Logical semantics of Esterel with unconstrained local signals

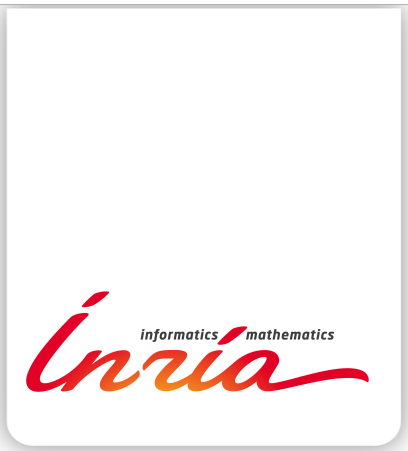
Bernard Paul Serpette

Logical semantics of Esterel with unconstrained local signals

Bernard P. Serpette

RESEARCH REPORT
N° 8942
Août 2016
Project-Team Indes

ISSN 0249-6399 ISRN INRIA/RR--8942--FR+ENG



Logical semantics of Esterel with unconstrained local signals

Bernard P. Serpette

Project-Team Indes

Research Report n° 8942 — Août 2016 — 16 pages

Abstract: Esterel is a synchronous programming language where processes interact through signals. The logical semantics of this language express the meaning of each syntactic constructions, knowing *a priori* the set of emitted signals. Nevertheless, a special case have to be made for local signals, making the semantics non-deterministic. In this paper, we propose a new logical semantics where the rules are deterministic. We formalise some correspondences for usual definitions (coherence, reactivity, determinism, correctness) between the two semantics. These correspondences are formally proved in Coq.

Key-words: Esterel, synchronous languages, local signals, operational semantics, determinism, formal proof

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Une sémantique logique d'Esterel sans contrainte sur les signaux locaux

Résumé : Esterel est un langage de programmation synchrone où les processus interagissent au travers de signaux. La sémantique logique de ce langage définit le comportement de chaque élément syntaxique en connaissant, à priori, l'ensemble des signaux émis. Néanmoins, un cas spécial doit être fait pour l'interprétation des signaux locaux, rendant ainsi la sématique non déterministe. Dans ce papier, nous proposons une sémantique logique où les règles deviennent déterministe. Nous formalisons une correspondance, entre les deux sémantique, pour les définitions usuelles (cohérence, réactivité, déterminisme, correction). Ces correspondances sont formellement prouvées avec le système Coq.

Mots-clés : Esterel, langages synchrones, signaux locaux, sémantique opérationnelle, déterminisme, preuve formelle

1 Introduction

We informally present the **Esterel**[1] programming language focusing on the notion of process. An Esterel program is made of several processes, each of them executing statements. The processes communicate via *signals*. A process can raise a signal S with the statement "**emit** S". Signals can be tested with the statement "**present** S **then** P_1 **else** P_2 **end**". If the signal S is emitted, the evaluation continues with P_1. We will detail this statement latter in this section. A process can emit a signal but cannot reset it, as if it was never been emitted. The only way to reset a signal is to erase all signals at the same time, defining a notion of *instant*. All processes must cooperate to reach the end of one instant. Individually, a process may decide to finish its instant by executing the statement "**pause**", and it is when all processes have executed a pause or have finished their execution that the current instant is globally and synchronously closed.

Coming back to the "**present**" statement, if one process have to evaluate a statement "**present** S **then** P_1 **else** P_2 **end**", and if the signal S is not already emitted, we have to *guess* the status of this signal (emitted or not in the instant). This *guess* is the hardest part of the dynamic semantics (constructive behavioural semantics, chapter 7 of [2]). But we can imagine that the *guess* is given as an oracle, and we can verify *a posteriori*, at the end of the instant, that the guess is conform to the reality. The semantics, driven by a guess of all emitted signal, is called *logical behavioural semantics* (chapter 6 of [2]). This semantics allow to clearly define the properties we need for a program : *coherency* (a guess is correct), *reactivity* (at least one guess is correct), *determinism* (at most one guess is correct) and *correctness* (exactly one guess is correct). In this paper, we will talk only on this logical semantics.

An another feature of Esterel is to permit the definition of *local signals*, a signal which have a meaning only inside a lexical scope. A local signal is introduced in a statement : **signal** S **in** P **end**. The logical semantics for local signals become less intuitive since the rules have to reintroduce some *guess* and have to check by hand the correctness, and thus leaving a non deterministic set of rules. Moreover, the property of correctness becomes difficult to specify : A program P is *strongly deterministic* if it is reactive and deterministic and if, furthermore, there exists a unique proof induced by the logical semantics rules ([2], p69).

In this paper, we will give a set of deterministic rules for the logical semantics. In section 2, we will recall the standard semantics with non-deterministic rules for local signals. In section 3 we present a new semantics with only one rule for local signals and we give the correspondence with the previous semantics. In section 4 we show other alternatives. In section 5 we details some lemmas needed for the main theorem introduced in section 3.

All the specifications and proofs[2] are done in the **Coq** system

2 Standard semantics

2.1 Abstract syntax

We start with the Coq inductive type ot the Abstract Syntax Tree (AST) of Esterel.

```
Inductive ast : Set :=
 | Nop
 | Pause
 | Emit (s:name)
 | Seq (p1:ast) (p2:ast)
 | Par (p1:ast) (p2:ast)
 | If (s:name) (p1:ast) (p2:ast)
 | Loop (p:ast)
 | Trap (tag:name) (p:ast)
 | Exit (tag:name)
 | Signal (s:name) (p:ast)
```

The domain of names (`name`) is unspecified, we only need an equality on them, the Leibniz equality of Coq - the structural equality - is convenient.

`Nop` is a statement that does nothing, runs in no time without emitting any signal. `Pause` delimits the

[1] In this paper, all underlined bold **words** can be found in Wikipedia.

[2] `ftp:ftp-sop.inria.fr/indes/rp/EsterelLogical.tar`

end of an instant for the process executing this statement. `Emit` emits the signal s. `Seq` put the two statements $p1$ and $p2$ in sequence, note that $p1$ and/or $p2$ may take several instants. `Par` creates two processes, one executing $p1$, the other executing $p2$, the main process waiting for the completion of its two sub-processes before continuing its own execution. The execution of $p1$ and $p2$ may take several instants. The two sub-processes must join on completion before resuming the execution of their creator. `If` checks if the signal s is present in the current instant. If so, the statement $p1$ is executed, otherwise $p2$ is executed. `Loop` makes an infinite repetition of the statement p. `Trap` allows the control to exit from a loop, it defines an escape point named *tag* during the execution of p. The execution of p may be aborted with a statement `Exit`(tag). `Signal` defines a local signal named s during the execution of p.

