A Proposal for Coroutines and Suspend/Resume in E-LOTOS*

Hubert Garavel, Mihaela Sighireanu

December 1996

Abstract

The importance of coroutines as a programming paradigm is recognized. This paper proposes to extend LOTOS with a coroutine operator, for which syntax, static semantics, and untimed semantics are provided. We show that this coroutine mechanism generalizes several other operatorrs, which exist in LOTOS or which have been proposed for E-LOTOS, including trap, suspend-resume, and hiding.

Keywords

Coroutine, ET-LOTOS, E-LOTOS, Formal Description Techniques, LOTOS, Process Algebra, Protocols.

1 Comments about the existing core behaviour language

At present, the definition of E-LOTOS, as provided by the Kansas City output document is far advanced. However, some issues remain unsolved, especially in the behaviour part:

- The definition of the core behaviour language, as produced after the Kansas City meeting, has introduced a new class of objects, *exceptions*, which can be felt as "similar to gates, but different".

  Following the famous Occam razor principle, we must question ourselves about the need for having two concepts: gates and exceptions.

  If we only consider an "untimed semantics" point of view, it is clear that gates and exceptions are much the same concept, and that we could replace exceptions by gates everywhere without any problem. This approach advocated in [GS96] brings simplicity, expressiveness and good algebraic properties.

  If we consider a "timed semantics" point of view, the major difference between gates and exceptions is urgency. According to ET-LOTOS principles, urgency is only attached to hidden gates; visible gates are not urgent. This is not the case with exceptions, which are always urgent.

  We notice that this definition of exceptions as urgent (although motivated by semantic "tricks") violates the ET-LOTOS principle stating that one cannot enforce urgency on visible events: from the outside of a system, we can observe visible, urgent exceptions, which is is not allowed for gates.

  If accepted, this design choice will have the following consequence. Practically, when designing a system in E-LOTOS, engineers will associate gates to non-urgent external events and exceptions

*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".

to urgent external events. However, as gates and exceptiosn are not symmetric (for instance, exceptions cannot be synchronized), they will face limitations (e.g., the impossibility of using constraint-oriented style for urgent signals).

This semantics also has unpleasant effects, especially in the definition of the synchronized termination (synchronization of two "**exit**s" by parallel composition).

- We are also worried by the fact that exceptions are systematically urgent, although this is not always necessary. For instance, if a process is programmed with sequential composition, it will use an urgent action "δ" that can be in conflict with a truly urgent external event.

  It is to be feared that a plain use of exceptions in real-time programs may create, as an artefact, undesirable urgency constraints, therefore leading to unsuitable non-determinism or time deadlocks.

  One can worry about the complexity of the resulting language, since urgency occurs at different places: for hidden gates, for exceptions, in the "**hide**" operator. Having urgency disseminated at many places and in different contexts would not make of E-LOTOS and elegant and easy-to-learn language, nor it would simplify the design and analysis of real-time programs with "true" urgency constraints.

- The definition of the core behaviour language does not include a suspend-resume operator. As it is now, E-LOTOS cannot be used for describing the behaviour of real-time systems with task switching and coroutines. The need for such features has been pointed out in many places, e.g. [GH93, GH94] in the case of avionics embedded software.

- Finally, the suspend-resume operator proposed in [Her96] seems incompatible with the "**trap**" operator of E-LOTOS. It would be better if the "**trap**" operator could be seen as a particular case of a suspend that does not resume.

2 Motivations and rationale

In order to solve the aforementioned problems, we suggest a refined design of the behaviour language, based upon the following principles:

1. The practical need for *suspend-resume* operator, and more generally for task switching and coroutines, should be addressed.

2. *Urgency should be introduced in a single place.* There should exist a unique operator to introduce urgency, all possible uses of urgency being obtained as shorthands from this operator.

3. Currently, the reason for distinguishing gates and exceptions is urgency. In the proposed new framework, this justification should no longer hold. *Gates and exceptions should be unified*, thus leading to a great reduction in syntactic and semantic complexity.

