A Proposal for Coroutines in E-LOTOS*

Hubert Garavel, Mihaela Sighireanu

July 1997

Abstract

This paper proposes a coroutine operator for E-LOTOS. The syntax, the static semantics, and the dynamic semantics of the operator are provided. We show by several examples that the coroutine operator is adequate to express suspension and resuming. Moreover, it is more general and simpler than the suspend-resume operator.

1 Introduction

The importance of coroutines as a programming paradigm is recognized. The GR3 [GS96] input document of the Grenoble meeting provides an overview of the coroutine mechanism in programming languages and a rationale for introducing a coroutine operator in E-LOTOS. Also, it proposes a concrete syntax, a static semantics, and a dynamic semantics for a such operator.

Although the utility of the coroutine operator has been recognized at the Grenoble meeting, some remarks have been raised about its proposed syntax and semantics:

- The syntax proposed in GR3 for the coroutine operator is:

  exec $[n_0$ **on** $]$ [**urgent**] $G_1 : T_1, ..., [$**urgent**$]$ $G_m : T_m$ **in**
  $G_1^1, ..., G_1^{m_1}$ $\rightarrow$ B_1
  ...
  $G_n^1, ..., G_n^{m_n}$ $\rightarrow$ B_n
  endexec

  This syntax is too complex, because it did: (1) the declaration of the gates used for control passing between coroutines, $G_1, ..., G_m$; (2) the type (urgent or not) of each gate; (3) the list of gates used to suspend or resume each coroutine.

- The static constraints are too complex. For example, it was required that each gate $G \in \{ G_1, ..., G_m \}$ appears at least in two gate lists $\{ G_i^1, ..., G_i^{m_i} \}$.

- Due to the complexity of the syntax, the dynamic semantics is also complex: it provides the urgency only on gates declared urgent and the simultaneous aging of all coroutines. Moreover, the treatment of urgency is not compatible with the E-LOTOS principles: although the operator

*This work has been supported in part by the European Commission, under project ISC-CAN-65 "EUCALYPTUS-2: An European/Canadian Lotos Protocol Tool Set".

declares gates (as a hide operator), these gates are visible outside its scope and may be urgent or not.

All these lacks are overtaken by the operator we propose in this paper. Moreover, the actual proposal is a real concurrent for the suspend/resume operator actually included in the Committee Draft document.

2 Requirements for the coroutine operator

We list below the main features that must be considered for a coroutine operator:

- **Determinism**: The coroutines were introduced in sequential languages to bring a "flavour of concurrency" by allowing computations to be interleaved (i.e., executed in pseudo-parallelism), according to a demand-driven scheduling strategy.

  Compared to the concurrency offered by a language like LOTOS, coroutines may be useful to specify a limited (i.e., deterministic) form of parallel composition. There are examples [GH93, GH94] where coroutines should be preferred to asynchronous parallel composition. This is typically the case for embedded systems, the code of which has to be deterministic in order to match certification requirements from the certification authorities.

- **Suspend/Resume capabilities**: The coroutine operator must provide at least the functionalities offered by the suspend/resume operator.

- **Simplicity**: The syntax and the semantics of the operator must be simple. For this reason, we consider that the operator must be concerned only with the control passing and not with hiding and urgency of the actions. These last features are appropriately treated by the hide operator.

3 A new coroutine operator

3.1 Formal definition

Syntax

$$\textbf{exec}\ [n_0]\ \ \textbf{in}\ ([G] \rightarrow B)^+\ \textbf{endexec}$$

The default initial coroutine number is 0. Some vocabulary: $\vec{G}$ are called special gates, $\vec{B}$ are called coroutine behaviours, and B_{n_0} is called the running (or active) coroutine.

