30

On the Introduction of Exceptions in E-LOTOS*

Hubert Garavel, Mihaela Sighireanu
Inria Rhône-Alpes, Verimag, Miniparc-Zirst, rue Lavoisier,
F-38330 Montbonnot Saint-Martin, France
E-mail: Hubert.Garavel@imag.fr, Mihaela.Sighireanu@imag.fr

Abstract

The advantages of exception handling are well-known and several sequential and parallel programming languages include exception handling mechanisms. Unfortunately, none of the three standardized Formal Description Techniques (ESTELLE, LOTOS, and SDL) supports exceptions. In 1992, Quemada and Azcorra pointed out the need for structuring protocol descriptions with exceptions and proposed to extend LOTOS with a so-called "generalized termination and enabling" mechanism. In this paper, we show that their proposal is not fully appropriate for a compositional description of complex systems. We propose a simpler exception mechanism for LOTOS, for which we provide a syntactic and semantic definition. We show that this exception mechanism is very primitive, as it allows several existing LOTOS operators to be expressed as special cases. We also suggest additional operators, such as symmetric sequential composition and iteration, which are derived from the exception mechanism.

Keywords

Exceptions, Exception handling, E-LOTOS, Formal Description Techniques, LOTOS, Process Algebra, Protocols.

1 INTRODUCTION

The Formal Description Technique LOTOS [ISO88] is intended for the unambiguous definition of the functional behaviour of information processing systems. LOTOS combines sound semantics concepts (borrowed from the theories of algebraic data types and process algebras) with software engineering features intended for the design of complex systems. It provides a rich set of specification styles. Robust tools are now available, which support design, verification, and code generation.

LOTOS is currently considered for revision by ISO. This revision process should lead to an enhanced standard language named E-LOTOS (for Extended LOTOS). The E-LOTOS Committee is working actively to elaborate this revised standard, which should increase substantially the expressiveness and user-friendliness of

*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".

R. Gotzhein et al. (eds.), *Formal Description Techniques IX*

LOTOS. In this paper, we propose to extend LOTOS with an exception handling mechanism, which we consider to be a fundamental concept providing powerful structuring capabilities.

Exceptions are generally recognized as a desirable programming feature. They provide a structured mean for dealing with errors and other "abnormal" situations in computer programs. Because the notion of "abnormal" is highly subjective, exceptions are often used as a plain programming paradigm, even for processing "normal" situations. Although the details may vary from one computer language to another, the different exception mechanisms present strong similarities: exceptions are usually *named* with an identifier; they can be *raised* when an error or an abnormal situation occurs, which aborts the execution of the program part that caused the exception; when raised, they can be *caught*, meaning that another program part (called the *exception handler*) is activated in place of the aborted one.

Exception handling is supported by many modern sequential programming languages, either algorithmical (e.g. ADA), functional (e.g. ML), or object-oriented (e.g. C++, EIFFEL, JAVA). As regards parallel languages, the importance of exceptions (combined with concurrency) has been pointed out by Berry in the framework of the synchronous language ESTEREL [Ber93]. Berry's proposal was adapted to the framework of asynchronous process algebras by Nicollin and Sifakis in their Algebra of Timed Processes ATP [NS94].

Unfortunately, none of the three standardized Formal Description Techniques (ESTELLE, LOTOS, and SDL) supports exceptions. Communication protocols are often described in terms of communicating state machines, which often leads to poorly structured descriptions. This problem was identified by Quemada and Azcorra [QA92], who suggested that protocol descriptions could be better structured by expliciting interrupts between protocol phases. They proposed to enhance LOTOS with a mechanism providing a (limited) form of exception handling. Building upon their work, we propose an exception mechanism for LOTOS, which, we believe, is simpler and more expressive.

The paper assumes some basic knowledge of LOTOS. It is organized as follows. Section 2 presents the protocol example used in [QA92] to illustrate the need for exceptions; from this example, a list of technical requirements is given, which an appropriate exception mechanism for LOTOS should satisfy; then, the proposal of [QA92] is presented and assessed with respect to the aforementioned requirements. Section 3 presents our proposal and its applications to the protocol example given in Section 2. Section 4 gives some algebraic properties of the proposed exception mechanism. Section 5 shows that several existing LOTOS operators can be expressed in terms of exceptions, and suggests to introduce new operators defined as shorthand notations (i.e., syntactic sugar) above the proposed exception mechanism. Finally, Section 6 gives some concluding remarks and a list of open issues for further work.

2 MOTIVATIONS AND RELATED WORK

2.1 A protocol specification problem

The ABRACADABRA protocol is a sample protocol that exhibits typical features of OSI communication protocols. This protocol has been used to illustrate how the standardized Formal Description Techniques can be applied to real protocols. A formal description of this protocol in standard LOTOS can be found in [ISO91, Tur93]. However, it was pointed out by Quemada and Azcorra [QA92] that the ABRACADABRA could be described in a better way if LOTOS was extended with a new feature, which they called *generalized termination and enabling* and which is close to an exception mechanism. Using this proposed extension, they produced a description of the ABRACADABRA protocol which was better structured and much shorter than other descriptions in standard LOTOS.

