A FIACRE V3.0 Primer

Bernard Berthomieu*, Silvano dal Zilio*, François Vernadat*

March 4, 2020

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

1 Introduction

FIACRE [1] stands for *Format Intermédiaire pour les Architectures de Composants Répartis Embarqués* (Intermediate Format for the Architectures of Embedded Distributed Components). It 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.

This document complements the formal description of FIACRE in [2] and subsequent revisions. It introduces the features of the language from a user point of view though a number of examples. For technical details, the reader is referred to the formal description.

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.

The next sections describe the different layers of FIACRE: types and expressions in Section 2, functions in Section 3, processes in Section 4 and components in Section 5.

2 Types and expressions

2.1 Boolean expressions

FIACRE supports a boolean type **bool**, with:

- constants **true** and **false**;
- primitive operators: **not**, **or**, **and**, =>;
- equality = and unequality <> functions;
- conditional expressions _ : _?_.

Operators **or**, **and** and => are evaluated eargerly: their result is computed from the result of evaluation of their arguments:

```
a or b
a and b
```

In contrast, the arguments of conditional expressions are evaluated on demand. Assuming expressions a and b are defined, the above expressions are equivalent to:

```
a:true?b
a:b?false
```

All FIACRE types supports equality, defined structurally as equality of their contents. Both arguments of the equality and unequality functions must have same type.

2.2 Numeric expressions

The numeric types of FIACRE include:

- the type **int** of all integers;
- the type **nat** of nonnegative integers;
- for each closed integer interval $[from, to]$, the type $from..to$ of all integers in the interval.

Numeric constants are overloaded at all numeric types including them. E.g. constant 4 has types **int**, **nat**, and all interval types $from..to$ such that $from \leq 4 \leq to$.

All numeric types support the usual arithmetic operators: negation $-$, addition $+$, subtraction $-$, multiplication $*$, division $/$, modulo $\%$, with their standard meanings, and a unary coercion operator written \$. All numeric primitives (except the coercion operator \$) are "homogeneous": their argument(s) and result have the same (numeric) type.

Their behavior may depend on the type choosen, however: not all arguments may be legal when a primitive is used at some particular types. For instance, the substraction primitive over **int** values is total, but it is only partial over **nat** values; in this case, it has type $\mathbf{nat} * \mathbf{nat} \rightarrow \mathbf{nat}$, implying that it admits any values of type **nat** as arguments, but it is undefined if its first argument is smaller than the second. In practice, invalid argument applications will fail dynamically (an exception will be raised and an error message printed).

The unary coercion operator is overloaded at all types $ty \rightarrow ty'$ where ty and ty'. It is defined as the identity function when its argument belongs to its result type, otherwise it raises an exception.

2.3 Naming types, Type abbreviations

Types can be given names using **type** declarations, as follows:

```
type byte is 0..7
```

If expressions a and b have equal types, then they can be used in the same contexts. The meaning of FIACRE types is based on their structure rather than on their names: type names are abbreviations standing for the type they are bound to; two types are equal if they are identical after recursively replacing the abbreviations they refer to their bound types.

2.4 Naming values, Constant declarations

Values can be declared at toplevel using **const** declarations. The declared identifiers must be given a type and a value:

```
const length : nat is 4
const width : nat is 7
const area : nat is length * width
```

Constants expressions can also appear in types, everywhere expressions are expected:

```
const count : nat is 8
type site is 0..count>0:count-1?0
```

2.5 Records

Records, or tagged-products, encapsulate several values of possibly different types into a single value, each constituting a "field" of the record.

```
type item is record weight : int, height : nat, length : nat end
```

Records can be built using record expressions:

```
const i1 : item is {weight=3, height=12, length=5}
```

And their components extracted using the dot notation:

```
const h : nat is item.height;
```

Naming record types is not mandatory. Equality on records is defined structurally: two records are equal if they have the same fields and, at each field, the values encapsulated are equal. The order in which fields occur in records is irrelevant.

2.6 Enumerations and Unions

Enumerations and unions are provided by a single tagged-union type **union**:

The first example defines a type by enumerating its elements, which all are constants:

```
type color is union red | green | blue end
```

The next example declares three constructors encapsulating respectively an integer, a nonnegative integer or a byte. Only a single value can be encapsulated by a union constructor, but that value can be of any type (including records).

```
type number is union INT of int | NAT of nat | BYTE of byte end;
const n : number is NAT(4)
```

Of course, a **union** type may declare both constant values and constructors in the same declaration, as in the following "option" type:

```
type option is union None | Some of int end;
```

The intended application domain of FIACRE (real time systems) precludes recursive data structures like lists of trees, hence union types cannot be defined recursively. Unions support equality, with the obvious meaning. There are no expressions extracting the contents of a union construction but a specific **case** statement is provided for this, described in Section 4.9.

2.7 Arrays

Arrays encapsulate a given number of components of same type into a single value; the components can be of any type. For instance, integer vectors and square matrices of size 4 can be defined as follows:

```
type vector is array 4 of int
type matrix is array 4 of vector
```

Arrays can be created from array expressions:

```
const v1 : vector is [1,0,0,0]
const m : matrix is [v1,[0,1,0,0],[0,0,1,0],[0,0,0,1]]
```

And their components accessed using the index notation:

```
const m23 : nat is m[2][3];
```

The indices of an array declared of size n have type `0..n-1`. Two arrays are equal if they have the same size and, at each index, they hold equal components.

