SYNCHRONOUS PROGRAMMING OF REACTIVE SYSTEMS

N HALBWACHS

Rapporteur: A Tully

SYNCHRONOUS PROGRAMMING OF REACTIVE SYSTEMS: AN INTRODUCTION TO ESTEREL

G. Berry
P. Couronné
G. Gonthier

Ecole Nationale Supérieure des Mines de Paris (ENSMP) Centre de Mathématiques Appliquées — *Place Sophie Lafitte Sophia-Antipolis 06565 Valbonne - France*

Institut National de Recherche en Informatique et Automatique (INRIA) — *Route des Lucioles Sophia-Antipolis 06565 Valbonne - France*

Research Supported by the GRECO C3, CNRS

1. Introduction

Harel and Pnueli [26], introduced the name *Reactive Systems* to designate systems that react to repetitive inputs from their environment by sending themselves outputs to that environment. Real-time process controllers, mouse-keyboard interfaces [17], video games, communication protocols, and all control automata are examples of reactive systems. An important character of most reactive systems is their intrinsic *determinism*: a reactive system determines a sequence of output signals from a sequence of input signals in a unique way. This has to be opposed to the inherent non-determinism of other systems, such as operating systems. The importance of determinism is clear: deterministic systems are much easier to specify and to debug than non-deterministic ones.

Several tools are currently used to program or analyze reactive systems. We review them with respect to several criteria: easiness to write and maintain programs, run-time efficiency, existence of verification tools, and determinism.

1. *Assembly language programs*, generally written to run under some real-time operating system. Such programs can be efficient but are hard to structure and to maintain. Verification tools are elementary (monitors, debuggers).
2. *Automata*, also called *state machines*, hand-coded in some low-level or even high-level programming language. By nature, automata are very efficient and their transition time does not depend on their size. However, they are hard to design and to maintain: an apparently minor change in a specification can entail a major change in the resulting automaton (this is well-known to parser generator users); the risk of hand-coding errors is important, since a single action such as a signal emission as to be replicated in many transitions. Very good verification tools are available: for example the EMC [19] or CESAR [35] systems are practical temporal logic formulae verification systems, able to work on real-size automata; the ECRINS system [30, 37] can be used to perform bisimulation proofs à la Milner [31, 32]. The deterministic character of reactive systems is preserved when using deterministic automata.

Paper published in *Programming of Future Generation Computers*, (K. Fuchi and M. Nivat, editors), pp. 35-55, Elsevier Science Pub. B.V. (North Holland), 1988. INRIA Report 647.

3. *Communicating automata* represent a practical intermediate solution, used for example in the specification language SDL [22]. Simple enough automata are made into communicating processes using some communication primitive. The design and maintenance can be simpler than with standard automata. However program verification is much harder since the overall system becomes non-deterministic.

4. *Petri Nets* or Petri Nets based formalisms such as the *GRAFCET* [11]. These tools are also used in real applications. Compared to automata, they provide their user with a notion of process concurrency and a crude form of process synchronization. Having no hierarchical structure, Petri Nets are somewhat difficult to understand and to maintain. Some verification tools exist [34]. Petri Nets are asynchronous and non-deterministic, but GRAFCET nets are synchronous and deterministic: all active transitions must be fired "simultaneously".

5. *High-level parallel programming languages* such as ADA [1], MODULA [39], CSP [27], OCCAM [28]. They are based on notions of process and process communications. One can use them to write well-structured programs that are comparatively easy to maintain. The necessary run-time process and communication handling can induce a possibly important execution overhead. Few verification tools are currently available, because of the complexity of the semantics of the languages. Almost all parallel languages are non-deterministic. They usually support some additions to handle real-time problems, such as watchdogs on some "universal" time. These additions are also non-deterministic and therefore not well-defined semantically.

6. *Algebraic Calculi of Processes* such as CCS [31], SCCS [32], MEIJE[12], or ACP [3]. They permit the specification and analysis of deterministic or non-deterministic reactive systems. They lead to nice theories of process observations and process equivalences. However, they are still far from being programming tools. Some verification systems are in development [30].

In reactive systems programming, parallel languages certainly represent the best programming tools, while automata certainly have the lead for efficiency. But none of the above tools provides both good programming style and execution efficiency. Our purpose is to introduce the new class of *synchronous languages* that aim at all these qualities. We center the discussion on the language ESTEREL[6], which was historically the first synchronous language [4]. We also discuss the synchronous programming languages LUSTRE[18], SIGNAL[24], and the *Statecharts* [25] specification formalism. Similar but less developped synchrony ideas appear in the SML imperative language [15] and in some hardware description languages. They will not be presented here.

ESTEREL is an imperative language, LUSTRE and SIGNAL are data-flow languages, and the Statecharts are graphical hierarchical notations for automata. Although different in style, these synchronous formalisms share a common paradigm that was introduced in the early versions of ESTEREL: a program acts as a *synchronous history transformer* that produces a sequence of outputs synchronously with any sequence of inputs.

The synchrony hypothesis amounts to consider *ideal* reactive systems that react to external inputs by *instantly* updating their internal state and producing outputs. A synchronous program behaves as if its internal actions were carried out by an *infinitely fast machine*, whose actions *take no time*. Two actions are synchronous if and only if they belong to the same reaction; the sequencing of input/output events determines the behavior of a synchronous program, not the converse as in asynchronous languages. The program is inactive between its reactions; it simply waits for more inputs.

The synchrony hypothesis is of course an approximation of the reality. It is similar to the approximations used in newtonian mechanics to deal with instantaneous body interactions, in chemistry to deal with values such as the pH of a solution, or in standard electricity when one ignores the transmission speed of electrical currents. Within their application ranges, these hypothesis greatly simplify the study of physical systems: there is of course no need to invoke complex quantum mechanics equations to deal with billiard balls trajectories.

In our case, the synchrony hypothesis has the following practical consequences: it makes the programming languages deterministic, it simplifies the process interaction mechanisms, and it permits perfectly rigorous time manipulations. Time can be dealt with without encoutering an "Heisenberg effect": when measuring a delay, the measure itself does not introduce its own delay. On the practical side, this leads to a new programming style that is well-adapted to reactive systems programming. On the theoretical side, the semantics of synchronous languages is very clean, in particular for what concerns time manipulations.

As in physics or chemistry, one has to carefully evaluate the application range of the synchrony hypothesis. Given a real reactive system, one has to define a "satisfaction point" where a user perceives the system as synchronous. For example, a video game player certainly wants his game to react as fast as possible, that is fast enough to consider any response as instantaneous compared to its own reactions; a telephone is good if the delays introduced by packet switching are not perceived by the user. In practice, the reaction to an input is viewed as taking time but being *uninterruptible*: no new input is taken into account during it. An implementation satisfies the synchrony hypothesis if the reaction time is always sufficiently fast not to cause input overrun. We then make the following observations:

(i) As far as *behavioral specifications* are concerned, performance considerations must be kept on a lower level than specification and design considerations. The synchrony hypothesis is clearly justified as soon as it helps improving the programming style.

