A Deterministic Logical Semantics for Esterel

Olivier Tardieu [1]

INRIA
Sophia Antipolis, France

Abstract

Esterel is a synchronous design language for the specification of reactive systems. There exist two main semantics for Esterel. On the one hand, the logical behavioral semantics provides a simple and compact formalization of the behavior of programs using SOS rules. But it does not ensure deterministic executions for all programs and all inputs. As non-deterministic programs have to be rejected as incorrect, this means it defines behaviors for incorrect programs, which is not convenient. On the other hand, the constructive semantics is deterministic (amongst other properties) but at the expense of a much more complex formalism. In this work, we construct and thoroughly analyze a new deterministic semantics for Esterel that retains the simplicity of the logical behavioral semantics, from which it derives. In our view, it provides a much better framework for formal reasoning about Esterel programs.

Key words: synchronous languages, concurrency theory, structural operational semantics.

1 Introduction

Esterel [7,8] is a high-level imperative parallel programming language for the specification of reactive systems [9,13]. It was born in the eighties [6], and evolved since then. In this work, we consider the Esterel v5 dialect [4,5] endorsed by current academic compilers [1,10]. Pure Esterel is the subset of the full Esterel language where data variables and data-handling primitives are abstracted away. As the issues we are interested in in this work are not related to data in any way, we shall concentrate on the pure Esterel language.

Esterel is a synchronous language [2]. Primitives constructs execute in zero time except for one `pause` instruction. Hence, time flows as a sequence of logical instants separated by explicit pauses. In each instant, several elementary instantaneous computations take place simultaneously.

Esterel deals with signals. Signals have a Boolean status, which obeys the

[1] Email: `olivier.tardieu@sophia.inria.fr`

signal coherence law: in each instant, a signal is absent by default, present if emitted in this instant. In "`present A then emit B end`" for instance, `B` is emitted, thus present, if `A` is present.

Both absence and presence are instantly broadcast, and simultaneously available to all threads of execution. This perfect synchrony hypothesis may result in causality cycles [4,14], as for example in the parallel composition:

```
present A then emit B end || present B then emit A end
```

which admits two possible executions conforming to the signal coherence law:

- both `A` and `B` are present and emitted;
- both `A` and `B` are absent and not emitted.

This program is said to be non-deterministic. Similarly, there exist non-reactive programs with no possible execution, for example:

```
present A then emit B end || present B else emit A end
```

In Esterel, we want programs to have deadlock-free deterministic executions. Therefore, non-reactive and non-deterministic programs have to be rejected as incorrect. Two main semantics have been formalized for Esterel:

- The logical behavioral semantics [3] simply formalizes the signal coherence law. It defines no execution for a non-reactive program, and several distinct executions for a non-deterministic program [2].
- The constructive semantics [4] is inspired from digital circuits and three-valued logic. It only defines a subset of the executions defined by the logical behavioral semantics. By rejecting more "unreasonable" programs than just non-reactive and non-deterministic programs, it ensures that executions can be "causally" computed. As a result, it defines no execution for non-reactive as well as non-deterministic programs.

These two semantics handle non-determinism in opposite manners. Neither is truly convenient.

- On the one hand, an execution defined by the logical behavioral semantics is not necessarily correct, as it may be the execution of a non-deterministic, thus incorrect program. Moreover, non-determinism sometimes compensates for non-reactivity making a program reactive and deterministic although it contains non-reactive or non-deterministic pieces of code.
- On the other hand, the constructive semantics only defines correct executions, but at the expense of a much more complex formalism.

Therefore, we introduce in this work a third alternative semantics that we derive from the logical behavioral semantics. It retains the simple formalism of the logical behavioral semantics, while only defining correct executions. In particular, it makes sure errors do not cancel one another.

[2] In general, determinism and reactivity depend on inputs (cf. Section 4).

The paper is organized as the following. In Section 2, we describe the pure Esterel language. We formalize its logical behavioral semantics in Section 3, and discuss reactivity and determinism in Section 4. We build our deterministic semantics in Section 5. In Section 6, we thoroughly compare the two semantics. We briefly discuss the constructive semantics of Esterel in Section 7, and conclude in Section 8.

2 Syntax and Intuitive Semantics

$p, q ::=$	$\texttt{nothing}$	does nothing, terminates instantly
	$\texttt{pause}$	stops the execution till next instant
	$p\texttt{;}\ q$	runs p, then q if/when p terminates
	$p\ \texttt{||}\ q$	runs p in parallel with q
	$\texttt{loop}\ p\ \texttt{end}$	repeats p forever
	$\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end}$	declares signal S in p
	$\texttt{emit}\ S$	emits signal S
	$\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end}$	runs p if S is present, q otherwise
	$\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end}$	declares, catches exception T in p
	$\texttt{exit}\ T_d$	raises exception T of depth d

Fig. 1. Primitive Pure Esterel Constructs

Without loss of generality, we focus in this work on a kernel language inspired from Berry [4], which retains just enough of the pure Esterel language to attain its full expressive power. Figure 1 describes the grammar of our kernel language, as well as the intuitive behavior of its constructs.

