The Syntax and Semantics of FIACRE

Bernard Berthomieu*, Jean-Paul Bodeveix+, Mamoun Filali+, Hubert Garavel†,
Frédéric Lang†, Didier Le Botlan*, François Vernadat*, Silvano dal Zilio*

Draft Version 3.0

March 8, 2012

* 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`

Deliverable n^o F.3.2.11 of projet TOPCASED – *Définition du langage pivot asynchrone V3*

1 Introduction

1.1 Fiacre

This document presents the syntax and formal semantics of the Fiacre language, version 3.0. 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.

1.2 From Fiacre V2.0 to Fiacre V3.0

This document presents the current draft specification of version 3.0 of the Fiacre language. This Section summarizes the differences from the previous revision 2.0.

- *Reading and writing shared variables:*

A strong point of the Fiacre language is that it natively supports the two most widely used communication paradigms: by messages or by shared variables. Hence, most of the interaction mechanisms found in practical applications should be easily translated into Fiacre.

But handling these two paradigms in the same language is also the source of subtle semantic issues that, in the first versions of the language were handled by limiting certain capabilities of the language. In Fiacre 2.0, for instance, no transition path could simultaneously perform a synchronization action (pure synchronization or communication) and manipulate a shared variable (read or write).

The practice of Fiacre, in particular in projects aimed at translating AADL into Fiacre for verification purposes, showed that this retriction led to overly complex translation schemes. For these reasons, revision 3.0 takes a more permissive approach: shared variables can be read everywhere in a process or component, and can be written everywhere except in **init** initialization statememts. The strong restriction on the use of shared variables in revision 2.0 is replaced by a simpler independance check of shared variable assignements in interactions. This way, many programs forbidden in revision 2.0 are now legal, while preserving the safety of shared variable assignement.

Allowing shared variable to be read or written everywhere significantly impacts the formal semantics of the language, however, since the possible interactions at some state can now depend on a silent transition. But the benefits are important in terms of usability of shared variables.

- *Time constraints on silent transitions of processes,* **wait** *statements:*

  In revision 2.0, time constraints could only be applied to communication ports. So the only means for timing an internal (silent) transition in a process was to add to it a dummy synchronization action (provided, in 2.0, that this transition did not use shared variables.

  Revision 3.0 introduces a **wait** statement for these purposes: any silent transition in a process may include a **wait** statement that specifies a time constraint for the execution paths it is found into, without the need for any dummy synchronizationm statement. Only one **wait** statement is allowed along any execution path of a transition.

  wait statements do not significantly impact the semantics of the language.

- *Priorities between silent transitions in proceses,* **unless** *clauses in* **select** *statements:*

  As for time constraints, revision 2.0 did not allow to specify priority constraints between silent transitions of a process, revision 3.0 allow this.

  select statements can be now layered. The set of statements constituting a **select** can now be partitionned into groups, separated by the **unless** keyword. The semantics is that a transition in a group is possible only when no transition in some group after the **unless** clause is possible.

  As the previous, this extension does not significantly impact the semantics of the language.

- *Decorrellation of timing constraints between transitions from same source state:*

  In revision 2.0, all timed transitions sourced at some process state were always reset synchronously when one of them was taken. Revision 3.0 keyword **loop** (used in place of **to** s)

allows one to specify the transitions are not reset when taking some transition from state s. This behavior was difficult to simulate in revision 2.0 (without **loop**), while mandatory for some applications.

This extension significantly impact the formal timed semantics of the language as we now have to maintain a clock for every possible transition (and path) from a state while 2.0 only required one clock per state. The timed semantics of Fiacre is revised accordingly (see Section 5).

- *New primitives:*

  A **length** primitive is added for queues, with the obvious meaning.

- *Miscellaneous syntax enhancements:*

  Finally, revision 3.0 implements a few concrete syntax improvements:

  1. *Guarded commands:* Though guarded commands could be expressed in 2.0, a specific notation is introduced for them.

     For any expression e and statement s:

     "**on** $e; s$" is handled like "**case** e **of true**$- > s$ **end**".

  2. **any** can be used as a value emitted in an output statement:

     If m_i is **any** and x a new variable (not occurring yet in the process), then:

     "$p!m_1, \ldots, m_i, \ldots, m_n$" is handled like "$x :=$ **any**; $p!m_1, \ldots, x, \ldots, m_n$"

  3. Finally, revision 3.0 implements some minor concrete syntax changes. In particular array access and field access expressions and patterns are now considered atomic. This removes the need for parenthesis in some contexts. E.g. expressions "**not** $a[1]$" or "$S\ a.l$" are now legal (while revision 2.0 required parentheses around the inner expressions).

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 "$expr^*_{\mathtt{s}}$" (resp. "$expr^+_{\mathtt{s}}$")

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 length loop nat none not null of on or out par port priority process queue read record select states then to true type union unless var wait 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** | **length**

`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
    | atomexp '[' exp ']'
    | atomexp '.' field
    | '(' exp ')'

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

const_decl ::= const var ':' type is exp
```

2.5 Ports, channels, channel declarations

```
port := IDENT

channel_id ::= IDENT

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

channel_decl ::= channel channel_id is channel
```

A port is a process interaction point. Ports can be used for synchronization, communication, or setting timing or priority constraints, determined by the port type, or channel, assigned to 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 ::=` $\texttt{port}^{+}_{,}$ `':'` [**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 ::=` $\texttt{([\&] var)}^{+}_{,}$ `':'` [**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 ::=` $\texttt{var}^{+}_{,}$ `':' 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
    | atompatt '[' exp ']'
```

| atompatt '.' field
| '(' pattern ')'

pattern ::= atompatt | constr atompatt

statement ::=
null
| **on** exp
| 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$^{+}_{[]}$ (**unless** statement$^{+}_{[]}$)* **end** [**select**]
| **case** exp **of** (pattern '->' statement)$^{+}_{|}$ **end** [**case**]
| **to** state
| **loop**
| **wait** time_interval
| statement ';' statement
| port
| port '?' pattern$^{+}_{,}$ [**where** exp]
| port '!' (exp | **any**)$^{+}_{,}$

