A Higher-Order Extension for Imperative Synchronous Languages
Eric Vecchié, Jean-Pierre Talpin, Sébastien Boisgérault

To cite this version:
Eric Vecchié, Jean-Pierre Talpin, Sébastien Boisgérault. A Higher-Order Extension for Imperative Synchronous Languages. SCOPES 2010, Jun 2010, France. hal-00533953

HAL Id: hal-00533953
https://hal-mines-paristech.archives-ouvertes.fr/hal-00533953
Submitted on 8 Nov 2010
A Higher-Order Extension for Imperative Synchronous Languages

Eric Vecchié  
Mines ParisTech  
60 Bd St Michel  
FR-75272 Paris Cedex 06  
Eric.Vecchie@ensmp.fr

Jean-Pierre Talpin  
INRIA Rennes - Bretagne Atlantique  
Campus de Beaulieu  
FR-35042 Rennes Cedex  
Jean-Pierre.Talpin@irisa.fr

Sébastien Boisgérault  
INRIA Rennes - Bretagne Atlantique  
Campus de Beaulieu  
FR-35042 Rennes Cedex  
Sebastien.Boisgerault@ensmp.fr

ABSTRACT

This article presents the very first effective design of higher-order modules in the synchronous programming language Esterel. Higher-order modules, together with the robust separate compilation scheme that implements it, allow us to address a yet unexplored application spectrum ranging from rapid prototyping of embedded functionality to hot reconfiguration of embedded software within the formal modeling framework of the “synchronous hypothesis”. While extensions of data-flow synchronous languages had already been proposed for Lustre [11] and Signal [25], the adaptation of similar programming concepts to imperative synchronous frameworks like Esterel has long posed major technical challenges, due to the specificity of its model of computation. We present a framework including a formal semantics, a type system, and a modular code generator, that tackle this challenge. We consider a specific stack-based module call convention and a simple event pooling protocol; in consequence signals can refer to modules and modules can be transmitted and instantiated by referencing a signal. We define a type system that computes the potential emissions of a module and prove it sound. Our type system seamlessly fits an extension of Esterel’s constructive semantics with higher-order modules.

1. INTRODUCTION

Synchronous programming languages are domain-specific languages dedicated to the design of real-time embedded systems. Their semantics are based on the synchronous hypothesis stating that the execution of a program is discretely divided into atomic reactions. Enjoying efficient compilation methods, verification and automatic distribution tools, these languages are now employed in various industrial design processes (avionics, nuclear plants, ...). In this domain, the imperative style of Esterel [5] has been specifically designed to address the problem of the constructiveness in the context of a modular execution scheme. Indeed, modularity in the sense of a separate compilation and execution of modules is a major issue in Esterel and also the key for the introduction of higher-order. For this purpose, we seamlessly extend the constructive semantics of Esterel with a single rule expressing the instantiation of a module from the value of a signal. To deal with constructiveness issues, we then define and show the correctness of a type system which allows us to know the potential emissions of a higher-order module.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SCOPES’10, June 28-29, 2010, St. Goar, Germany  
Copyright © 2010 ACM 978-1-4503-0084-1/10/06 ...$10.00.
main drawback of such approaches is the lack of high-level constructs and fine control on the behavior of the running program. As an example, let us consider a module $M$ that has to be dynamically replaced by another module $M'$. Let us suppose that $M'$ requires an initialization phase running over several reactions (or suppose that $M$ requires a finalization phase to reach a stable state). If $M'/M$ is a module responsible for some critical task and must not be interrupted, then $M$ must keep on running until $M'$ becomes fully operational. This simple example illustrates the fact that dynamic reconfiguration can rely on some strong interactions with the running program. Middleware solutions can then be arbitrarily complicated or even impossible to program whereas a higher-order based designs remain simple.

1.2 Modularity
The introduction of higher order in synchronous languages is made possible by the separate compilation and execution of synchronous modules. We start from the very same notion of modular compilation as in general-purpose languages like C: the translation of an Esterel module into binary code with a header file describing its interface. The instantiation of a module is simply realized by passing its parameters and executing its code through simple calling conventions. Our implementation is based on the key idea of defining concurrent coroutines and on a minimalist runtime system to execute separately compiled modules.

**Early Work.** Because it was thought to make causality analysis more difficult, modular compilation was not considered in early implementations of Esterel. Instead, early Esterel compilers generate code from a data-structure representing a flattened and inlined expansion of the source program. For instance, [6] translates a program into an extended finite state machine whose transitions are decorated with code fragments. The obvious disadvantage of this technique is the potential state-space explosion of the automaton. Its main advantage is the very short execution time of the generated code. Compilation techniques achieving a polynomial space complexity were first obtained by using systems of equations to symbolically represent the automaton of a program [4]. This approach was successfully used for hardware synthesis and it is still at the core of commercial tools, although the generated software is sometimes slower. Another approach is based on the translation of programs into concurrent control flow graphs [15, 21] whose sizes is in linear proportion of the given program. At each instant, the control flow graph is visited until active nodes are found to trigger the execution of the corresponding sub-tree. The approach followed by the Saxo-RT compiler [12] translates the program into an event graph.

