A Quick Guide to Esterel
Version 5.10, release 1.1

Gérard Berry
Ecole des Mines and INRIA
Sophia-Antipolis
06560 Valbonne
berry@cma.inria.fr

April 18, 1997

This document is a part of the ESTEREL Primer [4] that can be used as a stand-alone quick guide to the language constructs, syntax, and semantics. We assume that the reader is already familiar with the foundations of ESTEREL: reactive systems, signals, events, instantaneous broadcasting and control transmission, etc. These are abundantly described in the references [4, 7, 1, 2, 9, 8]. For fine points, please refer to the Esterel Reference Manual [5] and Constructive Semantics definition [3].

WARNING: The language covered is Version 5.10 to come. The following features are new to 5.10 and are not supported in the current versions 5.0x:

- The `float` primitive type, Section 3.
- The explicit definition of constants, Section 3.
- The use of `+`, `*`, `and`, `or` in combined signal declarations, Section 4.
- The `pause` statement, Section 7.1, which must be written "`await tick`".
- The `positive repeat` statement, Section 7.4.
- The renamings of functions by operators in `run` submodule inclusion, Section 7.15.

Contents

1 Lexical Aspects

Lexical aspects are classical:

- Identifiers are sequences of letters, digits, and the underline character '`_`' starting with a letter.
- Integer and floating-point numerical constants are as in C, e.g, `123`, `12.3`, `.123E2`, or `1.23E1`.
- Strings are written between double quotes, e.g., `"a string"`, with doubled double quotes as in `"a "" double quote"`.
- Keywords are reserved (the list is given in the Esterel Reference Manual [5]). Many constructs are bracketed, like "`present` ... `end present`". For such constructs, repeating the initial keyword is optional; one can also write "`present` ... `end`".
- Simple comments start with '`%`' and end at end-of-line. Multiple-line comments start with '`%{`' and end with '`}%`'.

2 Modules

An ESTEREL program is defined by its main module, which can use submodules called by the `run` statement. A module has a name, an interface declaration part, and a body, which is an executable statement:

```
module M :
 interface declaration
 statement
end module
```

The ESTEREL v5 compiler translates a module into a conventional circuit or program written in a *host language* that is chosen by the user.

The interface declaration contains two kinds of objects:

- Data objects, which are declared abstractly in ESTEREL. Their actual value is supposed to be given in the host language and linked to the ESTEREL compiled code in a way that depends on the compiler and target language.

- Signals and sensors, which are the primary objects the program deals with.

The data and signal declarations can be mixed in an arbitrary way, provided that any item is declared before being used. The scope of interface objects is the whole module. Here are complete examples of the module interface, the components of which will be explained below:

```
module WATCH :

input UL, UR, LL, LR;        % the four watch buttons
relation UL # UR # LL # LR; % they are incompatible

input S, HS;                 % second and 1/100 second
relation S => HS;            % no S without an HS

type Time;
constant Noon : Time;
function CompareTime (Time, Time) : boolean;
procedure IncrementTime (Time) (integer);
output CurrentTime := Noon : Time;

type Beep;
constant WatchBeep : Beep, AlarmBeep : Beep;
function CombineBeeps (Beep, Beep) : Beep;
output Beeper : combine Beep with CombineBeeps;

module ROBOT:

type Coord, Rectangle;

function MakeRectangle (Coord, Coord) : Rectangle;
function InRectangle (Coord, Rectangle) : boolean;
procedure TranslateAndRotate (Rectangle) (Coord, integer);

task MoveRobotInsideRectangle (Coord) (Rectangle);
return RobotInRectangle;

module MISC :
constant WordLength = 16 : integer;
sensor Temperature : float;
output YesVotes := 0 : combine integer with +;
```

3 Data

Data objects are divided between primitive and user-defined objects. Since data handling is not a primary concern in control-dominated reactive programming, we kept the data definition facilities minimal, heavily relying on the host language capabilities. All data objects are global to the program. Each data object used within a module must be declared in that module. For a multi-module program, an data object declared in several submodules must be identically declared in all of them, see Section 7.15.

3.1 Types and Operators

There are only four primitive types in ESTEREL: `boolean`, `integer`, `float`, and `string`. The Boolean constants are `true` and `false`. The numerical and string constants were described in Section 1.

The operations are the usual ones. Equality is written `=` and difference is written `<>` for all types. The `boolean` type is equipped with `and`, `or`, and `not`, and the operations `+`, `-`, `*`, `/`, `<`, `<=`, `>`, and `>=` are available for `integer` and `float`. There is no implicit type conversion. In particular, the user must call explicitly declared external functions to convert integers to floats and conversely.

The user can define his own types by declaring their names. For ESTEREL, a user type is a completely abstract object. Its actual definition will be given only in the host language. Here are the type declarations of the above module examples:

```
type Time;
type Beep;
type Coord, Rectangle;
```

3.2 Constants

Constants of any type can be declared as follows:

```
constant Noon : Time;
constant WatchBeep : Beep, AlarmBeep : Beep;
constant WordLength = 16;
```

There are two ways to declare a constant. In the implicit way, only the name and the type are given, see `Noon` above. The value is defined in the host language. In the explicit way, the name, the type, and the value are declared, see `WordLength` above. This is possible only for constants of predefined types.

3.3 Functions

Functions take a list of objects of arbitrary types and return a single object of arbitrary type:

```
function CompareTime (Time, Time) : boolean;
function CombineBeeps (Beep, Beep) : Beep;
function MakeRectangle (Coord, Coord) : Rectangle;
function InRectangle (Coord, Rectangle) : boolean;
```

Functions are defined in the host language. They are called in data expressions, and they must be side-effect free.

3.4 Procedures

Procedures have two lists of arguments of arbitrary types:

```
procedure IncrementTime (Time) (integer);
procedure TranslateAndRotate (Rectangle) (Coord, integer);
```

The first list is the list of reference arguments that are passed by reference and possibly modified by the call. The second list is that of value arguments that are passed by value and not modified. For example, in `TranslateAndRotate`, the rectangle is passed by reference and modified when translated and rotated, while the translation and rotation arguments are passed by value. Each of the lists can be empty. Procedures are defined in the host language. They are called by the `call` statement, which is instantaneous.

3.5 Tasks

Tasks are declared exactly as procedures:

```
task MoveRobotInsideRectangle (Coord) (Rectangle);
```

Their actual code is given in the host language. The difference between tasks and procedures is that task execution is assumed to be non-instantaneous, unlike procedure calls. Tasks are executed by the `exec` statement and coupled with `return` signal as described in Section 7.16. Tasks are supposed to run concurrently with the ESTEREL program. The way this is implemented depends on the compiler, on the host language, and on the run-time system.

4 Signals and Sensors

Signals and sensors are the logical objects received and emitted by the program or used for internal bookkeeping. They are instantaneously broadcast throughout the program, which implies that all statements see each of them in a consistent way. Pure signals have a presence status, *present* or *absent*. In addition to their status which is as for pure signals, valued signals carry a value of any type. Sensors have a value but no status.

There is one predefined signal, the special pure signal `tick` that represents the activation clock of the reactive program. Its status is *present* at each instant.

Signals can be interface signals declared in the module interface or local signals declared by the `signal` local signal declaration statement, see Section 4.5.

