A Deterministic Logical Semantics for Pure Esterel

Olivier Tardieu
Department of Computer Science, Columbia University, New York

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 deadlock-free executions as it may define zero, one, or many possible behaviors for a given program and input sequence. Since non-deterministic programs have to be rejected by compilers, this means it defines behaviors for incorrect programs, which is awkward. 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 build and thoroughly analyze a new deterministic semantics for Esterel that retains the simplicity of the logical behavioral semantics from which it derives. It defines at most one behavior per program and input sequence. We further extend this semantics with the ability to deal with errors so that incorrect programs are no longer (negatively) characterized by a lack of behavior but (positively) by the existence of an incorrect behavior. In our view, this new semantics, with or without explicit errors, provides a better framework for formal and automated reasoning about Esterel programs.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Theory

Additional Key Words and Phrases: synchronous languages, structural operational semantics

1. INTRODUCTION

Today more and more embedded computer systems are being used for safety-critical applications, resulting in an ever-increasing demand for reliable software and hardware design methods and tools.

The paradigm of synchrony [Benveniste and Berry 1991; Benveniste et al. 2003] has emerged as a simple and mathematically-sound foundation for the design of systems expressing a high level of concurrency, while maintaining determinism and predictable, reproducible system behaviors. Time is divided into discrete instants. Concurrent threads run in lockstep (to one or several clocks). Communications are instantaneous. Several programming languages have adopted the synchronous approach, in particular the three French pioneers: Esterel [Boussinot and de Simone 1991], Lustre [Halbwachs et al. 1991], and Signal [Le Guernic et al. 1991].

In contrast with traditional thread-based or event-based concurrency models that embed no precise or deterministic scheduling policy in the design formalism, synchronous languages semantics take care of all scheduling decisions. As a consequence, programs are guaranteed to behave the same whatever the execution platform, which is more uncommon than thought, but obviously very convenient and sometimes mandatory for the design of safety-critical applications.

The solid mathematical framework common to all synchronous languages facilitates validation and certification, as it enables formal reasoning about programs

email: olivier.tardieu@mines.org.

and implementations, including formal verification and exhaustive testing. In particular, code generation can be provably correct. Therefore, high-level synthesis tools should provide correct-by-construction design flows, thereby avoiding the all too often occurring problem of consistency between simulation and synthesis semantics found in neighboring formalisms such as HDLs like VHDL and Verilog. In practice, due to the many intricacies of an optimizing compiler, certifying such tools by means of a mathematical proof remains a scientific and technical challenge.

To start with, programming language semantics should not only precisely specify program behaviors, but they should also be carefully engineered to ease proofs. In the current paper, we undertake such a task for the Esterel language.

1.1 Pure Esterel

Esterel [Berry and Cosserat 1984; Berry and Gonthier 1992; Berry 1999; 2000a; 2000b] is a high-level synchronous programming language dedicated to the specification of reactive systems [Edwards 2000; Halbwachs 1993]. Its syntax is imperative and fit for the design of safety-critical embedded systems where control-handling aspects prevail. In addition to traditional control-flow operators, it defines suspension and preemption mechanisms that are compatible with concurrency and preserve determinism [Berry 1993]. Both hardware synthesis and efficient simulation are supported [INRIA et al. 2000; Potop-Butucaru 2002; Edwards et al. 2004].

Pure Esterel is the subset of the full Esterel language where data variables and data-handling primitives are omitted. In this work, we concentrate on this subset. We shall comment on that later.

Pure Esterel deals with pure signals only. Pure signals have a Boolean status, which obeys the *signal coherence law.* In each instant—time is divided into discrete instants—a signal is absent by default and present if emitted in the instant. Both absence and presence are instantly broadcast and made available to all threads of execution.

This perfect synchrony hypothesis may result in causality cycles [Berry 1999; Malik 1993], as for example in the parallel composition:

```
signal A, B in
  present A then emit B end || present B then emit A end
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.

For this reason, this program is said to be *non-deterministic.*

Similarly, there exist *non-reactive* programs with no possible execution, for example:

```
signal A, B in
  present A then emit B end || present B else emit A end
end
```

In Esterel, we want programs to have deadlock-free, deterministic executions. Thus, such non-reactive and non-deterministic programs have to be rejected.

1.2 Existing Formal Semantics

Two main semantics have been formalized for pure Esterel:

– The *logical behavioral semantics* [Berry and Gonthier 1992] simply formalizes the signal coherence law using SOS rules [Plotkin 1981]. For a given input sequence, it defines no execution for a non-reactive program, and several distinct executions for a non-deterministic program[1].

– The *constructive semantics* [Berry 1999] is inspired from digital circuits and based on a three-valued logic. It only reproduces a subset of the executions allowed by the logical behavioral semantics. It introduces the idea of *causal dependencies* and ensures that these executions can be *causally* computed: the status of a signal should be certain before being used (cf. Section 8). As a result, it defines no execution for non-reactive and non-deterministic programs, such as the two previous examples.

These two semantics handle non-determinism in opposite manners:

– 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. While reactivity may be shown with a simple proof tree (using the deduction rules of the semantics), establishing determinism is more complex and requires a meta proof: a proof about proof trees (proof of uniqueness), which is harder to formalize within a proof assistant.
Moreover, non-determinism sometimes compensates for non-reactivity: too many behaviors on the one hand, too few on the other hand, may result in exactly one behavior in the end, so that a correct program possibly contains "bad" code.

– The constructive semantics is intrinsically deterministic and thus only defines correct executions. But the formulation of its rules is rather involved and requires the definition of subtle auxiliary predicates. As a consequence, it makes formal reasoning about Esterel programs much more difficult, and should be avoided when causality is not an issue.

1.3 Toward a Deterministic Semantics

Neither semantics is truly convenient for dealing with non-determinism. Therefore, we introduce in this work an alternative semantics that we derive from the logical behavioral semantics. It retains the simple formalism of the logical behavioral semantics while defining at most one execution per program and input sequence.

As an additional benefit, we shall observe that errors cannot cancel one another in this new semantics. But this is also a minor drawback as it means we have get accustomed to a new definition of program correctness, even if better.

It should be understood that the goal of this deterministic semantics is not to replace the constructive semantics—the reference semantics when it comes to implementation—but to provide a formal framework for proving properties about Esterel programs (independently from causality), leading to more compact, better structured, and more manageable proofs than what can be achieved with both the logical behavioral and the constructive semantics.

[1]In general, determinism and reactivity depend on the input sequence (cf. Section 4).

$p, q ::=$ `nothing`	does nothing and terminates instantly
`pause`	suspends the execution for one instant
p`;` q	runs p, then q if/when p terminates
p `||` q	runs p in parallel with q
`loop` p `end`	repeats p forever
`signal` S `in` p `end`	declares signal S in p
`emit` S	emits signal S
`present` S `then` p `else` q `end`	runs p if S is present in this instant, q otherwise
`try` p `end`	runs p
`exit` n	exits $n \geq 1$ enclosing "`try` ... `end`" blocks

Fig. 1. Primitive Pure Esterel Constructs

It should also be understood that the benefits of this deterministic semantics do not extend beyond pure Esterel. Full Esterel enables data-dependent choices. The deterministic semantics is not meant to be extended to handle such choices, that is to say ensure they have a deterministic outcome. But it may be used to establish that the program behavior is deterministic assuming data-dependent choices are.

1.4 Overview of the Paper

In Section 2, we describe a kernel pure Esterel language. We formalize its logical behavioral semantics in Section 3 and define reactivity and determinism in Section 4. We introduce our deterministic semantics in Section 5. In Section 6, we compare the two semantics. In Section 7, we consider turning deadlocks into explicit errors, that is to say replace an absence of behavior by the existence of an error. We discuss the constructive semantics in Section 8 and conclude in Section 9.

2. SYNTAX AND INTUITIVE SEMANTICS

Without loss of generality, we focus on a kernel language inspired by Berry [1999]. 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*. Signals are lexically scoped and declared using the construct "`signal` S `in` ... `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*. Except for the `pause` instruction, primitive statements execute in zero time[2].

[2]This is an ideal view of the fact that reactions are supposed to converge and terminate before the next clock tick occurs, so that no overlap is possible. Of course, just as gates in digital circuits have delays, instructions in Esterel take physical time to execute. But at the formal semantics level, only logical time matters, that is to say instructions behave as if they are instantaneous. As a result, the main goal of an Esterel compiler is to ensure by a proper scheduling of instructions in each instant that the implemented behavior complies with the idealized semantics.

When the clock ticks, a reaction occurs that 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 until 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.

The program "`emit A; pause; emit B; emit C; pause; emit D`" emits the signal `A` in the first instant of its execution, emits `B` and `C` in the second instant, and finally emits `D` and terminates in the third instant. It takes three instants to complete, that is to say its execution consists of 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`" are equivalent.

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, emits `A` and `D` in the second instant, and finally emits `B` and `E` and terminates in the third instant.

2.3 Weak Preemption

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

```
try
  emit A; pause; emit B; exit 1; emit C
end;
emit D
```

emits `A` in the first instant, then `B` and `D` and terminates in the second instant. The statement "`emit C`" is never executed.

An `exit` in a parallel context causes all parallel branches to terminate instantly. In the next example, `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. The execution of "`exit 1`" does not prevent the simultaneous execution of "`emit F`". This is *weak* preemption.

```
try
  emit A; pause; emit B; exit 1; emit C
||
  emit E; pause; emit F; pause; emit G
end;
emit D
```

In case of simultaneous `exit`s and nested "`try` ... `end`" blocks, the control is transfered back to the outermost "`try` ... `end`" block targeted by the `exit`s. For instance `A` is not emitted whereas `B` is emitted in:

```
try
  try
    try
      exit 2 || exit 1
    end;
    emit A
  end;
  emit B
end
```

2.4 Suspension and Strong Preemption

Esterel also defines constructs for suspension and strong preemption. For simplicity, we do not consider them in this paper. Extending our results to such constructs is straightforward.

2.5 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`, `try` and `exit` statements, as for instance in the kernel expansion of "`await` S":