**Recent Work.** Some works [24, 26] address different notions of “modular compilation”. In [26], the generated code is able to partially deal with undefined inputs by using a three-valued logic. The generated program tries to compute as many outputs as it can while ignoring inputs that are still unknown. In [24], the Esterel dialect Quartz is translated into a “job language”. This translation can be conceptually regarded as a graph of simultaneously active jobs where each job contains an atomic task. In these approaches, the increased effort involved for the separate compilation of modules separately does not spare “tuning” the generated code where it is used and depending on the calling context. Unlike our approach, it implies that the code of instantiated modules must be duplicated. Some recent works [18] use multi-function interfaces to generate modular code for synchronous block diagrams. Each function in the interface is responsible for the evaluation of some outputs through the evaluation of the relevant part of the module. The trade-off between modularity (decreasing with the number of functions in the interface) and reusability (increasing with the number of functions) is also addressed. Earlier works on Signal and Cronos [3, 16] actually use very similar techniques of clustering, interface synthesis and scheduling to perform modular compilation. In our approach, modularity is maximized since the compilation of each module generates a single function. Reusability is nevertheless ensured by a flexible model of execution which allows causality issues to be resolved dynamically at run-time.

1.3 Syntax of Esterel
An Esterel module consists of an interface where input and output signals are defined, followed by a body. Syntax of program statements is provided by the following simple grammar:

$$
P ::= \text{pause} \mid S := E \mid P ; P \mid P \parallel P \mid \text{loop } P \text{ end} \mid \text{signal } S \text{ in } P \text{ end}\mid \text{if } E \text{ then } P \text{ else } P \mid \text{abort } P \text{ when } E \mid \text{suspend } P \text{ when } E
$$

with $S$ ranging over signals and $E$ ranging over expressions. Naive semantics of Esterel goes as follows: programs behaviors are discretely divided into instants or reactions. Control threads are executed until reaching a \text{pause} statement, which is the main statement which cuts behaviors into atomic instants. In a reaction cycle, input signals are read/sampled, and internal computation takes place until output signals are emitted in answer, and the program state is progressed. Instants are based on a common logical clock, which paces all parallel threads. Of course in a reaction various parallel threads do not run independently, as they may synchronize and affect one another causally (hardware people would say “combinationally”). When control reaches a \text{present}(S) boolean condition (testing signal presence at the current instant), it may have to postpone execution until a consistent definitive value and status are obtained for the signal inside the current reaction (either because it is emitted somewhere in parallel, or because other threads of execution progressed to a point where provably all potential emissions were discarded).

**Example.** Dynamically reconfigurable systems can be expressed in the syntax of Esterel as in the following example:

```plaintext
module RECONF(M : mod(...), params...)
loop
  abort
  run (? M)(params...); halt
  when present(M)
end
end module
```

where “halt” is syntactic sugar for “loop pause end”. In this example, a module $M$ is given as input of the module RECONF. The value of the module to be executed is carried by an input signal. This module is started/restarted each time the signal is emitted (each time its value changes). The module RECONF can thus be executed in parallel to another one responsible for choosing the module to be executed.

The paper is organized as follows: section 2 describes the considered constructive semantics of Esterel. The section 3 gives the semantic rules for the introduction of higher-order modules and section 4 describes the type system we use to deal with constructiveness issues in Esterel. The design of our modular compiler is then described in section 5. Section 6 provides some benchmarks and we finally conclude this article by some future works.
2. THE SEMANTICS OF ESTEREL

We consider the logical behavioral semantics of Esterel given in [8] (refer to these works for further details). This semantics is equivalent to the "official" semantics given in [5] but relies on a tiny but sufficient simple subset of the language where loops, preemption and suspension are discarded. In essence, loops are captured by the semantics of the sequence and preemption and suspensions are captured by the one of the test. Furthermore, constructiveness issues are more clearly addressed in this formalism. A process is interpreted by a series of transitions whose triggers are governed by rules of structured operational semantics. A run:

\[ P \frac{O_1}{T_1} P_1 \ldots P_{n-1} \frac{O_n}{T_n} P_n \]
depicts the successive reactions of a process \( P \) to the inputs \( I_1 \ldots I_n \) producing the outputs \( O_1 \ldots O_n \). The meaning of a transition is defined by an auxiliary chain of unconditional reactions subject to SOS rules:

\[ P \frac{O}{T} Q \Rightarrow P, I \downarrow P_1, E_1 \downarrow \ldots \downarrow P_m, E_m \uparrow k Q, I \cup O \]

where \( k \in \{0,1\} \). The status \( \downarrow \) means that the execution of the current reaction is not completed. At the end of a reaction, the status \( k \) is either 0 to mean stopped or 1 for waiting. We call \( P, Q \) and rules [5]:

- Nothing:
  \[ \text{nothing}, E \xrightarrow{\text{nothing}} E \]

- Parallelism:
  \[ \text{nothing}, E \xrightarrow{\text{nothing}} E \]

- Assignment (emission of values). As we consider valued signals here and for simplification issue, we only consider programs where each signal is emitted once in each atomic reaction:
  \[ s' = v, E \cup \{s'_v\} \xrightarrow{\text{nothing}} E \cup \{s'_v\} \]

where \( v \) is the value and \( b \) is the status (\( + \) for present, \( - \) for absent or \( \bot \) for undefined) of a signal \( s'_v \).