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 ::= '*' | $\text{port}^{+}_{,}$
```

```
compblock ::= instance | composition
```

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

```
component_decl ::=
    component name
       ['[' $\text{port\_dec}^{+}_{,}$ ']']
       ['(' $\text{arg\_dec}^{+}_{,}$ ')']
    is [var $\text{var\_dec}^{+}_{,}$]
       [port $\text{(port\_dec [in time\_interval])}^{+}_{,}$]
       [priority $(\text{port}^{+}_{|}$ '>' $\text{port}^{+}_{|})^{+}_{,}$]
       [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,* $\texttt{a}_1\texttt{|...|a}_n$ `>` $\texttt{b}_1\texttt{|...|b}_m$ *is a shorthand for* $(\forall i \in \{1,\ldots,n\})(\forall j \in \{1,\ldots,m\})(\texttt{a}_i > \texttt{b}_j)$*.*

2.8 Programs

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

```
program ::=
    $\text{declaration}^{+}$
    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 and **wait** statements, time intervals may not be empty (e.g. intervals like $[7, 3[$ or $]1, 1]$ are rejected);
12. In **priority** declarations, the transitive closure of the priority relation defined must be a strict partial order;
13. In any **priority** declaration $e_1 > e_2$ or $e_2 < e_1$, set e_1 may only contain ports locally declared in the same component;
14. In an assignment statement, the patterns on the left-hand side must be pairwise independent. A sufficient condition for that constraint is explained in Section 3.1.2;
15. Process initialization statements may not include communication, **wait**, **loop** or **unless** statements, and they may not write shared variables. In addition, each of their paths must include a **to** statement;
16. Component initialization statements, may not include communication, **wait**, **to**, **loop** or **unless** statements, and they may not write shared variables;

17. In any process transition, at most one communication, **wait**, or **select** statement with an **unless** clause may be found along each execution path. This constraint, referred to as the *single-communication* constraint is integrated with the well-typing condition for statements, see Section 3.2.5;

18. In any process or component, 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 in this section that the constants declared have been replaced in types and expressions by their values, and that the “size” parameters of queue and array types and the “range” parameters of interval types have been computed (these expressions may only hold constants and literals and so are computable statically).

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 $\{\textbf{in}, \textbf{out}\}$) 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 $\{\textbf{read}, \textbf{write}\}$) 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 (a union type).

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

3.2.2 Subtyping

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

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

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

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

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

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

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:

$$\frac{}{\textbf{none} \le \textbf{none}}\ \text{(SU11)} \qquad \frac{\tau_1 \le \tau_1' \quad \ldots \quad \tau_n \le \tau_n'}{\tau_1\#\ldots\#\tau_n \le \tau_1'\#\ldots\#\tau_n'}\ \text{(SU12)}$$

3.2.3 Typing expressions

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

Subsumption and any expressions

$$\frac{A \vdash E : \tau \qquad \tau \le \tau'}{A \vdash E : \tau'}\ \text{(ET1)} \qquad \frac{}{A \vdash \textbf{any} : \bot}\ \text{(ET2)}$$

any *as an expression is only allowed as the value sent in an output statement.*

Literals

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

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)

$$\frac{A(X) = \{\tau\}}{A \vdash X : \tau}\ \text{(ET5)} \qquad \frac{\{\textbf{read}, \tau\} \subseteq A(X)}{A \vdash X : \tau}\ \text{(ET6)} \qquad \frac{A(C) = \{\tau \to \tau'\} \qquad A \vdash e : \tau}{A \vdash C\ e : \tau'}\ \text{(ET7)}$$

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

Arithmetic and logical primitives

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

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

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

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

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

*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

$$\frac{\{f_1, \ldots, f_n\} \subseteq \texttt{Fields} \qquad A \vdash l_1 : \tau_1 \qquad \ldots \qquad A \vdash l_n : \tau_n}{A \vdash \{f_1 : l_1, \ldots, f_n : l_n\} : \mathbf{record}\ f_1 : \tau_1, \ldots, f_n : \tau_n\ \mathbf{end}} \; \text{(ET15)}$$

$$\frac{A \vdash P : \mathbf{record}\ \ldots, f : \tau, \ldots\ \mathbf{end}}{A \vdash P.f : \tau} \; \text{(ET16)}$$

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

Arrays

$$\frac{A \vdash k_1 : \tau \qquad \ldots \qquad A \vdash k_n : \tau}{A \vdash [k_1, \ldots, k_n] :\ \mathbf{array}\ n\ \mathbf{of}\ \tau} \; \text{(ET17)}$$