4.1 Interface Signal Declarations

Interface signals are either `input`, `output`, `inputoutput`, or `return`. The `return` signals are used to signal termination of external tasks, see Section 7.16. Here are the interface signals declarations of the above modules:

```
input UL, UR, LL, LR;       % the four watch buttons
input S, HS;                % second and 1/100 second
output CurrentTime := Noon : Time;
output Beeper : combine Beep with CombineBeeps;

return RobotInRectangle;

output YesVotes := 0 : combine integer with +;
```

Here, `UL`, `UR`, `LL`, `LR`, `S`, and `HS` are pure input signals. The `CurrentTime` output signal is a valued signal of abstract time `Time`, with value initialized to `Noon`. Similarly, the `YesVotes` signal is integer-valued with initial value 0. If no initial value is given, as for `Beeper`, the value is undefined until the first time the signal is received from the environment or emitted by the program itself.

The return signal `RobotInRectangle` is a special input signal used for signaling external task completion, see Section 3.5. A return signal can be valued just as a standard input signal.

A local signal declaration is declared using the `signal` keyword:

```
signal AuxSig1, AuxSig2 : integer in
    statement
end signal
```

The local signal declaration construct defines a statement that can be put wherever a statement can.

Notice that types must be declared separately for each signal in a signal declaration list. Here, `AuxSig1` is a pure signal.

4.2 Single and Combined Valued Signals

The above `CurrentTime` signal is single: it cannot be emitted twice in the same instance and it cannot be emitted if it is received from the environment. This restriction holds for any valued signal not declared using the `combine` keyword. The signals declared with that keyword are called combined signals.

In the above declarations, `Beeper` and `YesVotes` are combined. For them, the values simultaneously emitted by several emitters or received from the environment are gathered and combined using the specified binary function or operator that must be commutative and associative. For `Beeper`, the `Beep` type can represent a set of sound frequencies and combining several sounds by `CombineBeeps` can be taking the union of their frequencies. In this way, one can hear the timekeeper, alarm, and stopwatch beep together. For `YesVotes`, using addition as a combination function makes it easy to count simultaneous yes votes if each participant broadcasts the number of voices he or she represents.

For type `boolean`, the combination function can be `and` or `or`. For type `integer` and `float`, the combination function can be `+` or `*`. Any other combination function must be user-defined and declared prior to the signal declaration. The corresponding host language combination function is assumed to be commutative and associative, which obviously cannot be checked by ESTEREL.

4.3 Sensors

Sensors are valued input signals without presence information. A sensor is declared by giving its name and type:

```
sensor Temperature : integer;
```

Sensors differ from signals in the way they are interfaced with the environment. The value of a sensor is read by the program whenever needed. Therefore, the notion of an initial value is meaningless for sensors.

4.4 Input Relations

Input relations declare some Boolean condition about input or return signals that are assumed to be guaranteed by the environment. In the above `WATCH` example, the relations are:

```
relation UL # UR # LL # LR;
relation S => HS;
```

The first relation is called an incompatibility (or exclusion) relation. It asserts that the four input buttons are incompatible (or exclusive), i.e., that no two of them can be simultaneously present. The second relation is called an implication relation. It asserts that `S` can be present only if `HS` is, i.e., that a second is always synchronous with a 1/100 second.

Relations are useful to avoid specifying irrelevant behaviors. For example, in the `WATCH` example, the exclusion relation asserts that the user cannot simultaneously request to change to set-watch mode and to stopwatch mode; in practice, buttons are serialized by the low-level event handler. Relations are also useful to optimize automaton code generation, for circuit code optimization, and to speed-up program verification.

4.5 Local Signal Declaration

A local signal declaration is performed by the following statement construct:

```
signal Alarm,
       Distance : integer,
       Beep := OneBeep : combine Beep with CombineBeeps
in
   p
end signal
```

where p is any statement. The individual declarations are the same as for interface signals, see Section 4. The scope of a local signal declaration is the body p. Scoping is lexical: any re-declaration of a signal hides the outer declaration.

Signals are subject to reincarnation and may provoke causality problems, see Section 8 and Section 9.

5 Variables

Variables are assignable objects declared by the local variable declaration statement, which has the form

```
var X : float,
    Count := ? Distance : integer,
    Deadline : Time
in
   p
end var
```

where p is any statement. A variable declaration declares the names of the variables, their types, and possibly their initial values. The scope of a variable declaration is the body p. Scoping is lexical: any re-declaration of a variable hides the outer declaration. The type must be declared individually for each variable. The declaration

```
var X, Y : integer in
```

is incorrect since `X` has no type. One must write

```
var X : integer, Y : integer in
```

A variable has a name and a type, and it is modified by assignments and procedure calls, see Section 7.2. Unlike a signal, a variable can take several successive values at the same instant. For example, in the statement

```
X := 0;
emit S1(X);
X:= X+1;
emit S2(X)
```

the signals `S1` and `S2` are emitted simultaneously with respective values 0 and 1, the variable `X` taking these values in succession in the instant. This poses absolutely no problem in the constructive semantics of Esterel presented in Section 9, provided of course that variables cannot be shared in read-write mode between threads. More precisely, if a variable is written in a thread, then it can be neither read nor written in any concurrent thread.

6 Expressions

There are three kinds of expressions in Esterel: data expressions, signal expressions, and delay expressions.

6.1 Data Expressions

Data expressions are built as usual by combining basic objects using operators and function calls. Their evaluation is instantaneous. All expressions must type-check.

Constants and variables appear under their names. The current value of a valued signal or sensor `S` is written `?S`. Traps can carry values just as signals, see Section 7.13. However, the name space of traps is distinct from that of signals, and we must use a different symbol to access trap values. The current value of a valued trap `T` is written `??T`. Accessing a yet undefined signal or trap value is an error. Function calls are written as usual. Here are some expressions:

```
X * WordLength
FloatToInteger(?Temperature) * (??ExitCode + 5)
```

6.2 Signal Expressions

Signal expressions are Boolean expressions over signal statuses. They are used in instantaneous `present` tests or in delay expressions. Signal expressions are obtained by combining signal names or the `tick` predefined signal using the `not`, `and`, and `or` operators with the usual binding conventions: `not` binds tighter than `and` and `and` binds tighter than `or`. Here are examples:

```
Meter and not Second
Bit1 and Bit2 and not (Bit3 or Bit4)
not tick
```

The interpretation of signal expression is obvious, but some constructive causality aspects have to be well-understood, see Section 9 for details. Notice that the expression `tick` is always true, so the expression `not tick` is always false.

6.3 Delay Expressions

Delay expressions are used in temporal statements such as `await` or `abort`. There are three forms of delay expressions: standard delays, immediate delays, and count delays. A delay starts when the temporal statement that bears it starts, and it elapses at some later instant, possibly at the same instant for immediate delays.

Standard delays are defined by a signal expression. For instance, the delay

```
Meter and not Second
```

elapses at the *next* instant where a meter occurs without a simultaneous second. Standard delays never elapse instantaneously.

Immediate delays are defined by the `immediate` keyword followed by a signal expression, which must appear within brackets '[]' unless it is a single identifier. For instance, the immediate delay

```
immediate [Meter and not Second]
```

elapses instantaneously if a meter and no second are present when the delay is initiated, and it behaves as a standard delay otherwise. Notice that there is only one layer of brackets '[]' and that standard parentheses are used inside delay expressions, as for `Meter and (not Second)`.

Count delays are defined by an integer count expression followed by a signal expression.

The signal expression must be bracketed using square brackets '[]' if it is not reduced to a single signal. Here are examples:

```
3 Second
5*X Meter
3 [Second and not Meter]
```

The expression is evaluated only once when the delay is initiated. If the expression's value is 0 or less, it is set to 1. Therefore, a count delay never elapses instantaneously. This is fundamental for various kinds of static analysis including constructiveness analysis, see Section 9 and Section 7.6.