2.2 Semantics

The semantics of Esterel is generally based on Structural Operational Semantic rules of the form :

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

Where E is the set of present signals during the current instant, E' is the set of emitted signals during the execution of p in the same instant, k is called the *completion code* and denotes the fact that the evaluation of p have either finish its execution, either reach a `Pause`, or either aborting the computation with a pending `Exit`. p', called the *derivative* or the *residual* of p, is, according to the completion code, what remains to do for the next instant.

In most specifications, the completion codes are natural numbers. 0 denote the normal termination, 1 a paused statement and for $n > 1$, $n - 2$ gives the number of surrounding `Trap` to be traversed. This encoding can be related to the **de Bruijn indexes** for the **λ calculus** and allows to remove the trap names. Even if using natural numbers for completion codes have a fast comparison for the parallel synchronisation, we have decided to keep trap names in order to facilitate the proofs. Thus, these completion codes are defined with the following inductive type :

```
Inductive TermFlag : Set :=
 | TReturn
 | TPause
 | TExit (tag:name)
 | TError
```

We have added a `TError` constructor to complete the relation $\rightarrow$ for `Loop` expression as we will see later.

In order to compare trap names, the list of names of the surrounding `Trap` is added to the rules.

$$p \xrightarrow[\rho,E]{E',k} p'$$

Where ρ is the list of trap names pushed every time a `Trap` is analyzed. All the rules defining the standard semantics are listed in figure 1.

The first three axioms (`Nop`, `Pause` and `Exit`), are the ones that set the completion code without emitting. The last axiom (`Emit`), initiates the emitted signals set.

The two next rules (`If`) test the presence of a signal and take the corresponding branch. These rules are logical, the absence of a signal is known *a priori*. The hardest work, which is to guess the absent signals, is not depicted by the rules.

The rules for `Seq` analyse sequences. The first one is used when the completion code for left branch p says that p is either *paused* or raise a `TExit`. In this case, the right branch q has not to be analysed. The second rule, when we know that p is fully evaluated with a normal completion, has to analyse also the right branch.

The rule for `Par` analyses the two branches and collect both emitted signals.The only difficulty is to answer the correct completion code. There is a total order on completion codes depicted by the order in which constructors are listed in the definition of `TermFlag` (`TReturn` $<$ `TPause` $<$ `TExit` $<$ `Terror`)

saying that an *exit* is higher than everything and a *pause* is higher than a normal completion. This is depicted with the following function :

$$(\texttt{Nop}) \xrightarrow[\rho,E]{\emptyset,\texttt{TReturn}} (\texttt{Nop}) \qquad (\texttt{Pause}) \xrightarrow[\rho,E]{\emptyset,\texttt{TPause}} (\texttt{Nop}) \qquad (\texttt{Exit}\ t) \xrightarrow[\rho,E]{\emptyset,(\texttt{TExit}\ t)} (\texttt{Nop})$$

$$(\texttt{Emit}\ s) \xrightarrow[\rho,E]{\{s\},\texttt{TReturn}} (\texttt{Nop})$$

$$\frac{s \in E \ \wedge\ p \xrightarrow[\rho,E]{E',k} p'}{(\texttt{If}\ s\ p\ q) \xrightarrow[\rho,E]{E',k} p'} \qquad \frac{s \notin E \ \wedge\ q \xrightarrow[\rho,E]{E',k} q'}{(\texttt{If}\ s\ p\ q) \xrightarrow[\rho,E]{E',k} q'}$$

$$\frac{p \xrightarrow[\rho,E]{E',k} p' \ \wedge\ k \neq \texttt{TReturn}}{(\texttt{Seq}\ p\ q) \xrightarrow[\rho,E]{E',k} (\texttt{Seq}\ p'\ q)} \qquad \frac{p \xrightarrow[\rho,E]{E',\texttt{TReturn}} p' \ \wedge\ q \xrightarrow[\rho,E]{E'',k} q'}{(\texttt{Seq}\ p\ q) \xrightarrow[\rho,E]{E' \cup\ E'',k} q'}$$

$$\frac{p \xrightarrow[\rho,E]{E',k} p' \ \wedge\ q \xrightarrow[\rho,E]{E'',l} q'}{(\texttt{Par}\ p\ q) \xrightarrow[\rho,E]{E' \cup\ E'',(\texttt{maxk}\ k\ l\ \rho)} (\texttt{Par}\ p'\ q')}$$

$$\frac{p \xrightarrow[\rho,E]{E',k} p' \ \wedge\ k \neq \texttt{TReturn}}{(\texttt{Loop}\ p) \xrightarrow[\rho,E]{E',k} (\texttt{Seq}\ p'\ (\texttt{Loop}\ p))} \qquad \frac{p \xrightarrow[\rho,E]{E',\texttt{TReturn}} p'}{(\texttt{Loop}\ p) \xrightarrow[\rho,E]{E',\texttt{TError}} (\texttt{Seq}\ p'\ (\texttt{Loop}\ p))}$$

$$\frac{p \xrightarrow[(tag::\rho),E]{E',k} p' \ \wedge\ (k = \texttt{TReturn} \ \vee\ k = (\texttt{TExit}\ tag))}{(\texttt{Trap}\ tag\ p) \xrightarrow[\rho,E]{E',\texttt{Return}} (\texttt{Nop})} \qquad \frac{p \xrightarrow[(tag)::\rho,E]{E',k} p' \ \wedge\ \neg(k = \texttt{TReturn} \ \vee\ k = (\texttt{TExit}\ tag))}{(\texttt{Trap}\ tag\ p) \xrightarrow[\rho,E]{E',k} (\texttt{Trap}\ tag\ p')}$$

$$\frac{p \xrightarrow[\rho,E\cup\{s\}]{E',k} p' \ \wedge\ s \in E'}{(\texttt{Signal}\ s\ p) \xrightarrow[\rho,E]{E'-\{s\},k} (\texttt{Signal}\ s\ p')} \qquad \frac{p \xrightarrow[\rho,E-\{s\}]{E',k} p' \ \wedge\ s \notin E'}{(\texttt{Signal}\ s\ p) \xrightarrow[\rho,E]{E',k} (\texttt{Signal}\ s\ p')}$$

Figure 1: standard rules

Definition 1. `maxk`
(`maxk` k_1 k_2 ρ) $\Leftrightarrow$ **match** k_1, k_2 **with**
| `TReturn`, _ $\Rightarrow k_2$
| _, `TReturn` $\Rightarrow k_1$
| `TPause`, _ $\Rightarrow k_2$
| _, `TPause` $\Rightarrow k_1$
| `TError`, _ $\Rightarrow$ `TError`
| _, `TError` $\Rightarrow$ `TError`
| (`TExit` t_1), (`TExit` t_2) $\Rightarrow$ (`maxtag` t_1 t_2 ρ)

For example, for a program `Par` p q, if p is *paused* with a completion code `TPause`, while q has a normal termination `TReturn`, the program have to do something for the next instant and return the `TPause` completion code. When both branches raise a `TExit`, we have to compare the trap names, and the convention wants that the deepest trap name in the stack will be taken.