(ii) Many reactive systems don't really need execution speed, but rather *safety*. For controlling a lift or a subway, the time constants are large and performance is comparatively easy to achieve. The hypothesis is again justified as soon as it makes specification and programming simpler and safer.

(iii) More importantly and also more surprisingly, synchronous languages tend to be even *more efficient* than conventional parallel ones: their compilers perform process scheduling and communication *at compile time* and transform parallel programs into equivalent sequential finite automata (the ESTEREL compiler [7] realizes this transformation for ESTEREL programs). In most cases, the obtained object code is as efficient as carefully hand-written assembly code, and no state explosion occurs. Moreover, as with hand-written automata, the worst case reaction time is measurable, and many verification tools can be used for proving program properties.

Notice the close connection between synchronous languages compilers and parser generators: BNF is a high-level way of specifying context-free languages; a parser generator such as YACC [29] takes a BNF specification as input and produces an efficient automaton for analyzing input words. There are indeed strong relations between the algorithms used in parser or scanner generators and the algorithms used for synchronous languages.

In the body of the paper, we first introduce the synchrony hypothesis in more details. We then present the ESTEREL primitives and their naive semantics. We give some examples to illustrate the ESTEREL programming style. We introduce the notion of *causal correctness*, which is analogous to the notion of deadlock-freeness for asynchronous programs, but is statically checkable. We show how to transform an ESTEREL program into an automaton, using the mathematical semantics of the language and a variant of Brzozowski's algorithm. [16]. We discuss the quality of the obtained compiled code. We finally give an overview of other synchronous languages and explain how they view the synchrony hypothesis.

The reader will find more details on ESTEREL in [6]. Detailed examples of ESTEREL reactive system programming are presented in [8,9].

2. The Multiform Discrete Time Model

2.1. Signals and sensors

An ESTEREL program communicates with its environment via *signals* and *sensors*. Signals are used both as inputs and outputs, while sensors are used only as inputs. Signals can convey values; sensors always do.

For example a train controller can receive a signal every millisecond, a signal every wheel revolution, track signals conveying positional informations, and signals coming from the operator's keyboard; it can use sensors to measure the external temperature; it can emit power commands to the engines and brakes. A submodule of this controller may receive and emit additional software signals to communicate and synchronize with other submodules.

As in most parallel languages, all the signals and sensors are treated as messages, regardless of their hardware or software origin. They are identified by *names* like S, S1, etc. The notation S(v) expresses that S conveys the value v.

2.2. Broadcasting as the communication primitive

Asynchronous languages use several kinds of process communication mechanisms: simple rendez-vous in CSP or OCCAM, queued rendez-vous in ADA, and asynchronous queues in data-flow languages. All these conceptual mechanisms are close to implementation mechanisms, and communication is limited to be one-to-one.

On the contrary, in synchronous languages, signals and are assumed to be *broadcast* among processes *. One can think of programs as using radio waves as a communication medium, each signal being represented by a frequency.

Two kinds of informations are broadcast on the waves: *values* that are permanent, and *signal tops* that are intermittent. A sensor has a value, but no signal top. A pure signal has a signal top, but no value. A valued signal has both, and a value change is always synchronous with a signal top (hence the signal top is used to broadcast and detect value changes; there is no way to detect value changes for sensors).

In ESTEREL, signal and sensor values are available in *expressions*, (with the "?S" primitive, see §3.2.), while signal tops act as *control information* to be handled by the ESTEREL control structures (the **present** and **watching** statements below). A sensor is simply a degenerate signal for which no control information is available. From now on, we shall therefore include sensors in signals.

As we shall see in the next sections, broadcasting is excellent for modular programming and is a basis of the ESTEREL programming style. In asynchronous languages, broadcasting is usually not available: it must be done at run-time and is known to be expensive. Because of the synchrony hypothesis, we shall be able to perform broadcasting at *compile time* and to produce code that simply accesses a shared memory. This is the main task of the ESTEREL compiler.

2.3. The "absolute" time

In addition to external signals, most parallel languages introduce particular mechanisms to handle an implicitly broadcast *absolute time*. This time has a specific name (say the SECOND) and

* The choice of broadcasting as the communication primitive is explicit in ESTEREL, and implicit in LUSTRE, SIGNAL, and the Statecharts.

is manipulated by specific instructions (say delays and watchdogs). In asynchronous languages, the use of an absolute time is necessary to establish relations between the internal computation time of a program and the occurrence of external signals. In synchronous languages this problem disappears together with the notion of computation time. The `SECOND` has no particular role and can be treated as any other input signal, as it is automatically broadcast.

A small example shows the superiority of synchronous languages over asynchronous ones when dealing with time. Assume that the basic universal time unit is `MILLISECOND`. Then in any "real-time" language one can derive `SECOND` by writing a statement like:

```
every 1000 MILLISECOND do
   emit SECOND
end
```

However, such a statement behaves very differently in asynchronous and synchronous language:

- In asynchronous languages, `SECOND` acts as a normal signal and is *not* broadcast, unlike the basic `MILLISECOND` signal; moreover, because of asynchrony, `SECOND` is *never* synchronous with an occurrence of `MILLISECOND`. The "duration" of a `SECOND` can not be defined.
- In synchronous languages, `SECOND` is *broadcast* every 1000 `MILLISECOND`, is *simultaneous* with `MILLISECOND`, and a `SECOND` lasts *exactly* 1000 `MILLISECOND`.

2.4. Time is multiform

ESTEREL can treat the physical time as a standard signal. Conversely, it can also treat *any signal* as an independent "time unit", so that the time manipulation primitives can be uniformly used for all signals. In ESTEREL, one can write statements such as

```
await 2 METER;
do
   <task>
watching 100 WHEEL_REVOLUTION
```

This notion of *multiform time* will be detailed in section 4. It is one of the strengths of the ESTEREL programming style.

2.5. Simultaneity, events, and complete events

In synchronous languages, output is synchronous to input: we have to axiomatize a primitive notion of *simultaneity*. We call *event* the occurrence of an arbitrary number of simultaneous signals. An event where a signal `S1` appears together with a signal `S2` that carries a value v will be denoted by `S1` $\texttt{S2}(v)$. We denote events by $E, E_1, \ldots$.