Notice the restrictions on delays: there is no immediate count delay, and counts cannot be intertwined with Boolean signal operators. This is a deliberate choice. We think that expressions such as

```
immediate [ 3 [n Seconds or p [Meter and not Second]]]
```

are too difficult to understand.

7 Statements

We describe here all statements except local signal declaration described in Section 4.5 and local variable declaration described in Section 5. All constructs but sequencing '`;`' and concurrency '`||`' use bracketed keywords: `abort`—`end abort`, etc. Repetition of the initial keyword is optional, so `abort`—`end` is also correct. To resolve the remaining syntactic ambiguities, any statement can be explicitly bracketed using square brackets '[]'. ESTEREL is fully orthogonal: statements can be freely mixed in an arbitrary way. One can sequence parallel statements or put sequences in parallel, one can subject any statement to an abortion, etc.

In the sequel, we do not add indentation for parallel statements. Therefore,

```
signal S in
   p
||
   q
end signal
```

is the same as the more indented form

```
signal S in
      p
   ||
      q
end signal
```

7.1 Basic Control Statements

There are three basic pure control statements:

```
nothing
pause
halt
```

The `nothing` statement terminates instantaneously when started. The `pause` statement pauses for one instant. More precisely, it pauses when started, and it terminates at next instant (`pause` can also be written "`await tick`", see Section 7.9). The `halt` statements pauses forever and never terminates.

7.2 Assignment and Procedure Call

See Section 5 for the definition of a variable. Assignments have the form

```
X := e
```

where `X` is a variable and e is a data expression. The variable and expression must have the same type. Assignments are instantaneous.

Procedure calls have the form

```
call P (X, Y) (e1, e2)
```

where `X` and `Y` are variables and the e_i are expressions. The types must match those of the declaration. The variables are modified by the call. The call is instantaneous.

7.3 Signal Emission

Instantaneous signal emission is realized by the `emit` statement, which has one of two forms:

```
emit S
emit S(e)
```

For a pure signal `S`, the `emit` statement simply emits `S` and terminates instantaneously. For a valued signal, the `emit S(`e`)` statement evaluates the data expression e, emits `S` with that value, and terminates instantaneously.

For a single signal, if an `emit` signal is executed, it must be the only one in the instant; for an input single signal, no `emit` statement can be executed if the signal is received in the input event. For a combined signal, the emitted value is combined with those emitted by other `emit` statements executed in the instant using the combination function. For an input combined signal that is received from the environment and locally emitted at the same time, the received and emitted valued are combined.

Continuous emission of a signal is realized by the `sustain` statement:

```
sustain S
sustain S(e)
```

When started, the `sustain` remains active forever and it emits `S` at each instant. For a valued signal, the data expression e is re-evaluated at each instant. The "`sustain S`" statement abbreviates "`loop emit S each tick`", see Section 7.11.

7.4 Sequencing

Sequencing is done using the '`;`' sequence operator. In

p ; q

the first statement p is instantaneously started when the sequence is started, and it is executed up to completion or trap exit. If p terminates, q is immediately started and the sequence behaves as q from then on. If p exits enclosing traps, the exits are immediately propagated and q is never started, see Section 7.13. For example, "`exit T; emit S`" does not emit `S`.

7.5 Looping

A simple loop has the following form:

```
loop
   p
end loop
```

The body p is re-started afresh upon termination, this forever. If p exits enclosing traps, the exits are propagated instantaneously. This is the only way to exit a loop from inside. Of course, a loop can be killed by an external preemption statement, see Section 7.10 and Section 7.13.

The body of a loop is not allowed to be able to terminate instantaneously when started. This condition is static: there must be no potential direct path from starting to termination in p, even if that path cannot be taken dynamically. For instance, the following loop is rejected:

```
loop
   present I else
      ...
   end present;
   present J else
      ...
   end present
end loop
```

Even if the '`...`' statements have delays, there is a potential instantaneous path in the body, corresponding to the case where `I` and `J` are both present. If `I` and `J` are inputs declared incompatible by the relation `I # J`, then the instantaneous path is a false one since it cannot be taken in any valid input configuration. The program is rejected nevertheless. One must add an extra clause involving a delay:

```
loop
   present I else
      ...
   end present;
   present J then
      pause % unreachable
   else
      ...
   end present
end loop
```

7.6 Repeat Loops

A `repeat` loop executes its body for a finite number of times. The body is not allowed to terminate instantaneously. The simplest form is

```
repeat e times
   p
end repeat
```

The expression e must be of type `integer`. It is evaluated only once at starting time. The body is not executed at all if e evaluate to 0 or to a negative number. Therefore, the simple `repeat` statement is considered as possibly instantaneous and it cannot be put in a loop if not preceded or followed by a delay, this even if its own body is non-instantaneous. Therefore, the following statement is rejected as being a potentially instantaneous loop:

```
repeat 5 times
   repeat 3 times
      ...
   end repeat
end repeat
```

ESTEREL compilers are not required to perform static analysis and discover that 3 is never null, because one can replace 3 by user-defined constants or complex expressions. To solve this problem, we ask the user to assert that the body will be executed at least once by adding the `positive` keyword:

```
repeat 5 times
   positive repeat 3 times
      ...
   end repeat
end repeat
```

In the `positive repeat` statement, the test for repetition is performed only after the first execution of the body. The body is not allowed to be able to terminate instantaneously, and the whole `positive repeat` statement inherits the same property.

7.7 The present Signal Test

The `present` statement branches according to the instantaneous values of signal expressions. The simplest form checks for one signal expression and performs binary branching. Each of the `then` and `else` branches can be omitted, but at least one of them must be there. An omitted branch is implicitly `nothing`:

```
present S then p else q end present
present [Second and Meter] then p end present
present Meter else q end
```

The `case` form tests several signal expressions in sequence:

```
present
   case Meter do
      Distance := Distance+1;
      emit Distance(Distance)
   case Second do
      emit Speed(Distance)
end present
```

The tests are taken in order, and the first true expression starts immediately its `do` clause. If the `do` clause is omitted, the `present` statement simply terminates. If none of the expressions is true, the `present` statement terminates. One can add an `else` statement for that case:

```
present
   case [Bit0 and Bit1] do
      emit Load
   case [Bit0 and not Bit1] do
      emit Store
   case [not Bit0 and Bit1] % no-op
   else
      exit WrongOpCode
end present
```

7.8 The if Data Test

The `if` statement is used to test Boolean data expressions. In the basic binary form, either the `then` or the `else` clause can be omitted, as for the `present` statement:

```
if X>=0 then p else q end if
if X=Y and Y<>?Z then p end
if ?Flag else q end
```

Multiple cases can be checked in sequence using the `elsif` keyword, the `case` keyword being reserved for signal expressions:

```
if X > 5 then
   p
elsif X > 3 then
   q
else
   r
end if
```

The conditions are evaluated in sequence. The first true condition triggers the corresponding statement. If no condition is true, then the `if` statement executes the `else` statement if there is one and terminates otherwise.

7.9 The await Statement

The `await` statement is the simplest temporal statement. In its basic form, it simply waits for a delay:

```
await Second
await immediate Second
await immediate [Second and Meter]
await 2 Second
await 2 tick
await 2 [Second and not Meter]
```

The delay is started when the `await` statement is started. The statement pauses until the delay elapses and terminates at that instant. An immediate `await` statement terminates instantaneously if the signal expression is true at starting instant. Be careful: the sequence

```
await immediate Meter;
await immediate Meter
```

terminates instantaneously if `Meter` is present at starting instant.