```
try
  loop
    pause;
    present S then exit 1 end
  end
end
```

Loop bodies must not be *instantaneous* [Tardieu and de Simone 2003]. 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.

2.6 Signals

The instruction "`signal` S `in` p `end`" declares the *local* signal S in p. The free signals of a block are said to be *interface* signals for this block. Full Esterel defines *modules* and distinguishes *input*, *output* and *inputoutput* signals. In this work however, we have no need for such definitions, interface signals being both the inputs and outputs of a block.

Intuitively, a block may *read* from a signal S through `present` constructs, may *write* to a signal S via `emit` instructions. The read and written values of a signal S are related iff there exists an enclosing local declaration of S, as illustrated in Figure 2.

Formally, 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

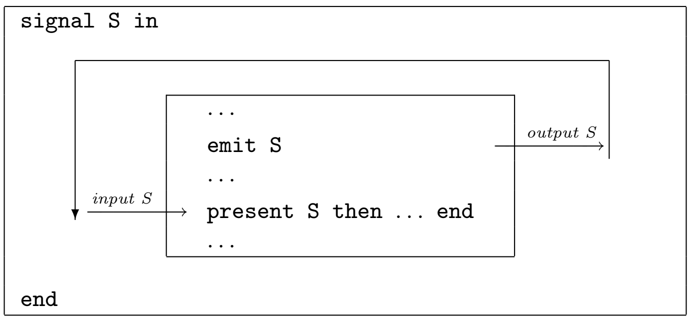

Fig. 2. The Semantics of Signals

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. Presence and absence obey the following rules:

– A local signal is present iff it is emitted.
– An interface signal is present iff it is provided by the *environment*[3].

Interface signal may be both absent and emitted, as the feedback loop of Figure 2 is missing for interface signals. Let's consider a few examples:

–
```
signal S in
  emit S;
  pause;
  present S then emit O end
end
```
The signal `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.

–
```
signal S in
  present S then emit O end
||
  emit S
end
```
Both `S` and `O` are emitted. The local signal `S` is present.

–
```
emit S;
present S then emit O end
```
In the absence of a declaration of `S`, despite "`emit S`", the status of `S` depends on the environment, hence `O` is emitted iff `S` is provided by the environment.

[3]Alternatively, we may state that an interface signal is present iff provided by the environment *or locally emitted*, in a way similar to [Tardieu and de Simone 2003]. This choice, however, would lead to a more intricate deterministic semantics, as we would have to later disentangle the local and global communication parts of such interface signals. Therefore, we prefer to stick here to the more traditional but maybe less intuitive rule of Berry [1999]: an interface signal is present iff provided by the environment, which makes it possible for an interface signal to be absent even if locally emitted.

3. LOGICAL BEHAVIORAL SEMANTICS

The *logical behavioral semantics* [Berry and Gonthier 1992; Berry 1999; Gonthier 1988] 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:

– I is the set of *present signals*,

– O is the set of *emitted signals*,

– the integer k is the *completion code* of the reaction,

– the statement p' is the *residual* of the reaction.

3.1 Present and Emitted Signals

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

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

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` $\xrightarrow[I]{O,\,k} q'$.

3.2 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. For example,

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

– If $k \neq 1$ then this reaction ends the execution of p and p' does not matter:
  – $k = 0$ if the execution completes *normally.* For example,

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

  – $k = n + 1$ if n enclosing "`try` ... `end`" blocks have to be *exited.* In particular,

$$\texttt{exit}\ n \xrightarrow[I]{\emptyset,\,n+1} \texttt{nothing}$$

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

$$\texttt{exit 3 || exit 5 end} \xrightarrow[I]{\emptyset,\,6} \texttt{nothing || nothing}$$

3.3 Deduction Rules

The logical behavioral semantics is generated by the fourteen deduction rules of Figure 3. As usual, variables are universally quantified. Each rule is of the form:

$$\frac{\mathit{Hypothesis}(1) \quad \mathit{Hypothesis}(2) \quad \dots \quad \mathit{Hypothesis}(n)}{\mathit{Conclusion}}$$

meaning that:

$$\forall\, \mathit{variables} : \bigwedge_{i=1}^{n} \mathit{Hypothesis}(i) \Rightarrow \mathit{Conclusion}$$

In other words, the logical behavioral semantics consists in the reactions that can be proved using these fourteen deduction rules.

In the sequel, we shall omit the "*true*" leaves of the proof trees.

3.4 Execution

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

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

– infinite execution: $p \xrightarrow[I_0]{O_0,\,1} p_1 \xrightarrow[I_1]{O_1,\,1} \dots \xrightarrow[I_n]{O_n,\,1} \dots$

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

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

3.5 Signal Coherence Law

The signal coherence law—a local signal is present iff emitted—is enforced by the (signal+) and (signal−) rules. If rule (signal+) applies to "`signal` S `in` p `end`" then S both is emitted by p and present in p. In contrast, if rule (signal−) applies to "`signal` S `in` p `end`" then S is neither emitted by p nor present in p.

For instance, for the statement "`signal S in pause end`" with inputs $I = \{\texttt{A},\texttt{S}\}$, we obtain the behavior:

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

The input set for `pause` is $\{\texttt{A},\texttt{S}\} \setminus \{\texttt{S}\} = \{\texttt{A}\}$. Indeed, `pause` does not emit `S` and the local declaration of `S` hides the interface signal `S`, thus `S` is absent.

$$\frac{\mathit{true}}{\texttt{nothing} \xrightarrow[I]{\emptyset, 0} \texttt{nothing}} \qquad \text{(nothing)}$$