The *synchronous product* $E = E_1 * E_2$ of two events E_1 and E_2 is the event resulting from their simultaneous occurrence. If the events involve only distinct signals, there is no problem in combining them: if $E_1 = \texttt{S1}\ \texttt{S2}(v_2)$ and $E_2 = \texttt{S3}(v_3)$, then $E = \texttt{S1}\ \texttt{S2}(v_2)\ \texttt{S3}(v_3)$. In ESTEREL, we also allow to combine events that contain the same signal `S` with different values (such events will be produced by executing simultaneously several `emit` statements). We call this phenomenon a *collision*. Assume that E_1 and E_2 both contain an occurrence of `S`, with respective values v_1 and v_2. One has to define the broadcast value v of `S` in $E = E_1 * E_2$. Following Milner [32] we associate an associative commutative operation $*_S$ with the signal `S`, and we set $v = v_1 *_S v_2$. Therefore one has

$$\texttt{S}(v_1) *_S \texttt{S}(v_2) = \texttt{S}(v_1 *_S v_2)$$

The product of two events is then defined componentwise on signals. The choice of the composition operation is left to the programmer. Here are some examples:

- In Ethernet-like local networks, signal broadcasting is physically realized on a cable. A special value NAK represents the collision of two messages. One sets $v_1 * v_2 =$ NAK for all v_1, v_2.
- In a request handling mechanism, several processes can request the same resource simultaneously, say by broadcasting their name. A natural choice is to take as result the set of the names of the processes that simultaneoulsy require the resource.
- In the digital watch programmed in [9], a timekeeper, a stopwatch, and an alarm can operate a beeper. The timekeeper beeps once a second, the stopwatch beeps twice a second, and the alarm beeps four times a second. If some of the units beep together, the resulting number of beeps per second is obtained by adding the individual numbers. Hence seven beeps per second occur when the three units beep together. We simply define a BEEP signal that carries an integer representing the required number of beeps per second and choose integer addition as the $*$ operation.

The events defined so far only contain information about emitted signals. *Complete events* also contain the remaining informations about non-emitted signals and sensors; Assume that a signal S1 is emitted with value v_1, while a signal S2 is not emitted but has currently value v_2 and a sensor S3 has currently value v_3; this corresponds to a complete event written $\hat{E} = \mathtt{S1}(v_1)\, \neg\mathtt{S2}(v_2)\, \neg\mathtt{S3}(v_3)$.

2.6. Histories

Having defined the notion of simultaneity, we are left with the notion of *succession of events*, which is handled by defining histories. An *history* $H = \hat{E}_1, \hat{E}_2, \ldots, \hat{E}_n, \ldots$ is a sequence of complete events that is required to be consistent in the following sense: for any valued signal S, if $\neg\mathtt{S}(v)$ appears in $\hat{E}_n$ and if S appears negatively in $\hat{E}_{n+1}$, then the S-component of $\hat{E}_{n+1}$ is also $\neg\mathtt{S}(v)$; this ensures that values only change synchronously with signal tops.

Remember finally that our reactive systems have infinitely fast reactions: they react only when receiving input signals, and therefore *nothing* happens "between" input events. The system has no "internal clock".

3. The ESTEREL programming primitives

We don't give here a precise definition of ESTEREL (see [6,7]), but we introduce enough of the language to be able to treat illustrative examples.

The basic programming unit is the *module*; a module contains a *declaration part* and a *statement*.

ESTEREL is not a general-purpose programming languages. The types, constants, functions, and procedures are just declared as abstract names in the declaration part; they are supposed to be implemented in some *host language*, say C or ADA (the ESTEREL compilers can produce object codes for different host languages). Therefore an application is programmed in two parts: the ESTEREL part that deals with signals, and the auxiliary part that deals with standard computations. The functions and procedures of the auxiliary part have no access to signals.

3.1. Declarations

In the declaration part, one declares the types, constants, functions, and procedures used by the module (and defined in the host language); one then declares the signals and sensors that define the module's interface. Here is a possible declaration part of a TIMER module:

```
module TIMER :
type TIME;
constant INITIAL_TIME : TIME;
procedure INCREMENT_TIME (TIME) ();
input SECOND,
      RESET_COMMAND;
output TIMER_VALUE (TIME),
       BEEP (combine integer with PLUS);
```

The output signal `TIMER_VALUE` has type `TIME`, and no collisions are allowed for it; this is the default. We allow collisions to occur for the integer output signal `BEEP` and we use the addition function `PLUS` to compute the combined value; this is declared using the `combine` keyword.

The declaration of a procedure such as `INCREMENT_TIME` involves two type lists: the first list types arguments passed by reference, the second list types arguments passed by value (it is empty here).

3.2. Expressions

The expressions are classically built from variables, constants, and function calls. A special expression "`?S`" gives access to the current value of a signal or sensor `S`. Its type is the type of the signal or sensor. (A similar expression "`??S`" gives access to the current value of a valued exception, see §3.4.)

3.3. Statements

There are two kinds of statements: *primitive statements* and *derived statements* that are defined in term of primitive statements. The mathematical semantics is defined only for primitive statements. The derived statements act as macros and can be expanded into primitive ones. The synchrony hypothesis is *necessary* to ensure the correctness of the expansions, in other words to ensure that the derived instructions do *exactly* what we intend them to do.

The primitive statements are themselves divided into two groups: classical basic imperative statements and temporal statements that deal with signals.

3.3.1. Basic imperative statements

Here is the list of the basic imperative statements:

`nothing`	*dummy statement*
`halt`	*halting statement*
`<var> := <exp>`	*assignment statement*
`call <id> (<varlist>)(<arglist>)`	*external procedure call*
`<stat> ; <stat>`	*sequence*
`if <exp> then <stat> else <stat> end`	*conditional*
`loop <stat> end`	*infinite loop*
`<stat> || <stat>`	*parallel statement*
`trap <id> in <stat> end`	*trap definition*
`exit <id>`	*exit from trap*
`var <var-decls> in <stat> end`	*local variable declaration*
`signal <signal-decls> in <stat> end`	*local signal declaration*

There are no shared variables: if a variable is updated in one branch of a parallel statement, it cannot be read or written in the other branches.

One has to remember that the execution machine is conceptually *infinitely fast*. Therefore `nothing` does nothing *in no time*, assignments and external procedure calls are instantaneous, the second statement of a sequence is started exactly when the first statement terminates, and the branches of a parallel statement start simultaneously (a parallel terminates synchronously with the last termination of its branches). Hence when a parallel statement is started, its branches work in the same signal environment.

The `trap-exit` mechanism is a classical escape mechanism: a `trap` statement defines a block that is instantly exited when a corresponding `exit` statement is executed*. This mechanism is the most powerful control mechanism in ESTEREL. It extends to a general exception handling facility, see [7] and §3.4. below.

Since the execution machine acts only upon reception of input events, any statement starts or terminates synchronously with some input event. When discussing the behavior of a statement, we shall call *current event* the event that starts the statement.

Although statements are executed simultaneously, they are executed *in the right order*. Hence a sequence

```
I:=0;
I:=I+1
```

yields instantly `I=1`. Only finitely many statements can be executed simultaneously. One imposes a staically checked finiteness constraint to forbid loops like

```
I:=0;
loop I:=I+1 end
```

that have of course no semantics in our instantaneous universe.

3.3.2. Temporal statements and signal handling

All statements described so far "take no time", besides `halt` that never terminates. We now describe the *temporal statements* that handle signals and can take time.

The signals can be either emitted by the program's environment or by the program itself. To emit a signal `S` with value that of an expression `<exp>`, one writes

```
emit S(<exp>)
```

If `S` is a pure signal, the expression is of course omitted. An emission is instantaneous. If several emissions occur simultaneously, the values are combined as described in section 2.5.