4. *Urgency should be made optional.* The E-LOTOS specifier should have the freedom to specify when urgency is needed and when it is not (in opposite to the current definitions where urgency is always a default, for hidden gates, for exceptions, etc.) This will allow a more accurate description of timed behaviours, avoiding over-specification issues in real-time constraints.

5. *Trap and suspend-resume should be integrated.* The '**trap**" operator should be equivalent to a suspend operator without resumption. It would be even better the both the trap and suspend-resume could be obtained as special cases of a more general mechanism.

3 A quick overview of coroutines in programming languages

Coroutines allow to model a set of cooperating processes that execute in mutual exclusion, only one of these processes being active at a time. We see coroutines as an intermediate step between sequential and parallel programming:

- Introduced in sequential languages, coroutines bring a "flavour of concurrency" that allows computations to be interleaved (i.e., executed in pseudo-parallelism), according to a demand-driven scheduling strategy.

  Coroutines were originally introduced by Conway [Con63] as a natural paradigm in compiler development. Several programming languages have incorporated the concept of a coroutine in their definition, such as SIMULA and MODULA-2.

  Another approach was adopted for classical programming languages as PASCAL and FORTRAN for which coroutine extensions have been proposed as a mean to add missing concurrent capabilities.

- Compared to the concurreny offered by a language like LOTOS, coroutines may be useful to specify a limited form (e.g., deterministic) form of parallel composition.

  The importance of coroutines (combined with concurrency) for parallel languages has been pointed out by Kahn and MacQueen [KM77]. They proposed a coroutine mechanism to specify networks of parallel processes interconnected by channels, as a part of theoretical work on "lazy evaluators" and "streams" for functional languages.

  But, not many parallel languages include coroutines, mostly due to the fact that parallel composition operator already provide a non-deterministic solution to the need for coroutines.

  However, there are examples 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. For these applications, the parallel composition of LOTOS cannot be used, as it relies non-determinism. Therefore, it would be useful to have a more limited form of cooperating processes that does not introduce non-determinism. This is typically the reason for the success of the synchronous languages (LUSTRE, ESTEREL, and SIGNAL) in the area of critical systems.

  Moreover, the need for coroutines in real-time systems appears clearly from the description in LOTOS of the Airbus A340 Flight Warning Computer [GH93, GH94], where LOTOS is unable to describe the context switching between different tasks.

A comparative study of coroutines is done in [PS80]. Although, the details may vary from one computer language to another, the various coroutine mechanisms present a number of similarities:

- A set of coroutines is a set of tasks that execute one at a time (presumably on a unique processor) and cooperate to perform a computation. As stated in [PS80], *a coroutine is a generalized procedure, in that execution of a coroutine can be temporarily suspended and subsequently resumed at the place it was last active.*

- Depending on the coroutine mechanism under study, there can exist an additional task (named *controller*) that enforces a given discipline (*scheduling*) in the execution of these tasks. If this controller exists, the coroutine scheme is said to be *semi-symmetrical*; otherwise it is said to be *symmetrical.*

- The *creation* of the coroutines must be done before they are referenced and executed. The execution may start either automatically upon creation (as in SIMULA), or by an explicit command

(as in SL5). [PS80] state that the creation and immediate activation is the best solution for implementation purpose.

- The *suspension* of the executing coroutine can be either *implicit* (when the coroutine executes an input/output request, it is suspended and the control is passed to the other coroutine concerned by the input-output operation), or *explicit* (when the control is passed to another coroutine or to the controller, if any).
- the *termination* of the coroutines raises the question of the place where the control returns. Some programming languages prohibit termination (as in SL5), but in general the control is passed to the behaviour that created the set of coroutines.

4 Definition of the coroutine operator

Following the ideas behind the "**trap**" operator, we propose to introduce in E-LOTOS a coroutine mechanism that can be seen as a generalization of the exception mechanism currently proposed for E-LOTOS.