The non-terminals p and q denote *statements* (i.e. *programs*), S *signals* and T *exceptions*. Signals and exceptions are identifiers lexically scoped and respectively declared within statements by the constructs “`signal` S `in` p `end`” and “`trap` T `in` p `end`”.

The infix “`;`” operator binds tighter than “`||`”. Brackets “`[`” and “`]`” may be used to group statements in arbitrary ways. In a `present` statement, `then` or `else` branches may be omitted. For example, “`present` S `then` p `end`” is a shortcut for “`present` S `then` p `else nothing end`”.

2.1 Instants and Reactions

An Esterel statement runs in steps called *reactions* in response to the *ticks* of a *global clock*. Each reaction takes one *instant*. Primitive constructs execute in zero time except for the `pause` instruction. When the clock ticks, a reaction occurs, which computes the *output signals* and the *new state* of the program, from the *input signals* and the *current state* of the program. It may either finish the execution instantly or delay part of it till the next instant, because it reached at least one `pause` instruction. In the latter case, the execution is resumed when the clock ticks again from the locations of the `pause` instructions reached in the previous instant. And so on.

"`emit A; pause; emit B; emit C; pause; emit D`" emits the signal `A` in the first instant of its execution, then emits `B` and `C` in the second instant, finally emits `D` and terminates in the third instant. It takes three instants to complete, that is to say proceeds by three reactions. The signals `B` and `C` are emitted *simultaneously*, as their emissions occur in the same instant of execution. In particular, "`emit B; emit C`" and "`emit C; emit B`" cannot be distinguished in Esterel.

2.2 Synchronous Concurrency

Concurrency in Esterel is synchronous. One reaction of the parallel composition "p `||` q" is made of exactly one reaction of each non-terminated branch, until the termination of all branches. For example,

```
[
  pause; emit A; pause; emit B
||
  emit C; pause; emit D
];
emit E
```

emits `C` in the first instant of its execution, then emits `A` and `D` in the second instant, then emits `B` and `E` and terminates in the third instant.

2.3 Exceptions

Exceptions are lexically scoped, declared and caught by the "`trap` T `in` p `end`" construct, raised by the "`exit` T_d" instruction. The integer d encodes the *depth* of "`exit` T":

- if "`exit` T_d" is enclosed in a declaration of T then d *must* be the number of exception declarations that have to be traversed before reaching that of T;
- if "`exit` T_d" is not enclosed in a declaration of T then d *must* be greater or equal to the number of exception declarations enclosing this `exit` statement.

For example,

```
trap T in
  trap U in
    exit T₁   has depth 1 because of the declaration of U
  ||
    exit U₀   has depth 0
  ||
    exit V₃   could have any depth greater or equal to 2
  end;
  exit T₀     has depth 0
end
```

Such a "De Bruijn" encoding of exceptions for Esterel was first advocated for by Gonthier [11]. As usual, we shall only make depths explicit when necessary.

In sequential code, the `exit` statement behaves as a "goto" to the end of the matching `trap` block. For example,

```
trap T in
  emit A; pause; emit B; exit T; emit C
end;
emit D
```

emits `A` in the first instant, then `B` and `D` and terminates in the second instant. Signal `C` is never emitted.

An exception raised in a parallel context causes all parallel branches to terminate instantly. In the example below, `A` and `E` are emitted in the first instant, then `B`, `F`, and `D` in the second and final one. Neither `C` nor `G` is emitted.

```
trap T in
  emit A; pause; emit B; exit T; emit C
||
  emit E; pause; emit F; pause; emit G
end;
emit D
```

Remark exceptions implement *weak preemption*: "`exit T`" in the first branch does not prevent `F` to be simultaneously emitted in the second one.

Exception declarations may be nested. In the following example, `A` is not emitted, as the outermost exception `T` has priority over inner ones, `U` here.

```
trap T in
  trap U in
    exit T₁ || exit U₀
  end;
  emit A
end
```

In other words, the exception of greater depth has always priority.

2.4 Loops

The statement "`loop emit S; pause end`" emits `S` at each instant and never terminates. Finitely iterated loops may be obtained by combining `loop`, `trap` and `exit` statements, as for instance in the kernel expansions of "`await` S":

`trap T in loop pause; present` S `then exit T end end end`

Loop bodies may not be *instantaneous* [17]. For example, "`loop emit S end`" is not a correct program. Such a pattern would prevent the reaction to reach completion. Therefore, loop bodies are required to raise an exception or retain the control for at least one instant, that is to say execute a `pause` or an `exit` statement in each iteration.