Their proposal takes into account the particular behaviour of an ABRACADABRA protocol entity, which is abstracted on Figure 1 in the form of a block diagram, where the rectangular blocks represent the various protocol phases (which some refinement allowing certain blocks to be nested in other blocks) and where the dotted arrows going out from blocks represent the exceptions which can be raised during the corresponding phases of the protocol.

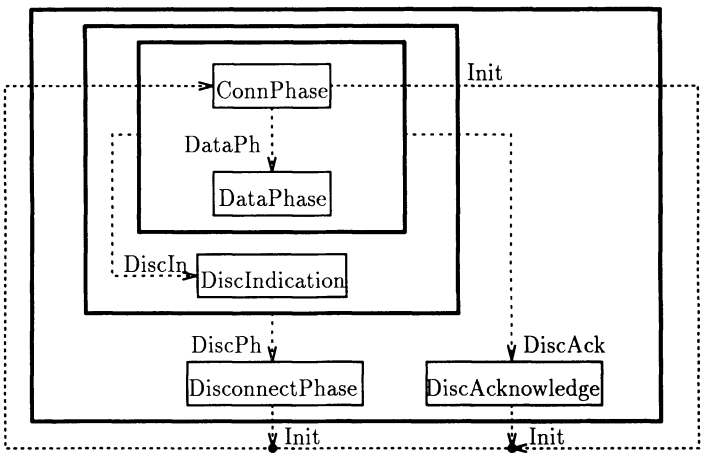

Figure 1 ABRACADABRA protocol

The ABRACADABRA protocol entity starts with the `ConnPhase`, which signals its termination by raising the `DataPh` exception, which enables the `DataPhase`. From any of these two phases, three exceptions can be raised: `DiscIn` enables the `DiscIndication` phase, `DiscAck` enables the `DiscAcknowledge` phase and `DiscPh`

enables the `DisconnectPhase`. The `DiscIndication` phase itself can be terminated by a `DiscPh` exception enabling the `DisconnectPhase`. The `Init` exception can be raised from `ConnPhase`, `DiscAcknowledge`, and `DisconnectPhase`: it restarts the protocol entity.

Although the use of an exception mechanism could be avoided by using standard LOTOS constructs, it clearly improves the readability and structuring of the ABRACADABRA protocol entity. The same remark also apply to algorithmic programming languages, such as ADA, where exceptions provide a structured mean to avoid cascades of "**if...then...else**" tests.

2.2 Requirements for an exception mechanism

From the ABRACADABRA example, we can list several important requirements that a proper exception mechanism for LOTOS should satisfy:

- **(R1)** The same behaviour may raise several exceptions: for example, the `ConnPhase` can raise `Init`, `DataPh`, `DiscIn`, `DiscAck` and `DiscPh` exceptions. Consequently, we need a mechanism which allows multiple exceptions to be caught.
- **(R2)** Exception handling should be done hierarchically, with different levels of nesting. In the ABRACADABRA protocol, we see four levels of exception handling: one for the `DataPh`, one for `DiscIn`, one for `DiscPh` and `DiscAck`, and the last one for `Init`; we can notice that exception `DiscPh` can be raised from two different levels.
- **(R3)** Sequential composition should be a particular form of exception handling, in which one behaviour terminates by raising an exception, which enables another behaviour. This is the case with the `ConnPhase`, which terminates with exception `DataPh`, thus enabling the `DataPhase`.

Although the ABRACADABRA protocol entity is a purely sequential behaviour, we should also mention additional requirements, which relate exception handling and parallel composition.

- **(R4)** Exceptions should be statically *scoped*, meaning that an exception could not propagate outside the scope of its definition (a problem that exists in ML, for instance).
- **(R5)** Exceptions should be allowed to carry data, meaning that it should be possible, when raising a exception, to specify values that will be passed to the corresponding exception handler.
- **(R6)** It should be possible to synchronize exceptions in parallel composition, meaning that several concurrent behaviours could agree to raise an exception if and only if all of them are ready to do so. This is implied by requirement (R3) and the need for compatibility with standard LOTOS, in which the termination of concurrent behaviours is always synchronous: all behaviours have to terminate simultaneously by executing an "**exit**" statement, which corresponds to a rendezvous on the special gate "δ".

