The Esterel Synchronous Programming Language: Design, Semantics, Implementation

Gérard Berry, Georges Gonthier

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

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

Research funded by the French Coordinated Research Program C3

1. Introduction

The ESTEREL programming language we present here is the oldest and presently most developed member of a novel family of *synchronous languages*, which also includes the LUSTRE [19] and SIGNAL [25] languages and the Statecharts formalism [26]. These languages are specifically designed to program *reactive systems*, a variety of computerized systems that includes real-time systems and all kinds of control automata. The mathematical semantics of ESTEREL was developed *together* with the language; the implementation of ESTEREL is simply a physical realization of its semantics. The paper presents the language concepts and constructs, the mathematical semantics, and the ESTEREL implementations that are now under distribution. See [8, 9] for complete reference manuals and [6, 7] for a short introduction to the ESTEREL programming style.

1.1. Reactive systems and programs

Many computer applications involve programs that maintain a permanent interaction with their environment, reacting to inputs coming from this environment by sending outputs to it. We follow Harel and Pnueli [27] and call these *reactive programs*; we call a system whose main component is a reactive program a *reactive system*. Real time process controllers, signal processing units, digital watches, video games are typical examples of reactive systems. Operating system drivers, mouse/keyboard interface drivers (e.g., menubar or scrollbar drivers), communication protocol emitters and receivers are examples of reactive programs embedded in complex systems. Notice the *input-driven* character of reactive programs.

It is often convenient to consider reactive programs as composed of three layers:

- An *interface* with the environment that is in charge of input reception and output production. It handles interrupts, reads sensors, activates effectors; it transforms external physical events into internal logical ones and conversely.
- A *reactive kernel* that contains the *logic* of the system. It handles the logical inputs and outputs. It decides what computations and what outputs must be generated in reacting to inputs.
- A *data handling* layer that performs classical computations requested by the reactive kernel.

In the rest of this paper, we shall mostly be concerned by reactive kernels that constitute the central and most difficult part of reactive systems. In fact, ESTEREL is not a full-fledged programming language, but rather a program generator used to program reactive kernels in the same way as YACC [32] is used to program parsers from grammars. The interface and data handling must be specified in some host language.

1.2. Deterministic reactive programs

Determinism is an important characteristic of reactive programs. A deterministic reactive program produces identical output sequences when fed with identical input sequences. All examples above are deterministic if physical time is considered as an input among others. The importance of determinism cannot be overestimated: deterministic systems are one order of magnitude simpler to specify, debug, and analyze than non-deterministic ones.

Purely sequential systems are obviously deterministic. But determinism does *not* mean sequentiality. Most reactive systems can indeed be decomposed into *concurrent* deterministic subsystems that cooperate in a deterministic way. For example, a typical digital wristwatch contains a timekeeper, a stopwatch, and an alarm, all of which naturally cooperate deterministically. Deterministic concurrency is the key to the modular development of reactive programs and, as we shall see, is only supported by synchronous languages such as ESTEREL.

Some complex reactive systems can involve several subsystems running concurrently on different processors and communicating with each other via asynchronous links (e.g., a distributed robot arm controller). Such systems are no longer globally deterministic. However, we think that it is always wise to isolate their deterministic reactive subsystems and to use our specific techniques for them. Thus we extend Hoare's Communicating Sequential Processes approach into a more general Asynchronously Communicating Deterministic Reactive Systems approach.

1.3. The current tools in reactive programming

Before presenting ESTEREL, we briefly review the tools that are currently in use for reactive programming:

- *Deterministic automata* (also called *finite state machines*) are often used to program relatively small reactive kernels, typically in protocols or controllers. The interface part is realized using operating system facilities. Data handling is done by calling routines written in conventional languages. Automata obviously yield excellent and *measurable* run-time efficiency. They are also mathematically well-known. Non-trivial correctness proofs can be performed by automatic temporal-logic formula checkers such as EMC [21], MEC [2], XESAR [39], or by automata observation systems such as AUTO [43, 42]. However, the human design and maintenance of automata turns out to be very difficult and error-prone. Non-trivial automata are difficult to draw and impossible to understand when not drawn. Small changes in specifications can involve deep changes in automata. Run-time actions must be duplicated on many transitions, thus increasing the chance of misplacing an action. Above all, automata are purely sequential and do not support concurrency: combining concurrent automata into a single automaton is never an easy task.
- Petri-Net inspired tools such as the GRAFCET [11] are widely used in programmable controllers. They run on specific machines that do not easily communicate with each other and with conventional computers. Although they include crude concurrency primitives, they do not support proper hierarchical development. Interface and data handling facilities only support simple data types such as boolean, integers, or reals. The programming and debugging tools are poor.
- *Sequential tasks* running under a "real-time" operating system are widely used. They provide some kind of concurrency by splitting a complex system into simpler communicating tasks, which can themselves be automata. Inter-task communication is often done by sharing memory, which is known to be error-prone. It can also use system communication primitives, which are generally low-level and differ from one system to another, yielding ad-hoc and highly non-portable programs. The internal program behavior is non-deterministic, unlike the applications one wants to treat. Task handling incurs run-time overhead. Execution times are hard to control. There are almost no generic simulation and debugging tools.
- *Concurrent programming languages* such as ADA [1] or OCCAM [31] are more elaborate. They naturally permit hierarchical and modular program development. Their tasking mechanism and communication primitives are defined at the language level and are portable. They often provide their user with interface and data manipulation facilities, allowing him to program in a single language all the three layers defined in section 1.1. Debugging environments exist or will exist. However, all classical concurrent languages are non-deterministic. The semantics of their time-handling primitives is somewhat vague. The execution overhead can be important, and execution times are unpredictable.

Quite amazingly, all the available techniques force the user to choose between determinism and concurrency, for they base concurrency on *asynchronous implementation models* where processes non-deterministically compete for computing resources. This leads to problems that are really unnatural when programming reactive systems and when reasoning about such programs:

- Reactions can compete with each other. New inputs can arrive before the end of a reaction; actions and communications in charge of performing the current reaction then compete with actions and communications in charge of starting the new reaction. Since there is no rule telling if and when a signal sent to another process will reach its destination, there is no systematic way of telling when a reaction is complete. The only practical solution is to guarantee the *atomicity* of each reaction. Generally this not supported by the systems and languages and it is never easy to do by hand.
- Temporal primitives such as watchdogs (e.g., "do a task in less than 3 seconds") have only tentative meanings, for nothing forces them to be accurately executed. Since they usually play a crucial role in real-time process control, one generally adds priority systems to improve the user confidence in time manipulations. Such additions burden programs and cannot be completely rigorous either.
- Each subprocess has its own perception of the whole system. One is even guaranteed that two distinct subprocesses perceive *differently* their environment. For instance, a single sensor read by two concurrent processes within a single reaction will probably return two different values, since the read operations are done at different times.

1.4. The synchrony hypothesis

All the above problems *disappear* when one adopts the *synchrony hypothesis*: each reaction is assumed to be *instantaneous* — and therefore atomic in any possible sense. Synchrony amounts to saying that the underlying execution machine *takes no time* to execute the operations involved in instruction sequencing, process handling, inter-process communication, and basic data handling (e.g., additions). To "take no time" has to be understood in a very strong sense. First, a reaction takes no time *with respect to the external environment*, which remains invariant during it. Second, each subprocess also takes no time *with respect to any other subprocess*; subprocesses react instantly to each other. In synchronous languages, inter-process communication is done by *instantly broadcasting events*; all processes therefore share the same vision of their environment and of each other. Statements take time *if and only if they say so*; temporal statements *mean exactly what they say*. For instance, the statement "`await 30 MILLISECOND`" lasts exactly 30 milliseconds, and the statement

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

means that a `SECOND` signal is sent exactly every thousandth `MILLISECOND`; in an asynchronous formalism, a `SECOND` would *never* be synchronous with a `MILLISECOND`.

Moreover, the "time" taken by a statement does not need to be measured in some predefined "universal time unit". One can as well write exact statements such as

```
every 1000 MILLIMETER do emit METER end
```

Synchrony is certainly natural from the user's point of view: the user of a watch does not worry about the internal reaction times, as long as he *perceives* that his watch reacts instantly to his commands. Synchrony is also natural from the programmer's point of view: it allows to reconcile concurrency and determinism, to write simpler and more rigorous programs, to reason about them (synchronous systems *compose* very well), and to dissociate the logic of a system from implementation-dependent features such as reaction times.

Of course, one should wonder how *realistic* the hypothesis can be from an implementor's point a view. It turns out that synchronous programs can be efficiently compiled into *highly efficient automata*, yielding excellent run-time efficiency and predictability. Performance is as good as that of carefully hand-written code. The obtained automata can be automatically implemented in any classical programming language, achieving object code portability. They can also be used as input for automata verification systems. We stress that ESTEREL is a *programming language* yielding small and efficient object code, not simply an idealized specification language that forces its user to rewrite a program after the specification is finished.

Notice that synchrony hypotheses are very classical in physics: instantaneous body interaction is the basis of Newtonian Mechanics, instantaneous propagation of electricity is the basis of Kirchoff's laws.

Within their (broad) application range, they make reasoning about the world simpler than more exact non-deterministic models such as Quantum Mechanics*. VLSI circuits rely on a similar but weaker synchrony hypothesis: all reactions take *one* clock cycle, no matter how complex they are inside; the SML reactive language [15] is based on the same hypothesis. This kind of half-way synchrony accurately reflects how circuits work. To our belief, it lacks good compositionality properties and cannot be used as the basis of a general reactive programming language.

1.5. The Esterel imperative programming language

As we mentioned in the beginning, several languages or formalisms have fully adopted the synchrony hypothesis. They have roughly the same power, but they differ by their programming style. Lustre [19] and Signal [25] are declarative data-flow languages very much in Kahn – Mac Queen style [33]. The Statecharts [26] are based on a hierarchical presentation of automata using graphical structures named *higraphs* that support concurrency and communication. Esterel adopts a more classical imperative style.

The Esterel statements handle either classical assignable variables that are local to concurrent statements and cannot be shared, or *signals* that are used to communicate with the environment and between concurrent processes. A signal carries a *status*, which is its presence or absence in a given reaction, and can carry a value of arbitrary type. The *sharing law* of signals is instantaneous broadcasting: within a reaction, all statements of a program see the same status and value for any signal. The events to which statements react are composed of possibly simultaneous occurrences of signals.

The Esterel statements fall into two classes:

- standard imperative statements like assignment, signal emission, sequencing, conditional, loop, trap-exit (or exception-block definition), and explicit concurrency. These statements are supposed to be executed on an infinitely fast machine (so that the null statement **nothing** does nothing *in no time*!).
- temporal statements such as triggers (**await** *event* **do** ...), watchdogs (**do** ... **watching** *event*), or temporal loops (**loop** ... **each** *event*).

As we have seen above with milliseconds and millimeters, the temporal primitives can be applied to *any* signal: each signal is thought of as defining an independent *time scale*. The style we promote in Esterel consists of *freely mixing* independent time scales. This favors the use of *preemptive* primitives such as watchdogs (that define for how long their body will be executed) and the nesting of such primitives. A typical Esterel statement looks like

```
do
   every STEP do
     emit JUMP
   end
watching 100 METER
```

which exactly means "jump every step during 100 meters". Alone the **every STEP** statement would last forever but it is killed after 100 METER by the enclosing **watching** statement that makes the whole statement terminate. Only a small example of Esterel programming will be given here, in section 5. More elaborate examples can be found in [9].

The Esterel modules and module interface declarations are presented in section 2. We present the Esterel statements in two steps. First, we present a set of *basic* Esterel statements in section 3, together with their intuitive semantics based on the notion of *instruction duration*. Then we present a richer, user-friendly set of *plain* Esterel statements in section 4. We show how to accurately expand plain Esterel statements into basic Esterel.

1.6. The mathematical semantics of Esterel

The intuitive semantics of section 3 can be turned into a formal *denotational semantics* that globally defines the output sequence of a program as a function of a timed input sequence. The denotational semantics is presented in [24]; it is not detailed here since it is useless for compiling algorithms.

* No one would compute billiard ball trajectories in Quantum Mechanics!

We present in detail two mathematical semantics, given by Plotkin style rewrite rules [38]: the *behavioral semantics* and the *computational semantics*. Given an input I to a program P, both determine the output O and a new program P' suited to treat the remaining inputs. The global temporal treatment of statements is therefore replaced by a local computation of each reaction. The behavior of a program on any input sequence can be computed in a step by step fashion.

The behavioral semantics is given in section 5. It defines *globally* each reaction. As in Kirshoff's electrical laws, the values O and P' are solutions of fixpoint equations that express the sharing law and determine the instantaneous information exchanges between concurrent statements. Since we want the language to be deterministic, we must require solutions to exist and to be unique. However, the equations involve non-monotonic operators (e.g., negative tests for signal presence). There is no immediate way of solving them and even of knowing whether unique solutions exist or not.

We exhibit in section 6 several kinds of paradoxical programs, that have very close electronic analogues. For example, we show a program that should mean "emit a signal **S** if and only if this signal is not present"; the electronic analogue is a *not* gate whose output is plugged into its input. We also show a program that should mean "the current integer value S of a signal **S** satisfies $S = S + 1$"; the electronic analogue is a positive feedback obtained by plugging the output of an amplifier into its input. For these nonsense programs, the equations have no solution. We also exhibit programs for which the semantic equations have several solutions.

In section 7, we present the *computational semantics* of programs. Instead of defining behaviors in a global way, we compute them as results of sequences of actions of an execution machine. Signals are implemented using shared memory, with the following read/write discipline to enforce the sharing law: a signal cannot be read until it can no longer be written (apparently simpler disciplines fail to reject all incorrect programs). A *calculus of potentials* allows us to compute action sequences that satisfy this new law or to detect if such sequences do not exist. We state our main theorem: when correct action sequences do exist, they all terminate and yield the same results (in technical terms, the computational semantics has Church-Rosser and strong normalization properties [3]); furthermore, the results are exactly those defined by the behavioral semantics. This theorem establishes the deterministic character of correct reactions.

1.7. From ESTEREL programs to automata

The computational semantics of programs can be rather efficiently implemented; it can therefore serve as basis for an ESTEREL *interpreter*. However, this interpreter would not be fast enough for actual real-time applications. Our next step is to compile ESTEREL programs into sequential automata. This is the purpose of section 7. We use a variant of Brzozowski's *derivative algorithm* [17, 10], which was originally designed to transform regular expressions into automata. The idea is to formally iterate the computational semantic calculations, building a graph whose nodes are ESTEREL terms and whose arcs bear the action sequences. Starting from a node bearing the initial program, we compute all possible reactions iteratively. Each time a new reaction is computed, the target ESTEREL term is compared to the previously computed terms. This process is easily shown to terminate.

Compiling is made very fast in the ESTEREL V3 system, which does not use the original ESTEREL language but a *kernel reactive language* described in [24]. For instance, the digital wristwatch described in [8] compiles in about 5 seconds on a SUN3, yielding a 41 state automaton involving 2494 actions. This automaton can be easily translated into C, LISP, ADA, or more generally into any suitable host language. In final machine code, it would occupy about 3K bytes of memory and have very fast and predictable reaction times, comparable to those of hand-coded automata. To perform behavior analysis and proofs, the automaton can also be used as input to the above-mentioned automata verification systems EMC, AUTO, MEC, or XESAR.

By itself, the translation to automata *justifies* the synchrony hypothesis. If not instantaneous, run-time reactions are *as fast as they can be*. Microstep sequences only contain actions that *must* be done at run-time. Process handling and synchronization are done at compile time, therefore produce no actions. This is clearly the best way to be infinitely fast.

As far as code size is concerned, the produced automata turn out to be minimal in most cases (we do not know exactly why). Unlike in asynchronous formalisms, automata explosion is not the rule. For programs that yield unreasonably big automata, the ESTEREL V3 system gives a way to replace the normal single automaton by a cascade of small automata that behave equivalently. See [9] for details.

2. The ESTEREL module structure and the global declarations

In both basic and plain ESTEREL, the programming unit is the *module*. A module has a name, a declaration part, a body, and ends with a period*:

```
module MOD :
     declaration part
     body
.
```

The declaration part declares the external objects used by the module: data objects to be implemented in the data handling layer, signals and sensors that define the reactive interface. Their declarations are inter-dependent since signals and sensors can carry values of types declared in the data declarations. All objects must be declared before they are used. The declarations are similar in basic and plain ESTEREL; some restrictions apply to interface signals in basic ESTEREL.

The body is an executable statement, written either in a restricted instruction set in basic ESTEREL or in a user-friendly instruction set in plain ESTEREL. The instruction sets will be detailed in sections 3 and 4.

2.1. Data declarations

Data declarations declare the types, constants, functions, and procedures that manipulate data. ESTEREL has a few primitive types described below, but no compound type constructors such as **record** or **array**. Complex data handling is done at an *abstract level*: data have abstract types and are manipulated by abstract functions and procedures only known by their *names*, to be implemented in a host language. See [8] for connecting ESTEREL declarations with actual definitions in host languages.