2.5 Signals

The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. The free signals of a statement are said to be *interface* signals for this statement.

In an instant, a signal S is *emitted* iff at least one "`emit` S" statement is executed in this instant. In an instant, the *status* of a signal S is either *present* or *absent*. If S is present then all "`present` S `then` p `else` q `end`" statements executed in this instant, execute their "`then` p" branch in this instant; if S is absent they all execute their "`else` q" branch.

- A local signal is present iff it is emitted.
- An interface signal is present iff it is provided by the *environment*.

Remark an interface signal may be both absent and emitted. For example,

- In "`signal S in emit S; pause; present S then emit O end end`", `S` is present in the first instant of execution only, thus `O` is not emitted by this statement, as `S` is absent at the time of the "`present S`" test.
- In "`signal S in present S then emit O end || emit S end`", both `S` and `O` are emitted, `S` is present.
- In "`emit X; present X then emit O end`", the status of `X` depends on the environment, hence `O` is emitted iff `X` is provided by the environment.

3 Logical Behavioral Semantics

The *logical behavioral semantics* of Esterel [4,11] formalizes the informal semantics of the previous section. It describes the reactions of a statement p via a labeled transition system:

$$p \xrightarrow[I]{O,k} p'$$

where:

- the set I is the set of *present signals*,
- the set O is the set of *emitted signals*,
- the integer k is the *completion of the reaction*,
- the statement p' is the *residual of the reaction*.

Figure 2 expresses the logical behavioral semantics of Esterel as a set of facts and deduction rules in a structural operational style [16].

3.1 Completion Code and Residual

The completion code k and the residual p' encode the status of the execution:

- If $k = 1$ then this reaction does not complete the execution of p.
  It has to be continued by the execution of p' in the next instant.

- If $k \neq 1$ then this reaction ends the execution of p (p' does not matter):
  - $k = 0$ if the execution completes *normally* (without exception).
  - $k = d + 2$ if an exception of depth d *escapes* from p.

In particular, the completion code of "`exit` T_d" is "2+d". In order to compute the completion code "$\downarrow k$" of "`trap` T `in` p `end`" from the completion k of p, we define:

$$\downarrow k = \begin{cases} 0 & \text{if } k = 0 \text{ or } k = 2 \\ 1 & \text{if } k = 1 \\ k - 1 & \text{if } k > 2 \end{cases}$$

Conveniently, if p terminates with completion code k and q with completion code l then "p `||` q" terminates with code "$\max(k, l)$". For example,

$$\texttt{trap T in exit T}_0\ \texttt{||}\ \texttt{exit V}_4\ \texttt{end} \xrightarrow[I]{\emptyset, 3} \texttt{trap T in nothing end}$$

3.2 Present and Emitted Signals

The set I, written below the arrow, lists the signals provided by the environment. It drives the reactions of `present` statements:

- if $S \in I$ and $p \xrightarrow[I]{O,k} p'$ then `present` S `then` p `else` q `end` $\xrightarrow[I]{O,k} p'$.
- if $S \notin I$ and $q \xrightarrow[I]{O,k} q'$ then `present` S `then` p `else` q `end` $\xrightarrow[I]{O,k} q'$.

The set O, written above the arrow, lists the emitted interface signals. In particular,

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

The *signal coherence law* – a local signal is present iff emitted – is enforced for the statement "`signal` S `in` p `end`" by the rules:

(signal+) if S is supposed present in p then it is emitted by p;
(signal−) if S is supposed absent in p then it is not emitted by p.

For instance, for inputs $I = \{\texttt{A}\}$,

$$\frac{\texttt{emit S} \xrightarrow[\{\texttt{A,S}\}]{\{\texttt{S}\}, 0} \texttt{nothing} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 0} \texttt{signal S in nothing end}} \text{ using (signal+)}$$

$$\frac{\texttt{pause} \xrightarrow[\{\texttt{A}\}]{\emptyset, 1} \texttt{nothing} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 1} \texttt{signal S in nothing end}} \text{ using (signal−)}$$

We shall further discuss these rules later.

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

$$\texttt{pause} \xrightarrow[I]{\emptyset, 1} \texttt{nothing} \tag{pause}$$

$$\texttt{exit}\ T_d \xrightarrow[I]{\emptyset, d+2} \texttt{nothing} \tag{exit}$$

