French-Romanian Proposal for Capture of Requirements and Expression of Properties in E-LOTOS Modules*

Version 1.0

Radu MATEESCU Hubert GARAVEL
INRIA Rhône-Alpes
VERIMAG — Miniparc-ZIRST
rue Lavoisier
38330 MONTBONNOT ST MARTIN
FRANCE

May 1996

Abstract

It has been agreed by the E-LOTOS Committee that ACTONE algebraical equations should be kept in the E-LOTOS modules language to express the properties of the entities declared in the data part. This document proposes a similar scheme to express properties of the entities declared in the behaviour part. Our proposal is based on two state-of-the-art techniques for expressing such properties: equivalences and preorder relations between Labelled Transition Systems and temporal logics. This extension will provide E-LOTOS with powerful capabilities for capturing requirements in the specification phase and expressing properties to be checked by verification tools.

Contents

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

1 Introduction

This paper proposes a mechanism allowing to express requirements in E-LOTOS descriptions. Our proposal is based upon the following considerations:

- It has been agreed by the E-LOTOS Committee that ACTONE algebraical equations should be kept in the E-LOTOS modules language to express the properties of the entities declared in the data part. Syntactically, these equations could occur as an optional part, either in function declarations, e.g.:

  ```
  function F [...] (...) : ... is
     ...
  eqns
     <equations>
  endfunc
  ```

  or in module signatures and/or implementations, e.g.:

  ```
  module M ... is
     ...
  eqns
     <equations>
  endmod
  ```

  It has also been decided that, in the formal definition of E-LOTOS, these equations should be considered as "comments", which do not affect the dynamic semantics of E-LOTOS descriptions in which they occur. However, the equations should be checked, with respect to syntax and static semantics. Some tools may even want to go further, by (trying to) prove that the invariant properties expressed by the equations are valid.

  Such equations are helpful for capturing requirements in the specification phase. For instance, it is possible to specify the expected properties of a function before implementing this function actually. In other words, one should be able to specify "what" must be done before "how" it can be done. This approach is well-known in "literate programming" context.

- It has also been agreed that the data part and the behaviour part of E-LOTOS should exhibit, whenever it is possible, a nice symmetry. For instance, it was agreed that E-LOTOS modules should embody process definitions, as well as type and function definitions. Some recent proposals for defining the "core language" semantics even suggest a syntactic and semantic unification between the data and behaviour part.

  For symmetry, it would be very suitable to provide means for expressing properties on behaviour expressions and processes. Although they wouldn't affect the dynamic semantics of E-LOTOS programs, such requirements could be extracted and checked automatically using specialized verification tools.

- We must notice that such features are already present in other FDTs, for example SDL. Indeed, the SDL definition [IT92] comes with a graphical language called MSC (for *Message Sequence Charts*) to express requirements about SDL descriptions. The MSC descriptions define execution sequences and can be used by SDL verification tools to check whether the SDL descriptions can perform such execution sequences.

  Therefore, it is mandatory for E-LOTOS to offer, at least, the same capabilities as SDL. We believe that E-LOTOS should even go further than MSCs by embodying state-of-the-art techniques for expressing such properties. Assuming that E-LOTOS semantics (as LOTOS semantics) will be based upon the *Labelled Transition System* model (LTS for short), we naturally propose

the use of the two most popular formalisms for specifying LTS properties: equivalences (and preorder) relations, and temporal logics.

The paper is structured as follows. Section 2 defines the places where properties could be allowed in E-LOTOS. Section 3 discusses the algebraical equations used for expressing data properties. Section 4 discusses the use of equivalence and preorder relations for expressing behaviour properties. Section 5 proposes a complete solution, based on the propositional μ-calculus and the branching-time temporal logic ACTL. Finally, we give some concluding remarks.

2 Data and behaviour properties in functions, processes, and modules

We propose to extend function definitions with an optional section (introduced by the "**eqns**" keyword) containing algebraical equations. A precise syntax of the "`<equations>`" symbol will be proposed in Section 3.

```
function F [...] (...) : ... is
   ...
eqns
   <equations>
endfunc
```

We propose to extend process definitions with two optional sections introduced respectively by the "**rels**"and "**props**" keywords[1]. A definition of the "`<relations>`" and "`<properties>`" symbols will be given in Sections 4 and 5 respectively.

```
process P [...] (...) : ... is
   ...
rels
   <relations>
props
   <properties>
endproc
```

Finally, we propose to extend modules with the three above sections, each of them being optional.

```
module M ... is
   ...
eqns
   <equations>
rels
   <relations>
props
   <properties>
endmod
```

3 Algebraical equations for data properties

We believe that the "**eqns**" section should be highly compatible with existing ACTONE equations.