As for the "**trap**" operator proposed in [GS96], which only needs a single new behaviour operator for introducing exceptions in LOTOS, our proposal only needs one additional beheaviour operator (named "**exec**") to introduce coroutines in E-LOTOS. No extra feature (especially, no new class of identifiers) is required.

In the design of a coroutine mechanism for E-LOTOS, the main problem is not to model suspension and resumption of coroutines. The control passing can be easily obtained by gates (in the same way as for the "**trap**" operator or the suspend-resume operator of [Her96]).

The difficult problem is rather value passing between the different coroutine, which is mandatory as coroutines are supposed to work together to produce information. For instance, this problem is not solved in the suspend-resume operator proposed in [Her96], which does not allow values to be passed from from the suspended/resumpted behaviours. On the other hand, an appropriate coroutine mechanism should be able to emulate the existing "**trap**" operator that allows the interrupted behaviour to pass values to the exception handler using gate parameters.

There are two possible approaches:

Shared variables: as coroutines execute one at a time (in mutual exclusion), they could very well communicate using global variables, shared by all the coroutines. A sufficient condition for ensuring that these variables are always set before used would be to require that they are syntactically initialized.

Although the shared variable paradigm is presumably sound, we do not consider it for E-LOTOS, because it would be too far from the remainder of the language, which is based upon communication by message passing (rendez-vous).

Moreover, the modeling of the existing "**trap**" operator using coroutines with shared variables would be tricky as the interrupted behaviour passes information to the exception handler using gate parameters, which should be somehow translated into global variable assignments.

Message passing: we will follow this approach which is compatible with the rendez-vous paradigm of LOTOS. Since input/outputs are modelled with rendez-vous in LOTOS, our proposal unifies, in some sense, the implicit and explicit control passing.

4.1 Notations

The following notations hold for the remainder of the paper.

$G, G_1, G_2, ...$ denote observable *gates*; we note "δ" the special gate generated by the "**exit**" operator of LOTOS.

$\Gamma, \Gamma_1, \Gamma_2, ...$ denote *channels* (i.e., gate types).

$B, B_1, B_2, ...$ denote *behaviour expressions.*

$S, S_1, S_2, ...$ denote *sorts*, i.e., data domains (also called *types* in this paper).

$V, V_1, V_2, ...$ denote *variables.*

$\vec{V}, \vec{V}_1, \vec{V}_2, ...$ denote *variable declarations*, i.e., (possibly empty) lists of the form "$(V_1 : S_1, ..., V_n : S_n)$", where each variable V_i is declared to have the sort S_i.

$E, E_1, E_2, ...$ denote *value expressions*, i.e., algebraic terms that may contain variables.

$e, e_1, e_2, ...$ denote *ground terms* of the initial algebra, i.e., canonical representatives of the quotient algebra. Ground terms are a subset of value expressions: they do not contain variables and play the role of "constant" value expressions.

$\vec{e}, \vec{e}_1, \vec{e}_2, ...$ denotes *value lists*, i.e., (possibly empty) lists of the form "$(e_1, ..., e_n)$".

"$[\vec{e}/\vec{V}]\ B$" denotes the behaviour expression B in which all variables of $\vec{V}$ are replaced with the corresponding values of $\vec{e}$ ($\vec{V}$ and $\vec{e}$ should have identical number of elements and the types of their elements should be pairwise compatible).

T denotes the countable time domain which is the alphabet of time actions. $T_{0,\infty} = T - \{0, \infty\}$.

4.2 Concrete and abstract syntax of the coroutine operator

The concrete syntax of the proposed coroutine operator is[1]:

exec $[n_0$ **on** $]$ [**urgent**] $G_1 : \Gamma_1, ...,$ [**urgent**] $G_m : \Gamma_m$ **in**
$G_1^1, ..., G_1^{m_1}$ -> B_1
...
$G_n^1, ..., G_n^{m_n}$ -> B_n
endexec

with $m > 0$, $n > 1$, $m_1 > 0$, ..., $m_n > 0$ and $1 \leq n_0 \leq n$. The n_0 value is optional: if absent, one assumes that $n_0 = 1$. The "**urgent**" keywords are optional.