$$\frac{\mathit{true}}{\texttt{pause} \xrightarrow[I]{\emptyset, 1} \texttt{nothing}} \qquad \text{(pause)}$$

$$\frac{\mathit{true}}{\texttt{exit}\ n \xrightarrow[I]{\emptyset, n+1} \texttt{nothing}} \qquad \text{(exit)}$$

$$\frac{\mathit{true}}{\texttt{emit}\ S \xrightarrow[I]{\{S\}, 0} \texttt{nothing}} \qquad \text{(emit)}$$

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

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

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

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

$$\frac{p \xrightarrow[I]{O, 2} p'}{\texttt{try}\ p\ \texttt{end} \xrightarrow[I]{O, 0} \texttt{nothing}} \qquad \text{(try-catch)}$$

$$\boxed{\downarrow k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ k-1 & \text{if } k > 2 \end{cases}} \qquad \frac{p \xrightarrow[I]{O, k} p' \qquad k \neq 2}{\texttt{try}\ p\ \texttt{end} \xrightarrow[I]{O, \downarrow k} \texttt{try}\ p'\ \texttt{end}} \qquad \text{(try-through)}$$

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

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

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

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

Fig. 3. Berry and Gonthier's Logical Behavioral Semantics

In contrast, in "`signal S in emit S end`" with inputs $I = \{\texttt{A}\}$, the local signal `S` is emitted:

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

The local signal `S` is present: the input set for the inner statement is $\{\texttt{A}\} \cup \{\texttt{S}\}$. Moreover, `S` does not escape its scope of definition: the output set for the whole statement remains empty even if `S` is locally emitted.

4. LOGICAL CORRECTNESS

Depending on the program p and inputs I, the logical behavioral semantics may define zero, one or several reactions. For example, for inputs $I = \{\texttt{A}\}$ and program "`signal S in present S then emit S else pause end end`",

– using rule (signal−) we can prove the reaction:

$$\frac{\dfrac{\texttt{S} \notin \{\texttt{A}\} \qquad \texttt{pause} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing}}{\texttt{present S then emit S else pause end} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{nothing}} \qquad \texttt{S} \notin \emptyset}{\texttt{signal S in present S then emit S else pause end end} \xrightarrow[\{\texttt{A}\}]{\emptyset,1} \texttt{signal S in nothing end}}$$

– using rule (signal+) we can prove the reaction:

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

The behavior of this program is not *deterministic.*

Moreover, one reaction may admit more than one proof tree. For example, for program "`signal S in present S then emit S end end`" and inputs $I = \{\texttt{A}\}$, the semantics defines exactly one reaction, but there are two different proofs:

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

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

The *internal behavior* of this program is not deterministic since the local signal `S` can be either present or absent. Its *observed behavior* is nevertheless deterministic.

	reactions	proofs
`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

Fig. 4. Numbers of Reactions and Proofs

More examples are gathered in Figure 4. Due to the lack of interface signals in these examples, the numbers do not depend on the input set I.

We expect programs to have deterministic deadlock-free executions. So, we have to discard programs with no or too many possible behaviors. We now 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 are.

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. The various inclusions between the program classes we define are summarized in Figure 5 using a Venn Diagram notation.

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 possible. Moreover, it allows the definition of reactivity and determinism, which are the agreed minimal correctness criteria for Esterel programs.

However, working with these criteria can be tedious as they are not part of the semantics, but defined on top of it. In particular, while reactivity may be attested with a simple proof tree, establishing (strong-)determinism requires a meta proof, that is to say a proof about a set of proof trees.

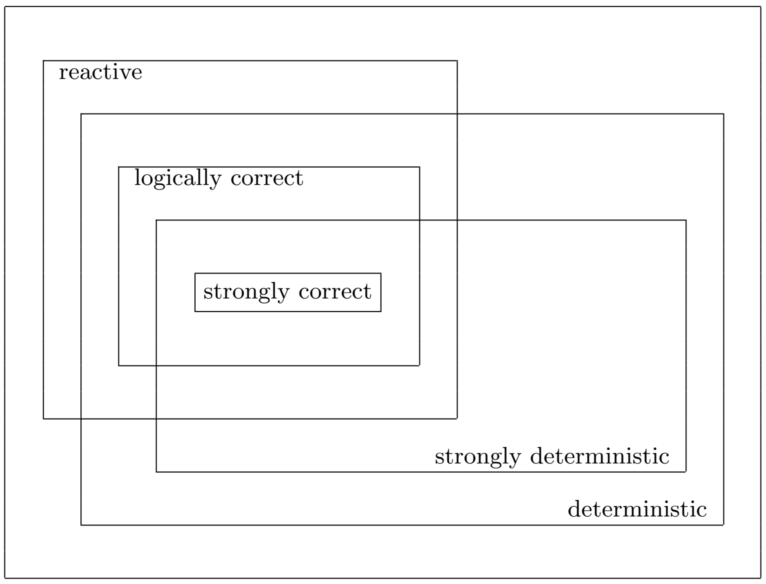

Fig. 5. Program Classes

5.1 Revised Rules

We propose to rewrite the rules for local signal declarations:

- (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.

into the following two rules:

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

Our goal with these definitions is to enforce in each signal rule that the other one does not apply, without introducing negative premises [Groote 1993] such as:

$$S, p, I, O_0, k_0, \text{ and } p'_0 \text{ are } \boxed{\text{not}} \text{ such that } p \xrightarrow[I \cup \{S\}]{O_0, k_0} p'_0 \text{ and } S \in O_0$$

Instead, we consider the test:

$$S, p, I, O_0, k_0, \text{ and } p'_0 \text{ are such that } p \xrightarrow[I \cup \{S\}]{O_0, k_0} p'_0 \text{ and } \boxed{\text{not}}\ S \in O_0$$

that is to say we swap the binary decision $S \in O_0$ for $S \notin O_0$.

Rules (signal++) and (signal−−) are formally defined in Figure 6. The additions to the initial (signal+) and (signal−) rules have been boxed for readability. We call the resulting semantics the *deterministic semantics* and denote the corresponding reactions by the transition symbol “$\mapsto$”.

