The Syntax and Semantics of FIACRE

Délivrable n° 4.2.4 du projet ANR05RNTL03101 OpenEmbeDD

Bernard Berthomieu*, Jean-Paul Bodeveix+, Mamoun Filali+, Hubert Garavel†, Frédéric Lang†, Florent Peres*, Rodrigo Saad*, Jan Stoecker†, François Vernadat*

Version 1.0 alpha

June 25, 2007

* LAAS-CNRS Université de Toulouse
7, avenue du Colonel Roche, 31077 Toulouse Cedex, France
E-mail: FirstName.LastName@laas.fr

+ IRIT
Université Paul Sabatier
118 Route de Narbonne, 31062 Toulouse Cedex 9, France
E-mail: FirstName.LastName@irit.fr

† INRIA
Centre de Recherche de Grenoble Rhône-Alpes / équipe-projet VASY
655, avenue de l'Europe, 38 334 Saint Ismier Cedex, France
E-mail: FirstName.LastName@inria.fr

1 Introduction

This document presents the syntax and formal semantics of the FIACRE language. FIACRE is an acronym for *Format Intermédiaire pour les Architectures de Composants Répartis Embarqués* (Intermediate Format for the Architectures of Embedded Distributed Components). FIACRE is a formal intermediate model to represent both the behavioural and timing aspects of systems —in particular embedded and distributed systems— for formal verification and simulation purposes. FIACRE embeds the following notions:

- *Processes* describe the behaviour of sequential components. A process is defined by a set of control states, each associated with a piece of program built from deterministic constructs available in classical programming languages (assignments, if-then-else conditionals, while loops, and sequential compositions), nondeterministic constructs (nondeterministic choice and nondeterministic assignments), communication events on ports, and jumps to next state.
- *Components* describe the composition of processes, possibly in a hierarchical manner. A component is defined as a parallel composition of components and/or processes communicating through ports and shared variables. The notion of component also allows to restrict the access mode and visibility of shared variables and ports, to associate timing constraints with communications, and to define priority between communication events.

FIACRE was designed in the framework of projects dealing with model-driven engineering and gathering numerous partners, from both industry and academics. Therefore, FIACRE is designed both as the target language of model transformation engines from various models such as SDL or UML, and as the source language of compilers into the targeted verification toolboxes, namely CADP [8] and TINA [3] in the first step. In this document, we propose a textual syntax for FIACRE, the definition of a metamodel being a different task, out of the scope of this deliverable.

FIACRE was primarily inspired from two works, namely V-COTRE [4] and NTIF [7], as well as decades of research on concurrency theory and real-time systems theory. Its design started after a study of existing models for the representation of concurrent asynchronous (possibly timed) processes [5]. Its timing primitives are borrowed from Time Petri nets [11, 2]. The integration of time constraints and priorities into the language was in part inspired by the BIP framework [1]. Concerning compositions, FIACRE incorporates a parallel composition operator [9] and a notion of gate typing [6] which were previously adopted in E-LOTOS [10] and LOTOS-NT [12, 13].

This document is organized as follows: Section 2 presents the concrete syntax of FIACRE processes, components, and programs. Section 3 presents the static semantics of FIACRE, namely the well-formedness and well-typing constraints. Finally, Section 4 presents the formal dynamic semantics of a FIACRE program, in terms of a timed state/transition graph.

2 Syntax

2.1 Notations

We describe the grammar of the FIACRE language using a variant of EBNF (*Extended Bachus Naur Form*). The EBNF describes a set of production rules of the form "`symb` ::= *expr*", meaning that the nonterminal symbol `symb` represents anything that can be generated by the EBNF expression *expr*. An expression *expr* may be one of the following:

- a keyword, written in bold font (e.g., **type**, **record**, etc.)
- a terminal symbol, written between simple quotes (e.g., ':', '(', etc.)
- a nonterminal symbol, written in teletype font (e.g., `type`, `type_decl`, etc.)
- an optional expression, written "`[` $expr_0$ `]`"
- a choice between two expressions, written "$expr_1$ | $expr_2$"
- the concatenation of two expressions, written "$expr_1$ $expr_2$"
- the iterative concatenation of zero (resp. one) or more expressions, written "$expr^*$" (resp. "$expr^+$")
- the iterative concatenation of zero (resp. one) or more expressions, each two successive occurrences being separated by a given symbol `s`, written "$expr^*_{\texttt{s}}$" (resp. "$expr^+_{\texttt{s}}$")

The star and plus symbols have precedence over concatenation. Parentheses may be used to group a sequence of expressions when iterative concatenation concerns the whole sequence.

2.2 Lexical elements

```
IDENT ::= any sequence of letters, digits, or '_', beginning by a letter

NATURAL ::= any nonempty sequence of digits

INTEGER ::= [ '+' | '-' ] NATURAL

DECIMAL ::= NATURAL [ '.' [ NATURAL ] ] | '.' NATURAL
```

No upper bound is specified for the length of identifiers or numeric constants. The code generation pass will check that numeric constants can indeed be interpreted.

Comments:

Any sequence of characters between the symbol '/*' and the first following occurrence of the symbol '*/' is considered a comment.

Reserved words and characters:

Keywords may not be used as identifiers, these are:

> **and any array bool channel component dequeue do else elsif empty end enqueue enum false first from full if in inf init int interval is nat new none not null of or out par port priority process queue read record select shuffle states sync then to true type var where while write**

The following characters and symbolic words are reserved:

```
[   ]   (   )   {   }   :     ..     .     =   <>   <   >   <=   >=
+   -   *   /   %   $   |   :=   ;   ?     !     ->   /*   */
```

2.3 Types and type declarations

```
type_id ::= IDENT

enum ::= IDENT

field ::= IDENT

size ::= NATURAL

type ::= bool | nat | int
    | interval INTEGER '..' INTEGER
    | enum enum,+ end
    | record (field,+ ':' type),+ end
    | array of size type
    | queue of size type
    | type_id

type_decl ::= type type_id is type
```

2.4 Values and expressions

```
var ::= IDENT

literal ::= NATURAL | true | false | enum | new NATURAL type

initializer ::= literal
    | ('+' | '-') NATURAL
    | '[' initializer,+ ']'
    | '{' (field '=' initializer),+ '}'

access ::= var
    | access '[' exp ']'
    | access '.' field

unop ::= '-' | '$' | not | full | empty | dequeue | first

infixop ::= or
    | and
    | '=' | '<>' |
    | '<' | '>' | '<=' | '>='
    | '+' | '-'
    | '*' | '/' | '%'
```

Infix operators are listed in order of increasing precedence, those in same line have same precedence. All are left associative.

```
binop ::= enqueue

exp ::= literal
    | access
    | unop exp
    | exp infixop exp
    | binop '(' exp ',' exp ')'
    | '(' exp ')'
```

2.5 Ports, channels and channel declarations

```
port := IDENT

channel_id ::= IDENT

profile ::= none | type+*

channel ::= channel_id | profile | channel '|' channel

channel_decl ::= channel channel_id is channel
```

> *A port is a process communication point. Ports can be used to exchange data. A port type determines the type of data that can be exchanged on the port. Channels stand for port types. They have the structure of sets of profiles. Profiles specified by a series of types separated by '*' are associated with ports transfering several values simultaneously. Ports may have several profiles assigned, using channel operator '|'. A port having profile* **none** *may be used as a synchronization port (without any value transfered).*

2.6 Processes

```
state ::= IDENT

tag ::= IDENT

name ::= IDENT

left ::= '[' DECIMAL
    | ']' DECIMAL

right ::= DECIMAL ']'
    | DECIMAL '['
    | inf '['

time_interval ::= left ',' right

port_dec ::= ([in] [out] port)+, ':' channel
```

Ports may have the optional **in** *and/or* **out** *attributes, specifying that values may only be received and/or sent through that port. By default, ports have both the* **in** *and* **out** *attributes. The attributes apply only to the following port. When several ports are specified, all share the same channel but their attributes remain private.*

```
arg_dec ::= ([read] [write] var)+, ':' type
```

Formal parameters have no attributes. Shared variables may have the **read** *and/or the* **write** *attribute, specifying the operations that can be done on the variable. The attributes apply only to the following variable. When several variables are specified, all share the same type but their attributes remain private.*

```
var_dec ::= var+, ':' type [':=' initializer]
```

All variables in the list share the same type and (optional) initial value.

```
process_decl ::=
    process name
       [ '[' port_dec+, ']' ]
       [ '(' arg_dec+, ')' ]
    is  states state+,  init state
       [ var var_dec+, ]
       transition+
```

Name of the process, port parameters, functional parameters or references, states and initial state, local variables, followed by a series of transitions.

```
transition ::= from state statement
```

```
statement ::=
      null
    | access+, ':=' exp+,
    | access+, ':=' any [where exp]
    | communication
    | while exp do statement end
    | if exp then statement (elsif exp then statement)* [else statement] end
    | select statement+[] end
    | to state
    | statement ';' statement
```

Additional well-formedness constraints are given in Section 3.

```
communication ::=
      port [ ':' profile ]
    | port [ ':' profile ] '?' var+, [where exp]
    | port [ ':' profile ] '!' exp+,
```

Synchronization over a port, or reception (?) or emission (!) of one or several values over a port. The optional profile constraint helps specifying the port referred to when it is overloaded.

