The Syntax and Semantics of FIACRE

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

Version 2.0

March 17, 2009

[*] 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, version 2.0. 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 Concrete 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 "$expr_1 \mid expr_2$"
- 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:

A comment is any sequence of characters between the comment brackets '/*' and '*/' in which comment brackets are properly nested.

Reserved words and characters:

Keywords may not be used as identifiers, these are:

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

The following characters and symbolic words are reserved:

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

2.3 Types, type declarations

`type_id ::= IDENT`

`constr ::= IDENT`

`field ::= IDENT`

`type ::=`
  bool
| **nat**
| **int**
| `type_id`
| `exp '..' exp`
| **union** `(constr`$^{+}_{,}$ `[`**of** `type])`$^{+}_{|}$ **end** `[`**union**`]`
| **record** `(field`$^{+}_{,}$ `':' type)`$^{+}_{,}$ **end** `[`**record**`]`
| **array** `exp` **of** `type`
| **queue** `exp` **of** `type`

The `exp`*'s in types are functional expressions. They may make use of declared constants (see Section 2.4). They should evaluate to nonnegative integers (in array and queue types, in which they specify sizes) or to integers (in interval types, in which they specify the interval bounds).*

`type_decl ::=` **type** `type_id` **is** `type`

2.4 Expressions, constant declarations

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

`binop ::=` **enqueue** | **append**

`infixop ::=`

    or
  | **and**
  | ’=’ | ’<>’ |
  | ’<’ | ’>’ | ’<=’ | ’>=’
  | ’+’ | ’-’
  | ’*’ | ’/’ | ’%’

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

var ::= IDENT

literal ::= INTEGER | **true** | **false**

atomexp ::= literal | var | constr | ’(’ exp ’)’

accessexp ::= atomexp | accessexp ’[’ exp ’]’ | accessexp ’.’ field

exp ::=
    accesexp
  | constr [atomexp]
  | unop atomexp
  | binop ’(’ exp ’,’ exp ’)’
  | exp infixop exp
  | exp ’?’ exp ’:’ exp
  | ’[’ exp$^{+}_{,}$ ’]’
  | ’{’ (field ’=’ exp)$^{+}_{,}$ ’}’
  | ’{|’ exp$^{*}_{,}$ ’|}’

const_decl ::= **const** var ’:’ type **is** exp

2.5 Ports, channels, channel declarations

port := IDENT

channel_id ::= IDENT

channel ::= **none** | type$^{+}_{\#}$ | channel_id

channel_decl ::= **channel** channel_id **is** channel

A port is a process communication point. Ports can be used to exchange data. A port type, or channel, determines the type of data that can be exchanged on the port. Channels specified by a series of types separated by ’#’ *are associated with ports transfering several values simultaneously. A port having channel* **none** *may be used as a synchronization port (without any value transfered).*

2.6 Processes

state ::= IDENT

name ::= IDENT

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

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

time_interval ::= left ',' right

port_dec ::= port$^{+}_{,}$ ':' [**in**] [**out**] 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 and channel of a port may be omitted when shared with the following port in the declaration.*

arg_dec ::= ([&] var)$^{+}_{,}$ ':' [**read**] [**write**] type

> *The parameters preceded by symbol* & *are passed by reference, the others are passed by value. Parameters passed by reference may have the* **read** *and/or the* **write** *attribute, specifying the operations that can be done on the variable, by default they have both attributes. The attributes and type of a parameter may be omitted when shared with the following parameter in the declaration.*

var_dec ::= var$^{+}_{,}$ ':' type [':=' exp]

> *Initial values are optional, they may also be specified by an initialisation statement (see below). The type and initial value of a variable may be omitted when shared with the following variable in the declaration.*

transition ::= **from** state statement

atompatt ::= **any** | literal | var | constr | '(' pattern ')'

accesspatt ::= atompatt | accesspatt '[' exp ']' | accesspatt '.' field

pattern ::= accesspatt | constr [atompatt]

statement ::=
  null
  | pattern$^{+}_{,}$ ':=' exp$^{+}_{,}$
  | pattern$^{+}_{,}$ ':=' **any** [**where** exp]
  | **while** exp **do** statement **end** [**while**]
  | **foreach** var **do** statement **end** [**foreach**]

| **if** exp **then** statement (**elsif** exp **then** statement)* [**else** statement] **end** [**if**]
| **select** statement$^{+}_{[]}$ **end** [**select**]
| **case** exp **of** (pattern '->' statement)$^{+}_{|}$ **end** [**case**]
| **to** state
| statement ';' statement
| port
| port '?' pattern$^{+}_{,}$ [**where** exp]
| port '!' exp$^{+}_{,}$

Additional well-formedness constraints are given in Section 3. The last three statement are referred to as "communication" statements.

process_decl ::=
process name
['[' port_dec$^{+}_{,}$ ']']
['(' arg_dec$^{+}_{,}$ ')']
is **states** state$^{+}_{,}$
[**var** var_dec$^{+}_{,}$]
[**init** statement]
transition^{+}

Name of the process, port parameters, functional parameters or references, states and initial state, local variables, followed by an optional initialization statement and a series of transitions. The initialization statement may not perform communications nor read or write variables passed by reference.

2.7 Components

arg ::= exp | '&' var

instance ::= name ['[' port$^{+}_{,}$ ']'] ['(' arg$^{+}_{,}$ ')']

Instance of a process or component. Arguments passed by reference are prefixed by symbol &.

portset ::= '*' | port$^{+}_{,}$

compblock ::= instance | composition

composition ::=
| **par** [portset **in**] ([portset '->'] compblock)$^{+}_{||}$ **end** [**par**]

component_decl ::=
component name
['[' port_dec$^{+}_{,}$ ']']
['(' arg_dec$^{+}_{,}$ ')']
is [**var** var_dec$^{+}_{,}$]

```
[port (port_dec [in time_interval])$^+_,$]
[priority (port$^+_|$ '>' port$^+_|$)$^+_,$]
[init statement]
composition
```

Name of the component, port parameters, functional parameters or references, local variables or references, local ports with delay constraints, priority constraints, followed by an optional initialization statement and a composition. The initialization statement may not include communications or **to** *statements, nor read or write variables passed by reference. In priority declarations,* $\mathtt{a}_1|\ldots|\mathtt{a}_n > \mathtt{b}_1|\ldots|\mathtt{b}_m$ *is a shorthand for* $(\forall i \in \{1,\ldots,n\})(\forall j \in \{1,\ldots,m\})(\mathtt{a}_i > \mathtt{b}_j)$.