- **(R7)** Conversely, it should also be possible *not* to synchronize exceptions in parallel composition, thus allowing the execution of a set of concurrent behaviours to be aborted as soon as one of these behaviours raises an exception (which will enable some continuing behaviour). The practical need for such *asynchronous termination* has been already pointed out by [QA92].
  Moreover, when considering *constraint-oriented* specification style [VSS88], one can also imagine situations in which synchronized and non-synchronized exceptions are both necessary. For instance, let us consider the situation in which a behaviour B is refined into two parallel sub-behaviours B_1 and B_2. Each sub-behaviour B_i raises two exceptions: X and X_i. The exception X is synchronized: B_1 and B_2 have to agree to raise X. On the other hand, exceptions X_1 and X_2 are non-synchronized: they can be raised independently, have different handlers, and can disrupt the whole behaviour B. More generally, synchronized exceptions correspond to an *and*-like composition, whereas non-synchronized exceptions correspond to an *or*-like composition; this presents strong similarities with the way constraints are composed together in LOTOS, with an obligation to synchronize or to interleave.
- **(R8)** We are convinced that *exceptions* should be the same concept as LOTOS *gates*. We therefore follow the same approach as in ESTEREL [Ber93], where exceptions are nothing but a special case of *signals*. In a first attempt to introduce exceptions in LOTOS [Que96, Annex F, part 2], we proposed to create two different syntactic categories for gates and exceptions. This approach led to a number of syntactic and semantic complications, among which: (1) the need for having a new operator to raise exceptions; (2) the need for extending the general parallel composition operator with a list of exceptions to be synchronized (in addition to the list of gates to be synchronized that already exists in LOTOS); (3) the need for extending process definitions and instantiations with a list of exception parameters (in addition to the lists of gate parameters and value parameters); (4) the need for introducing a notion of "exception typing", similar but different from the notion of "gate typing" [Gar95]; (5) the need for having two different kinds of transitions in the operational semantics (those labelled with a gate and those labelled with an exception); (6) the problem of deciding whether the "δ" gate used for sequential composition in LOTOS should be represented as a gate or as an exception. Moreover, a syntactic separation between gates and exceptions leads to a loss of expressiveness and convenience: for instance, it prevents to have an exception handler be executed when some gate occurs in a given behaviour. Also, the specifier has to make an early choice to decide whether an action has to be implemented by a gate or an exception.

2.3 Discussion of the proposal by Quemada and Azcorra

The solution proposed in [QA92] can be evaluated with respect to the requirements listed in the previous section:

- It does not satisfy requirement (R8), because a distinction is made between the concepts of gates and exceptions (which are called *terminations* in [QA92]).

Although gates and terminations are distinct, they can be combined together by means of a product operator noted "$G\ v_1...v_n \star X$", which expresses that a LOTOS action with gate G and value offers $v_1, ..., v_n$ happens as the same time as a termination X. This notion of simultaneous occurrence of gates and terminations is meant to solve the *intermediate state problem* mentioned in [Bri88], but at the price of introducing *compound events*, which require deep semantic changes. As gate typing strongly reduces the need for other forms of compound events [Gar95], it is not sure whether compound events are desirable for E-LOTOS.

- It satisfies requirement (R3): a *generalized enabling* operator, noted "B_1 >X> B_2" is introduced with the following meaning: B_1 executes and, as soon as it performs the termination X, it is aborted and the control is transferred to B_2. This operator is an extension of the existing enabling operator ">>" in LOTOS.
- It satisfies requirement (R4), because "B_1 >X> B_2" declares a termination X that is only visible in B_1. If raised, this termination will be necessarily caught and will not escape outside of its scope (i.e., B_1).
- It satisfies requirement (R5): although a termination cannot carry data, the gate to which it is combined (using the "$\star$" operator) can carry value offers, which gives more or less the expected meaning.
- It satisfies requirements (R6) and (R7), as the parallel composition operator of LOTOS is extended with a list of terminations to be synchronized.
- It satisfies requirements (R1) and (R2), but with strong limitations due to the fact that the generalized enabling "B_1 >X> B_2" is a binary operator handling a single exception. This operator cannot handle multiple exceptions at the same level, and therefore lacks compositionality.
  For instance, let us consider a simple example in which a behaviour B can raise two exceptions X_1 and X_2 having for respective exception handlers B_1 and B_2. According to the proposal made in [QA92], there are two possible ways of specifying this situation, either "(B >X_1> B_1) >X_2> B_2" or "(B >X_2> B_2) >X_1> B_1". None of these solutions is satisfactory, because some arbitrary nesting of the binary operators has to be fixed, which does not model accurately the situation. The first solution declares X_2 to be visible in B and B_1; the second solution declares X_1 to be visible in B and B_2. In both cases, this is not the expected control flow, as X_1 and X_2 should only be visible in B.
  In the ABRACADABRA protocol entity example, the same situation occurs with the exceptions `DiscAck` and `DiscPh` raised by the `ConnPhase` and the `DataPhase`. Although these exceptions are raised at the same level and should be handled at the same level, the solution proposed by [QA92] establishes an artificial hierarchy between them, by nesting the "`>DiscPh>`" operator inside the left-hand side of the "`>DiscAck>`".