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

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 0}{\texttt{loop}\ p\ \texttt{end} \xrightarrow[I]{O,k} p'\texttt{;}\ \texttt{loop}\ p\ \texttt{end}} \tag{loop}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad q \xrightarrow[I]{O',l} q'}{p\ \texttt{||}\ q \xrightarrow[I]{O \cup O', \max(k,l)} p'\ \texttt{||}\ q'} \tag{parallel}$$

$$\frac{S \in I \qquad p \xrightarrow[I]{O,k} p'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k} p'} \tag{present+}$$

$$\frac{S \notin I \qquad q \xrightarrow[I]{O,k} q'}{\texttt{present}\ S\ \texttt{then}\ p\ \texttt{else}\ q\ \texttt{end} \xrightarrow[I]{O,k} q'} \tag{present−}$$

$$\frac{p \xrightarrow[I]{O,2} p'}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,0} \texttt{nothing}} \tag{trap-catch}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 2}{\texttt{trap}\ T\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,\downarrow k} \texttt{trap}\ T\ \texttt{in}\ p'\ \texttt{end}} \tag{trap-through}$$

$$\frac{p \xrightarrow[I]{O,k} p' \qquad k \neq 0}{p\texttt{;}\ q \xrightarrow[I]{O,k} p'\texttt{;}\ q} \tag{sequence-p}$$

$$\frac{p \xrightarrow[I]{O,0} p' \qquad q \xrightarrow[I]{O',k} q'}{p\texttt{;}\ q \xrightarrow[I]{O \cup O',k} q'} \tag{sequence-q}$$

$$\frac{p \xrightarrow[I \cup \{S\}]{O,k} p' \qquad S \in O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O \setminus \{S\},k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \tag{signal+}$$

$$\frac{p \xrightarrow[I \setminus \{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal}\ S\ \texttt{in}\ p\ \texttt{end} \xrightarrow[I]{O,k} \texttt{signal}\ S\ \texttt{in}\ p'\ \texttt{end}} \tag{signal−}$$

Fig. 2. Logical Behavioral Semantics

3.3 Execution

An *execution* of the statement p is a potentially infinite *chain* of reactions, such that all completion codes are equal to 1, but the last one in the finite case:

- finite execution: $p \xrightarrow[I_0]{O_0, 1} p_1 \xrightarrow[I_1]{O_1, 1} ... \xrightarrow[I_n]{O_n, k} p_{n+1}$, with $k \neq 1$, for some $n \in \mathbb{N}$.
- infinite execution: $p \xrightarrow[I_0]{O_0, 1} p_1 \xrightarrow[I_1]{O_1, 1} ... \xrightarrow[I_n]{O_n, 1} ...$

We say that $\boldsymbol{I} = (I_0, I_1, ..., I_n)$ in the finite case and $\boldsymbol{I} = (I_n)_{n\in\mathbb{N}}$ in the infinite case is the *sequence of inputs* of the execution. Similarly, $\boldsymbol{O}$ is the *sequence of outputs*.

For example, the statement "`emit A; pause; emit B`" emits `A` and does not terminate instantly, with the residual "`nothing; emit B`" remaining to be executed. In the second and final instant of execution, `B` is emitted.

$$\texttt{emit A; pause; emit B} \xrightarrow[I_0]{\{\texttt{A}\}, 1} \texttt{nothing; emit B} \xrightarrow[I_1]{\{\texttt{B}\}, 0} \texttt{nothing}$$

We note $p \rightarrow p'$ iff there exists I and O such that $p \xrightarrow[I]{O, 1} p'$. We say that q is *reachable* from p iff $p \xrightarrow{*} q$ where $\xrightarrow{*}$ is the *reflexive transitive closure* of $\rightarrow$.

4 Logical Correctness

Depending on the statement p and inputs I, the logical behavioral semantics may define zero, one or several reactions. Moreover, a given reaction may admit more than one proof, that is to say result from more than one composition of the rules of the semantics. For example, for $I = \{\texttt{A}\}$,

	reaction	proof
`nothing`	1	1
`loop nothing end`	0	0
`signal S in present S else emit S end end`	0	0
`signal S in present S then emit S end end`	1	2
`signal S in present S then emit S else pause end end`	2	2

In particular, for "`signal S in present S then emit S end end`", the semantics defines exactly one reaction, but with two possible proofs, obtained by using either the (signal−) or the (signal+) rule:

$$\dfrac{\dfrac{\texttt{S} \notin \{\texttt{A}\} \qquad \texttt{nothing} \xrightarrow[\{\texttt{A}\}]{\emptyset, 0} \texttt{nothing}}{\texttt{present S then emit S end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 0} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then emit S end end} \xrightarrow[\{\texttt{A}\}]{\emptyset, 0} \texttt{signal S in nothing end}}$$

$$\dfrac{\dfrac{\texttt{S} \in \{\texttt{A},\texttt{S}\} \qquad \texttt{emit S} \xrightarrow[\{\texttt{A},\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}}{\texttt{present S then emit S end} \xrightarrow[\{\texttt{A},\texttt{S}\}]{\{\texttt{S}\},0} \texttt{nothing}} \qquad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in present S then emit S end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,0} \texttt{signal S in nothing end}}$$

The *internal behavior* of "`signal S in present S then emit S end end`" is not deterministic, since the local signal `S` can be both present or absent. Its *observed behavior* is nevertheless deterministic.