A `do` clause can be used to start another statement when the delay elapses:

```
await 2 Second do
   emit Beep
end await
```

This is simply an abbreviation for "`await 2 Second; emit Beep`". As for `present`, one can introduce a case list:

```
await
   case 2 Second do p
   case immediate Meter
   case Button do q
end await
```

The above statement immediately terminates if `Meter` occurs at start time. Otherwise, the first delay to elapse determines the subsequent behavior: p is started if `2 Second` elapses first, the `await` statement simply terminates if `Meter` occurs first, and q is started if `Button` occurs first. If several delays elapse at the same time, the first one in the list takes priority. For example, if `Meter` and `Button` occur simultaneously, then the `await` statement terminates and q is not started.

7.10 The abort Statements

An abortion statement kill its body when a delay elapses. For strong abortion, performed by `abort`, the body does not receive the control at abortion time. For weak abortion, performed by `weak abort`, the body receives the control for a last time at abortion time. The syntax is as follows:

```
abort p when 3 Meter
weak abort p when 3 Meter
```

For both constructs, the body p is run until termination or until the delay elapses. If p terminates before the delay elapses, so do the `abort` and `weak abort` statements. Otherwise, p is preempted when the delay elapses;

at that instant, p is not executed with strong abortion, and it is executed for a last time with weak abortion (p has rights to its "last wills").

If the delay is immediate and elapses immediately at starting time, the body is not executed at all with strong abortion, and it is executed for one instant with weak abortion For example, in

```
abort
   sustain O
when immediate I
```

the `abort` statement terminates immediately without emitting `O` if `I` is present at starting time. If `abort` is replaced by `weak abort`, the whole statement also terminates instantaneously but `O` is emitted once.

As for `await`, one can add a `do` clause to execute a statement q in case of delay elapsing:

```
abort % or weak abort
   p
when 3 Meter do
   q
end abort
```

With both weak and strong abortion, q is executed if and only if p did not terminate strictly before delay elapsing. At abortion time, with strong abortion, p is not executed and q is immediately started. With weak abortion, the first instant of q is done in sequence after the last instant of p.

As for `await`, one can introduce an ordered list of abortion cases:

```
abort % or weak abort
   p
when
   case Alarm do r
   case 3 Second do q
   case immediate Meter
end abort
```

Here, p is immediately aborted if there is a `Meter` at starting time. Otherwise, it is run for at least one instant. The elapsing of any of the three delays aborts p. If there is a `do` clause for the delay, that statement is immediately started; otherwise, the `abort` statement simply terminates. If more than one of the delays elapses at abortion time, then the first one in the list takes priority as for the `await` statement.

Nesting `abort` statements also builds priorities. In the statement

```
abort
   abort
      p
   when I do
      q
   end abort
when J
```

the signal J takes priority over I if they occur simultaneously, and q is not started in that case. This is no special rule, but just a consequence of the strong abortion semantics of `abort`.

Finally, notice that "`await S`" can be defined as "`abort halt when S`".

7.11 Temporal Loops

Temporal loops are loops over strong abortion statements. The first form is

```
loop
   p
each d
```

where d is a non-immediate delay. At starting time, the body p is started right away, and it is restarted afresh whenever the delay d elapses. If p terminates before d elapses, then one waits for the elapsing of d to restart p. The "`loop each`" statement is simply an abbreviation for

```
loop
   abort
      p; halt
   when d
end loop
```

The delay cannot be immediate, otherwise the loop body would be instantaneous.

The second temporal loop has the form

```
every d do
   p
end
```

The difference is that d is initially waited for before starting the body p. The delay d can be immediate. In that case, at starting instant, p starts immediately if the delay elapses immediately. The statement

```
every 3 Second do
   p
end every
```

abbreviates

```
await 3 Second;
loop
   p
each 3 Second
```

The statement

```
every immediate Centimeter do
   p
end
```

abbreviates

```
await immediate Centimeter;
loop
   p
each Centimeter
```

All temporal loops are infinite. The only way to terminate them is by exiting a trap, see Section 7.13 or by the elapsing of an enclosing abortion delay.

7.12 The suspend Statement

Abortion violently preempts a statement and kills it, in the same way as ^C kills a process in Unix. Suspension has a milder action, like ^Z in Unix. The basic syntax is

```
suspend
   p
when s
```

where s is any signal expression. When the `suspend` statement starts, p is immediately started. Then, at each instant, the following occurs:

- If the signal expression s is true, then p remains in its current state and the `suspend` statement pauses for the instant.

- If the signal expression s is false, then p is executed for the instant. If p terminates or exits a trap, so does the `suspend` statement. If p pauses, so does the `suspend` statement, and suspension is re-examined at next instant.

Here is an example:

```
suspend
   abort
      sustain O
   when J
when I
```

emits `O` at first instant and at all subsequent instants where `I` is absent, until the first instant where `I` is absent and `J` present. Then the `suspend` statement terminates and `O` is not emitted.

The default `suspend` statement is delayed, in the sense that the signal expression is not tested for at first instant. The immediate form performs that test:

```
suspend
   p
when immediate s
```

Here p is not started at first instant if s is true. The immediate form can be rewritten as follows:

```
await immediate [ not s ];
suspend
   p
when s
```

7.13 Traps

A trap defines an exit point for its body. The basic syntax is

```
trap T in
   p
end trap
```

The body *p* is immediately started when the `trap` statement starts. Its execution continues up to termination or trap exit, which is provoked by executing the "`exit T`" statement. If the body terminates, so does the `trap` statement. If the body exits the trap `T`, then the `trap` statement immediately terminates, weakly aborting *p*.

The `weak abort` statement can be defined using traps. The construct "`weak abort` *p* `when S`" is an abbreviation for

```
trap T in
   p;
   exit T
||
   await S;
   exit T
end trap
```

7.13.1 Nested Traps

When traps are nested, the outer one takes priority. Consider for example

```
trap U in
   trap T in
      p
   end trap;
   q
end trap;
r
```

If *p* exits `T`, then *q* is immediately started. If *p* exits `U`, then *r* is immediately started. If *p* exits simultaneously `T` and `U`, for example by executing "`exit T || exit U`", then `U` takes priority and only *r* is executed. From the point of view of the "`trap T`" statement, `T` is discarded and `U` is propagated.

7.13.2 Trap Handlers

A handler can be used to handle a trap exit, with the following syntax:

```
trap T in
   p
handle T do
   q
end trap
```

If *p* terminates, so does the trap statement. If *p* exits `T`, then *p* is weakly aborted and *q* is immediately started in sequence.

7.13.3 Concurrent Traps

Several traps can be declared using a single `trap` keyword. In this case, the traps are called concurrent traps. Concurrent traps are at the same priority level, and any of them can have a handler. If several traps are simultaneously exited, then the corresponding handlers are executed in parallel:

```
trap T, U, V
   p
handle T do
   q
handle U do
   r
end trap
```

Here, *q* and *r* are executed in parallel if *p* exits `T` and `U` simultaneously. Since they are concurrent, the *q* and *r* handlers cannot share variables. The `trap` statement simply terminates if *p* exits `V` that has no handler.