- Sequence:
  \[ P, E \xrightarrow{\delta} P', E' \quad k \neq 0 \]
  \[ P; Q, E \xrightarrow{\delta} P'; Q, E' \]

- Tests:
  \[ s^+_v \in E \]
  \[ \text{if } s \text{ then } P \text{ else } Q, E \xrightarrow{\delta} P, E \]
  \[ s^-_v \in E \]
  \[ \text{if } s \text{ then } P \text{ else } Q, E \xrightarrow{\delta} Q, E \]

- Parallelism:
  \[ P, E \xrightarrow{\delta} P', E' \]
  \[ P \parallel Q, E \xrightarrow{\delta} P', E' \]

- Local signals:
  \[ \text{signal } s'_v \text{ in } P, E \xrightarrow{\delta} \text{ signal } s'_v \text{ in } P', E' \]
  \[ \text{signal } s'_v \text{ in } P, E \xrightarrow{\delta} \text{ signal } s'_v \text{ in } P', E' \]

\[ P \parallel Q, E \xrightarrow{\delta} P', E' \]

where \( P(\Pi(b,P)) \) depends on the status of the signal \( s \) or on \( \pi(P) \), the set of potentially emitted signals in \( P \):

\[ \Pi(b,P) = \{ s^+_v \text{ if } b = \top \text{ and } s \notin \pi(P) \}
\]

2.1 Constructiveness

Several variations of the constructive semantics of Esterel are presented in [8]. The semantics can thus be more or less restrictive depending on the way the so-called “Potential Function” \( \pi(P) \) is defined. We consider the v3 version of the constructive semantics where \( \pi(P) \) is defined by:

- Nothing, pause:
  \[ \pi(\text{nothing}) = \pi(\text{pause}) = \emptyset \]

- Assignment:
  \[ \pi(s:=v) = \{ s \} \]

- Sequence:
  \[ \pi(P; Q) = \{ \pi(P) \cup \pi(Q) \text{ if } \gamma(P) \}
\]

\[ \pi(P; Q) = \{ \pi(P) \text{ otherwise} \]

- Tests:
  \[ \pi(\text{if } s \text{ then } P \text{ else } Q) = \pi(P) \cup \pi(Q) \]

- Parallelism:
  \[ \pi(P \parallel Q) = \{ \pi(P) \cup \pi(Q) \text{ if } k = l = \top \]

\[ \pi(P) \text{ if } k = \bot \text{ and } l = \bot \]

\[ \pi(Q) \text{ if } k = \bot \text{ and } l = \bot \]

\[ \emptyset \text{ otherwise} \]

- Local signals:
  \[ \pi(\text{signal } s'_v \text{ in } P) = \pi(P) \setminus \{ s \} \]

The predicate \( \gamma(P) \) is true if the execution of \( P \) can be instantaneous. We have:

- Nothing, assignment:
  \[ \gamma(\text{nothing}) = \gamma(s:=v) \]

- Sequence:
  \[ \gamma(P; Q) = \gamma(P) \]

- Tests:
  \[ \gamma(\text{if } s \text{ then } P \text{ else } Q) = \gamma(Q) \]

\[ Q, E \xrightarrow{\delta} Q', E' \]

\[ P_k \parallel Q, E \xrightarrow{\delta} P_k \parallel Q', E' \]

\[ k \neq \bot \quad l \neq \bot \]

\[ P_k \parallel Q, E \xrightarrow{\max(k,l)} P \parallel Q, E \]

- Local signals:
  \[ P, E \cup \{ (s'_v, P) \} \xrightarrow{\delta} P', E' \]

\[ P, E \cup \{ (s'_v, P) \} \xrightarrow{\delta} P', E' \]

\[ P, E \cup \{ (s'_v, P) \} \]

where \( \Pi(b,P) \) depends on the status of the signal \( s \) or on \( \pi(P) \), the set of potentially emitted signals in \( P \):

\[ \Pi(b,P) = \{ s^+_v \text{ if } b = \top \text{ and } s \notin \pi(P) \}
\]

\[ s^+_v \text{ otherwise} \]
• Parallelism:

$$\gamma(P \parallel Q) = \frac{\gamma(P)}{\gamma(P_0 \parallel Q)}$$

Causality Issues. Higher-order offers a way to transmit modules on signal, including modules that are written after the main program has been started. Causality in the context of higher-order is not a static issue. In our approach, causality issues are therefore resolved at run-time. Our major concern is the correct execution of the programs, that is the correct ordering of instructions. Let us consider the following example:

1. if present(I) then emit S1;
2. if present(S2) then emit 0
3. if present(S1) then emit S2

The correct execution of this program starts with the beginning of the first thread (at line 1), then carry on to the second one (at line 3) and finally resume the first one (at line 2). This program is a friendly illustration of the problem. In real life we might face some nastier configurations where such three conditional statements are nested in different higher-order modules. Section 5 provides practical solutions to this problem.

In Esterel, the composition of two correct modules may generate a causally incorrect program. This is typically the case in the following program:

```plaintext
present(S) then nothing else emit S
```

where S is absent if it emitted and present if it is not: this is incorrect in Esterel. In our approach, this problem is related to the presence of deadlocks in the program. Code generation and rejection of causally incorrect programs should be independent tasks. The detection of causally incorrect programs is a major concern that deserves a dedicated article but it is not in the scope of this one: in our approach, we guarantee the correct execution of higher-order programs under the hypothesis that the programs are causally correct. In our compiler however, deadlocks are easily detected at run-time.

