The Semantics of Pure Esterel

Article · January 1993

DOI: 10.1007/978-3-662-02880-3_12 · Source: CiteSeer

1 author:

Gérard Berry
Collège de France
126 PUBLICATIONS **8,016** CITATIONS

The Semantics of Pure Esterel *

G. Berry
Ecole des Mines, Centre de Mathématiques Appliquées
B.P. 207, 06904 Sophia-Antipolis CDX, France
INRIA Sophia-Antipolis
06560 Valbonne, France
e-mail: berry@cma.cma.fr

Abstract

We present a survey of the main semantics of PURE ESTEREL, the communication kernel of the ESTEREL synchronous reactive language. We start by an informal presentation of the PURE ESTEREL language. We then present the behavioral semantics that defines the language. We define the notion of a haltset that corresponds to a distributed program counter. Using haltsets, we show that PURE ESTEREL programs have only a finite number of states. The behavioral semantics semantics is not effective in the sense that its inference rules do not yield direct proof constructions. We study Gonthier's effective operational semantics used in the ESTEREL V3 compiler. Finally, we study the electrical semantics, i.e. the translation of PURE ESTEREL programs into circuits that forms the basis of the ESTEREL V4 compiler.

1 Introduction

ESTEREL is a synchronous programming language dedicated to hardware or software reactive systems. It is based on the perfect synchrony hypothesis: control transmission, signal broadcasting, and elementary computations are supposed to take no time, making the ouputs of a system perfectly synchronous with its inputs. Perfect synchrony has the major advantage of making concurrency and determinism live together in harmony; it is presented and analyzed in [1,9,6]. The ESTEREL language is presented in [9,13] and in the compiler's documentation. Programming examples can be found in [2,8,12,17,26]. Other synchronous languages are LUSTRE [22], SIGNAL [21], CSML [16], and STATECHARTS [23].

In the development of ESTEREL from pure research to industrial compilers, semantical aspects were always absolutely fundamental. The language itself is defined with the greatest rigor: we think that this is compulsory for the intended applications, which include embedded systems for which absolute precision is mandatory. The various compilers we developed over time directly implement semantically well-understood algorithms.

This paper is devoted to a survey of the mathematical semantics of PURE ESTEREL, the subset of ESTEREL that deals with pure synchronization and concentrates

*Work supported by the french Coordinated Research Projects C3 and C2A. Part of this work was done in cooperation with Digital Equipment Paris Research Laboratory

the difficulties of the language. Most of the material presented here has already been presented in [9,4] and in Georges Gonthier's thesis [20]. Our aim is to gather in a single paper various semantics leading to proper language definition as well as to efficient compiling algorithms. We shall state the main theorems that relate the different semantics and allow us to compare their power. Since the proofs are rater long and technical, we shall only sketch the easiest one, referring to the available material for complete details. We shall not justify the language constructs, since we already have done that in a number of papers [9,13]. We shall also not present denotational semantics. Such semantics have been investigated by Gonthier [20], but they turn out to be of little practical use, at least up to now.

We start in Section 2 by a description of the language syntax and an informal description of the semantics. We first present the language kernel on which the semantics is defined. We then show how to extend the language by defining user-friendly constructs that can be directly defined from kernel ones.

In Section 3, we present the central *behavioral semantics* that is the real language definition. The behavioral semantics is defined by a set of inference rules in Plotkin's Structural Operational semantics style, [27]. However, unlike for classical languages, the rule are not really usable to build interpretors and compilers since they call for the computation of non-trivial fixpoints without suggesting how to really compute them. Nevertheless, the behavioral semantics is *the* reference semantics to which any other semantics must conform. See [20] for theorems stating the equivalence between the denotational and behavioral semantics.

In Section 4, we study the question of program determinism, showing the need to reject programs that exhibit *causality cycles*. Such cycles prevent programs of having a well-defined meaning; they exist in all synchronous languages and are well-known in hardware design. We also study a phenomenon particular to ESTEREL, the fact that a single source code object can have several incarnations within a single program reaction.

In Section 5, we present the coding of program states by *haltsets*. This coding is essential to show that PURE ESTEREL is a regular language, i.e. that it deals with finite state machines. It is heavily used by the hardware and software compilers.

The first "effective" semantics leading to a true compiler was due to L. Cosserat [7] and the author; it served as the basis of the prototype ESTEREL V2 compiler. It will not be presented here, since it is subsumed by Gonthier's *computational semantics* presented in Section 6. This semantics leads to efficient compiling algorithms that translate ESTEREL programs into equivalent finite-state automata. However, it is not as powerful as the behavioral one, since it rejects programs that could be considered as having a deterministic meaning. Correctness of the computational semantics w.r.t. the behavioral one results from a difficult theorem that will only be stated here; see [20] for its proof. The ESTEREL V3 industrial compiler is directly based on the computational semantics.

The last semantics of interest will be the more recent *electrical semantics* presented in [4], where a program is translated into a boolean circuit. It is the basis of the new ESTEREL V4 compiler that translates programs either to hardware circuits or to software code. The electrical semantics has the important advantage of systematically avoiding the state space explosion that may occur when translating programs into finite-state automata as done by the computational semantics. We

briefly discuss causality issues in electrical semantics.

Finally, in Section 8, we discuss how the semantics are used in the actual ESTEREL v3 and ESTEREL v4 compilers.

We shall not discuss two recent additions to ESTEREL that are presented in [5]: the `exec` external task handling statement, and the unification of ESTEREL synchronous deterministic programming and CSP asynchronous non-deterministic programming [24] that we call "Communicating Reactive Processes". Although they are very important in practice, these additions do not bring any real novelty in the semantical treatment of ESTEREL.

2 The Esterel Language

We now present the original PURE ESTEREL language of [3,4]. We explain the language by giving a purely intuitive and non-systematic semantics based on examples. We first present the kernel statements that form the core of the language. Then we present some of the more user-friendly derived statements that are used in actual ESTEREL programming; expanding derived statements into kernel ones is an excellent exercise in ESTEREL programming.

2.1 Modules and Interfaces

The basic object of PURE ESTEREL is the *signal*. Signals are used for communication with the environment as well as for internal broadcast communication. There is a special signal called `tick`, whose role will be explained below.

The programming unit is the module. A module has an interface that defines its input and output signals and a body that is an executable statement:

```
module M:
input I1, I2;
output O1, O2;
input relations
statement
end module
```

Ignore input relations for the moment. At execution time, a module is activated by repeatedly giving it *input events* consisting of a possibly empty set of input signals assumed to be present. For each input event, the module reacts by executing its body and by outputting the emitted output signals that form the *output event*. We assume that the reaction is *instantaneous* or *perfectly synchronous* in the sense that the output event is produced in *no time*. Hence, all necessary computations are also done in no time. In PURE ESTEREL, these computations are either signal emissions or control transmissions between statements; in full ESTEREL, they can be value computations and variable updates as well. The only statements that consume time are the ones *explicitly* requested to do so. The reaction is also required to be *deterministic*: for any state of the program and any input event, there must be exactly one possible output event. Perfect synchrony is discussed at length in [9,1,22,21,23].

In perfectly synchronous languages, a reaction is also called an *instant*; an instant is comparable to a cycle in synchronous hardware, except that the output stabilization time is neglected to simplify the conceptual model.

Input relations are assertions on the behavior of the environment that can be used to restrict input events [9]; this is important for program specification and for efficient compilation. There are two kinds of relations, *exclusions* and *implications*. An exclusion relation is written

```
relation I1 # I2;
```

Such a relation means that no input event can contain `I1` and `I2` together; for example, the command buttons of a wristwatch [2] are declared incompatible to avoid handling simultaneous contradictory commands such as "go to stopwatch mode" and "go to set-watch mode". An implication relation is written

```
relation I1 => I2;
```

and a typical use is "`Second=>Millisecond`".

2.2 Statements

The list of kernel statements is:

`nothing`
`halt`
`emit S`
stat$_1$ `;` *stat*$_2$
`loop` *stat* `end`
`present S then` *stat*$_1$ `else` *stat*$_2$ `end`
`do` *stat* `watching S`
stat$_1$ `||` *stat*$_2$
`trap T in` *stat* `end`
`exit T`
`signal S in` *stat* `end`

The kernel statements are imperative in nature, and most of them are classical in appearance. The `trap-exit` constructs form an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal S in` *stat* `end`" declares a lexically scoped signal `S` that can be used for internal broadcast communication within *stat*. The `then` and `else` parts are optional in a `present` statement. If omitted, they are supposed to be `nothing`.

2.3 Intuitive Semantics

Signal Broadcasting

At each instant, a signal has a single *status*, which is either *present* or *absent*. The status is broadcast, which means that each signal is consistently seen as present or absent by all statements, ensuring determinism. The status of input signals is fixed by the environment. The special `tick` signal is assumed to be always present; it is similar to the constant 1 in circuits. The following *coherence law* determines the status of local and output signals:

A local or output signal is present at an instant if and only if it is emitted by executing an `emit` *statement at that instant.*

Notice that the default status of a signal is to be absent. We impose a restriction which makes the formal treatment simpler: we assume that input signals cannot be internally emitted by a program (this restriction does actually not appear in the compilers).

Elementary Control Transmission

To explain how control propagates, it is better to first give examples using the simplest derived statement that takes time: the waiting statement "`await S`", whose kernel expansion "`do halt watching S`" will be explained in a moment. When it starts executing, this statement simply retains the control up to the first *future* instant where `S` is present. If such an instant exists, the `await` statement terminates immediately; that is, the control is released instantaneously. If no such instant exists, then the `await` statements waits forever and never terminates. If two `await` statements are put in sequence, as in "`await S1; await S2`", one just waits for `S1` and `S2` in sequence: control transmission by the sequencing operator '`;`' takes no time by itself. In the parallel construct "`await S1 || await S2`", both `await` statements are started simultaneously right away when the parallel construct is started. The parallel statement terminates exactly when its two branches are terminated, i.e. when the last of `S1` and `S2` occurs. Again, the '`||`' operator takes no time by itself.

Instantaneous control transmission appears everywhere. The `nothing` statement is purely transparent: it terminates immediately when started. An "`emit S`" statement is instantaneous: it broadcasts `S` and terminates right away, making the emission of `S` transient. In "`emit S1; emit S2`", the signals `S1` and `S2` are emitted simultaneously. In a signal-presence test such as "`present S ...`", the presence of `S` is tested for right away and the `then` or `else` branch is immediately started accordingly. In a "`loop` *stat* `end`" statement, the body *stat* starts immediately when the `loop` statement starts, and whenever *stat* terminates, it is instantaneously restarted afresh. To avoid infinite instantaneous looping, the body of a loop is required not to terminate instantaneously when started.

Preemption

The `watching` and `trap-exit` statements deal with behavior *preemption*, which is the most important feature of ESTEREL.

In the watchdog statement "`do` *stat* `watching S`", the statement *stat* is executed normally up to proper termination or up to future occurrence of the signal `S`, which is called the *guard*. If *stat* terminates *strictly* before `S` occurs, so does the whole `watching` statement; then the guard has no action. Otherwise, the occurrence of `S` provokes immediate preemption of the body *stat* and immediate termination of the whole `watching` statement. Consider for example the statement

```
do
  do
    await I1; emit O1
  watching I2;
  emit O2
watching I3
```

If `I1` occurs strictly before `I2` and `I3`, then the internal `await` statement terminates normally, `O1` is emitted, the internal `watching` terminates since its body terminates, `O2` is emitted, and the external `watching` also terminates since its body does. If `I2` occurs before `I1` or at the same time as it, but strictly before `I3`, then the internal `watching` preempts the `await` statement, `O1` is *not* emitted even if `I1` is present, `O2` is emitted, and the external `watching` instantaneously terminates. If `I3` occurs before `I1` and `I2` or at the same time as them, then the external `watching` preempts its body and terminates instantaneously, no signal being emitted. Notice how nesting watching statements provides for *priorities*.

We can now explain why "`await S`" is defined as "`do halt watching S`". The semantics of `halt` is simple: it keeps the control forever and never terminates. When `S` occurs, `halt` is preempted and the whole construct terminates just as expected. Notice that `halt` is the only kernel statement that takes time by itself.

The `trap-exit` construct is similar to an exception handling mechanism, but with purely static scoping and concurrency handling. In "`trap T in` *stat* `end`", the body *stat* is run normally until it executes an "`exit T`" statement. Then execution of *stat* is preempted and the whole `trap` construct terminates. The body of a `trap` statement can contain parallel components; the `trap` is exited as soon as one of the components executes an "`exit T`" statement, the other components being preempted. However, `exit` preemption is weaker than `watching` preemption, in the sense that concurrent components execute for a last time when exit occurs. Consider for example the statement

```
trap T in
    await I1; emit O1
  ||
    await I2; exit T
end
```

If `I1` occurs before `I2`, then `O1` is emitted and one waits for `I2` to terminate. If `I2` occurs before `I1`, then the first branch is preempted, the whole statement terminates instantaneously, and `O1` will never be emitted. If `I1` and `I2` occur simultaneously, then *both branches do execute* and `O1` is emitted. Preemption occurs only after execution at the concerned instant: by exiting a trap, a statement can preempt a concurrent statement, but it does leave it its "last wills".

Since we accept simultaneity, we must define what it means to exit several traps simultaneously, i.e. define priorities between traps. The rule is simple: *only the outermost trap matters, the other ones being discarded.* For example, in

```
trap T1 in
  trap T2 in
       exit T1
     ||
       exit T2
  end;
  emit O
end
```

the traps `T1` and `T2` are exited simultaneously, the internal trap `T2` is discarded and `O` is not emitted.