3 INTRODUCING EXCEPTIONS IN LOTOS

3.1 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.

$B, B_1, B_2, ...$ 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).

3.2 Definition of the "trap" operator

To extend LOTOS with an exception mechanism that satisfies requirements (R1) to (R8), we introduce a new behaviour operator, whose syntax is the following:

trap
  $G_1\ \vec{V_1}$ -> B_1
  ...
  $G_n\ \vec{V_n}$ -> B_n
in
  B
endtrap

In this operator, B correspond to the "normal" behaviour; $G_1, ..., G_n$ are gates representing exceptions that can be raised from B; a list of formal parameters $\vec{V_i}$ and an exception handler B_i is attached to each G_i. The fragment of text located between the "**trap**" and "**in**" keywords will be called the *handling part.* As regards static semantics, we have the following rules:

- The occurrences of gate identifiers G_i in the handling part are *definition*-occurrences*. The gates G_i declared in the handling part are only visible in B. They must be pairwise distinct and different from the invisible gate "**i**". Some

*also called *binding occurrences* in [ISO88]

G_i can be equal to "δ" (in order to handle the successful termination of LOTOS). The gates G_i are typed according to the proposal for *gate typing* [Gar95]: when a gate G_i is used in B, its list of offers should be compatible, in number and types, with the list of variables $\vec{V}_i$. The *gate overloading* feature [Gar95] is not allowed for gates declared in the handling part of a "**trap**" operator.

- The occurrences of variable declarations $\vec{V}_i$ attached to gates G_i are *definition*-occurrences: the variables declared in $\vec{V}_i$ are only visible in the behaviour expression B_i.

Many languages (e.g. ADA, ML, ATP, ESTEREL) place the handling part after the normal behaviour (B here), because they put the emphasis on the usual processing rather than on the abnormal processing. We have chosen the opposite solution because we want the definitions of gates G_i to precede their uses in B. We hereby follow the example of LOTOS' "**let**" and "**hide**" operators.

Informally, the "**trap**" operator behaves like a "watchdog". The "normal" behaviour B is executed. When B performs any action of the form "$G_i\ \vec{e}$" (where gate G_i is declared in the handling part), then B is aborted and the exception handler B_i associated to G_i is executed, after values $\vec{e}$ have been assigned to the formal parameters $\vec{V}_i$ of G_i.

Formally, the dynamic semantics of LOTOS is given by means of a *Labelled Transition System*. A *transition* is a triple (B_1, L, B_2), where B_1 and B_2 are behaviour expressions and L is label having the form "$G\ \vec{e}$". We keep the same definition of label equality as in LOTOS: two labels are equal if they have the same gate G and carry the same values $\vec{e}$.

The transition relation for the "**trap**" operator is defined by the following rules:

$$\frac{B \xrightarrow{G\ \vec{e}} B' \ \wedge \ G \notin \{G_1, ..., G_n\}}{\left(\begin{array}{l}\textbf{trap}\\ \quad G_1\ \vec{V}_1 \rightarrow B_1\\ \quad ...\\ \quad G_n\ \vec{V}_n \rightarrow B_n\\ \textbf{in}\\ \quad B\\ \textbf{endtrap}\end{array}\right) \xrightarrow{G\ \vec{e}} \left(\begin{array}{l}\textbf{trap}\\ \quad G_1\ \vec{V}_1 \rightarrow B_1\\ \quad ...\\ \quad G_n\ \vec{V}_n \rightarrow B_n\\ \textbf{in}\\ \quad B'\\ \textbf{endtrap}\end{array}\right)}$$

$$\frac{B \xrightarrow{G_i\ \vec{e}} B' \ \wedge \ [\vec{e}/\vec{V}_i]\ B_i \xrightarrow{L} B'_i}{\left(\begin{array}{l}\textbf{trap}\\ \quad G_1\ \vec{V}_1 \rightarrow B_1\\ \quad ...\\ \quad G_n\ \vec{V}_n \rightarrow B_n\\ \textbf{in}\\ \quad B\\ \textbf{endtrap}\end{array}\right) \xrightarrow{L} B'_i} \quad [i \in 1..n]$$

The first rule defines the normal execution of B. The remaining rules describe exception handling (for a "**trap**" operator with gates $G_1, ..., G_n$, there are n such

rules; the number of such rules for a given extended LOTOS description is always finite). The handling of an exception G_i involves an *atomic* control passing, meaning that no transition labelled either by G_i or by the invisible gate "**i**" is performed before the first transition L of the exception handler B_i is executed.