For signal reception, there are two primitive statements. The first one tests for the presence of a signal in the current event:

```
present S then <stat1> else <stat2> end
```

The semantics is clear: if `S` is present in the current event, then `<stat1>` is instantly started. Otherwise `<stat2>` is instantly started.

The second statement is the most important ESTEREL construct; it is called the *watchdog* or *time guard*; it has the form

```
do
   <stat>
watching <occ>
```

* If several blocks are simultaneously exited, the effect is to instantly exit the outermost one

where `<stat>` is any statement and where `<occ>` is an *occurrence* of a signal. An occurrence is either a signal name (say `SECOND`) or a signal name preceded by a count factor (say `3 METER`).

A `watching` statement defines a time limit for the execution of its body. The time limit is given by the occurrence `<occ>`. If `<occ>` has the form `S`, the time limit is the first event in the strict future of the current event to contain an occurrence of the signal `S`; similarly, for an occurrence `n S`, the time limit is the n-th event in the strict future to contains an `S`.

The body of the `watching` statement is started synchronously with the `watching` statement; it is executed up to the time limit *excluded*:

- If the body terminates strictly before the limit, the whole `watching` statement terminates synchronously;
- If the body is not terminated when the time limit occurs, the body is instantly killed *without being executed at that time* and the `watching` statement instantly terminates.

Notice that the nesting of `watching` statements establishes natural *preemption relations* between the corresponding signals. Consider the following example:

```
do
   do
      <stat1>
   watching S1;
   <stat2>
watching S2
```

By definition of the semantics, if `S1` and `S2` occur simultaneously, then the outermost `watching` statement is terminated, and `<stat2>` is not executed. Hence `S2` preempts a simultaneous `S1`.

3.4. Derived statements

Many useful temporal statements can be derived from primitive ones. For example one writes

```
await <occ>
```

instead of

```
do halt watching <occ>
```

One writes

```
do <statement> upto <occ>
```

instead of

```
do
   <statement>;
   halt
watching <occ>
```

The `upto` statement differs from the `watching` statement by the fact that it terminates only on `<occ>`, not if its body terminates (Conversely, one could define `watching` from `upto`; in [5], `upto` was taken as a primitive).

It is often useful to add a `timeout` clause to a watchdog; this clause is executed if the time limit is reached before termination of the body:

```
do
   <stat1>
watching <occ>
timeout <stat2> end
```

abbreviates

```
trap T in
   do
      <stat1>;
      exit T
   watching <occ>;
   <stat2>
end
```

If `<stat>` terminates strictly before `<occ>`, one instantly exits the enclosing `trap`, thus skipping the timeout clause.

Temporal loops are useful derived statements. For example one writes

```
loop
   <stat>
each 3 METER
```

instead of

```
loop
   do
      <stat>
   upto 3 METER
end
```

and one writes

```
every 5 SECOND do
  <stat>
end
```

to abbreviate

```
await 5 SECOND;
loop
  <stat>
each 5 SECOND
```

In a `loop ... each` statement, the body starts immediately and is restarted on every occurrence of `<occ>`; in an `every` statement the body starts only on the first occurrence of `<occ>`.

Two other derived instructions are particularly useful. The first one is the *signal selection* or *multiple await*. The syntax is

```
await
   case <occ1> do <stat1>
   case <occ2> do <stat2>
   ...
   case <occn> do <statn>
end
```

Unlike similar statements in asynchronous languages, our selection is *deterministic*. The first occurrence satisfied determines the statement to execute; if several occurrences are satisfied simultaneously, only the statement corresponding to the first such occurrence in the list is executed (therefore the order in the list establishes a priority relation between occurrences). The expansion is not given here, see [7].

The last important derived statement is the *exception handling* statement, that generalizes the `trap` statement. Here is an example:

```
trap ALARM, FOUND(integer) in
   ...
   ... exit ALARM
   ||
   ... exit FOUND(VALUE+1)
   ...
handle ALARM do ...
handle FOUND do ... X:=??FOUND+5; ...
end
```

An exception acts both as a classical trap and as a signal that can carry values. If the body executes an `exit`, it is instantly terminated and the corresponding handler is instantly started. If the body executes several exits simultaneously, then the corresponding handlers are started in parallel. The whole construct terminates when all the started handlers have terminated. In a handler, the special expression "??S" has value that of the exit.

When `trap` blocks are nested, the outermost ones preempt the innermost ones; for example, in

```
trap T1 in
   trap T2 in
        exit T1
      ||
        exit T2
   handle T1 do <inst1>
   end;
   <inst2>
handle T2 do <inst3>
end
```

the exceptions `T1` and `T2` are simultaneously raised. Then `T2` preempts `T1`: `<inst3>` is executed, while `<inst1>` and `<inst2>` are not.

4. The ESTEREL programming style

We briefly illustrate the main aspects of the ESTEREL programming style: the use of multiple time units, the use of broadcasting, and the use of signal simultaneity. A more extensive discussion can be found in [8,9].

4.1. Using signals as time units

We already mentioned that time is multiform in ESTEREL: any signal is viewed as defining a "time unit". A good illustration of the induced programming style appears in the reflex game program presented in [8]. Let us first realize the following specification: "Wait for a hit on a `READY` button within a time limit of `10 SECOND`; in case of timeout, emit an `ALARM`; while waiting, any hit on the `STOP` button should ring a `BELL`":

```
do
   do
      every STOP do emit RING_BELL end
   upto READY
watching 10 SECOND
timeout emit ALARM end
```

(Here `upto READY` is equivalent to `watching READY`; we prefer to use `upto` whenever we are not interested in the termination of the body). Let us now realizes the following specification: "Wait for 10 `SECOND`; if `STOP` is hit during that time, terminate and emit an `ALARM`; while waiting, any hit on `READY` should ring the `BELL`":

```
do
   do
      every READY do emit RING_BELL
   upto 10 SECOND
watching STOP
timeout emit ALARM end
```

In this example, the second specification is *dual* to the first one; it can be read as "Wait for 10 SECOND with a time limit of STOP; in case of timeout, emit an alarm; while waiting ...".

This shows how useful it is to use watchdogs for *arbitrary* signals and to nest watchdogs on different signals. We go further in the same direction by programming the following specification, to be used as a training program for a mile runner: "Run two laps in the following way: run slowly 100 meters, then, during 20 seconds, jump high and breath deeply upon every step, then finish the lap by running as fast as possible; end the training session by taking a shower"

```
do
   loop
      do RUN_SLOWLY upto 100 METER;
      do
         every STEP do
               JUMP_HIGH
            ||
               BREATHE_DEEPLY
         end
      upto 20 SECOND;
      FULL_SPEED
   each LAP
upto 2 LAP;
TAKE_A_SHOWER
```

The identifiers JUMP_HIGH, BREATHE_DEEPLY, FULL_SPEED, and TAKE_A_SHOWER refer to submodules that can themselves synchronize on heart beats. Let us make the following remarks:

- All upto constructs control statements that would otherwise never terminate.
- The runner jumps only if a lap is longer than 100 meters. Otherwise the corresponding statement is never executed, since it is killed by the enclosing loop ... each LAP.
- Similarly, the runner runs full speed only if the lap is not finished after "100 meters plus 20 seconds" jumping.
- The overall program lasts *exactly* two laps plus the duration of the shower.