Traps also provide a way of breaking loops, which would otherwise never terminate:

```
trap T in
  loop ... exit T ... end
end
```

2.4 Derived Statements

The most important derived statements are now described. The reader will notice the heavy use of traps in their expansion. One has to be conscious that `trap` is an *essential primitive construct* of ESTEREL. Removing traps would simplify the semantics, but it would also suppress much of the programming power.

Waiting Statements

We have already seen the simple `await` derived statement defined by

`await S` $=_{def}$ `do halt watching S`

Notice that "`await tick`" waits for exactly one tick, i.e. terminates at the instant that follows its starting instant.

One often needs to wait for several signals instead of one; this is done by the derived statement

```
await
   case S1 do stat1
   case S2 do stat2
   ...
   case Sn do statn
end
```

The first occurrence of a signal `Si` appearing in the list provokes immediate execution of $stat_i$. Unlike the alternation statement in CSP-like asynchronous languages, our `await-case` statement is deterministic: if several signals appear at the same time, only the first one is the case list is considered, the other ones being discarded. Here is the expansion for the case $n = 2$:

```
trap T0 in
   trap T1 in
      trap T2 in
         do
            await S2;
            exit T2
         watching S1;
         exit T1;
      end;
      stat2;
      exit T0
   end;
   stat1;
end
```

In a `watching` statement, it is often useful to distinguish the cases of body termination and preemption. An additional `timeout` clause executes a timeout statement if preemption occurs before body termination. The derived statement

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

is an abbreviation for

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

The Upto Statement and Temporal Loops

The `upto` statement is a variant of the `watching` statement where body termination is ignored; actual termination occurs only when the signal waited for is present. The definition is:

`do` *stat* `upto S` $=_{def}$ `do [`*stat*`; halt] watching S`

Two *temporal loops* loop over temporal guards. The "`loop` *stat* `each S`" construct expands into

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

The body *stat* starts right away when the "`loop...each`" statement receives the control. Whenever `S` occurs, *stat* is preempted if not yet terminated and it is restarted afresh right away. The other temporal loop is "`every S do` *stat* `end`". It differs only by the fact that `S` is waited for before starting *stat* at initial time. The expansion is simply

```
await S;
loop stat each S
```

Immediate and Boolean Guards

Remember that a signal appearing in a temporal guard refers to an occurrence in the *strict* future of the starting instant. For example, "`await S`" is guaranteed not to terminate instantaneously. It is often useful to take into account an `S` present at the starting instant as well. This is done by the construct

```
await immediate S
```

that abbreviates

```
present S then nothing else await S end
```

The `immediate` keyword can be used for any temporal guard or loop.

In all the above guards, one can use arbitrary boolean expressions in place of single signals. For syntactic reasons, a boolean expression must be surrounded with brackets:

```
every 3 [S1 or not S2] do
   await [(S3 and S4) or (S5 and S6)];
   emit O
end
```

Signal boolean expressions do not add expressive power since they can be replaced by decision trees involving `present` statements (this is not entirely trivial for the `watching` statement).

Sustaining Signals

An `emit S` statement is transient and emits `S` for only one tick. It is often useful to emit a signal "continuously", i.e. at all instants. This is done by the `sustain` statement

```
sustain S    =def    loop emit S each tick
```

Notice the use of the special `tick` signal that is assumed to be always present. Usually, one sustains a signal until another signal is present; this is simply done by writing

```
do
   sustain S1
watching S2
```

In that construct, `S1` is not emitted when `S2` occurs. If one wants to emit `S1` at that instant, one uses a trap:

```
trap T in
   sustain S1
||
   await S2; exit T
end
```

Trap Handlers

Finally, one can declare concurrent traps and one can associate trap handlers to perform statements on exits:

```
trap T1, T2 in
   stat
handle T1 do stat1
handle T2 do stat2
end trap
```

Both handlers are run in parallel if `T1` and `T2` are exited simultaneously. We leave the expansion as a (non-trivial) exercise to the reader.

3 The Behavioral Semantics

Several mathematical semantics have been developed for ESTEREL, including a denotational semantics that precisely formalizes the intuitive temporal concepts presented in section 2, see [20]. Here we prefer to use the *behavioral semantics* [9] that defines execution reaction by reaction, using Plotkin's Structural Operational Semantics technique (SOS for short). It is shown equivalent to the denotational one in [20].

3.1 Form of the Rules

The behavioral semantics defines transitions of the form $M \xrightarrow[I]{O} M'$ where M is a module, I is an input event, O is the corresponding output event, and M' is a new module that will correctly respond to the next input events. In other words, M' is the new state of M after the reaction to I. The reaction $O_1, O_2, \ldots, O_n, \ldots$ to an input sequence $I_1, I_2, \ldots, I_n, \ldots$ is then defined inductively by chaining elementary reactions:

$$M \xrightarrow[I_1]{O_1} M_1 \xrightarrow[I_2]{O_2} M_2 \ldots M_{n-1} \xrightarrow[I_n]{O_n} M_n \xrightarrow[I_{n+1}]{O_{n+1}} \ldots$$

The intermediate terms M_n are called *derivatives* of M; as usual, M is considered to be a derivative of itself by the empty chain of reactions. The precise study of derivatives will be the heart of the next two sections.

A behavioral transition $M \xrightarrow[I]{O} M'$ is computed using an auxiliary relation

$$stat \xrightarrow[E]{E', k} stat'$$

defined by structural induction on statements. Here E is the *current event* in which *stat* evolves, E' is the event made of the signals emitted by *stat*, and k is an integer

termination level that codes the way in which *stat* terminates or exits and is precisely defined below.

The current event E is made of all the signals that are present at the given instant; because of the signal status coherence law, E must contain the set E' of emitted signals, which in turns depends on E. Hence E and E' will be computed as *fixpoints*, the fixpoint equation being located in the local signal rule below.

Let *stat* be the body of M and *stat'* be the body of M'. The relation between the behavioral and auxiliary transition systems is as follows:

$$M \xrightarrow[I]{O} M' \text{ iff } stat \xrightarrow[I \cup \mathtt{O} \cup \{\mathtt{tick}\}]{O,k} stat' \text{ for some } k$$

(under the minor restriction that no input signal is internally emitted by *stat*, see [9]).

Termination Levels

The termination level k of *stat* is determined as follows: it is 0 if *stat* terminates at the current instant, 1 if *stat* does not terminate and waits for further input events (for example, *stat* is "`await S`"), and $k+2$ if *stat* exits a trap `T` that is k trap levels above it, i.e. is if the exit must be propagated through $k-1$ traps before reaching its trap. To avoid counting traps, it is useful to first decorate the `exit` statements with the corresponding level, as in the following example:

```
trap T in
  exit T^2
||
  trap U in
    exit T^3
  ||
    exit U^2
  end
end
```

Here the first `T` exit and the `U` exit are labeled 2 since there is no intermediate `trap` statement to traverse, while the second `T` exit is labeled 3 since one must traverse the `trap U` statement to reach the `trap T` statement.

The key idea of termination level coding is as follows: at each instant, a parallel statement returns the *max* of the termination levels of its branches as its own termination level. Let us check that this is the intented behavior. The fact that a parallel terminates only if all its branches do is expressed by $max(0,0) = 0$. The fact that a parallel must wait if at least one branch waits while the other terminate or wait is expressed by $max(1,x) = 1$ if $x \leq 1$. The fact that an exit preempts termination and waiting results from $max(x,y) = x$ if $x > 1$ and $y \leq 1$. Finally, the fact that only the outermost trap matters if several traps are exited simultaneously results from the mere coding of exit levels.

This way of handling termination is simpler than the one used in [9], but equivalent to it as shown in [20] (see also [19]).

3.2 Inductive Rules

The `nothing` statement terminates instantaneously.

$$\texttt{nothing} \xrightarrow[E]{\emptyset, 0} \texttt{nothing}$$

The `halt` statements waits forever: it waits at the given instant and rewrites into itself.

$$\texttt{halt} \xrightarrow[E]{\emptyset, 1} \texttt{halt}$$

An `emit` statement emits its signal and terminates.

$$\texttt{emit S} \xrightarrow[E]{\{\texttt{S}\}, 0} \texttt{nothing}$$

If the first statement of a sequence terminates, the second statement starts at once; the emitted signals are merged to form the resulting emitted event, according to perfect synchrony.

$$\frac{stat_1 \xrightarrow[E]{E'_1, 0} stat'_1 \qquad stat_2 \xrightarrow[E]{E'_2, k_2} stat'_2}{stat_1\,\texttt{;}\ stat_2 \xrightarrow[E]{E'_1 \cup E'_2, k_2} stat'_2}$$

If the first statement of a sequence does not terminate, that is if it waits or exits a trap, the sequence behaves just as the first statement and the second statement is kept unchanged for further reactions.

$$\frac{stat_1 \xrightarrow[E]{E'_1, k_1} stat'_1 \qquad k > 0}{stat_1\,\texttt{;}\ stat_2 \xrightarrow[E]{E'_1, k_1} stat'_1\,\texttt{;}\ stat_2}$$

A loop instantaneously unfolds itself once. Its body is not allowed to terminate instantaneously.

$$\frac{stat \xrightarrow[E]{E', k} stat' \qquad k > 0}{\texttt{loop}\ stat\ \texttt{end} \xrightarrow[E]{E', k} stat'\texttt{;}\ \texttt{loop}\ stat\ \texttt{end}}$$

A `present` statement instantaneously selects its `then` branch if the signal tested for is present at the current instant. Otherwise, it instantaneously selects its `else` branch.

$$\frac{\mathtt{S} \in E \quad stat_1 \xrightarrow[E]{E_1', k_1} stat_1'}{\mathtt{present\ S\ then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end} \xrightarrow[E]{E_1', k_1} stat_1'}$$

$$\frac{\mathtt{S} \notin E \quad stat_2 \xrightarrow[E]{E_2', k_2} stat_2'}{\mathtt{present\ S\ then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end} \xrightarrow[E]{E_2', k_2} stat_2'}$$

A `watching` statement transfers the control to its body and rewrites itself into a `present` statement in order to set the time guard for further instants if the body waits.

$$\frac{stat \xrightarrow[E]{E', k} stat'}{\mathtt{do}\ stat\ \mathtt{watching\ S} \xrightarrow[E]{E', k} \mathtt{present\ S\ else\ [do}\ stat'\ \mathtt{watching\ S]\ end}}$$

A parallel statements starts its branches instantaneously, merges the emitted signals, and returns the *max* of the termination codes, as explained above.

$$\frac{stat_1 \xrightarrow[E]{E_1', k_1} stat_1' \quad stat_2 \xrightarrow[E]{E_2', k_2} stat_2'}{stat_1\ \mathtt{||}\ stat_2 \xrightarrow[E]{E_1' \cup E_2', max(k_1, k_2)} stat_1'\ \mathtt{||}\ stat_2'}$$

A `trap` terminates if its body terminates or exits the trap, that is, returns termination code 0 or 2. If the body waits, so does the trap. If the body exits an enclosing `trap`, then the exit is propagated by subtracting 1 to the exit level.

$$\frac{stat \xrightarrow[E]{E', k} stat' \quad k = 0 \text{ or } k = 2}{\mathtt{trap\ T\ in}\ stat\ \mathtt{end} \xrightarrow[E]{E', 0} \mathtt{nothing}}$$

$$\frac{stat \xrightarrow[E]{E', k} stat' \quad (k = 1 \text{ and } k' = 1) \text{ or } (k > 2 \text{ and } k' = k - 1)}{\mathtt{trap\ T\ in}\ stat\ \mathtt{end} \xrightarrow[E]{E', k'} \mathtt{trap\ T\ in}\ stat'\ \mathtt{end}}$$

An `exit` statement returns its exit level.

$$\mathtt{exit\ T}^k \xrightarrow[E]{\emptyset, k} \mathtt{halt}$$

Finally, the local signal declaration rules wind up the events E and E' according to the coherence law given in section 2. Within the body, they impose that a local signal is present in E if and only if it is emitted in E'. A local signal is obviously not propagated outside its declaration.

$$\frac{stat \xrightarrow[E\cup\{\mathtt{S}\}]{E'\cup\{\mathtt{S}\},k} stat' \quad \mathtt{S} \notin E'}{\mathtt{signal\ S\ in}\ stat\ \mathtt{end} \xrightarrow[E]{E',k} \mathtt{signal\ S\ in}\ stat'\ \mathtt{end}}$$

$$\frac{stat \xrightarrow[E-\{\mathtt{S}\}]{E',k} stat' \quad \mathtt{S} \notin E'}{\mathtt{signal\ S\ in}\ stat\ \mathtt{end} \xrightarrow[E]{E',k} \mathtt{signal\ S\ in}\ stat'\ \mathtt{end}}$$

Remarks

The resulting statement $stat'$ is unused and therefore immaterial for any rule returning $k > 1$; it is discarded by the exited `trap`. If a rule returns $k = 0$, then its resulting term is equivalent to `nothing`.

Because of the intrinsic fixpoint character of the local signal rule, our inference system does not yield a straightforward algorithm to compute a transition. Given any input I one must guess the right current event E and use the rules to check that there is a correct transition.