3.3 An application example

Using the proposed "**trap**" operator, we can describe the ABRACADABRA example mentioned in Section 2 (the "..." notation is used as a shorthand for the other gate parameters and value parameters used in [QA92]):

```
process AbracadabraProtocolEntity [...] :=
    trap
       Init -> AbracadabraProtocolEntity [...]
    in
       trap
           DiscPh -> DisconnectPhase [Init, ...] (...)
           DiscAck -> DiscAcknowledge [Init, ...]
        in
           trap
               DiscIn -> DiscIndication [DiscPh, ...]
           in
               trap
                   DataPh -> DataPhase [DiscIn, DiscPh, DiscAck, ...]
               in
                   ConnPhase [Init, DataPh, DiscIn, DiscPh, DiscAck, ...]
               endtrap
           endtrap
        endtrap
    endtrap
endproc
```

4 ALGEBRAIC PROPERTIES OF THE "TRAP" OPERATOR

In this Section, we give some properties of the "**trap**" operator with respect to the strong bisimulation equivalence [Par81] (which is noted "$\sim$" below). For concision, we note H the handling part of a "**trap**" operator and we omit the "**endtrap**" keyword. The proofs are given in [GS96].
When adding the "**trap**" operator to LOTOS, strong bisimulation remains a congruence:

$(B' \sim B'') \implies (\textbf{trap } H \textbf{ in } B') \sim (\textbf{trap } H \textbf{ in } B'')$
$(B'_i \sim B''_i) \implies (\textbf{trap } ...G_i\ \vec{V}_i \textbf{ -> } B'_i ... \textbf{ in } B) \sim (\textbf{trap } ...G_i\ \vec{V}_i \textbf{ -> } B''_i ... \textbf{ in } B)$

The "**trap**" operator distributes over non-deterministic choice:
$(\textbf{trap } H \textbf{ in } (B_1 \ [] \ B_2)) \sim (\textbf{trap } H \textbf{ in } B_1) \ [] \ (\textbf{trap } H \textbf{ in } B_2)$

In some cases, the "**trap**" operator commutes with action-prefix:
$(G \neq G_i) \Longrightarrow$ (**trap** $G_i\ \vec{V}_i$ **->** B_i **in** $(G\ \vec{e}\ ;\ B)) \sim (G\ \vec{e}\ ;$ **trap** $G_i\ \vec{V}_i$ **->** B_i **in** $B)$
$(G = G_i) \Longrightarrow$ (**trap** $G_i\ \vec{V}_i$ **->** B_i **in** $(G\ \vec{e}\ ;\ B)) \sim ([\vec{e}_i/\vec{V}_i]\ B_i)$

The following laws allow simplifications for "**stop**":
(**trap** H **in stop**) $\sim$ **stop**
$(G \neq \delta) \Longrightarrow$ (**trap** $(G\ \vec{V}$ **->** **stop**$)\ H$ **in** $B) \sim$ (**trap** H **in** $(B$ **|[**G**]| exit** $\mathcal{F}$**))**
where $\mathcal{F}$ denotes a list of "**any** S" clauses compatible, in number and types, with the functionality of B (i.e., the list of values returned by B upon exit).

5 OPERATORS DERIVED FROM THE "TRAP" OPERATOR

This section explains how the proposed "**trap**" operator allows to express several existing LOTOS operators as derived cases (shorthand notations) and to introduce new operators of practical interest (including the generalized enabling defined in [QA92]). The detailed proofs that the existing LOTOS operators are equivalent to their proposed translation in terms of "**trap**" can be found in [GS96].

5.1 Enabling operator

The "**>>**" operator of LOTOS has the following equivalent translation:

$$B_1 \text{ >> [accept } \vec{V} \text{ in] } B_2 \quad \sim \quad \left(\begin{array}{l} \textbf{trap} \\ \quad \delta\ \vec{V} \text{ -> i ; } B_2 \\ \textbf{in} \\ \quad B_1 \\ \textbf{endtrap} \end{array} \right)$$

In LOTOS, the "**>>**" operator is a primitive one, since it cannot be exactly expressed using parallel composition and hiding. The reason for this is the problem to give different names to the δ gate in case of nested "**>>**" operators (see [Gar89, chapter 2] for a discussion). Introducing the "**trap**" operator solves this problem elegantly: it is not necessary to hide "δ", because no δ-action can be observed from the outside since exception handling is atomic.