Such a simple program is not easy to write in classical languages (we leave this to the reader).

4.2. The use of broadcasting

Broadcasting simplifies process communication and improves modularity: when it emits a signal, a process doesn't need to know who is listening to that signal; conversely, when a process receives a signal, it doesn't need to know the emitter(s).

We illustrate this in the wristwatch example described in detail in [9]. A wristwatch is an excellent example of reactive systems; it is comparatively small, but has many features found in other systems: folding numerous commands into few buttons by using *command modes*, showing numerous data in few displays using display modes, establishing communications and instantaneous dialogues between submodules. The wristwatch programmed in [9] has five submodules: a WATCH that acts as a regular timekeeper, a STOPWATCH, an ALARM, a BUTTON_INTERPRETER that interprets wristwatch buttons as commands directed to the other modules according to the current command mode, and a DISPLAY_HANDLER that handles the various displays. Broadcasting makes life easier in several places:

- The external signal SECOND is automatically broadcast to all the modules that need it.
- Hitting a particular button in a particular mode provokes the toggling from 24H to 12H AM/PM time display mode. This concerns the watch and the alarm. The button interpreter broadcasts a message TOGGLE_24H_MODE_COMMAND, without worrying about who is expecting this message. Adding a second alarm would not modify the corresponding code.
- The timekeeper broadcasts a WATCH_TIME signal whenever its internal time is modified. This signal is used by both the alarm and display handler. Adding a second alarm can be done without any modification of the WATCH and ALARM modules.

4.3. Simultaneity and instantaneous dialogues

The synchrony hypothesis allows us to establish a new form of process communication, the *instantaneous dialogue.*

A typical example appears in the wristwatch code [9], more precisely in the body of the stopwatch; it will be abstracted here. An instantaneous dialogue appears whenever the behavior of a process P depends on some property of the internal state of another process P'. For simplification, assume that P' is a flip-flop on some signal FLIP_FLOP_COMMAND and that P must perform <stat1> if P' in in the flip state and <stat2> otherwise. Then one introduces two signals ARE_YOU_FLIP and YES_I_AM_FLIP and one writes P' as follows:

```
loop
   do
         loop
            emit YES_I_AM_FLIP
         each ARE_YOU_FLIP
      ||
         <flip code>
   upto FLIP_FLOP_COMMAND;
   do
      <flop code>
   upto FLIP_FLOP_COMMAND
end
```

Now the intended behavior of P is ensured by the following code:

```
emit ARE_YOU_FLIP;
present YES_I_AM_FLIP then
   <stat1>
else
   <stat2>
end
```

The signal ARE_YOU_FLIP emitted by P provokes an instantaneous reply YES_I_AM_FLIP from P' if and only if in flip mode.

This method is easy to extend since P only cares for a reply to its question and doesn't need to know much about the structure of P' (this is not the case for the method used in the Statecharts [25] to solve the same problem: there P must refer to the exact internal name of the state of P').

5. Causal correctness of ESTEREL programs

The synchrony hypothesis can generate temporal paradoxes, which are analogous to short-circuits or oscillations in electronics and to deadlocks in asynchronous parallel programs. Here is a first type of paradox:

```
signal S in
   present S then
      nothing
   else
      emit S
   end
end
```

The local signal S should be emitted if and only if not present, which is clearly a nonsense. This programs behaves more or less like a "not" gate with output plugged on input.

The second example is analogous to a short-circuit, or more precisely to a positive feedback effect. Consider the purely instantaneous program:

```
signal S (combine integer with PLUS) in
   emit S(0);
   await S(?X+1)
end
```

the signal S can have simultaneous emitters, the values being added (see section 2.5). Since the reception and emission of S are simultaneous, every reception of n should provoke the immediate emission of $n + 1$: this is clearly a nonsense. The short-circuit is initiated by emitting 0.

In both example, the statement had no possible behavior. Here is a "non-deterministic" case where infinitely many behaviors are possible:

```
signal S(integer) in
   emit S(?S)
end
```

Any integer value can be considered as the value of S.

Generally speaking, temporal paradoxes appear as soon as the input of a program depends on its output. They are statically detected by the ESTEREL compilers. This is an advantage over asynchronous languages where no deadlock detection is possible at compile-time*. See [6, 23] for details.

6. Compiling an ESTEREL program into a finite automaton

6.1. Mathematical semantics and simulation

As a programming language, ESTEREL is mathematically well-defined. Its semantics is given by a set of rewrite rules à la Plotkin [33]. The rules determine the behavior of a program given any input event. This behavior has to be defined in a circular way because of instantaneous signal broadcasting: the output of a program must be combined with its input in order to determine the event in which the program works, which in turn determines the output. We give no more detail here, see [5,21,23].

We first show how to use the mathematical semantics to build a simulator of the language. To simplify the discussion, we treat the case of pure signal programs, that is of programs that contain no valued signals or variables; we indicate how to extend the results to the general case.

For any causally correct program P and input event $E^{\dagger}$, the semantic rules uniquely determine the output event E' and a new ESTEREL program P' called *the derivative of P by E*. This derivative

* The ESTEREL v2 compilers sometimes reject programs that do not contain paradoxes. This problem disappears in the ESTEREL v3 compiler.

† complete events must be used in place of events when dealing with general programs; complete events are obviously useless for pure signal programs.

represents "P after E": if P produces the output history $E', E'_1, E'_2, \ldots, E'_n, \ldots$ when applied to the input history $E, E_1, E_2, \ldots, E_n \ldots$, then P' produces the output history $E'_1, E'_2, \ldots, E'_n, \ldots$ when applied to the input history $E_1, E_2, \ldots, E_n \ldots$ We write

$$P \xrightarrow[E]{E'} P' = \frac{\partial P}{\partial E}$$

This is enough for building an ESTEREL simulator: given an input history, one constructs the output events and the new derivatives step by step. This technique is used in the ESTEREL v2 system (the hard part of the system being of course the computation of the derivative, see [5,21,23]).