2.7 Components

```
instance ::= name [ '[' port+, ']' ]  [ '(' exp+, ')' ]
```

Instance of a process or component. Functional parameters are passed by value or by reference, according to the attributes of the corresponding formal parameters of the called component or process.

```
composition ::=
      shuffle composition+ end
    | sync composition+ end
    | par (port*, '->' composition)+ end
    | instance
```

More composition operators to be provided.

```
component_decl ::=
    component name
       [ '[' port_dec+, ']' ]
       [ '(' arg_dec+, ')' ]
    is [ var var_dec+, ]
       [ port (port_dec [ in time_interval ])+, ]
       [ priority (port+| > port+|)+, ]
       composition
```

Name of the component, port parameters, functional parameters or references, local variables or references, local ports with delay constraints, priority constraints, followed by a composition.

2.8 Programs

```
declaration ::=
      type_decl
    | channel_decl
    | process_decl
    | component_decl

program ::=
    declaration+
    name
```

The body of a program is specified as the name of a process or component. If that process or component admits parameters, then these parameters are parameters of the program.

3 Static semantics

3.1 Well-formed programs

3.1.1 Constraints

A program is *well-formed* if its constituents obey the following static semantic constraints.

1. Process and component identifiers should all be distinct;
2. Type and channel identifiers should all be distinct;
3. In any **enum** (resp. **record**) type, all values (resp. field labels) declared must be distinct;
4. In declarations of ports, formal parameters, or local variables, the identifiers declared must be distinct;
5. In a **state** declaration, all states must be distinct, and the initial state must be included;
6. There is a single syntactical class for shared variables, formal parameters of processes or components, local variables, and enum type components. As a consequence, the sets of global variable identifiers, formal parameter identifiers, local variable identifiers and enum component identifiers (declared globally or in the header of some process or component) should be pairwise disjoint;
7. No keyword (e.g. **if**, **from**, etc) may be used as the name of a component, process, variable, type, channel or port;
8. All variables, ports, states, referred to in a process or component must have been declared;
9. All enum components (resp. record labels) referred to in a process or component must appear in some enum (resp. record) type assigned to some declared variable or port of the process, or in the body of some type abbreviation;
10. In any interval type **interval** $x..y$, one must have $x \leq y$;
11. In a process, there may be at most one transition declared from each state declared after the **state** declaration;
12. In a delay declaration in a component, time intervals may not be empty (e.g. intervals like $[7, 3[$ or $]1, 1]$ are rejected);
13. In a **priority** declaration, the priority relation defined must induce a strict partial order;
14. In a deterministic assignment statement, the left hand side must have exactly the same number of components as the right hand side;
15. In an assignment statement, the access expressions in the left-hand side must be pairwise independent; that constraint is explained in Section 3.1.2;

16. In any transition, at most one communication statement may be found along any execution path and, along any path including a communication statement, no shared variable may be read or written. This constraint, referred to as the *single-communication* constraint is integrated with the well-typing condition for statements, see Section 3.2.5;

17. In any process, local variables or their constituents should be initialized before their first use, a sufficient static condition ensuring that property is discussed in Section 3.1.3;

18. In any component, all variables locally declared in the component must be statically initialized;

3.1.2 Well-formedness of assignment statements:

Left hand sides of assignment statements have the shape of series of access expressions. Each access expression is a sequence $a_0\ a_1\ \ldots\ a_n$ where a_0 is some variable and each a_i $(i > 0)$ is either a field access (shape $.f$) or an array component access (shape $[exp]$).

Two access expressions $a_0\ a_1\ ...\ a_n$ and $b_0\ b_1\ ...\ b_m$ are *independent* if:

- either $a_0 \neq b_0$
- or for some i such that $0 \leq i \leq \min(n, m)$:
  - either a_i and b_i have shapes $[x]$ and $[y]$, respectively, where x and y are different integer constants;
  - or a_i and b_i have shapes $.f$ and $.g$, respectively, where f and g are different record labels.

3.1.3 Initialization of variables:

A static condition ensures that the variables locally declared in processes, or any of their constituents, are initialized before any use. The condition is similar to that used for the same purpose in the NTIF intermediate form, the reader is referred to [7] for details.

3.2 Well-typed programs

3.2.1 Type declarations, type expressions, types

Let us first distinguish *type expressions* from *types*: type expressions may contain user-defined type identifiers, while types may not. Type declarations introduce abbreviations (identifiers) for types or type expressions. With each type expression t, one can clearly associate the type τ obtained from it by recursively replacing type identifiers in t by the type expressions they abbreviate.

Similarly, we will make the same distinction between channel expressions (possibly containing channel identifiers) and channels. With each channel expression p, one can associate the channel π obtained from it by replacing channel identifiers by the channels they abbreviate. Channels can be represented by sets of profiles.

All formal parameters of a process (ports or variables), and local variables, have statically assigned type or channel expressions, in the headers of the process, from which one can compute types or channels as above. By *typing context*, we mean in the sequel a map that associates:

- with each port, a set made of its attributes (a non empty subset of $\{\mathbf{in}, \mathbf{out}\}$) and profiles. Unless some attribute is made explicit in its declaration, a port has both **in** and **out** attributes;
- with each global variable, a set made of its attributes (a nonempty subset of $\{\mathbf{read}, \mathbf{write}\}$) and type. There are no default attributes for variables;
- with each formal parameter, its type;
- with each local variable, its type.

Typing contexts are written A in the sequel, or A_b when referring to the context of some particular process b. $A(x)$ denotes the information (attributes and type(s)) assigned to variable (or port) x in A.

3.2.2 Subtyping

Types (not type expressions) are partially ordered by a relation called *subtyping*, written $\leq$ and defined by the following rules:

$$\frac{\tau \in \{\mathbf{bool}, \mathbf{nat}, \mathbf{int}\}}{\tau \leq \tau} \text{ (SU1)} \quad \frac{}{\mathbf{nat} \ \leq \ \mathbf{int}} \text{ (SU2)} \quad \frac{x \geq 0}{\mathbf{interval}\ x..y \leq \mathbf{nat}} \text{ (SU3)}$$

$$\frac{}{\mathbf{interval}\ x..y \leq \mathbf{int}} \text{ (SU4)} \quad \frac{x_1 \geq x_2 \qquad y_1 \leq y_2}{\mathbf{interval}\ x_1..y_1 \leq \ \mathbf{interval}\ x_2..y_2} \text{ (SU5)}$$

$$\frac{\{c_1^1, \ldots, c_n^1\} \subseteq \{c_1^2, \ldots, c_m^2\}}{\mathbf{enum}\ c_1^1, \ldots, c_n^1\ \mathbf{end} \leq \ \mathbf{enum}\ c_1^2, \ldots, c_m^2\ \mathbf{end}} \text{ (SU6)}$$

$$\frac{\tau \leq \tau'}{\mathbf{array\ of}\ k\ \tau \leq \mathbf{array\ of}\ k\ \tau'} \text{ (SU7)} \quad \frac{\tau \leq \tau'}{\mathbf{queue\ of}\ k\ \tau \leq \mathbf{queue\ of}\ k\ \tau'} \text{ (SU8)}$$

$$\frac{\{f_1, \ldots, f_n\} = \{g_1, \ldots, g_n\} \qquad (\forall i, j)(f_i = g_j \Rightarrow \tau_i \leq \tau'_j)}{\mathbf{record}\ f_1 : \tau_1, \ldots, f_n : \tau_n\ \mathbf{end} \leq \mathbf{record}\ g_1 : \tau'_1, \ldots, g_n : \tau'_n\ \mathbf{end}} \text{ (SU9)}$$

> *Note that* **bool** *and* **nat** *are not related by subtyping, nor are records with different sets of fields, or arrays or queues of different sizes.*

Communication ports profiles are tuples of types. The subtyping relation is extended to profiles by:

$$\frac{}{\mathbf{none} \leq \mathbf{none}} \text{ (SU10)} \quad \frac{\tau_1 \leq \tau'_1 \qquad \ldots \qquad \tau_n \leq \tau'_n}{\tau_1 * \cdots * \tau_n \leq \tau'_1 * \cdots * \tau'_n} \text{ (SU11)}$$

3.2.3 Typing expressions

A is some typing context, the following rules define the typing relation (:) for expressions.