We expect programs to have deterministic deadlock-free executions. So, we have to discard as "incorrect" programs with no or too many possible behaviors. In this section, we formalize such a correctness criterion.

We define:

- p is *reactive* iff for all I, there exists at least one tuple (O, k, p') s.t. $p \xrightarrow[I]{O,k} p'$.
- p is *deterministic* iff for all I there is at most one tuple (O, k, p') s.t. $p \xrightarrow[I]{O,k} p'$.
- p is *strongly deterministic* iff p is deterministic and for all (I, O, k, p') the proof of $p \xrightarrow[I]{O,k} p'$ is unique if it exists.
- p is *logically correct* iff for all q reachable from p, q is reactive and deterministic.
- p is *strongly correct* iff for all q reachable from p, q is reactive and strongly deterministic.

Determinism ensures that the observed behavior of a statement is deterministic. Strong determinism guarantees that its internal behavior is deterministic, too. Reactivity combined with (strong) determinism ensures that there exists a unique reaction (with a unique proof) for this statement, whatever the inputs.

Logical correctness characterizes statements that have deterministic deadlock-free executions for any sequence of inputs. In addition, strong correctness ensures strong determinism. Strong correctness becomes a concern as soon as side effects or debugging have to be taken into account, as both may expose the internal behavior of a program. Of course, strong correctness implies logical correctness.

5 Deterministic Semantics

The logical behavioral semantics provides a very compact, structural formalization of the behavior of Esterel programs, which makes formal reasoning about the language tractable. Moreover, it defines reactivity and determinism, which are the agreed minimal correctness criteria for Esterel programs.

However, working with these criteria can be tedious. While, reactivity may be attested with a simple proof tree, establishing (strong-)determinism is more complex and formally requires a proof about proof trees (proof of uniqueness).

Moreover, defining first many (proofs of) reactions for non-(strongly)-deterministic statements, which we then discard because there are too many, seems utterly inefficient.

Therefore, we propose to rewrite the rules for local signal declarations:

$$\frac{p \xrightarrow[I\cup\{S\}]{O,k} p' \qquad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O\backslash\{S\},k} \texttt{signal } S \texttt{ in } p' \texttt{ end}} \qquad \text{(signal+)}$$

$$\frac{p \xrightarrow[I\backslash\{S\}]{O,k} p' \qquad S \notin O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xrightarrow[I]{O,k} \texttt{signal } S \texttt{ in } p' \texttt{ end}} \qquad \text{(signal−)}$$

as the following (where k^+, k^-, etc. are nothing but convenient names):

$$\frac{p \xmapsto[I\backslash\{S\}]{O^-,k^-} p^- \qquad S \in O^- \qquad p \xmapsto[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xmapsto[I]{O^+\backslash\{S\},k^+} \texttt{signal } S \texttt{ in } p^+ \texttt{ end}} \qquad \text{(signal++)}$$

$$\frac{p \xmapsto[I\backslash\{S\}]{O^-,k^-} p^- \qquad S \notin O^- \qquad p \xmapsto[I\cup\{S\}]{O^+,k^+} p^+ \qquad S \notin O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xmapsto[I]{O^-,k^-} \texttt{signal } S \texttt{ in } p^- \texttt{ end}} \qquad \text{(signal−−)}$$

We call the resulting semantics the *deterministic semantics*, and denote the corresponding reactions by the transition symbol "$\mapsto$".

Intuitively, it consists in enforcing in each signal rule that the other one does not apply, without introducing negative premises [12] such as:

$$S, p, I, O, k, \text{ and } p' \text{ are } \textit{not} \text{ such that } p \xrightarrow[I\cup\{S\}]{O,k} p' \text{ and } S \in O$$

Rather than negating the whole precondition, we only swap the binary decision $S \in O$ for $S \notin O$, and vice versa. In the logical behavioral semantics, we had:

- (signal+): if S is supposed present in p then it is emitted by p.
- (signal−): if S is supposed absent in p then it is not emitted by p.

In our deterministic semantics, the rules for the `signal` construct become:

- (signal++):
  · if S is supposed present in p then it is emitted.
  · if S is supposed absent in p then it is *still* emitted.
- (signal−−):
  · if S is supposed absent in p then it is not emitted.
  · if S is supposed present in p then it is not emitted *either*.