2.8 Programs

```
declaration ::=
       type_decl
     | channel_decl
     | const_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 **record** type all labels declared must be distinct;
4. In any **union** type all constructors declared must be distinct;
5. In declarations of ports, formal parameters, or local variables, all identifiers must be distinct;
6. In a **state** declaration, all states must be distinct;
7. There is a single syntactical class for shared variables, formal parameters of processes or components, local variables, and union constructors. As a consequence, the sets of shared variable identifiers, formal parameter identifiers, local variable identifiers and constructors identifiers (declared globally or in the header of some process or component) should be pairwise disjoint;
8. No keyword (e.g. **if**, **from**, etc) may be used as the name of a component, process, variable, constructor, type, channel or port;
9. In any interval type $x..y$, one must have $x \leq y$;
10. In a process, there may be at most one transition from each state declared;
11. In timed local port declarations, time intervals may not be empty (e.g. intervals like $[7, 3[$ or $]1, 1]$ are rejected);
12. In a **priority** declaration, the transitive closure of priority relation defined must be a strict partial order;
13. In an assignment statement, the access expressions in the left-hand side must be pairwise independent. A sufficient condition for that constraint is explained in Section 3.1.2;
14. 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;
15. 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;

3.1.2 Well-formedness of assignment statements:

Assuming constants are replaced by their values (their values must be statically computable), left hand sides of assignment statements have the shape of series of access expressions possibly surrounded by a series of constructors. 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 patterns are independent if, omitting the surrounding constructors, the remaining access expressions $a_0\ a_1\ ...\ a_n$ and $b_0\ b_1\ ...\ b_m$ obey:

- either $a_0 \neq b_0$
- or for some i such that $0 \leq i \leq \min(n, m)$, one of the following conditions hold:
  - a_i and b_i have shapes $[x]$ and $[y]$, respectively, where x and y are different integer constants;
  - a_i and b_i have shapes $.f$ and $.g$, respectively, where f and g are different record labels.
  - a_i or b_i is a universal **any** pattern, a 0-ary construction or a literal.

3.1.3 Initialization of variables:

A static sufficient 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

First, it is assumed throughout this section that the constants declared are replaced throughout (in types and expressions) by their values (values of constants can be statically computed), and that the "size" parameters (for queue and array types) and "range" parameters (for interval types) have been computed too (these expressions may only hold constants and literals).

Next, let us 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.

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 a channel. Unless some attribute is made explicit in its declaration, a port has both **in** and **out** attributes;

- with each shared 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.
- with each constructor its type. If the constructor is 0-ary, this type is a union type. If it is 1-ary, then that type is a function type $\tau_1 \to \tau_2$, in which τ_1 is the type expected for the constructor argument and τ_2 is the result type of the construction;

Typing contexts are written A in the sequel. $A(x)$ denotes the information (attributes and type(s)) assigned to variable (or port) x in A.

3.2.2 Subtyping

Types obtained as explained above are partially ordered by a relation called *subtyping*, written $\leq$ and defined by the following rules:

$$\frac{\tau \in \{\mathbf{bool}, \mathbf{nat}, \mathbf{int}\}}{\tau \leq \tau}\ (\text{SU1}) \qquad \frac{}{\bot \leq \tau}\ (\text{SU2})$$

$$\frac{}{\mathbf{nat}\ \leq\ \mathbf{int}}\ (\text{SU3}) \qquad \frac{x \geq 0}{x\ ..\ y \leq \mathbf{nat}}\ (\text{SU4}) \qquad \frac{}{x\ ..\ y \leq \mathbf{int}}\ (\text{SU5}) \qquad \frac{x_1 \geq x_2 \qquad y_1 \leq y_2}{x_1\ ..\ y_1 \leq x_2\ ..\ y_2}\ (\text{SU6})$$

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

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

$$\frac{\begin{array}{l}\{c^1_1, \ldots, c^1_n\} = \{c^2_1, \ldots, c^2_m\} \\ (\forall i, j)(c^1_i = c^2_j \Rightarrow ((i \leq u \land j \leq v) \lor (i > u \land j > v \land \tau_i \leq \tau'_j))\end{array}}{\begin{array}{ll} & \mathbf{union}\ c^1_1 \mid \ldots \mid c^1_u \mid c^1_{u+1}\ \mathbf{of}\ \tau_{u+1} \mid \ldots \mid c^1_n\ \mathbf{of}\ \tau_n\ \mathbf{end} \\ \leq & \mathbf{union}\ c^2_1 \mid \ldots \mid c^2_v \mid c^2_{v+1}\ \mathbf{of}\ \tau'_{v+1} \mid \ldots \mid c^2_m\ \mathbf{of}\ \tau'_n\ \mathbf{end}\end{array}}\ (\text{SU10})$$

Fields in record types are unordered, as well as variants in union types. In the above rule, constructors are assumed ordered so that those without arguments appear first.

bool *and* **nat** *types are not related by subtyping, nor are record or union types with different sets of fields or constructors, or arrays or queues of different finite sizes.*

The subtyping relation is extended to channels by:

$$\frac{}{\mathbf{none} \leq \mathbf{none}}\ (\text{SU11}) \qquad \frac{\tau_1 \leq \tau'_1 \qquad \ldots \qquad \tau_n \leq \tau'_n}{\tau_1 \# \ldots \# \tau_n \leq \tau'_1 \# \ldots \# \tau'_n}\ (\text{SU12})$$

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

$$\frac{K \in \texttt{INTEGER} \qquad Val(K) = k}{A \vdash K :\ k \mathrel{..} k} \text{ (ET2)} \qquad \frac{k \in \{\mathbf{true}, \mathbf{false}\}}{A \vdash k : \mathbf{bool}} \text{ (ET3)}$$

By abuse of notation, we write $K \in$ `INTEGER` to mean that K belongs to the `INTEGER` syntactical class. Function Val associates with a token in the `INTEGER` or `NATURAL` class the integer it denotes.

Variables, constants and constructions (union values)