6.2. Compiling an ESTEREL program into a finite automaton

Since any pure signal program has a finite number of input signals, the number of its possible input events is finite and the number of its derivatives by input events is also finite. Hence we can formally compute all the derivatives of P by all possible input events. For compiling programs into automata, the idea is to iterate this process and to explore completely the space state of the program. For this, let us extend the notion of derivative to arbitrary histories. Given a finite history $H = E_1, E_2, \ldots, E_n$, we set

$$\frac{\partial P}{\partial H} = \frac{\partial\left(\frac{\partial P}{\partial E_1, \ldots, E_{n-1}}\right)}{\partial E_n}$$

and, if ε denotes the empty history, we set

$$\frac{\partial P}{\partial \varepsilon} = P$$

Then we are able to prove the following result, which is analogous to Brzozowski's result on derivatives of regular expressions [16]:

THEOREM: *Any ESTEREL program has only a finite number of derivatives: the set* $\{\frac{\partial P}{\partial H} \mid H$ *an history*$\}$ *is finite.*

Hence we can construct the finite graph of all possibles transitions of P and of its derivatives. This graph is nothing but a finite automaton whose behavior is identical to that of P. Once the graph is constructed, we can of course remove the derivatives associated with the vertices, replace them by state numbers, and implement the obtained automaton in any classical programming language (the ESTEREL v2 compilers presently produces C code).

We can apply a similar process to general programs that handle valued signals and variables. At compile-time, the memory actions to execute are simply kept formal when computing a transition; because of conditionals, a transition is now a tree whose nodes are elementary memory actions (assignments, procedure calls, tests). At run time, the resulting automaton handles a memory by executing these elementary actions.

6.3. Quality of the compiled code

The compiling technique described above calls for several remarks concerning the efficiency of the compiling process, the size of the resulting automaton, the efficiency of the code produced for each transition, the validity of the synchrony hypothesis, and various problems about separate compilation and code distribution.

6.3.1. Efficiency of the compiling process

The presented compiling process is based on the formal computation of derivatives. As implemented in the ESTEREL v2 compilers, this process is fairly expensive in time and space since all the derivatives must be explicitely computed and kept in memory for comparisons; it is however practical: compiling the complete wristwatch described in [9] requires 2mn on a VAX 780, using 2 mega-bytes of memory (the system is written in Le_Lisp [20]).

In the ESTEREL v3 system that we are presently implementing, new algorithms avoid the explicit computation of derivatives. See [10] for a description of similar algorithms on regular expressions. The gain is important both in time and space (say a factor of 10 for both).

6.3.2. Size of the resulting automaton

The size of the resulting automaton determines the space occupied by the generated code. In asynchronous formalisms, it is well known that a complete exploration of the state space results in a rapid blow-up, for internal transitions of the system generate states. In synchronous languages, the situation is different: a single state transition can correspond to a complex behavior of a program, where *many* conceptually simultaneous internal actions take place. A state is really an input-output state, and no state is generated only by internal actions. However blow-up can still occur for two reasons:

- For any state, one has to compute the transitions corresponding to all possible input events. But n input signals generate 2^n distinct input events, since we have to handle the possible simultaneity of input signals. ESTEREL introduces a notion of *input signal relation* in order to break down to a more reasonable size. There are two kinds of relations:

  ▹ *Exclusion relations*, which tell that signals are exclusive: If we write `S1#S2#S3`, we require `S1`, `S2`, and `S3` to be pairwise incompatible. Input events such as `S1 S2` are then forbidden. Incompatibility relations suppose a serialization of the corresponding input signals by the underlying operating system, which is not a restriction in most cases.

  ▹ *Synchrony relations*, which on the contrary force input signals to be synchronous. If a watch receives signals for both the second and the hundredth of second, it is natural to require the second to be synchronous with some hundredth. One then writes `SECOND => HUNDREDTH`.

  In practice it is wise to declare as many relations as possible for input signals. For example, if all input signals are incompatible, the number of input events is n instead of 2^n.

- The number of actual input-output states can still be enormous: As for regular expressions, it is not hard to write programs causing exponential blowups. However practical programs tend to be really tractable. The wristwatch of [9] has no more than 41 states, and many other significant examples yield automata that have between 10 and 100 states. As with any other system, it is essential to understand what to put in the program's control and what to put in data. In a lift with 32 independent call buttons, the 2^{32} states are more efficiently stored in the 32 bits of a single memory word than in the 2^{32} states of an automaton!

It seems that the derivative algorithms tends to construct directly *minimal automata*, at least in practical cases (it is easy to construct ad-hoc counter-examples); for this reason, the ESTEREL systems don't embody a minimization algorithm on the compiled automata. This important phenomenon is not yet well-understood.

Let us finally mention that we use a compact representation of automata using byte code sequences in order to reduce the generated code size. For example the wristwatch's automaton occupies 2,500 code bytes.

As a conclusion, we think that the presented compiling technique is of practical use as soon as the input relations are carefully declared.

6.3.3. Efficiency of the transition code

The efficiency of the transition code determines the speed at which the generated code can react to input signals. This is where the results are the most spectacular. The transition code is purely sequential; it almost only contains actions that are *necessary* at execution time (such as assignments or emissions of output signals). There is no overhead for process handling and process communication. Pure communications generate *no code at all*: this is clearly the best way of being instantaneous. Value broadcasting generates a minimal number of assignments to global variables. For short, the process communication is done completely *at compile time* and only inevitable actions are deferred to run-time.

Although not instantaneous, the transition code is therefore *minimal* and *as fast as it can be*. Moreover, its speed is *measurable* given any particular processor: Hence the validity of the synchrony hypothesis can be precisely checked for any precise application.

6.3.4. Separate compilation and code distribution

The compiled code automaton form has two drawbacks:

- Being sequential, the code must be run on a single processor: no distribution is possible. When distribution is needed, one can however still use ESTEREL for writing the individually synchronous parts of the system and link these synchronous parts using procedure calls or asynchronous communication primitives (provided that the concerned submodules have no infinititely fast dialogues). This seems to be a reasonable tradeoff in practice: ESTEREL is good for synchronous applications, not for asynchronous ones. As an example, we present in [9] an implementation of our wristwatch with five communicating automata. The ESTEREL v3 system will provide its user with automatic tools for automata connection and code distribution.
- Separate compilation of modules is uneasy, although possible (see [36]). More precisely, it is not easy to use the compiled automaton form of a module when this module is used as a submodule. Some form of separate compilation exists in ESTEREL v3, but it concerns an intermediate code that is closer to the source code than to the automaton. Notice however that the efficient compiling algorithms are *global*, as are the efficient algorithms that transform regular expressions into automata [10]. Separate compilation would not necessarily lead to a gain in time efficiency.

6.3.5. Proving properties of programs

The translation of ESTEREL programs to automata has a major advantage: it permits to perform *automatic proofs* of properties of the resulting automata. There We import the work done by other researchers. We have interfaced the ESTEREL systems with the EMC system [19] that allows its user to prove or disprove temporal logic formulae; we are also performing experiments with the ECRINS system developed in our group [30]; this system is based on algebraic calculi of processes à la Milner [32, 12, 37, 38].

7. Brief overview of other synchronous languages

7.1. LUSTRE and SIGNAL

ESTEREL is an imperative language in which the temporal statements deal with events. On the contrary, LUSTRE [18] and SIGNAL [24] are synchronous data-flow languages that deal directly with histories. We describe LUSTRE; SIGNAL is conceptually similar.