4. TYPE SYSTEM

As a motivating example, let us consider the following simple Esterel program:

```plaintext
module MOD(M, S1, S2)
  if present(M) then
    if present(S1) then emit S1 else
      if present(S2) then emit 0
      end if
    end if
  end module
```

the type expected for the signal M where the flag “Π” indicates that the signal can be emitted immediately. We also have subtyping since a module of type:

```plaintext
mod(sig, Π sig)
```

could also be received on M without problem. Let us now consider the following examples:

```plaintext
module A0(s: sig)
  pause; emit s
end module

module A1(s: Π sig)
  emit s; pause
end module
```
The type of \( A_0 \) is “smaller” than the type of \( A_1 \) in the sense that you can use \( A_0 \) each time you expect \( A_1 \).

```plaintext
module B0(a: mod(sig), s: sig)
run (? a)(s)
end module

module B1(a: mod([I] sig), s: sig)
pause; run (? a)(s)
end module
```

We have \( B_1 \sqsubseteq B_0 \) (Contravariance).

```plaintext
module C0(a: mod(sig))
a := A0
end module

module C1(a: mod([I] sig))
a := A1
end module
```

We now have \( C_0 \sqsubseteq C_1 \) (Covariance). In the case of \( B_0 \) and \( B_1 \), signal \( a \) is used as an input: the subtyping relation is contravariant. In the case of \( C_0 \) and \( C_1 \), signal \( a \) is used as an output: the subtyping relation is covariant. In our type system, we then choose to make the distinction between the input type and the output type of signals.

### 4.1 Definition

As we want to allow subtyping, our type system has the structure of a lattice and the type \( t \) of a module is defined by:

\[
t ::= m \mid \top \mid \bot \\\nm ::= [I] \ mod([I] \ s, \ldots [I] \ s) \\\ns ::= \ sig \ of(\text{in} : t, \text{out} : t)
\]

where \( \Gamma \) and \( \Pi \) are the markers indicating respectively that a module can be instantaneous \( (\Gamma) \) and that a signal is possibly immediately emitted \( (\Pi) \). \( \top \) and \( \bot \) are respectively the greatest and the least module type such that \( \forall f, t \sqsubseteq \top \) and \( \bot \sqsubseteq t \). The parameters of a module are signal where value can flow from caller to callee and vice versa. For a matter of formalization, the type of a signal is then divided into its input type and its output type. This can also be seen as an assume/guarantee pair where:

1. the assumption is made that any value read on the signal is “smaller” than its input type,
2. the guarantee is given that any value emitted on the signal is “smaller” than its output type.

The subtyping relation between signals \( S \sqsubseteq S' \) is covariant in the input and contravariant in the output. If we suppose the types \( S \) and \( S' \) to be:

\[
S = \ sig \ of(\text{in} : I, \text{out} : O) \\
S' = \ sig \ of(\text{in} : I', \text{out} : O')
\]

Then we have:

\[
S \sqsubseteq S' \iff I \sqsubseteq I' \text{ and } O' \sqsubseteq O
\]

This distinction between the input type and the output type of a signal does not mean that these types can be completely different. Indeed, for any signal of type:

\[
sig \ of(\text{in} : I, \text{out} : O)
\]

there exists the implicit subtyping relation \( O \sqsubseteq I \) as we show in section 4.3. If the type of a signal does not respect this constraint then this signal will be impossible to use concretely without breaking the type correctness of the program.

defined in section 4.2. In this system, an input signal written “input \( s : I \)” in the concrete Esterel syntax has then a type of the form:

\[
sig \ of(\text{in} : I, \text{out} : \bot)
\]

An output signal written:

\[
output \ s : O
\]

has a type of the form:

\[
sig \ of(\text{in} : \top, \text{out} : O)
\]

The subtyping relation for module types is contravariant. If we suppose the module types \( M \) and \( M' \) to be:

\[
M = i \ mod(p_1, \ldots p_n, T_n) \\
M' = i' \ mod(p'_1, T'_1, \ldots p'_n, T'_n)
\]

We then have \( M \sqsubseteq M' \) if and only if:

\[
(i = \Gamma) \Rightarrow (i' = \Gamma) \quad \text{and} \quad \forall k \in [1..n] \ (p_k = \Pi) \Rightarrow (p'_k = \Pi) \quad \text{and} \quad T'_k \sqsubseteq T_k
\]

The main advantage of using two different types for signals is illustrated by the following example where a module \( M \) using one formal parameter is declared:

```plaintext
module M(s: sig of(\text{in} : I, \text{out} : O))
    ...
end module
```

The couple of types \( I \) and \( O \) defines a range for possible use cases for \( M \) with our type system, it is then possible to instantiate the module \( M \) with any signal of type \( T \) (i.e. \( \text{sig} of(\text{in} : T, \text{out} : T) \)) as soon as \( T \) verifies \( O \sqsubseteq T \sqsubseteq I \). This precision would be impossible to obtain with a unique type.

With such type system, the definition of the potential function for higher-order module instantiation can be refined. Thus, if the type of a signal \( s \) is:

\[
s : \ sig \ of(\text{in} : I, \text{out} : O)
\]

then we have:

\[
\pi(\text{run}(s)(v_1, \ldots v_n)) = \{v_i \mid (p_i = \Pi)\} \\
\gamma(\text{run}(s)(v_1, \ldots v_n))
\]

The types \( T_1, \ldots, T_n \) of the parameters are supposed to be given by the programmer but the flags \( i \) and \( p_1, \ldots, p_n \) could be inferred easily. In future works we intend to infer more type information from the structure of the program and from a static analysis of its dependencies.