$$\frac{A \vdash P : \mathbf{array}\ k\ \mathbf{of}\ \tau \qquad A \vdash E : 0\ ..\ k\text{–}1}{A \vdash P[E] : \tau} \; \text{(ET18)}$$

Queues

$$\frac{A \vdash k_1 : \tau \qquad \ldots \qquad A \vdash k_n : \tau \qquad 0 \leq n \leq m}{A \vdash \{|k_1, \ldots, k_n|\} :\ \mathbf{queue}\ m\ \mathbf{of}\ \tau} \; \text{(ET19)} \qquad \frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau}{A \vdash \mathbf{length}\ q : 0..(k-1)} \; \text{(ET20)}$$

$$\frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau}{A \vdash \mathbf{empty}\ q : \mathbf{bool}} \; \text{(ET21)} \qquad \frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau}{A \vdash \mathbf{full}\ q : \mathbf{bool}} \; \text{(ET22)}$$

$$\frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau}{A \vdash \mathbf{first}\ q : \tau} \; \text{(ET23)} \qquad \frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau}{A \vdash \mathbf{dequeue}\ q : \mathbf{queue}\ k\ \mathbf{of}\ \tau} \; \text{(ET24)}$$

$$\frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau \qquad A \vdash E : \tau}{A \vdash \mathbf{enqueue}\ (q, E) : \mathbf{queue}\ k\ \mathbf{of}\ \tau} \; \text{(ET25)} \qquad \frac{A \vdash q : \mathbf{queue}\ k\ \mathbf{of}\ \tau \qquad A \vdash E : \tau}{A \vdash \mathbf{append}\ (q, E) : \mathbf{queue}\ k\ \mathbf{of}\ \tau} \; \text{(ET26)}$$

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** 16 **of 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:

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

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

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

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

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 that the variables and expressions occurring in the transition are used consistently and that the statement obeys the "single-communication restriction" introduced in Section 3.1.1.

As for expressions, some information is infered for statements, assuming a typing context; the "typing" relation for statements is written "$:_s$".
Statement "types" are subsets of $\{\mathbf{Com}, \mathbf{Wait}, \mathbf{Prio}\}$. A statement has:

Com if it holds some communication or synchronization statement;

Wait if it holds some **wait** statement;

Prio if it holds some **select** statement with **unless** clause(s);

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 $\alpha \subseteq \{\mathbf{Com}, \mathbf{Wait}, \mathbf{Prio}\}$, according to the following rules.

Jumps, null

$$\frac{}{A \vdash \mathbf{to}\ s :_s \emptyset}\ \text{(ST1)} \qquad \frac{}{A \vdash \mathbf{loop} :_s \emptyset}\ \text{(ST2)} \qquad \frac{}{A \vdash \mathbf{null} :_s \emptyset}\ \text{(ST3)}$$

Sequential composition

$$\frac{A \vdash S_1 :_s \{\mathbf{Com}\} \qquad A \vdash S_2 :_s \emptyset}{A \vdash (S_1; S_2) :_s \{\mathbf{Com}\}} \text{ (ST4)} \qquad \frac{A \vdash S_1 :_s \emptyset \qquad A \vdash S_2 :_s \{\mathbf{Com}\}}{A \vdash (S_1; S_2) :_s \{\mathbf{Com}\}} \text{ (ST5)}$$

$$\frac{A \vdash S_1 :_s \alpha \qquad A \vdash S_2 :_s \beta \qquad \alpha, \beta \subseteq \{\mathbf{Wait}, \mathbf{Prio}\} \qquad \alpha \cap \beta = \emptyset}{A \vdash (S_1; S_2) :_s \alpha \cup \beta} \text{ (ST6)}$$

On, Assignments and Case

$$\frac{A \vdash E : \mathbf{bool}}{A \vdash \mathbf{on}\ E :_s \emptyset} \text{ (ST7)}$$

$$\frac{A \vdash P_1 :_p \tau_1 \quad \ldots \quad A \vdash P_n :_p \tau_n \qquad A \vdash E_1 : \tau_1 \quad \ldots \quad A \vdash E_n : \tau_n}{A \vdash P_1, \ldots, P_n := E_1, \ldots, E_n :_s \emptyset} \text{ (ST8)}$$

$$\frac{A \vdash P_1 :_p \tau_1 \quad \ldots \quad A \vdash P_n :_p \tau_n \qquad A \vdash E : \mathbf{bool}}{A \vdash P_1, \ldots, P_n := \mathbf{any\ where}\ E :_s \emptyset} \text{ (ST9)}$$

$$\frac{\begin{array}{c} A \vdash P_1 :_p \tau \quad \ldots \quad A \vdash P_n :_p \tau \qquad A \vdash E : \tau \\ A \vdash S_1 :_s \alpha_1 \quad \ldots \quad A \vdash S_n :_s \alpha_n \end{array}}{A \vdash \mathbf{case}\ E\ \mathbf{of}\ P_1 \rightarrow S_1 \mid \ldots \mid P_n \rightarrow S_n\ \mathbf{end} :_s \alpha_1 \cup \cdots \cup \alpha_n} \text{ (ST10)}$$

Choices and while loop

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S_1 :_s \alpha_1 \qquad A \vdash S_2 :_s \alpha_2}{A \vdash \mathbf{if}\ E\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} :_s \alpha_1 \cup \alpha_2} \text{ (ST11)}$$

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

$$\frac{A \vdash E : \mathbf{bool} \qquad A \vdash S :_s \alpha \qquad \mathbf{Com} \notin \alpha}{A \vdash \mathbf{while}\ E\ \mathbf{do}\ S\ \mathbf{end} :_s \alpha} \text{ (ST12)}$$

$$\frac{A(v) = \{x\ ..\ y\} \qquad A \vdash S :_s \alpha \qquad \mathbf{Com} \notin \alpha}{A \vdash \mathbf{foreach}\ v\ \mathbf{do}\ S\ \mathbf{end} :_s \alpha} \text{ (ST13)}$$

$$\frac{A \vdash S_1 :_s \alpha_1 \quad \ldots \quad A \vdash S_n :_s \alpha_n}{A \vdash \mathbf{select}\ S_1\ [\,]\ ...\ [\,]\ S_n\ \mathbf{end} :_s \alpha_1 \cup \cdots \cup \alpha_n} \text{ (ST14)}$$