Here is the translation of "`weak abort` *p* `when S do` *q* `end`" using concurrent traps:

```
trap Terminate, WeakAbort in
   p;
   exit Terminate
||
   await S;
   exit WeakAbort
handle WeakAbort do
   q
end trap
```

7.13.4 Valued Traps

Traps can be valued exactly as signals. Value initialization and combined traps are allowed. This is useful to pass a value to the handler. The value is obtained as the result of the expression '`??S`', which is allowed only in the handler:

```
trap Alarm : combine integer with + in
   ... exit Alarm(3) ...
   ... exit Alarm(5) ...
handle Alarm do
   emit Report(??Alarm)
end trap
```

Of course, concurrent traps can be valued:

```
trap T, U := 0 : integer, V : combine integer with + in
   p
handle T do
   q
handle U or V do
   emit O(??U + ??V)
end trap
```

Beware of uninitialized trap values.

7.14 The Parallel Statement

The parallel operator puts statements in synchronous parallel. The signals emitted by the branches or by the rest of the program are instantaneously broadcast to all branches at each instant.

A parallel can be binary, as in p `||` q, ternary, as in p `||` q `||` r, or of any arity. Syntactically, the sequencing operator '`;`' binds tighter than the parallel operator '`||`'. Therefore, p`;` q `||` r means `[`p`;` q`]` `||` r, which is different from p`;` `[`q `||` r`]` where the brackets are mandatory.

A parallel statement forks its incoming thread, starting instantaneously all its branches when it starts. The parallel terminates when all its branches have terminated, waiting for the last one if some branches terminate earlier. The parallel propagates a trap `T` as soon as one of its branches exits `T`, weakly aborting all its branches at that time. See Section 7.13 for the case where several traps are simultaneously exited.

Variables can only be shared among parallel branches if they are read-only. If a branch can write a variable `X`, then no other branch can read or write `X`. Signals are the only truly shared objects.

7.15 The run Module Instantiation Statement

A module can be instantiated within another module using the `run` executable statement. In the simplest form, one simply writes

```
run SPEED
```

This amounts to syntactically replace the `run` statement by the body of the `SPEED` module. Recursive or mutually recursive submodule instantiation is forbidden.

The data declarations of the instantiated submodule are exported to the parent module. If some data objects were already declared in the parent, the parent and child declarations must be the same, which means that all data objects are global.

The signal interface declarations of the instantiated module are simply discarded, as well as the relation declarations. This means that the interface signals of the instantiated submodule must exist in the parent module with the same type. Notice that a signal declared as input in the submodule is seen as global after instantiation. For instance, in

```
module M :
input I;
emit I
end module

module N :
output I;
run M
end module
```

the signal I is effectively emitted by `N` although it was declared as an input in `M`. This is an anomaly that should be corrected some day.

Any interface object can be renamed at module instantiation time using the following renaming syntax:

```
run GENERIC_SPEED [ type integer / T;
                    constant 0 / Initial,
                             1 / Increment;
                    function + / Add;
                    signal CarSpeed / Speed ]
```

A renaming `X / Y` is read "`X` renames `Y`". The renaming object `X` can be either an explicit constant or operator or an identifier. If it is an identifier, it must be declared in the parent module. The renamed object `Y` must be an identifier belonging to the data or signal interface of the instantiated module. The kinds and types must match.

Full renaming makes it possible to build generic modules. Partial renaming is also possible. In that case, a submodule interface object that is not renamed is captured by the parent object of the same name (and kind: a type is captured by a type, a signal by a signal, etc.).

The included module itself can be renamed:

```
      run CarSpeed / SPEED [...]
   ||
      run BicycleSpeed / SPEED [...]
```

This is useful for identifying submodule occurrences in symbolic debuggers.

7.16 The exec Task Execution Statement

External procedure calls performed using the `call` statement are supposed to be instantaneous. This does not fit with many practical applications where procedure computing times cannot be neglected. The `task-exec` mechanism we now describe makes it possible to control execution of external actions or *tasks* that take time.

Roughly speaking, tasks behave as procedures that are executed asynchronously with the ESTEREL program. At ESTEREL abstraction level, we take a logical view of tasks. We care about controlling them, and we do not care about how they are actually executed in the environment concurrently with the ESTEREL program. The only thing we are interested in is when external tasks start, when they terminate in ESTEREL sense, and when they should be suspended or aborted by other ESTEREL statements.

Tasks are not limited to be computationally intensive. They can also be of a more physical nature. For instance, in Robotics, a task may be "grasp this object".

Tasks are declared in the data interface part of a module, see Section 3.

7.16.1 The exec Statement and the Return Signals

The statement that executes a task is the `exec` statement. It has the form

`exec TASK (`*reference-params*`) (`*value-params*`) return R`

where `R` is called the *return signal*. A return signal is a special input signal declared using the `return` keyword instead of the `input` keyword in the module signal interface:

```
return R1;
return R2 : integer;
return R3 : combine FOO with F;
```

Notice that a return signal can have a value as any input signal. Return signals can also appear in exclusion or implication relations together with input signals, see Section 4.4. Like any other input signal, a return signal can be tested for presence or awaited concurrently with task execution. The use of this feature will be explained in Section 7.16.6. Unlike a standard input, a return signal cannot be internally emitted by the program.

7.16.2 External Task Execution

When an "`exec T return R`" statement starts, it signals to its environment that a fresh instance of the task `T` should start with parameters passed by reference and value just as for procedures. The signaling is instantaneous. The ESTEREL program does not wait for the task and continues reacting autonomously. At some instant in the strict future of the starting instant, the environment signals back task completion to the ESTEREL program by sending the return signal `R`. Within the ESTEREL program, receiving `R` provokes instantaneous update of reference arguments according to the values returned by the task and instantaneous termination of the `exec` statement.

During its execution, an `exec` statement can be suspended or aborted. This is signaled to the external task by sending appropriate suspension and abortion signals.

The task launching and signaling implementation mechanism entirely depends on the compiler and run-time system. The only implementation constraint is to respect the ESTEREL logical view.

7.16.3 Uniqueness of Return Signals

One may have several `exec` statements for a given task `T`; therefore, one may also have different concurrent instances of the same task in the environment. The return signal is used to tell the ESTEREL program which instance has terminated. For this to be possible, return signals must uniquely identify `exec` statements. Hence, we impose the following restriction:

> *No two* `exec` *statements in a program can have the same return signal.*

This condition must be verified after submodule expansion. Uniqueness of return signals may call for explicit renaming at submodule instantiation time:

```
module OneTask :
task TASK (integer) (integer);
return R;
var X := 0 : integer in
   exec TASK(X)(1) return R
end var
end module
```

```
module TwoTasks :
return R1, R2;
   run Task1 = OneTask [signal R1 / R]
||
   run Task2 = OneTask [signal R2 / R]
end module
```

7.16.4 Abortion of exec Statements

As any other ESTEREL statement, an `exec` statement is subject to abortion by `abort`, `weak abort`, or `trap` statements and to suspension by `suspend` statements. The simplest case of abortion is the weak one. Consider the example:

```
weak abort
   exec TASK (X) (1) return R;
when I
```

At first instant, the task is started. Then, the behavior is as follows:

- If `R` occurs before `I` or if `R` and `I` occur simultaneously, then `X` is updated and the whole `weak abort` statement terminates.
- If `I` occurs before `R`, then execution of `TASK` is aborted and the external task is aborted. There is no update of `X`.

Strong abortion is a little bit more delicate. Consider the example:

```
abort
   exec TASK (X) (1) return R;
when I
```

After starting the task at first instant, the behavior is as follows:

- If `R` occurs before `I`, then `X` is updated and the whole `abort` statement terminates.
- If `I` occurs before `R`, then execution of `TASK` is aborted and the external task is aborted. There is no update of `X`.
- If `I` and `R` occur simultaneously, then the `abort` statement terminates. Although the task did terminate, `X` is not updated since the body of the `abort` statement does not receive control. No abort signal is sent to the task either since it is terminated.