3.3 Example

Let us call $P0$ the following statement, where `I` is an input and `O` is an output:

```
signal S1, S2 in
   present I then emit S1 end
||
   present S1 else emit S2 end
||
   present S2 then emit O end
end
```

If I is present, there is only one possible status for local signals: `S1` must be present and `S2` must be absent; this is the only way to apply the local signal rule to both `S1` and `S2`. The output signal `O` is emitted. If `I` is absent, the only possible status is `S1` absent, `S2` present, and `O` absent. Of course, guessing these behaviors is easy by simple visual inspection. In more complex cases, algorithms are needed; they will be the subject of Section 6 and Section 7.

4 Causality Cycles and Signal Reincarnations

Fully adopting perfect synchrony and statement orthogonality yields some unusual problems that we describe here. The first one is the *causality problem* that is intrinsic to the combination of instantaneous broadcast and instantaneous decision.

It appears in other synchronous formalisms such as LUSTRE [22], SIGNAL [21], and STATECHARTS [23]. It is also well-known in hardware description languages.

The second problem, which we call the *reincarnation problem*, is particular to ESTEREL and is due to instantaneous control transmission in loops.

Both problems are direct consequences of the perfect synchrony assumption and can be dealt with accurately. Some authors have proposed to remove them by restricting the language (see for example non-instantaneous communication in CSML, [16]). We think that restrictions are not adequate since they strongly contrive the programmer's freedom. We prefer to perform careful compile-time analysis and produce adequate error messages.

4.1 Causality Cycles

Consider the following statement:

```
present S1 then emit S2 end
```

The statement establishes a *dependency* from `S1` to `S2`: the status of `S2` depends on that of `S1`. *Causality cycles* appear when there are circular dependencies. The simplest example is:

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

By the coherence law, `S` is present if and only if it is emitted. Here there are two distinct *coherent* status for `S`: assuming `S` absent makes sense since "`emit S`" is not executed in that case; assuming `S` present also makes sense since the `then` branch of the `present` statement does emit `S`. The situation resembles a deadlock, and, at first glance, one might accept that setting `S` absent is safe. But replace `then` by `else`:

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

Then there is *no adequate status* for `S`: assuming `S` absent makes it present, while assuming `S` present prevents its emission and makes it absent. Consider now the statement:

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

Here again one can consider that `S1` and `S2` can be either both present or both absent. Replacing `then` by `else` yields:

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

There are now two consistent solutions: `S1` present and `S2` absent, or `S1` absent and `S2` present. There is obviously no best choice.

4.2 Causal Correctness and Determinism

Several ways of thinking can be taken when dealing with causality problems. The first one is to accept them and let run-time systems or simulators detect a deadlock or be non-deterministic. This choice is taken by STATECHARTS. On the contrary, one can reject causally problematic programs at compile-time with appropriate error messages. This is the way we take, and so do LUSTRE, SIGNAL, and most logical-level hardware description languages (the phenomenon being called a *combinational loop* in hardware). Our choice is justified by our basic practical requirements of *reactivity* and *determinism* [1,9]. We wish a reactive system to always react to inputs, and we wish reactions to be fully reproducible. Therefore, we pose the following definition:

> *A program is locally correct if its body and all its substatements are such that each local and output signal can have a single status for any input event that satisfies the input relations. A program is correct if all its derivatives are locally correct.*

Correctness obviously implies determinism. However, our definition of correctness does not really solve the causality problem. Consider the following statement:

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

Although there are two circular dependencies, an easy case inspection shows that there is only one consistent choice: `S1` absent and `S2` absent! Since the program is behaviorally deterministic, we could accept it; see for example [11] for the analysis of this choice. In practice, we prefer to consider this program as being ill-behaved w.r.t. causality and to reject it.

Altogether, we are faced with the task of developing adequate causality analysis techniques that accept only "well-behaved programs", ensure *sufficient* determinism conditions, and are reasonably cheap at compile time to scale up to real-size programs. Note that checking determinism is probably NP-complete, although we did not verify that point exactly; it amounts to find whether a boolean formula is satisfiable in a unique way.

4.3 Static or Dynamic Causality

In all our previous examples, the causality problems appeared at first instant. Consider now the statement:

```
signal S1, S2 in
   await I1;
   present S1 then emit S2 end
||
   await I2;
   present S2 then emit S1 end
end
```

where `I1` and `I2` are two input signals. At first instant, both `await` statements just wait. Assume `I1` and `I2` appear at some further instant. Then we get the derivative:

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

which is known as causally incorrect. Hence, causal correctness must be checked over time, that is, in all derivatives. Input relations must be taken into account; if we add the exclusion

```
relation I1 # I2;
```

then `I1` and `I2` cannot appear at the same time and the program can be accepted: the potential danger never shows up for acceptable inputs.

Causality analysis is a complex issue; the right tradeoffs between accepting enough practical programs, producing good error messages for rejected programs, and keeping compile time reasonable are still a question of debate and are unfortunately well outside the scope of this paper. Roughly speaking, we use two essentially distinct kinds of analysis for ESTEREL.

- In the *dynamic analysis* technique, we compute all possible derivatives and perform a separate analysis for each of them. This technique is explained in Section 6 and is used in the ESTEREL V3 compiler. The above program is accepted if the input exclusion relation is given and rejected otherwise.

- In the *static analysis* technique, we check for causal correctness independently of derivative reachability, which is clearly weaker. We impose sufficient conditions on the source programs that are enough to guarantee that no derivative will exhibit a cycle. The above program is found incorrect since a circular dependency between `S1` and `S2` *might exist*, even if the input relation is given. This is the solution taken in the hardware translation of ESTEREL and the ESTEREL V4 compiler, see Section 7.

The dynamic analysis technique is much finer but is bound to fully compute reachability, which is worst case exponential. Static analysis is more restrictive and sometimes rejects programs which their authors would like to be accepted. However, it is very fast and it can be used to produce small circuits or run-time codes even for big programs that have an exponential number of reachable states. From our present experience, it seems that static analysis is acceptable for the majority of practical programs.

4.4 Reincarnations

Let us now turn to the reincarnation problem. Consider the following statement:

```
loop
   signal S in
      present S then emit O end;
      await I;
      emit S
   end
end
```

where I is an input signal. Let us first follow the intuitive semantics. At first instant, S is not emitted, O is not emitted either, and one waits for I. When I occurs at a further instant, the "await I" statement terminates and S is emitted. Since the loop body is terminated, it is restarted right away. The local signal declaration "signal S" opens a *new* scope for S, which is considered as *absent* by the "present S" test since S was emitted in the *old* scope. Therefore, we have dealt with two *distinct incarnations* of S at the same instant.

This reincarnation process is made explicit by the behavioral semantics. At the second instant, the derivative to be executed is

```
signal S in
   present I else await I end;
   emit S
end;
loop
   signal S in
      present S then emit O end;
      await I;
      emit S
   end
end
```

When invoked at first instant, the loop rule has duplicated the loop body. In the derivative, the incarnations correspond to two distinct signals having the same name but not the same scope. The emission of the first S is not seen by the second present statement.

Notice that duplicating the body of a loop preserves its semantics. To make the incarnations already apparent at first step, we can rewrite the original statement in the following form:

```
loop
   signal S in
      present S then emit O end;
      await I;
      emit S
   end;
   signal S in
      present S then emit O end;
      await I;
      emit S
   end
end
```

Then there is no more reincarnation phenomenon.

It is easy to see that there is a finite reincarnation bound for each given program: expanding each loop body once in an inside-out way for nested loops physically separates the logical incarnations. However, such a process is exponential; a careful analysis of the above examples shows that one can do much better, but this is well outside the scope of this survey paper.

Consider now the following trickier example of Gonthier:

```
loop
   trap T in
      signal S in
         loop
            present S then emit O1 else emit O2 end
         each tick
      ||
         await tick; emit S; exit T
      end
   end
end
```

At first instant, S is absent and `O2` is emitted. At second instant, `S` is emitted, the `then` branch of the concurrent `present` statement is taken, and `O1` is emitted. At the very same instant, `T` is exited and the parallel statement restarts with a fresh reincarnation of `S`. The `present` statement is re-executed with this new `S` absent, and `O2` is emitted. Therefore, the `present` statement is executed twice at the same instant with two different branch choices. Again, copying the body of the loops shows that there is no *problem*, i.e. no confusion between the two incarnations of `S`. There is simply a *difficulty* one has to be aware of when designing semantics and compilers.

By nesting enough parallels, traps, and local signal declarations, it is easy to find examples where the same statement is executed n times for any fixed n. Here is the case $n = 3$:

```
loop
   trap T1 in
      signal S1 in
         loop
            trap T2 in
               signal S2 in
                  stat
               ||
                  await tick; emit S2; exit T2
               end
            end
         end
      ||
         await tick; emit S1; exit T2
      end
   end
end
```

Here *stat* is executed three times: with `S1` and `S2` present, with `S1` present and `S2` absent when looping the inner loop, and with both signals absent when looping the outer loop.

The reader might think that the reincarnation phenomenon is a weird and undesirable one and that it should be forbidden, for example by manually expanding loops as above. However, practice has shown that reincarnation is a useful practical programming device; see for instance the graphical menubar program in [17]. Furthermore, let us repeat that the semantics and compilers do know how to handle reincarnations, see Section 6 and Section 7; we see no reason to put any restriction at the language level.

5 The Haltset Coding of Derivatives

We now present an essential concept of the theory of ESTEREL, the unambiguous coding of any derivative by a set of control points in the original program. So far, the behavioral semantics derivatives are defined by a rewriting process and bear no obvious structural relation with the source term *stat*. We show that any derivative can be unambiguously coded by a *haltset H* of *stat*, that is by a set of occurrences of `halt` statements in the kernel statement *stat*. Each `halt` acts as a control point, by denoting a place where the program is waiting for further events; notice that the expansion of any derived temporal statement generates at most one `halt`. Since ESTEREL is concurrent, a state is given by a *set* of such control positions, i.e. by a haltset.

The haltset coding is important in two respects. First, its existence shows the regularity of ESTEREL: only finitely many derivatives can be generated by the rewritings of a given statement. Second, it is the direct basis of the hardware implementation, and it is also heavily used in the software implementation.

In the sequel, we consider a fixed correct module `M` of expanded body *stat*. For technical reasons, we assume that the body of `M` never terminates, adding a trailing `halt` if necessary; this does not change the observable behaviors.

Consider for example the derivatives of "`await S1; await S2; halt`". There are three `halt` statements, the two first ones being respectively generated by the first and the second `await`. Number them $0, 1, 2$. The whole statement itself will be coded by the empty haltset $\emptyset$. The derivative that waits for `S1` is

```
present S1 else
  await S1
end;
await S2;
halt
```

Its haltset will be $\{0\}$, the index of the `halt` generated by the active "`await S1`" statement. The derivative that waits for `S2` is

```
present S2 else
  await S2
end;
halt
```

Its haltset will be $\{1\}$ since the second `await` is active. The final derivative is `halt`, coded by $\{2\}$. Non-singleton haltsets will be constructed by the parallel operator, which will return the union of the haltsets of its branches.

5.1 Haltsets

We number all occurrences of `halt` in *stat* by distinct integers from 0 to n, $n > 0$. Then a haltset H is a subset of $[0..n]$. that satisfies the following *separation* condition: If $stat_1$ and $stat_2$ are the two statements of a sequence or the two branches of a `present` test, then H cannot contain an occurrence of `halt` in $stat_1$ together with an occurrence of `halt` in $stat_2$.

We decorate the behavioral semantics rules by returning a haltset H when executing a numbered term. This haltset will record the places where the term is waiting. The rules take the new form $stat \xrightarrow[E]{E', k, H} stat'$. We always return $H = \emptyset$ when $k \neq 1$ and $H \neq \emptyset$ when $k = 1$. Adding haltsets is easy for all rules except the parallel one. Executed `halt` statements are put into the haltset by the rule of `halt` and propagated by the other rules. Since the transformation is fairly obvious, we just list a few rules and leave the other ones to the reader.

$$\texttt{nothing} \xrightarrow[E]{\emptyset, 0, \emptyset} \texttt{nothing}$$

$$\texttt{halt}^i \xrightarrow[E]{\emptyset, 1, \{i\}} \texttt{halt}^i$$

$$\frac{stat_1 \xrightarrow[E]{E_1', 0, \emptyset} stat_1' \qquad stat_2 \xrightarrow[E]{E_2', k_2, H_2} stat_2'}{stat_1\,\texttt{;}\ stat_2 \xrightarrow[E]{E_1' \cup E_2', k_2, H_2} stat_2'}$$

$$\frac{stat_1 \xrightarrow[E]{E_1', k_1, H_1} stat_1' \qquad k_1 > 0}{stat_1\,\texttt{;}\ stat_2 \xrightarrow[E]{E_1', k_1, H_1} stat_1'\,\texttt{;}\ stat_2}$$

$$\frac{stat \xrightarrow[E]{E', k, \emptyset} stat' \qquad k = 0 \text{ or } k = 2}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E', 0, \emptyset} \texttt{nothing}}$$

$$\frac{stat \xrightarrow[E]{E', k, H} stat' \qquad (k = 1 \text{ and } k' = 1) \text{ or } (k > 2 \text{ and } k' = k - 1)}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E', k', H} \texttt{trap T in } stat' \texttt{ end}}$$

For a parallel, we return the union of the haltsets returned by the branches unless one of the branches exits a trap, in which case we return an empty haltset. We make an additional technical modification explained later on: when one branch terminates, we rewrite it into `nothing`.

$$\frac{\begin{array}{c} stat_1 \xrightarrow[E]{E'_1, k_1, H_1} stat'_1 \\ stat_2 \xrightarrow[E]{E'_2, k_2, H_2} stat'_2 \\ H = \begin{cases} H_1 \cup H_2 & \text{if } max(k_1, k_2) \leq 1 \\ \emptyset & \text{if } max(k_1, k_2) > 1 \end{cases} \\ stat''_i = \begin{cases} stat'_i & \text{if } k_i \neq 0 \\ \texttt{nothing} & \text{if } k_i = 0 \end{cases} \end{array}}{stat_1 \texttt{ || } stat_2 \xrightarrow[E]{E'_1 \cup E'_2, max(k_1,k_2), H} stat''_1 \texttt{ || } stat''_2}$$

Since a module body is supposed to always wait, its global termination code must be 1. Hence the rules always returns a well-defined haltset H for any immediate derivative. This haltset is easily seen to satisfy the separation condition.