[1] Other keywords might be preferred, e.g., "**laws**", "**forms**", etc.

On the other hand, we must take into account the fact that value expressions might raise exceptions. We therefore propose the following syntax for the "`<equations>`" symbol referenced in Section 2:

$$
\begin{array}{lcl}
\texttt{<equations>} & ::= & [\textbf{ofsort}\ S]\ E_0\ \ ; \\
& | & [\textbf{ofsort}\ S]\ E_1\ \texttt{=}\ E_2\ \ ; \\
& | & [\textbf{ofsort}\ S]\ E_0\ \textbf{raises}\ X\ [(E_1, ..., E_n)]\ \ ;
\end{array}
$$

where S is a type identifier, X is an exception identifier, and where $E_0, ..., E_n$ are value expressions. Square brackets denote optional syntactic elements. These three lines have respectively the following, informal meanings:

1. E_0 is equal to *true*
2. E_1 and E_2 are equal
3. the evaluation of E_0 raises exception X with actual parameters $E_1, ..., E_n$.

The precise meaning of the above constructs will rely upon the design choices made for the data language.

4 Equivalences and preorders for behaviour properties

We propose the following syntax for the "`<relations>`" symbol referenced in Section 2:

$$
\begin{array}{lcl}
\texttt{<equations>} & ::= & B_1\ \texttt{=}\ B_2\ \textbf{mod}\ R\ \ ; \\
& | & B_1\ \texttt{<}\ B_2\ \textbf{mod}\ R\ \ ; \\
& | & B_1\ \texttt{>}\ B_2\ \textbf{mod}\ R\ \ ;
\end{array}
$$

where B_1 and B_2 are behaviour expressions, and where R is the name of an equivalence or preorder relation. Informally, these constructs express the fact that the LTS of B_1 is equivalent (respectively included in, or includes) the LTS of B_2 modulo R.

We may think of various relations R (some of which are already defined in an Annex of the existing LOTOS standard):

- strong equivalence [Par81]
- observational equivalence [Mil80]
- branching bisimulation [vGW89]
- $\tau^*.a$ bisimulation [Mou92]
- safety equivalence [BFG+91]
- etc.

and/or the corresponding preorders.

For example, it may be useful to specify properties of the form:

```
P [A, B] = P [B, A] mod strong_equivalence ;

P [A, A] = A ; stop mod strong_equivalence ;

P [G] (0) = stop mod observation_equivalence ;

PROTOCOL [SEND, RECEIVE] = SERVICE [SEND, RECEIVE] mod branching_equivalence ;

MSC [SEND, RECEIVE] < PROTOCOL [SEND, RECEIVE] mod safety_preorder ;
```

This approach is straightforward, since equivalences and preorders have been extensively studied over the past years. We may wish to restrict some equivalences to the case where the LTS are finite or finitely branching.

5 Temporal logic and μ-calculus for behaviour properties

Alternatively, temporal logic formulas are often convenient to express behaviour properties of systems. A wide range of temporal logics have been proposed in the literature. Given the specificities of LOTOS LTSs and assuming that LOTOS LTSs will be kept unchanged in E-LOTOS, we suggest to use a logical formalism based on the modal μ-calculus [Koz83] and the ACTL temporal logic [NV90].

We propose the following syntax for the "`<properties>`" symbol referenced in Section 2:

`<properties>` ::= B `|=` F ;

where B is a behaviour expression, and where F is a formula of the propositional μ-calculus (from which we can derive the ACTL temporal logic as a collection of shorthand notations). In the following sections, we formally define the syntax and semantics of the formulas.