Subsumption

$$\frac{A \vdash E : \tau \qquad \tau \leq \tau'}{A \vdash E : \tau'} \text{ (ET1)}$$

Literals, initializers

$$\frac{K \in \texttt{INTEGER} \qquad Val(K) = k}{A \vdash K : \textbf{ interval } k..k} \text{ (ET2)} \qquad \frac{k \in \{\textbf{true}, \textbf{false}\}}{A \vdash k : \textbf{bool}} \text{ (ET3)} \qquad \frac{c \in \texttt{Enums}}{A \vdash c : \textbf{enum } c \textbf{ end}} \text{ (ET4)}$$

$$\frac{A \vdash k_1 : \tau \qquad \ldots \qquad A \vdash k_n : \tau}{A \vdash [k_1, \ldots, k_n] : \textbf{ array of } n\ \tau} \text{ (ET5)} \qquad \frac{Val(N) = n}{A \vdash \textbf{new } N\ \tau : \textbf{ queue of } n\ \tau} \text{ (ET6)}$$

$$\frac{\{f_1, \ldots, f_n\} \subseteq \texttt{Fields} \qquad A \vdash l_1 : \tau_1 \qquad \ldots \qquad A \vdash l_n : \tau_n}{A \vdash \{f_1 : l_1, \ldots, f_n : l_n\} : \textbf{record } f_1 : \tau_1, \ldots, f_n : \tau_n \textbf{ end}} \text{ (ET7)}$$

> `Enums` *is the set of all constants declared in* **enum** *types,* `Fields` *is the set of record field identifiers declared in* **record** *types, By abuse of notation, we write* $K \in$ `INTEGER` *to mean that* K *belongs to the* `INTEGER` *lexical class. Function* Val *associates with a token in the* `INTEGER` *or* `NATURAL` *class the integer it denotes.*

Primitives

$$\frac{A \vdash x : \textbf{bool}}{A \vdash \textbf{not } x : \textbf{bool}} \text{ (ET8)} \qquad \frac{A \vdash x : \textbf{bool} \qquad A \vdash y : \textbf{bool} \qquad (@ \in \{\textbf{and}, \textbf{or}\})}{A \vdash x@y : \textbf{bool}} \text{ (ET9)}$$

$$\frac{A \vdash x : \tau \qquad \tau \leq \textbf{int}}{A \vdash -x : \tau} \text{ (ET10)} \qquad \frac{A \vdash x : \tau \qquad \tau' \leq \tau \leq \textbf{int}}{A \vdash \$\ x : \tau'} \text{ (ET11)}$$

> \$ *is a coercion operator for numeric values. It converts a numeric value of some type* $\tau \leq$ **int** *into a value of some sybtype* τ' *of* τ*. Conversion fails if the target value is out of range.*

$$\frac{A \vdash x : \tau \qquad A \vdash y : \tau \qquad \tau \leq \textbf{int} \qquad (@ \in \{+, -, *, /, \%\})}{A \vdash x@y : \tau} \text{ (ET12)}$$

$$\frac{A \vdash x : \tau \qquad A \vdash y : \tau \qquad \tau \leq \textbf{int} \qquad (@ \in \{<, <=, >, >=\})}{A \vdash x@y : \textbf{bool}} \text{ (ET13)}$$

$$\frac{A \vdash x : \tau \qquad A \vdash y : \tau \qquad (@ \in \{=, <>\})}{A \vdash x@y : \textbf{bool}} \text{ (ET14)}$$

$$\frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{empty } q : \textbf{bool}} \text{ (ET15)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{full } q : \textbf{bool}} \text{ (ET16)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{first } q : \tau} \text{ (ET17)}$$

$$\frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{dequeue } q : \textbf{queue of } k\ \tau} \text{ (ET18)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau \qquad A \vdash E : \tau}{A \vdash \textbf{enqueue } (q, E) : \textbf{queue of } k\ \tau} \text{ (ET19)}$$

Access

$$\frac{A(X) = \{\tau\}}{A \vdash X : \tau}\ (\text{ET20}) \qquad \frac{\{\mathbf{read}, \tau\} \subseteq A(X)}{A \vdash X : \tau}\ (\text{ET21})$$

$$\frac{A \vdash P : \mathbf{array\ of}\ k\ \tau \qquad A \vdash E : \mathbf{interval}\ 0..k\text{–}1}{A \vdash P[E] : \tau}\ (\text{ET22})$$

$$\frac{A \vdash P : \mathbf{record}\ \ \ldots, f : \tau, \ldots\ \mathbf{end}}{A \vdash P.f : \tau}\ (\text{ET23})$$

Global variables in **write***-only mode may not be read.*

3.2.4 Typing patterns

By "pattern", we mean the left hand sides of assignment statements, or the tuples of variables following "?" in input communication statements.

When used as arguments of some primitive, types of values can be promoted to any of their supertypes (by the use of the subsumption rule), but we want variables of all kinds to only store values of their declared type, and not of larger types. For this reason, patterns cannot be typed like expression.

As an illustrative example, assume variable X was declared with type **nat**, and array A with type **array of** 16 **int**. If lhs of assignments were given types by $\vdash$, then the statement $X := A[2]$ would be well typed, storing an integer where a natural is expected, since the rhs has type **int**, and the lhs has type **nat**, and **nat** is a subtype of **int**.

Patterns are given types instead by specific relation ($:_p$), defined by the following five rules. These rules are similar to those for ($:$) for variable and access expressions except that subsumption is restricted:

$$\frac{A(X) = \{\tau\}}{A \vdash X :_p \tau}\ (\text{LT1}) \qquad \frac{\{\mathbf{write}, \tau\} \subseteq A(X)}{A \vdash X :_p \tau}\ (\text{LT2})$$

$$\frac{A \vdash P :_p \mathbf{array\ of}\ k\ \tau \qquad A \vdash E : \mathbf{interval}\ 0..k\text{–}1}{A \vdash P[E] :_p \tau}\ (\text{LT3})$$

$$\frac{A \vdash P :_p \mathbf{record}\ \ \ldots, f : \tau, \ldots\ \mathbf{end}}{A \vdash P.f :_p \tau}\ (\text{LT4})$$

Global variables in **read***-only mode cannot be assigned.*

3.2.5 Typing statements, well typed processes

Well-typing of the statement captured in a transition ensures two properties:

(a) That variables and expressions occurring in the transitions are well-typed and used consistently;

(b) That at most one communication occurs along any possible end-to-end control path in the statement;

(c) That no shared variable is read of written along any end-to-end control path holding a communication;

As for expressions, some information is infered for statements, assuming a typing context; the "typing" relation for statements is written ($:_s$); the "types" derived are subsets α of $\{\mathbf{Shm}, \mathbf{Com}\}$. Presence of **Shm** means that shared storage is manipulated along some control path not performing any communication, presence of **Com** means that some control path performs a (single) communication but does not read nor writes shared variables. Note that a statement type may hold both **Shm** and **Com**.

A process is well-typed if all of its transitions are well-typed in the typing context obtained from its port declarations, formal parameter declarations and local variable declarations. A transition is well typed if the statement it is defined from is well-typed. A statement S is well typed if one can infer $S :_s \alpha$, for some $\alpha \subseteq \{\mathbf{Shm}, \mathbf{Com}\}$, according to the following rules.

The rules make use of an auxiliary predicate $\mathcal{Q}_{\mathcal{A}}(E)$, holding iff some shared variable occurs in expression E in context A, and of a "conditional" notation $b \to \alpha \mid \beta$, standing for α when b holds, or for β otherwise.

Jump, null

$$\frac{}{A \vdash \mathbf{to}\ s :_s \emptyset}\ (\text{ST1}) \qquad \frac{}{A \vdash \mathbf{null} :_s \emptyset}\ (\text{ST2})$$

Sequential composition

$$\frac{A \vdash S_1 :_s \emptyset \qquad A \vdash S_2 :_s \alpha}{A \vdash (S_1; S_2) :_s \alpha}\ (\text{ST3}) \qquad \frac{A \vdash S_1 :_s \alpha \qquad A \vdash S_2 :_s \emptyset}{A \vdash (S_1; S_2) :_s \alpha}\ (\text{ST4})$$

$$\frac{A \vdash S_1 :_s \{\mathbf{Shm}\} \qquad A \vdash S_2 :_s \{\mathbf{Shm}\}}{A \vdash (S_1; S_2) :_s \{\mathbf{Shm}\}}\ (\text{ST5})$$

Assignments

$$\frac{A \vdash P_1 :_p \tau_1 \quad \ldots \quad A \vdash P_n :_p \tau_n \qquad A \vdash E_1 : \tau_1 \quad \ldots \quad A \vdash E_n : \tau_n}{A \vdash P_1, \ldots, P_n := E_1, \ldots, E_n :_s \vee_{i \in 1..n} (\mathcal{Q}_{\mathcal{A}}(P_i) \vee \mathcal{Q}_{\mathcal{A}}(E_i)) \to \{\mathbf{Shm}\} \mid \emptyset}\ (\text{ST6})$$

$$\frac{A \vdash P_1 :_p \tau_1 \quad \ldots \quad A \vdash P_n :_p \tau_n \qquad A \vdash E : \mathbf{bool}}{A \vdash P_1, \ldots, P_n := \mathbf{any\ where}\ E :_s \vee_{i \in 1..n} \mathcal{Q}_{\mathcal{A}}(P_i) \vee \mathcal{Q}_{\mathcal{A}}(E) \to \{\mathbf{Shm}\} \mid \emptyset}\ (\text{ST7})$$

Choices and while loop

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S_1 :_s \alpha \qquad A \vdash S_2 :_s \beta \qquad \mathcal{Q}_{\mathcal{A}}(E) \Rightarrow \alpha \cup \beta \subseteq \{\mathbf{Shm}\}}{A \vdash \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} :_s \alpha \cup \beta}\ (\text{ST8})$$

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S :_s \alpha \qquad \mathbf{Com} \notin \alpha}{A \vdash \mathbf{while}\ E\ \mathbf{do}\ S\ \mathbf{end} :_s \mathcal{Q}_{\mathcal{A}}(E) \to \{\mathbf{Shm}\} \mid \alpha}\ (\text{ST9})$$

$$\frac{A \vdash S_1 :_s \alpha_1 \quad \dots \quad A \vdash S_n :_s \alpha_n}{A \vdash \textbf{select } S_1 \text{ [ ] ... [ ] } S_n \textbf{ end} :_s \alpha_1 \cup \dots \cup \alpha_n} \text{ (ST10)}$$

if e **then** s **end** *is handled like* **if** e **then** s **else null end**. **elsif** *is handled like* **else if**.