$$\frac{A(X) = \{\tau\}}{A \vdash X : \tau} \text{ (ET4)} \qquad \frac{\{\mathbf{read}, \tau\} \subseteq A(X)}{A \vdash X : \tau} \text{ (ET5)} \qquad \frac{A(C) = \{\tau \to \tau'\} \qquad A \vdash e : \tau}{A \vdash C\ e : \tau'} \text{ (ET6)}$$

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

Arithmetic and logical primitives

$$\frac{A \vdash x : \mathbf{bool}}{A \vdash \mathbf{not}\ x : \mathbf{bool}} \text{ (ET7)} \qquad \frac{A \vdash x : \mathbf{bool} \qquad A \vdash y : \mathbf{bool} \qquad (@ \in \{\mathbf{and}, \mathbf{or}\})}{A \vdash x@y : \mathbf{bool}} \text{ (ET8)}$$

$$\frac{A \vdash x : \tau \qquad \tau \leq \mathbf{int}}{A \vdash -x : \tau} \text{ (ET9)} \qquad \frac{A \vdash x : \tau \qquad \tau' \leq \tau \leq \mathbf{int}}{A \vdash \$\ x : \tau'} \text{ (ET10)}$$

$$\frac{A \vdash x : \tau \qquad A \vdash y : \tau \qquad \tau \leq \mathbf{int} \qquad (@ \in \{+, -, *, /, \%\})}{A \vdash x@y : \tau} \text{ (ET11)}$$

$$\frac{A \vdash x : \tau \qquad A \vdash y : \tau \qquad \tau \leq \mathbf{int} \qquad (@ \in \{<, <=, >, >=\})}{A \vdash x@y : \mathbf{bool}} \text{ (ET12)}$$

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

*Except for the coercion operator \$, arithmetic primitives are homogeneous: their argument(s) and result have the same type. \$ converts a numeric value of some type $\tau \leq$ **int** into a value of some subtype τ' of τ.*

Records

$$\frac{\{f_1, \ldots, f_n\} \subseteq \texttt{Fields} \qquad A \vdash l_1 : \tau_1 \qquad \ldots \qquad A \vdash l_n : \tau_n}{A \vdash \{f_1 : l_1, \ldots, f_n : l_n\} : \textbf{record } f_1 : \tau_1, \ldots, f_n : \tau_n \textbf{ end}} \text{ (ET14)}$$

$$\frac{A \vdash P : \textbf{record } \ldots, f : \tau, \ldots \textbf{ end}}{A \vdash P.f : \tau} \text{ (ET15)}$$

`Fields` *is the set of record field identifiers declared in* **record** *types.*

Arrays

$$\frac{A \vdash k_1 : \tau \qquad \ldots \qquad A \vdash k_n : \tau}{A \vdash [k_1, \ldots, k_n] : \textbf{ array of } n\ \tau} \text{ (ET16)}$$

$$\frac{A \vdash P : \textbf{array of } k\ \tau \qquad A \vdash E : 0\ ..\ k\text{–}1}{A \vdash P[E] : \tau} \text{ (ET17)}$$

Queues

$$\frac{A \vdash k_1 : \tau \qquad \ldots \qquad A \vdash k_n : \tau \qquad 0 \leq n \leq m}{A \vdash \{|k_1, \ldots, k_n|\} : \textbf{ queue } m \textbf{ of } \tau} \text{ (ET18)}$$

$$\frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{empty } q : \textbf{bool}} \text{ (ET19)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{full } q : \textbf{bool}} \text{ (ET20)}$$

$$\frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{first } q : \tau} \text{ (ET21)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau}{A \vdash \textbf{dequeue } q : \textbf{queue of } k\ \tau} \text{ (ET22)}$$

$$\frac{A \vdash q : \textbf{queue of } k\ \tau \qquad A \vdash E : \tau}{A \vdash \textbf{enqueue } (q, E) : \textbf{queue of } k\ \tau} \text{ (ET23)} \qquad \frac{A \vdash q : \textbf{queue of } k\ \tau \qquad A \vdash E : \tau}{A \vdash \textbf{append } (q, E) : \textbf{queue of } k\ \tau} \text{ (ET24)}$$

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:

$$\frac{A(X) = \{\tau\}}{A \vdash X :_p \tau}\ \text{(LT1)} \qquad \frac{\{\mathbf{write}, \tau\} \subseteq A(X)}{A \vdash X :_p \tau}\ \text{(LT2)} \qquad \frac{A(C) = \{\tau \to \tau'\} \qquad A \vdash e :_p \tau}{A \vdash C\ e :_p \tau'}\ \text{(LT3)}$$

$$\frac{K \in \mathtt{INTEGER} \qquad A \vdash K : \tau}{A \vdash K :_p\ \ \tau}\ \text{(LT4)} \qquad \frac{k \in \{\mathbf{true}, \mathbf{false}\}}{A \vdash k :_p \mathbf{bool}}\ \text{(LT5)} \qquad \frac{}{A \vdash \mathbf{any} :_p \tau}\ \text{(LT6)}$$

$$\frac{A \vdash P :_p \mathbf{array\ of}\ k\ \tau \qquad A \vdash E : 0\ ..\ k\text{–}1}{A \vdash P[E] :_p \tau}\ \text{(LT7)}$$

$$\frac{A \vdash P :_p \mathbf{record}\ \ \ldots, f : \tau, \ldots\ \ \mathbf{end}}{A \vdash P.f :_p \tau}\ \text{(LT8)}$$

Shared 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 three properties:

(a) That the variables and expressions occurring in the transition are used consistently;

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

(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. A statement type may hold both **Shm** and **Com**, provided shared variables and communications are performed along different control paths.

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

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

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

Assignments and Case

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

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

$$\frac{\begin{array}{l} A \vdash P_1 :_p \tau \quad \ldots \quad A \vdash P_n :_p \tau \qquad A \vdash E : \tau \\ A \vdash S_1 :_s \alpha_1 \quad \ldots \quad A \vdash S_n :_s \alpha_n \qquad (\forall i \in 1..n)(\mathcal{Q}_{\mathcal{A}}(P_i) \Rightarrow \alpha_i \subseteq \{\mathbf{Shm}\}) \end{array}}{A \vdash \mathbf{case}\ e\ \mathbf{of}\ P_1 \to S_1 \mid \ \ldots \ \mid P_n \to S_n\ \mathbf{end} :_s \alpha_1 \cup \cdots \cup \alpha_n}\ (\mathrm{ST8})$$

Choices and while loop

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S_1 :_s \alpha_1 \qquad A \vdash S_2 :_s \alpha_2 \qquad \mathcal{Q}_{\mathcal{A}}(E) \Rightarrow \alpha_1 \cup \alpha_2 \subseteq \{\mathbf{Shm}\}}{A \vdash \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} :_s \alpha_1 \cup \alpha_2}\ (\mathrm{ST9})$$

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S :_s \alpha \qquad \mathbf{Com} \notin \alpha}{A \vdash \mathbf{while}\ E\ \mathbf{do}\ S\ \mathbf{end} :_s \mathcal{Q}_{\mathcal{A}}(E) \to \{\mathbf{Shm}\} \mid \alpha}\ (\mathrm{ST10})$$

$$\frac{A(v) = \{x\ ..\ y\} \qquad A \vdash S :_s \alpha \qquad \mathbf{Com} \notin \alpha}{A \vdash \mathbf{foreach}\ v\ \mathbf{do}\ S\ \mathbf{end} :_s \mathcal{Q}_{\mathcal{A}}(v) \to \{\mathbf{Shm}\} \mid \alpha}\ (\mathrm{ST11})$$

$$\frac{A \vdash S_1 :_s \alpha_1 \quad \ldots \quad A \vdash S_n :_s \alpha_n}{A \vdash \mathbf{select}\ S_1\ [\,]\ ...\ [\,]\ S_n\ \mathbf{end} :_s \alpha_1 \cup \cdots \cup \alpha_n}\ (\mathrm{ST12})$$