Notice the subtle difference between weak abortion by `weak abort` or `exit` and strong abortion by `abort` in the case where `R` and `I` are simultaneous: with strong abortion, update of reference variables is not performed, while it is performed with weak abortion.

7.16.5 Suspension of exec Statements

Consider a program fragment of the form

```
suspend
   exec TASK (X) () return R
when S
```

When `S` occurs after the starting instant, the `exec` statement is suspended. This is signaled to the environment by sending an implementation-dependent suspension signal. The signal is sent at every instant where the `exec` statement is suspended.

Termination of the `exec` statement can occur only when that statement is active. Assume that `R` and `S` occur simultaneously. Then, `R` does not provoke termination of the `exec` statement and its occurrence is lost. It is the environment's responsibility to sustain `R` until the `exec` statement is not suspended any more, which is easy using the abort and suspend signaling mechanism.

The environment can also transform the suspension information available at each tick into a suspend–resume information usually more appropriate for operating systems.

7.16.6 Testing for the ReturnSignal

When an `exec` statement is strongly aborted, one may need to know if the external task did terminate in the instant. This is easy using a `present` test on the return signal:

```
abort
   exec TASK (X) (1) return R
when I do
   present R then ... else ... end present
end abort
```

The same can be done for suspension

```
   suspend
      exec TASK (X) () return R
   when S
||
   await R do ... end await
```

7.16.7 Multiple exec

The multiple `exec` statement makes it possible to control several tasks simultaneously. It resembles the "`await...case`" statement:

```
exec
   case T1 (...) (...) return R1 do p1
   case T2 (...) (...) return R2 do p2
   ...
   case Tn (...) (...) return Rn do pn
end exec
```

Reference variables can be shared between the cases. As for the multiple await statement, "`do` *pi*" can be omitted if *pi* is just `nothing`.

When a multiple `exec` statement starts, all tasks are started simultaneously and concurrently. Then, one waits for the return signals. When at least one return signal occurs, the `exec` statement terminates instantaneously; at that instant, all non-terminated tasks are aborted, *only one reference argument update is performed*, the one corresponding to the *first terminated case* in the case list, and only the corresponding `do` continuation is taken. In case of abortion, all tasks are aborted simultaneously. In case of suspension, all tasks are suspended simultaneously.

A typical use of the multiple `exec` statement is to try several ways to perform a given computation in parallel, stopping when the first computation is done:

```
exec
   case InvertMethod1 (Matrix) () return R1
   case InvertMethod2 (Matrix) () return R2
   case InvertMethod3 (Matrix) () return R3
end exec
```

All necessary bookkeeping is nicely performed by the ESTEREL compiler.

7.16.8 Immediate Restart of an exec Statement

An `exec` statement may be aborted and restarted immediately. Consider for instance

```
loop
   exec TASK (X) (1) return R;
each I
```

If `I` occurs before task completion, the ESTEREL program signals to the environment that the current instance of `TASK` should be aborted and that a fresh instance should be started right away.

A slightly more difficult situation appears in the following somewhat artificial program fragment borrowed from [3]:

```
loop
   trap T1 in
      loop
         trap T2 in
            exec TASK (X) (1) return R
         ||
            await I do exit T2 end
         end trap
      end loop
   ||
      await I do exit T1 end
   end trap
end loop
```

At first instant, a fresh instance of `TASK` is started. Then, if `I` occurs before `R`, the following happens instantaneously: the inner trap `T2` is exited and an abort information is sent to the environment to abort the running instance of `TASK`; the inner loop loops, and another `TASK` is restarted immediately; however, the outer trap `T1` is also exited, which implies that this new instance of `TASK` is aborted right away; since the "`trap T1`" statement terminates, the outer loop loops, and yet another instance of `TASK` is started, this time in a for-real way.

In such an intricate behavior, the intermediate launching of `TASK` by the inner loop does not provoke any signaling to the environment, the task being simply considered as stillborn by ESTEREL. Only the aborting of the current instance and the starting of the last instance are signaled.

Because of this special handling of stillborn tasks, we can guarantee the following property:

> *At any instant, at most two instances of a task launched by a given* `exec` *statement can be active. The only possibility to have two instances active at the same time is when an already active and not yet terminated instance is aborted, while a fresh instance is started. In* ESTEREL, *this means that the* `exec` *statement is aborted and is instantaneously restarted.*

8 Reincarnation

Because of instantaneous looping of loops, local signals can have several simultaneous instances that we call reincarnations. They pose no particular problem, but one has to be aware of their existence. Here is an example:

```
loop
   signal S in
      present S then emit O1 else emit O2 end;
      pause;
      emit S
   end signal
end loop
```

At first instant, the local signal `S` is declared. It is absent since there is no emitter for it. Therefore, the `else` branch of the `present` statement is taken and `O2` is emitted. At second instant, the `pause` statement terminates and `S` is emitted and set present. The loop body terminates and it is restarted afresh right away. The local signal declaration is immediately re-entered. It declares a *fresh* signal, distinct from the old one, whose status is lost. The fresh incarnation is absent, unlike the old one. The `present` statement tests the fresh incarnation and only `O2` is emitted. Everything happens as if the loop body was duplicated:

```
loop
   signal S in
      present S else emit O end;
      pause;
      emit S
   end signal;
   signal S in
      present S else emit O end;
      pause;
      emit S
   end signal;
end loop
```

In this obviously equivalent statement, the old and fresh incarnations are split into two syntactically distinct signals that happen to bear the same name `S` and the `present` statement is duplicated. In the original form, the single `S` generates two distinct dynamic incarnations, and the `present` statement dynamically tests the current incarnation of the signal.

9 Constructive Causality

The availability of instantaneous broadcasting and control transmission makes it possible to write syntactically correct but semantically non-sensical programs. The constructive semantics mathematically described in [3] characterizes sensible ESTEREL programs. It is the reference semantics of the language. In this section, we briefly present constructive correctness in terms of the intuitive operational semantics of ESTEREL programs, referring to [3] for the mathematical definition. Most of the examples already appeared in [3] and we keep the same names for them here. We also discuss the acyclicity that automatically guarantees constructiveness and is very easy to check at compile-time, unlike constructiveness.

9.1 Non-Reactive and Non-Deterministic Programs

In our class of application, reactivity and determinism are the minimal requirements a program should obey. A program is *reactive* if it provides a well-defined output for each input. A program is *deterministic* if it produces only one output for each input.

Here is the simplest example of a non-reactive program:

```
module P3:
output O;
present O else emit O end
end module
```

Broadcasting means that a signal is present if and only if it is emitted. Here, O cannot be present, otherwise it would not be emitted and therefore absent; it cannot be absent either, otherwise it would be emitted and therefore present.

Here is the simplest example of a reactive non-deterministic program:

```
module P4:
output O;
present O then emit O end
end module
```

Here, O present can be seen as valid since it is justified by the emission of O, and O absent can also be seen as valid because it implies non-emission of O.

9.1.1 Signal Dependency Cycles

Both examples involve an instantaneous dependency cycle between O and itself. Similar examples can be constructed from dependency cycles between two signals O1 and O2. Here is one:

```
module P5:
output O1, O2;
   present O1 then emit O2 end
||
   present O2 else emit O1 end
end module
```

Dependency cycles can easily be constructed for signal values. Consider the example

```
module PV1:
output O : integer;
emit O(0);
every tick do
    emit O(?O+1)
end every
end module
```

The programmer's intention is clear: emit `O(`n`)` at instant n. However, the program makes no sense. Call o the value of `O`. By definition of broadcasting, o must satisfy the equation $o = o + 1$, which is impossible. The right way to write this program is to use a variable that is incremented in the instant:

```
module PV2:
output O : integer;
var X := 0 in
   loop
      emit O(X);
      X := X+1
   each tick
end var
end module
```

This is one of the major uses of variables in ESTEREL[1].