Communications

π and π' are profiles (profile **none** or products $\tau_1 * \dots * \tau_n$ of types). Relation $\leq$ is extended to profiles as explained in Section 3.2.2.

$$\frac{\textbf{none} \in A(p) \qquad [\ \textbf{none} \leq \pi\ ]}{A \vdash p\ [\ \text{':'}\ \pi\ ]\ :_s \{\textbf{Com}\}} \text{ (ST11)}$$

$$\frac{\begin{array}{l} A \vdash E_1 : \tau_1 \quad \dots \quad A \vdash E_n : \tau_n \quad \neg(\vee_{i \in 1..n}(\mathcal{Q}_{\mathcal{A}}(E_i))) \\ \{\textbf{out}, \tau_1 * \dots * \tau_n\} \subseteq A(p) \quad [\ \tau_1 * \dots * \tau_n \leq \pi\ ] \end{array}}{A \vdash p\ [\ \text{':'}\ \pi\ ]\ !\ E_1, \dots, E_n :_s \{\textbf{Com}\}} \text{ (ST12)}$$

$$\frac{\begin{array}{l} A \vdash X_1 :_p \tau_1 \quad \dots \quad A \vdash X_n :_p \tau_n \quad A \vdash E : \textbf{bool} \quad \neg(\vee_{i \in 1..n}(\mathcal{Q}_{\mathcal{A}}(X_i)) \vee \mathcal{Q}_{\mathcal{A}}(E)) \\ \{\textbf{in}, \pi'\} \subseteq A(p) \quad \pi' \leq \tau_1 * \dots * \tau_n \quad [\ \pi' \leq \pi\ ] \end{array}}{A \vdash p\ [\ \text{':'}\ \pi\ ]\ ?\ X_1, \dots, X_n \textbf{ where } E :_s \{\textbf{Com}\}} \text{ (ST13)}$$

- In an output communication, the profile of the values sent must be a subtype of some profile assigned to the port.
- In an input communication, some profile assigned to the port must be a subtype of the types of the variables assigned.
- When several profiles are assigned to a port in the port declarations of a process, it must be ensured that each occurrence of that port in the body of the process may be nonambiguously assigned one of these profiles. It is the user's responsability to solve ambiguities, by ascribing profile constraint(s) to some port occurrences. The profile constraint at some occurrence of a port restricts the profiles of that occurrence of the port to those which are subtypes of the constraint.

3.2.6 Well typed components

Components are checked in a context made of:

- A typing context A, defined as for processes except that locally declared variables all have attributes **read** and **write**;
- An interface context I, that associates with all previously declared processes and components an interface of shape $((\dots, \mu_i, \dots), (\dots, \eta_j, \dots))$, in which μ_i is the set of attributes and profiles of the i^{th} port declared for the process or component, and η_j is the set of attributes and type of the j^{th} formal parameter of the component.

The expressions occurring in components are given types and attributes by relation $:_x$, defined by:

$$\frac{A(X) = \eta}{A, I \vdash X :_x \eta} \text{ (CT1)} \quad \frac{A \vdash E : \tau \quad (E \textit{ not a variable})}{A, I \vdash E :_x \{\tau\}} \text{ (CT2)}$$

A component is well-typed if **ok** can be inferred for it by relation $:_c$, defined as follows:

$$\frac{A, I \vdash c_1 :_c \mathbf{ok} \quad \ldots \quad A, I \vdash c_n :_c \mathbf{ok}}{A, I \vdash \mathbf{shuffle}\ c_1, \ldots, c_n\ \mathbf{end} :_c \mathbf{ok}} \text{ (CT3)} \quad \frac{A, I \vdash c_1 :_c \mathbf{ok} \quad \ldots \quad A, I \vdash c_n :_c \mathbf{ok}}{A, I \vdash \mathbf{sync}\ c_1, \ldots, c_n\ \mathbf{end} :_c \mathbf{ok}} \text{ (CT4)}$$

$$\frac{A, I \vdash c_1 :_c \mathbf{ok} \quad \ldots \quad A, I \vdash c_n :_c \mathbf{ok} \quad (\forall i)(Q_i \subseteq \Sigma(c_i))}{A, I \vdash \mathbf{par}\ Q_1 \to c_1, \ldots, Q_n \to c_n\ \mathbf{end} :_c \mathbf{ok}} \text{ (CT5)}$$

For any component c, $\Sigma(c)$ is its sort. Intuitively, the sort of a component is the set of ports it may use, component sorts will be formally defined in Section 4.3.1.

$$\frac{A \vdash e_1 :_x \eta_1 \quad \ldots \quad A \vdash e_n :_x \eta_n \quad ((A(p_1), \ldots, A(p_n)), (\eta_1, \ldots, \eta_m)) \prec I(C)}{A, I \vdash C\ [p_1, \ldots, p_n]\ (e_1, \ldots, e_m) :_c \mathbf{ok}} \text{ (CT6)}$$

Where $((\mu_1^1, \ldots, \mu_{n_1}^1), (\eta_1^1, \ldots, \eta_{m_1}^1)) \prec ((\mu_1^2, \ldots, \mu_{n_2}^2), (\eta_1^2, \ldots, \eta_{m_2}^2))$ holds iff:

- *$n_1 = n_2$ and for each i:*
  $\mu_i^2 \subseteq \mu_i^1 \wedge \mu_i^1 - \mu_i^2 \subseteq \{\mathbf{in}, \mathbf{out}\}$
- *$m_1 = m_2$ and for each j:*
  if $\{\mathbf{read}, \mathbf{write}\} \cap \eta_j^2 \neq \emptyset$ then $\eta_j^2 \subseteq \eta_j^1$ else $\tau_j^1 \leq \tau_j^2$ where $\eta_j^1 = \{\tau_j^1\}$ and $\eta_j^2 = \{\tau_j^2\}$

3.2.7 Well typed programs

A program is well typed if the declarations and component instance it contains are well typed.

4 Timed operational semantics

All programs in this section are assumed well-formed and well-typed.

4.1 Semantics of expressions

4.1.1 Semantic domains

The semantics of expressions is given in denotational style, it associates with every well-typed expression a value in some mathematical domain **D** built as follows.

Let $\mathbb{Z}$ and $\mathbb{N}$ be the set of integers and non-negative integers, respectively, equipped with their usual arithmetic and comparison functions;

$\mathbf{B} = \{true, false\}$ be a domain of truth values, equipped with functions *not*, *and* and *or*;

S be the set of strings containing letters, digits, and symbol '_';

$Arrays(E)$ be the set of mappings from finite subsets of $\mathbb{N}$ to E;

$Records(E)$ be set of mappings from finite subsets of **S** to E;

Then $\mathbf{D} = D_\omega$, where:

$D_0 = \mathbb{Z} \cup \mathbf{B} \cup \mathbf{S}$

$D_{n+1} = D_n \cup Arrays(D_n) \cup Records(D_n)$

FIACRE arithmetic expressions are given meanings in set $\mathbb{Z}$, boolean expressions in set **B**, enum expressions in **S**, arrays in some set $Arrays(D_n)$, and records in some set $Records(D_n)$, for some finite n. Queues denote some elements of $Arrays(D_n)$. The following mappings are defined for queue denotations ($\mathcal{D}(m)$ is the domain of mapping m):

- *empty* q is equal to *true* if $\mathcal{D}(q) = \emptyset$, or $false$ otherwise;
- $full\ k\ q$ ($n \in \mathbb{N}$) is equal to *true* if $k - 1 \in \mathcal{D}(q)$, or $false$ otherwise;
- $first\ q = q(0)$, assuming $0 \in \mathcal{D}(q)$;
- *dequeue* q, assuming $0 \in \mathcal{D}(q)$, is the mapping q' such that $q'(x-1) = q(x)$ for all $x \in \mathcal{D}(q)$;
- *enqueue* $q\ e$ is the mapping q' such that $q'(x) = q(x)$ for $x \in \mathcal{D}(q)$, and $q'(a) = e$, where a is the smallest non negative integer not in $\mathcal{D}(q)$.

4.1.2 Stores

Expression are given meanings relative to a store. The store associates values in **D** with (some) variables. Stores are written e, e', etc, $e(x)$ is the value associated with variable x in store e, $\mathcal{D}(e)$ is the domain of e. Entries are dynamically added to the store when non-initialized variables, or some of their components, are first assigned. The static condition in Section 3.1.3 ensures that variables or variable components are initialized before their first access.