4.3 Static semantics of the coroutine operator

In the definition of the the coroutine operator:

- The clauses "[**urgent**] $G_i : \Gamma_i$" are *definition*-occurrences[2].. Each of them declares a gate G_i of type Γ_i. In the sequel we will note $u(G_i)$ a boolean that is equal to true iff the "**urgent**" is present before the declaration of G_i.

[1] The keyword "**exec**" might be replaced with another one, e.g., "**exec**", "**group**" etc.
[2] also called *binding occurrences* in [ISO88]

- The clauses "$G_i^1, ..., G_i^{m_i}$ ->" are *use*-occurrences. In each of them, the gates G_i^j should be pairwise distinct and should belong to the set $\{G_1, ..., G_m\}$ of gates declared at the beginning of the "**exec**" operator.
- In each behaviour expression B_i, the gates $\{G_i^1, ..., G_i^{m_i}\}$ are visible. The other gates belonging to $\{G_1, ..., G_m\} - \{G_i^1, ..., G_i^{m_i}\}$ are not visible in B_i. For each B_i we will note $\Theta(B_i)$ the set of gates $\{G_i^1, ..., G_i^{m_i}\}$ associated to B_i.
- One may require that each gate $G \in \{G_1, ..., G_m\}$ appears at least in two $\Theta(B_i)$ (or even exactly in two $\Theta(B_i)$).

4.4 Dynamic semantics of the coroutine operator

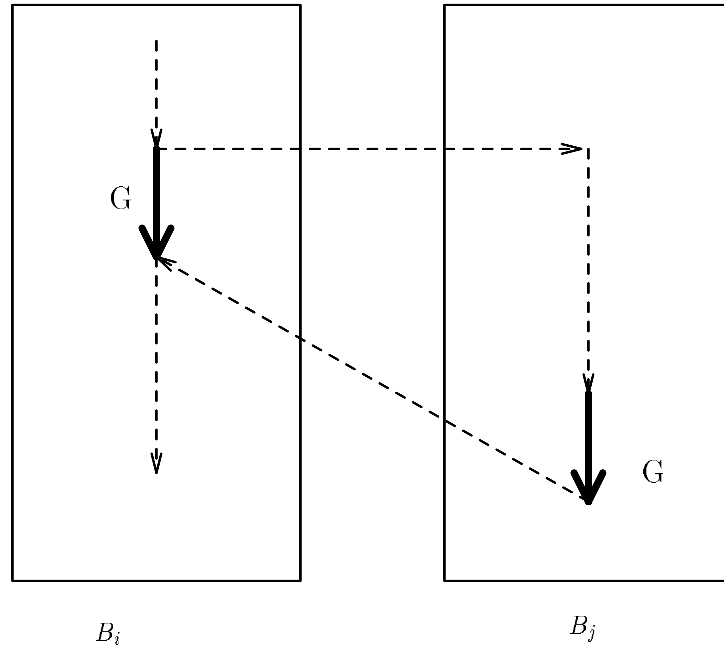
The intuitive semantics of this operator can be explained as follows:

- This operator creates and activates a set of n coroutines (represented by the behaviour expressions $B_1, ..., B_n$). The coroutine scheme is symmetric: there is no extra controller.
- Only one coroutine is active (i.e., has the *thread*) at a time. n_0 is the index of the first coroutine to be active. When the "**exec**" contruction is executed, the control is passed to B_{n_0}.
- Control passing between two coroutines B_i and B_j (assuming that B_i is active and B_j is suspended) means that B_j will become active and B_j suspended.
- Control passing between two coroutines can only be done using the "special" gates $\{G_1, ..., G_m\}$ that have been declared at the beginning of the "**exec**" operator.
- As for the generalized parallel composition operator of E-LOTOS, control passing only occurs between coroutines that share common gates. More precisely, two coroutines B_i and B_j may pass control on gate G iff $G \in \Theta(B_i) \cap \Theta(B_j)$.