Comment: *Remark here that $G \neq$ **exit**; if "**exit**" is a special gate, the problem of non-urgency of δ appears and should be treated as in synchronization on termination in the parallel composition.*

Static Semantics

$$\frac{\begin{array}{c}\mathcal{C} \vdash G_0 \Rightarrow \textbf{gate}\ T_0 \quad \cdots \quad \mathcal{C} \vdash G_n \Rightarrow \textbf{gate}\ T_n \\ \mathcal{C} \vdash B_0 \Rightarrow \textbf{exit}(RT) \quad \cdots \quad \mathcal{C} \vdash B_n \Rightarrow \textbf{exit}(RT) \\ \forall i, j \in 0..n.\ \ (i \neq j) \Longrightarrow (G_i \neq G_j)\end{array}}{\mathcal{C} \vdash (\textbf{exec}\ n_0\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \Rightarrow \textbf{exit}(RT)}\,[0 \leq n_0 \leq n]$$

A similar rule must be introduced to compute the guardedness of the "**exec**" behaviour.

Comment: *The premise* $\forall i, j \in 0..n.\ \ (i \neq j) \Longrightarrow (G_i \neq G_j)$ *is added in order to avoid non-determinism in control passing between coroutines. An alternative solution which allows this kind of non-determinism may provide a determinism behaviour by choosing different names for the gates* $G_0, ..., G_n$.

Untimed semantics

$$\frac{\mathcal{E} \vdash B_i \stackrel{\mu(RN)}{\longrightarrow} B_i' \qquad \forall k \in 0..n.\ \ (\mu \neq G_k) \wedge ((k \neq i) \Longrightarrow (B_k \equiv B_k'))}{\mathcal{E} \vdash (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \stackrel{\mu(RN)}{\longrightarrow} (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0'\ \ldots\ [G_n] \rightarrow B_n')}$$

$$\frac{\mathcal{E} \vdash B_i \stackrel{G_j(RN)}{\longrightarrow} B_i' \qquad \mathcal{E} \vdash (\textbf{exec}\ j\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \stackrel{G(RN)}{\longrightarrow} B'}{\mathcal{E} \vdash (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \stackrel{G(RN)}{\longrightarrow} B'}$$

$$\frac{\mathcal{E} \vdash B_j \stackrel{G_j(RN)}{\longrightarrow} B_j' \qquad \mathcal{E} \vdash B_i \stackrel{G_j(RN)}{\longrightarrow} B_i' \qquad \forall k \in 0..n.\ \ (k \neq i) \wedge (k \neq j) \Longrightarrow (B_k \equiv B_k')}{\mathcal{E} \vdash (\textbf{exec}\ j\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \stackrel{G(RN)}{\longrightarrow} (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0'\ \ldots\ [G_n] \rightarrow B_n')}$$

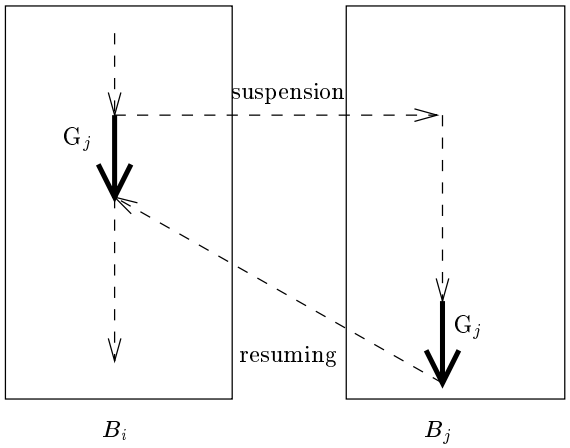
Timed semantics

$$\frac{\mathcal{E} \vdash B_i \stackrel{\epsilon(d)}{\longrightarrow} B_i' \qquad \forall k \in 0..n.\ \ (k \neq i) \Longrightarrow (B_k \equiv B_k')}{\mathcal{E} \vdash (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0\ \ldots\ [G_n] \rightarrow B_n) \stackrel{\epsilon(d)}{\longrightarrow} (\textbf{exec}\ i\ \textbf{in}\ [G_0] \rightarrow B_0'\ \ldots\ [G_n] \rightarrow B_n')}$$

3.2 Remarks on semantics

- When the running coroutine exits, the "**exec**" behaviour exits.
- Since we cannot know who will exit, the static semantics rule is similar to the choice operator rule.
- The first untimed rule defines the *normal execution of the active coroutine*. It states that the active coroutine B_i can execute freely any transition labelled with a gate (or a signal) μ not belonging to the set of special gates $G_0, ..., G_n$. After the transition, B_i remains active and the other coroutines do not evolve.