4.1.3 Semantic rules for expressions

Evaluation rules all have the following shape. The rule means that, under conditions P_1 to P_n, the semantics of expression E with store e is v. The store e may be omitted if the result does not depend on its contents.

$$\frac{P_1 \quad \ldots \quad P_n}{e \vdash E \rightsquigarrow v}$$

Literals, initializers

- Numeric constants denote integers in $\mathbb{Z}$. Implementations may choose to reject literals that are not machine representable;
- Enum values denote strings in **S**;
- The booleans **true** and **false** denote values $true$ and $false$ in **B**, respectively;
- **new** $n\ \tau$ (any empty queue) denotes $\emptyset$;
- Representing mappings by their graphs, constant records and arrays are given meanings by:

$$\frac{\vdash l_1 \rightsquigarrow v_1 \quad \ldots \quad \vdash l_n \rightsquigarrow v_n}{\vdash [l_1, \ldots, l_n] \rightsquigarrow \{(0, v_1), \ldots, (n-1, v_n)\}} \text{ (ES1)}$$

$$\frac{\vdash l_1 \rightsquigarrow v_1 \quad \ldots \quad \vdash l_n \rightsquigarrow v_n}{\vdash \{f_1 : l_1, \ldots, f_n : l_n\} \rightsquigarrow \{(f_1, v_1), \ldots, (f_n, v_n)\}} \text{ (ES2)}$$

Access expressions

$$\frac{X \in \mathcal{D}(e)}{e \vdash X \rightsquigarrow e(X)} \text{ (ES3)} \quad \frac{e \vdash P \rightsquigarrow a \quad e \vdash E \rightsquigarrow i \quad i \in \mathcal{D}(a)}{e \vdash P[E] \rightsquigarrow a(i)} \text{ (ES4)} \quad \frac{e \vdash P \rightsquigarrow r \quad f \in \mathcal{D}(r)}{e \vdash P.f \rightsquigarrow r(f)} \text{ (ES5)}$$

> *Well-typing ensures that array indices cannot be out of range, nor fields undefined in the records they are sought for. Non initialized or partially initialized variables are not in the stores, hence the condition on domains. Satisfaction of these conditions is guaranteed by the static semantic constraints explained in Section 3.1.3.*

Primitives

Well-typing implies that all primitives in an expression can be assigned at least one type. When several types can be assigned to some primitive, the typechecker is assumed to have computed a suitable one for it, typically the type that puts the weakest constraints on the arguments of the primitive.

The primitives whose semantics is type-dependent appear in the semantic rules with some annotations added (by the typechecker): arithmetic primitives are annotated with the type of their arguments, and some operators for queues are annotated with the length of the queue they are applied to.

Some primitives are partially defined (e.g. arithmetic functions over intervals, or taking an element from a queue). This appears in the rules by some extra hypothesis (side-conditions). The rules do not make precise any exception handling mechanism, it is assumed that implementations are able to detect when a rule is not applicable and take an adequate decision in that case.

- Arithmetic primitives at type τ (τ is some subtype of **int**):

$$\frac{e \vdash x \rightsquigarrow a \qquad In(-a, \tau)}{e \vdash -_\tau\ x \rightsquigarrow -a}\ \text{(ES6)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b \qquad In(a\ @\ b, \tau) \qquad @ \in \{+, -, *\}}{e \vdash x\ @_\tau\ y \rightsquigarrow a\ @\ b}\ \text{(ES7)}$$

$$\frac{e \vdash x \rightsquigarrow a \qquad In(a, \tau)}{e \vdash \$_\tau\ x \rightsquigarrow a}\ \text{(ES8)}$$

$$\frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b \qquad b \neq 0 \qquad In(a\ @\ b, \tau) \qquad @ \in \{/, \%\}}{e \vdash x\ @_\tau\ y \rightsquigarrow a\ @\ b}\ \text{(ES9)}$$

Operations over **nat** *or* **interval** *types behave like those over* **int** *type except that they are undefined if the result is not in the expected set. Predicate In is defined as follows: $In(v, \mathbf{int})$ always holds, $In(v, \mathbf{nat})$ holds if $v \geq 0$, and $In(v, \mathbf{interval}\ a..b)$ if $a \leq v \leq b$. Implementations may strengthen predicate In by conditions asserting that the results are machine representable.*

- Boolean primitives:

$$\frac{e \vdash x \rightsquigarrow a}{e \vdash\ \mathbf{not}\ x \rightsquigarrow\ not\ a}\ \text{(ES10)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b}{e \vdash x\ \mathbf{and}\ y \rightsquigarrow a\ and\ b}\ \text{(ES11)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b}{e \vdash x\ \mathbf{or}\ y \rightsquigarrow a\ or\ b}\ \text{(ES12)}$$

- Comparison and equality ($@ \in \{<, >, <=, >=, =, <>\}$):

$$\frac{e \vdash x \rightsquigarrow a \qquad e \vdash x \rightsquigarrow b \qquad a\ @\ b}{e \vdash x\ @\ y \rightsquigarrow true}\ \text{(ES13)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash x \rightsquigarrow b \qquad \neg(a\ @\ b)}{e \vdash x\ @\ y \rightsquigarrow false}\ \text{(ES14)}$$

- Primitives for queues:

$$\frac{e \vdash q \rightsquigarrow Q}{e \vdash \mathbf{empty}\ q \rightsquigarrow empty\ Q}\ \text{(ES15)} \qquad \frac{e \vdash q \rightsquigarrow Q}{e \vdash \mathbf{full_N}\ q \rightsquigarrow full\ N\ Q}\ \text{(ES16)}$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \mathbf{first}\ q \rightsquigarrow first\ Q}\ \text{(ES17)}$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \mathbf{dequeue}\ q \rightsquigarrow dequeue\ Q}\ \text{(ES18)} \qquad \frac{e \vdash q \rightsquigarrow Q \qquad e \vdash x \rightsquigarrow v \qquad N - 1 \notin \mathcal{D}(Q)}{e \vdash \mathbf{enqueue}_N\ (q, x) \rightsquigarrow enqueue\ Q\ v}\ \text{(ES19)}$$

Note that **first** *and* **dequeue** *are undefined on empty queues and that* **enqueue$_\mathbf{N}$** *is undefined on queues already holding N elements.*

4.1.4 Accesses in left-hand sides of assignments (L-values)

Left-hand sides of assignments evaluate to pairs z, g, in which z is a value and g maps values to stores. Intuitively, $g(v)$, where v is the value put into the location referred to by the lhs, is the updated store; z at some level is a partial value used to compute function g at the level above.

The evaluation relation for lhs of assignments is denoted $\rightsquigarrow^l$ and defined by the following rules, in which:

- $(\lambda v.\ f(v))$ is the mapping that, applied to value v, returns the value mapped by f to v;
- $extend\ f\ x = f\ x$ if $x \in \mathcal{D}(f)$, or $\emptyset$ otherwise;
- $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n] \oplus e$ is the function f such that $f(x_i) = v_i$ for any $i \in 1..n$, and $f(z) = e(z)$ for any $z \in \mathcal{D}(e) - \{x_1, \ldots, x_n\}$.

$$\frac{}{e \vdash X \rightsquigarrow^l (extend\ e\ X), (\lambda v.\ [X \mapsto v] \oplus e)} \text{ (LS1)}$$

$$\frac{e \vdash P \rightsquigarrow^l e', a \qquad e \vdash E \rightsquigarrow i}{e \vdash P[E] \rightsquigarrow^l (extend\ e'\ i), (\lambda v.\ a([i \mapsto v] \oplus e'))} \text{ (LS2)}$$

$$\frac{e \vdash P \rightsquigarrow^l e', r}{e \vdash P.f \rightsquigarrow^l (extend\ e'\ f), (\lambda v.\ r([f \mapsto v] \oplus e'))} \text{ (LS3)}$$

4.2 Semantics of Processes

4.2.1 Notations

The semantics of transitions is expressed by a labelled relation, called the *micro-step* relation. Micro steps have shape $(S, e) \stackrel{l}{\Rightarrow} (S', e')$ in which S, S' are statements and e, e' are stores. Action l is either a communication action or the silent action ϵ. Communication actions are sequences $p_\tau\ v_1 \ldots v_n$, in which p_τ is a port, identified by a label and a profile, and the v_i are values.

4.2.2 Rules, micro-steps

The micro-step relation is defined inductively from the structure of statements by a set of inference rules. Each micro-step rule $(S, e) \stackrel{l}{\Rightarrow} (S', e')$ obeys the invariant $S' \in \{\texttt{done}\} \cup \{\texttt{target}\ s | s \in \Lambda\}$, where Λ is the declared set of states of the process.