5.2 Recovering derivative from haltsets

We now recover the derivative $stat'$ from $stat$ and H. We proceed in two steps. First we define a labeled term $stat^H$ obtained by labeling the subterms of $stat$ by either $H+$ or $H-$; a subterm is labeled $H+$ if and only if it contains at least one occurrence of `halt` whose number is in H. If we care about the label of $stat^H$ itself, then we write it explicitly, as in $stat^{H+}$. The labels are of course redundant with H, but they make the definitions and proofs much simpler to write.

Then we define a term $\mathcal{R}(stat^H)$ by structural induction on $stat^H$. Subterms labeled by $H-$ and `halt` statements are left unchanged.

$$\begin{aligned} \mathcal{R}(stat^{H-}) &= stat \\ \mathcal{R}(\texttt{halt}^{i\,H}) &= \texttt{halt}^i \end{aligned}$$

`trap` and local signal declaration constructs are handled by trivial structural induction.

$$\begin{aligned} \mathcal{R}(\texttt{trap T in } stat \texttt{ end}^H) &= \texttt{trap T in } \mathcal{R}(stat^H) \texttt{ end} \\ \mathcal{R}(\texttt{signal S in } stat \texttt{ end}^H) &= \texttt{signal S in } \mathcal{R}(stat^H) \texttt{ end} \end{aligned}$$

The only non-trivial cases are:

$$\begin{aligned}
\mathcal{R}(stat_1^{H+};\ stat_2^{H-}) &= \mathcal{R}(stat_1^{H+});\ stat_2 \\
\mathcal{R}(stat_1^{H-};\ stat_2^{H+}) &= \mathcal{R}(stat_2^{H+}) \\
\mathcal{R}(\texttt{loop}\ stat_1^{H+}\ \texttt{end}) &= \left|\begin{array}{l}\mathcal{R}(stat_1^{H+});\\ \texttt{loop}\ stat_1\ \texttt{end}\end{array}\right. \\
\mathcal{R}(\texttt{present S then}\ stat_1^{H+}\ \texttt{else}\ stat_2^{H-}\ \texttt{end}) &= \mathcal{R}(stat_1^{H+}) \\
\mathcal{R}(\texttt{present S then}\ stat_1^{H-}\ \texttt{else}\ stat_2^{H+}\ \texttt{end}) &= \mathcal{R}(stat_2^{H+}) \\
\mathcal{R}(\texttt{do}\ stat_1^{H+}\ \texttt{watching S}) &= \left|\begin{array}{l}\texttt{present S else}\\ \quad\texttt{do}\ \mathcal{R}(stat_1^{H+})\ \texttt{watching S}\\ \texttt{end}\end{array}\right. \\
\mathcal{R}(stat_1^{H+}\ \texttt{||}\ stat_2^{H+}) &= \mathcal{R}(stat_1^{H+})\ \texttt{||}\ \mathcal{R}(stat_2^{H+}) \\
\mathcal{R}(stat_1^{H+}\ \texttt{||}\ stat_2^{H-}) &= \mathcal{R}(stat_1^{H+})\ \texttt{||}\ \texttt{nothing} \\
\mathcal{R}(stat_1^{H-}\ \texttt{||}\ stat_2^{H+}) &= \texttt{nothing}\ \texttt{||}\ \mathcal{R}(stat_2^{H+})
\end{aligned}$$

Notice that these definitions make sense only when the separation condition is satisfied. Notice also why we return `nothing` in the haltset semantics rules when a branch terminates: this simplifies the definition of $\mathcal{R}$.

Since they exactly reproduce the (new) behavioral rules right-hand side terms, one easily shows $\mathcal{R}(stat^H) = stat'$ as expected.

We now give the main result: the coding extends from immediate derivatives to general ones. This is not completely obvious since the $\mathcal{R}$ operator can duplicate halts in the `loop` case. The result is as follows:

Theorem: Let $stat$ be the body of a correct program. Let H be a haltset in $stat$. Then for any behavioral rewriting of the form

$$\mathcal{R}(stat^H) \xrightarrow[E]{E',1,H'} stat'$$

the haltset H' contains only halts occurring in $stat'$ and one has $stat' = \mathcal{R}(stat^{H'})$.

proof: The proof is by structural induction on $stat$ and by case inspection on the rule applied to the whole term $\mathcal{R}(stat^H)$ to yield $stat'$. All cases being similar, we treat the sequence and the loop as examples. We consider a given current event E.

Let first $stat = stat_1;\ stat_2$. There are three cases according to the labeling generated by H.

- If $stat_2$ is labeled by $H+$, then $\mathcal{R}(stat^H) = \mathcal{R}(stat_2^H)$. By correctness and by the hypothesis that *stat* halts, $\mathcal{R}(stat_2^H)$ has a unique rewriting

$$\mathcal{R}(stat_2^H) \xrightarrow[E]{E',1,H'} stat'$$

where H' is a nonempty haltset that only contains halts in $stat_2$. By induction, one has $stat' = \mathcal{R}(stat_2^{H'+})$. Since H' is all in $stat_2$ and nonempty, one has $\mathcal{R}(stat_2^{H'+}) = \mathcal{R}(stat^{H'})$ by definition of $\mathcal{R}()$ and the result follows.

- The two other cases can be grouped into one. They correspond to a term $stat = \mathcal{R}(stat_1^H)$; $stat_2$, taking H as given if $stat = \mathcal{R}(stat_1^{H+})$; $stat_2$ and $H = \emptyset$ if *stat* itself has label $H-$. By correctness, *stat* has a unique behavior, computed by either the first or the second sequence rule. If the first sequence rule is used, then $stat'$ is generated entirely by $stat_2$ and the results follows as in the first case. If the second sequence rule is used, the termination code of $\mathcal{R}(stat_1^H)$ is 1 since *stat* halts. By induction and by the form of the rule, one has $stat' = \mathcal{R}(stat_1^{H'+})$; $stat_2 = \mathcal{R}(stat^{H'})$ for some nonempty H' having all its halts in $stat_1$. The result follows.

Assume now $stat$ = `loop` $stat_1$ `end`. There are two subcases. If *stat* is labeled by $H-$, then $\mathcal{R}(stat^{H-})$ = `loop` $stat_1$ `end`. The only applicable rule is the loop rule. It asks for computing $stat_1$, which must halt since *stat* does. By induction and by the loop rule, one has $stat \xrightarrow[E]{E',1,H'} \mathcal{R}(stat_1^{H'+})$; *stat* for some H'. The last term is just $\mathcal{R}(stat^{H'})$ as expected. If *stat* is labeled by $H+$, then $\mathcal{R}(stat^{H+}) = \mathcal{R}(stat_1^{H+})$; *stat*. If the first term does not terminate, we proceed as in the first loop case. Otherwise, the loop must be unfolded once and we are back again in the first loop case.

Corollary: Let *stat* be a module body. Then any derivative $stat'$ of *stat* is equal to $\mathcal{R}(stat^H)$ for some haltset H, and there are only finitely many derivatives.

proof: by induction on the length of a rewriting sequence $stat \xrightarrow{*} stat'$, since *stat* itself is equal to $\mathcal{R}(stat^{\emptyset})$ and since *stat* always returns $k = 1$. The finiteness property is obvious since there are only finitely many possible haltsets.

6 The Computational Semantics

We now present Gonthier's *computational semantics*, which gives a partial algorithm to compute the behavioral semantics of a program. The computational semantics is based on a careful study of how control flows within an instant. Instead of being represented by a single transition as in the behavioral semantics, the computation performed at an instant is represented by a terminating sequence of elementary microsteps of a parallel interpretor described by inference rules. The interpretor is non-deterministic but confluent: all possible computations yield the same result, which is precisely the one defined by the behavioral semantics. Causality cycles are detected by absence of possible progress in the interpreter.

We start by explaining informally how the computational semantics works and discuss causality issues. Then, we present the formal rules and the correctness result.

6.1 Informal Presentation

Let us come back to the example of Section 3, with some added decoration:

```
• signal S1⁻, S2⁻ in
    present I then emit S1 end
 ||
    present S1 else emit S2 end
 ||
    present S2 then emit O end
 end
```

The additional symbol ' • ' denotes a program counter that we shall use to represent microsteps control propagation within the instant; the labels $-$ on the local signals represent the fact that their status is presently unknown.

Control first passes through the signal declaration and the program counter is forked by the parallel statement:

```
signal S1⁻, S2⁻ in
    • present I then emit S1 end
||
    • present S1 else emit S2 end
||
    • present S2 then emit O end
end
```

The last two `present` statement cannot proceed since the status of their signals are yet unknown. Since `I` is an input, its status is given by the environment and we can execute the first `present`. Assume that `I` is present. Then the `then` branch is taken:

```
signal S1⁻, S2⁻ in
   present I then  • emit S1 end
||
    • present S1 else emit S2 end
||
    • present S2 then emit O end
end
```

Next, the `emit S1` statement is executed and `S1` is set present, which we indicate by a '+' sign in its declaration:

```
signal S1⁺, S2⁻ in
   present I then emit S1 end •
||
    • present S1 else emit S2 end
||
    • present S2 then emit O end
end
```

The first branch of the parallel is terminated. Since `S1` is present, we can proceed with the implicit `then` branch of the second `present`:

```
signal S1+, S2− in
   present I then emit S1 end •
||
   present S1 else emit S2 end •
||
    • present S2 then emit O end
end
```

Now we are somewhat stuck since the status of S2 is still unknown. Clearly, we have to show that S2 is absent. The idea is to deduce that fact from the absence of potential emitters. We start a forward worst-case analysis from the current program counter positions to see how control could propagate, recording which signals could be potentially emitted on some control path. Here, only O could be emitted in the then branch of the third present. Therefore, S2 cannot be emitted and we are entitled to set it absent, putting a '−' sign on its declaration:

```
signal S1+, S2− in
   present I then emit S1 end •
||
   present S1 else emit S2 end •
||
    • present S2 then emit O end
end
```

We can now proceed and take the implicit else branch of the third present:

```
signal S1+, S2− in
   present I then emit S1 end •
||
   present S1 else emit S2 end •
||
   present S2 then emit O end •
end
```

The computation is now over and O is not emitted. If, instead, we suppose I absent, we first get to the state

```
signal S1−, S2− in
   present I then emit S1 end •
||
    • present S1 else emit S2 end
||
    • present S2 then emit O end
end
```

Forward analysis from current program counter positions shows that only S2 and O can be emitted. We can deduce that S1 is absent and proceed by emitting S2 and O.

As we shall see in the formal presentation, the computational semantics also computes the residual of a term for a given input.

Causality Cycles Detection

Causality cycles are detected in a simple way: the interpreter "deadlocks" when the program counters reach a state where no signal can be released since all signals are seen to be potentially emissible. Here are some examples:

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

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

Cycle detection is safe, but not optimal: the computational semantics rejects deterministic programs such as:

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

or such as the last example of Section 4. However, the test for acceptance is rather natural: a program is accepted when the status of all signals cannot be accepted by "guessing the future" in the microsteps of an instant; in other words, dependencies must drive control propagation.

6.1.1 Making the Computational Semantics Symmetric

So far, we have established an asymmetry between present and absent signals: we have stated that a signal is present as soon as it is emitted, using the fordward analysis only to set signals absent. In fact, in the ESTEREL V3 compiler, we use a fully symmetric version. We wait for a signal to be non-potentially emissible to set it present if it has been emitted and absent otherwise. Consider the following statement, written using the derived "or" construct:

```
signal S1, S2 in
   emit S1;
   present [S1 or S2] then emit O end;
   emit S2
end
```

The kernel expansion is:

```
signal S1, S2 in
   emit S1;
   present S1 then
      emit O
   else
      present S2 then emit O end
   end;
   emit S2
end
```

If signals are released as soon as they are emitted, this program is accepted with `S2` and `O` present. However, if "`S1 or S2`" is replaced by "`S2 or S1`", we get

```
signal S1, S2 in
   emit S1;
   present S2 then
      emit O
   else
      present S1 then emit O end
   end;
   emit S2
end
```

which is rejected since `S2` is potentially emissible when analyzing the first `present`. This makes '`or`' non-commutative. Releasing signals only when they are no more potentially emissible rejects symmetrically both programs and makes '`or`' commutative as expected.

Furthermore, in the full ESTEREL language language, the symmetric strategy is really compulsory for combined valued signals which can be emitted at several places and of which the value is known only when all emitters have received the control, see [9].

6.2 Formal Presentation

The formal definition of the computational semantics uses three equally important ingredients: a transition relation $\rightarrow$ between statements which defines how statements act, a partial termination function $\tau(stat)$ which applies to statements that cannot act any more, and a potential function $\pi(stat)$ that computes the signals potentially emitted by a statement. Local signal declarations are decorated by labels as explained above.