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

Communications

$$\frac{\mathbf{none} \in A(p)}{A \vdash p :_s \{\mathbf{Com}\}}\ (\mathrm{ST13})$$

$$\frac{A \vdash E_1 : \tau_1 \quad \ldots \quad A \vdash E_n : \tau_n \qquad \{\mathbf{out}, \tau_1\# \ldots \#\tau_n\} \subseteq A(p) \qquad \neg(\exists i \in 1..n)(\mathcal{Q}_{\mathcal{A}}(E_i))}{A \vdash p\ !\ E_1, \ldots, E_n :_s \{\mathbf{Com}\}}\ (\mathrm{ST14})$$

$$\frac{\begin{array}{l} A \vdash X_1 :_p \tau_1 \quad \ldots \quad A \vdash X_n :_p \tau_n \qquad A \vdash E : \mathbf{bool} \\ \{\mathbf{in}, \tau_1\# \ldots \#\tau_n\} \subseteq A(p) \qquad \neg(\exists i \in 1..n)(\mathcal{Q}_{\mathcal{A}}(X_i)) \vee \mathcal{Q}_{\mathcal{A}}(E) \end{array}}{A \vdash p\ ?\ X_1, \ldots, X_n\ \mathbf{where}\ E :_s \{\mathbf{Com}\}}\ (\mathrm{ST15})$$

In an output communication, the tuple of types of the values sent must be a subtype of the channel declared for the port. In an input communication, the channel declared for the port must be a subtype of the tuple of types of the reception pattern.

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 $((\ldots,\mu_i,\ldots),(\ldots,\eta_j,\ldots))$, in which μ_i is the set of attributes and channel 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 in components are given types and attributes by relation $:_x$, defined by:

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

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

$$\frac{A,I\vdash c_1 :_c \mathbf{ok} \quad \ldots \quad A,I\vdash c_n :_c \mathbf{ok} \qquad (\forall i)(Q_i\subseteq\Sigma(c_i))}{A,I\vdash \mathbf{par}\ Q_1\rightarrow c_1 \ ||\ \ldots\ ||\ Q_n\rightarrow c_n\ \mathbf{end} :_c \mathbf{ok}}\ \text{(CT3)}$$

The sort $\Sigma(c)$ *of a composition* c *is computed as follows, according to the structure of* c*:*

$$\begin{array}{lcl} \Sigma(\mathbf{par}\ e_1\rightarrow c_1\ ||\ \ldots\ ||\ e_n\rightarrow c_n\ \mathbf{end}) & = & \Sigma(c_1)\cup\cdots\cup\Sigma(c_2) \\ \Sigma(P\ [q_1,\ldots,q_m]\ (v_1,\ldots,v_l)) & = & \{q_1,\ldots,q_m\} \end{array}$$

$\mathbf{par}\ldots\ ||\ c_i\ ||\ \ldots\ \mathbf{end}$ *stands for* $\mathbf{par}\ldots\ ||\ \emptyset\rightarrow c_i\ ||\ \ldots\ \mathbf{end}$

$\mathbf{par}\ Q\ \mathbf{in}\ Q_1\rightarrow c_1\ ||\ \ldots\ ||\ Q_n\rightarrow c_n\ \mathbf{end}$ *stands for*
$\mathbf{par}\ (Q\cup Q_1)\rightarrow c_1\ ||\ \ldots\ ||\ (Q\cup Q_n)\rightarrow c_n\ \mathbf{end}$

If $*\in Q$*, then* $Q\rightarrow c$ *is handled like* $\Sigma(c)\rightarrow c$*.*

$$\frac{A\vdash e_1 :_x \eta_1 \quad \ldots \quad A\vdash e_n :_x \eta_n \qquad ((A(p_1),\ldots,A(p_n)),(\eta_1,\ldots,\eta_m))\prec I(C)}{A,I\vdash C\ [p_1,\ldots,p_n]\ (e_1,\ldots,e_m) :_c \mathbf{ok}}\ \text{(CT4)}$$

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.

3.3 Choosing types for expressions

Expressions may have in general several types: arithmetic expressions typically have several types, resulting from the subtyping rules of arithmetics, the empty queue constant $\{|\ |\}$ has any queue type.

The typing rules and method explained in section 3.2 ensure that all expressions in some program can be given at least one type such that the whole program is well-typed.

Now, as will be seen, the semantics of arithmetic operations depends on their type, which is why it is necessary to explain the rules leading to a choice of a particular type for arithmetic primitives and constants when several types are admissible.

The rule retained is the following: when several types are admissible for an expression, the Fiacre typechecker assigns to it the largest (by the subtyping relation) type possible permitted by the context. If no such largest type is implied by the context, then the expression is rejected (considered ill-typed).

As an illustrative example, consider the following statements, with the assumption that the enclosing process or component holds the declarations x : 0..255, y : **int**, q : **queue** 5 *of* **nat**:

1. $y := x + 5$

   Pattern y has type **int**, expression x has any supertype of 0..255. Hence, expression $x+5$ has all types which are subtypes of **int** and supertypes of 0..255; type **int** will be selected;

2. $x := x + 5$

   Expression $x + 5$ admits a single type: 0..255;

3. **if** $x > 1000$ **then**...

   The arguments of $>$ are only required to be subtypes of **int**, hence both that instance of x and constant 1000 are assigned type **int**;

4. **if** $q = \{|\ |\}$ **then**...

   Similarly, constant $\{|\ |\}$ has here the type of q : **queue** 5 *of* **nat**;

5. **if** $\{|\ |\} = \{|\ |\}$ **then**...

   The context does not provide any upper bound for the types of the empty queue constants, hence this statement is rejected.

In the next Section, overloaded primitives whose interpretation depends on their type are assumed annotated with an indication of the type chosen by the above method. This concerns arithmetic primitives (annotated by the type of their(s) argument(s)) and queues primitives (annotated by the type of their queue argument).

4 Timed operational semantics

All programs in this section are assumed well-formed and well-typed. Declared constants are assumed replaced throughout by their statically computed values. Overloaded primitives are assumed annotated as explained in Section 3.3.

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 $\mathbf{D}$ 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*;

$\mathbf{S}$ be the set of finite 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 $\mathbf{S}$ 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 $\mathbf{B}$, arrays in some set $Arrays(D_n)$, union constants as strings, tagged unions and records in some set $Records(D_n)$, for some finite n, all subsets of D. 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 $\mathcal{D}(q)$.
- *append* $q\ e$ is the mapping q' such that $q'(0) = e$ and $q'(x+1) = q(x)$ for $x \in \mathcal{D}(q)$.

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.