2.1.1. Type declarations

Basic ESTEREL has three primitive types: **integer**, **boolean** (with constants **true** and **false**), and **triv** (with a unique constant also called **triv**). These types are necessary to translate plain ESTEREL into basic ESTEREL (**triv** is used to turn plain ESTEREL pure signals into basic ESTEREL valued signals of type **triv**, see below). For user's convenience, plain ESTEREL defines some other basic types such as **string** and **float**, with the classical syntax of string and float literals.

The user can declare his own *abstract types* by listing them after the *type* keyword:

```
type DOUBLE, TIME;
```

2.1.2. Constant declarations

One can declare constants of predefined or abstract types:

```
constant MEASURE_NUMBER: integer, PI : DOUBLE, NOON : TIME;
```

Of course the types must have already been declared. The values are given in the host language, not in ESTEREL.

2.1.3. Function declarations

Functions are declared as usual:

```
function SQRT (DOUBLE) : DOUBLE,
         EQUAL_TIME (TIME, TIME) : boolean;
```

* The lexical aspects of ESTEREL are classical; the keywords are in lower case and reserved; we write identifiers in upper-case but this is not compulsory.

Functions are assumed to be free of side effects. Their implementation is written in the host language.

2.1.4. Procedure declarations

Procedures have two argument lists: in a procedure call, the first list contains variables passed by reference and subject to side-effects (like `var` parameters in PASCAL or `inout` parameters in ADA); the second list contains expressions passed by value (like `val` parameters in PASCAL or `in` parameters in ADA). In the declaration, only the argument types are declared. For example, to add-in-place a number of seconds to a time, one can declare:

```
procedure INCREMENT_TIME_BY_SECONDS (TIME) (integer);
```

2.2. Interface declarations

One must declare the *signals* and *sensors* that constitute the module's reactive interface (a sensor is a degenerate kind of signal available in plain ESTEREL). One can also declare *input relations* that restrict possible input events and are important for compiling programs.

Signals have instantaneous *ticks* (i.e. interrupts) that serves as *control information* for the temporal statements described in section 3 and 4. Clock pulses, button depressions, or message arrivals are typical examples of ticks. A signal `S` can also have a persistent *value* of some type, that can be accessed at any time in ESTEREL programs by the expression "`?S`". For example, the value of a message signal can be the contents of the message.

The following relation between ticks and values is assumed to hold for input signals: the value of a signal can change only when a tick occurs; in this case the new value instantly replaces the old value, which is lost. In our message example, the message value can only change when a new message is received. Hence, a program driven by the message ticks is guaranteed to correctly treat all messages. This fundamental relation between ticks and values will automatically hold for output and local signals, see the sharing law in the next section.

In plain ESTEREL, there is a special *sensor* declaration for passive external devices such as thermometers, which yield values on demand but do not generate ticks. Only the value access operation "`?`" is available for sensors.

2.2.1. Basic ESTEREL interface declarations

In basic ESTEREL, there are only two kinds of interface signals: input signals and output signals. Input signals come from the environment; they cannot be emitted internally in basic ESTEREL. They are declared with the form:

input `S` (*type*);

Conversely, output signals are emitted towards the environment of the module by the "**emit**" statement: "**emit** `S`(*exp*)" emits a signal `S` with the value of the expression *exp*. Since control transmission is instantaneous in ESTEREL, several emitters can emit the same signal at the same time with different values, as in

```
emit S(1) || emit S(2)
```

where "`||`" is the ESTEREL parallel operator. We call this phenomenon a *collision*. When collisions occur, we have to define the actual value `?S` of the signal. Following Milner [36], we associate an associative commutative *combination* function *comb* with each signal `S`. If the emitters emits the values $v_1, v_2, \ldots, v_n$, the actual value of `S` is

$$comb(v_1, comb(v_2, \ldots comb(v_{n-1}, v_n) \ldots))$$

An output signal declaration has the form:

output `S` (**combine** *type* **with** *comb*);

where *type* and *comb* must already have been declared, with *comb* declared as

```
function comb(type,type):type;
```

Here are some useful combination functions:

1. In Ethernet-like local networks, signal broadcasting is physically realized on a cable. A special value NAK represents the collision of any two messages. One sets $comb(v_1, v_2) = \mathtt{NAK}$ for all v_1 and v_2.
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 these names. The appropriate combination operation is set union.
3. In the digital watch programmed in [9], the timekeeper, stopwatch, and 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 these 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 and choose integer addition as the *comb* function.

2.2.2. Plain ESTEREL interface declarations

In practice, one often uses pure control signals whose values are meaningless, such as SECOND, METER etc. In basic ESTEREL one has to declare such signals of type triv. In plain ESTEREL, one can simply omit the type declaration, writing

```
input SECOND, METER;
output ALARM;
```

Also, one may know that collisions will never take place for a given signal (this indeed tends to be the default case). The combination function can then be omitted:

```
output SPEED (float);
```

The ESTEREL compiler then checks that collisions can never appear.

Basic ESTEREL establishes a sharp distinction between input and output signals. This restriction is relaxed in plain ESTEREL, which allows for signals that can be both input and output. A natural example is:

```
inputoutput BUS_REQUEST;
```

The semantics of inputoutput signals is a bit delicate and will not be detailed here. See [8] for details.

Finally, sensors are declared in the following way:

```
sensor TEMPERATURE (FAHRENHEIT);
```

The compiler will check that temporal instructions such as delays are not applied to sensors (remember that only the "?" operator applies to them).

2.2.3. Relation declarations

Relation declarations restrict the possible input events of a module. There are two kind of relations:

1. *incompatibility relations* of the form S_1#S_2#S_3; such a relation states that the signals S_1, S_2, and S_3 are mutually exclusive in input events.
2. *synchrony relations* of the form S_1=>S_2; this relation tells that S_2 will be present in an input event whenever S_1 is.

Here is an example of relation declarations:

```
relation LEFT_BUTTON # RIGHT_BUTTON,
         SECOND => HUNDREDTH_OF_SECOND;
```

There are two reasons to use input relations. First, the specification may *require* signals not to appear together: for a watch, it makes no sense to go simultaneously in stopwatch and alarm mode. Second, relations are essential to reduce the size of the generated automaton. See section 9.3 for details.

3. The basic ESTEREL instruction set and its naive semantics

We describe the expressions and statements used in basic ESTEREL, together with their intuitive semantics. The basic statements form the heart of ESTEREL. They are independent of each other. We use the meta-variables *type*, *exp*, and *stat* to range over types, expressions, and statements; we also use self-explanatory meta-variables in italic when necessary.

ESTEREL expressions and statements manipulate variables and signals, which can be declared locally at any point. The variables and signals strongly differ in that only signals can be shared. Within statements, there is no difference between input, output, or local signals.

3.1. Expressions

Expressions are used in a classical way to denote values. They are built up from constants, variables, and signal values, by operators and function calls. They are strongly typed in a classical way (see [8] for precise type-checking laws).

The constants are the natural numbers such as **123**, the boolean constants **true** and **false**, and the user-defined constants declared in the module's constant declaration part. The variables are classical identifiers (see variable declarations below). If **S** is a signal of type *type*, then **?S** is an expression of type *type* that denotes the current value of the signal **S** *at the time the expression is evaluated*, see below.

The operators are the usual integer and boolean operators (+, $*$, <=, etc). The function calls are standard (the function must be declared in the module's declaration part).

3.2. Basic statements

Here is the list of the basic statements:

nothing	*dummy statement*
halt	*halting statement*
X := *exp*	*assignment statement*
call P (*variable−list*) (*expression−list*)	*external procedure call*
emit S(*exp*)	*signal emission*
$stat_1$; $stat_2$	*sequence*
loop *stat* **end**	*infinite loop*
if *exp* **then** $stat_1$ **else** $stat_2$ **end**	*conditional*
present S **then** $stat_1$ **else** $stat_2$ **end**	*test for signal presence*
do *stat* **watching** S	*watchdog*
$stat_1$ || $stat_2$	*parallel statement*
trap T **in** *stat* **end**	*trap definition*
exit T	*exit from trap*
var X : *type* **in** *stat* **end**	*local variable declaration*
signal S (**combine** *type* **with** *comb*) **in** *stat* **end**	*local signal declaration*

The **emit**, **present**, and **watching** statements are specific to ESTEREL; they deal uniformly with input signals, output signals, or local signals declared by local signal declarations. An **exit** statement exits a control block defined by a **trap** statement. This kind of construct is well-known in LISP as the catch-throw or tag-exit construct, in ML as the failure construct, or in ADA as the exception construct. In our case, the interaction between exit and parallel statement has to be carefully defined; we shall give a first-class semantic status to **trap**−**exit** statements, instead of explaining them loosely as control-flow diverters. All other statements are common in imperative languages. Notice that the parallel statement can be used at any level; there is no static notion of process as in CSP [29].

In compound statements, the sequencing operator "**;**" has priority over the parallel operator "**||**". When necessary, statements can be grouped by bracketing them with square brackets, as in "**[**$stat_1$ **||** $stat_2$**]** **;** $stat_3$".

All variables, signals, or trap labels must be declared before they are used. Their declarations have static scope. Input, inputoutput, and output signals have global scope.

Variables cannot be shared: if a variable is updated in one branch of a parallel statement, it cannot be read or updated in the other branch (a variable is updated by an assignment or a procedure call where it appears in the first argument list).

The following additional restrictions apply to basic ESTEREL programs:

- Input signals cannot be internally emitted.
- The `present S` statement and the value access `?S` are not allowed for output signals.

These restrictions simplify the mathematical semantics. They are suppressed in plain ESTEREL (however, the compiler produces warnings when they are not satisfied).

3.3. The intuitive semantics

The intuitive semantics describes the behavior of a module on a given input history. Let us call *input event* the occurrence of one or possibly several simultaneous input signals coming from the environment. The module reacts to each input event by updating local variables and emitting local and output signals. The emitted output signals make up the *output event* sent to the module's environment. This whole process is called the *reaction* to an input event. The reaction is assumed to be instantaneous: the output event is synchronous with the input event. A sequence of input events is called an *input history*; the events define the *instants* of the history. Reactions only occur on input events; the underlying execution machine is inactive between input events.

The signals that constitute the events all obey the following *sharing law*:

- A signal has a fixed *status* in each reaction: it is either *present* or *absent*. To be present, a signal must either be present in the input event if it is an input signal or be emitted by the program if it is a local or output signal.
- A signal has a unique *current value* `?S` in each reaction. If a signal is present in a reaction, its value is its current input value if the signal is input or is the combination of all the emitted values if the signal is output or local. If a signal is absent, its current value is the same as in the previous reaction. Before its first emission, the value of a local or output signal is the undefined value $\perp$.

(Intuitively, the sharing law should imply program determinism; as we shall see in section 7, this is only true for "correct programs".)

Since variables are not shared between statements, they can be updated several times within a single reaction. Their initial value is also $\perp$.

The key idea of the intuitive semantics is to describe formally not only the *actions* performed by each statement (memory updates, signal emissions, or tests), but also their *timing*, that is, at which "instant" they are performed. Signal current values and in general all subprocess interaction will be defined solely in terms of timing.

To describe this timing in a structural way, the semantics relies on four notions. First, the context of each statement in a program determines the instant this statement *starts* executing; second, the internal execution of this statement determines when it *terminates*, if it ever does. When a statement terminates on the *same* instant it starts, we say it *terminates instantly*, or that it is *instantaneous*. Almost everything in ESTEREL is instantaneous: expression computations, memory updates, communication, and control transmission. Third, since ESTEREL has block exits, the execution of a statement can also determine when it *exits a trap*; a statement that exits a trap does *not* terminate in the above sense (however, it is inactive from there on). A statement that does not terminate nor exit a trap instantly is said to *take time*. Finally, a statement can be *aborted* or *killed* by some other part of the program, at some instant; it is then prevented from performing any actions (or terminating, or exiting) from then on.

The semantics is structural and describes the relations between these notions for statements and their substatements:

- The module body starts upon reception of the first input event. It never terminates (it is therefore implicitly followed in sequence by a `halt` statement).
- `nothing` performs no action and terminates instantly.
- `halt` performs no action and never terminates nor exit traps.
- An assignment updates the memory and terminates instantly.
- A procedure call updates the memory and terminates instantly. (Long computations to be performed while the program is running should *not* be realized by procedure calls. They should be realized by sending the arguments to some external computing devices and waiting for the results, using signals for value communication. A specific time-consuming `exec` primitive will be added to ESTEREL in subsequent versions.)
- When it starts, an `emit` statement evaluates its expressions to a value, emits its signal with this value, and terminates.
- The first statement of a sequence starts when the sequence starts. When the first statement exits a trap, so does the sequence (the second statement then never starts). When the first statement terminates, the second statement starts instantly and the sequence behaves as the second statement from then on.
- The body of a `loop` starts when the loop starts. When the body terminates, the loop is instantly restarted. A loop never terminates. When the body exits a trap, so does the whole loop (hence exiting an outer trap is the only way to exit a loop).
- When started, a conditional instantly evaluates its condition. If the condition is true, the `then` statement starts instantly and the conditional behaves as this statement from then on. The behavior is symmetric if the condition is false.
- A `present S` statement acts as a conditional, the condition being the presence of the signal `S` in the current reaction (notice that this condition cannot be expressed by a boolean expression).
- A "`do` *stat* `watching S`" statement gives a *time limit* to the execution of its body *stat*; the limit is the *next* reaction where `S` is present. The body starts instantly when the `watching` statement starts. If the body terminates or exits a trap *strictly* before `S` occurs, so does the `watching` statement. Otherwise the `watching` statement terminates as soon as `S` occurs. In this case, the body is instantly killed *without being executed*; it performs no action and exits no trap*. (Unlike in other languages, there is no implicit loop in a `watching` statement, which is not restarted when terminated.)
- The two branches of a parallel start when the parallel starts. If one of the branches exits a trap, so does the parallel statement, and both branches are inactive from then on. If both branches exit traps simultaneously, the parallel only exits the outermost trap, the other one being discarded. Otherwise a parallel statement terminates if and when both its branches have terminated.
- A `trap T` construct defines an exit point for its body. The body starts instantly and determines the behavior of the `trap` statement until it terminates or exits a trap. If the body terminates or exits `T`, the `trap` statement instantly terminates. If the body exits an outer trap `T'`, so does the `trap` statement.
- An `exit T` statement exits `T` and doesn't terminate.
- A local variable declaration declares a variable initialized to $\perp$ and behaves as its body from then on.
- A local signal declaration declares a signal initialized to $\perp$, and behaves as its body from then on.

3.4. Remarks on imperative statements

A reaction can instantly execute several actions. The following statement executes two successive assignments to X and two signal emissions in the reaction that starts it:

* Be careful: an `S` present in the reaction that starts the `watching` statement is *not* taken into account for termination; the `watching` statement described here was called "`watching next S`" in older versions of ESTEREL; `next` is always the default in the present version.

```
      X:= 1; X:=X+1
   ||
      emit S1; emit S2
```

Even when they are done "simultaneously", variable updates are done in the specified order; as expected, the final value of X is 2. On the contrary, because of the sharing law, the ordering between the two signal emissions is immaterial. One could just as well put them in parallel.

There is a problem with loops. One can write absurd loops such as

```
   loop
      X := X+1
   end
```

where one should instantaneously execute infinitely many additions and memory updates. To forbid this situation, we impose that the body of a loop cannot terminate instantly when started (this is detected at compile time).

Notice that we introduced two conditional statements: "`if`" that tests boolean expressions, and "**`present`**" that tests for the presence of signals. We could as well introduce a boolean expression "**`present S`**" true if and only if `S` is present in the current reaction and use the form "**`if present S then`**..." to test for the presence of `S`. However, signals and booleans behave differently: the presence and value of a signal are uniquely defined in a reaction, while a boolean variable can be updated several times in an instant; moreover, the **`watching`** primitive is available for signals and not for boolean expressions.

In conditionals, we allow ourselves to suppress **`nothing`** statements appearing after **`then`** or **`else`** keywords, together with the corresponding keyword. For example:

`if` *exp* **`then`** *stat* **`end`**	stands for	**`if`** *exp* **`then`** *stat* **`else nothing end`**
`present S else` *stat* **`end`**	stands for	**`present S then nothing else`** *stat* **`end`**

This form is especially useful for the **`present`** statement: "**`present S else`** *stat* **`end`**" is a test for *absence* of `S`.

Finally, to illustrate the interaction between **`exit`** and parallel statements, let us analyze a toy example:

```
   trap T1 in
      trap T2 in
            X:=0
         ||
            Y:=0; exit T2
         ||
            Z:=0; exit T1; Z:=1
      end;
      U:=0
   end
```

Here the assignments `X:=0`, `Y:=0`, and `Z:=0` are performed simultaneously, as the actions of the parallel branches. The assignment `Z:=1` is not performed since it follows in sequence an **`exit`** statement that does not terminate. The parallel branches exits both `T1` and `T2`. Since the **`trap T1`** construct encloses the **`trap T2`** one, the `T2` exit is discarded and only `T1` is exited. Therefore the whole statement terminates instantly, and the assignment `U:=0` is not executed.