We start by defining the partial termination function τ that returns an integer termination code as in the behavioral semantics. It handles terms of the form "`nothing||nothing`", "`trap T in exit T end`", etc. It is undefined for terms such as "`emit S`" that can perform actions.

$$
\begin{aligned}
\tau(\texttt{nothing}) &= 0 \\
\tau(\texttt{halt}) &= 1 \\
\tau(\texttt{exit T}^k) &= k \\
\tau(stat_1\,;\, stat_2) &= \begin{cases} \tau(stat_1) \text{ if } \tau(stat_1) > 0 \\ \tau(stat_2) \text{ if } \tau(stat_1) = 0 \end{cases} \\
\tau(\texttt{loop } stat \texttt{ end}) &= \tau(stat) \text{ if } \tau(stat) > 0 \\
\tau(\texttt{do } stat \texttt{ watching S}) &= \tau(stat) \\
\tau(stat_1 \,||\, stat_2) &= max(\tau(stat_1), \tau(stat_2))
\end{cases}
\end{aligned}
$$

$$\tau(\texttt{trap T in } stat \texttt{ end}) = \begin{cases} 0 \text{ if } \tau(stat) = 0 \text{ or } \tau(stat) = 2 \\ 1 \text{ if } \tau(stat) = 1 \\ k-1 \text{ if } \tau(stat) = k, k > 2 \end{cases}$$

$$\tau(\texttt{signal S}^x \texttt{ in } stat \texttt{ end}) = \tau(stat)$$

We now define the potential $\pi(stat)$ of a statement $stat$, using an auxiliary inductive function $\Pi(stat) = (\pi(stat), K)$ where K is the set of potential termination codes of $stat$. To simplify notation, we write implicitly $\Pi(stat_i) = (\pi_i, K_i)$. We use three auxiliary notations:

$$E\backslash x = \{y \in E \mid y \neq x\}$$

$$Max(K, K') = \{max(k, k') \mid k \in K, k' \in K'\}$$

$$\downarrow K = \begin{cases} \{0\} \text{ if } 0 \in K \text{ or } 2 \in K \\ \cup \ \{1\} \text{ if } 1 \in K \\ \cup \ \{k-1 \mid k \in K, k > 2\} \end{cases}$$

The inductive potential Π is defined as follows:

$$\Pi(\texttt{nothing}) = (\emptyset, \{0\})$$

$$\Pi(\texttt{halt}) = (\emptyset, \{1\})$$

$$\Pi(\texttt{exit T}^k) = (\emptyset, \{k\})$$

$$\Pi(\texttt{emit S}) = (\{\texttt{S}\}, \{0\})$$

$$\Pi(stat_1 \texttt{; } stat_2) = \begin{cases} (\pi_1, K_1) \text{ if } 0 \notin K_1 \\ (\pi_1 \cup \pi_2, (K_1\backslash 0) \cup K_2) \\ \qquad \text{if } 0 \in K_1 \end{cases}$$

$$\Pi(\texttt{loop } stat \texttt{ end}) = \Pi(stat) \text{ if } 0 \notin K$$

$$\Pi(\texttt{present S then } stat_1 \texttt{ else } stat_2 \texttt{ end}) = (\pi_1 \cup \pi_2, K_1 \cup K_2)$$

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

$$\Pi(stat_1 \texttt{||} stat_2) = (\pi_1 \cup \pi_2, Max(K_1, K_2))$$

$$\Pi(\texttt{trap T in } stat \texttt{ end}) = (\pi, \downarrow K)$$

$$\Pi(\texttt{signal S in } stat \texttt{ end}) = (\pi\backslash \texttt{S}, K)$$

Notice that potentials detect possible instantaneous loops, by using the clause $0 \notin K$ in the loop potential definition.

Finally, we give the action rules. They have the form

$$\mathcal{S} : stat \rightarrow \mathcal{S}' : stat'$$

where the signal environment $\mathcal{S}$ records a label $-$, $\dagger$, $+$, or $-$ for each signal. The label $\dagger$ is used when a signal is known to be emitted but before its status is fixed to $+$, an operation that happens only when the signal is found not to be in the potential, see the discussion above. As usual, $\mathcal{S}$ acts as a stack: $\mathcal{S}(\mathtt{S})$ denotes the value of S in $\mathcal{S}$, $\mathcal{S}[\mathtt{S} = x]$ denotes the environment obtained by pushing the new binding, and $\mathcal{S}[\mathtt{S} := x]$ denotes $\mathcal{S}$ where the topmost value of S is changed into x (formal definition left to the reader).
An emission is recorded by storing the $\dagger$ label in the signal environment:

$$\mathcal{S} : \mathtt{emit\ S} \rightarrow \mathcal{S}[\mathtt{S} := \dagger] : \mathtt{nothing}$$

A sequence acts if its first statement acts:

$$\frac{\mathcal{S} : stat_1 \rightarrow \mathcal{S}' : stat_1'}{\mathcal{S} : stat_1\,;\ stat_2 \rightarrow \mathcal{S}' : stat_1'\,;\ stat_2}$$

If the first statement of a sequence is terminated, the second statement can act. This is the only place where termination is used:

$$\frac{\tau(stat_1) = 0 \qquad \mathcal{S} : stat_2 \rightarrow \mathcal{S}' : stat_2'}{\mathcal{S} : stat_1\,;\ stat_2 \rightarrow \mathcal{S}' : stat_2'}$$

A loop unfolds and acts if its body acts:

$$\frac{\mathcal{S} : stat \rightarrow \mathcal{S}' : stat'}{\mathcal{S} : \mathtt{loop}\ stat\ \mathtt{end} \rightarrow \mathcal{S}' : stat'; \mathtt{loop}\ stat\ \mathtt{end}}$$

Parallelism is simply handled by interleaving:

$$\frac{\mathcal{S} : stat_1 \rightarrow \mathcal{S}' : stat_1'}{\mathcal{S} : stat_1 \,||\, stat_2 \rightarrow \mathcal{S}' : stat_1' \,||\, stat_2}$$

$$\frac{\mathcal{S} : stat_2 \rightarrow \mathcal{S}' : stat_2'}{\mathcal{S} : stat_1 \,||\, stat_2 \rightarrow \mathcal{S}' : stat_1 \,||\, stat_2'}$$

A `present` statement selects the adequate branch only if the signal status is known:

$$\frac{\mathcal{S}[\mathtt{S}] = +}{\mathcal{S} : \mathtt{present\ S\ then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end} \rightarrow \mathcal{S} : stat_1}$$

$$\frac{\mathcal{S}[\mathtt{S}] = -}{\mathcal{S} : \mathtt{present\ S\ then}\ stat_1\ \mathtt{else}\ stat_2\ \mathtt{end} \rightarrow \mathcal{S} : stat_2}$$

The `watching` and `trap` statements act as their bodies:

$$\frac{\mathcal{S} : stat \rightarrow \mathcal{S}' : stat'}{\mathcal{S} : \texttt{do}\ stat\ \texttt{watching S} \rightarrow \mathcal{S}' : \texttt{do}\ stat'\ \texttt{watching S}}$$

$$\frac{\mathcal{S} : stat \rightarrow \mathcal{S}' : stat'}{\mathcal{S} : \texttt{trap T in}\ stat\ \texttt{end} \rightarrow \mathcal{S}' : \texttt{trap T in}\ stat'\ \texttt{end}}$$

A local signal declaration acts as its body after having pushed the recorded signal status in the signal memory; it records the new status back after action:

$$\frac{\mathcal{S}[\texttt{S} = x] : stat \rightarrow \mathcal{S}'[\texttt{S} = y] : stat'}{\mathcal{S} : \texttt{signal S}^{x}\ \texttt{in}\ stat\ \texttt{end} \rightarrow \mathcal{S}' : \texttt{signal S}^{y}\ \texttt{in}\ stat'\ \texttt{end}}$$

If the signal is not in the potential, the signal status can be fixed to either $+$ or $-$. This is the only place where potentials are used:

$$\frac{\texttt{S} \notin \pi(stat) \qquad (x = - \ and\ y = -) \ \text{ or } \ (x = \dagger \text{ and } y = +)}{\mathcal{S} : \texttt{signalS}^{x}\ \texttt{in}\ stat\ \texttt{end} \rightarrow \mathcal{S} : \texttt{signalS}^{y}\ \texttt{in}\ stat\ \texttt{end}}$$

Example

The ways in which the various components of the computational semantics interact is subtle. Consider the statement

```
signal S⁻ in
   emit S; present S then emit O end
end
```

The `emit` statement can act. Using the `emit` action rule, the first sequence action rule, and the `signal` action rule, one gets in one action step:

```
signal S† in
   nothing; present S then emit O end
end
```

Although `nothing` is terminated, no action is possible since the status of `S` is only †. However, `S` is not in the potential of the body of its declaration. Its status can be turned to $+$, yielding in one action step:

```
signal S+ in
   nothing; present S then emit O end
end
```

Then one can use the termination predicate for `nothing`, the `emit` action rule, the first `present` action rule, and the second sequence action rule to set the status of `O` to †.

6.3 Correctness of the Computational Semantics

When there are parallel statements, the evaluation can be non-deterministic. The first two results show that this non-determinism is only apparent: the global evaluation process is strongly normalizing and confluent. The first easy lemma states strong normalization, i.e. finiteness of all action sequences. This follows from the fact that a loop body cannot terminate. The second lemma states local strong confluence. It is much harder and proved in [20].

Lemma (strong normalization): For any statement and any signal memory, there exist no infinite sequence of action transitions.

Lemma (strong confluence): Let $\mathcal{S} : stat \rightarrow \mathcal{S}_1 : stat_1$ and $\mathcal{S} : stat \rightarrow \mathcal{S}_2 : stat_2$ be two distinct action transitions. Then there exists a unique pair $\mathcal{S}', stat'$ such that $\mathcal{S}_1 : stat_1 \rightarrow \mathcal{S}' : stat'$ and $\mathcal{S}_2 : stat_2 \rightarrow \mathcal{S}' : stat'$

Call *successful* an action sequence $\mathcal{S} : stat \stackrel{*}{\rightarrow} \mathcal{S}' : stat'$ such that $\tau(stat') = 1$, remembering that a trailing `halt` is always added to module bodies. Causality cycles appear when maximal sequences are not successful. Clearly, by the two above lemmas, all successful action sequences lead to the same result $\mathcal{S}' : stat'$. We shall use successful sequences to show that the computational semantics is adequate w.r.t. the behavioral one. But, before, we have to reconstruct the required residual out of $stat'$. This is easy using an auxiliary expansion function $\mathcal{E}()$ defined as follows:

$$
\begin{aligned}
\mathcal{E}(\texttt{nothing}) &= \texttt{nothing} \\
\mathcal{E}(\texttt{halt}) &= \texttt{halt} \\
\mathcal{E}(stat_1\texttt{;}\ stat_2) &= \begin{cases} \mathcal{E}(stat_2) & \text{if } \tau(stat_1) = 0 \\ \mathcal{E}(stat_1)\texttt{;}\ stat_2 & \text{otherwise} \end{cases} \\
\mathcal{E}(\texttt{loop}\ stat\ \texttt{end}) &= \mathcal{E}(stat)\texttt{;}\ \texttt{loop}\ stat\ \texttt{end} \\
\mathcal{E}(\texttt{do}\ stat\ \texttt{watching S}) &= \left| \begin{array}{l} \texttt{present S else} \\ \quad \texttt{do}\ \mathcal{E}(stat)\ \texttt{watching S} \\ \texttt{end} \end{array} \right. \\
\mathcal{E}(stat_1 \texttt{||} stat_2) &= \mathcal{E}(stat_1) \texttt{||} \mathcal{E}(stat_2) \\
\mathcal{E}(\texttt{trap T in}\ stat\ \texttt{end}) &= \begin{cases} \texttt{nothing} & \text{if } \tau(stat) \in \{0, 2\} \\ \texttt{trap T in}\ \mathcal{E}(stat)\ \texttt{end} & \text{otherwise} \end{cases} \\
\mathcal{E}(\texttt{signal S in}\ stat\ \texttt{end}) &= \texttt{signal S in}\ \mathcal{E}(stat)\ \texttt{end}
\end{aligned}
$$

Define now the computational transition $stat \stackrel{O}{\underset{I}{\Longrightarrow}} stat'$ of a module for an input event I as follows: let $\mathcal{S}_I$ be the signal environment where inputs are allocated with their status as in I, `tick` being allocated with $+$, and where outputs are allocated with

value $-$. Let $\mathcal{S}_I : stat \xrightarrow{*} \mathcal{S}' : stat''$ be any successful action sequence. Construct O as the set of output signals having value † in $\mathcal{S}'$ (output signals are never set to $+$). Finally, take $stat' = \mathcal{E}(stat'')$. Then the expected result of this section is:

Theorem (correctness of the computational semantics): For any module M of body $stat$, if $stat \overset{O}{\underset{I}{\Longrightarrow}} stat'$ holds in the computational semantics, then $stat \overset{O}{\underset{I}{\longrightarrow}} stat'$ holds in the behavioral semantics and this transition is the only possible behavioral transition from $stat$ and `I`.

Once again, see [20] for the proof.