5.2 Disabling operator

The "**[>**" operator of LOTOS has the following equivalent translation:

$$B_1 \text{ [> } B_2 \quad \sim \quad \left(\begin{array}{l} \textbf{trap} \\ \quad \xi \text{ -> } B_2 \\ \textbf{in} \\ \quad B_1 \text{ ||| (exit } \mathcal{F} \text{ [] } \xi) \\ \textbf{endtrap} \end{array} \right)$$

where ξ is special gate identifier (not used in B_1) and where $\mathcal{F}$ denotes a list of "**any** S" clauses compatible, in number and types, with the functionality of B_1 (i.e., the list of values returned by B_1 upon exit). This definition deserves a few comments:

- The gate ξ is not synchronized by the parallel operator and, therefore, can be spontaneously triggered at any time; if so, the execution of B_1 is aborted and the control flow is transferred to B_2. However, B_1 can also execute normally; if B_1 reaches an "**exit**" statement (also proposed by the right operand of the parallel process), then the δ gate is triggered and propagated outside, because it is not caught by the "**trap**" operator.
- Omitting the "**exit** $\mathcal{F}$" alternative on the right hand-side of the parallel operator would prevent B_1 from terminating successfully, as the "δ" gate is always synchronized in parallel composition.
- Of course, the very useful watchdog construct "$(B_1$ [> $B_2)$ >> B_3" can still be obtained as a particular form of "**trap**". But the "**trap**" operator allows more general forms of watchdog, in which several actions, leading to different behaviours, can be used to escape the watchdog (in LOTOS, only the "δ" action can be used).

5.3 Generalized enabling

The generalized enabling operator proposed in [QA92] can be derived from the "**trap**" operator (we do not consider here the concept of compound events discussed in Section 2.3):

$$B_1 >X> B_2 \quad \sim \quad \textbf{trap } X \textbf{ -> } B_2 \textbf{ in } B_1 \textbf{ endtrap}$$

The proposed "**trap**" operator is more expressive than the one of [QA92], since it allows multiple gates to be handled at the same level. For instance, the problem mentioned in Section 2.3 can be solved by writing simply:

trap
  X_1 **->** B_1
  X_2 **->** B_2
in
  B
endtrap

5.4 Another sequential composition operator

In addition to the definition of ">>" as a shorthand, we suggest to introduce another sequential composition operator, noted ";", inspired from the sequential composition of ACP [BK84]. A restricted form of this operator, without value passing, can be defined as follows:

$$B_1 \ ; \ B_2 \ \sim_{def} \ \left(\begin{array}{l} \textbf{trap} \\ \quad \delta \rightarrow B_2 \\ \textbf{in} \\ \quad B_1 \\ \textbf{endtrap} \end{array} \right)$$

This sequential operator is more primitive than the existing operator ">>" of LOTOS, since "B_1 >> B_2 $\sim$ B_1 ; i ; B_2". Moreover, this new operator has several nice properties: (a) it is associative, as a consequence of the fact that "**trap**" removes gates when they are caught; (b) it has "**exit**" for neutral element (on its left-hand and right-hand sides), whereas the ">>" operator has no neutral element; (c) it has "**stop**" for absorbing element on its left-hand side; (d) it implements a fully atomic and invisible sequential composition, on the opposite of the ">>" operator, which generates an internal action "i" when continuation passing occurs (this has the unpleasant effect of increasing the sizes of the labeled transition systems generated from LOTOS programs, thus contributing to state explosion without any practical benefit from the specifier' point of view).

An extended form of this operator can be defined, which enables B_1 to pass a list of values to B_2 using the "**exit**" statement (this is similar to the "**accept**" clause of the ">>" operator):

$$B_1 \ ; \ \textbf{accept} \ \vec{V} \ \textbf{in} \ B_2 \ \sim_{def} \ \left(\begin{array}{l} \textbf{trap} \\ \quad \delta \ \vec{V} \rightarrow B_2 \\ \textbf{in} \\ \quad B_1 \\ \textbf{endtrap} \end{array} \right)$$

5.5 Choice operator

The "[]" operator of LOTOS has the following equivalent translation:

$$B_1 \ \Box \ B_2 \ \sim \ \left(\begin{array}{l} \textbf{trap} \\ \quad G_1 \rightarrow B_1 \\ \quad G_2 \rightarrow B_2 \\ \textbf{in} \\ \quad (G_1 \ ; \ \textbf{stop} \ ||| \ G_2 \ ; \ \textbf{stop}) \\ \textbf{endtrap} \end{array} \right)$$

where G_1 and G_2 are two new gate identifiers not used in B_1 or B_2. Intuitively, this translation can be justified as follows: according to the semantics of the interleaving operator "|||", one gate G_i is triggered non-deterministically and caught by the "**trap**" operator, which aborts the parallel composition and enables the execution of the corresponding exception handler B_i.

5.6 Choice-over-values operator

The "**choice**" operator of LOTOS has the following equivalent translation:

$$\textbf{choice } V_1 : S_1, ..., V_n : S_n \ \square \ B_0 \ \sim \left(\begin{array}{l} \textbf{trap} \\ \quad G\ (V_1 : S_1, ..., V_n : S_n) \rightarrow B_0 \\ \textbf{in} \\ \quad G\ ?V'_1 : S_1\ ...\ ?V'_n : S_n\ ;\ \textbf{stop} \\ \textbf{endtrap} \end{array} \right)$$

where G is a new gate identifier not used in B_0. Intuitively, this translation relies on the semantics of the LOTOS clause "$?V : S$", which performs a non-deterministic selection of a value in the domain of sort S and assigns this value to the variable V. The resulting behaviour is B_0 in which variables $V_1, ..., V_n$ are bound to values generated non-deterministically.