3.5. Remarks on temporal statements

According to our sharing law, two simultaneous actions are executed in the *same* signal environment (for signals that are visible in both statements). This realizes communication by instantaneous broadcasting between emitters and receivers. Consider the following example, where `S` in an integer signal with addition as combination function:

```
      emit S(2);
      Y := ?S
   ||
      emit S(1);
      present S then
         X:=?S
      end
```

Here the two emissions are simultaneous, the **present** statement sees S as present, and both X and Y receive the value ?S equal to 3=1+2.

The only way for a statement to take time is to involve a **halt** statement, which takes an infinite amount of time; conversely, a **watching** statement limits the time taken by its body. All temporal manipulations in basic ESTEREL are combinations of **halt** and **watching** statements. The simplest example of such an interaction is the following construct, which waits for the next occurrence of S and terminates:

```
do halt watching S
```

In plain ESTEREL, this construct is abbreviated into "**await** S". To simplify our examples, we also use this abbreviation in basic ESTEREL examples.

To illustrate the temporal behavior of the **watching** statement, let us look at a simple example. Assume that I1, I2, and I3 are input signals and that O1 and O2 are output signals. Let "I1 $\rightarrow$ O1" be a statement that emits O1 whenever it receives I1 (in plain ESTEREL, we shall write "**every immediate** I1 **do emit** O1 **end**", see the next section). Then the statement

```
do
   I1 → O1
watching I2;
emit O2
```

emits O1 whenever it receives I1, up to the first reaction where I2 occurs in the input event, initial reaction excluded; when I2 occurs, O1 is not emitted since the body of the **watching** statement is not executed, and O2 is emitted since the **watching** statement terminates. Here are some behaviors:

input	I1	I3	I1	I2	I1
output	O1	ϵ	O1	O2	ϵ

input	I1 · I2	I3	I1	I2	I1
output	O1	ϵ	O1	O2	ϵ

input	ϵ	I3	I1	I1 · I2	I1
output	ϵ	ϵ	O1	O2	ϵ

input	I2	I2	I1	I2	I1
output	ϵ	O2	ϵ	ϵ	ϵ

In a **watching** S statement, the body is *not* executed if the reception of S terminates the **watching** construct. We want to motivate this important choice. First, notice that the other choice, "execute the body action in the instant where S is present and terminate", can be coded as

```
trap TERMINATE in
      stat;
      exit TERMINATE
   ||
      await S;
      exit TERMINATE
end
```

The **trap** construct is exited either when *stat* terminates or when S occurs. In the latter case, *stat* is executed at the instant where S occurs, since it precedes the **exit** statement in a sequence. No simple reverse coding exists to obtain our original semantics for **watching** from this alternative one.

Second, nesting **watching** statements establishes *priorities* between signals. Consider the example:

```
do
   await S1;
   X := 0
watching S2
```

The behavior is as follows:

- S1 *occurs strictly before* S2*:* when S1 occurs, X is set to 0 and the whole construct terminates.

- `S2` *occurs strictly before* `S1`*:* the whole construct terminates when `S2` occurs; the assignment is not executed.
- `S1` *and* `S2` *occur simultaneously:* the body of the `watching S2` is *not* executed when the signals occurs; hence the whole construct terminates without executing the assignment, as in the previous case.

Therefore the outermost signal `S2` has *instantaneous priority* over the innermost signal `S1`.

Asynchronous languages usually possess watchdogs analogous to our `watching` statement but restricted to some "absolute time" measured in seconds (or worse, in some machine-dependent unit). Their semantics cannot be made completely rigorous. In ESTEREL, `watching` statements are well-defined and applicable to *any* signal, not just to absolute time; the nesting of `watching` statements on different signals is one of the basis of the ESTEREL programming style (see section 5).

Let us finally mention a subtlety concerning local signals: a local signal can be emitted *simultaneously in different scopes*. Consider the example

```
module FOO:
input S1 (integer);
output S2 (integer), S3 (integer);
loop
   signal S (integer) in
         emit S(0);
         await S1;
         emit S(1)
      ||
         emit S2(?S);
         await S;
         emit S3(?S)
   end
end
```

When the loop starts, `S` is emitted with value 0 and `S2` is output with value 0. Then, when `S1` occurs, `S` is emitted with value 1 and `S3` is output with value 1, the parallel and local signal declarations terminate, the body of the loop is restarted instantly *with* `S` *reinitialized*, `S` is instantly emitted with value 0, and `S2` is instantly output with value 0. The two simultaneous emissions of `S` don't occur in the same instance of the local signal declaration and don't have to be combined; this is clear when one unfolds once the loop before executing it.

The semantics presented here treats correctly this example, while the initial semantics of ESTEREL [5] and the ESTEREL V2 system treated it incorrectly.

4. The plain ESTEREL instruction set

There are three kinds of extensions available in plain ESTEREL: extensions concerning the signals and the signal interface, some user-friendly statements, and the `copymodule` directive for modular programming. We have already detailed the plain ESTEREL signals in section 2; the `copymodule` directive will be described at the end of this section.

The extended instructions are derived from the basic ones by macro-expansion. Being the most powerful control structure, the `trap-exit` mechanism is heavily used in the expansions (we have already seen an example in the previous section).

4.1. Miscellaneous easy extensions

We give some extensions without detail, see [8] for exact expansions. One can initialize variables at declaration time (the appropriate assignment is generated). One can declare several variables or signals in local variable or signal declaration. One can define a `repeat` loop of the form

```
repeat exp times
   stat
end
```

The **repeat** construct expands into a **loop** nested within a **trap**, an **if-then-else** statement exiting the **trap** when the count expires.

4.2. Occurrence counts and timeouts in watching statements

In **watching** statements, it is often convenient to distinguish between the body's normal termination and the timeout termination caused by the occurrence of the signal. For this, one adds a **timeout** clause:

```
do
   stat₁
watching S
timeout stat₂
end
```

The expansion is easy:

```
trap TERMINATE in
   do
      stat₁;
      exit TERMINATE
   watching S;
   stat₂
end
```

In basic ESTEREL, a time limit is the next occurrence of some event; in plain ESTEREL, one allows two other forms of time limits. The first one is an *occurrence count* of the form "*exp* **S**", where *exp* is an integer expression. For instance, one can build seconds from milliseconds in the following way:

```
loop
   await 1000 MILLISECOND;
   emit SECOND
end
```

(This is still heavy and will be further improved later on.) The theoretical expansion uses an auxiliary local signal and a **repeat** loop (we leave this to the reader). In practice, occurrence counters are so useful that they are built-in primitives in the actual ESTEREL systems [8].

The second generalized occurrence form is called the **immediate** form. Remember that the starting event is not taken into account for timeout in a **watching** statement. The following form takes it into account*:

```
do
   stat
watching immediate S
```

The expansion is

```
present S else
   do stat watching S
end
```

Occurrence counts and immediate occurrences are available for all the temporal statements described below.

4.3. The upto statement

The **upto** statement is similar to the **watching** statement, but it doesn't terminate when its body terminates: only the timeout terminates an **upto** statement. The construct "**do** *stat* **upto** S" expands into

```
do
   stat;
   halt
watching S
```

Unless its body exits some enclosing tag, an **upto** statement is guaranteed to take the *exact* time mentioned in its time limit.

* Be careful: "**watching immediate**" was called "**watching**" in older versions of ESTEREL!

The `upto` statement was taken as primitive in the older versions of ESTEREL [5]. One can indeed define `watching` from `upto`; however `watching` has several advantages as a semantic primitive (see [24]).

4.4. Await statements

We have already written "`await S`" for "`do halt watching S`". To point out the dependency of a statement on the arrival of a signal `S`, one can also write

```
await S do stat end
```

instead of "`await S; stat`".

The most general form of the `await` statement is the *multiple await*. For example, one can write:

```
await
   case SECOND do stat1
   case 2 METER do stat2
   case immediate ALARM do stat3
end
```

The first elapsed delay determines the case to execute. Unlike the similar selection statement found in classical asynchronous languages, ours is deterministic. If two delays elapse simultaneously, only the first case in the list order is selected. For instance, if there is no `ALARM` and if `2 METER` and `SECOND` are reached simultaneously, only $stat_1$ is executed. See the expansion in [8].

4.5. Temporal loops

Let us come back to the production of seconds from milliseconds programmed above. A much simpler form is

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

The expansion of "`every S do stat end`" is simply

```
await S;
loop
   do stat upto S
end
```

One first waits for the signal (or the signal's occurrence count). One then starts the body *stat*; this body is restarted afresh at each occurrence of the signal (or of the signal's occurrence count).

The next temporal loop is similar but the body is started at once:

```
loop stat each S
```

expands into

```
loop
   do stat upto S
end
```

4.6. Exceptions

Plain ESTEREL includes a general exception handling mechanism that extends the basic `trap` mechanism by allowing exit handlers and value passing. Here is an example of the general construct:

```
trap ALARM(combine integer with +), ZERO_DIVIDE, TERMINATE in
   stat
handle ALARM do stat1
handle ZERO_DIVIDE do stat2
end
```

We call *stat* the *body* and $stat_1$ and $stat_2$ the *exception handlers*. The body can contain generalized `exit` statements of the form "`exit ALARM(exp)`" or "`exit ZERO_DIVIDE`". If such a statement is executed when executing the body then the body is instantly exited and the corresponding handler is instantly started if present (here `TERMINATE` has no handler). As far as values are concerned, valued exits behave much like signals: if several "`exit ALARM(`*exp*`)`" are raised then the value of the `ALARM` exit is obtained by combining the values with the addition function; this value can be accessed in the `ALARM` handler via the special expression

```
??ALARM
```

The double question mark is used to distinguish exceptions from normal signals: the signals and exits don't belong to the same name space. The expression "`??ALARM`" is only allowed in the `ALARM` handler.

Several different exits can be raised simultaneously. If they belong to different **trap** construct, only the outermost ones matter. If they belong to a single **trap** construct, the corresponding handlers are executed in parallel.

See [8] for the general expansion of this statement.

4.7. The copymodule directive

Plain ESTEREL has a limited form of modularity, given by the directive

```
copymodule MODULE
```

This directive can appear anywhere in a statement's place. It is replaced by the *text* of the copied module with consistency verifications for interface and declarations. Copying cannot be recursive but can be nested to any depth. Renamings are also allowed in the copying process, see [8]. See section 5 and annex 1 for an example.

5. A programming example

As a simple but illustrative example, we program the reflex game machine described in detail in [9]; a more complex wristwatch example appears in the same paper. The full ESTEREL reflex game program is shown in Annex 1.

5.1. The reflex game specifications

The player controls the machine with three commands : putting a coin in a `COIN` slot, to start the game; pressing a `READY` button, to start a reflex measure; pressing a `STOP` button, to end a measure.

The machine reacts to these commands by operating the following devices: a numerical display `DISPLAY` that shows reflex times; a `GO` lamp that signals the beginning of a measure; a `GAME_OVER` lamp that signals the end of a game; a `RED` lamp that signals that the player has tried to cheat or has abandoned the game; a `BELL` that rings when the player hits a wrong button.

When the machine is turned on, the display shows 0, the `GAME_OVER` lamp is on, the `GO` and `RED` lamps are off. The player then starts a game by inserting a coin. Each game is composed of a fixed number `MEASURE_NUMBER` of reflex measures. A measure starts when the player presses the `READY` button; then, after a random amount of time, the `GO` lamp turns on and the player must press the `STOP` button as fast as he can. When he does so, the `GO` lamp turns off and the reflex time measured in milliseconds is displayed on the numerical display. A new measure starts when the player presses `READY` again. When the cycle of `MEASURE_NUMBER` measures is completed, the average reflex time is displayed after a pause of `PAUSE_LENGTH` milliseconds and the `GAME_OVER` lamp is turned on.

There are five exception cases. Two of them are simple mistakes and make the bell ring:

- the player presses `STOP` instead of `READY` to start a measure
- the player presses `READY` during a measure

In the other three cases, the `RED` and `GAME_OVER` lamps are turned on, the `GO` lamp is turned off, and the game ends:

- the player does not press the `READY` button within `LIMIT_TIME` milliseconds when he is expected to (one assumes that the player has abandoned the game)
- the player does not press the `STOP` button within `LIMIT_TIME` milliseconds when he is expected to (that is, after the `GO` light turns on; this is also assumed to be an abandon)

- the player presses the STOP button after he has pressed the READY button but before the machine turns the GO light on, or *at the same time* that this happens (this is a cheat!)

A last anomaly appears if the player inserts a coin during a game. Then a new game is started afresh at once.

5.2. The declarations of REFLEX_GAME

There are three parameters to the game which are declared as integer constants. Notice that their values are not given in the ESTEREL program; they must be given in the host language. To determine the random delay length, we use an external function RANDOM also defined in the host language.

The input declarations declare the millisecond time unit MS and the three user commands. Notice that no absolute time is predefined in ESTEREL: time is just one signal among others. As far as input relations are concerned, all input signals are assumed to be incompatible, except MS and STOP: if the player presses STOP simultaneously with the occurrence of MS that terminates the random delay then he must be considered as a cheater.

To control a lamp (say GO), we introduce two output signals ON and OFF (hence GO_ON and GO_OFF). We could as well use a single signal conveying a boolean value, as in the wristwatch example [9]. As well we have output signals for the display and to ring the bell.

5.3. An AVERAGE submodule

We use a general-purpose submodule to compute the average response time. This simple module emits AVERAGE_VALUE whenever it receives INCREMENT_AVERAGE with a new argument:

```
module AVERAGE :

input INCREMENT_AVERAGE(integer);
output AVERAGE_VALUE(integer);

var TOTAL:=0, NUMBER:=0 : integer in
   every immediate INCREMENT_AVERAGE do
      TOTAL := TOTAL + ? INCREMENT_AVERAGE;
      NUMBER := NUMBER + 1;
      emit AVERAGE_VALUE (TOTAL/NUMBER)
   end
end.
```

5.4. The body of REFLEX_GAME

The body is composed of two successive parts: some overall initializations and a main loop over a single game which is restarted whenever a coin is inserted. This main loop is simply controlled by an "every COIN" statement. Within a single game, we declare an ERROR exit to handle the cheating tentatives and an END_GAME exit to handle the normal game termination; we need this last exit since the actual statement that treats a single game is put in parallel with a copy of the AVERAGE module which never terminates. A single game is a sequence of a measure loop and a termination action (turning GAME_OVER on). Each measure is divided into three phases. In phase 1, one waits for READY with a time limit of LIMIT_TIME MS, ringing the bell whenever STOP is pressed. This is easily written, nesting three temporal statements bearing on three different time units:

```
do
   do
      every STOP do emit RING_BELL end
   upto READY
watching LIMIT_TIME MS timeout exit ERROR end
```

Phase 2 consists of waiting for RANDOM MSwhile phase 3 is waiting for the STOP button. During these phases, pressing READY rings the bell: this is treated by putting an "every READY" statement in parallel with the phase 2 – phase 3 sequence. In phase 2, the specification says "STOP should not be pressed within RANDOM MS"; we rewrite this in the positive form "RANDOM MS should occur within a time limit of STOP", in ESTEREL:

```
do
   await RANDOM() MS
watching STOP timeout exit ERROR end
```

This shows how useful temporal statements are for arbitrary signals and not just on some privileged absolute time unit. The full code is in Annex 1. It should be self–explanatory.

6. The behavioral semantics of Esterel

Our purpose is to give a mathematical definition of the semantics of basic Esterel. The intuitive semantics was given in terms of statement *duration* measured in signal occurrences. A formal *denotational semantics* corresponding to this intuitive concept is presented in [24]. The *behavioral semantics* we present in this section is different in spirit and more suited to practical implementations. Given any program and input event, it determines the output event generated by the program reaction *and a new program able to handle subsequent input events.* The duration semantics is replaced by a one-shot semantics involving program rewritings, in the spirit of *natural deduction semantics* based on structural deduction rules [38].

Usual natural deduction semantics are "executable": their rules can be directly used for building interpreters. The behavioral semantics presented here doesn't have this property, since the treatment of the sharing law involves an "uneffective" fixpoint operation in the local signal declaration rule. However, the behavioral semantics serves as a *formal definition* of Esterel: any operational semantics should agree with it as far as input-output behavior is concerned. In section 8, we present such a more effective but more complex *execution semantics* together with the theorem that ensures it agrees with the behavioral semantics.

6.1. Formal definitions of events and histories

An *event* $E = \mathtt{S}_1(v_1)\cdot\mathtt{S}_2(v_2)\cdots\mathtt{S}_n(v_n), n \geq 0$, is a set of signals that are simultaneously emitted with the corresponding values. If $\mathtt{S}$ appears in E with value v, we write $\mathtt{S} \in E$, $\mathtt{S}(v) \in E$, and $E(\mathtt{S}) = v$. Otherwise we write $\mathtt{S} \notin E$. The empty event is called ϵ; it contains no signal.