### 4.2 Type Correctness

Let \( \mathcal{C}(D, P) \) be the predicate indicating that a program \( P \) is correctly typed given the environment \( D \). An environment \( D \) is a list of declarations relating names \( x \) to types \( T \). We write \( D(x) \) for the type \( T \) associated with the name \( x \) in the environment \( D \) containing the declaration \( [x : T] \):

\[
D ::= \epsilon | D, [x : T]
\]

By definition of \( \mathcal{C}(D, P) \), the following properties hold:

- Nothing, pause:

\[
\mathcal{C}(D, \text{nothing}) \quad \mathcal{C}(D, \text{pause})
\]
• Sequence:
  \[ \frac{C(D, P) \quad C(D, Q)}{\overline{C}(D, P ; Q)} \]

• Tests:
  \[ \frac{C(D, P) \quad C(D, Q)}{\overline{C}(D, \text{if } s \text{ then } P \text{ else } Q)} \]

• Parallelism:
  \[ \frac{C(D, P) \quad C(D, Q)}{\overline{C}(D, P \parallel Q)} \]

• Assignment. If we have:
  \[ D(s) = \text{sig of} \langle \text{in} : l, \text{out} : O \rangle \]
  \[ D(v) = I' \quad \text{or} \quad D(v) = \text{sig of} \langle \text{in} : l', \text{out} : O' \rangle \]
  then:
  \[ I' \subseteq O \quad \overline{C}(D, \text{.sig} = v) \]

• Local signals. If we have:
  \[ D' = D, \langle s : \text{sig of} \langle \text{in} : T, \text{out} : T \rangle \rangle \]
  then:
  \[ \overline{C}(D', P) \quad \overline{C}(D, \text{signal} s : T \in P) \]

• Module instantiation. If we have:
  \[ D(s) = \text{sig of} \langle \text{in} : l, \text{out} : O \rangle \]
  \[ I = i \mod(p_1 T_1, \ldots, p_n T_n) \]
  \[ D(v_k) = T_k' \]
  then:
  \[ \forall k \in [1..n] \quad T_k' \subseteq T_k \quad \overline{C}(D, \text{run} (?s)(v_1, \ldots, v_n)) \]

• Module declaration. If we have:
  \[ D(m) = i \mod(p_1 T_1, \ldots, p_n T_n) \]
  \[ D' = D, \langle s_1 : T_1, \ldots, s_n : T_n \rangle \]
  then:
  \[ \forall k \in [1..n] \quad s_k \in \tau(P) \Rightarrow (p_k = \Pi) \quad \overline{C}(D', P) \]
  \[ \overline{C}(D, \text{module} m (s_1 : T_1, \ldots, s_n : T_n), P) \]

4.3 Soundness
As for the semantics, the type system is sound if all signals that are emitted are indeed detected:
\[ \forall D : \overline{C}(D, P) \quad \text{and} \quad P, E \triangleleft P', E' \]
\[ \Rightarrow \overline{C}(D, P') \quad \text{and} \quad \forall s_k^E \in E' - E \quad s \in \tau(P) \]
This property were proven for the standard Esterel semantics in [8]. The introduction of higher-order does not change the structure of this proof but we have to prove additionally that a signal used for the instantiation of a module contains no “spurious” value. Concerning the type system, this comes down to say that all the potential emissions of a higher-order module being instantiated are indeed captured by the type system. In other words, we have to show that, given an environment for signal values \( E \) and an environment of declarations \( D \), then the instantiation of a higher-order module \( \text{run} (?s)(v_1, \ldots, v_n) \) always verifies:
\[ D(s) = \text{sig of} \langle \text{in} : l, \text{out} : O \rangle \]
\[ I = i \mod(p_1 T_1, \ldots, p_n T_n) \]
\[ s_k^E \in E \quad \text{and} \quad v = \text{module} (s_1, \ldots, s_n), . P \]
\[ \gamma(P) \Rightarrow (i = \Gamma) \]
\[ \forall k \in [1..n] \quad s_k \in \tau(P) \Rightarrow (p_k = \Pi) \]

It is then sufficient to prove that, if a program \( P \) is correctly typed according to \( \overline{C}(D, P) \), then the type of \( v \) is always “smaller” than the type \( I \). We then have to prove by structural induction that for any signal \( s \), any value emitted on \( s \) is “smaller” than any type expected on \( s \). As a remark, we only consider “correct” programs where signals are always assigned before being read.

**Sketch of Proof.** We prove by induction that for any signal \( s \), there exists a type \( T_0 \) such that \( T_0 \) is “smaller” than any expected type and “bigger” than any value emitted on \( s \):

• The trivial case arise in the context of a local declaration:
  \[ \text{signal} \ s : T \in P \]
  In this case, the type system says:
  \[ D(s) = \text{sig of} \langle \text{in} : l, \text{out} : O \rangle \]
  \[ D(\text{run} (?s)) = i \mod(p_1 T_1, \ldots, p_n T_n) \]
  First, the property that the value of \( m \) is correct with respect to the ample types has to and can itself be proven by structural induction: more generally, the proof of the property must always consider signals of type \( \text{mod}(T) \) before signals of type \( T \). As recurrence hypothesis, we then suppose we have \( T_0 \) such that \( T_0 \subseteq T \) and \( O \subseteq T_0 \).