From, to, null

$$\frac{(S, e) \stackrel{l}{\Rightarrow} (S', e')}{(\textbf{from}\ s\ S, e) \stackrel{l}{\Rightarrow} (S', e')} \text{ (SS1)} \qquad \frac{}{(\textbf{null}, e) \stackrel{\epsilon}{\Rightarrow} (\texttt{done}, e)} \text{ (SS2)} \qquad \frac{}{(\textbf{to}\ s, e) \stackrel{\epsilon}{\Rightarrow} (\texttt{target}\ s, e)} \text{ (SS3)}$$

Deterministic assignment

$$\frac{\begin{array}{llll} e \vdash E_1 \rightsquigarrow v_1 & e \vdash E_2 \rightsquigarrow v_2 & \ldots & e \vdash E_n \rightsquigarrow v_n \\ e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \ldots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ e' = a_n(v_n) & & & \end{array}}{(P_1, P_2, \ldots, P_n := E_1, E_2, \ldots, E_n, e) \stackrel{\epsilon}{\Rightarrow} (\texttt{done}, e')} \text{ (SS4)}$$

The independence property for accesses in multiple assignments, enforced by the static semantic constraint in Section 3.1.2, ensures that the resulting store is invariant by any permutation of accesses $P_1, \ldots, P_n$ and the corresponding expressions $E_1, \ldots, E_n$.

Nondeterministic assignment

$$\frac{\begin{array}{lll} e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \ldots \quad a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ e' \vdash E \rightsquigarrow \mathit{true} & e' = a_n(v_n) & \end{array}}{(P_1, P_2, \ldots, P_n := \textbf{any where } E, e) \stackrel{\epsilon}{\Rightarrow} (\texttt{done}, e')} \text{ (SS5)}$$

v_i ranges over all values of the type of P_i.

While loops

$$\frac{e \vdash E \rightsquigarrow \mathit{true} \qquad (S; \textbf{while } E \textbf{ do } S \textbf{ end}, e) \stackrel{l}{\Rightarrow} (S', e')}{(\textbf{while } E \textbf{ do } S \textbf{ end}, e) \stackrel{l}{\Rightarrow} (S', e')} \text{ (SS6)}$$

$$\frac{e \vdash E \rightsquigarrow \mathit{false}}{(\textbf{while } E \textbf{ do } S \textbf{ end}, e) \stackrel{\epsilon}{\Rightarrow} (\texttt{done}, e)} \text{ (SS7)}$$

It is assumed that condition E eventually evaluates to false.

Deterministic choice

$$\frac{e \vdash E \rightsquigarrow \mathit{true} \qquad (S_1, e) \stackrel{l}{\Rightarrow} (S, e')}{(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}, e) \stackrel{l}{\Rightarrow} (S, e')} \text{ (SS8)} \qquad \frac{e, E \rightsquigarrow \mathit{false} \qquad (S_2, e) \stackrel{l}{\Rightarrow} (S, e')}{(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}, e) \stackrel{l}{\Rightarrow} (S, e')} \text{ (SS9)}$$

if *e* **then** *s* **end** *is handled like* **if** *e* **then** *s* **else null end**. **elsif** *is handled like* **else if**.

Nondeterministic choice

$$\frac{(\exists i \in 1..n)((S_i, e) \stackrel{l}{\Rightarrow} (S_i', e'))}{(\textbf{select } S_1 \text{ [ ] ... [ ] } S_n \textbf{ end}, e) \stackrel{l}{\Rightarrow} (S_i', e')} \text{ (SS10)}$$

Sequential composition

$$\frac{(S_1, e) \stackrel{l}{\Rightarrow} (\texttt{target } s, e')}{(S_1; S_2, e) \stackrel{l}{\Rightarrow} (\texttt{target } s, e')} \text{ (SS11)} \qquad \frac{(S_1, e) \stackrel{l_1}{\Rightarrow} (\texttt{done}, e') \qquad (S_2, e') \stackrel{l_2}{\Rightarrow} (S', e'')}{(S_1; S_2, e) \stackrel{l_1.l_2}{\Longrightarrow} (S', e'')} \text{ (SS12)}$$

with "." such that $\epsilon.\epsilon = \epsilon$ and $\epsilon.l = l.\epsilon = l$, for any l.
The well-formedness condition implies $l_1 = \epsilon \vee l_2 = \epsilon$.
Note that the statements following a **to** *statement are dead code.*

Communication

$$\frac{}{(p_\tau, e) \stackrel{p_\tau}{\Longrightarrow} (\texttt{done}, e)} \text{ (SS13)}$$

$$\frac{e \vdash E_1 \rightsquigarrow v_1 \quad \ldots \quad e \vdash E_n \rightsquigarrow v_n}{(p_\tau!E_1, \ldots, E_n, e) \stackrel{p_\tau\ v1,\ldots,vn}{\Longrightarrow} (\texttt{done}, e)} \text{ (SS14)}$$

$$\frac{e' = e[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n] \quad [e' \vdash E \rightsquigarrow \mathit{true}]}{(p_\tau?X_1, .., X_n\ [\textbf{where}\ E], e) \stackrel{p_\tau\ v_1,\ldots,v_n}{\Longrightarrow} (\texttt{done}, e')} \text{ (SS15)}$$

Types decorate ports to distinguish the different variants of overloaded ports.
v_i ranges over all values of the type of X_i.

4.2.3 Macro-steps

By the previous rules, FIACRE transitions expand into series of micro-steps originating at the process state specified by the **from** statement, and ending at some target process state specified in a **to** statement. Each of these sequences identifies a possible discrete *macro-step*, or *process action*, as follows. The semantics of a process is the union of the action relations of all its transitions.

$$\frac{(\textbf{from}\ s\ S, e) \stackrel{l}{\Rightarrow} (\texttt{target}\ s', e')}{(s, e) \stackrel{l}{\rightarrow} (s', e')}$$

A *process configuration* is a pair constituted of a process state and a store capturing the values of its initialized local variables. Note that shared variables are not considered part of process configurations, intuitively, configurations characterize the local state information for the process.

The *initial configuration* of a process is the pair (s_0, e_0) in which s_0 is the declared initial state of the process and store e_0 captures the values of the formal parameters of the process and those of its statically initialized local variables.

4.3 Semantics of programs and components

A program is a series of declarations followed by component identifier or a closed component instance. The semantics of a program is the semantics of that component or component instance.

4.3.1 Component states, terminology

Labels, ports: Let us recall a communication action is either the silent action ϵ or a sequence $p_\pi\ v_1\ \ldots\ v_n$, in which p_τ is a *port* with *label* p and *profile* π, and the v_i are values. If $l \neq \epsilon$, then $\mathcal{L}(l)$ denotes the port used in action l.

Component states: Component states are abstract terms built from the following grammar:

c	::=	$c_1 \mid c_2$	product
	|	**hide** H I c	hiding, H is a set of ports and I maps them to time intervals
	|	**relab** R c	relabelling, R is a surjective mapping over ports
	|	**prio** Π I c	priority assignement, Π is a priority relation between ports
	|	**proc** P (s, e)	process call, P is a process identifier, (s, e) a process configuration
	|	**comp** C c	component call, C is a component identifier, c a component state

For simplifying the semantics rules, all composition operators will be expressed in terms of relabelling and a binary product written "|". "|" denotes the usual product for timed transition systems. Equivalence of compositions will be briefly addressed in Section 4.4.

Process and component sorts: The *sort* of a named process P or named component C, written $\Sigma(P)$ or $\Sigma(C)$, is the set of visible ports declared for that process or component. If some port p has several profiles $\pi_1, \ldots, \pi_n$ declared, then the overloaded port appears in the sort as several ports $p_{\pi_1}, \ldots, p_{\pi_n}$ with the same label but different profiles. The notion of sort is extended to component states as follows:

$$
\begin{array}{lcl}
\Sigma(c_1 \mid c_2) & = & \Sigma(c_1) \cup \Sigma(c_2) \\
\Sigma(\textbf{hide } H\ I\ c) & = & \Sigma(c) - H \\
\Sigma(\textbf{relab } R\ c) & = & \{R(p_\tau) \mid p_\tau \in \Sigma(c) \cap \mathcal{D}(R)\} \cup (\Sigma(c) - \mathcal{D}(R)) \\
\Sigma(\textbf{prio } \Pi\ c) & = & \Sigma(c) \\
\Sigma(\textbf{proc } P\ (s_0, e_0)) & = & \Sigma(P) \\
\Sigma(\textbf{comp } C\ c)) & = & \Sigma(C)
\end{array}
$$

Note that the sort of a named component C, defined from its declared list of ports, may differ from the sort one could compute from its body (for instance because some port declared in the component parameters does not occur in its body nor in that of all components invoked from it). Similarly, process headers may introduce ports never used in the process body.