In semantic rules, we replace the Esterel notation $comb(x, y)$ for signal value combinations in collisions by the more convenient notation $x \star_{\mathtt{S}} y$. We extend the combination operation to events, defining the *synchronous product* $E = E_1 \star E_2$ of two events E_1 and E_2 componentwise on signals:

$$\begin{array}{rll} \mathtt{S}(v_1) \in E & \text{if} & \mathtt{S}(v_1) \in E_1 \text{ and } \mathtt{S} \notin E_2 \\ \mathtt{S}(v_2) \in E & \text{if} & \mathtt{S}(v_2) \in E_2 \text{ and } \mathtt{S} \notin E_1 \\ \mathtt{S}(v_1 \star_{\mathtt{S}} v_2) \in E & \text{if} & \mathtt{S}(v_1) \in E_1 \text{ and } \mathtt{S}(v_2) \in E_2 \\ \mathtt{S} \notin E & \text{if} & \mathtt{S} \notin E_1 \text{ and } \mathtt{S} \notin E_2 \end{array}$$

Clearly, $E \star \epsilon = E$ holds for any event E.

Events only contain positive information about emitted signals. To model the persistence of values, we introduce *complete events* $\hat{E}$ that also include the value information of absent signals. Call a set of signals a *sort* $\mathcal{S}$. In a complete event, a signal of a given sort $\mathcal{S}$ can appear either as $\mathtt{S}^+(v)$ if $\mathtt{S}$ is emitted with value v or as $\mathtt{S}^-(v)$ if $\mathtt{S}$ is not emitted and has current value v.

An *history* $H = E_0, E_1, \ldots, E_n, \ldots$ is a possibly infinite sequence of events; $H[n]$ denotes the finite history $E_0, E_1, \ldots, E_n$. A *complete history* $\hat{H}$ is a sequence of complete events $\hat{E}_n$ that respects the persistence law for signal values. That is, it satisfies:

$\mathtt{S}^- \in \hat{E}_0$ implies $E_0(\mathtt{S}) = \perp$

$\mathtt{S}^-(v) \in \hat{E}_{n+1}$ implies $\hat{E}_n(\mathtt{S}) = v$

Assume that all signals in an history H belong to a sort $\mathcal{S}$. Then the *completion* $\hat{H}$ of H with respect to $\mathcal{S}$ is defined as follows, for each $\mathtt{S} \in \mathcal{S}$:

$\mathtt{S}^-(\perp) \in \hat{E}_0$ iff $\mathtt{S} \notin E_0$

$\mathtt{S}^+(v) \in \hat{E}_n$ iff $\mathtt{S}(v) \in E_n$

$\mathtt{S}^-(v) \in \hat{E}_{n+1}$ iff $\mathtt{S} \notin E_{n+1}$ and $\hat{E}_n(\mathtt{S}) = v$

Example: Consider the sort $\mathcal{S} = \{\mathtt{S1}, \mathtt{S2}\}$. Here is an history and its completion:

H	$\mathtt{S1}(0)$	$\mathtt{S2}(1)$	ϵ	$\mathtt{S1}(2)\cdot\mathtt{S2}(2)$
$\hat{H}$	$\mathtt{S1}^+(0)\cdot\mathtt{S2}^-(\bot)$	$\mathtt{S1}^-(0)\cdot\mathtt{S2}^+(1)$	$\mathtt{S1}^-(0)\cdot\mathtt{S2}^-(1)$	$\mathtt{S1}^+(2)\cdot\mathtt{S2}^+(2)$

The *current value* of a signal **S** at n-th step in an history H is thus $\hat{H}_n(\mathtt{S})$.

6.2. Module derivatives

Given an input history I, a program **P** computes an output history O such that the n-th output event O_n only depends on the input sequence $I[n]$ formed by the n first events of I. A classical idea of natural semantics [34, 38] is to compute the output history in a step-by-step fashion. To apply this idea, we build a sequence of the form

$$\mathtt{P}=\mathtt{P}_0 \xmapsto[\hat{I}_0]{O_0} \mathtt{P}_1 \xmapsto[\hat{I}_1]{O_1} \mathtt{P}_2 \xmapsto[\hat{I}_2]{O_2} \cdots \mathtt{P}_n \xmapsto[\hat{I}_n]{O_n} \cdots$$

where the $\mathtt{P}_n$ are basic ESTEREL programs and the $\hat{I}_n$ are the complete events of the completed input history $\hat{I}$ associated with I (we need to use complete input events to treat the fact that input signal values are permanent).

The key point is to rewrite at each step the program $\mathtt{P}_n$ into a new program $\mathtt{P}_{n+1}$ called the *derivative* of $\mathtt{P}_n$ with respect to $\hat{I}_n$. The derivative $\mathtt{P}_{n+1}$ is the ESTEREL program that computes the output history starting at step $n+1$ from the input history starting at step $n+1$. It has the same declarations as $\mathtt{P}_n$ but a different body.

Here is a simple illustrative example (using plain ESTEREL constructs):

```
module P:
input SECOND;
output BEEP;
await 2 SECOND do emit BEEP end;
halt.
```

The derivative by the empty event is the program itself. The derivative by the **SECOND** event is the program having as body

```
await 1 SECOND do emit BEEP end;
halt
```

Upon reception of another **SECOND**, **BEEP** is emitted and the program stops; its body becomes the statement "**halt**" that accepts input but never produces output.

The derivative technique transforms a temporal problem into two instantaneous ones: find the instantaneous reaction on an input and find the derivative. The technique was introduced by Brzozowski [17] to compute the automaton recognizing the language generated by a regular expression.

6.3. Inductive rules

The $\mapsto$ relation between programs is deduced from a similar $\rightarrow$ relation between statements, which is defined by deduction rules that determine the transition of any ESTEREL construct from the transitions of its subconstructs. In order to handle control transmission and expression computation, the $\rightarrow$ relation has more components than the $\mapsto$ relation. It has the form:

$$< stat\,,\,\rho > \xrightarrow[\hat{E}]{E',b,T} < stat'\,,\,\rho' >$$

with the following conventions:

- ρ and ρ' are *memories* that allocate the free variables of *stat* (memories are described below).
- $\hat{E}$ is a complete event that represents the complete signal environment in which *stat* is executed. Its sort is the set of input signals and of local signals visible from *stat*.

- E' is an event that contains the signals *emitted* by *stat* and their values. Its sort is the set of output signals and of local signals visible from *stat*.
- b is a *termination* boolean having value *tt* if *stat* terminates and *ff* otherwise.
- T is a set of trap labels that contain the labels of the `exit` statements executed by *stat*. Our treatment of parallel exits will slightly differ from the one we gave in the intuitive semantics, but remain equivalent; we said there "if the arms of a parallel exit several traps simultaneously, only the outermost trap matters, the other ones being discarded". Mimicking such a statement in semantic equations would require a preorder between trap labels that is heavy to compute in a structural way. Our formal solution will be to retain all labels exited by a parallel in the set T, and to choose the action to perform at the **trap** statement level: a **trap** `T` statement is considered as exited if its body *exactly* exits the set $\{\mathtt{T}\}$; otherwise, it simply propagates the other exits in T.

Memories ρ are manipulated as follows

- There is an *empty memory* ϕ that allocates no variable.
- If ρ is a memory that allocates a set $\mathcal{V}$ of variables and if `X` is a variable, then $\rho.(\mathtt{X}{=}v)$ is a memory that allocates $\mathcal{V} \cup \{\mathtt{X}\}$.
- If ρ allocates $\mathcal{V}$ and if $\mathtt{X} \in \mathcal{V}$, then $\rho(\mathtt{X})$ denotes the value of `X` in ρ, defined by $\rho.(\mathtt{X}{=}v)(\mathtt{X}) = v$ and $\rho.(\mathtt{Y}{=}v)(\mathtt{X}) = \rho(\mathtt{X})$ if $\mathtt{X} \neq \mathtt{Y}$.
- If ρ allocates $\mathcal{V}$ and if $\mathtt{X} \in \mathcal{V}$, then $\rho[\mathtt{X} \leftarrow v]$ denotes the memory ρ where `X` receives the new value v. Formally, $\rho.(\mathtt{X}{=}v')[\mathtt{X} \leftarrow v] = \rho.(\mathtt{X}{=}v)$ and $\rho(\mathtt{Y}{=}v')[\mathtt{X} \leftarrow v] = \rho[\mathtt{X} \leftarrow v].(\mathtt{Y}{=}v')$ if $\mathtt{Y} \neq \mathtt{X}$.

Notice that the memory handling respects static scoping: a memory ρ may allocate several times a variable `X`, but the accesses to `X` concern the most recent allocation.

We are now in position to define the relation between $\mapsto$ and $\rightarrow$. Let *stat* be the body of a program `M`. For technical reasons, it is simpler to assume that a program body never terminates, adding a **halt** statement in sequence if needed. Then we set

$$\mathtt{P} \overset{E'}{\underset{\hat{E}}{\longmapsto}} \mathtt{P}' \quad \text{iff} \quad < stat\,,\, \phi > \xrightarrow[\hat{E}]{E',ff,\emptyset} < stat'\,,\, \phi >$$

where $stat'$ is the body of $\mathtt{P}'$. The memory and exited label set are always empty, since we deal with syntactically correct programs for which there are no free variables or free exits.

Within the structural induction that computes $\longrightarrow$, the difficulty will be to correctly compute the status (presence or absence) of the signals and their values. The key idea of the behavioral semantics is to directly exploit the synchrony hypothesis: all statements for which a signal **S** is visible see the *same* status and value for this signal. For input signals, the status and value are simply determined by the input event of the global program. A local signal is seen as present by all statements in its scope if and only if it is emitted by some of them; the local signal rule below expresses this consistency constraint in a simple but non-effective way.

6.4. Expression evaluation

In addition to the inductive rules for statements, we need expression evaluation rules of the form

$$< exp\,,\, \rho > \underset{\hat{E}}{\longrightarrow} v$$

If **C** is a constant of semantic value c, the rule is

$$< \mathtt{C}\,,\, \rho > \underset{\hat{E}}{\longrightarrow} c$$

The rule for a variable `X` is

$$<\texttt{X}\,,\,\rho> \;\underset{\hat{E}}{\longrightarrow}\; \rho(\texttt{X})$$

If **S** is a signal, the rule for its value access **?S** is

$$<\texttt{?S}\,,\,\rho> \;\underset{\hat{E}}{\longrightarrow}\; \hat{E}(\texttt{S})$$

The rules for operators and function calls are obvious and left to the reader.

6.5. Inductive rules for statements

6.5.1. Axiom of `nothing`

A **nothing** statement terminates and leaves the memory unchanged:

$$<\textbf{nothing}\,,\,\rho> \;\xrightarrow[\hat{E}]{\epsilon,tt,\emptyset}\; <\textbf{nothing}\,,\,\rho>$$

6.5.2. Axiom of `halt`

A **halt** statement doesn't terminate and reproduces itself:

$$<\textbf{halt}\,,\,\rho> \;\xrightarrow[\hat{E}]{\epsilon,ff,\emptyset}\; <\textbf{halt}\,,\,\rho>$$

6.5.3. Rule for assignment

An assignment statement updates the memory and terminates; it becomes **nothing**:

$$\frac{<exp\,,\,\rho> \;\underset{\hat{E}}{\longrightarrow}\; v}{<\texttt{X:=}exp\,,\,\rho> \;\xrightarrow[\hat{E}]{\epsilon,tt,\emptyset}\; <\textbf{nothing}\,,\,\rho[\texttt{X} \leftarrow v]>}$$

6.5.4. Rule for procedure call

The procedure call rule is similar to the assignment rule, and is left to the reader. The expression list is computed in the current memory and signal environment, and the external procedure updates the memory. A procedure call terminates.

6.5.5. Rule for `emit`

An **emit** statement emits the expression's value and terminates:

$$\frac{<exp\,,\,\rho> \;\underset{\hat{E}}{\longrightarrow}\; v}{<\textbf{emit S(}exp\textbf{)}\,,\,\rho> \;\xrightarrow[\hat{E}]{\texttt{S}(v),tt,\emptyset}\; <\textbf{nothing}\,,\,\rho>}$$

6.5.6. Rules for sequence

If the first statement doesn't terminates then the behavior of the sequence is that of the first statement. The sequence is rewritten into the sequence of the first derivative and of the second statement:

$$\frac{<stat_1\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,ff,T_1} <stat'_1\;,\;\rho'_1>}{<stat_1\,;\;stat_2\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,ff,T_1} <stat'_1\,;\;stat_2\;,\;\rho'_1>}$$

If the first statement terminates then the second statement is executed in the memory state produced by the first one; the global rewriting is that of the second statement except that emitted signals are merged:

$$\frac{<stat_1\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,tt,\emptyset} <stat'_1\;,\;\rho'_1> \qquad <stat_2\;,\;\rho'_1> \xrightarrow[\hat{E}]{E'_2,b_2,T_2} <stat'_2\;,\;\rho'_2>}{<stat_1\,;\;stat_2\;,\;\rho> \xrightarrow[\hat{E}]{E'_1\star E'_2,b_2,T_2} <stat'_2\;,\;\rho'_2>}$$

6.5.7. Rule for `loop`

The rule performs an instantaneous unfolding of the loop into a sequence. Note that a loop can never terminate.

$$\frac{<stat\,;\;\textbf{loop}\;stat\;\textbf{end}\;,\;\rho> \xrightarrow[\hat{E}]{E',ff,T} <stat'\;,\;\rho'>}{<\textbf{loop}\;stat\;\textbf{end}\;,\;\rho> \xrightarrow[\hat{E}]{E',ff,T} <stat'\;,\;\rho'>}$$

6.5.8. Rule for `if-then-else`

The boolean expression is instantly evaluated and the selected branch is instantly executed. Here is the rule for the *true* case

$$\frac{<exp\;,\;\rho> \xrightarrow[\hat{E}]{} true \qquad <stat_1\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,b_1,T_1} <stat'_1\;,\;\rho'_1>}{<\textbf{if}\;exp\;\textbf{then}\;stat_1\;\textbf{else}\;stat_2\;\textbf{end}\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,b_1,T_1} <stat'_1\;,\;\rho'_1>}$$

The rule for the *false* case is symmetric:

$$\frac{<exp\;,\;\rho> \xrightarrow[\hat{E}]{} false \qquad <stat_2\;,\;\rho> \xrightarrow[\hat{E}]{E'_2,b_2,T_2} <stat'_2\;,\;\rho'_2>}{<\textbf{if}\;exp\;\textbf{then}\;stat_1\;\textbf{else}\;stat_2\;\textbf{end}\;,\;\rho> \xrightarrow[\hat{E}]{E'_2,b_2,T_2} <stat'_2\;,\;\rho'_2>}$$

6.5.9. Rules for `present`

The rules for present are similar to the rules for "`if-then-else`". If the signal is present in the current event, the **then** clause is instantly executed:

$$\frac{\texttt{S}^+ \in \hat{E} \qquad <stat_1\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,b_1,T_1} <stat'_1\;,\;\rho'_1>}{<\textbf{present S then}\;stat_1\;\textbf{else}\;stat_2\;\textbf{end}\;,\;\rho> \xrightarrow[\hat{E}]{E'_1,b_1,T_1} <stat'_1\;,\;\rho'_1>}$$

Otherwise, the `else` clause is instantly executed:

$$\frac{\mathtt{S}^- \in \hat{E} \qquad < stat_2\,,\, \rho > \xrightarrow[\hat{E}]{E'_2, b_2, T_2} < stat'_2\,,\, \rho'_2 >}{< \mathtt{present\ S\ then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end}\,,\, \rho > \xrightarrow[\hat{E}]{E'_2, b_2, T_2} < stat'_2\,,\, \rho'_2 >}$$

6.5.10. Rule for `watching`

A "**do** *stat* **watching S**" statement executes its body, *stat*, which yields a derivative *stat'*. The derivative of the **watching** statement is

present S else do *stat'* **watching S end**

that is "**do** *stat'* **watching immediate S**" in plain ESTEREL: when receiving the next input, this derivative will terminate instantly if **S** is present, or will behave like "**do** *stat'* **watching S**" if **S** is absent. This is precisely the intuitive behavior of **watching**.

$$\frac{< stat\,,\, \rho > \xrightarrow[\hat{E}]{E', b, T} < stat'\,,\, \rho' >}{< \mathtt{do}\ stat\ \mathtt{watching\ S}\,,\, \rho > \xrightarrow[\hat{E}]{E', b, T} < \mathtt{present\ S\ else\ do}\ stat'\ \mathtt{watching\ S\ end}\,,\, \rho' >}$$

6.5.11. Rule for parallel

The branches are executed independently but in the same signal environment. Their output events are merged. Since there are no shared variables, the branches cannot update the same variable; the resulting memory $\rho' = merge(\rho, \rho'_1, \rho'_2)$ is obtained as follows: if $\rho(\mathtt{X}) = \rho'_1(\mathtt{X}) = \rho'_2(\mathtt{X})$, then $\rho'(\mathtt{X}) = \rho(\mathtt{X})$; if $\rho'_1(\mathtt{X}) \neq \rho(\mathtt{X})$ then $\rho'(\mathtt{X}) = \rho'_1(\mathtt{X})$; if $\rho'_2(\mathtt{X}) \neq \rho(\mathtt{X})$ then $\rho'(\mathtt{X}) = \rho'_2(\mathtt{X})$.

$$\frac{< stat_1\,,\, \rho > \xrightarrow[\hat{E}]{E'_1, b_1, T_1} < stat'_1\,,\, \rho'_1 > \qquad < stat_2\,,\, \rho > \xrightarrow[\hat{E}]{E'_2, b_2, T_2} < stat'_2\,,\, \rho'_2 >}{< stat_1\ ||\ stat_2\,,\, \rho > \xrightarrow[\hat{E}]{E'_1 \star E'_2, b_1 \wedge b_2, T_1 \cup T_2} < stat'_1\ ||\ stat'_2\,,\, merge(\rho, \rho'_1, \rho'_2) >}$$

6.5.12. Rules for `trap`

The **trap** terminates if its body terminates or if the body's exited label set contains *exactly* the **trap**'s label:

$$\frac{< stat\,,\, \rho > \xrightarrow[\hat{E}]{E', b, T} < stat'\,,\, \rho' > \qquad b = tt \text{ or } T = \{\mathtt{T}\}}{< \mathtt{trap\ T\ in}\ stat\ \mathtt{end}\,,\, \rho > \xrightarrow[\hat{E}]{E', tt, \emptyset} < \mathtt{nothing}\,,\, \rho' >}$$

Otherwise the **trap** statement behaves as its body, except that its own label is removed from the exited label set (this handles correctly parallel exits from nested traps).

$$\frac{< stat\,,\, \rho > \xrightarrow[\hat{E}]{E', ff, T} < stat'\,,\, \rho' >}{< \mathtt{trap\ T\ in}\ stat\ \mathtt{end}\,,\, \rho > \xrightarrow[\hat{E}]{E', ff, T - \{\mathtt{T}\}} < \mathtt{trap\ T\ in}\ stat'\ \mathtt{end}\,,\, \rho' >}$$

6.5.13. Axiom of `exit`

An exit doesn't terminate and puts its label in the exited label set:

$$< \texttt{exit T} \, , \, \rho > \xrightarrow[\hat{E}]{\epsilon, f\!f, \{\texttt{T}\}} < \texttt{halt} \, , \, \rho >$$

6.5.14. Rule for local variable declaration

In order to retain the value of the variables from step to step when reacting to an input history, we slightly modify the basic ESTEREL variable declaration construct. The new construct is

var X= $\boxed{v}$ **in** *stat* **end**

The "$\boxed{}$" operator transforms any semantic value into an ESTEREL constant of the appropriate type. The new construct allows us to save the current value of a variable in the program text itself. The standard basic ESTEREL declaration initially sets this value to $\perp$.