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 value of expression E with store e is v. The store e may be omitted if the result does not depend on its contents.

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

Core expressions

- Numeric constants denote integers in $\mathbb{Z}$. Implementations may choose to reject literals that are not machine representable;
- 0-ary constructors (union constants) denote strings in $\mathbf{S}$;
- The booleans **true** and **false** denote values $true$ and $false$ in $\mathbf{B}$, respectively;
- Representing mappings by their graphs, records, arrays and queues are given meanings by:

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

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

$$\frac{}{\vdash \{|\ |\} \rightsquigarrow \emptyset}\ \text{(ES3)} \qquad \frac{\vdash l_1 \rightsquigarrow v_1 \quad \dots \quad \vdash l_n \rightsquigarrow v_n}{\vdash \{|l_1, \dots, l_n|\} \rightsquigarrow \{(0, v_1), \dots, (n-1, v_n)\}}\ \text{(ES4)}$$

- 0-ary constructors (union constants) denote strings in $\mathbf{S}$, constructions denote pairs in $\mathbf{S} \times \mathbf{D}$.

$$\frac{}{e \vdash C \rightsquigarrow |C|}\ \text{(ES5)} \qquad \frac{e \vdash E \rightsquigarrow v}{e \vdash C\ E \rightsquigarrow \{(|C|, v)\}}\ \text{(ES6)}$$

  $|C| \in \mathbf{S}$ *the name of the constructor.*

  Given a value v and a pattern P, the *matching* predicate $\mathcal{M}(v, P)$, read "v matches P" is defined as follows, according to the structure of P (C is a constructor):

  $\mathcal{M}((c, v), C\ P)$ iff $c \in \mathbf{S} \wedge c = |C| \wedge \mathcal{M}(v, P)$

  $\mathcal{M}(c, C)$ iff $c \in \mathbf{S} \wedge c = |C|$

  $\mathcal{M}(l, L)$ iff $\vdash L \rightsquigarrow l$ (L is a numeric or boolean literal)

  $\mathcal{M}(v, X)$ true (X is a variable or an access pattern)

- Variables evaluate to the values they are bound to in the store. 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.

$$\frac{X \in \mathcal{D}(e)}{e \vdash X \rightsquigarrow e(X)}\ \text{(ES7)}$$

- Array and record access evaluate the obvious way (arrays are indexed from 0). Well-typing ensures that array indices, when their evaluation succeed, cannot be out of range, nor fields undefined in the records they are sought for.

$$\frac{e \vdash P \rightsquigarrow a \quad e \vdash E \rightsquigarrow i \quad i \in \mathcal{D}(a)}{e \vdash P[E] \rightsquigarrow a(i)} \text{(ES8)} \qquad \frac{e \vdash P \rightsquigarrow r \quad f \in \mathcal{D}(r)}{e \vdash P.f \rightsquigarrow r(f)} \text{(ES9)}$$

- Conditional expressions are given meanings as follows:

$$\frac{e \vdash E_c \rightsquigarrow true \qquad e \vdash E_1 \rightsquigarrow v}{e \vdash E_c \ ?\ E_1 : E_2 \rightsquigarrow v} \text{(ES10)} \qquad \frac{e \vdash E_c \rightsquigarrow false \qquad e \vdash E_2 \rightsquigarrow v}{e \vdash E_c \ ?\ E_1 : E_2 \rightsquigarrow v} \text{(ES11)}$$

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 (see Section 3.3). The primitives whose semantics is type-dependent appear in the semantic rules with type annotations added (by the typechecker).

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**):

$$\frac{e \vdash x \rightsquigarrow a \qquad In(-a,\tau)}{e \vdash -_\tau\ x \rightsquigarrow -a} \text{(ES12)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b \qquad In(a \ @\ b,\tau) \qquad @ \in \{+,-,*\}}{e \vdash x\ @_\tau\ y \rightsquigarrow a\ @\ b} \text{(ES13)}$$

$$\frac{e \vdash x \rightsquigarrow a \qquad In(a,\tau)}{e \vdash \$_\tau\ x \rightsquigarrow a} \text{(ES14)}$$

$$\frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b \qquad b \neq 0 \qquad In(a\ @\ b,\tau) \qquad @ \in \{/,\%\}}{e \vdash x\ @_\tau\ y \rightsquigarrow a\ @\ b} \text{(ES15)}$$

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* $In(v, a..b)$ *if* $a \leq v \leq b$*. Implementations may strengthen predicate* In *by conditions asserting that the results are machine representable.*

- Boolean primitives:

$$\frac{e \vdash x \rightsquigarrow a}{e \vdash\ \mathbf{not}\ x \rightsquigarrow\ not\ a} \text{(ES16)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b}{e \vdash x\ \mathbf{and}\ y \rightsquigarrow a\ and\ b} \text{(ES17)} \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash y \rightsquigarrow b}{e \vdash x\ \mathbf{or}\ y \rightsquigarrow a\ or\ b} \text{(ES18)}$$

Boolean operators are evaluated functionally. Lazy boolean operators can be implemented with conditional expressions.

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

$$\frac{e \vdash x \rightsquigarrow a \qquad e \vdash x \rightsquigarrow b \qquad a\ @\ b}{e \vdash x\ @\ y \rightsquigarrow true}\ (\text{ES19}) \qquad \frac{e \vdash x \rightsquigarrow a \qquad e \vdash x \rightsquigarrow b \qquad \neg(a\ @\ b)}{e \vdash x\ @\ y \rightsquigarrow false}\ (\text{ES20})$$

- Primitives for queues at type τ (a queue type)

Assuming τ is some queue type **queue** N **of** τ', $Cap(\tau)$ denote capacity N.

$$\frac{e \vdash q \rightsquigarrow Q}{e \vdash \textbf{empty}\ q \rightsquigarrow empty\ Q}\ (\text{ES21}) \qquad \frac{e \vdash q \rightsquigarrow Q}{e \vdash \textbf{full}_\tau\ q \rightsquigarrow full\ (Cap(\tau))\ Q}\ (\text{ES22})$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \textbf{first}\ q \rightsquigarrow first\ Q}\ (\text{ES23}) \qquad \frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \textbf{dequeue}\ q \rightsquigarrow dequeue\ Q}\ (\text{ES24})$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad e \vdash x \rightsquigarrow v \qquad Cap(\tau) - 1 \notin \mathcal{D}(Q)}{e \vdash \textbf{enqueue}_\tau\ (q, x) \rightsquigarrow enqueue\ Q\ v}\ (\text{ES25})$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad e \vdash x \rightsquigarrow v \qquad Cap(\tau) - 1 \notin \mathcal{D}(Q)}{e \vdash \textbf{append}_\tau\ (q, x) \rightsquigarrow append\ Q\ v}\ (\text{ES26})$$

first *and* **dequeue** *are undefined on empty queues.* **enqueue**$_\tau$ *and* **append**$_\tau$ *are undefined on full queues (already holding $Cap(\tau)$ elements).*

4.1.4 Patterns