Initial state of a component: The initial state of a component is the result of its rewriting by relation $\rightsquigarrow_0$, defined in the sequel. Intuitively, $\rightsquigarrow_0$ "inlines" component definitions, keeping track of component boundaries for preserving the sorts of subcomponents, and replaces process definitions by their initial configurations. Relation $\rightsquigarrow_0$ is defined by the following rules;

(**shuffle** $c_1\ \ldots\ c_n$ **end**) is handled like (**par** $\rightarrow c_1\ \ldots\ \rightarrow c_n$ **end**)

$$\frac{e \vdash c_1 \rightsquigarrow_0 c'_1 \qquad \ldots \qquad e \vdash c_n \rightsquigarrow_0 c'_n}{e \vdash \textbf{sync } c_1\ \ldots\ c_n \textbf{ end} \rightsquigarrow_0 c'_1 \mid \ldots \mid c'_n} \text{ (CE1)}$$

$$\frac{e \vdash c_1 \rightsquigarrow_0 c'_1 \qquad \ldots \qquad e \vdash c_n \rightsquigarrow_0 c'_n}{\textbf{par } s_1 \rightarrow c_1\ \ldots\ s_n \rightarrow c_n \textbf{ end} \rightsquigarrow_0 \textbf{relab } R\ (\textbf{relab } R_1\ c'_1 \mid \ \ldots\ \mid \textbf{ relab } R_n\ c'_n)} \text{ (CE2)}$$

where $A = \bigcup_{i \in \{1,\ldots,n\}} \Sigma(c_i)$

B *is some set of ports disjoint from* A

Ψ *is some bijection from* A *to* B

R_i *is* Ψ *restricted to the domain* $\Sigma(c_i) - s_i$

$$R = \Psi^{-1}$$

$$\frac{(\forall i \in F)(e \vdash e_i \rightsquigarrow v_i) \qquad e \oplus \{x_i \mapsto v_i \mid i \in F\} \oplus a \vdash c \rightsquigarrow_0 c'}{e \vdash C\ [q_i]_{i \in E}\ (e_i)_{i \in F} \rightsquigarrow_0 \textbf{hide}\ H\ I\ (\textbf{prio}\ \Pi\ (\textbf{relab}\ \{p_i \mapsto q_i \mid i \in E\}\ (\textbf{comp}\ C\ c')))} \quad \text{(CE3)}$$

C is the name of a component expecting as parameters ports p_i and variables x_i, and of body c. Store a captures the variables locally declared and statically initialized in C, H is the set of ports locally declared in C, I maps time intervals to them (default $[0, \infty[$). Without loss of generality, it is assumed that the non-shared x_i and the variables locally declared in C have different names than those of all variables in store e, and that the names of the shared x_i match the names of the variables in e they are bound to. Π is the priority relation defined in C, that is the transitive closure of the port pairs occurring in the **priority** *declarations, if any; $(p_\tau, q_{\tau'}) \in \Pi$ means that communication actions over port p_τ have higher priority than those over port $q_{\tau'}$.*

$$\frac{(\forall i \in F)(e \vdash e_i \rightsquigarrow v_i)}{e \vdash P\ [q_i]_{i \in E}\ (e_i)_{i \in F} \rightsquigarrow_0 \textbf{relab}\ \{p_i \mapsto q_i \mid i \in E\}\ (\textbf{proc}\ P\ (s_0, \{x_i \mapsto v_i \mid i \in F^p\} \oplus a))} \quad \text{(CE4)}$$

P is the name of a process expecting as parameters ports p_i and variables x_i, with initial state s_0. F^p is the subset of F corresponding to the parameters passed to P "by value" (excluding the shared variable parameters); store a captures the variables locally declared and statically initialized in P. (s_0, e_0), where $e_0 = \{x_i \mapsto v_i \mid i \in F^p\} \oplus a$ is the initial configuration of the process. As for component instances, it is assumed without loss of generality that the non-shared x_i and the variables locally declared in P have different names than those of the variables in e, and that the names of the shared x_i match the names of the variables in e they are bound to.

4.3.2 Component interactions

Interactions are triples (c, e, i) in which c is a component state, e is a store, and i an interaction label. Intuitively, the set of communication actions of a component is partitionned into interactions; two communication actions "matching" the same interaction will share their timing information.

Interaction labels are abstract terms built from the following grammar:

k	::=	$\bullet$	the *silent* interaction
	|	p_τ	port interactions
	|	$\textbf{inl}(k) \mid \textbf{inr}(k)$	injection interactions
	|	(k_1, k_2)	product interactions

The interaction label $\langle l \rangle$ associated with a process action l is defined by: $\langle \epsilon \rangle = \bullet, \langle p_\tau\ v_\cdot\ \ldots\ v_n \rangle = p_\tau$. Computation of component interaction labels will be explained with the semantic rules.

$L(c, k)$ is the communication label of the actions of component c matching interaction label k, if any. If no action matching k is visible, then $L(c, k) = \emptyset$. $L(c, k)$ is computed as follows:

$$
\begin{array}{lcl}
L(c, \bullet) & = & \emptyset \\
L(c_1 \mid c_2, \mathbf{inl}(k)) & = & L(c_1, k) \\
L(c_1 \mid c_2, \mathbf{inr}(k)) & = & L(c_2, k) \\
L(c_1 \mid c_2, (k_1, k_2)) & = & L(c_1, k_1) \\
L(\mathbf{hide}\ H\ I\ c, k) & = & \mathbf{if}\ L(c,k) \in E\ \mathbf{then}\ \emptyset\ \mathbf{else}\ L(c,k) \\
L(\mathbf{relab}\ R\ c, k) & = & \mathbf{if}\ L(c,k) \in \mathcal{D}(R)\ \mathbf{then}\ R(L(c,k))\ \mathbf{else}\ L(c,k) \\
L(\mathbf{prio}\ \Pi\ c, k) & = & L(c,k) \\
L(\mathbf{comp}\ C\ c, k)) & = & L(c,k) \\
L(\mathbf{proc}\ P\ (s_0, e_0), k) & = & k
\end{array}
$$

4.3.3 Component configurations

Component configurations are triples (c, e, ϕ) in which c is a component state, as defined in Section 4.3.1, e is a store, and ϕ is an *interval function*.

Function ϕ maps the enabled interactions at the source configuration to the time intervals in which actions "matching" those interactions may occur. Since the source state and store are implicit, ϕ simply maps interaction labels to time intervals; the set of interactions enabled at some configuration (c, e, ϕ) is the set of triples $\{(c, e, i) \mid i \in \mathcal{D}(\phi)\}$.

The initial configuration of a component is the initial component state c_0 resulting from its evaluation by $\rightsquigarrow_0$, associated with the store constituted of all variables locally declared in the component and all component instances invoked in it (all variables declared are assumed to have have different names, according to our convention Section 4.3.1), and the initial interval function ϕ_0.

ϕ_0 maps time intervals to the labels of the interactions corresponding to the actions enabled at the initial configuration of the component. It is defined as the result of applying function Φ defined below to the initial component state and initial store. Function Φ is defined as follows:

$$
\begin{array}{lcl}
\Phi(\mathbf{proc}\ P\ (s_0, e_0), e) & = & \{(\langle l \rangle, [0, \infty[) \mid (\exists s, e')((s_0, e \oplus e_0) \xrightarrow{l} (s, e'))\} \\
\Phi(c_1 \mid c_2, e) & = & \Phi(c_1, e) \mid \Phi(c_2, e) \\
\Phi(\mathbf{hide}\ H\ I\ c, e) & = & \{(k, r) \in \phi \mid L(c,k) \notin H\} \cup \{(k, I(L(c,k)) \in \phi \mid L(c,k) \in H\} \\
 & & \text{where } \phi = \Phi(c, e) \\
\Phi(\mathbf{relab}\ R\ c, e) & = & \Phi(c, e) \\
\Phi(\mathbf{prio}\ \Pi\ c, e) & = & \Phi(c, e) \\
\Phi(\mathbf{comp}\ C\ c, e) & = & \Phi(c, e)
\end{array}
$$

where

$$
\begin{array}{lcl}
\phi_1 | \phi_2 & = & \{(\mathbf{inl}(k), r) \mid (k, r) \in \phi_1 \wedge L(c_1, k) \in \Sigma(c_1) - \Sigma(c_2)\} \\
 & \cup & \{(\mathbf{inr}(k), r) \mid (k, r) \in \phi_2 \wedge L(c_2, k) \in \Sigma(c_2) - \Sigma(c_1)\} \\
 & \cup & \{((k_1, k_2), r) \mid (k_1, r) \in \phi_1 \wedge (k_2, r) \in \phi_2 \wedge \\
 & & \qquad L(c_1, k_1) = L(c_2, k_2) \wedge L(c_1, k_1) \in \Sigma(c_1) \cap \Sigma(c_2)\}
\end{array}
$$

4.3.4 Semantic rules for components

The semantics of a component is a *Timed Transition System*. These are Labelled Transition Systems extended with state properties and time-elapsing transitions. The labelled semantics relation,

linking component configurations, has two sorts of transitions: discrete (communication actions) and continuous (real delays).