5.1 The model of an E-LOTOS program

Our formulas are interpreted over the LTS model corresponding to an E-LOTOS description. Formally, an LTS is defined as a 4-tuple $\langle Q, A, T, q_0 \rangle$ where:

- Q is a set of *states* of the E-LOTOS program. A state $q \in Q$ is characterized by the values of the program variables, which are of no interest here;
- A is a set of *actions* performed by the E-LOTOS program. An action $a \in A$ is a tuple $GV_1, ..., V_n$ where G is a *gate* and $V_1, ..., V_n$ ($n \geq 0$) are the E-LOTOS values exchanged (i.e. sent or received) during the rendez-vous on G. For the *silent* action τ, the value list must be empty;
- $T \subseteq Q \times A \times Q$ is the *transition relation*. A transition $\langle q_1, a, q_2 \rangle \in T$ means that the program can pass from the state q_1 to the state q_2 by performing the action a;
- $q_0 \in Q$ is the *initial state* of the program.

In the sequel, we assume the existence of an E-LOTOS program model $M = \langle Q, A, T, q_0 \rangle$ on which the temporal logic formulas are interpreted.

5.2 Syntax and semantics of action formulas

In order to express predicates on the program actions, a small auxiliary logic of actions similar to the one in [NV90] is introduced. The formulas ψ of this logic have the following syntax:

$$\begin{array}{rcl} \psi & ::= & \mathbf{true} \\ & | & \{\ G\ O_1,\ ...,\ O_n\ [E]\ \} \\ & | & \mathbf{not}\ \psi_1 \\ & | & \psi_1\ \mathbf{and}\ \psi_2 \end{array}$$

The construction $\{\ G\ O_1,\ ...,\ O_n\ [E]\ \}$ denotes an *action pattern* where: G is a gate name, E is a boolean expression, and O_i $(1 \leq i \leq n,\ n \geq 0)$ are *offers* of the form:

$$\begin{array}{rcl} O_i & ::= & !E_i \\ & | & ?X_i : S_i \end{array}$$

E_i are E-LOTOS value expressions and X_i are *matching variables* of E-LOTOS sorts S_i. The matching variables X_i occurring in the offers O_1, ..., O_n are visible inside the boolean expression E of the action pattern.

Of course, the usual derived boolean operators are also allowed:

$$\begin{array}{lcl} \mathbf{false} & = & \mathbf{not\ true} \\ \psi_1\ \mathbf{or}\ \psi_2 & = & \mathbf{not}\ (\mathbf{not}\ \psi_1\ \mathbf{and\ not}\ \psi_2) \\ \psi_1\ \mathbf{implies}\ \psi_2 & = & \mathbf{not}\ \psi_1\ \mathbf{or}\ \psi_2 \\ \psi_1\ \mathbf{iff}\ \psi_2 & = & (\psi_1\ \mathbf{implies}\ \psi_2)\ \mathbf{and}\ (\psi_2\ \mathbf{implies}\ \psi_1) \\ \psi_1\ \mathbf{xor}\ \psi_2 & = & \mathbf{not}\ (\psi_1\ \mathbf{iff}\ \psi_2) \end{array}$$

All the binary operators above are left-associative. The **not** operator has the highest precedence, followed in order by **and**, **or** and **xor**, **implies**, **iff**.

Let $\mathcal{A}$ be the set of action formulas. The semantics of an action formula $\psi \in \mathcal{A}$ over the set of actions A of a model M is given by the *interpretation function* $[[.]]_M : \mathcal{A} \rightarrow 2^A$ defined inductively as follows:

$$\begin{array}{lcl} [[\mathbf{true}]]_M & = & A; \\ [[\{\ G\ O_1, ..., O_n\ [E]\ \}]]_M & = & \{a \in A \mid a \text{ has the form } GV_1, ..., V_n; \\ & & \quad \text{for each offer } O_i \text{ of the form } !E_i,\ V_i = E_i; \\ & & \quad \text{for each offer } O_j \text{ of the form } ?X_j : S_j, \text{ the sort of } V_j \text{ is } S_j \\ & & \quad \text{(this also has the side-effect of assigning } V_j \text{ to } X_j\text{)}; \\ & & \quad \text{if present, the boolean expression } E \text{ evaluates to true}\}; \\ [[\mathbf{not}\ \psi]]_M & = & A \setminus [[\psi]]_M; \\ [[\psi_1\ \mathbf{and}\ \psi_2]]_M & = & [[\psi_1]]_M \cap [[\psi_2]]_M\,. \end{array}$$