• By structural induction, a signal \( s \) can be passed as parameters to another module \( \text{run} (?m)(s) \). The type of \( s \) and \( m \) are supposed to be:
  \[ D(s) = \text{sig of} \langle \text{in} : l, \text{out} : O \rangle \]
  \[ D(\text{run} (?m)) = i \mod(p_1 T_1, \ldots, p_n T_n) \]
  \[ D(s) \quad \text{and} \quad D(\text{run} (?m)) \]
  Then the type correctness rules imply that:
  \[ T_0 \subseteq T \quad \text{and} \quad I \subseteq I_0 \quad \text{and} \quad O \subseteq O_0 \]
  We finally can prove that the property is also verified by structural induction:
  \[ T_0 \subseteq I_0 \quad \text{and} \quad O \subseteq O_0 \]

5. OUR COMPILER
We implemented a lightweight compiler able to deal with higher-order Esterel modules and complying with the constructive semantics given in section 2. The compilation method tries to mimic the well-known code generation techniques described in classical compilation books, like the well-known Dragon Book [1]. Expressions are flattened and translated into assembly code sequences. Conditional statements are replaced by conditional gotos and function calls use the
stack for passing the parameters and saving the register context of the caller.

In our approach, the execution scheme is inspired by the reactive kernel of Junior [17]. This tool is a relative of the Esterel language used to write reactive applications through a Java API and featuring a delayed reaction to absence. In Junior, each reaction step executes and reduces a tree representing the instantaneous state of the reactive program and reflecting its original structure. We use a mechanism similar to coroutines [14] so that the control yields at some points of the generated program. These coroutines are then hierarchically nested to reflect the control structure of the source program.

Runtime System. A Runtime System maintains a collection of cooperative threads to be executed during the current logical instant (reaction) and a second one to be executed at the next logical instant. The “current” list of threads can be dynamically enlarged. This is typically the case when a thread reaches a parallel statement. Thus, the execution of a reaction step means the execution of all the threads of the current list until it is made completely empty. The same way, the execution of a synchronous program means the execution of reaction steps until the “next” list of threads is empty. The second task of the Runtime System is also to deal with the signal emissions and resets but we shall come back to this later.

Our compiler generates sequential C code in assembly style (computed gotos, explicit stack manipulation,...). Each module in the input language is translated into a C function that can be executed through the Runtime System. Code generation rules are given in the following.

5.1 Sequential Statements
The compilation of classical sequential statements follows a classical compilation scheme [1] using conditional gotos for if-then-else and do-while statements. We shall simply consider these statements as part of our “high-level assembler”. In the same way, local variables are referred through their names rather than FP[i] where FP is the Frame Pointer register and i would be the relative address of the concerned variable in the current frame. New stack frames are allocated when scopes are entered. In our implementation, a particular care has been taken to reallocate unused frames through a simple garbage collection mechanism.

Schizophrenia is a usual issue in the compilation of imperative synchronous programs [5]. It comes from the fact that, because of loops, several instances of a same variable can coexist simultaneously (in a same reaction step). The problem does not arise here thanks to the use of stack frames: each time a scope is re-entered a new data frame is allocated. The separation between control and data is at the heart of our modular compilation technique since it allows to simultaneously run several threads sharing the same code but working on distinct data.

5.2 Parallel Statements
In cooperative multithreading, each thread is responsible for relinquishing control. This is ensured by the assembly instruction “stop”. Starting a new thread is done with the instruction “start pc, fp” where pc is the address of the first instruction of the started thread and fp is its frame pointer. The translation of a parallel statements $P_1 \mid P_2$ is then the following:

\[
\text{sync} = 2 \\
\text{start FORK_lbl, FP} \\
P_1 \\
goto \text{SYNC_lbl}
\]

where $\text{FORK_lbl}$:

\[
P_2
\]

\[
\text{SYNC_lbl:} \\
\text{sync} = \text{sync}-1 \\
\text{if sync} > 0 \text{ then} \\
\text{stop} \\
\text{endif}
\]

5.3 Hierarchical Execution Scheme
The execution of a synchronous program is a succession of atomic reactions during which threads run in parallel until reaching a pause statement. In the beginning of each reaction step, the threads are resumed at the very locations where the program eventually paused at the end of the previous reaction. However, not all threads are resumed: because of abort and suspend statements, the resumption of threads should actually be performed hierarchically, according to the structure of the source program and the dynamically fulfilled conditions. In the following Esterel example:

\[
\text{abort} \\
\text{suspend} \ P \\
\text{when} C_2 \\
\text{when} C_1
\]

the compilation into assembly code should provide a solution for hierarchically testing $C_1$ then $C_2$ before executing $P$, whatever it comprises of (involving complex parallel compositions and/or module instantiations).

5.3.1 Guarded “pause”
Each abort and suspend statement is then responsible for starting the threads running inside its body. In other words, each pause statement is guarded by the closest abort or suspend parent statement or, at the top level, by the Runtime System. The assembly code for pause statements is then the following:

\[
\text{RPC} = \text{PAUSE_lbl} \\
\text{RFP} = \text{FP} \\
\text{FP} = \text{FP}[0] \times n \\
goto \text{parent_guard_lbl} \\
\text{PAUSE_lbl:}
\]

where $\text{FP}[0] \times n = (\text{FP}[0]) \ldots [0]$, $n$ times.