5.7 Iteration operator

Many reactive systems exhibit cyclical behaviours. In most sequential languages, this can be described using either iteration or recursion. In LOTOS, however, only recursion is available: all cyclical behaviours have to be described using recursive processes. We therefore propose to introduce a "functional" iterator in E-LOTOS, which is merely a shorthand notation, defined using a recursive process and the "**trap**" operator.

We note "θ" a special gate identifier, which (informally) expresses a branch to the loop entry. We then introduce two new operators, "**continue**" and "**loop**" defined as follows (square brackets "[...]" denote optional elements):

$$\textbf{continue } [(E_1, ..., E_n)] \ \sim_{def} \ \theta\ [!E_1\ ...\ !E_n]\ ;\ \textbf{stop}$$

and:

$$\left(\begin{array}{l} \textbf{loop } [V_1 : S_1 := E_1, ..., V_n : S_n := E_n \textbf{ in}] \\ \quad B_0 \\ \textbf{endloop} \end{array} \right) \ \sim_{def} \ P\ [\mathcal{G}]\ [(E_1, ..., E_n)]$$

where $\mathcal{G}$ denotes the set of gates visible in the behaviour expression B_0 and where P is a new process identifier whose definition is:

$$\begin{array}{l} \textbf{process } P\ [\mathcal{G}]\ [(V_1 : S_1, ..., V_n : S_n)] \\ \quad \textbf{trap} \\ \quad\quad \theta\ [(V'_1 : S_1, ..., V'_n : S_n)] \rightarrow P\ [\mathcal{G}]\ [(V'_1, ..., V'_n)] \\ \quad \textbf{in} \\ \quad\quad B_0 \\ \quad \textbf{endtrap} \\ \textbf{endproc} \end{array}$$

The "**loop**" operator is used to repeat infinitely a given behaviour B_0. Optionally, it allows values to be computed in the loop and transmitted from one iteration to the next one. These values are stored in variables $V_1, ..., V_n$ whose initial values are $E_1, ..., E_n$ respectively. In the loop body B_0, the occurrence of a "**continue** $[(E_1, ..., E_n)]$" operator has the effect of assigning the values of

$E_1, ..., E_n$ to $V_1, ..., V_n$ respectively, and to start a new iteration† by triggering the θ gate. Finally, the "**exit**" operator of LOTOS can be used to go out of the loop (it triggers the "δ" gate, which is not caught by the "**trap**" operator).

For instance, the following behaviour reads a stream of values on its `INPUT` gate until the sum of these values exceeds 1000 (in which case, it returns the number of values he has read):

```
loop COUNT:NAT := 0, SUM:REAL := 0 in
   INPUT ?Xi:REAL;
      if (SUM + Xi > 1000) then
         exit (COUNT + 1)
      else
         continue (COUNT + 1, SUM + Xi)
endloop
```

6 CONCLUSION AND OPEN ISSUES

As pointed out in [QA92], the description of communication protocols and services can be improved by the use of an exception mechanism. Although exceptions are available in many computer languages, they do not exist in any of the three standardized Formal Description Techniques (ESTELLE, LOTOS, SDL).

Our work builds upon a proposal for extending LOTOS with a *generalized termination and enabling* mechanism [QA92]. Noticing that this mechanism was not fully appropriate for a compositional description of multiple exceptions, we have proposed a different, simpler mechanism for LOTOS, consisting in a new "**trap**" operator, for which we have given a syntax, a static semantics and a dynamic semantics. We have shown that our proposal generalizes the one by [QA92], by allowing several exceptions to be handled at the same level.

We have proved that our proposal is a consistent extension of LOTOS, so that strong bisimulation remains a congruence after the "**trap**" is added to LOTOS. We have also studied some algebraical properties of the "**trap**" operator.

We have shown that the complexity added by the "**trap**" operator is greatly compensated by simplifications, as several existing LOTOS operators ("`>>`", "`[>`", "`[]`", "**choice**") can be obtained as shorthand notations. Moreover, our proposal allows to extend LOTOS with a symmetric, atomic sequential composition operator and a loop iterator, which are both missing in the language.

At this point, it seems that consensus exists in the E-LOTOS Committee to introduce such a "**trap**" operator into E-LOTOS. However, a number of issues are still open, which are currently under study:

- The E-LOTOS Committee has decided to replace the LOTOS abstract data types with a functional data language (see [JGL+95] for a discussion about this topic). This data language should include an exception handling mechanism similar at

†This operator has the same effect as the "**continue**" instruction of the C language (value assignment excepted)

the one of ML. It is therefore desirable to design a common exception handling for the behaviour part and the data part of E-LOTOS, so that exceptions generated in the data part (during evaluation of expressions) can be handled in the behaviour part.