  Note: this explains why we require that all m_i's are strictly greater than zero, i.e., that each $\Theta(B_i)$ is not empty. Unless the trivial case where $n = 1$ (i.e., there is only one B_i in the coroutine), this situation would not make sense. Let us consider a B_i such that $\Theta(B_i) = \emptyset$: either B_i is initially active ($i = n_0$), in which case it cannot pass the control to another coroutine or B_i is initially inactive ($i \neq n_0$), in which case it never receives the control.

  Note: this also explain why we may wish to require that each gate G occurs in exactly in two $\Theta(B_i)$. A gate G that occurs in no $\Theta(B_i)$ is useless and can be removed. A gate G that occurs in a single $\Theta(B_i)$ will create a deadlock in the coroutine. A gate G that occurs in a more than two $\Theta(B_i)$ may create non-determinism when determining which process is to be resumed.
- In LOTOS, parallel composition involves "rendez-vous": when two processes are synchronized on a gate G, each of them has to perform a transition labelled with G and both trasitions must be done simultaneously.

  The proposed coroutine mechanism follows similar principles and achieves a "uniprocessor" version of the LOTOS rendez-vous. If the active coroutine B_i wants to execute a transition labelled with a special gate G shared by B_i and another coroutine B_j, then B_i must pass the control to B_j, which will become active and will execute until it can also perform a transition labelled with G. The following diagram illustrates this suspend/resume mechanism for two coroutines:



- During control passing on a gate G, typed values can be exchanged on this gate using experiment offers in the same way as for ordinary LOTOS rendez-vous.
- As for the "**trap**" operator, control passing is atomic, in the sense that it creates no transition (even a "**i**" transition) observable from the outside.

More formally, the dynamic semantics of the coroutine operator can be defined as follows. For conciseness, we abbreviate the concrete syntax:

exec n_0 **on** [**urgent**] $G_1 : \Gamma_1, ..., $[**urgent**] $G_m : \Gamma_m$ **in**
$G_1^1, ..., G_1^{m_1}$ -> B_1
...
$G_n^1, ..., G_n^{m_n}$ -> B_n
endexec

as follows — in order to get rid from gate lists, which remain constant during the execution and which will be considered implicitly through the predicates $u(G_i)$ and $\Theta(B_i)$:

exec n_0 **in** $B_1, ..., B_n$

The value of n_0 is worth being kept in the concise notation, because this value denotes the index of the active coroutine, which will vary during the execution. In the dynamic semantics rule, we will note B_i the active coroutine (where i is not necessarily equal to n_0).

4.4.1 Untimed semantics

The first 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 G not belonging to the set of special gates of $\Theta(B_i)$. After the transition, B_i remains active and the other coroutine do not evolve.

$$\frac{G \notin \Theta(B_i) \ \wedge \ B_i \xrightarrow{G\ \vec{e}} B_i'}{\textbf{exec } i \textbf{ in } ..., B_i, ... \xrightarrow{G\ \vec{e}} \textbf{exec } i \textbf{ in } ..., B_i', ...}$$

Notice that, due to the static semantic constraints: $G \notin \Theta(B_i) \Longleftrightarrow G \notin \{G_1, ..., G_m\}$; therefore if G is not a special gate for B_i, it is neither a special gate for another coroutine B_j.

The two remaining rules concern the case of two coroutines B_i (the active coroutine) and B_j "connected" with the same special gate G. They both deal with the case where B_i wants to execute a transition labelled $G\ \vec{e}$, G being a special gate and $\vec{e}$ a list of values. Two cases are to be distinguished, depending whether B_j is ready or not to execute a similar transition $G\ \vec{e}$. In both cases, the control is passed from B_i to B_j (provided that B_j is able to progress).

