8] E. M. Clarke, D. E. Long, and K. L. McMillan, *A Language for Compositional Specification and Verification of Finite State Hardware Controllers*, this volume.

[9] B. Espiau and E. Coste-Manière, "A synchronous approach for control sequencing in robotics application," in *Proc. IEEE Int. Workshop on Intelligent Motion Control,* Istanbul, Turkey, 1990.

[10] *Esterelv3 manuals*, Ecole des Mines, Centre de Mathématiques Appliquées, Sophia-Antipolis, 1988.

[11] G. M. Gherardi and J.P. Paris, "Manuel d'utilisation de Sahara," Internal Rep., ENSMP-CMA, 1990.

[12] G. Gonthier, "Sémantiques et Modèles d'Exécution des Langages Réactifs Synchrones; Application à Esterel," Thèse de Doctorat en Informatique, Univ. d'Orsay, 1988.

[13] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre, this volume.

[14] D. Harel, *Statecharts: A Visual Approach to Complex Systems, Science of Computer programming, 8–3, pp. 231–275, 1987.*

[15] D. Harel and A. Pnueli, *On the Development of Reactive Systems*, Logic and Models of Concurrent Systems, Springer-Verlag, pp. 477–498, 1985.

[16] V. Lecompte and F. Boussinot, "Une application de la programmation synchrone: le modem du minitel en Esterel," rapport C2A, 3, 1989.

[17] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire, *Programming Real Time Applications with Signal*, this volume.

[18] M. C. Mejia Olvera, *Contribution à la Conception d'un Réseau Local Temps Réel pour la Robotique*, Thèse de Docteur-Ingénieur, Université de Rennes, 1989.

[19] R. Milner, *A Calculus of Communicating Systems*, LNCS 92, Springer-Verlag, 1980.

[20] G. Murakami and R. Sethi, "Terminal call processing in Esterel," Res. Rep. 150, AT&T Bell Labs., 1990.

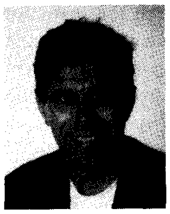
[21] J.P. Paris, "Communications synchrones asynchrones," Application à Esterel. thesis, to appear in 1991.

[22] G.D. Plotkin, "A structural approach to operational semantics," Lectures Notes, Aarhus Univ., 1981.

[23] V. Roy, "AUTOGRAPH, Un Outil de Visualisation pour les Calculs de Processus," Thèse de Doctorat en Informatique, Université de Nice, 1990.

[24] B. Stroustrup, *The C++ Programming Language.* Reading, MA: Addison-Wesley, 1986.

[25] D. Vergamini, "Verification by means of observational equivalence on automata," INRIA Rep. 501, 1986.



Frederic Boussinot received the Doctorate Thesis in 1981 from the Jussieu Paris 7 University.

From 1979 to 1984 he was a Research Engineer at Thomson CSF-LCR. In 1984 he joined the Applied Mathematics Center Laboratory of the Ecole des Mines de Paris as a Senior Researcher. His research and teaching interests include semantics of parallelism, reactive formalisms, and their implementation.

Robert de Simone received the Doctorate Thesis in 1984 from the Jussieu Paris 7 University.

In June 1984 he became a Fellow Researcher at INRIA-Sophis Antipolis and in 1990 he then became Research Director. He spent the academic year of 1986 on leave as a Teaching Assistant to the Ecole Normale Superieure de Paris. His research and teaching interests include algebraic models of parallelism, semantics, and verification.