The rule allocates the variable with the currently saved value, executes the body, and saves the new value for the next step:

$$\frac{< stat \, , \, \rho.(\texttt{X=}v) > \xrightarrow[\hat{E}]{E', b, T} < stat' \, , \, \rho'.(\texttt{X=}v') >}{< \textbf{var} \ \texttt{X=}\boxed{v} \ \textbf{in} \ stat \ \textbf{end} \, , \, \rho > \xrightarrow[\hat{E}]{E', b, T} < \textbf{var} \ \texttt{X=}\boxed{v'} \ \textbf{in} \ stat' \ \textbf{end} \, , \, \rho' >}$$

6.5.15. Rules for local signal declaration

These are the fixpoint rules that realize the sharing law. With respect to a declared signal S, *we require the body to work in the environment that it builds itself*. To retain values between execution steps we use the $\boxed{}$ operator introduced for variables.

There is a slight problem due to the static scope of signals: the event $\hat{E}$ may already contain a different signal having the same name S; we introduce the notation $\hat{E}\backslash\texttt{S}$ to denote the complete event obtained by removing the S-component of $\hat{E}$, if present.

The first rule applies when the signal is emitted by the body: the signal is then received by the body, the emitted and received values must coincide, and the new signal value is stored in the local signal declaration of the derivative:

$$\frac{< stat \, , \, \rho > \xrightarrow[(\hat{E}\backslash\texttt{S})\star\texttt{S}^{+}(v')]{E'\star\texttt{S}(v'), b, T} < stat' \, , \, \rho' > \qquad \texttt{S} \notin E'}{< \textbf{signal} \ \texttt{S=}\boxed{v} \ \textbf{in} \ stat \ \textbf{end} \, , \, \rho > \xrightarrow[\hat{E}]{E', b, T} < \textbf{signal} \ \texttt{S=}\boxed{v'} \ \textbf{in} \ stat' \ \textbf{end} \, , \, \rho >}$$

The second rule applies when the signal is not emitted. Thus, it is not received and the previous signal value is retained from the declaration:

$$\frac{< stat \, , \, \rho > \xrightarrow[(\hat{E}\backslash\texttt{S})\star\texttt{S}^{-}(v)]{E', b, T} < stat' \, , \, \rho' > \qquad \texttt{S} \notin E'}{< \textbf{signal} \ \texttt{S=}\boxed{v} \ \textbf{in} \ stat \ \textbf{end} \, , \, \rho > \xrightarrow[\hat{E}]{E', b, T} < \textbf{signal} \ \texttt{S=}\boxed{v} \ \textbf{in} \ stat' \ \textbf{end} \, , \, \rho >}$$

6.6. A simple example

As a simple example, we study two reactions of the following program:

```
module P:
input I(integer);
output O(combine integer with +);
signal S(integer) in
      present I then emit S(?I+1); emit O(1) end
   ||
      present S then
         emit O(?S); halt
      else
         await I
      end
end.
```

6.6.1. The input I is present

First assume that I is present with value 3. According to the intuitive semantics, we guess that S is emitted with value 4, and O is emitted twice with values 1 and 4, yielding 5 as a combined value. The derivative P′ has body nothing || halt. Hence the reaction should be

$$< \mathtt{P}\,,\,\phi > \overset{\mathtt{O}(4)}{\underset{\mathtt{I}^+(3)}{\longmapsto}} < \mathtt{P}'\,,\,\phi >$$

We sketch the proof, omitting expression evaluations and the memory part that are useless here. The body of the declaration of S must be analyzed in the guessed complete event $\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)$. Once this guess is made, the branches of the parallel statement can be analyzed in any order. We start by analyzing the first one. The emit rule yields

$$\mathtt{emit\ S(?I+1)}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{S}(4),tt,\emptyset}\ \mathtt{nothing}$$

$$\mathtt{emit\ O(1)}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{O}(1),tt,\emptyset}\ \mathtt{nothing}$$

The second sequence rule yields

$$\mathtt{emit\ S(?I+1);\ emit\ O(1)}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{S}(4)\cdot\mathtt{O}(1),tt,\emptyset}\ \mathtt{nothing}$$

The first rule for present yields

$$\mathtt{present\ I\ then}\ \cdots\ \mathtt{end}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{S}(4)\cdot\mathtt{O}(1),tt,\emptyset}\ \mathtt{nothing}$$

For the other half of the parallel, the rule for emit yields

$$\mathtt{emit\ O(?S)}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{O}(4),tt,\emptyset}\ \mathtt{nothing}$$

The axiom of halt and the second sequence rule yield

$$\mathtt{emit\ O(?S);\ halt}\ \xrightarrow[\mathtt{I}^+(3)\cdot\mathtt{S}^+(4)]{\mathtt{O}(4),ff,\emptyset}\ \mathtt{halt}$$

The first rule for **present** yields

$$\texttt{present S then} \cdots \texttt{end} \xrightarrow[\texttt{I}^+(3)\cdot\texttt{S}^+(4)]{\texttt{O}(4),tt,\emptyset} \texttt{halt}$$

The rule for parallel combines the emissions and terminations of both branches:

$$\cdots \texttt{||} \cdots \xrightarrow[\texttt{I}^+(3)\cdot\texttt{S}^+(4)]{\texttt{S}(4)\cdot\texttt{O}(5),ff,\emptyset} \texttt{nothing || halt}$$

We can finally apply the first local signal rule to S. It yields

$$\texttt{signal S=}\boxed{\perp}\texttt{ in} \cdots \texttt{end} \xrightarrow[\texttt{I}^+(3)]{\texttt{O}(5),ff,\emptyset} \texttt{signal S=}\boxed{4}\texttt{ in nothing || halt end}$$

Our guess about $\texttt{S}^+(4)$ is easily seen to be the only possible one. The reaction is deterministic.

6.6.2. The input signal I is absent

Here an intuitive analysis shows that S and O are not emitted. The parallel statement must be computed in the complete event $\texttt{I}^-(\perp)\cdot\texttt{S}^-(\perp)$. The first **present** statement emits no signal and terminates:

$$\texttt{present I then} \cdots \texttt{end} \xrightarrow[\texttt{I}^-(\perp)\cdot\texttt{S}^-(\perp)]{\epsilon,tt,\emptyset} \texttt{nothing}$$

For the second **present** statement, the **else** part is evaluated, using the **halt** and **watching** rules (according to the definition of **await**)

$$\texttt{await I} \xrightarrow[\texttt{I}^-(\perp)\cdot\texttt{S}^-(\perp)]{\epsilon,ff,\emptyset} \texttt{present I else await I end}$$

The parallel rule yields

$$\cdots \texttt{||} \cdots \xrightarrow[\texttt{I}^-(\perp)\cdot\texttt{S}^-(\perp)]{\epsilon,ff,\emptyset} \texttt{nothing || present I else await I end}$$

and we can apply the second local signal rule; O is not emitted, and the body is the resulting parallel statement above. As before, this is the only possible proof and the reaction is deterministic.

7. Determinism and program correctness

To establish a behavioral semantics proof, one has to guess the presence or absence of the local signals as well as their values. In the above example, there was a *unique* correct guess found using the intuitive semantics. To build simulators and compilers, we need a more effective process to determine the presence and values of signals. Such a process will be presented in the next section. In the present section, we study the *determinism* of the behavioral semantics.

According to our design rationale we want programs to be deterministic, hence to have a unique behavioral semantics for any input. However the sharing law is not enough to guarantee determinism: some programs have no semantics and some have several semantics. To simplify the discussion, all the examples given here will be closed programs without inputs and will yield problems on the execution of their initial reaction.

Let us start with pure signal examples. The following program P_1 has two semantics:

```
signal S in
   present S then emit S end
end
```

the signal **S** can be consistently considered as being emitted or not emitted. In both cases, the body becomes **signal S in nothing end**.

Changing **then** into **else** yields a program P_2 that has no behavioral semantics:

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

The signal **S** should be emitted if and only if **S** is not present, which is clearly nonsense.

The next example P_3 is similar to P_1 but involves two signals:

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

There are again two possible behavioral semantics: either **S1** and **S2** are not emitted, or they are both emitted. Changing **then** into **else** yields a program P_4 with another form of nondeterminism:

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

There are again two solutions: either **S1** is emitted and **S2** is not, or **S1** is not emitted and **S2** is emitted.

Other problems appear with valued signals. Consider the program P_5:

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

where **C** is an integer constant. When **S** is emitted its value v must satisfy $v = v + c$ if c is the value of **C**. This equation has no solution for $c \neq 0$ and infinitely many solutions if $c = 0$.

The above programs must clearly be rejected. Nondeterminism is, however, not a necessary condition for rejection; if we build P_6 by changing in P_5 the type **integer** into a type **triv** that has only one value, the semantics of P_6 becomes unique; the semantics of ESTEREL programs should not depend on type implementation details and both P_5 and P_6 should be rejected for a common reason.

All the problems are due to the fixpoint form of the local signal rule which does not respect the intuitive sequentiality constraints in program reactions. The "infinitely fast machine" on which we run programs should still behave sequentially as far as sequential control transmission is concerned. The first statement of a semicolon should be executed "before" its second statement, the test in a conditional or in a **present** statement should be computed "before" the **then** or **else** statements are started. As a consequence, the second statement of a semicolon (resp. the arms of a conditional or **present** statement) should not interfere with the execution of the first statement (resp. with the test). The programs P_1-P_4 above should be rejected for this reason. Similarly, the value of a signal should not be read "before" it is emitted, which is enough to reject P_5 and P_6.

Very similar problems exist in synchronous circuits: the logical behavior of circuit can be defined by fixpoint equations on transistor states, provided that the circuit does not contain *races*; races can appear whenever the sequential propagation of electrical currents is not compatible with the topological structure. The program P_2 above is the ESTEREL version of a NOT gate with its output plugged into its input.

More formally, we say that a signal is *written* by an **emit** statement and is *read* by a **present** test and by the **?** operator. We say that the "before" relation is generated by sequences and tests. The right correctness condition is as follows: a signal should not be read if it can still be written. This determines *correct executions*; a program is correct (w.r.t. some input) if it can be correctly executed. Note that a simpler correctness condition such as "a signal should not be written before it is read" is not enough to reject programs P_1, P_3, and P_4 above.

Determining if a signal can be written from a given position in a program is in general unfeasible (in particular because of uninterpreted conditionals). We cannot therefore obtain necessary and sufficient correctness conditions. What we need are sufficient conditions that are effective and efficient enough to be used in compilers.

- We can perform a *static dependency analysis* on signals, based on a structural control flow analysis; this analysis produces a *signal dependency graph* that contains an arrow from `S1` to `S2` whenever `S1` is read before `S2` is written in some possible execution path. Any program that yields a cyclic dependency graph must be rejected. This technique is proved correct in [23] and is used in the ESTEREL V2.2 compiler. It has two drawbacks: it sometimes rejects correct programs; when the graph is cyclic, the debugging information is limited to a list of cycles in the graph which is hard to exploit.
- Following Boussinot [16] and Gonthier [24], we can consider that an ESTEREL program is executed on a conventional sequential machine; the zero time reaction hypothesis then amounts to not observing its computation time. Signals are handled by a shared memory with the above read/write discipline. This technique is technically more involved but yields better results. It is used in the ESTEREL V3 compiler. We present it in the next section.

The notion of correctness presented above is local to each reaction. A program can be correct for some input and incorrect for another one. Assume that `I` is an input signal. Then the following program is incorrect if and only if `I` is received:

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

We say that a program is *locally correct* if it is correct for all inputs. A program can also become incorrect after several reactions:

```
signal S in
   await I;
   present S else emit S end
end
```

This program is locally correct; upon receiving `I`, it becomes the locally incorrect program P_2. We say that a program is *globally correct* if it is locally correct and if any sequence of reactions only produces locally correct programs.

8. An execution semantics

This section presents an execution semantics and its local correctness criterion. The underlying execution machine is a conventional sequential machine. Signals are implemented as *controlled shared variables* with a read/write discipline that ensures that any signal has a unique well-defined status (emitted or not emitted) and a unique well-defined value in any reaction. Each reaction is realized by an *execution*, which is a sequence of elementary *actions* (also called *microsteps* as in [28]), followed by an *expansion step* that prepares the program for the next reaction. The expansion step can only be applied once the execution is properly *halted*. A program is said to be *correct* w.r.t. an input if it has a halted execution for this input. That is, if the reaction can be completed while respecting the signal memory read/write discipline.

The actions are determined by structural operational rules. The parallel operator interleaves the actions of its branches, as in usual asynchronous models. The execution of correct programs is therefore non-deterministic. However reactions of correct programs are deterministic: our main theorem states that all halted executions of correct programs yield the same final results that agree with the behavioral semantics after expansion, and, furthermore, that the behavioral semantics of correct programs is unique. Therefore our read/write discipline is a correct realization of the sharing law where this law makes sense. The proofs are omitted and not even sketched; they require the introduction of many technical concepts that are outside the scope of this paper. See [24] for complete proofs.

8.1. The implementation of signals

The structure of the signal memory is similar to that of the standard memory of variables; there are primitives to allocate cells, to write in cells, and to reads cell values. However, cells also contain status information (written as an exponent) that describes the current status of their content. A cell can have four forms:

- $(\mathtt{S}^{\perp}{=}v)$: the cell S has not yet been written in the current reaction (its content v is that of the previous reactions).
- $(\mathtt{S}^{\dagger}{=}v)$: the cell S has already been written in the current reaction; its current content is v; it cannot yet be read since other write operations can still occur.
- $(\mathtt{S}^{+}{=}v)$ the cell S has been written in the current instant, and it can no longer be written. Its content v can be read as the current value ?S of the signal S, which is known to be present in the current reaction.
- $(\mathtt{S}^{-}{=}v)$ the cell S has not been written in the current reaction, and it can no longer be written. Its content v can be read as the current value ?S of the signal S, which is known to be absent in the current reaction.