7 The Electrical Semantics

In this section, borrowed from [4], we show how to translate a PURE ESTEREL program into a digital circuit that computes the reaction of the program to any input in one clock cycle. For programs that do not have reincarnation problems, the translation is structural: the circuit logical geometry is the same as that of the original program. The translation is directly based on the haltset coding theory of Section 5. Programs with reincarnations need some logic duplication. This will be briefly explained here, but the formal treatment will be postponed to a forthcoming paper.

We present the main ideas on a first example that involves only `halt` and `watching` statements. Then, on a second example, we show how to handle concurrency and exceptions. We discuss how to handle reincarnations. We formalize the translation and state the correctness result for programs without reincarnations.

We assume basic knowledge about clocked digital circuit. A circuit is made of wires define by boolean expressions of other wires. At each clock cycle, i.e. reactive instant in our sense, the circuit solves the full set of boolean equations to find the value of each wire. Some wires are registers: they have initial value 0, and, at each cycle, their current value is the value to which they were set at the previous cycle.

7.1 A First Example

Consider the following program:

```
module M:
input I, R;
output O;
loop
  loop
    await I; await I; emit O
  end
each R.
```

After an initialization instant in which `I` is ignored, the behavior is to emit `O` every two `I`, restarting this behavior afresh each `R`. Expanded into kernel statements, the body becomes:

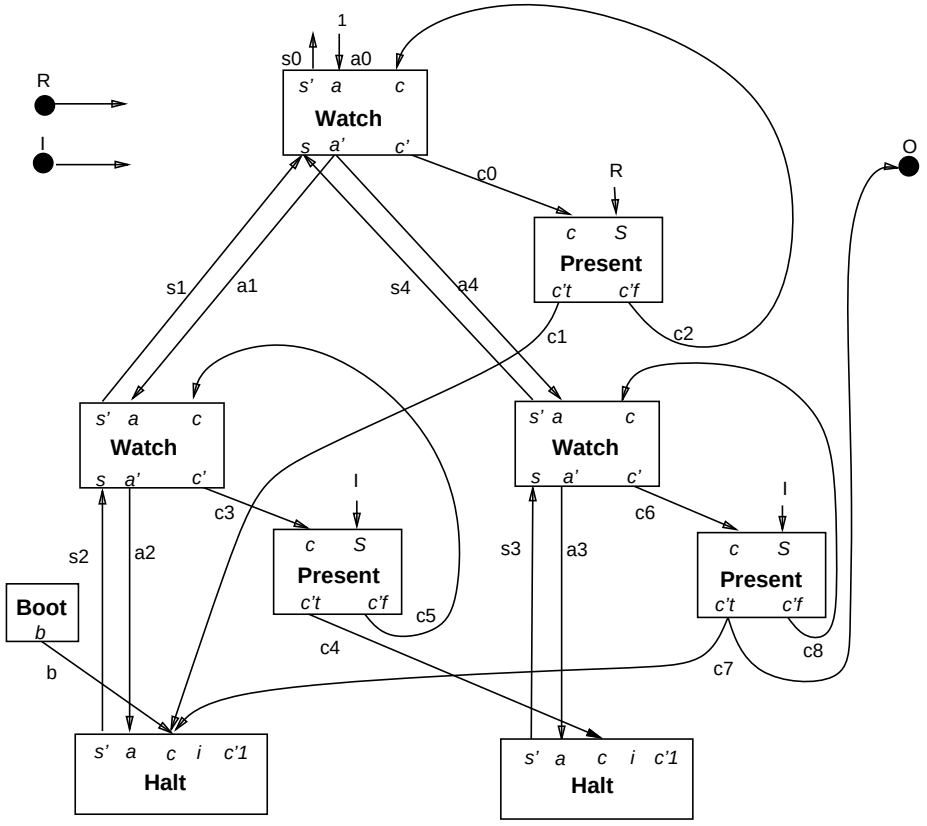

Figure 1: first example

```
loop
  do
    loop
      do
        halt
      watching I;
      do
        halt
      watching I;
      emit O;
    end
  watching R
end
```

The corresponding circuit is drawn in figure 1. It has two input pins for `I` and `R` and one output pin for `O`. There are four kinds of cells, called `Boot`, `Watch`, `Present`, and `Halt`. Cell output pins are primed.

The `Boot` and `Halt` cells each contain one register, assuming to initially contain value 0 and to be clocked by the global circuit's clock. The other cells are purely combinational, i.e., they don't contain registers. The `Present` cell are used for `present` and `watching` source statements, each source "`watching S`" statement being concep-

tually rewritten into "`watch present S`"; This slight syntactic modification simplifies the cells and makes it easy to implement boolean expressions.

The circuit contains three sorts of wires: the *selection* wires `s0`–`s5`, the *activation* wires `a0`–`a5`, and the *control* wires `c0`–`c8`. The unconnected i and $c'1$ pins of `Halt` cells corresponds to other wires unused here and described later on. Whenever two wires go to the same place, they are implicitly assumed to be combined by an `or` gate.

The selection and activation wires go in reverse directions and form a tree that is called the *skeleton* of the circuit. This tree is determined by the nesting of `halt`, `watching`, and `||` statements in the source program, following the abstract syntax revealed by the source code indentation. The leftmost `Halt` and `Watch` cells correspond to the first `await`, the rightmost ones to the second `await`.

The selection wires are used to determine which part of the circuit can be active in a given state: in our example, both `await` statements are in mutual exclusion, and only one of them can be active at a time. When the first `await` is active, the wires `s2`, `s1`, and `s0` are set to 1. When the second `await` is active, the wires `s4`, `s3`, and `s0` are set to 1. The sources of the selection wires are the `Halt` cell registers. The upper selection wire `s0` is unconnected here, but we left it there to emphasize the structural character of the translation.

The activation and control wires bear the flow of control. The activation wires handle preemption between `watching` statements. In our example, the outermost `watching` preempts the innermost one: by the semantics of ESTEREL, if `R` is present, the outermost `watching` terminates without letting its body execute. The upper activation wire `a0` is always set.

The cells are defined as follows:

$$\texttt{Boot} \quad \begin{cases} n := 1 \\ b = \neg n \end{cases}$$

$$\texttt{Watch} \quad \begin{cases} s' = s \\ c' = s \wedge a \\ a' = c \end{cases}$$

$$\texttt{Present} \quad \begin{cases} c't = c \wedge S \\ c'f = c \wedge \neg S \end{cases}$$

$$\texttt{Halt} \quad \{ \; s' := c \vee (a \wedge s)$$

The notation is as follows: '$\vee$' is or, '$\wedge$' is and, '$\neg$' is negation, an equality is valid at all times, and a register is denoted by '$:=$'. Registers are supposed to contain initially 0. In the sequel, we say that a wire is *high* or *set* if it has value 1 and *low* or *reset* if it has value 0. We say that a register is *set* if it gets value 1 and *reset* if it gets value 0. Signals are assumed to be present when their wire is set and absent when their wire is reset.

The output signal b of the `Boot` cell is high at first clock tick and remains then low. For a `Halt` cell, the value of the output signal s' is initially low and then that of $c \vee (a \wedge s)$ delayed one clock cycle. Hence a register is set either if an incoming

control wire is set or if the activation wire is set and the register was already set[1]. The definition of `Halt` is only temporary: further pins will be added in Section 7.2.

A Sample Execution

At boot time, the `Halt` cell registers contain 0 and the selection wires are all low; the boot control wire `b` is high. Because of the cell equations, all other wires are low. Hence the only effect is to set the leftmost `Halt` register.

On next clock tick, assume that `I` is present and `R` is absent. Then `s2`, `s1`, and `s0` are set by the `Halt` register. Since `a0` is always set, the control flows down by setting `c0` that triggers the test for `R` in the upper `Present` cell. Since `R` is low, the control flows through the $c'f$ pin and sets `c2`, which is connected to the c input pin of the `Watch` cell. This pin is directly connected to the a' output pin, and the control flows though `a1` and `a4` (which are connected with each other and form in fact a single equipotential). Since both `s2` and `a1` are high, the leftmost `Watch` cell sets `c3` and the leftmost `Present` cell sets `c4` since `I` is present. This sets the rightmost `Halt` register. Since `s4` is low, the rightmost `Watch` cell is inactive. Having no incoming control set, the leftmost `Halt` register is reset. This terminates the first "`await I`" statement.

On next clock tick, if `I` is present, the execution is symmetrical: the rightmost `Halt` is reset and the leftmost one is set. The wires set are `s3`, `s4`, `a0`, `c0`, `c2`, `a1` = `a4`, `c6`, and `c7`. Since `c7` is also connected to the output `O`, this output is set. If instead `R` is present, the wires set are `s3`, `s4`, `a0`, `c0`, `c1`, and one is back to the state just after boot. If neither `I` nor `R` are present, then the wires set are `s3`, `s4`, `a0`, `c0`, `c2`, `a1` = `a4`, `c6`, `c8`, and `a3`, and the state is simply restored as expected.

Relation with the Haltset Coding

Intuitively, the relation between our circuit and the haltset coding of derivatives is as follows:

- A state of the circuit is a set of `Halt` cells set to 1. It is therefore exactly a haltset.
- The selection wires just compute the $+$ and $-$ labels of statements, $+$ being represented by a 1 in the selection wire.
- Sending the control to the translation of a subterm $stat_1$ by setting an incoming control wire amounts to execute $stat_1$. For example, setting `b` executes the whole statement, setting `b` or `c1` execute the first `await I`, and setting `c4` executes the second `await I`.
- Sending the control to the translation of a subterm $stat_1$ by setting its incoming activation wire amounts to executing $\mathcal{R}(stat_1^H)$ if $stat_1$ is labeled by $+$ in H, i.e. if the corresponding selection wire is set.

[1] The multiplication by s is there to prevent setting the second `Halt` register in a term such as "`do halt; halt watching S`" when a is set.

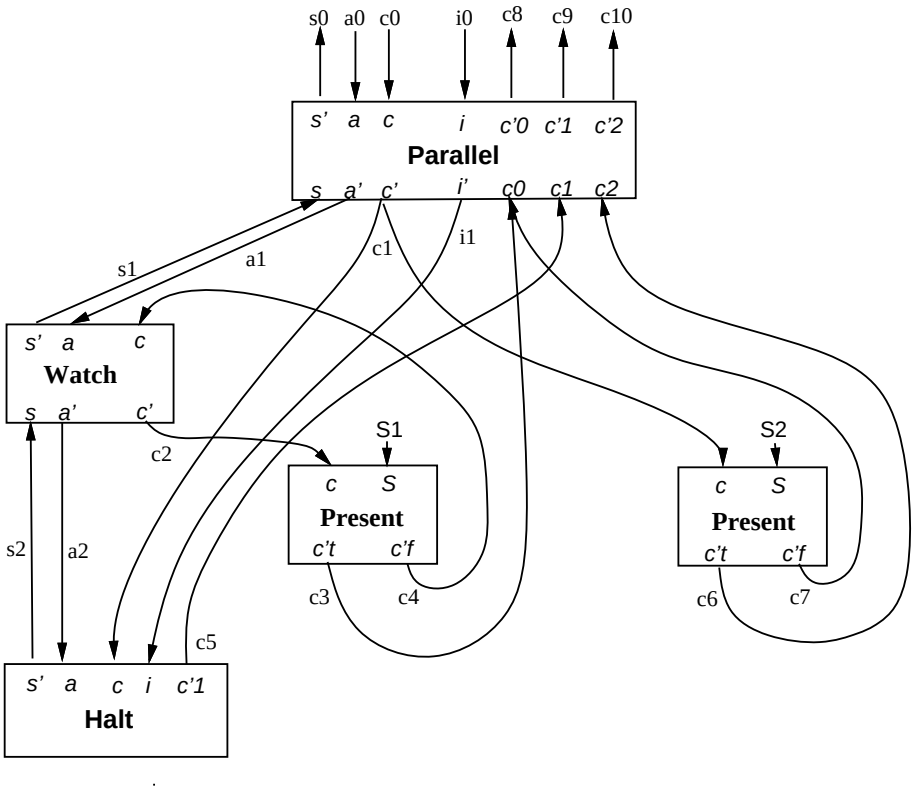

Figure 2: second example

Hence, in a haltset H and an input I, the circuit just mimics the behavioral proof of $\mathcal{R}(stat^H)$ in I. This points will be made very precise in Section 7.6.

Notice that the `Boot` cell is not really necessary since the initial state can also be recognized as the only state where all `Halt` cells have value 0, that is where the wire `s0` is low. We could as well connect the `b` wire to the negation of `s0`. However, it is convenient in practice to add the auxiliary `Boot` cell to reduce the length of wires and the number of logical levels.

7.2 Translating Parallel and Exceptions

The most complex operator is of course the parallel one, since it must synchronize the termination of its branches and propagate exceptions. Consider the following program fragment:

```
trap T in
  await S1
||
  present S2 then exit T end
end
```

The corresponding circuit fragment is shown in figure 2. The leftmost `Watch-Present-Halt` cell group is generated by "`await S1`". The rightmost `Present` cell is generated by "`present S2`", (where "`else nothing`" was omitted as usual). The

branches are simply put in parallel and synchronized by the `Parallel` cell. The circuit fragment starts when it receives control by setting the `c0` wire.

The `Parallel` cell has two parts: the fork part, which involves the six leftmost pins, and the synchronization part, which involves the eight rightmost ones.

The fork part is simple: selection wires are gathered by an `or` gate and activation and control are dispatched to branches.