Definition 2. `maxtag`
(`maxtag` t_1 t_2 ρ) $\Leftrightarrow$ **match** ρ **with**
| `nil` $\Rightarrow$ `TError`
| $t :: \rho \Rightarrow$ **if** $t = t_1$
then (`TExit` t_2)
else if $t = t_2$
then (`TExit` t_1)
else (`maxtag` t_1 t_2 ρ)

If the end of the stack is reached, the two trap names doesn't have a corresponding `Trap` in the expression, the specific `TError` code is returned. The same effect may be achieved by surrounding the main program by one adequate `Trap`, but this approach is cleaner.

The first rule of `Loop` checks explicitly that the body is not instantaneous. If so, the second rule raises a `TError` completion code. In a preliminary version of this article, we have chosen to force the termination of the body by raising the termination code to `TPause` in case of instantaneous loop. In Atar's thesis [1] a similar rule is used where a program (`Loop` p) is rewritten in (`Seq` (`Par` p (`Pause`)) (`Loop` p))[3]. Adding the explicit treatment for instantaneous loops needs only minor changes in the proofs, so we have adopted this new rule with the idea that the `TError` completion may have other utilities.

[3]This idea seems to be assigned to Louis Mandel

The two next rules (`Trap`) evaluate the body with a new stack where the trap name is pushed. The first rule is applied when either the body has a normal code completion (`TReturn`) or a `TExit` with the same trap name. In this case, the whole statement have a `TReturn` code completion. The second rule analyses all other cases and returns the code completion of the body.

The two last rules focus on local signal declaration, this paper is mainly concerned by them. In a standard way, the presence (left) or the absence (right) of the local signal is explicitly checked by the rules. The left rule adds the local signal in the set of present signals E and check if this signal is really emitted in the result E'. In the inverse, the right rule removes the local signal from E and checks that this signal is still absent in the result E'. It exists some programs as (`Signal` s (`If` s (`Emit` s) (`Nop`))) that are accepted by the two rules (non determinism) and some others as (`Signal` s (`If` s (`Nop`) (`Emit` s))) that are rejected by the two rules.

2.3 Definitions of properties

The rules given in figure 1 doesn't check, except for the local signals, any correspondence between the present signals in E and the emitted signal E'. For example, the fact (`Emit` s) $\xrightarrow[\texttt{nil},\emptyset]{\{s\},\texttt{TRteurn}}$ (`Nop`), where the signal s is absent but emitted, is directly accepted by the rule `emit`. The correspondence between E and E' is given informally by the coherence law ([2] chapter 3) : *A signal* `S` *is present in an instant if and only if an "*`emit S`*" statement is executed in this instant.* A more formal definition can be found in [2] (section 6.2) via a *behavioral transition* on programs depicted by :

$$P \xrightarrow[I]{O} P'$$

Where I and O are respectively the *input*s (the set of signals that are forced by some external device) and the *output*s (the set of signals that are emitted during the execution of P) of the program. The coherence law is defined by :

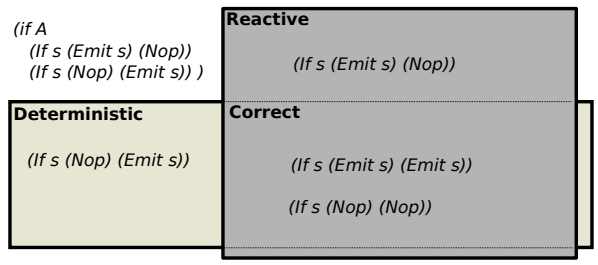


Figure 2: standard definition diagram

Definition 3 (Standard coherence)**.**
$(\texttt{coherent}\ P\ I\ O\ P') \Leftrightarrow \exists k, P \xrightarrow[\texttt{nil},I\cup O]{O,k} P'$

Then, we can define the set of programs that have at least a behavioral transition, a property known as *reactivity*.

Definition 4 (Standard reactivity)**.**
$(\texttt{reactive}\ P) \Leftrightarrow \forall I, \exists O\ P', (\texttt{coherent}\ P\ I\ O\ P')$

In a similar way, we can define the set of programs that have at most one behavioral transition, a property known as *deterministic*:

Definition 5 (Standard determinism)**.**
$(\texttt{deterministic}\ P) \Leftrightarrow \forall I\ O_1\ P_1\ O_2\ P_2,$
$(\texttt{coherent}\ P\ I\ O_1\ P_1) \Rightarrow$
$(\texttt{coherent}\ P\ I\ O_2\ P_2) \Rightarrow$
$((O_1 = O_2)\ \wedge\ (P_1 = P_2))$

Finally a program is said to be *correct* when it is both reactive and deterministic:

Definition 6 (Standard correctness)**.**
$(\texttt{correct}\ P) \Leftrightarrow (\texttt{reactive}\ P) \wedge (\texttt{deterministic}\ P)$

In figure 2 we give a set-oriented representation of these properties with a program example for each subset, "A" represents an input event that can be emitted by an external device.

The definitions of coherence, reactivity, determinism and correctness affect only *global* signals. For local signals, the coherence in insured by the rules, but it exists programs, like (`Signal` s (`If` s (`Emit` s) (`Nop`)))), that are *globally* deterministic but *locally* non deterministic since both s being present or absent is accepted. To control this fact, [2] gives the following definition: a program P is *strongly deterministic* if it is correct and if, furthermore, *there exists a unique proof of the unique transition* $P \xrightarrow[I]{O} P'$. Since this kind of definition is not directly translatable in Coq, we have switched to another specification where local signals can be manipulated by general properties (reactivity, determinism, ...).

3 Logical semantics with paths

Given a program P=(`Seq` (`Signal` s ...) (`Signal` s ...)), it is difficult, for P, to talk about s without ambiguity. The simple idea of this section is to refer a local signal not by its name but by the *path* where the local definition occurs.

3.1 Paths

A path is a sequence of edges from a root, the main program, to a dedicated sub-expression. An edge represents the link between an expression and one of its direct sub-expressions. For a specific constructor of the abstract syntax tree, (`If` $s\ p_1\ p_2$) for example, we create as much edge's contractors as we have recursive fields to the abstract syntax tree. For our example, we create two constructor `IfLeft` and `IfRight`. Putting all the constructors together we obtain the following definition for edges:

```
Inductive edge : Set :=
 | SeqLeft
 | SeqRight
 | ParLeft
 | ParRight
 | IfLeft
 | IfRight
 | LoopIn
 | TrapIn   (tag:name)
 | SignalIn (s:name)
```

Local signal names and trap names are kept in the edge structure to be used by the new semantics. With this definition, paths are simply list of edges:

```
Definition path := (list edge).
```

Given the statement (`Par` E_1 (`Seq` (`If` s E_2 (`Signal`...)) E_3)), the path for the local signal declaration is generally given with (`ParRight SeqLeft IfRight`), giving the path from the main program to the local definition. For our specification, for induction purpose, it is simpler to built the paths in the opposite way, (`IfRight SeqLeft ParRight`), showing the path from the local definition to the root.