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 $z \in \mathcal{D}(e) - \{x_1, \ldots, x_n\}$.

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

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

$$\frac{}{e \vdash C \rightsquigarrow^l \emptyset, e}\ (\text{LS3})$$

Where C is a literal (numeric or boolean constant) or a 0-ary constructor.

$$\frac{e \vdash P \rightsquigarrow^l e', r}{e \vdash C\ P \rightsquigarrow^l e', r}\ \text{(LS4)}$$

Where C is a 1-ary constructor.

4.2 Semantics of Processes

4.2.1 Semantics of statements

The semantics of statements is expressed operatioally by a labelled relation. The relation holds triples $(S, e) \overset{l}{\Rightarrow} (S', e')$ in which:

- S is a statement;
- e, e' are stores;
- $S' \in \{\texttt{done}\} \cup \{\texttt{target}\ s | s \in \Lambda\}$, where Λ is the declared set of states of the process;
- l is either a communication action or the silent action ϵ. Communication actions are sequences $p\ v_1 \dots v_n$, in which p is a port and $v_1 \dots v_n$ $(n \geq 0)$ are values.

Relation $\overset{l}{\Rightarrow}$ is defined inductively from the structure of statements, by the following rules.

To, null

$$\frac{}{(\textbf{to}\ s, e) \overset{\epsilon}{\Rightarrow} (\texttt{target}\ s, e)}\ \text{(SS1)} \qquad \frac{}{(\textbf{null}, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)}\ \text{(SS2)}$$

Deterministic assignment

$$\frac{\begin{array}{llll} e \vdash E_1 \rightsquigarrow v_1 & e \vdash E_2 \rightsquigarrow v_2 & \dots & e \vdash E_n \rightsquigarrow v_n \\ e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \dots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ \mathcal{M}(v_1, P_1) & \mathcal{M}(v_2, P_2) & \dots & \mathcal{M}(v_n, P_n) \\ e' = a_n(v_n) & & & \end{array}}{(P_1, P_2, \dots, P_n := E_1, E_2, \dots, E_n, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e')}\ \text{(SS3)}$$

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 $P_1, \dots, P_n$ and the corresponding expressions $E_1, \dots, E_n$.

Nondeterministic assignment

$$\frac{\begin{array}{llll} e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \dots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ e' = a_n(v_n) & [e' \vdash E \rightsquigarrow \mathit{true}] & & \end{array}}{(P_1, P_2, \dots, P_n := \textbf{any}\ [\textbf{where}\ E], e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e')}\ \text{(SS4)}$$

v_i ranges over all values of the type of P_i (necessarily a boolean or numeric type).

While, foreach

$$\frac{e \vdash E \rightsquigarrow true \qquad (S; \textbf{while } E \textbf{ do } S \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{while } E \textbf{ do } S \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')} \text{ (SS5)}$$

$$\frac{e \vdash E \rightsquigarrow false}{(\textbf{while } E \textbf{ do } S \textbf{ end}, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)} \text{ (SS6)}$$

It is assumed that condition E eventually evaluates to $false$.

$$\frac{(V := v_1 \ ; \ S \ ; \ldots V := v_n \ ; \ S, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{foreach } V \textbf{ do } S \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')} \text{ (SS7)}$$

Where $v_1, \ldots, v_n$ is the set of values of interval type V, in increasing order.

Deterministic choice

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

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

Case

$$\frac{e \vdash E \rightsquigarrow v \qquad \mathcal{M}(v, P_1) \qquad (P_1 := E \ ; \ S_1, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{case } E \textbf{ of } P_1 \rightarrow S_1 \mid \ldots \mid P_n \rightarrow S_n \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')} \text{ (SS10)}$$

$$\frac{e \vdash E \rightsquigarrow v \qquad \neg\mathcal{M}(v, P_1) \qquad (\textbf{case } E \textbf{ of } P_2 \rightarrow S_2 \mid \ldots \mid P_n \rightarrow S_n \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{case } E \textbf{ of } P_1 \rightarrow S_1 \mid \ldots \mid P_n \rightarrow S_n \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')} \text{ (SS11)}$$

Nondeterministic choice

$$\frac{(S_i, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{select } S_1 \ [\,] \ \ldots \ [\,] \ S_n \textbf{ end}, e) \overset{l}{\Rightarrow} (S', e')} \text{ (SS12)}$$

Sequential composition

$$\frac{(S_1, e) \overset{l}{\Rightarrow} (\texttt{target } s, e')}{(S_1; S_2, e) \overset{l}{\Rightarrow} (\texttt{target } s, e')} \text{ (SS13)} \qquad \frac{(S_1, e) \overset{l_1}{\Rightarrow} (\texttt{done}, e') \qquad (S_2, e') \overset{l_2}{\Rightarrow} (S', e'')}{(S_1; S_2, e) \xRightarrow{l_1.l_2} (S', e'')} \text{ (SS14)}$$

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

$$\frac{}{(p_\tau, e) \overset{p}{\Longrightarrow} (\texttt{done}, e)} \text{ (SS15)}$$

$$\frac{e \vdash E_1 \rightsquigarrow v_1 \quad \ldots \quad e \vdash E_n \rightsquigarrow v_n}{(p!E_1, \ldots, E_n, e) \xRightarrow{p\ v1\ \ldots\ vn} (\texttt{done}, e)} \text{ (SS16)}$$

$$\frac{\begin{array}{llll} e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \ldots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ e' = a_n(v_n) & [e' \vdash E \rightsquigarrow \mathit{true}] & & \end{array}}{(p?P_1, P_2, .., P_n\ [\textbf{where}\ E], e) \xRightarrow{p\ v_1\ \ldots\ v_n} (\texttt{done}, e')} \text{ (SS17)}$$

v_i ranges over all values of the type of P_i.

4.2.2 Process configurations, initial process configurations

A *process configuration* is a pair (s, e) constituted of a process state s and a store e capturing the values of all variables referred to in the process. Each process has a set of *initial configurations*, obtained as follows:

- Let s_0 be a store capturing the values for all parameters and local variables of the process, given their actual declared values. If some variable was not initialized in its declaration, then any value can be chosen for it in s_0 (e.g. $\emptyset$) as the "well-initialized" condition explained in Section 3.1.3 guarantees that this default value will not be used;

- Then:

– If the process has no **init** statement, it admits a single *initial configuration*: (s_0, e_0), in which s_0 is the source state of the first transition of the process;

– If the process has an **init** statement S_i then, from the static restrictions put upon **init** statements, (S_i, e_0) necessarily evaluates by $\overset{\epsilon}{\Rightarrow}$ to some pair $(\texttt{target}\ s, e)$. Each such $(\texttt{target}\ s, e)$ defines a possible initial configuration (s, e) for the process.

4.3 Semantics of components

The semantics, or behavior, of a component is a *Timed Transition System*. These are Labelled Transition Systems extended with state properties and time-elapsing transitions. The semantics of a component is obtained compositionally from the semantics of the process instances and component instances it captures and the composition operator. Priority and timing constraints possibly restrict the result of compositions. Finally, some ports may be made local to the component, preventing further interactions through these.

4.3.1 Component configurations, terminology

Abstract components: Consider the following grammar of abstract components:

c	$::=$	$\textbf{hide}\ H\ c$	hiding
	$\mid$	$\textbf{prio}\ \Pi\ c$	priority
	$\mid$	c'	composition
c'	$::=$	$\textbf{par}\ g_1 \rightarrow c'_1 \mid\mid \ldots \mid\mid g_2 \rightarrow c'_2\ \textbf{end}$	composition
	$\mid$	$\textbf{comp}\ \Phi\ F\ (c, a)$	component instance
	$\mid$	$\textbf{proc}\ \Phi\ F\ (s, a)$	process instance

Any Fiacre component will be represented by an abstract term of the form: $\textbf{hide}\ H\ (\textbf{prio}\ \Pi\ c)$, in which c is some term involving only compositions and instances. In this term:

- H maps port labels to time intervals (default $[0, \infty)$), as made explicit in the local port declaration of the component, if any;
- Π is the priority relation, translating the priorities declared in the process, if any;

Compositions are assumed "normalized": stars and factorized port sets are eliminated as explained in Section 3.2.6. The leaves of compositions are component instances $\textbf{comp}\ \Phi\ F\ (c, a)$ or process instances $\textbf{proc}\ \Phi\ F\ (s, a)$, in which:

- A process or component instance interacts through the ports passed as arguments to it. Φ is a port substitution, mapping the ports declared in the header of the process or component to those passed as arguments. Substitution Φ is extended to communication action by:

  $$\Phi(\epsilon) = \epsilon$$

  $$\Phi(p\ v_1\ \ldots\ v_n) = (\Phi\ p)\ v_1\ \ldots\ v_n$$

- (c, a) in $\textbf{comp}\ \Phi\ F\ (c, a)$ is an abstract component and a store. F is a function that, given the variables of the component passed as references to the instance, renames these so that they bear the names they have in the instance body.
- (s, a) in $\textbf{proc}\ \Phi\ F\ (s, a)$ is a process configuration. F is defined as for component instances.

Interactions: For any communication action l, let us define $\mathcal{L}(l)$ by:

$$\mathcal{L}(\epsilon) = \epsilon$$
$$\mathcal{L}(p\ v_1\ \ldots\ v_n) = p$$

Then, assuming transition ($\textbf{from}\ s\ S$) is among those of a process, the enabled interactions of that process at configuration (s, e) is the set:

$$\Omega(s, e) = \{\mathcal{L}(l) \mid (\exists s', e', l)((S, e) \overset{l}{\Rightarrow} (\texttt{target}\ s', e'))\}$$

Components are typically defined as products of process instances and/or other components, hence process interactions are typically products of process or component interactions. Abstractly, component interactions obey the following syntax, in which symbol $\bullet$ is used to mean that a particular subcomponent does not participate in an interaction:

k	$::=$	ϵ	silent process interaction
	$\mid$	p	labelled process interaction
	$\mid$	$(k'_1, \ldots, k'_n)$	product interaction, for any $n >= 2$
k'	$::=$	$\bullet$	not involved
	$\mid$	k	component interaction

An interaction is *independant* from another, written $k \iota k'$, if the sets of subcomponents they respectively involve (the indices of their members distinct from $\bullet$) are disjoints.

Component configurations, initial component configurations Component configurations are triples (c, e, ψ) in which c is an abstract component, as defined in Section 4.3.1, e is a store, and ψ is a function called the *interaction environment.* Function ψ maps component interactions at that configuration to a time interval and a communication label or ϵ.

As for processes, the values of the parameters passed to a process, of its local variables, and possibly its initialization statement defines its initial store s_0. A component may admit several initial stores.

The initial configurations of a component all have the shape of the abstract component, in which abstract process instances **proc** $\Phi\ F\ (s, a)$ capture an initial configuration of the process and abstract component instances **comp** $\Phi\ F\ (c, a)$ capture an initial abstract state and initial store of the subcomponent.

4.3.2 Semantic rules for components

The labelled semantics relation, linking component configurations, has two sorts of transitions: discrete ($\xrightarrow[k]{l}$, specifying a communication action and an interaction) and continuous (time elapse transitions $\xrightarrow{\theta}$ where θ is some nonnegative real number).

Continuous transitions

$$\frac{(\forall k \in \mathcal{D}(\psi))(\theta \leq \uparrow\psi(k))}{(c, e, \psi) \xrightarrow{\theta} (c, e, \psi \mathbin{\dot{-}} \theta)} \text{ (PS1)}$$

Time may elapse as long as no time-constrained interaction overflows its deadline.
For any interval r, $\uparrow r$ is its right endpoint, or ∞ if r is not right-bounded.
For any k: $(\psi \mathbin{\dot{-}} \theta)(k) = (I, b)$ where $I = \{x - \theta \mid (x - \theta) \geq 0 \wedge x \in I'\}$ and $(I', b) = \psi(k)$.

Discrete Transitions

Compositions

For any abstract composition $c =$ **par** $g_1 \to c_1 \mid\mid \ldots \mid\mid g_n \to c_n$ **end** with n components and any $A = \{a_1, \ldots, a_m\} \subseteq \{1, \ldots, n\}$, let $c[g_{a_1} \to c_{a_1}, \ldots, g_{a_m} \to c_{a_m}]^n_A$ denote the result of replacing the components of c indexed over A by those made explicit between the brackets.