The synchronization part is more subtle. The pins $c0$, $c1$, and $c2$ record the different termination modes according to the termination codes of Section 3: $c0$ means termination, $c1$ means halt, and $c2$ means exiting `T`. With each termination pin ci is associated a continuation pin $c'i$. (In fact, $c'1$ is not really a continuation in a usual sense: it is recursively linked to the $c1$ entry of the enclosing `Parallel` cell when such a cell exists.)

As explained in Section 3, the synchronization realized by the parallel amounts to compute the *max* of the termination codes of its branches and to only activate the corresponding continuation. It therefore uses a priority queue.

In our example, the left branch can halt, as signaled by setting wire `c5`, or terminate, as signaled by setting wire `c3`. The rightmost branch can terminate or exit `T` as respectively signaled by setting wires `c7` and `c6`. Since exiting `T` or terminating the parallel lead to the same continuation, the continuations wires `c8` and `c10` will reach the same input pin in any global circuit in which our fragment is placed.

When the right branch exits `T`, the leftmost branch must be killed; technically, its `halt` statements must be removed from the current haltset. This is the role of the *inhibition* (or disabling) wire `i1` that sends an inhibition signal to the `halt` register. In an actual execution context, the inhibition signal can also come from an enclosing parallel statement itself killed by some trap exit. It is then received on pin i by the wire `i0`.

The final equations of the `Parallel` and `Halt` cells are:

$$\texttt{Parallel} \quad \begin{cases} s' = s \\ a' = a \\ c' = c \\ c'2 = c2 \\ p1 = c'2 \\ c'1 = c1 \wedge \neg p1 \\ p0 = c1 \vee p1 \\ c'0 = c0 \wedge \neg p0 \\ i' = i \vee p1 \end{cases}$$

$$\texttt{Halt} \quad \begin{cases} c'1 = c \vee (a \wedge s) \\ s' := (c' \vee (a \wedge s)) \wedge \neg i \end{cases}$$

where `p0` and `p1` are local wires used to compute the parallel continuation and inhibition values: if ci is the selected continuation, ci is set and all continuations cj are reset for $j \leq i$, and i' is set if `p1` is.

A Sample Execution

Assume the circuit receives control by `c0` and therefore sets `c1`.

- Assume `S2` is present. Then `c5` is set by the `Halt` cell and `c6` is set by the right `Present` cell. The parallel cell selects the appropriate continuation `c10` and inhibits the halt register by setting `i1`.
- Assume instead `S2` is absent. Then `c5` is set by the `Halt` cell and `c7` is set by the right `Present` cell. The selected continuation is `c9`; it signals halting to an eventual enclosing parallel statement. Since the inhibition wire `i1` is low, the `Halt` cell register is set. The circuit then remains in the same state in further clock cycles as long as the activation wire `a0` remains high and `S1` remains low: the wires set are `s2`, `s1`, `s0`, `a1`, `c2`, `c4`, `a2`, `c5`, and `c9`. If `a0` remains high and `S1` is set, the wires set are `s2`, `s1`, `s0`, `a1`, `c2`, `c3`, and `c8`. The whole construct terminates and the register is reset since `c1` and `a2` are low. The incoming activation wire `a0` can also become low before `S1` occurs, for example because an enclosing watchdog elapses. Then the `Halt` register is also reset.

General Parallel Cells

In fact, the size of the priority queue in a parallel cell depends on the number of nested traps exited from within its source parallel statement. The number of pins $ci, c'i$ for $i \geq 2$ correspond to the number of enclosing traps. With no trap, there is no such pin. The example explained one level of trap. With two levels of traps, as in

```
trap U in
  trap T in
    ...||...
  end
end
```

there would be a pin $c2$ for `T` and a pin $c3$ for `U`, and so on.

7.3 Summary of the Translation

The translation is done by connecting together cells corresponding to source statements. The cells are the same for all programs, but the parallel cells have a variable continuation arity according to the number of enclosing traps.

The logical *skeleton* of the translation is given by the tree of `Halt`, `Watch`, and `Parallel` cells which mimics the tree of source `halt`, `watching`, and `||` statements. Each edge of the tree is composed of an upward *selection* wire and a downward *activation* wire. Two sets of wires reinforce the skeleton: *control* wires that signal halting and go upwards from `Halt` and `Parallel` cells to `Parallel` cells, and opposite *inhibition* wires that force resetting the `Halt` registers in case of exceptions.

In addition to the above cells, one finds a `Boot` cell used to boot the circuit, and `Present` cells generated by source `present` and `watching` statements. These cells are linked together and to skeleton cells by *control* wires. Each `Present` cells also receives as input a *signal* wire. Signal wires come either from input signal pins or from local signal cells, which are simply `or` gates. Control wires transfer the control

from cell to cell. They also emit signals by being connected to output signal pins or to local signal `or` gates. The wiring of control wires is determined by a continuation analysis, see section Section 7.5.

7.4 The Reincarnation Problem

Our translation does not translate correctly all programs. There are difficulties with reincarnations of local signals as well as with loops over parallel statements.

First, we have allocated a single wire for a local signal. Clearly, this cannot deal with signal reincarnation since a wire can have only one status at a time, contrarily to an ESTEREL signal which can have several distinct reincarnations. There is also a reincarnation phenomenon for parallel statements. Consider the statement:

```
loop
  await S
end
```

Its translation is clearly correct. However, the translation of the equivalent statement

```
loop
  await S
||
  nothing
end
```

is not correct since it builds an unstable combinational loop through the parallel synchronizer: when `S` occurs, the parallel terminates, and the detection of that fact requires one synchronization in the priority queue. But the loop restarts its body at once, starting a new incarnation of the parallel statement. A second synchronization then tells that the new incarnation waits. If one tries to use a single synchronizer for both incarnations, one sees that the second synchronization just prevents the termination that should provoke it, hence the combinational loop. Unfolding the body would solve the problem (it still builds a combinational loop, but this time a safe one that can be removed by a careful potential analysis similar to the one done in Section 6).

A clear solution is to perform enough duplications to remove all implicit reincarnations. However, this solution is much too expensive in practice. The ESTEREL V4 compiler uses a clever duplication limited to the *surface* of statements, that is to the part of them that can be reached when the statement is started afresh. This is too technical to be explained here. Since the basic principle of the translation remains the same, it is worth studying formally the case where there are no reincarnations. This is what we now do.

7.5 The Formal Electrical Semantics

We define the translation formally and prove its correctness in absence of reincarnation problems.

Circuits

We consider a circuit to be given by a set of *input wires*, a set of *output wires*, a set of *local wires* and a set of *wire definitions* that define output and local wires. It is technically convenient to use two kinds of wire definitions:

- An *implication* definition $w \Leftarrow exp$ expresses a partial definition, read as "connect exp to w". There can be several implications per wire.
- A *register* definition $w := exp$ defines a wire to be initially 0 and then the value of exp at previous clock cycle. There can be only one register definition per wire.

Given a circuit C and a wire w, the set of implications $w \Leftarrow exp_i$ in C defines w as $w = \bigvee_i exp_i$. Hence the right-hand-sides of implications are connected to an `or` gate. If a wire w has no definition, it is considered to have an empty set of implication definitions, and therefore to be defined by $w = 0$. To stress the fact that a wire has a single implication definition in a circuit, we can write this definition using '=' instead of '$\Leftarrow$'.

Given any register state and any input, the semantics of a circuit is classically defined as a unique fixpoint of the equations, and a circuit is correct if a unique fixpoint always exist in any (reachable) state. We assume this to be well-known.

The Translation Environment

The formal translation is done by natural semantics inference rules [25]. The sequents have the form $\rho \vdash stat \rightarrow C$, where ρ is a wire environment, *stat* is an ESTEREL statement, and C is the resulting circuit.

As in natural semantics or in PROLOG, allocation of new wires is implicit and done when encountering free variables. To make things clear, we shall comment each rule and explicitly tell which are the newly allocated wires.

The environment ρ is made of several wires, whose functions have been explained in Section 7.1. It contains the following fields

- An incoming control wire c.
- A selection wire s.
- An activation wire a.
- An inhibition wire i.
- A vector of continuation wires $\vec{c}$. The wire $\vec{c}^{\,0}$ corresponds to termination, the wire $\vec{c}^{\,1}$ corresponds to halting, the wire $\vec{c}^{\,k+2}$ corresponds to exit $k+2$, that is to exiting k trap levels.
- A set of signal wires `S`, one for each input, output, or local signal `S`. For simplicity, we assume that all local signals have distinct names; then all local signal wires can be preallocated.

We use the classical dot notation to get environment components: for instance, $\rho.c$ denotes the control wire of ρ. Given an environment ρ, we shall often need to consider another environment ρ' that differs from ρ by the value of one field, say by changing $\rho.c$ into c'. We then write $\rho' = \rho[c \leftarrow c']$. The notation extends naturally when changing several fields.

To translate a module, we allocate a boot control wire `b` and a register `n` of equations $\texttt{b} = \neg\texttt{n}$, $\texttt{n} := 1$ as in Section 7.1, a dummy selection wire `s`, two dummy wires `c0` and `c1` for the (unused) continuations, a dummy inhibition wire `i`, and one wire `S` per signal, defining `tick` as set to 1 and declaring respectively input and output signals as inputs and outputs to the circuit. We translate the module body in the environment

$$\rho_0 = (\texttt{b}, \texttt{s}, 1, \texttt{i}, (\texttt{c0}, \texttt{c1}), \vec{\texttt{S}})$$

The Translation Rules

The cells of Section 7.1 were useful for an intuitive explanation, but in rules it is simpler to produce directly equations.

For a `nothing` statement, we connect the incoming control to the termination continuation wire.

$$\rho \vdash \texttt{nothing} \longrightarrow \rho.\vec{c}^{\,0} \Leftarrow \rho.c$$

For a `halt` statement, we connect the incoming control to the halt continuation wire, to signal halting to an enclosing parallel statement. We allocate a new selection wire s' defined as a register with input as explained in Section 7.2. We connect it to the environment selection wire $\rho.s$.

$$\rho \vdash \texttt{halt} \longrightarrow \left| \begin{array}{l} \rho.\vec{c}^{\,1} \Leftarrow \rho.c \vee (\rho.a \wedge \rho.s) \\ \rho.s \Leftarrow s' \\ s' := (\rho.c \vee (\rho.a \wedge \rho.s)) \wedge \neg\rho.i \end{array} \right.$$

For an `emit` statement, we connect the incoming control to the termination wire and to the signal wire.

$$\rho \vdash \texttt{emit S} \longrightarrow \left| \begin{array}{l} \rho.\vec{c}^{\,0} \Leftarrow \rho.c \\ \rho.\texttt{S} \Leftarrow \rho.c \end{array} \right.$$

For a sequence, we allocate a new wire c' for control transmission. We translate the first statement with c' as termination and the second statement with c' as incoming control.

$$\frac{\begin{array}{c}\rho[\vec{c}^{\,0} \leftarrow c'] \vdash \mathit{stat}_1 \longrightarrow C_1 \\ \rho[c \leftarrow c'] \vdash \mathit{stat}_2 \longrightarrow C_2\end{array}}{\rho \vdash \mathit{stat}_1 ; \mathit{stat}_2 \longrightarrow \left| \begin{array}{l} C_1 \\ C_2 \end{array} \right.}$$

For a loop, we allocate a new wire c' to handle looping and we connect the incoming control to it. We translate the body with c' both as incoming control and as outgoing continuation.

$$\frac{\rho[c \leftarrow c', \vec{c}^{\,0} \leftarrow c'] \vdash \mathit{stat} \longrightarrow C}{\rho \vdash \texttt{loop}\ \mathit{stat}\ \texttt{end} \longrightarrow \left| \begin{array}{l} c' \Leftarrow \rho.c \\ C \end{array} \right.}$$

For a `present` statement, we allocate two new control wires c_1 and c_2; then c_1 is set when the incoming control is present and the signal is present, while c_2 is set when the incoming control is present and the signal is absent. We translate the branches with c_1 and c_2 as respective incoming controls.

$$\frac{\begin{array}{c}\rho[c \leftarrow c_1] \vdash \mathit{stat}_1 \longrightarrow C_1 \\ \rho[c \leftarrow c_2] \vdash \mathit{stat}_2 \longrightarrow C_2\end{array}}{\rho \vdash \texttt{present S then}\ \mathit{stat}_1\ \texttt{else}\ \mathit{stat}_2\ \texttt{end} \longrightarrow \left| \begin{array}{l} c_1 = \rho.c \wedge \rho.\mathsf{S} \\ c_2 = \rho.c \wedge \neg\rho.\mathsf{S} \\ C_1 \\ C_2 \end{array} \right.}$$

For a `watching` statement, we allocate a new selection wire s' and connect it to $\rho.s$, and we allocate a new activation wire a'. The outgoing activation wire a' is set if if s' and $\rho.a$ are set and the signal is absent. The outgoing termination wire $\rho.\vec{c}^{\,0}$ is set if s' and $\rho.a$ are set and the signal is present.

$$\frac{\rho[s \leftarrow s', a \leftarrow a'] \vdash \mathit{stat} \longrightarrow C}{\rho \vdash \texttt{do}\ \mathit{stat}\ \texttt{watching S} \longrightarrow \left| \begin{array}{l} \rho.s \Leftarrow s' \\ a' = \rho.a \wedge \rho.s \wedge \neg\rho.\mathsf{S} \\ \rho.\vec{c}^{\,0} \Leftarrow \rho.a \wedge \rho.s \wedge \rho.\mathsf{S} \\ C \end{array} \right.}$$