$$\frac{A \vdash S_1 :_s \alpha_1 \quad \ldots \quad A \vdash S_n :_s \alpha_n \qquad (\forall i \in \{k+1, \ldots, n\})(\mathbf{Com} \notin \alpha_i)}{A \vdash \mathbf{select}\ S_1\ [\,]\ ...\ [\,]\ S_k\ (\mathbf{unless}\ \ldots)^*\ \mathbf{unless}\ \ldots\ [\,]\ S_n\ \mathbf{end} :_s \alpha_1 \cup \cdots \cup \alpha_n \cup \{\mathbf{Prio}\}} \text{ (ST15)}$$

Communications and wait

$$\frac{\mathbf{none} \in A(p)}{A \vdash p :_s \{\mathbf{Com}\}} \text{ (ST16)} \qquad \frac{}{A \vdash \mathbf{wait}\ \mathit{interval} :_s \{\mathbf{Wait}\}} \text{ (ST17)}$$

$$\frac{A \vdash E_1 : \tau_1 \quad \dots \quad A \vdash E_n : \tau_n \quad \{\mathbf{out}, \tau_1 \# \dots \# \tau_n\} \subseteq A(p)}{A \vdash p \ ! \ E_1, \dots, E_n :_s \{\mathbf{Com}\}} \text{ (ST18)}$$

$$\frac{A \vdash X_1 :_p \tau_1 \quad \dots \quad A \vdash X_n :_p \tau_n \quad A \vdash E : \mathbf{bool} \quad \{\mathbf{in}, \tau_1 \# \dots \# \tau_n\} \subseteq A(p)}{A \vdash p \ ? \ X_1, \dots, X_n \ \mathbf{where} \ E :_s \{\mathbf{Com}\}} \text{ (ST19)}$$

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 $((\dots, \mu_i, \dots), (\dots, \eta_j, \dots))$, 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:

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

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

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

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

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

$\mathbf{par} \dots \ || \ c_i \ || \ \dots \ \mathbf{end}$ *stands for* $\mathbf{par} \dots \ || \ \emptyset \to c_i \ || \ \dots \ \mathbf{end}$
$\mathbf{par} \ Q \ \mathbf{in} \ Q_1 \to c_1 \ || \ \dots \ || \ Q_n \to c_n \ \mathbf{end}$ *stands for*
  $\mathbf{par} \ (Q \cup Q_1) \to c_1 \ || \ \dots \ || \ (Q \cup Q_n) \to c_n \ \mathbf{end}$
If $* \in Q$*, then* $Q \to c$ *is handled like* $\Sigma(c) \to c$.

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

Where $((\mu^1_1, \dots, \mu^1_{n_1}), (\eta^1_1, \dots, \eta^1_{m_1})) \prec ((\mu^2_1, \dots, \mu^2_{n_2}), (\eta^2_1, \dots, \eta^2_{m_2}))$ *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 $x : 0..255$, $y : \mathbf{int}$, $q : \mathbf{queue}\ 5\ of\ \mathbf{nat}$:

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 Operational semantics, part I

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.

For readability, the semantics is broken into two parts. This section introduces the "bahavioral" semantics, the next one will explain the effects of time constraints and priorities.

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 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 **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;
- $length\ q$ is $l - 1$, where l the largest integer in $\mathcal{D}(q)$;
- $first\ q = q(0)$, 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.

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

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:

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

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

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

- 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: $In(v, \textbf{int})$ always holds, $In(v, \textbf{nat})$ 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:

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

  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.

$$\frac{e \vdash q \rightsquigarrow Q}{e \vdash \textbf{length}\ q \rightsquigarrow length\ Q}\ \text{(ES21)}$$

$$\frac{e \vdash q \rightsquigarrow Q}{e \vdash \textbf{empty}\ q \rightsquigarrow empty\ Q}\ \text{(ES22)} \qquad \frac{e \vdash q \rightsquigarrow Q}{e \vdash \textbf{full}_\tau\ q \rightsquigarrow full\ (Cap(\tau))\ Q}\ \text{(ES23)}$$

$$\frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \textbf{first}\ q \rightsquigarrow first\ Q}\ \text{(ES24)} \qquad \frac{e \vdash q \rightsquigarrow Q \qquad \mathcal{D}(Q) \neq \emptyset}{e \vdash \textbf{dequeue}\ q \rightsquigarrow dequeue\ Q}\ \text{(ES25)}$$

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

$$\frac{e \vdash q \rightsquigarrow Q \qquad e \vdash x \rightsquigarrow v \qquad Cap(\tau) - 1 \notin \mathcal{D}(Q)}{e \vdash \textbf{append}_\tau\ (q, x) \rightsquigarrow append\ Q\ v}\ \text{(ES27)}$$

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

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

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

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

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

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

Where C is a 1-ary constructor.

4.2 Semantics of Processes

4.2.1 Semantics of statements

The semantics of statements is expressed operationally 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;
- $S' \in \{\texttt{done}\} \cup \{\texttt{self}\} \cup \{\texttt{target}\ s | s \in \Lambda\}$, 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 $p\ v_1 \ldots v_n$, in which p is a port and $v_1 \ldots v_n$ $(n \geq 0)$ are values.

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

To, loop, null, wait

$$\frac{}{(\mathbf{to}\ s, e) \overset{\epsilon}{\Rightarrow} (\texttt{target}\ s, e)}\ \text{(SS1)} \qquad \frac{}{(\mathbf{loop}, e) \overset{\epsilon}{\Rightarrow} (\texttt{self}, e)}\ \text{(SS2)}$$