3.2 Signals

In order to keep the AST definition, signals occurring in the syntax (`Emit`, `If` and `Signal` constructors) are still referenced by their names. But in the sets of present and emitted signals (sets E and E' of the semantics) we have to add a path associated to local signals. We define the type of these signals with :

```
Inductive psignal : Set :=
 | Local  (p:path)
 | Global (s:name)
```

A `psignal` is either a local signal associated with the path of its definition or a global one, without path, only referenced by its name. For example, (`Local nil`) refers to the signal s for all programs with the structure (`Signal` s ...). But we can still talk about this signal for a program not beginning with a signal declaration, (`Nop`) for example. For a local signal (`Local` π), we *suppose* that, at the place pointed by the path π, we will find a local signal declaration, the same way that for all signal names declared as present in E in the previous rules, we *suppose* that the signal occurs in the program. It is the coherent property that will reject present signals with wrong paths.

3.3 Semantics

With these new kind of signals, we can define a relation :

$$p \xrightarrow[\pi,E]{E',k} p'$$

Where E and E', the present and emitted signals, are now sets of `psignal`, and π is the path where p occurs in the main program. This path is also used to retrieve the information kept in the previous ρ argument (i.e. the surrounding traps) with the function $\pi 2\rho$ (`path2stack` in Coq):

Definition 7 (path to list of trap names : $\pi 2\rho$)**.**
$(\pi 2\rho\ c) =$ **match** π **with**
| **nil** $\Rightarrow$ **nil**
| $g :: \pi \Rightarrow$ **match** g **with**
| (`TrapIn` t) $\Rightarrow t :: (\pi 2\rho\ c)$
| _ $\Rightarrow (\pi 2\rho\ \pi)$

The only major changes, regarding previous semantics, concern rules using signals : emission, test and local signal declarations. For the emission we change the rule with :

$$(\texttt{Emit}\ s) \xrightarrow[\pi,E]{\{\uparrow_s^\pi\},\texttt{TReturn}} (\texttt{Nop})$$

The function $\uparrow_s^\pi$, named `get_psignal` in Coq, given a path π and a signal name s, will find, using π, the `psignal` corresponding to s. This function is defined by :

Definition 8. `get_psignal`
$\uparrow_s^\pi =$ **match** π **with**
| **nil** $\Rightarrow$ (`Global` s)
| $g :: \pi \Rightarrow$ **match** g **with**
| (`SignalIn` s') $\Rightarrow$
if $s = s'$
then (`Local` π)
else $\uparrow_s^\pi$
| _ $\Rightarrow \uparrow_s^\pi$

The function scans the path from left to right, or upward regarding the tree structure. If an edge (`SignaIn` s) is reached then a (`Local` π), with the current path, is returned. If the end of the path is reached, then the signal is global and (`Global` s) is returned.

For the signal test, we change the two rules by :

$$\frac{\uparrow_s^\pi \in E \ \wedge\ p \xrightarrow[\texttt{IfLeft}::\pi,E]{E',k} p'}{(\texttt{If}\ s\ p\ q) \xrightarrow[\pi,E]{E',k} p'}$$

$$\frac{\uparrow_s^\pi \notin E \ \wedge \ q \xrightarrow[\texttt{IfRight}::\pi,E]{E',k} q'}{(\texttt{If}\ s\ p\ q) \xrightarrow[\pi,E]{E',k} q'}$$

The only change regarding standard rules is that we use the $\uparrow$ to verify the presence of the signal in E. Note also that every time a subtree have to be analysed, a corresponding edge must be pushed on the path.

Finally, since the present local signals are also given *a priori* in E, the two rules for local signal declaration are resumed in :

$$\frac{p \xrightarrow[(\texttt{SignalIn}\ s)::\pi,E]{E',k} p'}{(\texttt{Signal}\ s\ p) \xrightarrow[\pi,E]{E',k} (\texttt{Signal}\ s\ p')}$$

We have now only one rule for local definitions and the premisses for all other rules are mutually exclusive, the relation $\rightarrow$ is restricted to a function $(p, \pi, E) \rightarrow (E', k, p')$. The non-determinism appearing in the `Signal` standard rules is vanished. Instead of returning a triplet, in Coq, we have defined three functions with the same parameters (p, c and e) : `term` which returns the completion code, `emit` which returns the emitted signals and `reduce` which returns the residual program. We can observe, which is not immediate showing the rules, that these three functions are not mutually dependent : `term` is independent of the two others and `term` is used inside `emit` and `reduce` only for analyzing `Seq` nodes.

3.4 Definitions of properties

We will keep the names I and O for inputs and outputs of the previous specification. For inputs, we have to keep set of signal names : an external device don't have access to local signals. To convert a set of signal names to a set of global definitions we will use the function G^+ defined as :

Definition 9. `globalize`
$(G^+\ I) = \{(\texttt{Global}\ s)/s \in I\}$

Outputs correspond to emitted signals of the semantics. They are sets of `psignal`. We will note O^+ these kind of sets. Since we want to compare the two semantics, we will have to retrieve global signal names from set of `psignal` via the G^- definition :

Definition 10. `unglobalize`
$(G^-\ O^+) = \{s/(\texttt{Global}\ s) \in O^+\}$

Having functions instead of a relation allow to remove the residual program and the existential constructor in the coherency property definition :

Definition 11 (path coherency)**.**
$(\texttt{coherent}^+\ P\ I\ O^+) \Leftrightarrow$
$(\texttt{emit}\ P\ \texttt{nil}\ ((G^+\ I) \cup O^+)) = O^+$

The new definition of reactivity is similar with one useless existential operator :

Definition 12 (path reactivity)**.**
$(\texttt{reactive}^+\ P) \Leftrightarrow \forall I, \exists O^+, (\texttt{coherent}^+\ P\ I\ O^+)$

In a similar way, we can define the *deterministic* property :

Definition 13 (path determinism)**.**
$(\texttt{deterministic}^+\ P) \Leftrightarrow \forall I\ O_1^+\ O_2^+,$
$(\texttt{coherent}^+\ P\ I\ O_1^+) \Rightarrow$
$(\texttt{coherent}^+\ P\ I\ O_2^+) \Rightarrow$
$(O_1^+ = O_2^+)$

We hope that $\texttt{deterministic}^+$ characterise the *strongly* determinism property of [2]. Finally the *correct* property :

Definition 14 (path correctness)**.**
$(\texttt{correct}^+\ P) \Leftrightarrow$
$(\texttt{reactive}^+\ P)\ \wedge\ (\texttt{deterministic}^+\ P)$

3.5 Correspondences

The two definitions of coherency are related. The following theorem may be used as an alternative definition of standard coherency.