The context of the thread is saved in two data registers: the resume point is saved in RPC (Resume Program Counter) and the current frame pointer in RFP (Resume Frame Pointer). The frame pointer is then popped of as many levels (given in $n$) as to retrieve the frame pointer of the guarding abort/suspend statement. At the address $\text{parent_guard_lbl}$ is the code responsible for managing the paused thread. The values $\text{parent_guard_lbl}$ and $n$ are defined at compile time.

In the following, we shall use the simpler assembly instruction “pause res” where res is the address where the pause has to resume at the next reaction step. The “pause res” instruction is comparable to that of the $\text{yield()}$ instruction of languages implementing coroutines [14] (Java, Python, Lua, ...). Thus, the previous code becomes simply:

\[
\text{pause PAUSE_lbl} \\
\text{PAUSE_lbl:}
\]
5.3.2 Guard Statements

The task of abort and suspend statements comprises of guarding the execution of their bodies. Their compilation has to provide the assembly code for managing the nested pause statements through the callback mechanism described before and the data registers RPC and RFP. The compilation of abort statements produces the following code:

\[ P \]
\[ \text{goto ENDLbl} \]
\[ \text{GUARDLbl:} \]
\[ \text{if child == [] then} \]
\[ \quad \text{child = (RPC, RFP)::child} \]
\[ \quad \text{pause RESUMELbl} \]
\[ \text{else} \]
\[ \quad \text{child = (RPC, RFP)::child} \]
\[ \quad \text{stop} \]
\[ \text{endif} \]
\[ \text{RESUMELbl:} \]
\[ \text{if not(C) then} \]
\[ \quad \text{start_all(child)} \]
\[ \quad \text{child = []} \]
\[ \quad \text{stop} \]
\[ \text{endif} \]
\[ \text{ENDLbl:} \]

The assembler block starting at the addresses RESUMELbl is responsible for resuming the execution of the paused threads inside P, under the condition C. Each time a thread of P is paused, it is added to a list “child” (a local variable stored in the current frame). These threads shall be passed through the registers RPC and RFP and managed from the address GUARDLbl. This label has to be provided when compiling pause statements inside P, so that the control jumps to this very address each time a thread is paused. When the first thread is registered (child == []), the abort statement has also to register itself to the parent guard (pause RESUMELbl). The translation of suspend statements is hardly different so that when the condition C holds, the paused threads are not started. Instead, the list is kept and the suspend statement registers itself again to the parent guard:

\[ \text{RESUMELbl:} \]
\[ \text{if C then} \]
\[ \quad \text{pause RESUMELbl} \]
\[ \text{else} \]
\[ \quad \text{start_all(child)} \]
\[ \quad \text{child = []} \]
\[ \quad \text{stop} \]
\[ \text{endif} \]

In the case of the abort statement, P is simply not restarted and the control flows to the rest of the program.

The translation of weak abort statements is slightly more complex since the abortion of P has to take place one logical instant later than it would be in the case of an abort statement. Nevertheless, this translation is similar enough to that of the abort statement for doing without such extra details here.

5.4 Modules

The instantiation of synchronous modules can be compiled as an almost classical function call where caller context and parameters are passed through a stack frame. The only difference with respect to the classical function call of sequential programs is that the guard context has to be stacked (since the module could be reentered several times along the successive reactions). The compilation of a module instantiation:

\[ \text{mod_name}(p_1, \ldots, p_m) \]

is then the following:

\[ SP = \text{new_frame}(m + 4) \]
\[ SP[0] = \text{RETURNLbl} \]
\[ SP[1] = FP \]
\[ SP[2] = \text{parent_guardLbl} \]
\[ SP[3] = FP[0] \]
\[ SP[1 + 3] = p_1 \]
\[ \ldots \]
\[ SP[m + 3] = p_m \]
\[ FP = SP \]
\[ \text{goto mod_name} \]
\[ \text{RETURNLbl:} \]

As for pause statements, the value of parent_guardLbl and n are defined at compile time. As to cope with our calling conventions, the declaration of a module:

\[ \text{module mod_name}(S_1, \ldots, S_n) \]
\[ P \]
\[ \text{end module} \]

generates the following code:

\[ \text{mod_name:} \]
\[ P \]
\[ \text{ret = FP[0]} \]
\[ FP = FP[1] \]
\[ \text{goto ret} \]
\[ \text{GUARDLbl:} \]
\[ \text{ret = FP[2]} \]
\[ FP = FP[3] \]
\[ \text{goto ret} \]

Where GUARDLbl is the parent guard label of the module body P. It is necessarily provided for the compilation of any pause, abort or suspend statement of P.

5.5 Dealing with Signals

Over the rigid framework provided by the structural translation of synchronous modules into cooperative sequential threads, we shall now provide a solution to deal with run-time causality induced by signals.

5.5.1 Implementation of Signals

At the end of a reaction step, the status of any signal has to be either present or absent. Inside a reaction, every signals but inputs remain undefined until reaching an emit statement or reaching a state of the program where all potential emissions are discarded.