Behaviorally, **loop** *is equivalent to* **to** `s`*, in which* `s` *is the state originating the transition in which* **loop** *is found (this will be made clear in section 4.3.3). As will be seen in Section 5, statements* **to** *s and* **loop** *only differ by their timed semantics.*

$$\frac{}{(\mathbf{null}, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)}\ \text{(SS3)} \qquad \frac{}{(\mathbf{wait}\ i, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)}\ \text{(SS4)}$$

null *is the empty statement.* **wait** *specifies a timed constraint explained in Section 5.*

Deterministic assignment, on

$$\frac{\begin{array}{llll} e \vdash E_1 \rightsquigarrow v_1 & e \vdash E_2 \rightsquigarrow v_2 & \ldots & e \vdash E_n \rightsquigarrow v_n \\ e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \ldots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ \mathcal{M}(v_1, P_1) & \mathcal{M}(v_2, P_2) & \ldots & \mathcal{M}(v_n, P_n) \\ e' = a_n(v_n) & & & \end{array}}{(P_1, P_2, \ldots, P_n := E_1, E_2, \ldots, E_n, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e')} \quad \text{(SS5)}$$

The statement **on** *exp is handled like* **true** := *exp*

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

$$\frac{\begin{array}{llll} e \vdash P_1 \rightsquigarrow^l e_1, a_1 & a_1(v_1) \vdash P_2 \rightsquigarrow^l e_2, a_2 & \ldots & a_{n-1}(v_{n-1}) \vdash P_n \rightsquigarrow^l e_n, a_n \\ e' = a_n(v_n) & [e' \vdash E \rightsquigarrow \mathit{true}] & & \end{array}}{(P_1, P_2, \ldots, P_n := \textbf{any} \ [\textbf{where}\ E], e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e')} \quad \text{(SS6)}$$

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

While, foreach

$$\frac{e \vdash E \rightsquigarrow \mathit{true} \qquad (S; \textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(SS7)}$$

$$\frac{e \vdash E \rightsquigarrow \mathit{false}}{(\textbf{while}\ E\ \textbf{do}\ S\ \textbf{end}, e) \overset{\epsilon}{\Rightarrow} (\texttt{done}, e)} \quad \text{(SS8)}$$

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

$$\frac{(V := v_1\ ;\ S\ ; \ldots V := v_n\ ;\ S, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{foreach}\ V\ \textbf{do}\ S\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')} \quad \text{(SS9)}$$

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

Deterministic choice

$$\frac{e \vdash E \rightsquigarrow \mathit{true} \qquad (S_1, e) \overset{l}{\Rightarrow} (S, e')}{(\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}, e) \overset{l}{\Rightarrow} (S, e')} \quad \text{(SS10)} \qquad \frac{e, E \rightsquigarrow \mathit{false} \qquad (S_2, e) \overset{l}{\Rightarrow} (S, e')}{(\textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end}, e) \overset{l}{\Rightarrow} (S, e')} \quad \text{(SS11)}$$

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

Case

$$\frac{e \vdash E \leadsto v \qquad \mathcal{M}(v, P_1) \qquad (P_1 := E\ ;\ S_1, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{case}\ E\ \textbf{of}\ P_1 \rightarrow S_1 \mid ... \mid P_n \rightarrow S_n\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}\ \text{(SS12)}$$

$$\frac{e \vdash E \leadsto v \qquad \neg\mathcal{M}(v, P_1) \qquad (\textbf{case}\ E\ \textbf{of}\ P_2 \rightarrow S_2 \mid ... \mid P_n \rightarrow S_n\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{case}\ E\ \textbf{of}\ P_1 \rightarrow S_1 \mid ... \mid P_n \rightarrow S_n\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}\ \text{(SS13)}$$

Nondeterministic choice

$$\frac{(S_i, e) \overset{l}{\Rightarrow} (S', e')}{(\textbf{select}\ S_1\ [\ ]\ ...\ [\ ]\ S_n\ \textbf{end}, e) \overset{l}{\Rightarrow} (S', e')}\ \text{(SS14)}$$

Behaviorally, occurrences of **unless** *can be replaced by "[ ]".* **unless** *clauses introduce priorities among the alternants of a* **select** *statement, explained in Section 5.*

Sequential composition

$$\frac{(S_1, e) \overset{l}{\Rightarrow} (\texttt{target}\ s, e')}{(S_1; S_2, e) \overset{l}{\Rightarrow} (\texttt{target}\ s, e')}\ \text{(SS15)} \qquad \frac{(S_1, e) \overset{l}{\Rightarrow} (\texttt{self}, e')}{(S_1; S_2, e) \overset{l}{\Rightarrow} (\texttt{self}, e')}\ \text{(SS16)}$$

$$\frac{(S_1, e) \overset{l_1}{\Rightarrow} (\texttt{done}, e') \qquad (S_2, e') \overset{l_2}{\Rightarrow} (S', e'')}{(S_1; S_2, e) \xRightarrow{l_1.l_2} (S', e'')}\ \text{(SS17)}$$

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

Communication

$$\frac{}{(p_\tau, e) \xRightarrow{p} (\texttt{done}, e)}\ \text{(SS18)}$$

$$\frac{e \vdash E_1 \leadsto v_1 \qquad \dots \qquad e \vdash E_n \leadsto v_n}{(p!E_1, \dots, E_n, e) \xRightarrow{p\ v1\ ...\ vn} (\texttt{done}, e)}\ \text{(SS19)}$$

If some E_i is **any** *then v_i is any value of the type of E_i*

$$\frac{\begin{array}{llll} e \vdash P_1 \leadsto^l e_1, a_1 & a_1(v_1) \vdash P_2 \leadsto^l e_2, a_2 & \dots & a_{n-1}(v_{n-1}) \vdash P_n \leadsto^l e_n, a_n \\ e' = a_n(v_n) & [e' \vdash E \leadsto \mathit{true}] & & \end{array}}{(p?P_1, P_2, .., P_n\ [\textbf{where}\ E], e) \xRightarrow{p\ v_1\ ...\ v_n} (\texttt{done}, e')}\ \text{(SS20)}$$

v_i ranges over all values of the type of P_i.