A LUSTRE variable `X` denotes a sequence of values of a given type. The indices of the sequence represent an "universal time". All variables are synchronous in the following sense: the n-th components X_n and Y_n of two sequences are assumed to be simultaneously available. An equation

```
Z = X+Y
```

defines a sequence `Z` such that $Z_n = X_n + Y_n$ for all n. Therefore the primitive operations take no time as in ESTEREL.

A program is a set of equations such that each variable has exactly one defining equation. Any variable can appear in the right-hand side of any equation, as in

```
Z = X+Y
T = X+Z
```

Here `X` appears in the definitions of `Z` and `T`; this is a form of broadcasting. There are short-circuits and oscillations, as in

```
X = not X
Y = Z+1
Z = Y+1
```

The main temporal operators are `pre(X)` that defines the sequence `nil`$.X_0, X_1, \ldots, X_n$, where `nil` is an undefined value, and `X->Y` that defines the sequence $X_0, Y_1, \ldots, Y_n$. The following program counts the number of times a boolean variable `X` changes value:

```
COUNT = 0 -> if X=pre(X) then pre(COUNT) else pre(COUNT)+1 fi
```

There are also primitives to deal with multiple time units. A *clock* is a boolean variable; intuitively the corresponding signal is present when the clock is true. If a variable `X` is conceptually synchronous with a clock `C`, an operator "`X when C`" brings `X` back to the universal time. Finally a `current` operator provides an asynchronous access to a variable's value (as our "`?`" operator, that was absent in the early versions of ESTEREL and introduced after the LUSTRE `current` operator). We give no more detail here.

LUSTRE is a functional language that satisfies the substitution property: In the right-hand side of any equation, any variable may be replaced by its definition. The ordering of equations is not significant.

SIGNAL differs mainly by its *clock calculus* that permits to leave the clocking of variables implicit; its syntax is similar to notations commonly used in signal processing.

Altogether, ESTEREL, LUSTRE, and SIGNAL are very close in spirit. The main difference is the induced programming style: LUSTRE and SIGNAL are easier for applications that have a simple control structure, such as signal processing: a program can then be very close to the original set of mathematical equations that specify a problem; on the contrary, ESTEREL is more suited to applications having many control states, such as the wristwatch mentioned above: when a single button can have different meanings according to the command mode, the ESTEREL imperative primitives help associating pieces of program text with the different modes, while LUSTRE and SIGNAL forces to handle explicit state variables. The techniques used in the ESTEREL and LUSTRE compilers are very similar, and the output code formats are unified to make the languages compatible.

7.2. The Statecharts

The Statecharts [25] are graphical hierarchical notations for automata. The main idea is to introduce an and/or decomposition of states that allows to zoom states in and out, considering a state to be a single object at one level and connected parallel automata one level below. Many of the difficulties that we mentioned for classical automata then disappear.

Arrows are labeled with signals and may go from any state to any other state, including structured states. This realizes a function similar to that of **upto** in ESTEREL: if the signal labeling an arrow is present, the source state is immediately exited, no matter which is the present configuration within it. An additional "enter by history" mechanism permits to re-enter a state at the point where it was left, or in some initial position. Signals are broadcast. Instantaneous actions can take place at states. There are explicit primitives for delays and time outs with respect to the universal time.

The basic synchrony principle is therefore the same in the Statecharts and in ESTEREL. However, the styles are orthogonal: when using the Statecharts, one describes what the user should see (typically running modes), while in ESTEREL one programs a system from basic bricks (typically reusable modules). The interested reader can compare the watch specified using the Statecharts [25] to the watch programmed in ESTEREL[9].

The Statecharts form a specification system rather than a programming system. However executable automata production should be feasible (provided a complete definition and study of the semantics).

7.3. SML

SML[15] is an imperative parallel language designed for building circuits. It is not completely synchronous, but has many points in common with the synchronous languages described so far. SML is based one one universal clock that represents a chip's clock. Signals are represented by (shared) boolean variables, as in LUSTRE. As in ESTEREL, control takes no time. However memory actions such as assignments take one unit of time. Several actions can be grouped under a **compress** statement. Then all the actions are realized in one unit of time. There is no more possibility of instantaneous communication: if a variable is read and written at the same instant, the value read is the value *before assignment*. This is certainly a loss of power at the programmer's level. But short-circuits and oscillations disappear (deadlocks can still be detected using the EMC system that is interfaced on SML).

As in ESTEREL, the SML compiler transforms a program into a finite automaton by an exhaustive exploration of its state space. Here internal operations can generate states.

8. Conclusion

We have presented new programming concepts applicable to reactive systems: synchrony (instantaneous actions and control flow), signal broadcasting, and the use of multiple time units. Several languages implement these concepts in various forms. We have presented our own language ESTEREL. We have briefly described other synchronous languages: LUSTRE (SIGNAL being close to LUSTRE could have been presented too), the Statecharts, and SML. Each of these languages induces its own programming style. It is yet uneasy to compare the languages and their induced styles, since too few common examples have been completely treated. We can however draw three conclusions from our own experience:

1. Synchrony is a good idea for programming; it is in fact *simpler and more powerful* than asynchrony when dealing with reactive systems. The deterministic character of programs makes their realization and debugging simpler. Instantaneous dialogues permit modular parallel object programming at no cost, since they don't produce code.
2. The translation of synchronous programs to automata by exhaustive exploration of the state space is a *practical* compiling process. The quality of the object code is excellent. Many striking optimizations are realized automatically during the compiling process; in particular the inter-process communication is completely done *at compile time* and generates basically no code. Progress is currently being made to make the compilers themselves faster, to bring them at the same level than scanner or parser generators.
3. Interesting proofs can be performed on the resulting automata, using systems such as EMC [19], CESAR [35], or ECRINS[30,38]. This is another advantage of the translation to deterministic automata.

To our belief, there are now two main directions to investigate:

1. On the practical side, many more realistic programs must be written in the different synchronous languages. This is of course necessary to gain more experience and to understand what are the qualities and drawbacks of the languages.
2. On the theoretical side, the theory of synchrony has to be studied independently of the presented languages. The languages could then be compared on firmer grounds, and the theories of program correctness and program equivalences could be developped further. The present theories of communicating systems are yet too weak to deal with synchronous languages: the most advanced models such as SCCS [32] or MEIJE[12] cannot express things like "two actions are performed simultaneously but in the right order". Boudol, Castellani, and Gonthier [13, 14, 23] recently introduced promizing algebras of instantaneous actions that need to be studied further.

REFERENCES

[1] ADA, *The Programming Language ADA Reference Manual*, Springer-Verlag, LNCS 155 (1983).

[2] A. Arnold, *Construction et analyse des systèmes de transitions : le système MEC*, Actes du colloque C3 d'Angoulème, CNRS (1985).

[3] J.A. Bergstra, J.W. Klop, *Process Algebra for Communication and Mutual Exclusion*, Report CS-R8409, Centrum voor Viskunde in Informatica, Amsterdam (1984).