- The E-LOTOS Committee has decided to extend LOTOS with the concept of quantitative time (an example of such a proposal can be found in [LL93]). In this paper, we have based the definition of our "**trap**" operator upon the existing, untimed LOTOS semantics. Adapting the "**trap**" operator to a timed semantics may require further work.
  In addition to the known reasons for introducing time in LOTOS, we foresee a particular advantage in using a timed semantics. Because the proposal for the "**trap**" operator given here remains into the asynchronous framework of LOTOS, it can only model *may*-interruptions, not *must*-interruptions [Ber93]. Using a timed semantics would improve the expressiveness of our exception mechanism, as the notion of *urgent actions* proposed in [LL93] allows *must*-interruptions (this is already the case with ATP [NS94], where exceptions are modelled by urgent actions).
- The E-LOTOS Committee is also considering various proposals for a "**suspend-resume**" operator, which would provide for interrupts with return. One may wonder whether the "**trap**" operator, which models interrupt without return, could not be merged with a "**suspend-resume**" operator. Another possible approach would consist in defining a general *coroutine* mechanism, from which the "**trap**" and "**suspend-resume**" operators could be derived as special cases.

ACKNOWLEDGEMENTS

The authors are grateful to Xavier Nicollin, Radu Mateescu, the four anonymous referees, and all the members of the ISO/IEC JTC1/SC21/WG7 E-LOTOS Committee chaired by Juan Quemada, for their helpful comments and suggestions.

REFERENCES

[Ber93] Gérard Berry. Preemption and Concurrency. In *Proceedings of FSTTCS 93*, volume 761 of *Lecture Notes in Computer Science*, pages 72–93, Berlin, 1993. Springer Verlag.

[BK84] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Computation*, 60:109–137, 1984.

[Bri88] Ed Brinksma. *On the Design of Extended LOTOS, a Specification Language for Open Distributed Systems.* PhD thesis, University of Twente, November 1988.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS.* Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[Gar95] Hubert Garavel. On the Introduction of Gate Typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP Interna-*

tional Workshop on Protocol Specification, Testing and Verification (Warsaw, Poland), London, June 1995. IFIP, Chapman & Hall.

[GS96] Hubert Garavel and Mihaela Sighireanu. On the Introduction of Exceptions in LOTOS. Rapport Recherche INRIA, INRIA, Grenoble, April 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.

[ISO91] ISO/IEC. Guidelines for the Application of Estelle, LOTOS and SDL. Technical Report 10167, International Organization for Standardization — Open Systems Interconnection, Genève, 1991.

[JGL+95] Alan Jeffrey, Hubert Garavel, Guy Leduc, Charles Pecheur, and Mihaela Sighireanu. Towards a proposal for datatypes in E-LOTOS. Annex A of ISO/IEC JTC1/SC21 N10108 Second Working Draft on Enhancements to LOTOS. Output document of the edition meeting, Ottawa (Canada), July, 20–26, 1995, October 1995.

[LL93] Luc Léonard and Guy Leduc. An Enhanced Version of Timed LOTOS and its Application to a Case Study. In Richard L. Tenney, Paul D. Amer, and M. Umit Uyar, editors, *Proceedings of the 6th International Conference on Formal Description Techniques FORTE'93 (Boston, MA, USA)*, pages 483–498, Amsterdam, October 1993. North-Holland.

[NS94] Xavier Nicollin and Joseph Sifakis. The Algebra of Timed Processes ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[Par81] David Park. Concurrency and Automata on Infinite Sequences. In Peter Deussen, editor, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, March 1981. Springer Verlag.

[QA92] J. Quemada and A. Azcorra. Structuring Protocols with Exception in a LOTOS Extension. In *Proceedings of the 12th IFIP International Workshop on Protocol Specification, Testing and Verification (Orlando, Florida, USA)*, Amsterdam, June 1992. IFIP, North-Holland.

[Que96] Juan Quemada, editor. Revised Working Draft on Enhancements to LOTOS (V3). ISO/IEC JTC1/SC21/WG7 N1053 Project 1.21.20.2.3. Output document of the Liège meeting, March 1996.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques – An Introduction to ESTELLE, LOTOS, and SDL.* John Wiley, 1993.

[VSS88] C. Vissers, G. Scollo, and M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In S. Aggarwal and K. Sabnani, editors, *Proceedings of the 8th International Workshop on Protocol Specification, Testing and Verification (Atlantic City, NJ, USA)*, pages 189–204, Amsterdam, 1988. IFIP, North-Holland.