5.3 Syntax and semantics of μ-calculus formulas

The μ-calculus formulas have the following syntax:

$$
\begin{array}{rcl}
\varphi & ::= & \mathbf{true} \\
& | & X_i \\
& | & \mathbf{not}\ \varphi_1 \\
& | & \varphi_2\ \mathbf{and}\ \varphi_2 \\
& | & <\ \psi\ >\ \varphi_1 \\
& | & \mathbf{lfp}\ X\ .\ \varphi_1\ (X)
\end{array}
$$

X_i $(1 \leq X \leq n)$ are *propositional variables*. The construction $\mathbf{lfp}X.\varphi_1(X)$ denotes the least fix-point of $\varphi_1(X)$. All the occurrences of a propositional variable X in a subformula $\mathbf{lfp}X.\varphi(X)$ are said to be *bound*. All other occurrences are *free*. We write $\varphi(X)$ to explicitly indicate that all the occurrences of X in φ are free.

To ensure the well-definedness of the fix-point formulas, we require that every occurrence of a propositional variable X bounded by an **lfp** operator falls under an even number of negations.

In addition to the usual derived boolean operators **false**, **or**, **implies**, **iff**, **xor** (defined as in section 5.2), we also allow the following derived modal and fix-point operators:

$$
\begin{array}{lcl}
[\ \psi\ ]\ \varphi & = & \mathbf{not} <\ \psi\ >\ \mathbf{not}\ \varphi \\
\mathbf{gfp}\ X\ .\ \varphi\ (X) & = & \mathbf{not}\ \mathbf{lfp}\ X\ .\ \mathbf{not}\ \varphi\ (\mathbf{not}\ X)
\end{array}
$$

The construction $\mathbf{gfp}X.\varphi(X)$ stands for the greatest fix-point of $\varphi(X)$. The unary operators **lfp**, **gfp**, `<.>`, and `[.]` have the same level of precedence as the **not** operator.

A *valuation* $\vec{V} = \langle V_1, ..., V_n \rangle$ assigns the sets of states $V_i \in 2^Q$ to the free propositional variables X_i.

Let $\mathcal{F}$ be the set of μ-calculus formulas. The semantics of a μ-calculus formula $\varphi \in \mathcal{F}$ over a model M is given by the *interpretation function* $[[.]]_M : \mathcal{F} \times (2^Q)^n \rightarrow 2^Q$ defined inductively as follows:

$$
\begin{array}{lcl}
[[\mathbf{true}]]_M(\vec{V}) & = & Q; \\
[[X_i]]_M(\vec{V}) & = & V_i; \\
[[\mathbf{not}\ \varphi]]_M(\vec{V}) & = & Q \setminus [[\varphi]]_M(\vec{V}); \\
[[\varphi_1\ \mathbf{and}\ \varphi_2]]_M(\vec{V}) & = & [[\varphi_1]]_M(\vec{V}) \cap [[\varphi_2]]_M(\vec{V}); \\
[[<\psi>\varphi]]_M(\vec{V}) & = & \{p \in Q \mid \exists a \in [[\psi]]_M, \exists q \in [[\varphi]]_M(\vec{V}) \text{ such that } \langle p, a, q \rangle \in T\}; \\
[[\mathbf{lfp}X.\varphi(X)]]_M(\vec{V}) & = & \cap\{Q' \subseteq Q \mid [[\varphi(Q')]]_M(\vec{V}) \subseteq Q'\}.
\end{array}
$$

A μ-calculus formula φ without free variables is called a *sentence*. A state $q \in Q$ of a model M satisfies a μ-calculus sentence φ if it is contained in the interpretation of φ over M:

$$q \models \varphi \text{ iff } q \in [[\varphi]]_M$$

5.4 ACTL formulas

The practice of specifying temporal logic properties in the μ-calculus has shown that in most cases, the full power of the formalism is not needed. For example, as pointed out in [CPS89], the users of the Concurrency Workbench tool prefer to define in μ-calculus a set of usual temporal logic operators as "macros" and use only these operators to express program properties.

As the models of E-LOTOS programs are LTSs in which the important information is contained in the actions rather than the states, it is appropriate to use a temporal logic interpreted over actions.

A useful temporal logic to reason about programs in terms of actions is ACTL [NV90]. We propose here a simplified variant of ACTL operators to be included in our temporal logic.

The basic ACTL operators are given below. They can be defined in the μ-calculus, but since we don't

provide any mechanism to define new operators, we include them as "built-in" operators.