9.1.2 Acyclic Programs

The problems we mentioned are generally called *causality problems*. They resemble deadlocks in asynchronous languages, and they are indeed "instantaneous deadlocks". To avoid them, most synchronous languages require signal dependency to be acyclic. In that case, it is obvious that any signal has one and only one status and one and only one value, i.e., that programs are reactive and deterministic. Acyclicity is very easy to check and it is also quite natural in data-flow programs or in electronic circuits. However, acyclicity has been rejected by ESTEREL users as being too restrictive a condition. Consider the following programs:

```
module P13:
input I;
output O1, O2;
present I then
   present O1 then emit O2 end
else
   present O2 then emit O1 end
end present
end module
```

[1]Data-flow languages have primitives to directly access the previous value of a signal at any time, which makes `PV1` much simpler; such primitives are not yet available in ESTEREL.

```
module P14:
output O1, O2;
present O1 then emit O2 end;
pause;
present O2 then emit O1 end
end module
```

In both P13 and P14, there is a static cyclic dependency between O1 and O2. However, for both programs, it is clear that the cycle is a false one and that everything goes well at run-time. In P13, only one branch of the test can be taken at a time, according to the externally defined status of I. In P14, the dependency from O1 to O2 is valid at first instant only while the reciprocal dependency is valid at second instant only. The imperative syntax of ESTEREL makes the correctness of P13 and P14 obvious, which would not be true of their data-flow counterparts. The more practical example of a cyclic symmetrical bus arbiter will be presented in Section 9.4.

9.2 Logical Correctness

At first glance, it appears natural to simply require programs to be reactive and deterministic. This is what we call *logical correctness*. Logical correctness fits reasonably well with data-flow languages [10], but not with the imperative style of ESTEREL. Consider the following example:

```
module P9:
output O1, O2;
   present O1 then emit O1 end
||
   present [O1 and not O2] then emit O2 end
end module
```

Surprisingly enough, P9 is logically correct, with unique behavior O1 and O2 absent. Indeed, this hypothesis *self justifies*: O1 absent implies non-emission of O1 and non-emission of O2, which is consistent with the assumption. We leave it to the reader to check that no other hypothesis is consistent. The problem is that self-justification does not fit with the standard control propagation intuition of imperative language, where the evaluation of a test should *precede* the evaluation of its branches, at least in a causal sense.

Another interesting example is the ESTEREL analogue of the Boolean equation "O = O and not O":

```
module P12:
output O;
present O then emit O else emit O end
end module
```

Here, O present is justified by O emitted and O absent is not justified since O would be emitted. Once more, the program is logically correct by self-justification, the flow of control going *backwards* from the `then` part to the test.

9.3 Constructiveness

The idea of the constructive semantics is to forbid self-justification and any kind of speculative reasoning, replacing them by pedestrian fact-to-fact propagation. Ignore values and signal expressions for a while, concentrating on pure signal tests of the form "`present S`". We use a three-valued logic for signals, where the status of a signal is *present*, *absent*, or *unknown*. At each instant, the statuses of the input signals are given by the environment and the statuses of the other signals are initially set to *unknown*. The only inferences we can perform are as follows:

1. An unknown signal can be set present if it is emitted.
2. An unknown signal can be set absent if no emitter can emit it.
3. The `then` branch of a test can be executed if the test is executed and the signal is present.
4. The `else` branch of a test can be executed if the test is executed and the signal is absent.
5. The `then` branch of a test cannot be executed if the signal is absent.
6. The `else` branch of a test cannot be executed if the signal is present.

The rules forbid speculative execution, since (3) and (4) can be applied only if it is already known that the `present` statement must be executed. The rules allow free false path pruning, since (5) and (6) can be applied anywhere, see example P2 below.The precise mathematical rules are given in [3].

We say that a program is *constructive* if the status of each local or output signal can be determined using these rules; it is then determined in a unique way. Let us work through an acyclic example:

```
module P1:
input I;
output O;
signal S1, S2 in
   present I then emit S1 end
||
   present S1 else emit S2 end
||
   present S2 then emit O end
end signal
```

We start with status *unknown* for S1, S2, and O. Assume I is present. Then, the first test takes its then branch and emits S1, which sets S1 present. The second test can proceed by terminating, which implies that S2 cannot be emitted since its only emitter has been discarded. Therefore, S2 can be set absent. Finally, the third present statement can proceed and terminate. Since the "emit O" statement is discarded, O can be set absent. Conversely, assume I absent. Then S1 cannot be emitted and is set absent, which triggers emission of S2, which itself triggers emission of O. Notice that present tests are locked until the status of the signal they test becomes known.

Well-behaved cyclic programs are handled without much difficulty. For example, in P13 above, if I is present, then the emitter of O1 is discarded, O1 is set absent, the first present test terminates and discards "emit O2", and O2 is set absent. Here is a more sophisticated example:

```
module P2:
output O;
signal S in
   emit S;
   present O then
      present S then
         pause
      end present;
      emit O
   end present
end signal
end module
```

In P2, S is emitted and set present before the control reaches the test for O. Execution cannot proceed since the status of O is unknown. However, we can perform false path pruning using rule (6) and infer that the implicit else branch of the "`present S`" cannot be executed. This means that the "`emit O`" statement cannot be reached, which implies that `O` can be set absent since it has no other emitter.

Logically incorrect programs are easily rejected. For example, in `P3` or `P4`, there is no way to make any progress from the unknown state. Logically correct programs that require self-justification or speculative computation are rejected as well. For example, in `P12`, the two "`emit O`" statements can neither be executed nor be discarded from the initial *unknown* status of `O`[2].

When a signal has several simultaneous incarnations, each of them must be handled independently. In practice, it is sufficient to reset the status to *unknown* when entering the signal declaration.

9.3.1 Constructiveness and Preemption

Preemption statements are easily handled, noticing that they behave just as tests for the guard at each instant where the guard is active. For example, the following program is not constructive:

```
module P3bis:
output O;
abort
   sustain O
when O
```

At first instant, the guard is inactive and `O` is emitted. At second instant, the guard becomes active, and it must be tested *before* the body is executed, in the constructive order. The body is neither found to be executed nor to be discarded, and the program is non-constructive. At second instant, the program's body just behaves as

```
module P3ter:
present O else
   sustain O
end present
```

[2]In [11, 12], we prove that an electronic circuit that implements the Boolean equation "`O = O or not O`" indeed behaves in a constructive way rather than in a logical one. For some wire and gate delays, the output voltage won't stabilize. We show that constructiveness is the logical counterpart of delay-independent electrical stabilization, which gives strong physical roots to the constructive semantics.

which is a variant of `P3`. If `abort` is replaced by `weak abort`, the problem disappears, since at second instant the statement behaves as

```
module P3bisWeak :
trap T in
   present O then exit T end
||
   sustain O
end trap
```

which is obviously constructive, emits `O`, and terminates.

9.3.2 Constructiveness of Signal Expressions

Signal expressions are evaluated as follows in the constructive semantics: `not` e evaluates to *false* if e evaluates to *true* and conversely; e_1 `or` e_2 evaluates to *true* as soon as one of e_1 or e_2 evaluates to *true*, even if the other one is still unknown. and it evaluates to *false* if both e_1 and e_2 evaluate to *false*. The evaluation of e_1 `and` e_2 is dual. Notice that the evaluation is parallel or lazy: the evaluation of an expression does not require the evaluation of all its subexpressions.