Similarly, given a product $\psi = \psi_1 \times \cdots \times \psi_n$ of n interaction environments (soon to be defined) and $A \subseteq \{1, \ldots, n\}$, $\psi[\psi_{a_1}, \ldots, \psi_{a_m}]^n_A$ will denote ψ in which the members indexed over A are replaced by those between the brackets, respectively.

Finally, let us denote $\bullet[k_{a_1}, \ldots, k_{a_m}]^n_A$ a product of n interactions all of them being $\bullet$ except those indexed over A which have the values shown between the brackets.

The semantics of compositions is then defined by the following two rules:

$$\frac{(c', e', \psi') \xrightarrow[k]{l} (c'', e'', \psi'') \qquad l = \epsilon \vee \mathcal{L}(l) \notin g}{(c[g \to c']^n_{\{i\}}, e', \psi[\psi']^n_{\{i\}}) \xrightarrow[\bullet[k]^n_{\{i\}}]{l} (c[g \to c'']^n_{\{i\}}, e'', \psi[\psi'']^n_{\{i\}})} \text{(PS2)}$$

$$\frac{(\forall j \in A)((c_j, e, \psi_j) \xrightarrow[k_j]{p\ v_1\ \ldots\ v_k} (c'_j, e, \psi'_j) \qquad (\forall i \in \{1, \ldots, n\})(c = c[(g \cup \{p\}) \to c']^n_{\{i\}} \Rightarrow i \in A)}{\begin{array}{l}(c[g_{a_1} \to c_{a_n}, \ldots, g_{a_m} \to c_{a_m}]^n_A, e, \psi[\psi_{a_1}, \ldots, \psi_{a_m}]^n_A) \\ \xrightarrow[\bullet[k_{a_1},\ldots,k_{a_m}]^n_A]{p\ v_1\ \ldots\ v_k} (c[g_{a_1} \to c'_{a_1}, \ldots, g_{a_m} \to c'_{a_m}]^n_A, e, \psi[\psi'_{a_1}, \ldots, \psi'_{a_m}]^n_A)\end{array}} \text{(PS3)}$$

Where the product $\psi_1 \times \cdots \times \psi_n$ of n interaction environments is defined as follows, depending on the guards g_i in the composition operator:

$$\begin{array}{lll} & \{(\bullet[k]^n_{\{i\}}, (r, b)) & | \quad (k, (r, b)) \in \psi_i \wedge (b = \epsilon \vee b \notin g_i)\} \\ \cup & \{(\bullet[k_{a_1}, \ldots, k_{a_m}]^n_A, (r, b)) & | \quad (\forall j \in A)((k_j, (r, b)) \in \psi_j \wedge b \in g_j) \wedge \\ & & \quad\ (\forall i \in \{1, \ldots, n\})(b \in g_i \Rightarrow i \in A)\} \end{array}$$

Intuitively, the domain of $\psi_1 \times \cdots \times \psi_n$ is the set of possible interactions of the composition, built from the interactions of the subcomponents and the particular composition operator. Note in this rule that the store does not change; well-typing implies that the store may only change on silent transitions.

Hiding

$$\frac{(c, e, Relax_H\ \psi) \xrightarrow[k]{l} (c', e', \psi') \qquad \mathcal{L}(l) \notin \mathcal{D}(H)}{(\textbf{hide}\ H\ c, e, Hide_H\ \psi) \xrightarrow[k]{l} (\textbf{hide}\ H\ c', e', Hide_H\ \psi')} \text{(PS4)}$$

Where $Relax_H$ relaxes the intervals of interactions labelled in $\mathcal{D}(H)$ and $Hide_H$ makes these interactions silent. Formally:

$$\begin{array}{lll} Relax_H\ \psi & = & \{(k, (r, b)) \in \psi \mid b \notin \mathcal{D}(H)\} \\ & \cup & \{(k, ([0, \infty), b)) \mid b \in \mathcal{D}(H) \wedge (\exists r)((k, (r, b)) \in \psi)\} \\ Hide_H\ \psi & = & \{(k, (r, b)) \in \psi \mid b \notin \mathcal{D}(H)\} \\ & \cup & \{(k, (r, \epsilon)) \mid (\exists r)(\exists b \in \mathcal{D}(H))((k, (r, b)) \in \psi)\} \end{array}$$

$$\frac{(c, e, Relax_H\ \psi) \xrightarrow[k]{p\ v_1\ \ldots\ v_n} (c', e', \psi') \qquad p \in \mathcal{D}(H) \qquad 0 \in E \textit{ where } (E, b) = \psi(k)}{(\textbf{hide}\ H\ c, e, Hide_H\ \psi) \xrightarrow[k]{\epsilon} (\textbf{hide}\ H\ c', e', Hide_H\ (Reset_H\ \psi'))} \text{(PS5)}$$

Where $Reset_H\ \psi'$ sets the intervals of the "newly enabled" interactions (those not independant from k) having their label in $\mathcal{D}(H)$. Formally:

$$\begin{array}{rcl} \mathit{Reset}_H\ \psi & = & \{(k', (r, b)) \in \psi \mid k'\ \iota\ k \vee b \notin \mathcal{D}(H)\} \\ & \cup & \{(k', (H\ b, b)) \mid k' \not\iota\ k \wedge b \in \mathcal{D}(H) \wedge (\exists r)((k', (r, b)) \in \psi)\} \end{array}$$

Discrete transitions must be possible without additional delay.

Priorities

$$\frac{(c, e, \psi) \xrightarrow[k]{l} (c', e', \psi') \qquad (\forall l', k', b)((c, e, \psi) \xrightarrow[k']{l'} b \Rightarrow (\mathcal{L}(l'), \mathcal{L}(l)) \notin \Pi)}{(\mathbf{prio}\ \Pi\ c, e, \psi) \xrightarrow[k]{l} (\mathbf{prio}\ \Pi\ c', e', \psi')} \quad \text{(PS6)}$$

Priorities over labels induce priorities over interactions. An interaction may not occur when some other interaction with higher priority is possible. Priorities never relate silent interactions.

Component instance

$$\frac{(c, F\ e_s \cup a, \psi) \xrightarrow[k]{l} (c', F\ e'_s \cup a', \psi')}{(\mathbf{comp}\ \Phi\ F\ (c, F\ e_s \cup a), e_s \cup e, R\ \Phi\ \psi) \xrightarrow[k]{\Phi(l)} (\mathbf{comp}\ \Phi\ F\ (c', F\ e'_s \cup a'), e'_s \cup e, R\ \Phi\ \psi')} \quad \text{(PS7)}$$

Where $R\ \Phi\ \psi = \{(r, \Phi(b)) \mid (r, b) \in \psi\}$

The store of the component makes explicit the entries corresponding to the variables passed to the component instance by reference (e_s) (those passed by value have been captured when determining the initial configuration). Those entries appear in the store of the instance renamed according to function F (see Section 4.3.1).

Process instance

$$\frac{\begin{array}{l}(S, F\ e_s \cup a) \overset{l}{\Rightarrow} (\mathtt{target}\ s', F\ e'_s \cup a') \qquad (\mathbf{from}\ s\ S) \in \mathcal{T} \\ \psi' = \{(k, ([0, \infty), \Phi(k))) \mid k \in \Omega(s', F\ e'_s \cup a')\}\end{array}}{(\mathbf{proc}\ \Phi\ F\ (s, F\ e_s \cup a), e_s \cup e, \psi) \xrightarrow[\mathcal{L}(l)]{\Phi(l)} (\mathbf{proc}\ \Phi\ F\ (s', F\ e'_s \cup a'), e'_s \cup e, \psi')} \quad \text{(PS8)}$$

Where:

- *$\mathcal{T}$ is the set of transitions of the process;*
- *$\Omega(s, e)$ is the set of interactions enabled at configuration (s, e) in the process (see Section 4.3.1);*
- *Stores are handled exactly as in the rule for component instances.*

The interval function at the target configuration associates interval $[0, \infty)$ and label $\Phi(k)$ with each enabled interaction k.

4.4 Semantics of programs

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

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.