Theorem 1. `coherent_corr`
$(\texttt{coherent}\ P\ I\ O\ P') \Leftrightarrow \exists O^+,$
$\quad (\texttt{coherent}^+\ P\ I\ O^+)$
$\quad \wedge\ O = (G^-\ O^+)$
$\quad \wedge\ P' = (\texttt{reduce}\ P\ \texttt{nil}\ ((G^+\ I) \cup O^+))$

If a program is coherent with one logical semantic, it is also coherent with the other. This theorem is used to compare reactivity and determinism of the two semantics. For the reactivity, since it is enforced by the standard rules for local signals, we have an exact correspondence :

Theorem 2. `reactive_corr`
$(\texttt{reactive}\ P) \Leftrightarrow (\texttt{reactive}^+\ P)$

For determinism, the standard rules can only express a *global* determinism, while the rules with paths may also talk about determinism on local signals. To compare the two definitions of determinism, we have to define the notion of *local determinism*, which verify that a coherent program have a unique solution restricted to local signals :

Definition 15 (local determinism)**.**
$(\texttt{local\_deterministic}\ P) \Leftrightarrow \forall\ I\ O_1^+\ O_2^+,$
$(\texttt{coherent}^+\ P\ I\ O_1^+) \Rightarrow$
$(\texttt{coherent}^+\ P\ I\ O_2^+) \Rightarrow$
$(\texttt{locals}\ O_1^+) = (\texttt{locals}\ O_2^+)$

Where `locals` extract the local signals from its argument :

Definition 16 (local definitions filter)**.**
$(\texttt{locals}\ e^+) = \{(\texttt{Local}\ c)/(\texttt{Local}\ c) \in e^+\}$

With these definitions we can make the correspondence between the two semantics for deterministic programs.

Theorem 3 (Determinism correspondence)**.**
$(\texttt{deterministic}\ P)\ \wedge\ (\texttt{local\_determinitic}\ P)$
$\quad \Leftrightarrow (\texttt{deterministic}^+\ P)$

In figure 3, we update the set-oriented representation showing the correspondences. Note that we don't have a simple example for a non-reactive program which is deterministic with the standard semantics but not with semantics with paths.

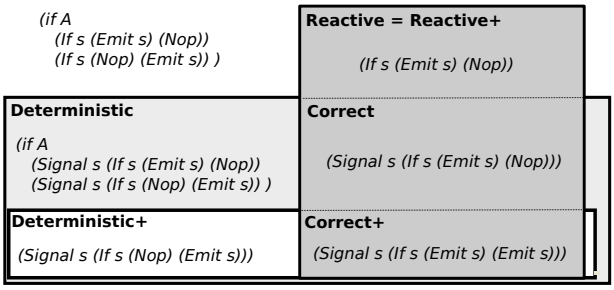


Figure 3: correspondence diagram

4 Discussions

4.1 Full renaming

In the presented semantics, paths are used to differentiate local signals from each other in order to name them from outside. Why not insure that all local signals have different names as a kind of Barendregt convention for the λ-calculus [4] ? By doing so, we need to formalize the subset of `ast` we have to use, characterising a notion of *well formed* terms. But the well-formedness property is not composable, for example if P_1 and P_2 are individually well formed, it may, when they define the same local signal, that $(\texttt{Par}\ P_1\ P_2)$ will not well-formed. Therefore, direct induction on program structure becomes impossible. In each premise, we have to insure that the main program is well formed and to keep a link from the main program to each sub-expressions. These links are paths.

Moreover, the renaming process will have to be applied between each instant, because the well-formedness property is not preserved by the `reduce` function. This is due to the duplication of loop's body. For example the well formed term $P = (\texttt{Loop}\ (\texttt{Seq}\ (\texttt{Pause})\ (\texttt{Signal}\ s\ (\texttt{Nop}))))$ is reduced to $(\texttt{Seq}\ (\texttt{Signal}\ s\ (\texttt{Nop}))\ P)$. This is known as schizophrenia phonemena [7]. Therefore the renaming process must be formally described and its cor-

[4] note that **De Bruijn index**es are not adequate because we may refer local names outside of their definitions

rectness have to be proved.

4.2 Path versus context

The first version of the semantics used contexts instead of paths (in the sources, you will see that variables denoting paths are generally named c, historically for *c*ontext).

Contexts was introduced by James H. Morris [4]. A context is a tree (an element of `ast` for our specification) where a *hole* take place of a specific sub-tree. Generally, contexts are used to specify, with the hole, where transformations/reductions would take place. For example, Esterel's context definition can be found in Olivier Tardieu's Thesis [5] (fig 2.4 p45) or for ReactiveML in Louis Mandel's thesis [3] (p66).

A path is a simplified context. Paths definition have the same number of constructors as contexts definition, but each constructor having less fields. A path is a direct projection (injection) of a context. So, if contexts are used in the constructive semantics (the rules which really decide the set of present signals), the correspondence with the logical semantics with paths will be easy.

4.3 On the fly renaming

Instead of doing, *a priori*, a full renaming (see section 4.1), Louis Mandel in his thesis ([3] Fig 3.4 p53) makes an on the fly renaming each time a local signal declaration is reached. If we restrict ReactiveML to the core of Esterel used here, we have a rule like :

$$\frac{n \ a \ fresh \ name \ \wedge \ p[s \leftarrow n] \xrightarrow[\rho,E]{E',k} p'}{(\texttt{Signal}\ s\ p) \xrightarrow[\rho,E]{E',k} p'}$$

$p[s \leftarrow n]$ creates a new term where each free occurrence of s is replaced by n. This strategy is appealing because, in one hand, it doesn't need to introduce the notion of well formed term discussed in 4.1, and, in other hand, it reflects what is done in a real interpreter : when a local signal declaration is evaluated, a piece of memory, containing all the information needed for this signal, is allocated. It is an axiom of the memory manager that the allocated memory is *fresh*, so the address (reference) of this memory can be used as the fresh name n introduced in the rule.

But two difficulties remains : how to formally define the *fresh* notion and how, from the outside, can we add or remove n from the present set E ? For the simple program (`Par` (`Signal` s p) (`Signal` s p)), we have to know *a priori* what will be the signal names generated for the local signal declarations. For the freshness, these two names have to be differents. Using the path, where the local signal declaration occurs, to generate the fresh name seems to be natural.