5.1 Examples

For example, the deterministic semantics produces the same reactions as the logical behavioral semantics, in the following two cases (cf. Section 3):

$$\frac{\texttt{pause} \xmapsto[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing} \quad \texttt{S} \notin \emptyset \qquad \texttt{pause} \xmapsto[\{\texttt{A,S}\}]{\emptyset,1} \texttt{nothing} \quad \texttt{S} \notin \emptyset}{\texttt{signal S in pause end} \xmapsto[\{\texttt{A}\}]{\emptyset,1} \texttt{signal S in nothing end}}$$

$$\frac{\texttt{emit S} \xmapsto[\{\texttt{A}\}]{\{\texttt{S}\},0} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\} \qquad \texttt{emit S} \xmapsto[\{\texttt{A,S}\}]{\{\texttt{S}\},0} \texttt{nothing} \quad \texttt{S} \in \{\texttt{S}\}}{\texttt{signal S in emit S end} \xmapsto[\{\texttt{A}\}]{\emptyset,0} \texttt{signal S in nothing end}}$$

The deterministic semantics defines no reaction for:

- the non-reactive statement:
  “`signal S in present S else emit S end end`”
- the non-deterministic statement:
  “`signal S in present S then emit S else pause end end`”
- the non-strongly-deterministic statement:
  “`signal S in present S then emit S end end`”

5.2 Determinism

The new semantics is *globally deterministic*:

Theorem 5.1 *For all p and I, there exists at most one (O,k,p') s.t. $p \xmapsto[I]{O,k} p'$.*

Theorem 5.2 *For all p, I, O, k, p', the proof of $p \xmapsto[I]{O,k} p'$ is unique if it exists.*

Proof. Simple structural induction on p. □

There is no need to count proofs and reactions in the deterministic semantics.

5.3 Properness

Since, the uniqueness of proofs and reactions is ensured, we shall say that the statement p is correct with respect to the deterministic semantics, i.e. *proper*, iff the deterministic semantics defines at least one reaction at any stage of the execution of p for any sequence of inputs. Formally, we define:

- p is *initially proper* iff for all I, there exists (O,k,p') such that $p \xmapsto[I]{O,k} p'$.
- $p \mapsto p'$ iff there exists I and O such that $p \xmapsto[I]{O,1} p'$.
- $\overset{*}{\mapsto}$ is the reflexive transitive closure of $\mapsto$.
- p is *proper* iff for all q such that $p \overset{*}{\mapsto} q$, q is initially proper.

6 Comparison

We now precisely relate the logical behavioral and the deterministic semantics.

6.1 Properness implies strong correctness

Theorem 6.1 *If* $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ *then* $p \xrightarrow[I]{O,k} p'$.

Theorem 6.2 *If* $p \overset{O_0,k_0}{\underset{I}{\longmapsto}} p'_0$ *and* $p \xrightarrow[I]{O_1,k_1} p'_1$ *then* $O_0 = O_1$, $k_0 = k_1$, $p'_0 = p'_1$.

Theorem 6.3 *If* $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ *then the proof of* $p \xrightarrow[I]{O,k} p$ *is unique.*

Proof. cf. Appendix A. □

By writing

$$p \overset{O,k}{\underset{I}{\longmapsto}} p'$$

we not only express that p *may* react to inputs I, with outputs O, completion code k, and residual p' in the deterministic semantics, thus in the logical behavioral semantics as well (Th. 6.1), but also that it *must* react this way in both semantics (Th. 5.1 and 6.2), and that its internal behavior is deterministic (Th. 5.2 and 6.3). As a consequence,

Corollary 6.4 *If* p *is proper then* p *is strongly correct.*

6.2 Strong correctness does not imply properness

Reciprocally, a strongly correct statement is not necessarily proper, as reactivity combined with strong determinism does not imply initial properness. Let's consider two examples:

-
  ```
  signal S in
    present S then loop nothing end end
  end
  ```

  For all inputs I, the logical behavioral semantics defines the following unique proof tree for this program:

  $$\dfrac{\dfrac{\texttt{S} \notin I\backslash\{\texttt{S}\} \qquad \texttt{nothing} \xrightarrow[I\backslash\{\texttt{S}\}]{\emptyset,0} \texttt{nothing}}{\texttt{present S then loop nothing end end} \xrightarrow[I\backslash\{\texttt{S}\}]{\emptyset,0} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then ... end end} \xrightarrow[I]{\emptyset,0} \texttt{signal S in nothing end}}$$

  The deterministic semantics however defines no reaction for this statement, whatever I. Neither the rule (signal++), nor the rule (signal−−) applies, as "`loop nothing end`", thus "`present S then loop nothing end end`" are not initially proper.

-
  ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body "`signal S in present S then emit S else pause end end`" of the loop may react in two possible ways in the logical behavioral semantics, whatever I, with respective completion codes 0 and 1:

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset, 0} \ldots$$

  $$\texttt{signal S in present S then emit S else pause end end} \xrightarrow[I]{\emptyset, 1} \ldots$$

  Since exactly one of these two reactions admits a non-zero completion code, the whole loop statement is both reactive and strongly deterministic. On the other hand, the deterministic semantics defines no reaction for the body, hence no reaction for the loop.