4.2.2 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. Otherwise, from the static restrictions put on **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 an 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. We focus in this section on the discrete transitions of the semantics; Time elapsing transitions and the effects of priorities will be explained in Section 5.

The semantics of a component is obtained compositionally from the semantics of the process instances and component instances it captures and that of the composition operator.

4.3.1 Abstract components:

Consider the following grammar of abstract components:

$$
\begin{array}{lcll}
c & ::= & \textbf{hide}\ H\ c & \text{hiding} \\
 & \mid & \textbf{priority}\ \Pi\ c & \text{priority} \\
 & \mid & c' & \text{composition} \\
c' & ::= & g_1 \rightarrow c'_1 \mid \ldots \mid g_2 \rightarrow c'_2 & \text{composition} \\
 & \mid & \textbf{comp}\ (c, a) & \text{component instance} \\
 & \mid & \textbf{proc}\ (s, a) & \text{process instance}
\end{array}
$$

Any Fiacre component can be represented by an abstract term of form **hide** H (**priority** Π c), where c is some term involving only compositions and instances, set H is the set of ports declared locally in the component and Π is a relation on port (the transitive closure of the relation specified by the Fiacre **priority** declaration).

Compositions and components are assumed "normalized": stars and factorized port sets in **par** compositions are eliminated as explained in Section 3.2.6; proces states, bound ports and bound variables in each process and component instances are assumed renamed so that no two process of component instances share some port or variable name. The leaves of compositions are component instances **comp** (c, a) in which (c, a) is an abstract component and a store, or process instances **proc** (s, a), in which (s, a) is a process configuration.

4.3.2 Component configurations

Component configurations are pairs (c, e) in which c is an abstract component, as defined in Section 4.3.1 and e is a store.

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** (s, a) capture an initial configuration of the process and abstract component instances **comp** (c, a) capture an initial abstract state and initial store of the subcomponent.

4.3.3 Semantic rules for components

The labelled semantics relation, linking component configurations, is written $\xrightarrow{l}$, specifying an action.

For any action l, let us define $\mathcal{L}(l)$ by $\mathcal{L}(\epsilon) = \epsilon$, $\mathcal{L}(p\ v_1\ \dots\ v_n) = p$.

Process instance

$$\frac{(S, e_s \cup a) \overset{l}{\Rightarrow} (\texttt{target}\ s', e'_s \cup a') \qquad (\textbf{from}\ s\ S) \in \mathcal{T}}{(\textbf{proc}\ (s, e_s \cup a), e_s \cup e) \xrightarrow{l} (\textbf{proc}\ (s', e'_s \cup a'), e'_s \cup e)}\ \text{(PS1)}$$

$$\frac{(S, e_s \cup a) \overset{l}{\Rightarrow} (\texttt{self}, e'_s \cup a') \qquad (\textbf{from}\ s\ S) \in \mathcal{T}}{(\textbf{proc}\ (s, e_s \cup a), e_s \cup e) \xrightarrow{l} (\textbf{proc}\ (s, e'_s \cup a'), e'_s \cup e)}\ \text{(PS2)}$$

Where $\mathcal{T}$ is the set of transitions of the process.

The store of a process instance includes the local variables (a) and those shared by the process instance (e_s).

self *in the second rule is handled exactly as* **target** *s.*

Component instance

$$\frac{(c, e_s \cup a) \xrightarrow{l} (c', e'_s \cup a')}{(\textbf{comp}\ (c, e_s \cup a), e_s \cup e) \xrightarrow{l} (\textbf{comp}\ (c', e'_s \cup a'), e'_s \cup e)}\ \text{(PS3)}$$

Stores are handled exactly as for process instances.

Hiding

$$\frac{(c, e) \xrightarrow{l} (c', e') \qquad \mathcal{L}(l) \notin H}{(\textbf{hide}\ H\ c, e) \xrightarrow{l} (\textbf{hide}\ H\ c', e')}\ \text{(PS4)} \qquad \frac{(c, e) \xrightarrow{l} (c', e') \qquad \mathcal{L}(l) \in H}{(\textbf{hide}\ H\ c, e) \xrightarrow{\epsilon} (\textbf{hide}\ H\ c', e')}\ \text{(PS5)}$$

Compositions

For any abstract composition $C = g_1 \rightarrow c_1 \mid \ldots \mid g_n \rightarrow c_n$ with n components and any $A = \{a_1, \ldots, a_m\} \subseteq \{1, \ldots, n\}$, let $C[g_{a_1} \rightarrow c_{a_1}, \ldots, g_{a_m} \rightarrow c_{a_m}]_A$ denote the result of replacing the components of C indexed over A by those between brackets.

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

$$\frac{(c, e) \xrightarrow{l} (c', e') \qquad \mathcal{L}(l) \notin g}{(C[g \rightarrow c]_{\{i\}}, e) \xrightarrow{l} (C[g \rightarrow c']_{\{i\}}, e')} \text{ (PS6)}$$

$$\frac{(c_{a_1}, e) \xrightarrow{p\ v_1\ \ldots\ v_k} (c'_{a_1}, e^1) \quad \ldots \quad (c_{a_m}, e^{m-1}) \xrightarrow{p\ v_1\ \ldots\ v_k} (c'_{a_m}, e') \qquad (\forall i)(p \in g_i \Rightarrow i \in A)}{(C[g_{a_1} \rightarrow c_{a_n}, \ldots, g_{a_m} \rightarrow c_{a_m}]_A, e) \xrightarrow{p\ v_1\ \ldots\ v_k} (C[g_{a_1} \rightarrow c'_{a_1}, \ldots, g_{a_m} \rightarrow c'_{a_m}]_A, e')} \text{ (PS7)}$$

The store may change as the result of any subcomponent move. It is assumed that the effect on the store of all subcomponent moves are independent; implementations of Fiacre should only accept specifications ensuring that property.