$$\frac{\boxed{p \xmapsto[I\backslash\{S\}]{O_0,\,k_0} p'_0 \quad S \in O_0} \qquad p \xmapsto[I\cup\{S\}]{O,\,k} p' \quad S \in O}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xmapsto[I]{O\backslash\{S\},\,k} \texttt{signal } S \texttt{ in } p' \texttt{ end}} \quad \text{(signal++)}$$

$$\frac{p \xmapsto[I\backslash\{S\}]{O,\,k} p' \quad S \notin O \qquad \boxed{p \xmapsto[I\cup\{S\}]{O_0,\,k_0} p'_0 \quad S \notin O_0}}{\texttt{signal } S \texttt{ in } p \texttt{ end} \xmapsto[I]{O,\,k} \texttt{signal } S \texttt{ in } p' \texttt{ end}} \quad \text{(signal−−)}$$

Fig. 6. Deterministic Semantics: the `signal` Construct

5.2 Examples

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

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

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

Going back to the examples of Figure 4, the deterministic semantics defines no reaction for:

– the non-reactive statements:
`loop nothing end` and `signal S in present S else emit S end end`
– the non-strongly-deterministic statement:
`signal S in present S then emit S end end`
– the non-deterministic statement:
`signal S in present S then emit S else pause end end`

5.3 Determinism

The purpose of the deterministic semantics is to move the meta-reasoning required for establishing determinism in the logical behavioral semantics to the rules of the semantics themselves. Indeed, the new semantics is *globally deterministic*:

THEOREM 1. *For all p and I, there exists at most one (O, k, p') s.t. $p \xmapsto[I]{O,\,k} p'$.*

THEOREM 2. *For all p, I, O, k, p', the proof of $p \xmapsto[I]{O,\,k} p'$ is unique if it exists.*

PROOF SKETCH. By induction on the proof of $p \xmapsto[I]{O,\,k} p'$.
Let's for instance suppose that the last rule used in the proof is (signal−−):

$$\frac{q \xmapsto[I\backslash\{S\}]{O,\,k} q' \quad S \notin O \qquad q \xmapsto[I\cup\{S\}]{O_0,\,k_0} q'_0 \quad S \notin O_0}{\texttt{signal } S \texttt{ in } q \texttt{ end} \xmapsto[I]{O,\,k} \texttt{signal } S \texttt{ in } q' \texttt{ end}}$$

By induction hypothesis, p (thus q) and I being fixed, there exist:

– a unique (O, k, q') such that $q \xmapsto[I\backslash\{S\}]{O,k} q'$, with a unique proof;

– a unique (O_0, k_0, q'_0) such that $q \xmapsto[I\cup\{S\}]{O_0,k_0} q'_0$, with a unique proof.

Since $S \notin O$, rule (signal++) does not apply to p with inputs I, so that:

– if $p \xmapsto[I]{O_1,k_1} p'_1$ then $O = O_1$, $k = k_1$ and $p' = p'_1$;

– the proof of $p \xmapsto[I]{O,k} p'$ is unique. □

There is no need to count proofs and reactions in the deterministic semantics. In particular, there are exactly as many proofs as there are reactions.

It should be understood that proving a reaction in the determinism semantics and establishing strong determinism in the logical behavioral semantics have equivalent computational complexity. Nevertheless, in our view, merging the reasoning and meta-reasoning levels is a major improvement.

5.4 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 exist 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 WITH THE LOGICAL BEHAVIORAL SEMANTICS

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

6.1 Properness implies strong correctness

Theorem 3. *If* $p \xmapsto[I]{O,k} p'$ *then* $p \xrightarrow[I]{O,k} p'$.

Theorem 4. *If* $p \xmapsto[I]{O,k} p'$ *and* $p \xrightarrow[I]{O_1,k_1} p'_1$ *then* $O = O_1$, $k = k_1$, $p' = p'_1$.

Theorem 5. *If* $p \xmapsto[I]{O,k} p'$ *then the proof of* $p \xrightarrow[I]{O,k} p$ *is unique.*

Proof Sketch. By induction on the proof of $p \xmapsto[I]{O,k} p'$.

Let's again suppose that the last rule used in the proof is (signal−−):

$$\frac{q \xmapsto[I\backslash\{S\}]{O,k} q' \qquad S \notin O \qquad q \xmapsto[I\cup\{S\}]{O_0,k_0} q'_0 \qquad S \notin O^+}{\texttt{signal } S \texttt{ in } q \texttt{ end} \xmapsto[I]{O,k} \texttt{signal } S \texttt{ in } q' \texttt{ end}}$$

By induction hypothesis,

- $q \xrightarrow[I\backslash\{S\}]{O,k} q'$ with a unique proof. If $q \xrightarrow[I\backslash\{S\}]{O_1,k_1} q'_1$ then $O = O_1$, $k = k_1$, and $q' = q'_1$.
- $q \xrightarrow[I\cup\{S\}]{O_0,k_0} q'_0$ with a unique proof. If $q \xrightarrow[I\cup\{S\}]{O_1,k_1} q'_1$ then $O_0 = O_1$, $k_0 = k_1$, and $q'_0 = q'_1$.

Since $S \notin O_0$, rule (signal+) does not apply to p with inputs I. Since $S \notin O$, using rule (signal−), we obtain:

- $p \xrightarrow[I]{O,k} p'$ with a unique proof;
- if $p \xrightarrow[I]{O_1,k_1} p'_1$ then $q \xrightarrow[I\backslash\{S\}]{O_1,k_1} q'_1$, thus $O = O_1$, $k = k_1$, $p' = p'_1$. □

In summary, 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. 3), but also that it *must* react this way in both semantics (Th. 1 and 4), and that its internal behavior is deterministic (Th. 2 and 5).

COROLLARY 6. *If p is proper then p is strongly correct.*

PROOF SKETCH. If p is proper and p' is such that $p \xrightarrow{*} p'$ then, by induction on the length of the derivation, $p \overset{*}{\longmapsto} p'$. Since p' is initially proper, p' is reactive and strongly deterministic. Therefore 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
```
  Intuitively, this program is logically correct because the logical behavioral semantics does not care about the instantaneous loop "`loop nothing end`" as `S` is always absent. The deterministic semantics on the other hand expects branches to have coherent behaviors even if not executed, so this program is not proper. Formally, whatever the input set I, the logical behavioral semantics defines the following unique proof tree for this program:

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

The deterministic semantics however defines no reaction for this statement. Neither the (signal++) nor the (signal−−) rule applies, as "`loop nothing end`" and "`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 loop body "`signal S in present S then emit S else pause end end`" is not deterministic. It may react in two possible ways in the logical behavioral semantics with resulting completion codes 0 and 1 (cf. Section 4). Since the (loop) rule of the logical behavioral semantics requires "$k \neq 0$", exactly one of the two proof trees can be extended into a proof tree for the whole program, which is therefore both reactive and deterministic. In other words, the enclosing loop enforces `S` to be absent (because it requires the "`else pause`" branch to be executed).

On the other hand, the deterministic semantics defines no reaction for the body of the loop, hence no reaction for the loop, which is not initially proper.

6.3 Strongly correct non-proper statements

We have shown that: Proper statements $\subsetneq$ Strongly-correct statements. Let's now consider those statements that are strongly correct but not proper.

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.

Theorem 7. *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 Sketch. By structural induction on p, we prove that if p and all its subterms are reactive and strongly deterministic then p is initially proper.

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. There exists $(k^-, O^-, q^-, k^+, O^+, q^+)$ such that:

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

There are four cases:

(1) $S \in O^-$, $S \in O^+$, then by rule (signal++), $p \xmapsto[I]{O^+ \setminus \{S\}, k^+}$ `signal` S `in` q^+ `end`. In this case p is initially proper.

(2) $S \notin O^-$, $S \notin O^+$, then by rule (signal−−), $p \xmapsto[I]{O^-, k^-}$ `signal` S `in` q^- `end`. In this case p is initially proper.

(3) $S \in O^+$, $S \notin O^-$:

– by rule (signal+), $p \xrightarrow[I]{O^+ \setminus \{S\}, k^+}$ `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.

(4) $S \notin O^+$, $S \in O^-$, then neither (signal+) nor (signal−) is applicable. Therefore, p is not reactive. Contradiction.

Cases (3) and (4) are not possible, so p is initially proper. □

Going back to Theorem 7, if q is a non-reactive or non-strongly-deterministic subterm of the reactive and deterministic statement p, then 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 fact, 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[4] program "`signal S in nothing end`", which is proper.

–
```
loop
  signal S in
    present S then emit S else pause end
  end
end
```
The loop 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 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, which we happily discard.

6.4 The order of signal declarations matters

The following program is proper:

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

[4]Technically, they are strongly bisimilar [Milner 1989] w.r.t. the logical behavioral semantics.

However, as observed by one reviewer, the program obtained by permuting the declarations of `S` and `T` is no longer proper even if it remains strongly correct:

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

While this may seem unfortunate, this again illustrates Theorem 7. Indeed, for the initial program, the boxed statement when extracted from its context remains strongly correct whereas the boxed statement in the revised program is not.

7. REACTIVE DETERMINISTIC SEMANTICS

The existence of a single proof tree of the deterministic semantics establishes that a statement is reactive with respect to a given set of inputs. Establishing non-reactivity however still requires a meta proof. In this section, we further transform the semantics, so that non-reactivity (more exactly non-initial-properness) may be shown with a simple proof tree as well. We shall use the symbol "$\bullet\!\!\longrightarrow$" for the transitions of the resulting semantics.

In order to reason about non-reactivity within the rules of the semantics themselves, we have to encode it somehow: we reuse the existing exception propagation mechanism of the semantics to this aim. We introduce the completion code "∞" to represent non-reactivity. It obeys the obvious arithmetic relations:

$$\forall k \in (\mathbb{N} \cup \{\infty\}) : \max(k, \infty) = \infty \qquad\qquad \downarrow\infty = \infty - 1 = \infty$$

The `loop` and `signal` constructs are "responsible" for non-reactivity as well as non-initial-properness. Indeed, if we try to prove by structural induction that all statements are initially proper it fails because:

– even if $p \xmapsto[I]{O,k} p'$, if $k = 0$ then "`loop` p `end`" is not initially proper;

– even if $p \xmapsto[I\cup\{S\}]{O^+,k^+} p^+$ and $p \xmapsto[I\backslash\{S\}]{O^-,k^-} p^-$:
  – if $S \in O^+$ and $S \notin O^-$ then "`signal` S `in` p `end`" is not initially proper;
  – if $S \notin O^+$ and $S \in O^-$ then "`signal` S `in` p `end`" is not initially proper.

7.1 Revised Rules

We propose to update the semantics of these two constructs and replace the rules (loop), (signal++), and (signal−−) of the deterministic semantics by the rules of Figure 7, where:

– (loop) specifies the behavior of correct loops (in fact unchanged);

– (loop-error) reports instantaneous loop bodies;

– (signal++) defines present signals;

– (signal−−) defines absent signals;

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad k \neq 0}{\texttt{loop } p \texttt{ end} \bullet\!\xrightarrow[I]{O,\,k} p' \texttt{; loop } p \texttt{ end}} \quad \text{(loop)}$$

$$\frac{p \bullet\!\xrightarrow[I]{O,\,k} p' \qquad k = 0}{\texttt{loop } p \texttt{ end} \bullet\!\xrightarrow[I]{\emptyset,\,\infty} \texttt{nothing}} \quad \text{(loop-error)}$$