Signal memories θ are constructed from the empty signal memory ϕ. If θ is a signal memory then $\theta.(\mathtt{S}^{x}{=}v)$, $x \in \{\perp, \dagger, +, -\}$, is also a signal memory. The read operation $\theta(\mathtt{S})$ can only be performed if the cell status is $+$ or $-$:

$$\theta.(\mathtt{S}^{x}{=}v)(\mathtt{S}) = v \text{ if } x \in \{+,-\}$$
$$\theta.(\mathtt{S}_1^{x}{=}v)(\mathtt{S}) = \theta(\mathtt{S}) \text{ if } \mathtt{S}_1 \neq \mathtt{S}$$

Similarly, one can test for the status of a signal in a memory:

$$\mathtt{S}^{x} \in \theta.(\mathtt{S}_1^{y}{=}v) \quad \text{iff} \quad \begin{cases} y = x & \text{if } \mathtt{S}_1 = \mathtt{S} \\ \mathtt{S}^{x} \in \theta & \text{if } \mathtt{S}_1 \neq \mathtt{S} \end{cases}$$

The result of writing a value v in a cell S of a signal memory θ is denoted by $\theta[\mathtt{S} \leftarrow v]$. Writing is done in the most recently allocated occurrence of a signal, so

$$\theta.(\mathtt{S}_1^{x}{=}v')[\mathtt{S} \leftarrow v] = (\theta[\mathtt{S} \leftarrow v]).(\mathtt{S}_1^{x}{=}v') \quad \text{for } \mathtt{S}_1 \neq \mathtt{S}$$

There are two cases for the actual writing. If the memory has not yet been written, the new value replaces the old value and the memory state goes from $\mathtt{S}^{\perp}$ to $\mathtt{S}^{\dagger}$; if the signal has already been written, the new value is combined with the old one using the combination function associated with S.

$$\theta.(\mathtt{S}^{\perp}{=}v)[\mathtt{S} \leftarrow v'] = \theta.(\mathtt{S}^{\dagger}{=}v')$$
$$\theta.(\mathtt{S}^{\dagger}{=}v)[\mathtt{S} \leftarrow v'] = \theta.(\mathtt{S}^{\dagger}{=}v \star_{\mathtt{S}} v')$$

There is no way to write in a cell of the form $\mathtt{S}^{+}$ or $\mathtt{S}^{-}$. The potential rules below show when a memory goes from $\mathtt{S}^{\perp}$ to $\mathtt{S}^{-}$ or from $\mathtt{S}^{\dagger}$ to $\mathtt{S}^{+}$.

To relate the execution semantics to the behavioral semantics, we have to relate signal memories to complete input events and output events. Given a program of input sort $\mathcal{I}$ and of output sort $\mathcal{O}$, we associate a signal memory $\theta_{\hat{\mathtt{I}}}$ with any complete input event $\hat{\mathtt{I}}$. This memory allocates the signals in $\mathcal{I}$ and $\mathcal{O}$; it is composed of the cells

$(\mathtt{S}^{x}{=}v)$ for all $\mathtt{S}^{x}(v) \in \hat{\mathtt{I}}$

$(\mathtt{S}^{\perp}{=}\perp)$ for all output signals $\mathtt{S} \in \mathcal{O}$

The order of the cells in $\theta_{\mathtt{I}}$ is immaterial. Conversely, we associate to any final signal memory θ an output event $\mathtt{O}_{\theta}$ containing all $\mathtt{S}(v)$ such that $(\mathtt{S}^{\dagger}{=}v)$ is a cell of θ (thus the sort of $\mathtt{O}_{\theta}$ is the set of all $\mathtt{S} \in \mathcal{O}$ such that $\mathtt{S}^{\dagger} \in \theta$; an output signal never gets gets a status $+$ or $-$, but this does not matter, since it is never read internally).

8.2. Computing actions

The execution semantics is given by a set of rules that determine actions of the form:

$$< stat, \rho, \theta > \longrightarrow < stat', \rho', \theta' >$$

The action rules determine how the statements update the memories. Whenever writing triples $< stat, \rho, \theta >$, we assume that ρ and θ allocate the free variables and signals of *stat*.

We also need three auxiliary sets of rules, which are presented either as rules with arrows or as equations with equal signs, the difference being somewhat immaterial here:

- *Expression evaluation rules* govern evaluations of the form

  $$< exp, \rho, \theta > \rightarrow v$$

  The signal memory discipline expresses that the "**?S**" operator can only be evaluated when the exponent of **S** in θ is $+$ or $-$.
- *Termination rules* compute a partial function $\mathcal{T}(stat) = \langle b, T \rangle$ where b is a termination boolean and T is a set of exited trap labels, as in the behavioral semantics. Termination rules are used only on terms that can perform no action. Execution and termination were treated together in the behavioral semantics; here it is simpler to use separate rules for execution and termination. Termination rules are used in two places: first, to detect completion of the current reaction; second, to start executing the second statement of a sequence when its first statement is terminated (a terminated statement has the form **nothing**, **nothing || nothing**, **trap T in exit T end**, etc).
- *Potential rules* compute the *potential* $\pi(stat)$ of a statement, that is the set of signals that *stat* can emit in some execution. Potentials are used to change the status of local signals in the signal memory: a signal goes from $\mathtt{S}^{\perp}$ to $\mathtt{S}^{-}$ or from $\mathtt{S}^{\dagger}$ to $\mathtt{S}^{+}$ when it can no longer be emitted, hence, when it does not belong to the potential of the current term.

8.2.1. Action rules

An assignment can act if its expression can be evaluated:

$$\frac{< exp, \rho, \theta > \rightarrow v}{< \mathtt{X} := exp, \rho, \theta > \longrightarrow < \mathbf{nothing}, \rho[\mathtt{X} \leftarrow v], \theta >}$$

The rule for procedure calls is similar and left to the reader.

An emission can be executed when its expression can be computed; it updates the signal memory θ as described earlier:

$$\frac{< exp, \rho, \theta > \rightarrow v}{< \mathbf{emit}\ \mathtt{S}(exp), \rho, \theta > \longrightarrow < \mathbf{nothing}, \rho, \theta[\mathtt{S} \leftarrow v] >}$$

If the first statement of a sequence can act, so can the sequence:

$$\frac{< stat_1, \rho, \theta > \longrightarrow < stat'_1, \rho'_1, \theta'_1 >}{< stat_1\ ;\ stat_2, \rho, \theta > \longrightarrow < stat'_1\ ;\ stat_2, \rho'_1, \theta'_1 >}$$

If the first statement of the sequence is terminated, the second one can act (this is the only rule that connects execution and termination rules):

$$\frac{\mathcal{T}(stat_1) = \langle tt, \emptyset \rangle \qquad <stat_2, \rho, \theta> \;\longrightarrow\; <stat_2', \rho_2', \theta_2'>}{<stat_1 \;;\; stat_2, \rho, \theta> \;\longrightarrow\; <stat_2', \rho_2', \theta_2'>}$$

A loop can act iff its body can; it then unfolds:

$$\frac{<stat, \rho, \theta> \;\longrightarrow\; <stat', \rho', \theta'>}{<\texttt{loop}\ stat\ \texttt{end}, \rho, \theta> \;\longrightarrow\; <stat' \;;\; \texttt{loop}\ stat\ \texttt{end}, \rho', \theta'>}$$

A conditional acts when it can compute its condition. It selects the corresponding branch:

$$\frac{<exp, \rho, \theta> \;\rightarrow\; true}{<\texttt{if}\ exp\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \rho, \theta> \;\longrightarrow\; <stat_1, \rho, \theta>}$$

$$\frac{<exp, \rho, \theta> \;\rightarrow\; false}{<\texttt{if}\ exp\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \rho, \theta> \;\longrightarrow\; <stat_2, \rho, \theta>}$$

Similarly, a **present** statement can act as soon as its signal has status + or −; it selects the corresponding branch (it is essential here that "**present**" statement can never be applied to output signals, which never get the + or − status):

$$\frac{\texttt{S}^+ \in \theta}{<\texttt{present S then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \rho, \theta> \;\longrightarrow\; <stat_1, \rho, \theta>}$$

$$\frac{\texttt{S}^- \in \theta}{<\texttt{present S then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \rho, \theta> \;\longrightarrow\; <stat_2, \rho, \theta>}$$

A **watching** statement acts as its body (remember that the temporal guard does not take effect immediately):

$$\frac{<stat, \rho, \theta> \;\longrightarrow\; <stat', \rho', \theta'>}{<\texttt{do}\ stat\ \texttt{watching S}, \rho, \theta> \;\longrightarrow\; <\texttt{do}\ stat'\ \texttt{watching S}, \rho', \theta'>}$$

A parallel statement can act as any of its branches (standard interleaving semantics):

$$\frac{<stat_1, \rho, \theta> \;\longrightarrow\; <stat_1', \rho_1', \theta_1'>}{<stat_1 \;||\; stat_2, \rho, \theta> \;\longrightarrow\; <stat_1' \;||\; stat_2, \rho_1', \theta_1'>}$$

$$\frac{<stat_2, \rho, \theta> \;\longrightarrow\; <stat_2', \rho_2', \theta_2'>}{<stat_1 \;||\; stat_2, \rho, \theta> \;\longrightarrow\; <stat_1 \;||\; stat_2', \rho_2', \theta_2'>}$$

A **trap** statement can act iff its body can:

$$\frac{<stat, \rho, \theta> \;\longrightarrow\; <stat', \rho', \theta'>}{<\texttt{trap T in } stat \texttt{ end}, \rho, \theta> \;\longrightarrow\; <\texttt{trap T in } stat' \texttt{ end}, \rho', \theta'>}$$

A local variable declaration can execute its body after binding its variable. The value of the variable is kept in the variable declaration, as in the behavioral semantics:

$$\frac{<stat, \rho.\texttt{(X=}v\texttt{)}, \theta> \;\longrightarrow\; <stat', \rho'.\texttt{(X=}v'\texttt{)}, \theta>}{<\texttt{var X=}\boxed{v}\texttt{ in } stat \texttt{ end}, \rho, \theta> \;\longrightarrow\; <\texttt{var X=}\boxed{v'}\texttt{ in } stat' \texttt{ end}, \rho', \theta>}$$

A local signal declaration binds its signal in the current signal memory θ. As in the behavioral semantics, the binding is stored in the signal declaration; here we store the value and the current signal status. If the signal can no longer be emitted by the body, it is set to $\texttt{S}^+$ if it was $\texttt{S}^\dagger$ and to $\texttt{S}^-$ if it was $\texttt{S}^\perp$; this is the only place where potentials are used. After execution, the new signal state is stored in the signal declaration. Let us define an auxiliary operation:

$$\pm(\texttt{S}, stat, x) = \begin{cases} + & \text{if } x = \dagger \text{ and } \texttt{S} \notin \pi(stat) \\ - & \text{if } x = \perp \text{ and } \texttt{S} \notin \pi(stat) \\ x & \text{otherwise} \end{cases}$$

The local signal rule is then:

$$\frac{y = \pm(\texttt{S}, stat, x) \qquad <stat, \rho, \theta.\texttt{(S}^y\texttt{=}v\texttt{)}> \;\longrightarrow\; <stat', \rho', \theta'.\texttt{(S}^z\texttt{=}v'\texttt{)}>}{<\texttt{signal S}^x\texttt{=}\boxed{v}\texttt{ in } stat \texttt{ end}, \rho, \theta> \;\longrightarrow\; <\texttt{signal S}^z\texttt{=}\boxed{v'}\texttt{ in } stat' \texttt{ end}, \rho', \theta'>}$$

8.2.2. Expression evaluation

A variable is evaluated as usual:

$$<\texttt{X}, \rho, \theta> \;\rightarrow\; \rho(\texttt{X})$$

A signal value can be accessed only when it has status $+$ or $-$:

$$\frac{\texttt{S}^- \in \theta \quad \text{or} \quad \texttt{S}^+ \in \theta}{<\texttt{?S}, \rho, \theta> \;\rightarrow\; \theta(\texttt{S})}$$

An operator can operate as soon as its two arguments are computed:

$$\frac{<exp_1, \rho, \theta> \;\rightarrow\; v_1 \qquad <exp_2, \rho, \theta> \;\rightarrow\; v_2}{<exp_1 \text{ op } exp_2, \rho, \theta> \;\rightarrow\; v_1 \text{ op } v_2}$$

8.2.3. Termination rules

The termination function can only be computed when execution is no longer possible. Assignments, `emit` statements, conditionals, and `present` statements that can directly act have no termination rule. A termination $\mathcal{T}(stat)$ is a pair of a termination status b and of an exited label set T. We say that *stat is terminated* if $b = tt$ (then $T = \emptyset$); we say that *stat is halted* if $b = ff$ and $T = \emptyset$.

Clearly, **nothing** is terminated and **halt** is halted:

$$\mathcal{T}(\texttt{nothing}) = \langle tt, \emptyset \rangle$$

$$\mathcal{T}(\texttt{halt}) = \langle ff, \emptyset \rangle$$

For sequences, there are two cases. If the first statement has a termination but is not terminated, so is the sequence; if the first statement is terminated and if the termination of the second statement is defined, the sequence has the same termination as the second statement (this rules handles actionless sequences such as "**nothing; nothing**", "**nothing; exit T**", etc.):

$$\mathcal{T}(stat_1\texttt{;}\ stat_2) = \begin{cases} \langle ff, T \rangle & \text{if } \mathcal{T}(stat_1) = \langle ff, T \rangle \\ \langle b, T \rangle & \text{if } \mathcal{T}(stat_1) = \langle tt, \emptyset \rangle \text{ and } \mathcal{T}(stat_2) = \langle b, T \rangle \end{cases}$$

The body of a loop must not be terminated. The termination is that of the body.

$$\mathcal{T}(\texttt{loop}\ stat\ \texttt{end}) = \langle ff, T \rangle \quad \text{if } \mathcal{T}(stat) = \langle ff, T \rangle$$

A **watching** statement has the same termination as its body:

$$\mathcal{T}(\texttt{do}\ stat\ \texttt{watching S}) = \mathcal{T}(stat)$$

As in the behavioral semantics, the termination of a parallel statement is obtained from the termination of its branches:

$$\mathcal{T}(stat_1\ \texttt{||}\ stat_2) = \langle b_1 \wedge b_2, T_1 \cup T_2 \rangle \quad \text{if } \mathcal{T}(stat_1) = \langle b_1, T_1 \rangle \text{ and } \mathcal{T}(stat_2) = \langle b_2, T_2 \rangle$$

The terminations of a **trap** and of an **exit** are computed as in the behavioral semantics:

$$\mathcal{T}(\texttt{trap T in}\ stat\ \texttt{end}) = \langle b \vee (T = \{\texttt{T}\}),\ T - \{\texttt{T}\} \rangle \quad \text{if } \mathcal{T}(stat) = \langle b, T \rangle$$

$$\mathcal{T}(\texttt{exit T}) = \langle ff, \{\texttt{T}\} \rangle$$

Local variable and signal declarations simply propagate the termination of their body:

$$\mathcal{T}(\texttt{var X in}\ stat\ \texttt{end}) = \mathcal{T}(stat)$$

$$\mathcal{T}(\texttt{signal S in}\ stat\ \texttt{end}) = \mathcal{T}(stat)$$

8.2.4. Potential rules

The *potential* $\pi(stat)$ of a statement *stat* is the set of signals it can emit in some of its executions. We compute potentials by a simple structural control flow analysis. To perform the structural induction, we compute *extended potentials* $\hat{\pi}(stat) = \langle \pi, b, T \rangle$ where π is the potential of *stat*, where the boolean termination status b is true iff *stat* can terminate in some execution path, and where the exited label set T is the set of trap labels that *stat* can exit in some execution path. The potential of a statement *stat* is then defined by

$$\pi(stat) = \pi \quad \text{iff} \quad \hat{\pi}(stat) = \langle \pi, b, T \rangle$$

The computed termination status $\langle b, T\rangle$ is similar to the termination status computed by the termination rules; it is really a statically computed approximation of it, which represents the termination information that we can get without executing a statement*. A termination status like $\langle tt, \{\mathtt{T}\}\rangle$ can never be obtained from termination rules; it can however be obtained when computing potentials, since it represents the information that a statement can terminate and can also exit a trap labeled `T`, as in "**`if`** *exp* **`then nothing else exit`** `T` **`end`**".∎

The extended potentials of **`nothing`**, **`halt`**, and assignments are trivial:

$$\hat{\pi}(\mathtt{nothing}) = \langle \emptyset, tt, \emptyset\rangle$$

$$\hat{\pi}(\mathtt{halt}) = \langle \emptyset, ff, \emptyset\rangle$$

$$\hat{\pi}(\mathtt{X := } exp) = \langle \emptyset, tt, \emptyset\rangle$$

An **`emit`** statement adds its signal to the potential and terminates:

$$\hat{\pi}(\mathtt{emit\ S(}exp\mathtt{)}) = \langle \{\mathtt{S}\}, tt, \emptyset\rangle$$

If the first statement of a sequence cannot terminate, the extended potential of the sequence is that of the first statement:

$$\frac{\hat{\pi}(stat_1) = \langle \pi_1, ff, T_1\rangle}{\hat{\pi}(stat_1\,;\ stat_2) = \langle \pi_1, ff, T_1\rangle}$$

If the first statement of a sequence can terminate, the extended potential of the sequence is obtained by taking the union of the potentials of the first and second statements, the termination boolean of the second statement, and the labels potentially exited by both statements:

$$\frac{\hat{\pi}(stat_1) = \langle \pi_1, tt, T_1\rangle \qquad \hat{\pi}(stat_2) = \langle \pi_2, b_2, T_2\rangle}{\hat{\pi}(stat_1\,;\ stat_2) = \langle \pi_1 \cup \pi_2, b_2, T_1 \cup T_2\rangle}$$