Priorities

The semantics of priorities will be explained in Section 5.

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.

5 Operational semantics, part II

This section completes the semantics of Fiacre started in Section 4. It focuses on the effects of timing and priority constraints on the semantics of processes and components (the semantics of expressions is unaffected).

Time constraints appear as time interval annotations in **port** declarations in components or as **wait** or **loop** statements in processes. Time constraints introduce time-elapsing transitions and possibly restrict the set of possible discrete transitions from a configuration.

Priority constraints appear as explicit constraints on ports in **priority** declarations in components or as **unless** clauses in **select** statements in processes. Priority constraints possibly restrict the set of possible discrete transitions from a configuration.

5.1 Semantics of Processes

In Section 4.2, process statements were given a semantics in terms of relation $(s, e) \overset{l}{\Rightarrow} (s', e')$. It may happen that statement s has several execution paths. The timed semantics requires to refine this relation so that paths are made explicit.

The set of paths of a particular statement is finite. We will assume that paths are represented by elements $p_1, p_2, \ldots$ of some set P. Relation $(s, e) \overset{l}{\Rightarrow} (s', e')$ is extended so that the path taken in statement s for that particular derivation is made explicit. The updated relation is written as follows, in which p is a path:

$$(s, e) \underset{p}{\overset{l}{\Rightarrow}} (s', e')$$

5.2 Semantics of Components

5.2.1 Interactions

Elementary actions of a process instance: Assuming each statement derivation is associated with some path p, as in $(s, e) \underset{p}{\overset{l}{\Rightarrow}} (s', e')$, each process instance derivation will be associated with a triple (s, p, l) in which s is a process state, p a transition path and l an action. We will call such triples *elementary actions*.

We only need to consider elementary actions in which the path leads to a statement **target** s' or **self**, and statement s is attached with some fiacre transition (**from** $s \in \mathcal{T}$). In addition, since states in component instances are assumed uniquely renamed, a triple (s, p, l) identifies a single elementary action among those of all process instances involved in a component.

Interactions of a component instance: Component instances typically involve several process or component instances. Their actions, called interactions, may involve several actions from their constituents.

Assuming a component makes use of n subcomponents, its interactions are the tuples $(\iota_1, \ldots, \iota_n)$ in which each ι_i is an elementary action (if subcomponent i is a process instance), an interaction (if subcomponent i is a component instance) or the particular interaction $\bullet$, meaning that subcomponent i is not involved in that interaction.

By abuse of language, elementary actions will also be called interactions, even though they involve a single component.

5.2.2 Clocks, clock assignements, persistent interactions

With each interaction of a component will be bijectively associated a *clock*.

We will say that an elementary action $k = (s, p, l)$ is *enabled* when there exists some transition $(s, e) \overset{l}{\underset{k}{\Rightarrow}} (s', e')$ and that an interaction is *enabled* if all its constituent interactions different from $\bullet$ are enabled (in their respective subcomponents).

We will say that an an elementary action is *persistent* when its path p leads to a **self** statement (rather than a **target** statement), and that an interaction $(\iota_1, \ldots, \iota_n)$ is *persistent* if all its constituents different from $\bullet$ are persistent.

Finally, with each interaction k will be associated a time interval $I_s(k)$, called its *static firing interval*. If k is an elementary action and its path contains a wait statement **wait** ti, then $I_s(k) = ti$. If k is an interaction labelled p and port p was assigned a time interval ti, then $I_s(k) = ti$, otherwise $I_s(k)$ is the trivial interval $[0, \infty[$.

Given a time interval i, $\uparrow i$ denotes its right end-point, or ∞ if the interval is unbounded.

5.2.3 Component configurations

Compared to Section 4.3, component configurations are enriched with a new element. Given a set K of interactions (including its own interactions), a component configuration is a triple (s, e, ϕ) in which:

- c is an abstract component (cf. Section 4.3.1);
- e is a store;
- $\phi : K \rightarrow \mathbb{R}^+$ is a *clock assignement* to interactions. For each interaction k, $\phi(k)$ holds the time elapsed since k was last enabled.

The semantics of a component is obtained as the union of two relations linking configurations: The discrete transition relation $\overset{l}{\underset{k}{\rightarrow}}$ (k is an *interaction*), and the continuous time-elapsing relation $\xrightarrow{\theta}$ ($\theta \in \mathbb{R}^+$). We now describe these relations.

Rajouter initial configurations ...