6.3 Strongly correct non-proper statements

In the logical behavioral semantics, non-determinism may compensate for non-reactivity, or the other way around, so that a piece of incorrect code may be embedded into a strongly correct program. More precisely,

Theorem 6.5 *If p is reactive and strongly deterministic but not initially proper then there exists a subterm q of p such that q is not reactive or not strongly deterministic.*

Proof. cf. Appendix B. □

Intuitively, q behaves *well* in p only because of its context of occurrence, which constrains the execution of q from the outside, making sure the non-reactive or non-strongly-deterministic behaviors of q are never triggered. In other words, q could be simplified while preserving the behavior of p. Let's consider again our two examples in this new light:

-
  ```
  signal S in
    present S then loop nothing end end
  end
  ```

  The subterm "`present S then loop nothing end end`" is not reactive because of its `then` branch, but never used with `S` present. Therefore, it can be replaced by its implicit `else` branch, that is to say `nothing`, leading to the equivalent[3] program "`signal S in nothing end`", which is proper.

[3] Technically, they are strongly bisimilar [15] w.r.t. the logical behavioral semantics.

-
  ```
  loop
    signal S in
      present S then emit S else pause end
    end
  end
  ```

  The body “`signal S in present S then emit S else pause end end`” is not deterministic, but the enclosing loop enforces `S` to be absent. Again, the “`present S then emit S else pause end`” statement can simplified. The resulting program “`loop signal S in pause end end`” is proper and logically equivalent.

Therefore, there is something *wrong* with these programs, even if neither logical correctness nor strong correctness are sensitive to it. In any case, they are intricate constructions with no practical purpose.

7 Constructive Semantics

The constructive semantics of Esterel [4] ensures that behaviors can be effectively computed, that is to say *causally* computed. For instance, although the following program is logically correct, even strongly correct as `S` can only be present, it is rejected by the constructive semantics:

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

Intuitively, this program is not *constructive* because the status of `S` must be “guessed” prior to its emission. Such an argument however is not relevant to the deterministic semantics, which considers this program to be proper.

On the other hand, the deterministic semantics sometimes rejects constructive programs, such as:

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

Since `S` cannot be emitted – there is no “`emit S`” statement – the `then` branch of the `present` statement is never “visited” by the constructive semantics. As a result, this program is constructive. On the other hand, the deterministic semantics does explore this branch, so that the program is not proper.

Executions in the constructive semantics being defined by a (complex) monotonous information propagation process, there is at most one reaction defined for each program and each set of inputs. In other words, the constructive semantics is globally deterministic in the sense of Section 5.

In summary, even if both semantics are globally deterministic, the reasons for this property are very different, and the corresponding correctness criteria are unrelated. They both make sense and could be combined.

8 Conclusion

In contrast with the logical behavioral semantics of Esterel, the deterministic semantics we introduce in this work, defines at most one execution for all programs and all inputs. In particular, if the deterministic semantics defines the execution of a program, then this execution is unique, thus correct.

Importantly, the deterministic semantics does not change the semantics of "reasonable" programs. If the deterministic semantics of a program is defined then it matches its logical behavioral semantics. Reciprocally, if the deterministic semantics of a program is not defined then the program or some subterm of the program is incorrect w.r.t. the logical behavioral semantics.

Moreover, our new semantics achieves determinism at a much lower cost than the constructive semantics of Berry. As a result, we claim that the deterministic semantics provides a much better starting point for formal reasoning about Esterel programs than both the logical behavioral semantics and the constructive semantics.

A Proof of Theorems 6.1 to 6.3

By structural induction on p, we prove that if $p \xmapsto[I]{O,k} p'$ then:

- $p \xrightarrow[I]{O,k} p'$ with a unique proof;
- if $p \xrightarrow[I]{O_0,k_0} p'_0$ then $O = O_0$, $k = k_0$, $p' = p'_0$.

Proof. Let's consider the case $p =$ "`signal` S `in` q `end`", and choose a set I. By hypothesis, there exists (O, k, p') such that:

$$\texttt{signal } S \texttt{ in } q \texttt{ end} \xmapsto[I]{O,k} p'$$

Either rule (signal++) or (signal−−) has to be used to define this reaction. Let's for instance consider the case (signal−−). The case (signal++) is similar. There exists $(O^-, k^-, q^-, O^+, k^+, q^+)$ such that:

- $q \xmapsto[I\backslash\{S\}]{O^-,k^-} q^-$ and $q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+$
- $S \notin O^-, S \notin O^+, O = O^-, k = k^-, p' =$ "`signal` S `in` q^- `end`".

so that the following deduction holds in the deterministic semantics:

$$\frac{q \xmapsto[I\backslash\{S\}]{O^-,k^-} q^- \quad S \notin O^- \quad q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+ \quad S \notin O^+}{p = \texttt{signal } S \texttt{ in } q \texttt{ end} \xmapsto[I]{O,k} \texttt{signal } S \texttt{ in } q^- \texttt{ end} = p'}$$

By induction hypothesis:

- $q \xrightarrow[I\backslash\{S\}]{O^-,k^-} q^-$ with a unique proof.
- if $q \xrightarrow[I\backslash\{S\}]{O_0^-,k_0^-} q_0^-$ then $O^- = O_0^-$, $k^- = k_0^-$, $q^- = q_0^-$.
- $q \xrightarrow[I\cup\{S\}]{O^+,k^+} q^+$ with a unique proof.
- if $q \xrightarrow[I\cup\{S\}]{O_0^+,k_0^+} q_0^+$ then $O^+ = O_0^+$, $k^+ = k_0^+$, $q^+ = q_0^+$.