The second rule defines *control passing with synchronization*. It applies when both B_i and B_j are ready to execute a transition $G\ \vec{e}$. In such case, B_i and B_j execute G simultaneously and the control is passed from B_i to B_j (this models coroutine resumption as a particular case). The fact that $\vec{e}$ is the same for B_i and B_j implicitly provides for value communication, because of pattern-matching of experiment offers takes place exactly in the same way as for LOTOS parallel composition. Notice that, because the control passing has to be atomic, one requires that B_j, if active, can execute another transition L after $G\ \vec{e}$. From the "outside" of the coroutine, as a result of the whole evolution, only the transition L will be observed.

$$\frac{i \neq j \ \wedge \ G \in \Theta(B_i) \cap \Theta(B_j) \ \wedge \ B_i \xrightarrow{G\ \vec{e}} B_i' \ \wedge \ B_j \xrightarrow{G\ \vec{e}} B_j' \ \wedge \ \textbf{exec } j \textbf{ in } ..., B_i', B_j', ... \xrightarrow{L} B}{\textbf{exec } i \textbf{ in } ..., B_i, B_j, ... \xrightarrow{L} B}$$

The third rule defines *control passing without synchronization*. It applies when B_i is ready to execute a transition $G\ \vec{e}$, where G is the special gate but when B_j, if active, can do another action L. In such case, B_i is suspended before executing $G\ \vec{e}$, and the control is passed to B_j. The suspension is atomic, in the sense that only the transition L will be observed from the outside.

$$\frac{i \neq j \ \wedge \ G \in \Theta(B_i) \cap \Theta(B_j) \ \wedge \ B_i \xrightarrow{G\ \vec{e}} B_i' \ \wedge \ \textbf{exec } j \textbf{ in } ..., B_i, B_j, ... \xrightarrow{L} B}{\textbf{exec } i \textbf{ in } ..., B_i, B_j, ... \xrightarrow{L} B}$$

By induction on the three rules, it is clear that the gate of the label L used in the second and third rules is never equal to a special gate of the coroutine. This justifies our assertion that control passing is atomic: no special gate can be observed "outside of" the coroutine.

4.4.2 Timed semantics

$$\frac{(\forall j \neq i \ B_j \xrightarrow{d} B_j') \ \ \wedge \ \ B_i \xrightarrow{d} B_i' \textbf{ refusing } \Omega}{\textbf{exec } i \textbf{ in } ...B_k... \xrightarrow{d} \textbf{exec } i \textbf{ in } ...B_k'...}$$

$$\frac{\begin{array}{c} j \neq i \ \ \wedge \ \ G \in \Theta(B_i) \cap \Theta(B_j) \ \ \wedge \ \ B_i \xrightarrow{G\ \vec{e}} B_i' \textbf{ refusing } \Omega - \{G\} \ \wedge \\ B_j \xrightarrow{G\ \vec{e}} B_j' \ \ \wedge \ \ \textbf{exec } j \textbf{ in } ...B_i', B_j'... \xrightarrow{d} B \end{array}}{\textbf{exec } i \textbf{ in } ...B_i, B_j... \xrightarrow{d} B}$$

$$\frac{j \neq i \quad \wedge \quad G \in \Theta(B_i) \cap \Theta(B_j) \quad \wedge \quad B_i \xrightarrow{G\ \vec{e}} B_i' \ \textbf{refusing}\ \Omega - \{G\} \ \wedge \quad \textbf{exec}\ j\ \textbf{in}\ ...B_i, B_j ... \xrightarrow{d} B}{\textbf{exec}\ i\ \textbf{in}\ ...B_i, B_j ... \xrightarrow{d} B}$$

where Ω denotes the set of urgent gates ($\Omega = \{G_i \mid u(G_i) = true\}$)

where $B \xrightarrow{d} B'$ **refusing** $\{G_k\}$ is true iff:

$$\forall d' \in [0, d] \ \ \forall L \quad gate(L) \in \{G_k\} \ \wedge \ B_{d'} \not\xrightarrow{L}$$

and where $B \xrightarrow{G\ \vec{e}} B'$ **refusing** $\{G_k\}$ is true iff:

$$\forall L \quad gate(L) \in \{G_k\} \ \wedge \ B \not\xrightarrow{L}$$

5 Expressiveness of the proposed coroutine operator

To prove the interest of our proposed coroutine mechanism, we show that a number of E-LOTOS behaviour operators can be expressed simply in terms of the proposed coroutine operator.