4.4 Determinism by rules

Using paths to define the logical semantics is a way to formally express the strong determinism property in Coq. The fact that the rules becomes deterministic must be view as a consequence[5]. If the strong correctness is not a goal and if we want that the rules exclude directly non-deterministic programs (instead of using an external property like definition 15), we can apply the strategy used by Olivier Tardieu in [6] which imposes that the premises in the two rules for local variable declaration are mutually exclusive :

$$\frac{p \xrightarrow[\rho,E\cup\{s\}]{E_1,k_1} p_1 \ \wedge \ p \xrightarrow[\rho,E-\{s\}]{E_2,k_2} p_2 \ \wedge \ s \in (E_1 \cap E_2)}{(\texttt{Signal}\ s\ p) \xrightarrow[\rho,E]{E_1-\{s\},k_1} (\texttt{Signal}\ s\ p_1)}$$

$$\frac{p \xrightarrow[\rho,E\cup\{s\}]{E_1,k_1} p_1 \ \wedge \ p \xrightarrow[\rho,E-\{s\}]{E_2,k_2} p_2 \ \wedge \ s \notin (E_1 \cup E_2)}{(\texttt{Signal}\ s\ p) \xrightarrow[\rho,E]{E_2,k_2} (\texttt{Signal}\ s\ p_2)}$$

This way, the relation $\rightarrow$ becomes a partial function. Following [6], if a program is correct with these rules, it will correct with the rules with paths. But the inverse is not true, as pointed in [6], the following program is correct with the rules with paths (the only solution is when both signals s_1 and s_2 are absent)

[5] The rules are deterministic when the $\rightarrow$ relation is a partial function (no more than one successor), the definition 5 is when the relation `coherent` is a partial function

while it is not with deterministic rules :
(`Signal` s_1
(`Signal` s_2
(`If` s_2
(`If` s_1 (`Emit` s_2) (`Nop`))
(`Nop`))))

Regarding the figure 3, these rules define a box inside the `correct`$^+$ one. Unfortunately, it seems that programs that are `correct`$^+$ but not correct with these rules cannot be simply characterized. [6] presents a theorem showing that these programs must have a non-reactive or non-correct$^+$ sub-term, but the property may be shared by a larger class of programs.

4.5 Reusability

The way paths are introduced, and the way how they are managed for local signals, in not dedicated to the semantics of the core of Esterel we have taken. Some branches of reactive programming initiated by Frédéric Boussinot[6], where a program cannot react instantaneously to absence of signals, can use the same kind of rules. The only difference with the rules of figure 1 is the second rule for `If` which becomes.

$$\frac{s \notin E}{(\texttt{If}\ s\ p\ q) \xrightarrow[\rho,E]{\emptyset,\texttt{TPause}} q}$$

We can propagate this modification in the functions `term`, `emit` and `reduce`, and the theorems are still valid with only 5 minor changes in the proofs. But the real question is : *does a path oriented semantics helps to prove other theorems ?*.

We have made a try with the central lemma of reactive programming ([3] Lemma 1, p 61) stating that all coherent programs have a minimal solution for the present set :

Lemma 1 (semilattice of coherency)**.**
$\forall P\ I\ O_1\ O_2$,
(`coherent`$^+$ $P\ I\ O_1) \Rightarrow$
(`coherent`$^+$ $P\ I\ O_2) \Rightarrow$
(`coherent`$^+$ $P\ I\ (O_1 \cap O_2)$)

[6] `http://www-sop.inria.fr/mimosa/rp/`

But this lemma is false due to the introduction of the couple **Trap**/**Exit**. One counterexample is :

```
Trap T0 in
   Trap T1 in
         present S1 then
            emit S1;
            exit T0
         end
      ||
         present S2 then
            emit S2;
            exit T0
         end
      ||
         exit T1
   end;
   emit S1;
   emit S2
end
```

Having `S1` or `S2`, or both, as present make the program coherent. In each case the `exit T0` overtake the `exit T1`. But when both signals are absent, the `exit T1` take place and the two pending emission are executed, thus violating the coherency.

It seems that *the feature of the (Boussinot's) semantics that guarantees that absence of a signal cannot generates some emission* ([3] p 61), can be stated only when `TPause` is the highest completion code. We have to change more deeply the semantics in order to achieve the previous lemma.

5 Lemmas

The theorem 1 is central. It serve to prove the other given theorems. This coherency correspondence must be proved by induction on the structure of the program. Therefore we have to make a correspondence under any path and consequently make a correspondence between all standard parameters of the standard semantics (mainly the set of present and emitted signals) and the arguments and results of the semantic functions (mainly the `emit` function).

These correspondences are given in Lemmas 9 and 10 in section 5.3, before that we have to introduce

some other definitions and lemmas.

5.1 Prefix partial order on paths

Many lemmas have to talk about local signal definitions *under* a given path. Since a local signal definition is also given with a path, paths must have a partial order. We will say that a path π_1 is less or equal to a path π_2 when π_1, as a list of edges, is a suffix of π_2. For example, if π_1 is the list $(g_2\ g_1)$[7] and if π_2 is the list $(g_4\ g_3\ g_2\ g_1)$, then $\pi_1 \leq \pi_2$. $\leq$ is a partial order, the paths (g_1) and (g_2) are not comparable when the two edges differ. More formally this order is defined with:

Definition 17 (partial order on paths)**.**
$\pi_1 \leq \pi_2 \Leftrightarrow$ **if** $(\pi_1 = \pi_2)$
then true
else match π_2 **with**
| **nil** $\Rightarrow$ **false**
| $(g{::}\pi_2) \Rightarrow \pi_1 \leq \pi_2$

We extend this partial order to compare a `psignal` to a path where global signals are less to any paths:

Definition 18 (psignal/path comparison)**.**
$p \leq \pi \Leftrightarrow$ **match** p **with** `Global` $\_ \Rightarrow$ **true**
| `Local` $\pi' \Rightarrow\ \pi' \leq \pi$

$\pi \leq p \Leftrightarrow$ **match** p **with** `Global` $\_ \Rightarrow$ **false**
| `Local` $\pi' \Rightarrow\ \pi \leq \pi'$

For simplicity, when there is no ambiguity, the term $a \leq b$ stands for the proposition $a \leq b = \texttt{true}$. As usual, $a < b$ stands for $(a \leq b\ \wedge\ a \neq b)$. Following Wikipedia, we will use the notation $a \perp b = a \leq b\ \vee\ b \leq a$ when a and b are comparable and the notation $a \parallel b = \neg(a \perp b)$ when a and b are incomparable.

The function `locals`, defined in definition 16, can redefined as: $(\texttt{locals}\ e^+) = \{p \in e^+ / \texttt{nil} \leq p\}$.

Paths reflect the tree structure of programs but we need to prove that there is only one path from a given node to the root. This property is achieve with the fact that the partial order is a *prefix* order:

Lemma 2 ($\leq$ is a prefix order (`c_le_prefix`))**.**
$\forall\ c\ a\ b,\quad a \leq c\ \wedge\ b \leq c \Rightarrow a \leq b\ \vee\ b \leq a$

For proving this lemma we have to switch to an alternative definition of the partial order

Lemma 3 ($\leq$ seen as suffix (`c_le_prop`))**.**
$\forall\ c_1\ c_2,\quad c_1 \leq c_2 \Leftrightarrow \exists l, l \bullet c_1 = c_2$

The other lemmas on the partial order, reflexivity, antisymmetry and transitivity are easily proved.