On the one hand, as $S \notin O^+$, no reaction for p can be defined using (signal+). On the other hand, by rule (signal−),

- $p \xrightarrow[I]{O^-,k^-}$ `signal` S `in` q^- `end` with a unique proof.
- if $p \xrightarrow[I]{O_0,k_0} p_0'$ then $O_0 = O^-$, $k_0 = k^-$, $p_0' =$ `signal` S `in` q^- `end`.

And similarly for all other cases. □

B Proof of Theorem 6.5

By structural induction on p, we prove that if p and all its subterms are reactive and strongly deterministic then p is initially proper.

Proof. Let's consider the case $p =$ "`signal` S `in` q `end`", and choose a set I. By hypothesis, q and all its subterms are reactive and strongly deterministic. By induction hypothesis, q is initially proper. Thus, there exists $(k^-, O^-, q^-, k^+, O^+, q^+)$ such that:

$$q \xmapsto[I\backslash\{S\}]{O^-,k^-} q^- \quad \text{and} \quad q \xmapsto[I\cup\{S\}]{O^+,k^+} q^+$$

There are four cases:

- $S \in O^-$, $S \in O^+$, then by rule (signal++), $p \xmapsto[I]{O^+\backslash\{S\},k^+}$ `signal` S `in` q^+ `end`.
- $S \notin O^-$, $S \notin O^+$, then by rule (signal−−), $p \xmapsto[I]{O^-,k^-}$ `signal` S `in` q^- `end`.
- $S \in O^+$, $S \notin O^-$:
  · by rule (signal+), $p \xrightarrow[I]{O^+\backslash\{S\},k^+}$ `signal` S `in` q^+ `end`
  · by rule (signal−), $p \xrightarrow[I]{O^-,k^-}$ `signal` S `in` q^- `end`
  Therefore, p is not strongly deterministic. Contradiction.
- $S \notin O^+$, $S \in O^-$, then neither (signal+) nor (signal−) is applicable. Therefore, p is not reactive. Contradiction.

Similarly, in all other cases, the deterministic semantics defines a reaction for p, whatever I. As a consequence, p is initially proper. □

References

[1] *The Esterel v5_92 Compiler*, http://www-sop.inria.fr/esterel.org/.

[2] Benveniste, A., P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, *The Synchronous Languages Twelve Years Later*, Proceedings of the IEEE **91** (2003), pp. 64–83.

[3] Berry, G., *The semantics of pure Esterel*, in: M. Broy, editor, *Program Design Calculi*, Series F: Computer and System Sciences **118** (1993), pp. 361–409.

[4] Berry, G., *The constructive semantics of pure Esterel. Draft version 3* (1999), http://www-sop.inria.fr/esterel.org/.

[5] Berry, G., *The Esterel v5 language primer. Version 5_91* (2000), http://www-sop.inria.fr/esterel.org/.

[6] Berry, G. and L. Cosserat, *The synchronous programming language Esterel and its mathematical semantics*, LNCS **197** (1984), pp. 389–448.

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

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

[9] Edwards, S., "Languages for Digital Embedded Systems," Kluwer, 2000.

[10] Edwards, S. A., V. Kapadia and M. Halas, *Compiling Esterel into Static Discrete-Event Code*, in: *Synchronous Languages Applications and Programming*, 2004.

[11] Gonthier, G., "Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel," Thèse d'informatique, Université d'Orsay, Paris, France (1988).

[12] Groote, J. F., *Transition system specifications with negative premises*, in: *CONCUR'90*, Springer, 1990 pp. 332–341.

[13] Halbwachs, N., "Synchronous Programming of Reactive Systems," Kluwer, 1993.

[14] Malik, S., *Analysis of Cyclic Combinational Circuits*, in: *IEEE/ACM International Conference on CAD*, ACM/IEEE (1993), pp. 618–627.

[15] Milner, R., "Communication and Concurrency," Series in Computer Science, Prentice Hall, 1989.

[16] Plotkin, G., *A structural approach to operational semantics*, Report DAIMI FN-19, Aarhus University (1981), to be published in the JLAP special issue on SOS, 2004.

[17] Tardieu, O. and R. de Simone, *Instantaneous termination in pure Esterel*, in: *SAS'03*, LNCS **2694**, 2003, pp. 91–108.