In particular, we show that *suspend-resume and "***trap***" can both be derived from the coroutine mechanism.*

Note: all the following is only achieved in the framework of an untimed semantics. However, we believe (expect?) that these results could be extended to the timed framework. The freedom to declare the coroutines are urgent or not should provide the desirable flexibility to achieve this result.

5.1 Suspend-resume

It is clear that the proposed coroutine mechanism allows to specify suspend-resume behaviour. For instance, the behaviour depicted on the Figure of of Section 4.4 can be described as follows:

exec G : **none in**
G -> B_1
G -> B_2
endexec

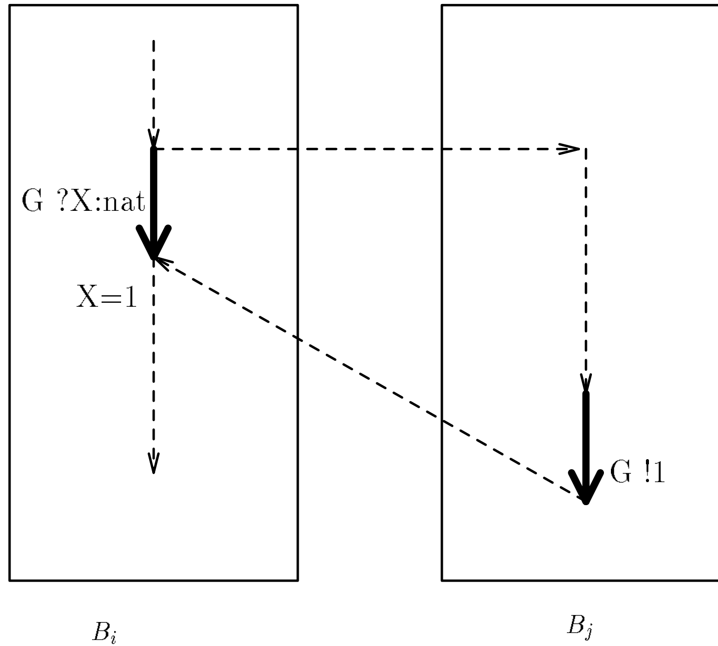
When comparing this solution to the existing proposals for suspend-resume [Her96], one can make the following comments:

- There is a single action for suspend and resume, instead of two different actions in [Her96].
- Suspend and resume are both atomic. This is the most powerful scheme. If needed, one can always add visible suspend/and resume. This can be done by replacing every gate G in B_1 by a sequential composition "`SUSPEND;` G`; RESUME`".
- Values can be passed between both processes, a feature that is mentioned in [Her96] but not tackled satisfactorily.

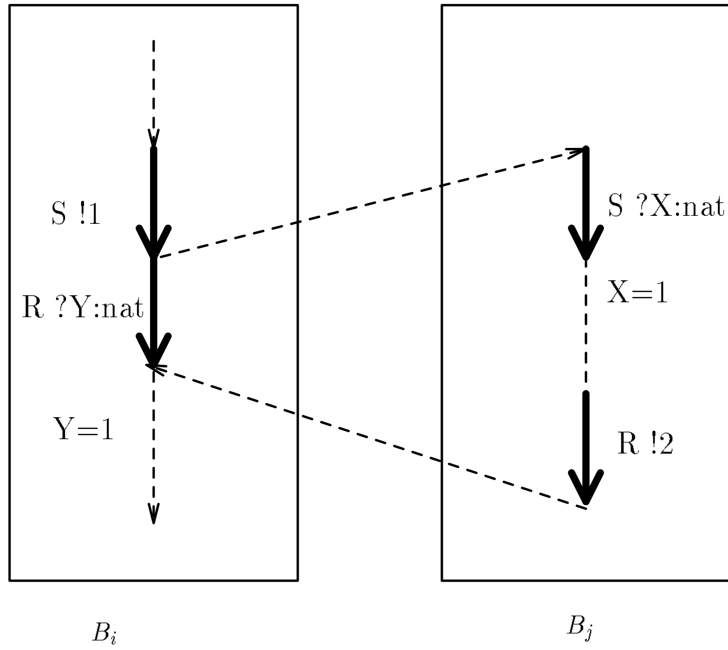
- The overall mechanism is similar to the LOTOS rendez-vous: synchronization is realized in a "uniprocessor" fashion; value communication is exactly the same as in LOTOS.