For making easier the definition of the transition relation, discrete transitions will be concretely labelled both with a communication action (above the arrow) and the corresponding interaction label (below the arrow), as shown in the sequel. Note that, in this formula, we always have $l = \epsilon$ and $L(c,k) = \bullet$, or otherwise $\mathcal{L}(l) = L(c,k)$.

$$(c, e, \phi) \xrightarrow[k]{l} (c', e', \phi')$$

Continuous transitions

$$\frac{(\forall k)(\phi(k) \neq \bot \Rightarrow \theta \leq \uparrow\phi(k))}{(c, e, \phi) \xrightarrow{\theta} (c, e, \phi \stackrel{\bullet}{-} \theta)} \text{ (PS1)}$$

Time may elapse as long as no time-constrained interaction passes its deadline. For any interval r, $\uparrow r$ is its right endpoint, or ∞ if r is not right-bounded. For any k: $(\phi \stackrel{\bullet}{-} \theta)(k) = \{x - \theta \mid (x - \theta) \geq 0 \wedge x \in \phi(k)\}$

Product

$$\frac{(c_1, e, \phi_1) \xrightarrow[k]{l} (c_1', e', \phi_1') \quad l = \epsilon \vee \mathcal{L}(l) \notin \Sigma(c_2)}{(c_1|c_2, e, \phi_1|\phi_2) \xrightarrow[\mathbf{inl}(k)]{l} (c_1'|c_2, e', \phi_1'|\phi_2)} \text{ (PS2)}$$

$$\frac{(c_2, e, \phi_2) \xrightarrow[k]{l} (c_2', e', \phi_2') \quad l = \epsilon \vee \mathcal{L}(l) \notin \Sigma(c_1)}{(c_1|c_2, e, \phi_1|\phi_2) \xrightarrow[\mathbf{inr}(k)]{l} (c_1|c_2', e', \phi_1|\phi_2')} \text{ (PS3)}$$

$$\frac{(c_1, e, \phi_1) \xrightarrow[k_1]{l} (c_1', e, \phi_1') \qquad (c_2, e, \phi_2) \xrightarrow[k_2]{l} (c_2', e, \phi_2') \qquad l \neq \epsilon}{(c_1|c_2, e, \phi_1|\phi_2) \xrightarrow[(k_1,k_2)]{l} (c_1'|c_2', e, \phi_1'|\phi_2')} \text{ (PS4)}$$

Hiding

$$\frac{(c, e, \phi \ominus H) \xrightarrow[k]{l} (c', e', \phi') \qquad l = \epsilon \vee \mathcal{L}(l) \notin H}{(\mathbf{hide}\ H\ I\ c, e, \phi) \xrightarrow[k]{l} (\mathbf{hide}\ H\ I\ c', e', \phi' \setminus_I \phi)} \text{ (PS5)}$$

$$\frac{(c, e, \phi \ominus H) \xrightarrow[k]{p_\tau\ v_1\ \ldots\ v_n} (c', e', \phi') \qquad p_\tau \in H \qquad 0 \in \phi(k)}{(\mathbf{hide}\ H\ I\ c, e, \phi) \xrightarrow[k]{\epsilon} (\mathbf{hide}\ H\ I\ c', e', \phi' \setminus_I \phi)} \text{ (PS6)}$$

Discrete transition through hidden ports must be possible without additional delay. $\phi \ominus H$ relaxes the intervals of the actions labelled in H, $\phi' \setminus_I \phi$ restores them for persistent actions, and initializes them for newly-enabled actions:

$$
\begin{array}{lcl}
(\phi \ominus H) & = & \{(k,r) \in \phi \mid L(c,k) \notin H\} \cup \{(k,[0,\infty[) \mid k \in \mathcal{D}(\phi) \wedge L(c,k) \in H\} \\
(\phi' \setminus_I \phi) & = & \{(k,r) \in \phi \mid k \in \mathcal{D}(\phi')\} \cup \{(k, I(L(c,k)) \in \phi' \mid k \in \mathcal{D}(\phi') - \mathcal{D}(\phi)\}
\end{array}
$$

Relabelling

$$
\frac{(c,e,\phi) \xrightarrow[k]{l} (c',e',\phi') \qquad l = \epsilon \vee \mathcal{L}(l) \notin \mathcal{D}(R)}{(\mathbf{relab}\ R\ c,e,\phi) \xrightarrow[k]{l} (\mathbf{relab}\ R\ c',e',\phi)} \text{ (PS7)}
$$

$$
\frac{(c,e,\phi) \xrightarrow[k]{p_\tau\ v_1\ \ldots\ v_n} (c',e',\phi') \qquad p_\tau \in \mathcal{D}(R)}{(\mathbf{relab}\ R\ c,e,\phi) \xrightarrow[k]{(R(p_\tau))\ v_1\ \ldots\ v_n} (\mathbf{relab}\ R\ c',e',\phi')} \text{ (PS8)}
$$

Priorities

$$
\frac{(c,e,\phi) \xrightarrow[k]{l} (c',e',\phi') \qquad l = \epsilon \vee (\forall l',k',b)((c,e,\phi) \xrightarrow[k']{l'} b \wedge l' \neq \epsilon \Rightarrow (\mathcal{L}(l'),\mathcal{L}(l)) \notin \Pi)}{(\mathbf{prio}\ \Pi\ c,e,\phi) \xrightarrow[k]{l} (\mathbf{prio}\ \Pi\ c',e',\phi')} \text{ (PS9)}
$$

Priorities forbid a communication action to occur when some other communication labelled with a port with higher priority is possible. Remember that silent actions are never involved in priorities.

Component actions

$$
\frac{(c,e,\phi) \xrightarrow[k]{l} (c',e',\phi')}{(\mathbf{comp}\ C\ c,e,\phi) \xrightarrow[k]{l} (\mathbf{comp}\ C\ c',e',\phi')} \text{ (PS10)}
$$

Process actions

$$
\frac{(s,e \oplus a) \xrightarrow{l} (s',e' \oplus a') \qquad \phi' = \{(\langle l' \rangle,[0,\infty[) \mid (\exists s',e')((s,e \oplus a) \xrightarrow{l'} (s',e'))\}}{(\mathbf{proc}\ P\ (s,a),e,\phi) \xrightarrow[\langle l \rangle]{l} (\mathbf{proc}\ P\ (s',a'),e',\phi')} \text{ (PS11)}
$$

$\langle l \rangle$ (see Section 4.3.2) is the process interaction associated with action l. Function ϕ' associates interval $[0,\infty[$ to all interactions associated with the actions enabled at process configuration $(s,e \oplus a)$. The transition above the line is a process macro-step, the transition below is a component step. From the variable naming hypotheses for component and process instances, (see Section 4.3.1), the store of a process instance can always be seen as partitionned into local variables with "new" names, and shared variables named as in the store e of the component.

4.4 Composition identities

Several composition operators have been provided to accomodate various specification styles, and more might be introduced in the future. For the purpose of formalizing their semantics, all these operators have been expressed in Section 4.3 in terms of rebalelling and product |, the usual synchronous product for timed transition systems. For specification purposes, composition operators enjoy some useful relationships, in particular:

$$\begin{array}{lcl} \textbf{shuffle}\ c_1\ \ldots\ c_n\ \textbf{end} & = & \textbf{par}\ \rightarrow c_1\ \ldots\ \rightarrow c_n\ \textbf{end} \\ \textbf{sync}\ c_1\ \ldots\ c_n\ \textbf{end} & = & \textbf{par}\ \Sigma(c_1) \rightarrow c_1\ \ldots\ \Sigma(c_n) \rightarrow c_n\ \textbf{end} \end{array}$$

References

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in BIP. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Pune*, pages 3–12, September 2006.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. on Software Engineering*, 17(3):259–273, 1991.

[3] B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool tina – construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42-No 14, 2004.

[4] B. Berthomieu, P.-O. Ribet, F. Vernadat, J. Bernartt, J.-M. Farines, J.-P. Bodeveix, M. Filali, G. Padiou, P. Michel, P. Farail, P. Gaufillet, P. Dissaux, and J.-L. Lambert. Towards the verification of real-time systems in avionics: the Cotre approach. In *Proceedings of the 8th International Workshop on Formal Methods for Industrial Critical Systems FMICS'2003, (Trondheim, Norway)*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 201–216. Elsevier, June 2003. Also published as Rapport LAAS Nr. 03185.

[5] Mamoun Filali, Frédéric Lang, Florent Péres, Jan Stoecker, and François Vernadat. Modèles pivots pour la reprśentation des processus concurrents asynchrones, February 2007. Délivrable n^o 4.2.3 du projet ANR05RNTL03101 OpenEmbeDD.

[6] Hubert Garavel. On the introduction of gate typing in E-LOTOS. In Piotr Dembinski and Marek Sredniawa, editors, *Proceedings of the 15th IFIP International Workshop on Protocol Specification, Testing and Verification (Warsaw, Poland)*. IFIP, Chapman & Hall, June 1995.

[7] Hubert Garavel and Frédéric Lang. NTIF: A general symbolic model for communicating sequential processes with data. In Doron Peled and Moshe Vardi, editors, *Proceedings of the 22nd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2002 (Houston, Texas, USA)*, volume 2529 of *Lecture Notes in Computer Science*, pages 276–291. Springer Verlag, November 2002. Full version available as INRIA Research Report RR-4666.

[8] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2006: A toolbox for the construction and analysis of distributed processes. In *Proceedings of the 19th International Conference on Computer Aided Verification CAV'07 (Berlin, Germany)*, 2007.

[9] Hubert Garavel and Mihaela Sighireanu. A graphical parallel composition operator for process algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, 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'99 (Beijing, China)*, pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[10] ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève, September 2001.

[11] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Tr. Comm.*, 24(9):1036–1043, Sept. 1976.

[12] Mihaela Sighireanu. *Contribution à la définition et à l'implémentation du langage "Extended LOTOS"*. Thèse de doctorat, Université Joseph Fourier (Grenoble), January 1999.

[13] Mihaela Sighireanu. LOTOS NT user's manual (version 2.1). INRIA projet VASY. `ftp://ftp.inrialpes.fr/pub/vasy/traian/manual.ps.Z`, November 2000.