9.3.3 Constructiveness for Valued Signals

Consider now a valued signal `S`. The most general case is that of a combined signal with combination function `F`. Since each emitter can contribute to a part of the final combined value, that value is known only when all emitters are either executed or discarded. Unlike the computation of the status that succeeds as soon as one emitter emits, the computation of the value cannot be lazy.

A reader of the value is an expression '`?S`'. The expression must lock the control until the value is defined. All data operators are strict, i.e., must evaluate all their arguments before giving their result. This is an important difference between pure and valued signals. Let `X` and `Y` be two Boolean-valued signals. For the status test

```
present [ X or Y ] then p else q end
```

the statement p is executed as soon as one of `X` or `Y` is emitted. For the value test

```
if ?X or ?Y then p else q end
```

the statement p is executed if one of `?X` or `?Y` is true, but only after the status of both `X` and `Y` is known[3].

Value handling combines nicely with status handling. A statement such as "`emit S(2)`" should be thought of as a sequence "`emit S; ?S:=2`" (this for a single signal; one should invoke the combination function for a combined signal). Consider the following toy example:

```
module OK :
output O1 : integer, O2: integer;
   emit O1(?O2)
||
   present O1 then emit O2(1) end
end module
```

This program is constructively correct, and both `O1` and `O2` are emitted with value 1. The constructive reasoning is as follows: The "`emit O1`" statement in the first branch is executed, hence, `O1` is present, but its value is still unknown. Since `O1` is present, the `then` branch of the `present` statement is executed, and `O2` is emitted with value 1. From then on, the value `?O2` becomes readable, and the value of `O1` is determined to be 1.

9.4 Constructiveness vs. Acyclicity

Although compile-time constructiveness analysis is available, acyclic programs should be preferred whenever possible, since their compilation is much faster and generally more efficient. However, we mentioned that cyclic programs can be more natural. Let us show the example of a symmetric bus arbitration mechanism[4].

The bus is a ring on which a bunch of identical stations are hooked. At each instant, the user of the bus can request the bus and he can obtain it or not. A priority mechanism arbitrates simultaneous requests. A token defines the current initial station. At any time, the bus is granted to the first station that asks for it, starting from the initial station in clockwise order. To obtain fairness, the token is moved to the next station at each instant.

The ESTEREL code of one station is

[3]In the ESTEREL v5 C translator, the Boolean `or` value operator is implemented by the C '`|`' operator that evaluates both its arguments. It is *impossible* to define a disjunction operator that returns 1 as soon as one if its arguments is 1 in any sequential language such as C, see [6]. Statuses are handled in a very different way.

[4]Thanks to R. de Simone for the example.

```
module STATION :
input Request;         % from user
output Granted;        % to user
output PreviousPassed; % from previous station
output Pass;           % to next station
input Token;           % from previous station
output PassToken;      % to next station
   loop
      present [ Token or PreviousPassed ] then
         present Request then
            emit Granted
         else
            emit Pass
         end present
      end present
   each tick
||
   loop
      present Token then
         await tick;
         emit PassToken
      else
         await tick
      end present
   end loop
end module
```

A bus with three stations is programmed in Figure 1.

The Pass1, Pass2, and Pass3 signals form a static dependency cycle. At any time, the cycle is dynamically cut at the station that possesses the token. This is easily found by the constructive reasoning, that figures out in which order things must be done in each state. However, there is no *uniform* order in which to do things.

```
module BUS :
input Request1, Request2, Request3;
output Granted1, Granted2, Granted3;
signal Pass1, Pass2, Pass3,
       Token1, Token2, Token3,
       PassToken1, PassToken2, PassToken3
in
   emit Token1
||
   run Station1 /
          STATION [ signal Request1 / Request,
                           Granted1 / Granted,
                           Pass3 / PreviousPassed,
                           Pass1 / Pass,
                           Token1 / Token,
                           Token2 / PassToken ]
||
   run Station1 /
          STATION [ signal Request2 / Request,
                           Granted2 / Granted,
                           Pass1 / PreviousPassed,
                           Pass2 / Pass,
                           Token2 / Token,
                           Token3 / PassToken ]
||
   run Station1 /
          STATION [ signal Request3 / Request,
                           Granted3 / Granted,
                           Pass2 / PreviousPassed,
                           Pass3 / Pass,
                           Token3 / Token,
                           Token1 / PassToken ]
end signal
end module
```

Figure 1: The 3-stations BUS program

Finally, notice that dependencies can be somewhat hidden, since circuit generation from Esterel programs is non-trivial. Here is an example

```
trap T in
   loop
      emit O
   each A
||
   await I;
   exit T
end;
emit B
```

Unexpectedly, that statement builds a dependency from `A` to `B`, which may cause a cycle if the reverse dependency exists somewhere else. Understanding why unfortunately requires understanding the circuit translation presented in [3][5].

[5] Since it always pauses, the first branch always returns termination code 1 to the parallel, and the wire that carries this information depends on `A`. The second branch has a wire for **`exit`** `T` that enters the parallel synchronizer at code 2. The two wires join at the synchronizer **`and`** gate at code 2. The wire sourced at this gate reaches the **`emit`** `B` statement, hence the dependency.

References

[1] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[2] G. Berry. Preemption and concurrency. In *Proc. FSTTCS 93*, Lecture Notes in Computer Science 761, pages 72–93. Springer-Verlag, 1993.

[3] G. Berry. *The Constructive Semantics of Pure Esterel.* Draft book, available at http://www.inria.fr/meije/esterel/esterel-eng.html, 1996.

[4] G. Berry. The Esterel primer. Technical report, Ecole des Mines de Paris and INRIA, 1997 (to appear).

[5] G. Berry. The Esterel reference manual. Technical report, Ecole des Mines de Paris and INRIA, 1997 (to appear).

[6] G. Berry, P-L. Curien, and J-J. Lévy. *Full Abstraction for Sequential Languages*, pages 89–132. Cambridge University Press, 1985.

[7] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[8] F. Boussinot and R. de Simone. The Esterel language. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1293–1304, 1991.

[9] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, 1993.

[10] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro'95*, Como (Italy), september 1995.

[11] T. Shiple and G. Berry. Constructive analysis of cyclic circuits. In *Proc. International Design and Test Conference ITDC 96, Paris, France*, 1996.

[12] Thomas R. Shiple, Vigyan Singhal, Gérard Berry, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Analysis of combinational cycles. Technical Report UCB/ERL M96, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, 1996.

Appendix A: Old Syntax

Previous versions of ESTEREL used a different syntax for some constructs. For backward compatibility, we have chosen to still parse the old syntax, also we try to discourage its usage.

Valued signal used to be declared using parentheses as in `emit` statements:

```
output Speed (integer);
output Beeper (combine Beep with CombineBeeps);
```

the standard notation is now the colon '`:`' as for variables.

Abortion used to be written with the `watching` and `upto` keywords. For instance,

```
abort
   p
when S
```

used to be written

```
do
   p
watching S
```

and

```
abort
   p
when S do
   q
end abort
```

used to be written

```
do
   p
watching S timeout
   q
end timeout
```

The `upto` statement used to be written

```
do
   p
upto S
```

with the following meaning:

```
abort
   p; halt
when S
```

This statement turned out to be not fundamental and its name is not fully clear. It is still available.

Index