[4] G. Berry, S. Moisan, J-P. Rigault, *ESTEREL: Towards a Synchronous and Semantically Sound High-Level Language for Real-Time Applications*, Proc. IEEE Real-Time Systems Symposium, IEEE Catalog 83CH1941-4 ,pp. 30-40 (1983).

[5] G. Berry, L. Cosserat, *The Synchronous Programming Language ESTEREL and its Mathematical Semantics*, "Seminar on Concurrency", Springer-Verlag LNCS 197 (1984).

[6] G. Berry, G. Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Research Report 842, INRIA (1987).

[7] G. Berry, F. Boussinot, P. Couronné, G. Gonthier, *ESTEREL v2.2 System Manuals*. Collection of Technical Reports, Ecole des Mines, Sophia-Antipolis (1986).

[8] G. Berry, F. Boussinot, P. Couronné, G. Gonthier, J-P. Marmorat, *ESTEREL v2.2 Programming Examples*, Collection of Technical Reports, Ecole des Mines, Sophia-Antipolis (1986).

[9] G. Berry, *Programming a Digital Watch in ESTEREL v2.2*, Technical Report, Ecole des Mines, Sophia-Antipolis (1986).

[10] G. Berry, R. Sethi, *From Regular Expressions to Deterministic Automata*, to appear in Theoretical Computer Science (1987).

[11] M. Blanchard, *Comprendre, Maitriser et Appliquer le Grafcet*. Cepadues Editions (1979).

[12] G. Boudol, *Notes on Algebraic Calculi of Processes*, INRIA Report 395 (1985).

[13] G. Boudol, *Communication is an Abstraction*, Actes du Colloque C3 d'Angoulème, CNRS — to appear as INRIA Report (1987).

[14] G. Boudol, I. Castellani, *On the Semantics of Concurrency: Partial Orders and Transition Systems*, Proc. Coll. on Trees in Algebra in Programming (CAAP), Pisa, Italy (1987).

[15] M.C. Browne, E.M. Clarke, *SML - A High Level Language for the Design and Verification of Finite State Machines*, Carnegie-Mellon University Report CMU-CS-85-179 (1985).

[16] J. A. Brzozowski, *Derivatives of Regular Expressions*. JACM, vol. 11, no. 4 (1964).

[17] L. Cardelli, R. Pike, *SQUEAK, A Language for Communicating with Mice*, AT&T Bell Laboratories Report, Bell Laboratories, Murray Hill, New Jersey 07974 (1985).

[18] P. Caspi, D. Pilaud, N. Halbwachs, J. Plaice, *LUSTRE, a Declarative Language for Real-Time Programming*, Proc Conf. on Principles of Programming Languages, Munich. (1987).

[19] E.M. Clarke, E.A. Emerson, 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 (1983).

[20] J. Chailloux, *LeLisp v15.2: Le Manuel de Référence*, INRIA Technical Report (1986).

[21] L. Cosserat, *Sémantique Opérationnelle du Langage Synchrone ESTEREL*, Thèse de Docteur Ingénieur, Université de Nice (1985).

[22] G.J. Dickson, P.E. de Chazal, *Status of CCITT Description Techniques and Application to Protocol Specification*, Proc. IEEE, vol. 71, no. 12 ,pp. 1346-1355 (1983).

[23] G Gonthier, *Sémantiques et modèles d'exécution des langages réactifs synchrones; Application à ESTEREL*, Thèse d'Informatique (1988).

[24] P. Le Guernic, A. Benveniste, P. Bournal, T. Gauthier, *SIGNAL : A Data Flow Oriented Language For Signal Processing*, IRISA Report 246, IRISA, Rennes, France (1985).

[25] D. Harel, *Statecharts : A visual Approach to Complex Systems*, Weizmann Institute of Science, Rehovot, Israel (1984).

[26] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, Weizmann Institute of Science, Rehovot, Israel (1985).

[27] C.A.R. Hoare, *Communicating Sequential Processes*, Comm. ACM vol. 21 no. 8 ,pp. 666-678 (1978).

[28] Inmos Ltd., *The Occam Programming Manual*, Prentice-Hall International (1984).

[29] S. C. Johnson, *YACC: Yet Another Compiler Compiler*, Bell Laboratories, Murray Hill, New-Jersey 07974 (1978).

[30] E. Madelaine, D. Vergamini, *ECRINS v1-5, Manuel d'Utilisation*, to appear as INRIA report (1986).

[31] R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag Lecture Notes in Computer Science 92 (1980).

[32] R. Milner, *Calculi for Synchrony and Asynchrony*, Theoretical Computer Science, vol. 25, no. 3 ,pp. 267-310 (1983).

[33] G.D. Plotkin, *A Structural Approach to Operational Semantics*, Lectures Notes, Aarhus University (1981).

[34] B. Pradin, *Un Outil Graphique Interactif pour la Vérification des Systèmes à Évolutions Parallèles Décrits par Réseaux de Petri*, Thèse de Docteur-Ingénieur, Université Paul Sabatier, Toulouse, France (1979).

[35] J-P. Queille, J. Sifakis, *Specification and Verification of Concurrent Systems in CESAR*, Proc. International Symposium on Programming, Springer-Verlag LNCS 137 (1982).

[36] J-M. Tanzi, *Traduction Structurelle des Programmes ESTEREL en Automates*, Thèse de Troisième Cycle, Université de Nice (1985).

[37] D. Vergamini, *Verification by Means of Observational Equivalence on Automata*, INRIA report 501 (1986).

[38] D. Vergamini, *Vérification du Protocole de Stenning*, To appear as an INRIA report (1987).

[39] N. Wirth, *Programming in Modula-2*, Springer-Verlag (1982).

DISCUSSION

Rapporteur: Alan Tully

Professor Whitfield asked why reactive systems were considered to be more deterministic. Dr Halbwach replied saying that such systems react to a signal according to their own internal state and are deterministic in function and time, whereas the behaviour of interactive systems depends on loading.

A speaker questioned Dr Halbwach's assumption that computation may be considered instantaneous. Dr Halbwach said that although it was only an ideal he was trying to achieve, by adopting certain compilation strategies, execution time could be considered negligible when compared to the reaction time of the environment.

Professor Turski asked Dr Halbwach's for clarification on his meaning of "at the same time". Dr Halbwach said that two events could be considered to occur at the same time if they occurred in the same time frame as perceived by the system. Further that simultaneous events were just different branches of a partial ordering, so although they may not occur at the same instant, the system behaves as if they were.

Dr Halbwach was asked if the choice between two simultaneous events was arbitrary, he replied that the choice was deterministic.

The comment was made that although ESTEREL was an attractive language for system specification, it's zero delay assumption was unrealistic when applied to distributed systems. Dr Halbwach put forward the view that distributed systems were not really necessary so ESTEREL wasn't designed to program such systems!

Professor Randell then asked if, in connection with it's application to aircraft systems, ESTEREL was executed on isolated centralized computers. Dr Halbwach answered yes, stating that ESTEREL was primarily used in the implementation of man/machine interfaces to individual computers.