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

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \notin O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \notin O^+ \qquad \max(k^-, k^+) < \infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \bullet\!\xrightarrow[I]{O^-,\,k^-} \texttt{signal } S \texttt{ in } p^- \texttt{ end}} \quad \text{(signal−−)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \in O^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \notin O^+ \qquad k^+ \neq \infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \bullet\!\xrightarrow[I]{\emptyset,\,\infty} \texttt{nothing}} \quad \text{(signal+−)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad S \notin O^- \qquad k^- \neq \infty \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad S \in O^+}{\texttt{signal } S \texttt{ in } p \texttt{ end} \bullet\!\xrightarrow[I]{\emptyset,\,\infty} \texttt{nothing}} \quad \text{(signal−+)}$$

$$\frac{p \bullet\!\xrightarrow[I\backslash\{S\}]{O^-,\,k^-} p^- \qquad p \bullet\!\xrightarrow[I\cup\{S\}]{O^+,\,k^+} p^+ \qquad \max(k^-, k^+) = \infty}{\texttt{signal } S \texttt{ in } p \texttt{ end} \bullet\!\xrightarrow[I]{\emptyset,\,\infty} \texttt{nothing}} \quad \text{(signal}\infty\text{)}$$

Fig. 7. Reactive Deterministic Semantics: the `loop` and `signal` Constructs

- (signal+−) reports non-reactive signals;
- (signal−+) reports non-strongly-deterministic signals;
- (signal∞) propagates errors through local signal declarations.

By construction, if the completion code k of a reaction is infinite then its output set O is empty. As a result, there is no need to check explicitly that k^+ in rule (signal−+) is finite for instance.

Thanks to the existing exception propagation mechanism, there is no need for further error propagation rules. Rules (nothing), (pause), (exit), (emit), (parallel), (present+), (present−), (try-catch), (try-through), (sequence-p), and (sequence-q) remains the ones of the logical behavioral semantics (cf. Figure 3).

7.2 Determinism and Reactivity

The new semantics is *globally deterministic* in the sense of Section 5.3:

LEMMA 8. *For all p and I, there exists at most one (O, k, p') s.t. $p \bullet\!\xrightarrow[I]{O,\,k} p'$.*

THEOREM 9. *For all p, I, O, k, p', the proof of $p \bullet\!\xrightarrow[I]{O,\,k} p'$ is unique if it exists.*

Proof Sketch. Similar to that of Th. 1 and 2. □

The new semantics is *globally reactive*:

Lemma 10. *For all p and I, there exists at least one (O, k, p') s.t. $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$.*

Proof Sketch. By structural induction on p. If p is "`signal` S `in` q `end`" then, by induction hypothesis, there exist (O^-, k^-, q^-) and (O^+, k^+, q^+) such that:

$$q \overset{O^-,k^-}{\underset{I\backslash\{S\}}{\bullet\!\!\longrightarrow}} q^- \text{ and } q \overset{O^+,k^+}{\underset{I\cup\{S\}}{\bullet\!\!\longrightarrow}} q^+$$

which implies that there exists (O, k, p') such that $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ by rule:

$S \in O^-$	$S \in O^+$	$k^- < \infty$	$k^+ < \infty$	
?	?	no	?	(signal∞)
?	?	?	no	(signal∞)
yes	yes	yes	yes	(signal$++$)
yes	no	yes	yes	(signal$+-$)
no	yes	yes	yes	(signal$-+$)
no	no	yes	yes	(signal$--$)

And similarly for `loop` constructs. □

Corollary 11. *For all p and I, there exists exactly one (O, k, p') s.t. $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$.*

Therefore, the reactive deterministic semantics defines exactly one execution per program and input sequence. Correct and incorrect executions can be equally easily described using the rules of the reactive deterministic semantics.

7.3 Errors

As a sanity check, we verify that:

Lemma 12. *If $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ then $k = \infty$ iff the proof of this reaction involves one of the rules* (signal$-+$), (signal$+-$), *or* (loop-error).

Proof Sketch. These three rules are the only rules that can deduce the completion code ∞ without assuming the code completion ∞ as a premise. Reciprocally, by induction on proofs, if $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ with $k < \infty$ then all completion codes occurring in the proof are finite completion codes. □

More importantly, a statement is initially proper iff it cannot react with completion code ∞ in the reactive deterministic semantics:

Theorem 13. *For all (p, I, O, k, p'), $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ iff $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ and $k < \infty$.*

Proof Sketch. A proof tree of $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ can be extended into a proof tree of $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ by adding finiteness checks for completion codes whenever required. Reciprocally, a proof tree of $p \overset{O,k}{\underset{I}{\bullet\!\!\longrightarrow}} p'$ with $k < \infty$ can be changed into a proof tree of $p \overset{O,k}{\underset{I}{\longmapsto}} p'$ by removing such checks, since by Lemma 12 this proof tree cannot contain deduction of the form (signal−+), (signal+−), (loop-error), or, as an immediate consequence, (signal∞). □

As a consequence, correctness w.r.t. to the deterministic semantics, i.e., properness, is equivalent to correctness w.r.t. to the reactive deterministic semantics:

Corollary 14. *The program p is proper iff there exists no execution of p in the reactive deterministic semantics that terminates with completion code* ∞.