The extended potential of a loop is obtained from that of its body by returning the *ff* boolean termination status, since a loop can never terminate:

$$\frac{\hat{\pi}(stat) = \langle \pi, b, T\rangle}{\hat{\pi}(\mathtt{loop}\ stat\ \mathtt{end}) = \langle \pi, ff, T\rangle}$$

For a conditional or a **`present`** statement, the potential is the union of the potentials of the branches. The conditional can terminate iff one of its branches can and a label can be exited iff it can be exited by one arm:

$$\frac{\hat{\pi}(stat_1) = \langle \pi_1, b_1, T_1\rangle \qquad \hat{\pi}(stat_2) = \langle \pi_2, b_2, T_2\rangle}{\hat{\pi}(\mathtt{if}\ exp\ \mathtt{then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end}) = \langle \pi_1 \cup \pi_2, b_1 \vee b_2, T_1 \cup T_2\rangle}$$

* The potential rules implemented in the Esterel v3 system are a bit finer than the one presented here, see [24].

$$\frac{\hat{\pi}(stat_1) = \langle \pi_1, b_1, T_1 \rangle \qquad \hat{\pi}(stat_2) = \langle \pi_2, b_2, T_2 \rangle}{\hat{\pi}(\texttt{present S then } stat_1 \texttt{ else } stat_2 \texttt{ end}) = \langle \pi_1 \cup \pi_2, b_1 \vee b_2, T_1 \cup T_2 \rangle}$$

The extended potential of a **watching** is that of its body:

$$\hat{\pi}(\texttt{do } stat \texttt{ watching S}) = \hat{\pi}(stat)$$

The potential of a parallel is the union of the potentials of its branches; a parallel can terminate if both branches can terminate; it can exit a label if one arm can:

$$\frac{\hat{\pi}(stat_1) = \langle \pi_1, b_1, T_1 \rangle \qquad \hat{\pi}(stat_2) = \langle \pi_2, b_2, T_2 \rangle}{\hat{\pi}(stat_1 \texttt{ || } stat_2) = \langle \pi_1 \cup \pi_2, b_1 \wedge b_2, T_1 \cup T_2 \rangle}$$

The potential of a **trap** statement is that of its body; the **trap** statement can terminate if its body can terminate or exit T; the label T is removed from the exited label set:

$$\frac{\hat{\pi}(stat) = \langle \pi, b, T \rangle}{\hat{\pi}(\texttt{trap T in } stat \texttt{ end}) = \langle \pi\, , b \vee (\texttt{T} \in T)\, , T - \{\texttt{T}\} \rangle}$$

An **exit** generates the corresponding exited label:

$$\hat{\pi}(\texttt{exit T}) = \langle \emptyset, \mathit{ff}, \{\texttt{T}\} \rangle$$

A local variable declaration does not affect potentials:

$$\hat{\pi}(\texttt{var X in } stat \texttt{ end}) = \hat{\pi}(stat)$$

Finally, a local signal declaration removes its signal from the potential:

$$\frac{\hat{\pi}(stat) = \langle \pi, b, T \rangle}{\hat{\pi}(\texttt{signal S in } stat \texttt{ end}) = \langle \pi \backslash \texttt{S}, b, T \rangle}$$

8.3. The confluence properties of executions

A reaction is realized by a finite well-terminated execution sequence. As in the behavioral semantics, we shall always assume that a program body is followed in sequence by a **halt** statement, so that it can halt but never terminate. We shall also assume that all local signals have initial status and value $\perp$, i.e. that all local signal declarations **signal S in** *stat* **end** are initially replaced by **signal** $\texttt{S}^{\perp}\texttt{=}\boxed{\perp}$ **in** *stat* **end**.

Definition: An *execution* is a sequence

$$< stat, \rho, \theta > \;\longrightarrow\; < stat_1, \rho_1, \theta_1 > \;\longrightarrow\; \cdots \;\longrightarrow\; < stat_n, \rho_n, \theta_n >$$

It is *maximal* if $< stat_n, \rho_n, \theta_n >$ has no further action. It is *halted* if $\mathcal{T}(stat_n) = \langle \mathit{ff}, \emptyset \rangle$. For any halted execution, $< stat_n, \rho_n, \theta_n >$ is called the *result* of the execution.

Halted execution sequences do not always exist. A statement such as

```
loop X:=X+1 end
```

has neither maximal nor halted execution sequences. The programs P_1-P_6 of section 7 have no halted execution sequences. Consider for example P_1, that is the statement

```
signal S in
   present S then emit S end
end
```

in the empty variable and signal memories. The potential of the `present` statement is $\{\mathtt{S}\}$. Therefore the signal execution rule imposes to find an execution of the triple

$$<\mathtt{present\ S\ then\ emit\ S\ end}, \phi, \phi.(\mathtt{S}^{\perp})>$$

The rule for **present** cannot be applied to such a triple, since it requires the exponent to be $+$ or $-$. No action step is possible. Hence the only execution of P_1 is the empty one. But the termination of P_1 is undefined, since there is no termination rule for `emit`. Therefore the only possible execution is not halted. The programs P_2-P_6 are rejected in the same way, using the rule for **?** instead of the rule for **present** for P_5 and P_6.

When halted executions do exist, there can be several executions for a reaction, since the parallel execution rule is non-deterministic: if the two branches of a parallel can act, the parallel can act as any of them. The *confluence properties* ensure that all these sequences yield the same result, or in other words, that the order of actions is immaterial. This first property is classically called strong confluence [30]:

Theorem 1 *(strong confluence theorem)*: For any two distinct actions

$$< stat, \rho, \theta > \longrightarrow < stat'_1, \rho'_1, \theta'_1 > \text{ and } < stat, \rho, \theta > \longrightarrow < stat'_2, \rho'_2, \theta'_2 >$$

there exists $stat''$, ρ'' and θ'' such that

$$< stat'_1, \rho'_1, \theta'_1 > \longrightarrow < stat'', \rho'', \theta'' > \text{ and } < stat'_2, \rho'_2, \theta'_2 > \longrightarrow < stat'', \rho'', \theta'' >$$

From this theorem, it is easy to deduce the global confluence property of halted execution sequences, that really expresses the determinism of reactions:

Corollary 2 *(global confluence theorem)*: Let P be a program of body $stat$, let $\hat{\mathtt{I}}$ be an input event, let $\theta = \theta_{\hat{\mathtt{I}}}$ be the corresponding signal memory. If $< stat, \phi, \theta >$ has a halted execution sequence, then it has no infinite execution sequence, all its maximal sequences are halted, and they all yield the same result.

8.4. The expansion step

Unlike in the behavioral semantics, the statement that appears in the result of a reaction is not directly ready for the following reaction. Three things must be done first:

- Reset the local signal status to $\perp$ in local signal declarations.
- Turn "**do** *stat* **watching S**" statements into "**present S else do** *stat* **watching S end**". This expansion was done on the fly in the behavioral semantics (see the rule for **watching**). It is easier to do at the end of execution sequences in the execution semantics.
- Perform some cleanup: for example, a halted term such as "**nothing;halt**" must be transformed into "**halt**" to match the behavioral semantics.

The following equations describe the expansion step:

$$\mathcal{E}(\mathtt{nothing}) = \mathtt{nothing}$$

$$\mathcal{E}(\mathtt{halt}) = \mathtt{halt}$$

$$\mathcal{E}(stat_1\texttt{; }stat_2) = \begin{cases} \mathcal{E}(stat_2) & \text{if } \mathcal{T}(stat_1) = \langle tt, \emptyset\rangle \\ \mathcal{E}(stat_1)\texttt{;}\, stat_2 & \text{otherwise} \end{cases}$$

$$\mathcal{E}(\texttt{loop } stat \texttt{ end}) = \mathcal{E}(stat)\texttt{; loop } stat \texttt{ end}$$

$$\mathcal{E}(\texttt{do } stat \texttt{ watching S}) = \texttt{present S else do } \mathcal{E}(stat) \texttt{ watching S}$$

$$\mathcal{E}(stat_1 \texttt{ || } stat_2) = \mathcal{E}(stat_1) \texttt{ || } \mathcal{E}(stat_2)$$

$$\mathcal{E}(\texttt{trap T in } stat \texttt{ end}) = \begin{cases} \texttt{nothing} & \text{if } \mathcal{T}(\texttt{trap T in } stat \texttt{ end}) = \langle tt, \emptyset\rangle \\ \texttt{trap T in } \mathcal{E}(stat) \texttt{ end} & \text{otherwise} \end{cases}$$

$$\mathcal{E}(\texttt{var X=}\boxed{v}\texttt{ in } stat \texttt{ end}) = \texttt{var X=}\boxed{v}\texttt{ in } \mathcal{E}(stat) \texttt{ end}$$

$$\mathcal{E}(\texttt{signal S}^{x}\texttt{=}\boxed{v}\texttt{ in } stat \texttt{ end}) = \texttt{signal S}^{\perp}\texttt{=}\boxed{v}\texttt{ in } \mathcal{E}(stat) \texttt{ end}$$

8.5. Equivalence of the behavioral and execution semantics

We are now in position to state our main theorem: the execution semantics matches the behavioral semantics. We first associate a reaction with a halted execution sequence followed by an expansion step:

Definition: Let P be a program of body $stat$, let $\hat{\texttt{I}}$ be a complete input event. We say that P is *causally correct w.r.t.* $\hat{\texttt{I}}$ and write

$$P \overset{\texttt{O}}{\underset{\hat{\texttt{I}}}{\Longrightarrow}} P'$$

if $stat$ has a halted execution sequence of the form

$$< stat, \phi, \theta_{\hat{\texttt{I}}} > \;\longrightarrow\; < stat', \phi, \theta' >$$

such that the body of P' is equal to $\mathcal{E}(stat')$ and that $\texttt{O}$ is equal to the output event $E_{\theta'}$ determined by θ'.

Theorem 3 *(correctness and determinism theorem)*: Let P be a program, let $\hat{\texttt{I}}$ be a complete input event such that P is causally correct in $\theta_{\hat{\texttt{I}}}$. There exist a unique program P' and a unique output event $\texttt{O}$ such that $P \overset{\texttt{O}}{\underset{\hat{\texttt{I}}}{\longmapsto}} P'$ and $P \overset{\texttt{O}}{\underset{\hat{\texttt{I}}}{\Longrightarrow}} P'$.

8.6. An execution example

To illustrate the execution semantics, we sketch the executions of the example of section 6.6, that is:

```
module P:
input I(integer);
output O(combine integer with +);
signal S(integer) in
      present I then emit S(?I+1); emit O(1) end
   ||
      present S then
         emit O(?S); halt
      else
         await I
      end
end.
```

8.6.1. The input I is present

Assume that I is present with value 3. The initial signal memory is ϕ.(I^{+}=3).(O$^{\perp}$=$\perp$). When entering the body of the local signal declaration, we compute the potential $\{$S, O$\}$ and therefore add (S$^{\perp}$=$\perp$) to the signal memory. The "**present S**" statement cannot act; we must execute the "**present I**" statement first. This **present** statement selects its first branch. This ends the first action, which yields the term:

```
signal $S^{\perp}=\boxed{\perp}$ in
      emit S(?I+1); emit O(1)
   ||
      present S then
         emit O(?S); halt
      else
         await I
      end
end
```

The potential is unchanged. One must execute the first "**emit**" statement of the sequence. The S cell of the signal memory becomes (S†=4). The second action therefore yields the following term result:

```
signal $S^{\dagger}=\boxed{4}$ in
      nothing; emit O(1)
   ||
      present S then
         emit O(?S); halt
      else
         await I
      end
end
```

The potential of the body is now $\{$O$\}$. Hence the S cell of the signal memory on which the body executes is (S^{+}=4). Since **nothing** is terminated, both the "**emit O**" and the "**present S**" statement can be executed. We have to choose one of them. Let us choose the "**present**" one. The third action yields the term

```
signal $S^{+}=\boxed{4}$ in
      nothing; emit O(1)
   ||
      emit O(?S); halt
end
```

Then the two emissions of O can be executed in any order. After the two corresponding actions, we get the halted term

```
signal $S^{+}=\boxed{5}$in
      nothing
   ||
      nothing; halt
end
```

in the signal memory $\{\phi$.(I^{+}=3).(O†=5). Therefore O(5) is output. The expansion step clears the status of S and the second **nothing**, leaving the term

```
signal S$^\perp$=$\boxed{5}$in
      nothing
   ||
      halt
end
```

8.6.2. The input I is absent

We enter the local signal declaration as before. In the first action, the "**present I**" statement selects its second branch "**nothing**", leaves the signal memory unchanged and yields the term

```
signal S$^\perp$=$\boxed{\perp}$ in
      nothing
   ||
      present S then
         emit O(?S); halt
      else
         do halt watching I
      end
end
```

The potential of the body becomes $\{\texttt{O}\}$, the body's S signal memory becomes $(\texttt{S}^{-}=\perp)$. The second action selects the "**else**" branch of the "**present**" statement, yielding the halted term

```
signal S$^-$=$\boxed{\perp}$ in
      nothing
   ||
      do halt watching I
end
```

The execution is finished. The expansion step prepares the term for the next reaction, transforming it into

```
signal S$^\perp$=$\boxed{\perp}$ in
      nothing
   ||
      present I else
         do halt watching I
      end
end
```

9. Compiling Esterel programs into deterministic automata

The execution semantics is effective and can be used as a basis for building *interpreters* of the language. Such interpreters exist in the ESTEREL V2.2 and ESTEREL V3 systems. Their performances are reasonable (say reaction times of 1/100 to 1/10 second), but not sufficient for real-time applications. In this section, we show how to produce very efficient sequential automata that are behaviorally equivalent to a source ESTEREL program. The algorithm is similar to the Brzozowski algorithm for translating regular expressions into finite automate, see [17, 10]. We discuss the pros and cons of this compiling technique and the practical validity of the synchrony hypothesis.

9.1. Compiling pure synchronization programs

We first study the simple case of *pure synchronization programs*, that is of programs that only contain pure signals (no variables, constants, nor values of any kind). For such programs, only the presence or absence of signals matter. The memory parts are useless in the semantics equations.

A pure synchronization program P has finitely many possible input events $\hat{I}$, corresponding at most to all sets of input signals. Thus P has finitely many immediate derivatives P′ such that $\mathtt{P} \xmapsto[\hat{I}]{O} \mathtt{P}'$. The next theorem shows that this finiteness property also holds for general derivatives of P, that is for programs P′ obtained after arbitrary long sequences of reactions

$$\mathtt{P} \xmapsto[\hat{I}_0]{O_0} \mathtt{P}_1 \xmapsto[\hat{I}_1]{O_1} \mathtt{P}_2 \xmapsto[\hat{I}_2]{O_2} \cdots \mathtt{P}_n \xmapsto[\hat{I}_n]{O_n} \mathtt{P}'$$

Theorem 4: Any pure synchronization program has only finitely many derivatives.

We can therefore completely replace a program P by its *reaction graph* considered as a finite state automaton with derivatives as states. The automaton starts in state P. Given a current state P′ and an input $\hat{I}$, the automaton emits an output event O and goes to state P″ iff $P' \overset{O}{\underset{\hat{I}}{\longmapsto}} P''$. The reaction graph can be fully computed *at compile time*; at run-time, the program texts are useless and can simply be replaced by state numbers. For example, the program

```
module P :
input I;
output O;
signal S in
   emit O;
   loop
         await I;
         await I do emit S end
      ||
         await S do emit O end
   end
end.
```

yields the automaton

```
state 0 :
   ϵ -> output O; goto 1
   I -> output O; goto 1
state 1 :
   ϵ -> goto 1
   I -> goto 2
state 2 :
   ϵ -> goto 2
   I -> output O; goto 1
```

where "output O" means "emit the output signal O" and "goto n" means "the current reaction is over, treat the next reaction from state n". The body of state 2 is

```
signal S in
   [
      present I else await I end; emit S
   ||
      present S else await S end; emit O
   ];
   loop
         await I;
         await I do emit S end
      ||
         await S do emit O end
   end
end
```

Notice that our algorithm translates a concurrent program into a sequential one. Concurrency is treated at compile time, not at run time. Notice also that the local signal S *completely disappears* in the compiled code. A local signal acts as an auxiliary non-terminal in a parser generator [32]. Writing modular programs that use many local signals for better architecture yields no run-time overhead.

9.2. Compiling general Esterel programs

Although they manipulate data, general Esterel programs can be translated to automata almost as simply as pure synchronization ones. The key idea is to keep the memory actions formal at compile time since they can only be performed at run-time. An automaton transition then consists of a sequence of run-time actions (more precisely of a tree of actions due to conditionals).

The actions operate on globally allocated variables. An object code variable is allocated for each explicitly declared variable or valued signal of the source code. Other implicit variables are allocated, such as occurrence or repeat loop counters, booleans indicating input signal presence, etc. The following actions can appear in transitions:

- *assignments*: they are generated by source assignments, by explicit or implicit variable initializations, and by valued signal emissions (a source `emit` statement generates an assignment to the signal's variable).
- *external procedure calls*: they are generated simply by source procedure calls.
- *tests*. Three kinds of tests are generated: boolean expression tests appearing in conditionals, decrement-and-tests of internal counter variables (for signal occurrences or **repeat** statements), and tests for presence of input signals (instead of generating a transition per input event as suggested before, it is better in practice to test for the presence of input signals on call-by-need basis).
- *output signal emissions*: they transmit the emission of a signal to the program's environment.