  The second untimed rule defines *control passing without synchronization*. It applies when B_i is ready to execute a transition $G_j(RN)$. In such case, B_i is suspended before executing $G_j(RN)$, and the control is passed to B_j.



The third untimed rule defines *control passing with synchronization*. It applies when both B_i and B_j are ready to execute a transition $G_j(RN)$. In such case, B_i and B_j execute G_j simultaneously and the control is passed from B_j to B_i (this models coroutine resumption as a particular case).

Comment: *Several behaviours B_i may be able to synchronize on G_j. In this case, the non-determinism appears.*

- According to the idea that the coroutines have to share the "processor" time, the aging is done only for the active coroutine. In this case, the time captured by each coroutine is the running time. The waiting time may be captured by synchronization with a global behaviour, as we will show into the section 4.

4 Two examples

4.1 Scheduling

We show here how our coroutine operator may be used to describe scheduling processes. The example chosen below was proposed in the LG9 [HF95] to illustrate the need for a suspend/resume operator. It describes three independent tasks sharing the same processor till 5 time units each time. The scheduling is done randomly and the scheduler centralizes the running time for each task.

The main behaviour is the following:

hide Sch, Tk_1, Tk_2, Tk_3 : **none in**
  exec 0 **in**
    $[Sch] \rightarrow Scheduler[Tk_1, Tk_2, Tk_3]\ (d_1 \Rightarrow 5, d_2 \Rightarrow 10, d_3 \Rightarrow 15)$
    $[Tk_1] \rightarrow (Task[start, end]\ (id \Rightarrow 1, d \Rightarrow 5) \,|||\, CP[Tk_1])$
    $[Tk_2] \rightarrow (Task[start, end]\ (id \Rightarrow 2, d \Rightarrow 10) \,|||\, CP[Tk_2])$
    $[Tk_3] \rightarrow (Task[start, end]\ (id \Rightarrow 3, d \Rightarrow 15) \,|||\, CP[Tk_3])$
  endexec
endhide

Notice that the scheduler is distributed into two places: the *Scheduler* process which controls the random allocation of the processor, and the *CP*s processes which measure the quota for each task.

The processes *Scheduler*, *Task*, and *CP* are defined as follows:

```
process Scheduler[Tk1, Tk2, Tk3: none] (d1, d2, d3: time) : noexit is
  if d1 > 0 then Tk1; Scheduler[...] (d1 − 5, d2, d3) else stop endif
  []
  if d2 > 0 then Tk2; Scheduler[...] (d1, d2 − 5, d3) else stop endif
  []
  if d2 > 0 then Tk3; Scheduler[...] (d1, d2 − 5, d3) else stop endif
endproc

process Task[start, end: nat] (id: nat, d: time) : noexit is
  start!id; wait(d); end!id; stop
endproc

process CP[Tk: none] : noexit is
  loop forever wait(5); Tk endloop
endproc
```

All the examples of scheduling processes given in LG9 [HF95] may be treated with the coroutine operator.

4.2 Multilevel Interruption System

This example was also presented in LG9 [HF95]. It describes a processor running infinitely and having accepting interruption from the environment by meaning of the *Int* gate. When an interruption is raised, the processor will run the interruption task till either a higher level interruption is raised or the task finishes. In the last case, the control is given back to the last interrupted task or to the processor.

The main behaviour is the following:

```
hide IT1, IT2, IT3: none in
  exec 0 in
    [Pr] → Processor[Int, IT1, IT2, IT3]
    [IT1] → ITask1[Int, IT1, IT2, IT3]
    [IT2] → ITask2[Int, IT1, IT2, IT3]
    [IT3] → ITask3[Int, IT1, IT2, IT3]
  endexec
endhide
```

where the processes *Processor* and *ITask* are defined as follows:

```
process Processor[Int:nat, IT1, IT2, IT3:none]:noexit is
  loop forever (* the normal execution of the processor *) endloop
  |||
  loop forever var it:nat in
    Int?it;
    if it = 1 then IT1
    elsif it = 2 then IT2
    elsif it = 3 then IT3
    endif
  endloop
endproc

process ITask1[Int:nat, IT1, IT2, IT3:none] :noexit is
  loop forever
  hide end:none in
    (* the normal execution of the IT1 *); end; IT1
    |[end]|
    loop X var it:nat in
      Int?it[it > 1]; if it = 2 then IT2 elsif it = 3 then IT3 endif
      []
      end; break X
    endloop
  endhide
  endloop
endproc

(* similar definitions for ITask2 and ITask3 processes *)
```