5.2 Signal conversion

The function $\uparrow_s^\pi$, introduced in section 3.3, given a path, convert a signal name to a `psignal`. We extend this function to set of names, which can be simply defined by:

$$\Uparrow_e^\pi = \{\uparrow_s^\pi / s \in e\}$$

Since main proofs make induction on the program structure (`ast`), we have to switch to an equivalent definition using a recursion on the path:

Definition 19. `fetch_def`
$\Uparrow_e^\pi\ \Leftrightarrow$ **match** π **with**
| **nil** $\Rightarrow (G^+\ e)$
| $((\texttt{SignalIn}\ s){::}\pi) \Rightarrow$
if $s \in e$
then $\Uparrow_{e-\{s\}}^\pi \cup \{(\texttt{Local}\ \pi)\}$
else $\Uparrow_e^\pi$
| $(\_{::}c) \Rightarrow \Uparrow_e^\pi$

The main lemma prove that `fetch_def` recursively defined on a path structure have the same meaning with the one recursively define on a set structure. This is done firstly by proving $\Uparrow_{e\cup\{s\}}^\pi = \Uparrow_e^\pi\ \cup\ \{\uparrow_s^\pi\}$ by induction on π using the definition 19 (lemma `fetch_def_add` in Coq) and by proving the correspondence between the two definitions by induction on the set e (lemma `fetch_def_corr` in Coq). The same lemma is needed for the set difference:

Lemma 4. `fetch_def_rem`
$\forall s\ \pi\ e,\quad \Uparrow_{e-\{s\}}^\pi = \Uparrow_e^\pi - \{\uparrow_s^\pi\}$

[7] Remember that the list gives the path from a node to the root

The proof needs that $\uparrow$ is injective for a given path, or, in other words, that only one local signal can be defined in a given path.

Lemma 5. `get_psignal_injectivet`
$\forall\ \pi,\ (\textbf{fun}\ s \rightarrow \uparrow_s^\pi)$ *is injective.*

In a similar way. we also need a function that convert a set of `psignal` to a set of names

Definition 20. `unfetch_def`
$\Downarrow_e^\pi\ \Leftrightarrow \textbf{match}\ \pi\ \textbf{with}$
$\quad \mid\ \textbf{nil} \Rightarrow (G^-\ e)$
$\quad \mid\ ((\texttt{SignalIn}\ s)::\pi) \Rightarrow$
$\qquad \textbf{if}\ (\texttt{Local}\ \pi) \in e$
$\qquad \textbf{then} \Downarrow_e^\pi \cup \{s\}$
$\qquad \textbf{else} \Downarrow_e^\pi - \{s\}$
$\quad \mid\ (\_::\pi) \Rightarrow \Downarrow_e^\pi$

As for $\Uparrow$ we has a lemma which rely $\Downarrow$ and $\uparrow$.

Lemma 6. `get_psignal_unfetch_def`
$\forall\ s\ \pi\ e,\quad s \in \Downarrow_e^\pi\ \Leftrightarrow\ \uparrow_s^\pi \in e$

We also need lemmas playing on the structure of the set argument.

Lemma 7. `unfetch_def_sing`
$\forall\ s\ \pi,\quad \Downarrow_{\{\uparrow_s^\pi\}}^\pi\ =\ \{s\}$

Lemma 8. `unfetch_def_union`
$\forall\ \pi\ e_1\ e_2,\quad \Downarrow_{e_1 \cup e_2}^\pi = \Downarrow_{e_1}^\pi \cup \Downarrow_{e_2}^\pi$

5.3 The two main lemmas

In one direction, we have to prove that, in a given path π, if the set argument of the functions `emit`, `term` and `reduce` is *locally* coherent, the standard rules can be applied. Before defining the meaning of *local coherency*, we say that two sets of path signals are equivalent under a given path π ($\bowtie_\pi$) when they coincide for all local signals under (greater than) π.

Definition 21 ($\bowtie_\pi$ (`same_inside`)).
$e_1 \bowtie_\pi e_2 \Leftrightarrow \{x \in e_1 / \pi \leq x\} = \{x \in e_2) / \pi \leq x\}$

A program p is said *locally* coherent under a path π with a set of present signal e, when the set of signals emitted by p under this path coincide, considering the equivalence relation $\bowtie_\pi$, with e .

Definition 22. `local_coherence`
$(\texttt{coherent}_{\mathcal{L}}\ p\ \pi\ e)\ \Leftrightarrow e \bowtie_\pi (\texttt{emit}\ p\ \pi\ e)$

Local coherency is what we need to switch from semantics with paths to standard semantics.

Lemma 9 (from path to standard)**.** `ter_st`
$\forall\ p\ \pi\ e,\quad (\texttt{coherent}_{\mathcal{L}}\ p\ \pi\ e)$
$\quad \Rightarrow \exists\ o,\quad p \xrightarrow[(c2\rho\ c),\Downarrow_e^\pi]{o,(\texttt{term}\ p\ \pi\ e)} (\texttt{reduce}\ p\ \pi\ e)$
$\qquad \wedge\ o = \Downarrow^c_{(\texttt{emit}\ p\ \pi\ e)}$

The local coherency precondition is enough to predict which one of the two standard rules for local signal declaration must be taken: in figure 1 the predicate $s \in E'$ can be checked looking in $\Downarrow_e^\pi$. The existential constructor ($\exists\ o \ldots$) in the conclusion of the lemma is needed since we have refer the set o constructed by the standard rules with a structural equality, the order in which elements are added determines this equality, the proposition $o = \ldots$ in the conclusion refers to the set equality.

In the other direction, if the standard rules can be applied, then we can extend the given present set of signal e to some set of path signals e^+ on which the functions `emit`, `term` and `reduce` will return the same results.

Lemma 10 (from standard to path)**.** `st_ter`
$\forall\ p\ \pi\ e\ k\ o\ p',\quad p \xrightarrow[(c2\rho\ c),e]{o,k} p'$
$\quad \Rightarrow \exists\ l^+,\ l^+ = \{x \in (\texttt{emit}\ p\ \pi\ e^+)/\pi \leq x\}$
$\qquad \wedge\ k = (\texttt{term}\ p\ \pi\ e^+)$
$\qquad \wedge\ o = \Downarrow^c_{(\texttt{emit}\ p\ \pi\ e^+)}$
$\qquad \wedge\ p' = (\texttt{reduce}\ p\ \pi\ e^+)$
$\quad \textbf{where}\ e^+ = \Uparrow_e^\pi \cup l^+$

Note that the lemma give in l^+ only the present (and emitted) local signals occurring in p. Globals signals and local signals already declared in the path π are computed with $\Uparrow_e^\pi$. Since the sets $\Uparrow_e^\pi$ and l^+ are disjoins, we also have $(\texttt{coherent}_{\mathcal{L}}\ p\ \pi\ e^+)$, so we can reformulate this lemma in a way which look like the reverse of the lemma 9. As described here, the lemma 10 is more precise and simpler to prove.