We propose the following approach: before a signal is read or tested, we check its status. If it is undefined, then the execution of the thread is suspended. Each signal maintains a list of pending threads, so that they are immediately restarted as soon as a definitive value or status of the signal is determined (observer pattern). For this purpose, we use an assembly instruction “wait”, defined as a macro for the following code:

\[ \text{if S.status == UNDEFINED then} \]
\[ \quad S.pending = (\text{RESUMELbl, FP})::S.pending \]
\[ \quad \text{stop} \]
\[ \text{endif} \]
\[ \text{RESUMELbl:} \]

The emission or absence of a signal is realized through the assembly instruction “emit S, status, value”, defined as a macro for the following code:
$S$.status = $status$
if $status == $PRESENT then
    $S$.value = $value$
endif

start_all($S$.pending)

5.5.2 Immediate Reaction to Absence
Reacting to the absence of a signal depends on the global behavior of the program. Our approach involves the Run-time System at the top level which maintains a list “pending” of undefined signals. At some point some signals have to be declared absent so as to achieve the current reaction step. However, not all the pending signals can be declared as absent. Let us consider the following example:

```c++
if present($S1$) then emit $S3$ else emit $S2$
else
    if present($S2$) then emit $S3$
else
    if present($S3$) then emit 0
```

where $S1$, $S2$ and $S3$ are local signals. The execution of this program leads to a point where the threads are blocked on these signals. At this very point, only $S1$ is safe to be set absent since $S2$ and $S3$ still have potential emitters.

5.5.3 Potential Emissions
Our solution relies on a local knowledge of the potentially emitted signals at each point of the program [22]. We use reference counters on signal to indicate which signal can be safely set absent. We introduce the instructions “can $S$” to mark a signal as possibly emitted in the current thread ($S$.refcount = $S$.refcount+1). We introduce the instruction “cannot $S$” to mark that the control just reached a point where a signal cannot be emitted any more in the current instant and by the current thread ($S$.refcount = $S$.refcount-1). This strategy complies with the definition of our Potential Function given in section 2.

5.5.4 Reference Counter Policy
At compile time, we locally compute the set of potentially emitted signals for each statement of the program. We then generate the can statements at each resumption point of the program (i.e for each pause statement), so that a thread immediately declares its potentially emitted signals as soon as it is started or resumed. Finally, we generate the cannot statements at each point of the target code where the set of potential emissions decreases. For example, the insertion of can and cannot statements in the code of the figure 1.a will produce the code of the figure 1.b.

```c++
pause
    pause;
    can A; can B; can C
emit A
    emit A
cannot A
if present($S$) then
    if present($S$) then
        cannot C
    emit B
    emit B
else
    emit C
    emit C
cannot C
pause
```

(a) (b)

Figure 1: Insertion of can and cannot statements.

A signal can thus be safely declared absent when its reference counter is equal to zero and when the list of active threads is empty (which means that any active threads had the opportunity to increase the reference counter of the concerned signals before suspending their execution). The global information about the potentially emitted signals is then obtained by the execution of all the active threads providing local partial information. This strategy for the generation of can and cannot instructions is not unique and leaves room for optimizations.

In the context of a modular execution, the correct compilation of programs requires some minimal knowledge about the modules. So as to identify the potentially emitted signals, we then need to know:

1. which parameters can be emitted at the first reaction of the module and
2. if the module can be instantaneous.

This information can be carried by a type system and stored in a header file. When a module is about to return, it also requires the list of signals that are potentially emitted by the caller after the termination of the callee. This information is actually passed on stack frames.

5.6 Higher-Order Modules
The main originality of this approach with respect to the related others lies in that the scheduling of the program blocks is not settled statically at compile time. Furthermore, the possibility of compiling synchronous modules as “black boxes” offers us the key for seamlessly introducing higher-order modules in imperative synchronous languages. For our compiler indeed, a module is a simple pointer Modules can thus be transmitted through signals and be instantiated from a signal value easily.

6. EXPERIMENTAL RESULTS
We developed a lightweight compiler based on these works called fnec that generates C-code. We use a standard extension to the C language known as “computed goto”, where program labels can be handled as any (void*) pointers. We tested our compiler on many little examples and on the standard wristwatch example (see table 2). The generated code is quite big and slow, which is not a surprise considering that no effort was made for performance. The C code generation however is itself very fast.

These experimental results are more illustrative than relevant since our unique motivation was the introduction of higher-order modules in Esterel. In future works, efficient techniques [21, 15] shall be used to improve the execution speed of our modular code.

7. CONCLUSION
We presented an extension of the Esterel language for the introduction of higher-order modules. The benefit of it lies in its increased flexibility and its ability to better design synchronous systems. Constructiveness issues are managed by the mean of a type system indicating the potentially emitted signals of the modules. This extension has been implemented as a lightweight compiler generating sequential software (C) code.

As future works, we plan to use static analysis techniques to infer as much type information as possible. The idea would be to exploit a graph describing possible instantaneous dependencies between signals. Then, depending on the program locations where higher-order modules are instantiated, infer as much as possible which parameters are safe to be emitted in the first reaction and if the module can safely terminate instantaneously. We shall also investigate code migration as the next-step extension of Esterel. The principle would be to start a module somewhere in the program, then suspend its execution and resume it at another
location in the code. Actually, such an extension would be quite easy to implement in our higher-order-tolerant compiler. The challenge lies mainly in finding the best (simplest) way to express migration in Esterel without conflicting with the existing constructs of the language. On a technical level, Bruno, T. Cordyly, M. Bouzouit, and J.-B. Stefani. An open component model and its implementation. Science of Computer Programming, 21(2):87–152, 1992.