5 Comparison with the suspend/resume operator

Operator arity Our operator is as general as possible, its arity being n (as the "**trap**" and "**par**" operators). The suspend/resume operator is only a binary operator; the parallel composition operator is needed to express a set of n tasks sharing the same processor.

Aging In our operator, only the active coroutine ages, all other suspended coroutines do not.

For the suspend/resume operator, the semantics is more complex. Consider the behaviour B_1 [X>B_2; when B_1 is running, the aging of B_1 demands the aging of B_2; when B_1 is suspended and B_2 is running, B_1 does not age. This semantics is adopted only for compatibility with the disable operator. *Since the suspend/resume operator does not substitute the disable operator, we consider that it is not reasonable to have a such odd semantics!*

Value passing The value passing between the different coroutines is mandatory as long as coroutines are supposed to work together to produce information.

This problem is not solved in the suspend-resume operator of the Committee Draft document.

Our "**exec**" operator follows the message passing approach to solve this problem: the input/output of values are modeled with the rendez-vous on the special gates. So our proposal unifies, in some sense, the implicit (value passing) and explicit control passing.

Interruption There are two forms of interruption: (1) interruption due to an external event, and (2) auto-interruption of a process. The suspend/resume operator may model *only* the first form of interruption.

Our "**exec**" operator may model both forms: (1) the running process may be interrupted by an external process by synchronization to the interruption event and after that self-suspension; (2) the auto-interruption corresponds to the suspension at waiting synchronization at a special gate.

Suspend/resume actions In our proposal, the suspend and the resume actions are the same action (the special gate of the called coroutine, e.g. G_j). Moreover, this action is visible.

The suspend/resume operator considers only the resume action (the X signal), and this action cannot be observed. The suspending is implicit. In this case, the synchronization on resuming or suspending cannot be done.

6 Conclusion

We proposed the "**exec**" operator to provides a coroutine (and suspend/resume) mechanism in E-LOTOS. The operator has the following features:

- it is a n-ary operator with a simple syntax;
- its semantics is simple (2 rules for the static semantics and 5 rules for the dynamic semantics);
- it does not need any auxiliary operator for its definition (as suspend/resume operator);
- it allows value passing by the classical mechanism of rendez-vous;
- the suspend/resume action is visible to the external environment;
- it can models external and self interruption mechanisms.

These features are illustrated by two examples: scheduling and task switching.

It results that the "**exec**" operator is simpler than the suspend/resume operator and can be used to describe complex systems.

References

[GH93] Hubert Garavel and René-Pierre Hautbois. An Experiment with the Formal Description in LOTOS of the Airbus A340 Flight Warning Computer. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *First AMAST International Workshop on Real-Time Systems (Iowa City, Iowa, USA)*, November 1993.

[GH94] Hubert Garavel and René-Pierre Hautbois. *Experimenting LOTOS in Aerospace Industry.* In Teodor Rus and Charles Rattray, editors, *Theories and Experiences for Real-Time System Development*, volume 2 of *Amast Series in Computing*, chapter 11. World Scientific, 1994.

[GS96] Hubert Garavel and Mihaela Sighireanu. A Proposal for Coroutines an Suspend/Resume in E-LOTOS. Input document [GR3] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Grenoble, France, December, 12–17, 1996, December 1996.

[HF95] C. Hernalsteen and A. Février. A suspend/resume operator for ET-LOTOS. Input document [LG9] to the ISO/IEC JTC1/SC21/WG7 Meeting on Enhancements to LOTOS (1.21.20.2.3), Liège, Belgium, December, 19–21, 1995, December 1995.