EX_A (ψ, φ)	=	< ψ > φ
AX_A (ψ, φ)	=	[ψ] φ **and** [**not** ψ] **false**
EU_A (φ_1, ψ, φ_2)	=	**lfp** X . (φ_2 **or** φ_1 **and EX_A** (ψ, X))
AU_A (φ_1, ψ, φ_2)	=	**lfp** X . (φ_2 **or** φ_1 **and** < **true** > **true and AX_A** (ψ, X))

We also allow the usual derived operators:

EU_A_B (φ_1, ψ_1, ψ_2, φ_2)	=	**EU_A** (φ_1, ψ_1, φ_1 **and EX_A** (ψ_2, φ_2))
AU_A_B (φ_1, ψ_1, ψ_2, φ_2)	=	**AU_A** (φ_1, ψ_1, φ_1 **and** < **true** > **true and AX_A** (ψ_2, φ_2))
EF_A (ψ, φ)	=	**EU_A** (**true**, ψ, φ)
EF (φ)	=	**EF_A** (**true**, φ)
AF_A (ψ, φ)	=	**AU_A** (**true**, ψ, φ)
AF (φ)	=	**AF_A** (**true**, φ)
EG_A (ψ, φ)	=	**not AU_A** (**true**, ψ, **not** φ)
EG (φ)	=	**EG_A** (**true**, φ)
AG_A (ψ, φ)	=	**not EU_A** (**true**, ψ, **not** φ)
AG (φ)	=	**AG_A** (**true**, φ)

The ACTL operators are sufficiently expressive to describe useful properties like safety and liveness. For example, a protocol ensuring mutual exclusion between two processes P_1 and P_2 must satisfy the following properties:

- safety property (mutual exclusion) "after the process P_1 has entered the critical section (action `OPEN_1`), the process P_2 cannot do the same (action `OPEN_2`) unless P_1 leaves the critical section (action `CLOSE_1`)", expressed in ACTL as follows:

  [{ `OPEN_1` }] **not EF_A** (**not** { `CLOSE_1` }, < { `OPEN_2` } > **true**)

- liveness property (fairness) "if a process P_i ($i = 1, 2$) wants to enter the critical section, it eventually succeeds", written in ACTL as:

  [{ `OPEN_i` }] **EF** (< { `CLOSE_i` } > **true**)

6 Conclusions

Having a mechanism for requirement capture and property expression is vital in a Formal Description Technique such as E-LOTOS. Competitor languages such as SDL already embody similar features, although in a primitive form.

Taking into account the Committee's decision to keep algebraical equations for the data part, we propose a symmetric approach for the behaviour part. We suggest the use of two complementary formalisms: equivalences and preorder relations, and modal μ-calculus with its derived logic, ACTL.

Our proposal is based upon well known, state-of-the-art concepts and will allow verification tools to be applied directly to E-LOTOS descriptions.

Our proposal should be easily combined into E-LOTOS module language, which is still under design. Among possible extensions, one may wish to consider the possibility of naming μ-calculus formulas by defining "functions for formulas" with names and formal parameters.

References

[BFG+91] Ahmed Bouajjani, Jean-Claude Fernandez, Susanne Graf, Carlos Rodríguez, and Joseph Sifakis. Safety for Branching Time Semantics. In *Proceedings of 18th ICALP*, Berlin, July 1991. Springer Verlag.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Proceedings of the 1st Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, volume 407 of *Lecture Notes in Computer Science*, pages 24–37, Berlin, June 1989. Springer Verlag.

[IT92] ITU-T. Specification and Description Language (SDL). ITU-T Recommendation Z.100, International Telecommunication Union, Genève, 1992.

[Koz83] D. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1980.

[Mou92] Laurent Mounier. *Méthodes de vérification de spécifications comportementales : étude et mise en œuvre*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), January 1992.

[NV90] R. De Nicola and F. W. Vaandrager. *Action versus State based Logics for Transition Systems*. In *Proceedings Ecole de Printemps on Semantics of Concurrency*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer Verlag, 1990.

[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.

[vGW89] R. J. van Glabbeek and W. P. Weijland. Branching-Time and Abstraction in Bisimulation Semantics (extended abstract). CS R8911, Centrum voor Wiskunde en Informatica, Amsterdam, 1989. Also in proc. IFIP 11th World Computer Congress, San Francisco, 1989.