We give a few diagrams explaining some possible modes of value passing between coroutines.

In the first example, B_1 wants to input a value on the special gate G. It passes the control to another coroutine B_2, which executes until it is ready to output the requested value on gate G. The value is passed to B_1 and B_1 is resumed:



In the second example, when B_1 is suspended (gate S) and passes a value to B_2. Later, B_1 is resumed (gate R) and receives a value sent by B_2.



5.2 Trap

The "**trap**" operator defined in [GS96]:

trap
 G_1 $(\vec{V}_1 : \vec{S}_1)$ -> B_1
 ...
 G_n $(\vec{V}_n : \vec{S}_n)$ -> B_n
in
 B
endtrap

can be defined in terms of "**exec**":

exec $G_1 : \vec{S}_1, ..., G_n : \vec{S}_n$ **in**
 $G_1, ..., G_n$ -> B
 G_1 -> $(G_1$?$\vec{V}_1 : \vec{S}_1$; $B_1)$
 ...
 G_n -> $(G_n$?$\vec{V}_n : \vec{S}_n$; $B_n)$
endexec

5.3 Hide

The "**hide**" operator:

hide $G_1 \vec{S}_1, ..., G_n : \vec{S}_n$ **in** B

can be defined in terms of “**exec**”:

$$
\begin{array}{l}
\textbf{exec}\ G_1 \vec{S}_1, ..., G_n : \vec{S}_n\ \textbf{in} \\
\ G_1, ..., G_n\ \texttt{->}\ B \\
\ G_1\ \texttt{->}\ \textbf{loop i;}\ \ G_1\ \ ?\vec{V}_1 : \vec{S}_1\ \textbf{endloop} \\
\ ... \\
\ G_n\ \texttt{->}\ \textbf{loop i;}\ \ G_n\ \ ?\vec{V}_n : \vec{S}_n\ \textbf{endloop} \\
\textbf{endexec}
\end{array}
$$

or also:

$$
\begin{array}{l}
\textbf{exec}\ G_1 \vec{S}_1, ..., G_n : \vec{S}_n\ \textbf{in} \\
\ G_1, ..., G_n\ \texttt{->}\ B \\
\ G_1, ..., G_n\ \texttt{->}\ \textbf{loop i;}\ \ (G_1\ \ ?\vec{V}_1 : \vec{S}_1\ \ \texttt{[]}\ \ ...\ \ \texttt{[]}\ \ G_n\ \ ?\vec{V}_n : \vec{S}_n)\ \textbf{endloop} \\
\textbf{endexec}
\end{array}
$$

References

[Con63] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, July 1963.

[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. On the Introduction of Exceptions in LOTOS. In Reinhard Gotzhein and Jan Bredereke, editors, *Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'96 (Kaiserslautern, Germany)*, pages 469–484. IFIP, Chapman & Hall, October 1996.

[Her96] Christian Hernalsteen. Remarks on the introduction of time and suspend/resume operator in the core language. May 1996.

[ISO88] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, September 1988.

[KM77] G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. In *Proc. IFIP Congress 77*, pages 993–998. North-Holland, Amsterdam, 1977.

[PS80] W. Pauli and M. L. Soffa. Coroutine Behaviour and Implementation. *Software—Practice and Experience*, 10(3):189–204, Mars 1980.