Proof Sketch. By Th. 13, $p \overset{O,1}{\underset{I}{\longmapsto}} p'$ iff $p \overset{O,1}{\underset{I}{\bullet\!\!\longrightarrow}} p'$. Therefore, by induction on the length of the derivation, $p \overset{*}{\mapsto} p'$ iff $p \overset{*}{\bullet\!\!\to} p'$, where $\overset{*}{\bullet\!\!\to}$ is the reflexive transitive closure of $\bullet\!\!\to$: $p \bullet\!\!\to p'$ iff $\exists I, \exists O : p \overset{O,1}{\underset{I}{\bullet\!\!\longrightarrow}} p'$. Thus the result, again by Th. 13. □

In summary, the reactive deterministic semantics provides an alternate characterization of non-proper programs. While it relies on additional deduction rules, since it precisely makes explicit the kind of deductions that are required to prove non-properness, it may be more convenient than the deterministic semantics for dealing with incorrect programs. For instance, it can be easily extended to discriminate “instantaneous loop errors” from “causality errors”, as further discussed in [Tardieu 2004], an old, up to now fuzzy distinction in the Esterel literature.

8. COMPARISON WITH THE CONSTRUCTIVE SEMANTICS

The constructive semantics of Esterel [Berry 1999] 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* (i.e., correct w.r.t. the constructive semantics) because the status of S has to 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.

Since reactions in the constructive semantics are defined as least fixed points[5], there is at most one execution defined for each program and input sequence. In other words, the constructive semantics is globally deterministic in the sense of Section 5.3.

In summary, even if both semantics are globally deterministic, the reasons for this property are different and the corresponding correctness criteria do not match. They could be combined. We leave this for future work.

9. CONCLUSION

We introduce two new semantics for pure Esterel, respectively called the *deterministic semantics* and the *reactive deterministic semantics*.

In contrast with the logical behavioral semantics of Berry and Gonthier, our deterministic semantics 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. In contrast with the constructive semantics of Berry (three-valued logic), our deterministic semantics retains the simple formalism of the logical behavioral semantics (binary logic).

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.

Thanks to these properties, we believe that our deterministic semantics provides a better starting point for formal reasoning about pure Esterel programs than both the logical behavioral semantics and the constructive semantics. In addition, the reactive deterministic semantics encodes errors explicitly, thus making formal reasoning about incorrect programs easier as well.

ACKNOWLEDGMENT

I wish to thank the reviewers and associate editor for their comments, which helped to substantially improve the readability and content of the paper.

REFERENCES

Benveniste, A. and Berry, G. 1991. The synchronous approach to reactive real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue 79,* 9, 1270–1282.

[5]In each instant of execution, a pure Esterel program denotes a system of Boolean equations $\mathcal{E} : \vec{v} = F(\vec{v})$ whose n variables $\vec{v}$ are the various signals occurring the program. The logical behavioral semantics is only concerned with the existence and uniqueness of solutions for $\mathcal{E}$ (the fixed points of the function F in the domain $\{\textsc{true}, \textsc{false}\}^n$). The constructive semantics defines and computes the (unique) least fixed point of F in the meet-semilattice $(\{\textsc{true}, \textsc{false}\}_\perp)^n$, having first appropriately extended F to this new structure in which "$\perp$" intuitively represents unknown statuses.

Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and de Simone, R. 2003. The synchronous languages twelve years later. *Embedded Systems, Proceedings of the IEEE, Special issue 91,* 1, 64–83.

Berry, G. 1993. Preemption and concurrency. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science, vol. 761. Springer, 72–93.

Berry, G. 1999. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org/.

Berry, G. 2000a. The Esterel language primer v5_91. http://www-sop.inria.fr/esterel.org/.

Berry, G. 2000b. The foundations of Esterel. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, Cambridge, MA, 425–454.

Berry, G. and Cosserat, L. 1984. The synchronous programming language Esterel and its mathematical semantics. In *Seminar on Concurrency*. Lecture Notes in Computer Science, vol. 197. Springer, Pittsburg, PA, 389–448.

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

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

Edwards, S. 2000. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, Norwell, MA.

Edwards, S. A., Kapadia, V., and Halas, M. 2004. Compiling Esterel into static discrete-event code. In *Proceedings of the Synchronous Languages, Applications, and Programming Workshop*. Electronic Notes in Theoretical Computer Science. Elsevier, Barcelona, Spain.

Gonthier, G. 1988. Sémantique et modèles d'exécution des langages réactifs synchrones: application à Esterel. Ph.D. thesis, Université d'Orsay.

Groote, J. F. 1993. Transition system specifications with negative premises. *Theoretical Computer Science 118,* 2, 263–299.

Halbwachs, N. 1993. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue 79,* 9, 1305–1320.

INRIA, ENSMP, and ARMINES. 2000. The Esterel v5_92 Compiler.
http://www-sop.inria.fr/esterel.org/.

Le Guernic, P., Le Borgne, M., Gauthier, T., and Lemaire, C. 1991. Programming real time applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue 79,* 9, 1321–1336.

Malik, S. 1993. Analysis of cyclic combinational circuits. In *Proceedings of the 1993 IEEE/ACM International Conference on CAD*. IEEE, Santa Clara, CA, 618–625.

Milner, R. 1989. *Communication and Concurrency.* Series in Computer Science. Prentice Hall.

Plotkin, G. 1981. A structural approach to operational semantics. Report DAIMI FN-19, Aarhus University, Denmark.

Potop-Butucaru, D. 2002. Optimizations for faster execution of Esterel programs. Ph.D. thesis, Ecole des Mines de Paris.

Tardieu, O. 2004. Loops in Esterel: from operational semantics to formally specified compilers. Ph.D. thesis, Ecole des Mines de Paris. http://olivier.tardieu.free.fr/papers/these.pdf.

Tardieu, O. and de Simone, R. 2003. Instantaneous termination in pure Esterel. In *Proceedings of the 10th International Static Analysis Symposium*. Lecture Notes in Computer Science, vol. 2694. Springer, San Diego, CA, 91–108.

Received February 2005; accepted October 2005