5.2.4 Time-elapsing transitions

Time-elapsing transitions on component configurations are defined as follows:

$$\frac{(\forall k \in K)(k \ enabled \Rightarrow \phi(k) + \theta \leq \uparrow(I_s(k))}{(c, e, \psi) \xrightarrow{\theta} (c', e, \psi \dot{+} \theta)} \ \text{(TPS1)}$$

where function $\dot{+}$ *is defined by:* $(\forall k \in K)((\phi \dot{+} \theta)(k) = \phi(k) + \theta)$.

The rule says that time can elapse as long as no enabled interaction overflows its deadline: $\phi(k)$ captures the time elapsed since interaction k was last enabled; enabled interactions may not be delayed longer than the longest delay in their assigned static time interval I_s.

Time elapsing only affects the clock assignement components of configurations; the clocks of all interactions are increased by θ.

5.2.5 Discrete transitions

Discrete transitions can affect all configuration components except the priority relation on interactions (left in configurations for completness). They are defined by the following rules:

Process instance

$$\frac{\begin{array}{l}(S, e_s \cup a) \underset{p}{\overset{l}{\Rightarrow}} (\texttt{target}\ s', e'_s \cup a') \qquad (\textbf{from}\ s\ S) \in \mathcal{T} \\ \phi(k) \in I_s(k) \\ (\forall k' \neq k \in K)(\neg(k \prec^* k'))\end{array}}{(\textbf{proc}\ (s, e_s \cup a), e_s \cup e, \phi) \xrightarrow[k=(s,p,l)]{l} (\textbf{proc}\ (s', e'_s \cup a'), e'_s \cup e, \phi')} \ \text{(TPS2)}$$

$$\frac{\begin{array}{l}(S, e_s \cup a) \underset{p}{\overset{l}{\Rightarrow}} (\texttt{self}, e'_s \cup a') \qquad (\textbf{from}\ s\ S) \in \mathcal{T} \\ \phi(k) \in I_s(k) \\ (\forall k' \neq k \in K)(\neg(k \prec^* k'))\end{array}}{(\textbf{proc}\ (s, e_s \cup a), e_s \cup e, \phi) \xrightarrow[k=(s,p,l)]{l} (\textbf{proc}\ (s, e'_s \cup a'), e'_s \cup e, \phi')} \ \text{(TPS3)}$$

Precondition on $\phi(k)$ means that elementary action k can be performed without delay.

In both rules, ϕ' is obtained as follows, for any elementary action k:

- *If k is persistent, then $\phi'(k) = \phi(k)$;*
- *Otherwise $\phi(k) = 0$ (the clock associated with k is "reset").*

The last precondition means that no elementary action with higher priority than k can be performed. $\prec^*$ is the transitive closure of the priority relation $\prec$ on the elementary actions of the process, defined as follows:

If $k = (s, p, l)$ and $k' = (s', p', l')$ are elementary actions, then we have $k \prec k'$ if and only if $s = s'$, and for some statements x along path p and x' along path p', x and x' occurs in different groups of some **select** *statement of the fiacre process transition and x' occurs after x in the* **select** *statement (the groups of a* **select** *statement are the sets of statements separated by* **unless***).*

Component instance

$$\frac{(c, e_s \cup a, \phi) \xrightarrow[k]{l} (c', e'_s \cup a', \phi')}{(\textbf{comp}\ (c, e_s \cup a), e_s \cup e, \phi) \xrightarrow[k]{l} (\textbf{comp}\ (c', e'_s \cup a'), e'_s \cup e, \phi')} \ \text{(TPS4)}$$

Hiding

$$\frac{(c, e, \phi) \xrightarrow[k]{l} (c', e', \phi') \qquad \mathcal{L}(l) \notin H}{(\textbf{hide}\ H\ c, e, \phi) \xrightarrow[k]{l} (\textbf{hide}\ H\ c', e', \phi')} \ \text{(TPS5)} \qquad \frac{(c, e, \phi) \xrightarrow[k]{l} (c', e', \phi') \qquad \mathcal{L}(l) \in H}{(\textbf{hide}\ H\ c, e, \phi) \xrightarrow[k]{\epsilon} (\textbf{hide}\ H\ c', e', \phi')} \ \text{(TPS6)}$$

Priorities

$$\frac{(c, e, \phi) \xrightarrow[k]{l} (c', e', \phi') \qquad (\forall l', k', b)((c, e, \phi) \xrightarrow[k']{l'} b \Rightarrow (\mathcal{L}(l'), \mathcal{L}(l)) \notin \Pi)}{(\textbf{prio}\ \Pi\ c, e, \phi) \xrightarrow[k]{l} (\textbf{prio}\ \Pi\ c', e', \phi')} \text{(TPS7)}$$

Priorities over labels induce priorities over interactions. An interaction may not occur when some other interaction with higher priority is possible.

Compositions

Using the notations of Section 4, the semantics of compositions is obtained as follows:

$$\frac{(c_i, e, \phi/i) \xrightarrow[k_i]{l} (c'_i, e', \phi'/i) \qquad \mathcal{L}(l) \notin g}{(C[g_i \to c_i]_{\{i\}}, e, \phi) \xrightarrow[k]{l} (C[g_i \to c'_i]_{\{i\}}, e', \phi')} \text{(TPS8)}$$

$$\frac{\begin{array}{l}(c_{a_1}, e, \phi/a_1) \xrightarrow[k_{a_1}]{p\ v_1\ \dots\ v_k} (c'_{a_1}, e^1, \phi'/a_1) \quad \dots \quad (c_{a_m}, e^{m-1}, \phi_{a_m}) \xrightarrow[k_{a_m}]{p\ v_1\ \dots\ v_k} (c'_{a_m}, e', \phi'_{a_m}) \\ (\forall i)(p \in g_i \Rightarrow i \in A)\end{array}}{(C[g_{a_1} \to c_{a_n}, \dots, g_{a_m} \to c_{a_m}]_A, e, \phi) \xrightarrow[k]{p\ v_1\ \dots\ v_k} (C[g_{a_1} \to c'_{a_1}, \dots, g_{a_m} \to c'_{a_m}]_A, e', \phi')} \text{(TPS9)}$$

where:

- *For each i:*
  - *k_i is the i^{th} component of k;*
  - *The domain of ϕ_i is the set of interactions of c_i;*
  - *ϕ_i is "consistent" with ϕ in the sense that:*
    - *$\phi_i(k_i) = \phi(k)$;*
    - *for each j in domain of ϕ_i there exists an interaction z in domain of ϕ containing j and such that $\phi_i(j) = \phi(z)$;*
- *$\phi' = \phi$ except that "newly enabled" interactions at destination have their clock reset (cf. the process instance rules).*

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.