The parallel rule is of course the most complex one. It follows exactly the intuitive explanation given in Section 7.2. We allocate a selection wire s' connected to $\rho.s$, an inhibition wire i', a continuation vector $\vec{c'}$ of the same length k as $\rho.\vec{c}$, and a priority vector $\vec{p}$ of length $k-1$. We recursively translate the body with the new selection, inhibition, and continuation wires. Then we establish the priority queue to compute the outgoing continuations and the new inhibition wire i'.

$$\frac{\begin{array}{c} k = |\rho.\vec{c}| \\ \rho[s \leftarrow s', i \leftarrow i', \vec{c} \leftarrow \vec{c'}] \vdash stat_1 \longrightarrow C_1 \\ \rho[s \leftarrow s', i \leftarrow i', \vec{c} \leftarrow \vec{c'}] \vdash stat_2 \longrightarrow C_2 \end{array}}{\rho \vdash stat_1 \,||\, stat_2 \longrightarrow \left| \begin{array}{l} \rho.s \Leftarrow s' \\ \rho.\vec{c}^{\,k-1} \Leftarrow \vec{c'}^{\,k-1} \\ \vec{p}^{\,k-2} = \vec{c'}^{\,k-1} \\ \rho.\vec{c}^{\,k-2} \Leftarrow \vec{c'}^{\,k-2} \wedge \neg \vec{p}^{\,k-2} \\ \vec{p}^{\,k-3} = \vec{c'}^{\,k-2} \vee \vec{p}^{\,k-2} \\ \ldots \\ \vec{p}^{\,0} = \vec{c'}^{\,1} \vee \vec{p}^{\,1} \\ \rho.\vec{c}^{\,0} \Leftarrow \vec{c'}^{\,0} \wedge \neg \vec{p}^{\,0} \\ i' = \left\{ \begin{array}{ll} \rho.i & \text{if } k \leq 3 \\ \rho.i \vee \vec{p}^{\,1} & \text{if } k > 3 \end{array} \right. \\ C_1 \\ C_2 \end{array} \right.}$$

For a `trap`, we shift by 1 all wires in $\rho.\vec{c}$ after position 2 and we insert the termination continuation $\rho.\vec{c}^{\,0}$ at exit position 2. The vector notations are obvious.

$$\frac{\rho[\vec{c} \leftarrow (\rho.\vec{c}^{\,0}, \rho.\vec{c}^{\,1}, \rho.\vec{c}^{\,0}) \bullet \rho.\vec{c}^{\,2\cdot\cdot}] \vdash stat \longrightarrow C}{\rho \vdash \texttt{trap T in } stat \texttt{ end} \longrightarrow C}$$

For an `exit`, we connect the incoming control to the appropriate continuation.

$$\rho \vdash \texttt{exit T}^k \longrightarrow \rho.\vec{c}^{\,k} \Leftarrow \rho.c$$

For a local signal declaration, we simply translate the body since the signals have been pre-allocated.

$$\frac{\rho \vdash stat \longrightarrow C}{\rho \vdash \texttt{signal S in } stat \texttt{ end} \longrightarrow C}$$

7.6 Correctness of the translation

We first explain roughly the proof idea as if the translation was always correct. Consider the body *stat* of a correct module placed in the initial environment where the local signal wires have been cut. Then there are two separate wires for each local signal, one for input and one for output. Consider a signal environment E and a haltset H. There exists a unique behavior $stat_1 \xrightarrow[E]{E', 1, H'} stat_1'$ with $stat_1' = \mathcal{R}(stat_1^{H'})$, and a unique behavior $\mathcal{R}(stat_1^{H}) \xrightarrow[E]{E', 1, H''} stat_1''$ with $stat_1'' = \mathcal{R}(stat_1^{H''})$.

The circuit fragment $C(stat_1)$ obtained by translating $stat_1$ has two incoming control wires c and a. Then setting c realizes the first behavior, while setting the activation wire a realizes the second behavior. Furthermore, because of loops, c and

a can be both set. Then the circuit sums up both behaviors with no interference between them. The proof goes simply by structural induction.

Once this is shown, close the local signal wires. Then, for the module body *stat*, for any state H and real input event I, there exists a unique local event L and a unique output event O such that

$$\mathcal{R}(stat^H) \xrightarrow[I \cup L \cup O \cup \{\texttt{tick}\}]{O, 1, H'} \mathcal{R}(stat^{H'})$$

But closing the local signal wires in the circuit has exactly the same coherence effect as in the semantics: a signal is present if and only if it is emitted. Since the circuit can do nothing but mimic the behavioral semantics and since there is only one fixpoint in the semantics by the correctness hypothesis, there is only one fixpoint in the circuit and it is the required one[2].

Therefore, one can view the circuit as a *folding of all possible behavioral semantics proof trees* of a program and of its residuals in all possible environments. What the electrons do is to select the right prooftree in one clock cycle given a residual and an input.

The only problem with the above proof argument is that reincarnations are not properly handled. Closing local signals make sense only if they cannot reincarnate. Sending control to a parallel by both c and a does *not* sum up the behaviors if the parallel is schizophrenic: one of the required continuations can be discarded by the other one. Let us call *non-schizophrenic* a program where there are no reincarnations of signals or parallels[3].

Theorem (correctness of the behavioral semantics): For any correct non-schizophrenic ESTEREL module M, the circuit $C(M)$ has exactly the same input-output behavior as M.

See [4] for the proof. In the full translation where reincarnations are properly handled, the same result holds without the non-schizophrenia condition.

Causality

As far as causality is concerned, deterministic programs are translated into logically deterministic circuits. Therefore, causality analysis need not be done at ESTEREL level: it can be transported to the comparatively simpler world of circuits. However, causality in circuits is not that simple either! We discuss this point further in the next section.

8 The Esterel Compilers

The ESTEREL compilers are directly based on the above semantics. They handle the full ESTEREL language, which adds no major complication. The compilers do not

[2] We talk here of abstract circuits, or equivalently we assume that concrete circuits do always find the unique fixpoint when it exists.

[3] Technically speaking, this can be checked using the behavioral or computational semantics: no local signal declaration and no parallel statement should appear twice in the proof of a reaction to an input for any derivative

start from raw ESTEREL source; instead, they use an intermediate code named `ic` that corresponds to kernel ESTEREL where control has been predigested by building a control graph. Again, this is mostly a technical convenience that does not change the semantics.

8.1 The Esterel v3 compiler

The ESTEREL v3 compiler is based on the computational semantics. It translates a program into a finite state machine where states are represented by haltsets. Starting from the empty haltset the compiler computes and records all transitions for all input events satisfying the input relations. This process continues until there is no unexplored haltset. Therefore a full reachability analysis is performed. The automaton is printed out in an intermediate `oc` format common with LUSTRE, and then translated into classical languages such as C or ADA.

The resulting object code is very efficient as far as speed is concerned: concurrency and communication are fully handled at compile time and local signals do not produce actual run-time code on transitions — just like non-terminal in bottom-up parser generation. Space explosion can occur, since finite state machines can be exponential in the source code size. However, most useful medium-size programs tend to correspond to automata having on the order of 10 to 500 states that are easily handled and yield compact code (this practical observation seems to be due to the very nature of control applications). When explosion does occur, it is very important to state as many input relations as possible to reduce the number of possible input events.

8.2 The Esterel v4 compiler

This more recent compiler uses the electrical semantics, and, more precisely, the version that handle reincarnations and was not described here. A program is translated into a circuit that can be either fed into standard circuit CAD systems or translated into software.

As far as hardware applications are concerned, the reader may find that our circuits contain lots of wires and of logical levels, even for simple programs. In fact, this is because they are obtained by a structural translation process; there is much room for automatic optimization. Many wires are simply connected with each other. Many generated logical functions are readily grouped by logic optimizers. Constant folding can also be used: for instance, the top activation wire is always set; using this fact, one can statically simplify many gates. Therefore, our circuits should not be directly implemented; they should instead be given as input to logic optimizers. We presently use optimizers based on Binary Decision Dags (or BDD's), see [14, 18,28]. They drastically reduces the actual size of circuits. They can also discover redundancies between registers and suppress some of them [10]. In practice, circuits are 20% to 40% bigger than human-designed ones, with good speed performance (performance analysis of the circuits and comparizon with other logic synthesis tools will be the subject of a forthcoming paper). The advantage of using ESTEREL for circuits is of course source code quality.

In the software implementation, there are two choices: either producing object

code in standard languages using a classical topological sorting of the wires, or performing a reachability analysis yielding the very same automaton as ESTEREL v3.

The use of topological sorting guarantees that explosion never occurs: even when there are complicated reincarnations, which is very rare in practice, the size of the circuit or of its software translation cannot exceed the square of the size of the source program. In practice, small linear factors are always measured. Relations are not needed to reduce the circuit's size, but they can yield interesting optimizations. Of course, there is a speed penalty compared to finite state machines: signals are implemented by boolean variables instead of being compiled away, and communication does generate code. However, speed performance is still very good. The ESTEREL v4 compiler is also able to produce an oc code finite state machine by state exploration of the generated circuit.

When using topological sorting, the causality analysis is purely static instead of being dynamic as in ESTEREL v3. Therefore, some programs accepted by ESTEREL v3 are rejected by ESTEREL v4. It is too soon to conclude whether this is a real problem in practice, but preliminary experience suggests the contrary. Since the circuit contains the same information as the original program, it is also possible to perform dynamic causality analysis on it. This has not yet been done.

9 Conclusion

In this survey, we have presented the three most important semantics of ESTEREL: the behavioral semantics, that actually defines the language, the computational semantics on which the ESTEREL v3 compiler is based, and the electrical semantics that founds the ESTEREL v4 compiler and the translation of PURE ESTEREL to hardware circuits. Since they have less practical interest, we have left aside other semantics such as the denotational semantics of [20], structural translation to automata [29], or attempts to translate ESTEREL in process calculi [15].

Defining these semantics was not only an intellectual exercise. The goal was really to found industrial-level tools on very precise scientific grounds, which, to us, should become compulsory for critical software or hardware. We hope to have shown that it is now possible to base full scale languages on well-understood formal semantics and theorems.

Many aspects such as formal verification of programs were not handled here. The ESTEREL programming actually environment contains formal verification tools that also make full use of the semantical definition of the language. They will be described elsewhere.

Acknowledgements

We owe much to L. Cosserat, who designed the first usable semantics, to G. Gonthier, who designed the computational semantics and the ESTEREL v3 compiler and who proved the most difficult results, to P. Bertin and J. Vuillemin, who suggested to use ESTEREL for circuits and suggested the electrical semantics, and to H. Touati, who designed the specific optimizer that makes ESTEREL usable for real hardware.

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. Programming a digital watch in Esterel v3_2. Rapport de recherche 08/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

[3] G. Berry. Esterel on hardware. *Philosophical Transaction Royal Society of London A*, 339:87–104, 1992.

[4] G. Berry. A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992.

[5] G. Berry. Communicating reactive processes. In *Proc. 20th ACM Conf. on Principles of Programming Languages, Charleston, Virginia*, 1993.

[6] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[7] G. Berry and L. Cosserat. The synchronous programming languages Esterel and its mathematical semantics. In S. Brookes and G. Winskel, editors, *Seminar on Concurrency*, pages 389–448. Springer Verlag Lecture Notes in Computer Science 197, 1984.

[8] G. Berry and G. Gonthier. Incremental development of an HDLC protocol in Esterel. In *Proc. Ninth International Symposium on Protocol Specification, Testing, and Verification*. North-Holland, 1989.

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

[10] C. Berthet, O. Coudert, and J.-C. Madre. New ideas on symbolic manipulations of finite state machines. In *Proc. of International Conference on Computer Design (ICCD), Cambridge, USA*, 1990.

[11] F. Boussinot. Une sémantique du langage Esterel. Research Report 577, INRIA, 1986.

[12] F. Boussinot. Programming a reflex game in Esterel v3_2. Rapport de recherche 07/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

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

[14] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[15] J. Camerini. Sémantique mathématique de primitives temps-réel. Thèse de troisième cycle, Université de Nice, 1982.

[16] E. M. Clarke, D.E. Long, and K. L. McMillan. A language for compositional specification and verification of finite state hardware controllers. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[17] D. Clément and J. Incerpi. Programming the behavior of graphical objects using Esterel. In *TAPSOFT '89, Springer-Verlag LNCS 352*, 1989.

[18] O. Coudert and J.-C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. of International Conference on Computer Aided Design (ICCAD), Santa Clara, USA*, 1990.

[19] G. Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12:175–192, 1980.

[20] G. Gonthier. Sémantique et modèles d'exécution des langages réactifs synchrones; application à Esterel. Thèse d'informatique, Université d'Orsay, 1988.

[21] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[22] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[23] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[24] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[25] G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. Elsevier Science Publisher B.V. (North Holland), 1988.

[26] G. Murakami and Ravi Sethi. Terminal call processing in Esterel. In *Proc. IFIP 92 World Computer Congress, Madrid, Spain*, 1992.

[27] G. Plotkin. A structural approach to operational semantics. Technical Report report DAIMI FN-19, University of Aarhus, 1981.

[28] H. Savoj, H. Touati, and R. K. Brayton. The Use of Image Computation Techniques in Extracting Local Don't Cares and Network Optimization. In *to appear in Proceedings of IEEE International Conference on Computer-Aided Design*, November 1991.

[29] J-M Tanzi. Traduction structurelle des programmes Esterel en automates. Thèse de docteur-ingénieur, Université de Nice, 1985.