The two lemmas 9 and 10 are proved by induction on the structure of p. The proofs need some properties of the functions `emit`, `term` and `reduce` which

are independent of the standard rules. We give these properties in a separate subsection.

5.4 Properties of the semantics functions

The proof of the lemma 9 essentially needs that its local coherency precondition can be followed by the induction. The following lemma is quickly needed :

Lemma 11. `same_inside_on_fork`
$\forall\ p_1\ g_1\ p_2\ g_2\ \pi\ e,$
$\quad e \bowtie_c (\texttt{emit}\ p_1\ (g_1 :: \pi)\ e) \cup (\texttt{emit}\ p_2\ (g_2 :: \pi)\ e)$
$\quad \wedge\ g_1 \neq g_2$
$\Rightarrow (\texttt{coherent}_{\mathcal{L}}\ p_1\ (g_1 :: \pi)\ e)$

This lemma is applied for statement with two sub-statements, for example with $g_1 =$ `ParRight` and $g_2 =$ `ParLeft`. The proof of this lemma needs the fact that all the elements in the result of `emit` , applied to a path π, are comparable to π.

We have also to prove than `emit` cannot return a *hidden* signal. For a statement (`Signal` s (`Signal` $s\ P$)), the sub-statement P can only access to the inner signal s, the external signal s is said *hidden*. In other word we have to prove that $\downarrow_s^\pi \notin (\texttt{emit}\ p\ ((\texttt{SignalIn}\ s) :: \pi)\ e)$.

All result's emit properties are resumed in the following lemma:

Lemma 12 (emit properties)**.** `emit_prop`
$\forall\ x\ p\ \pi\ e,\quad x \in (\texttt{emit}\ p\ \pi\ e) \Rightarrow$
$\quad (\pi < x\ \wedge\ \exists\ \pi_i\ s,\ \uparrow_s^{\pi_i} = x)$
$\quad \vee\ (x = (\texttt{Local}\ \pi)\ \wedge\ \exists p'\ s,\ p = (\texttt{Signal}\ s\ p'))$
$\quad \vee\ (x < \pi\ \wedge\ \exists\ s,\ \uparrow_s^c = x)$

In other words, all free signals emitted by p in a path π are comparable to π and reachable via $\uparrow$.

In an other direction, some signals declared present have no effect on the emitted signals. For example a hidden signal cannot be emitted since it is not reachable by $\uparrow$. Since the `emit` function use the function `term` for sequences, we have to prove before, that these useless signals have also no effect on `term`'s results. This give the following lemma :

Lemma 13. `term_useless_hidden`
$\forall\ p\ e\ u\ s\ \pi,\quad u \leq \uparrow_s^c \Rightarrow$
$(\texttt{term}\ p\ \pi\ e) = (\texttt{term}\ p\ \pi\ (e - \{\uparrow_s^u\}))$

If we apply this lemma with a path of a local signal declaration, we got the following lemma which is necessary for the proof of the main lemma 10 and is related to modification of emitted signals ($E \cup \{s\}$ and $E - \{s\}$) in the standard rules.

Lemma 14. `term_hidden`
$\forall\ s\ p\ \pi\ e,\quad (\texttt{term}\ p\ ((\texttt{SignalIn}\ s) :: \pi)\ e)$
$= (\texttt{term}\ p\ ((\texttt{SignalIn}\ s) :: \pi)\ (e - \{\uparrow_s^\pi\}))$

The main lemma 10 is proved by induction on the structure of the program. For a statement like (`Par` $p_1\ p_2$), in a path π, by induction we got $k_1 = (\texttt{term}\ p_1\ (\texttt{ParLeft} :: c) \Uparrow_e^c \cup l_1^+)$ and we have to prove that $k_1 = (\texttt{term}\ p_1\ (\texttt{ParLeft} :: c) \Uparrow_e^c \cup (l_1^+ \cup l_2^+))$ where l_1^+ and l_2^+ are local signals emitted by p_1 and p_2. So we have to prove that the internal signals emitted by p_2 can be considered as present and has no effet on the computation of (`term` $p_1 \ldots$). This is depicted by the following lemma :

Lemma 15. `term_useless_incomparable`
$\forall\ p\ \pi_1\ \pi_2\ e_1\ e_2,\quad \pi_1 \parallel \pi_2 \Rightarrow$
$(\texttt{term}\ p\ \pi_1\ (e_1 \cup \{x \in e_2/\pi_2 \leq x\})) = (\texttt{term}\ p\ \pi_1\ e_1)$

The three previous lemmas have to be reformulated for the two other functions `emit` and `reduce`.

6 Conclusions

We have specified a new logical semantics for Esterel by adding a path to local signals. The rules of this new semantics become deterministic and thus are fully constructive. Even if the change seems to be minor, the full specification and proofs in Coq take 3600 lines of code.

We hope that the simplification done for local signals in the semantics, the specification and part of the proofs would be reused for further work.

References

[1] Pejman Attar. *Towards a safe and secure synchronous language*. These, Université Nice Sophia Antipolis, December 2013.

[2] G. Berry. *The Constructive Semantics of Pure Esterel Draft Version 3*. 2002.

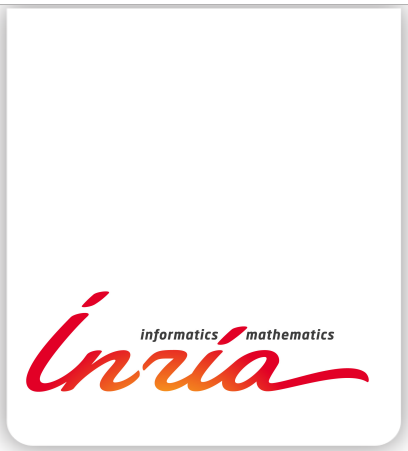
[3] Louis Mandel. *Conception, Sémantique et Implantation de ReactiveML : un langage à la ML pour la programmation réactive*. PhD thesis, Université Paris 6, 2006.

[4] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[5] Olivier Tardieu. *Loops in Esterel: From Operational Semantics to Formally Specified Compilers*. Theses, École Nationale Supérieure des Mines de Paris, September 2004.

[6] Olivier Tardieu. A deterministic logical semantics for pure esterel. *ACM Trans. Program. Lang. Syst.*, 29(2), April 2007.

[7] Olivier Tardieu and Robert de Simone. Curing schizophrenia by program rewriting in esterel. In *MEMOCODE*, pages 39–48. IEEE, 2004.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399