The actions are gathered in an action table, the transitions referring to entries in that table. Notice that the synchronization needed for the internal communication of values is simply implemented by the *order* of the assignments in the transitions; thus it generates *no code*.

The reflex game program generates a 6 state automaton that is presented in Annex 2. See [8] for more details.

9.3. Time and space efficiency of the generated code

The run-time efficiency of the automaton object code is obvious. Only actions that *must* be done at run-time appear in the transitions. As we already mentioned, communication by pure internal signals generate no code — which is a way to say that they are *really* infinitely fast. Communication by valued local signals are done by assignments and therefore as fast as possible. There is no process handling overhead.

If not infinitely fast, the generated code is essentially as fast as it can be, comparable to carefully hand-written low level code. By itself, this *justifies* the synchrony hypothesis: if our code is not fast enough for a given target machine then there might be no implementation of the desired reactive program on this machine, at least at reasonable cost. Furthermore, since the code is sequential, the reaction speed is *measurable* on any given processor. The user can verify whether its speed requirements will be satisfied at run-time.

We have no general result concerning *space* efficiency; this question is less clear. As for grammar parsers, it is easy to build examples that produce an exponential blow-up in size. In practice, the obtained automata tend to be of reasonable size; they are almost always minimal or close to minimal (the same property holds for Brzozowski's original algorithm — the reason is not yet completely understood). When produced in a host language, the automaton is generated in a compact byte code form [8]. For example, the reflex game automaton occupies about 300 bytes, and the automaton of the wristwatch program presented in [9] occupies 2.5 Kbytes.

There can be two causes of size explosion: the number of transitions from each state or the number of states. The *input signal relations* presented in section 2 allow to control the first case. If a program has n input signals, it has 2^n possible input events, i.e. sets of input signals. Input relations dramatically reduce this number. For example, if all signals are declared to be incompatible, the number of input events decreases to $n + 1$ (including the empty event). The relations of the game example decrease the number of input events from 16 to 9. The user should always be aware of the importance of relations when compiling programs. The number of states is less controllable, as for parser generators. However, the size of practical applications is resonable most of the time, say from 10 to 100 states. One has to notice that the "internal moves" of a program do not generate states, unlike in asynchronous formalisms: the states really correspond to observable input-output states. To our belief, this is a major advantage of deterministic synchronous formalisms over non-deterministic asynchronous ones. Furthermore, it is often possible to obtain dramatic size reductions by splitting big automata into cascades of small automata. The ESTEREL V3 compiler can automatically perform such splits in some particular cases. We shall not discuss this subject here, see [8, 9] for details.

9.4. Efficiency of the compiling process

The derivative algorithm is used as such in the ESTEREL V2 system, which is written in LE_LISP [20]. It involves two rather expensive operations: the symbolic evaluation of programs on given inputs, the storing and comparison of program text (or trees). It also involves a complex dynamic memory handling that requires garbage collection in practice. To fix ideas, our standard wristwatch example (an average size non-trivial program) compiles in 60 seconds on a SUN 3 machine, within 1.5 Mbyte of memory. Bigger programs can require an order of magnitude more time and space.

The ESTEREL V3 compiler is based on a deeply optimized version of our algorithm (a similar but much simpler optimization to the original Brzozowski algorithms for regular expressions is presented in [10]).

First, ESTEREL programs are translated in a low-level intermediate code that compiles away their *control structure*, while preserving concurrency. The translation is structural, so that application of the **copymodule** directive (inter-module linking) can be done at this code level. Then, the code is symbolically executed to extract the automaton. This is done by operating an *abstract execution machine* on the code. This machine has very simple execution states (basically described by a pair of stacks) and simple operations. Derivatives are represented by sets of program pointers; they are easy to compare and inexpensive to store. Moreover, the execution machine outputs the automaton and its transitions directly as a stream (without backpatching). It operates in very little memory with a simple allocation strategy and no garbage collection. Thus the v3 compiler can be implemented in a more traditional imperative language (we chose C++ [40]).

All of this makes the v3 compiler considerably more efficient than its predecessor. On the SUN 3 workstation, the wristwatch example now compiles in less than 6 seconds, using about 100K bytes. The payoff is even larger on bigger programs, since the v3 compiler is immune to the garbage collector thrashing that occurs in v2 on larger examples. Compiling in v3 is qualitatively different, since the performance of the compiler is more limited by the size of its output than by its own time/space requirements.

10. Conclusion

We have presented in details the ESTEREL programming language, its temporal constructs based on the synchrony hypothesis, its mathematical semantics, and its currently available implementations ESTEREL V2 and ESTEREL V3 that translate concurrent synchronous programs into sequential automata. The ESTEREL V3 implementation is now developed on an industrial basis. Numerous examples have been successfully treated in different areas such as real-time process control, graphics [22], and communication protocols. We believe that the practical interest of synchronous programming compared to classical asynchronous programming is now well-established in the framework of reactive systems. We pursue our work in the following areas:

- *Language Design*: The kernel synchronous calculus developed by the second author [24] will allow us to implement temporal statements that are not yet available in ESTEREL, such as *process suspension*. The rather weak module structure of ESTEREL will be extended to support hierarchical module definitions, following the Standard ML module structure [35, 37].
- *Implementation*: the ESTEREL V3 compiler is already efficient. But we can still gain speed by improving the internal coding of objects. Several codings of the output automaton should also be available to match various time/space ratio constraints.
- *Programming environments*: we are currently building an interactive programming environment using the CENTAUR system [12]. We plan to implement advanced features such as visual source stepping of programs (this is of course harder for a concurrent language than for a sequential one, but determinism should help keep the environment simple). The *same* source stepping facilities should be available from the source or compiled code: we know how to maintain object/source correspondences at low cost.
- ESTEREL *program proving*: this is a very important area in practice, since reactive programs can control devices for which safety is critical. We mentioned various kind of available proof techniques. They must be evaluated on real examples. We must build nice interfaces between the ESTEREL compilers and the proof systems so that non-specialists can also perform proofs.
- *Large scale experiments*: obviously, the synchronous programming style is not yet completely understood. For large applications, it is clear to us that mixed synchronous/asynchronous strategies will be

needed. There should be no technical difficulty to asynchronously run communicating synchronous automata. But only experiments will show where it is reasonable to put the boundary between synchronous and asynchronous techniques.

Acknowledgements: We would like to thank J-P. Marmorat and J-P. Rigault, who had the first ideas on the design of ESTEREL and continue to participate the project, L. Cosserat who designed the first correct behavioral semantics and was co-author of the initial version of this paper [5], P. Couronné who implemented the ESTEREL V2 system with the authors, F. Boussinot who analyzed the causality problem, proposed alternative semantics, and wrote the ESTEREL V3 compiler kernel, R. Bernhard, A. Ressouche, J-B. Saint, J-M. Tanzi who wrote the rest of the ESTEREL V3 system, and J. Incerpi for her careful proof-reading.

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] H. BARENDREGT, *The Lambda-Calculus: its Syntax and Semantics*, North-Holland (1981).

[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, P. COURONNÉ, G. GONTHIER, *Programmation Synchrone des Systèmes Réactifs: le Langage ESTEREL*, Techniques et Sciences de l'Informatique vol. 6, n. 4, pp. 305-316 (1987).

[7] G. BERRY, P. COURONNÉ, G. GONTHIER, *Synchronous Programming of Reactive Systems: an Introduction to ESTEREL*, INRIA report 647 (1987).

[8] G. BERRY, F. BOUSSINOT, P. COURONNÉ, G. GONTHIER, *ESTEREL System Manuals*, Collection of Technical Reports, Ecole des Mines / INRIA, Sophia-Antipolis (1986).

[9] G. BERRY, F. BOUSSINOT, P. COURONNÉ, G. GONTHIER, *ESTEREL Programming Examples*, Collection of Technical Reports, Ecole des Mines / INRIA, Sophia-Antipolis (1986).

[10] G. BERRY, R. SETHI, *From Regular Expressions to Deterministic Automata*, Theoretical Computer Science 48, pp. 117-126 (1987).

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

[12] P. BORRAS, D. CLÉMENT, T. DESPEYROUX, J. INCERPI, G. KAHN, B. LANG, V. PASCUAL, *CENTAUR: the System*, INRIA report 777 (1987).

[13] G. BOUDOL, *Communication is an Abstraction*, Actes du Colloque C3 d'Angoulème, CNRS, and INRIA Report 636 (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] F. BOUSSINOT, *Une Sémantique du Langage ESTEREL*, INRIA Report 577 (1986).

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

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

[19] 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).

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

[21] 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).

[22] D. CLÉMENT, J. INCERPI, *Specifying the Behavior of Graphical Objects Using ESTEREL*, INRIA report 836 (1988).

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

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

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

[26] D. Harel, *Statecharts : A visual Approach to Complex Systems*, Science of Computer Programming, Vol. 8-3, pp. 231-275 (1987).

[27] D. Harel, A. Pnueli, *On the Development of Reactive Systems: Logic and Models of Concurrent Systems*, Proc. NATO Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems, NATO ASI Series F, vol.13, Springer-Verlag, pp. 477-498 (1985).

[28] D. Harel, A. Pnueli, J. Pruzan-Schmidt, R. Sherman, *On the Formal Semantics of Statecharts*, Proc. Symposium on Logic in Computer Science, pp. 54-64 (1987).

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

[30] G. Huet, *Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems*, Journal of ACM, vol 27, n. 4, pp. 797-821 (1980).

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

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

[33] G. Kahn, D. Mac Queen, *Coroutines and Networks of Parallel Processes*, Proc. IFIP 77, North-Holland, Amsterdam, pp. 993-998 (1977).

[34] G. Kahn, *Natural Semantics*, Proceedings of STACS 1987 Conference, Lecture Notes in Computer Science, Vol. 247, Springer-Verlag (1987).

[35] D. Mac Queen, *Modules for Standard ML*, Polymorphism, 2(2) (1985).

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

[37] J. Mitchell, R. Harper, *The Essence of ML*, Proc. ACM Conf. on Principles of Programming Languages (1988).

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

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

[40] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley (1986).

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

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

[43] D. Vergamini, *Vérification de Réseaux d'Automates Finis par Equivalences Observationnelles: le système AUTO*, Thèse d'Informatique, Université de Nice (1987).

Annex 1 : The Reflex Game Esterel Program

```
module REFLEX_GAME :

constant LIMIT_TIME, MEASURE_NUMBER, PAUSE_LENGTH : integer;

function RANDOM():integer;

input MS, COIN, READY, STOP;

relation MS # COIN # READY, COIN # STOP, READY # STOP;

output DISPLAY(integer),
       GO_ON, GO_OFF,
       GAME_OVER_ON, GAME_OVER_OFF,
       RED_ON, RED_OFF,
       RING_BELL;

 % overall initializations
emit DISPLAY(0);
emit GO_OFF;
emit GAME_OVER_ON;
emit RED_OFF;
 % loop over a single game
every COIN do
   % initializations
   emit DISPLAY(0);
   emit GO_OFF;
   emit GAME_OVER_OFF;
   emit RED_OFF;
   % exception handling
   trap END_GAME, ERROR in
      signal INCREMENT_AVERAGE(integer),
             AVERAGE_VALUE(integer) in
         [
            copymodule AVERAGE
         ||
            repeat MEASURE_NUMBER times
               % phase 1
               do
                  do
                     every STOP do emit RING_BELL end
                  upto READY
               watching LIMIT_TIME MS timeout exit ERROR end;
               % phases 2 and 3
               trap END_MEASURE in
                  [
                     every READY do emit RING_BELL end
                  ||
                     % phase 2
                     do
                        await RANDOM() MS
                     watching STOP timeout exit ERROR end;
                     emit GO_ON;
```

```
                    % phase 3
                    do
                       var TIME:=0:integer in
                          do
                            every MS do TIME:=TIME+1 end
                          upto STOP;
                          emit DISPLAY(TIME);
                          emit INCREMENT_AVERAGE (TIME)
                       end
                    watching LIMIT_TIME MS timeout exit ERROR end;
                    emit GO_OFF;
                    exit END_MEASURE
                 ]
              end
           end;
           % final display
           await PAUSE_LENGTH MS do
              emit DISPLAY(? AVERAGE_VALUE)
           end;
           exit END_GAME
        ]
     end
  handle ERROR do
     emit RED_ON;
     emit GO_OFF
  end;
  %end of the game
  emit GAME_OVER_ON
end.
```

Annex 2: the REFLEX_GAME automaton

1. Memory allocation

```
V0  : boolean (presence of signal MS)
V1  : boolean (presence of signal COIN)
V2  : boolean (presence of signal READY)
V3  : boolean (presence of signal STOP)
V4  : integer (value of signal DISPLAY)
V5  : integer (value of signal INCREMENT_AVERAGE)
V6  : integer (value of signal AVERAGE_VALUE)
V7  : integer (count variable)
V8  : integer (count variable)
V9  : integer (count variable)
V10 : integer (source variable TIME )
V11 : integer (count variable)
V12 : integer (count variable)
V13 : integer (source variable TOTAL )
V14 : integer (source variable NUMBER )
```

2. Actions

2.1 Test expressions for input signals

```
A1  : V0 (presence of MS)
A2  : V1 (presence of COIN)
A3  : V2 (presence of READY)
A4  : V3 (presence of STOP)
```

2.2 Output signal actions

```
A5  : output DISPLAY(V4)
A6  : output GO_ON
A7  : output GO_OFF
A8  : output GAME_OVER_ON
A9  : output GAME_OVER_OFF
A10 : output RED_ON
A11 : output RED_OFF
A12 : output RING_BELL
```

2.3 Assignment actions

```
A13 : V4  := 0
A14 : V4  := 0
A15 : V7  := MEASURE_NUMBER
A16 : V8  := LIMIT_TIME
A17 : V9  := RANDOM()
A18 : V10 := 0
A19 : V10 := V10+1
A20 : V4  := V10
A21 : V5  := V10
A22 : V11 := LIMIT_TIME
A23 : V12 := PAUSE_LENGTH
A24 : V4  := V6
A25 : V13 := 0
A26 : V14 := 0
A27 : V13 := V13+V5
A28 : V14 := V14+1
A29 : V6  := V13/V14
```

2.4 Test expressions

```
A30: V7>0
```

2.5 Decrement and test expressions

```
A31 : decr V7
A32 : decr V8
A33 : decr V9
A34 : decr V11
A35 : decr V12
```

3. The automaton

The full automaton has 6 states. We only list states 4 (READY pressed, waiting for GO_ON) and 5 (GO_ON emitted, waiting for STOP). For input tests, we recall the input signal, as in `A2 [COIN]`. For output actions, we recall the output signal, as in `A11 {RED_OFF}`.

State 4

```
if A1 [MS] then
  if A33 then
     A22; A18; A6 {GO_ON}; goto 5
   else
     goto 4
   end
end;
if A2 [COIN] then
   A14; A15;
   if A30 then
     A16; A25; A26;
     A5 {DISPLAY}; A7 {GO_OFF}; A9 {GAME_OVER_OFF};  A11 {RED_OFF}; goto 2
   else
     A23; A25; A26;
     A5 {DISPLAY}; A7 {GO_OFF}; A9 {GAME_OVER_OFF}; A11 {RED_OFF}; goto 3
   end;
end;
if A3 [READY] then A12 {RING_BELL}; goto 4 end;
if A4 [STOP] then A7 {GO_OFF}; A8 {GAME_OVER_ON}; A10 {RED_ON}; goto 1 end;
goto 4
```

State 5

```
if A1 [MS] then
   if A34 then
      A7 {GO_OFF}; A8 {GAME_OVER_ON}; A10 {RED_ON}; goto 1
   else
      if A4 [STOP] then
         A20; A21;
         if A31 then
            A23; A27; A28; A29; A5 {DISPLAY}; A7 {GO_OFF}; goto 3
         else
            A16; A27; A28; A29; A5 {DISPLAY}; A7 {GO_OFF}; goto 2
         end
      end;
      A19; goto 5
   end
end;
if A2 [COIN] then
   A14; A15;
   if A30 then
      A16; A25; A26;
      A5 {DISPLAY}; A7 {GO_OFF}; A9 {GAME_OVER_OFF}; A11 {RED_OFF}; goto 2
   else
      A23; A25; A26;
      A5 {DISPLAY}; A7 {GO_OFF}; A9 {GAME_OVER_OFF}; A11 {RED_OFF}; goto 3
   end
end;
if A3 [READY] then A12 {RING_BELL} goto 5 end;
if A4 [STOP] then
   A20; A21;
   if A31 then
      A23; A27; A28; A29; A5 {DISPLAY}; A7 {GO_OFF}; goto 3
   else
      A16; A27; A28; A29; A5 {DISPLAY}; A7 {GO_OFF}; goto 2
   end
end;
goto 5
```