2.8 Queues

Bounded queues can be represented by arrays but as they occur frequently in the application domain of Fiacre, a primitive **queue** type is provided. The **queue** type allows one to implement a number of "dynamic" data structures of bounded size like bounded stacks or lists of bounded length.

```
type fifo is queue 8 of number
```

Queues can be created from queue expressions. E.g. the following declarations create an empty queue `q0` and a queue `q2` holding two numbers, respectively.

```
const q0 : fifo is {||}
const q2 : fifo is {|INT(4),NAT(2)|}
```

A number of primitives operate on queues:

- **empty** `q` (resp. **full** `q`) returns true if queue `q` is empty (resp. full);
- **enqueue**`(q,e)` (resp. **append**`(q,e)`) return a queue equal to `q` with element `e` added at the back (resp. at the front);
- **first**`(q)` returns the front element of `q`, **dequeue**`(q)` returns a queue equal to `q` without its front element.

Note that all these primitive are functional: all leave their argument(s) unchanged and return new queues or components. These functions are partially defined: **first** or **dequeue** cannot be applied to an empty queue, nor **append** or **enqueue** to a full queue.

```
const q3 : fifo is enqueue(enqueue(enqueue({||},BYTE(4)),NAT(5)),INT(-2))
const f : number is empty(q3) : INT(0) ? first(q3)
```

Two queues are equal if they have the same size and they hold equal components at the same positions.

2.9 Typing of expressions, subtyping

Since numeric expressions can have several types, so is the case of Fiacre expressions, in general. For instance, if expressions `e` and `f` are numeric and have all types in set E and F, respectively, then the expression {`fst=e, snd=f`} has all types **record** `fst:ty1, snd:ty2` **end**, in which `(ty1,ty2)` $\in E \times F$.

Numeric types are organized into a subtyping relation. Intuitively, $ty \leq ty'$ means that type ty' contains all elements of type ty. That relation on numeric types is extended to a relation on Fiacre types in the natural covariant way (the formal treatment is found in [2]). Note that Fiacre only admits "depth subtyping"; record types with different sets of fields, or arrays of different sizes, are unrelated by subtyping ("width subtyping" is not supported).

Due to subtyping, Fiacre functions in isolation may be used at several types, in general. But since their behavior may differ on the type chosen, each occurrence of a primitive in a Fiacre program will be assigned a single type: The largest type permitted by the surrounding context.

3 Functions

Fiacre V3 supports functions, either native (defined in Fiacre) or extern (defined in C with their profile declared in Fiacre).

3.1 Native functions

Functions, whether extern or native, are evaluated applicatively.

As an example, here is the definition of a `reorder` function operating on queues of messages. The function reorders the content of the queue so as urgent messages appear first, urgent messages being those packed with constructor `p2`.

```
type msg is p1 | p2 of int | p3 of 0..4 end

function urgent (m : msg) : bool is
   begin
       case m of
         p2 any -> return true
       | any -> return false
       end
   end

type mbuff is queue 7 of msg

function reorder (q: mbuff) : mbuff is
   var u: mbuff := ||, n: mbuff := ||, h: msg
   begin
```

```
      while not (empty q) do
          h := first q;
          q := dequeue q;
          if urgent h then
              u := enqueue (u,h)
          else
              n := enqueue (n,h)
          end
      end;
      while not (empty n) do
          u := enqueue (u,first n);
          n := dequeue n
      end;
      return u
  end
```

The header specifies a type for each parameter, and a type for the result. Fiacre functions do not allow side-effects (no shared variables as arguments). If the body resumes to a **return** statement then the enclosing **begin** and **end** may be omitted.

Function bodies make use of standard statements (if-then-else, while-do, sequence, assignements), and a **case** statement for extracting the contents of union values. Control must reach a return statement. Conversely to processes (in the next section), no function statement is blocking (e.g. a case statement (see Section 4.9) in which no match is possible makes the function call fail with a `Match` error).

Fiacre functions may be recursive. The following is a recursive variant of the above function. It takes an extra argument accumulating its result. The calls `rec_reorder(q,{||})` and `reorder(q)` return the same queue.

```
function rec_reorder (q: mbuff, n: mbuff) : mbuff is
  begin
      if empty q then
          return n
      elsif urgent (first q) then
          return append (rec_reorder (dequeue q, n), first q)
      else
          return rec_reorder (dequeue q, enqueue (n,first q))
      end
  end
```

3.2 Extern functions

Finally, functions may be defined externally, in language C if using the frac compiler, rather than in Fiacre. This solution should be reserved to functions that cannot be efficiently defined in Fiacre, for instance because Fiacre lacks a primitive essential for that function (e.g. power).

For the sake of illustration, this is how the above reorder function would appear in Fiacre and would be implemented in C.

A profile for the function would be declared in Fiacre, as follows. The declaration also associates a C function (`c_reorder`) with the Fiacre function (`reorder`):

```
extern reorder (mbuff) :  mbuff is c_reorder
```

Assuming the above function appears in a Fiacre specification named app.fcr, compiling app.fcr with frac would build a app.tts folder including app.net, app.c and app.h. app.h is a header file associating Fiacre types with their C implementations. The C profile to be used for function `c_reorder` can be extracted fom file app.c in which it is deflared extern. The implementation of function `c_reorder` must appear in a C file compiled together with app.c, and including file app.h:

```
#include "zc.h"

struct q1 c_reorder (struct q1 q) {
   struct q1 r;
   int i, j = 0;
   // copy urgent elements from qa into ra
   for (i=0; i<q.len; i++) {
       if (q.at[i].con == 1) {
           r.at[j++] = q.at[i];
       }
   }
   // append non urgent elements of qa to ra
   for (i=0; i<q.len; i++) {
       if if (q.at[i].con) {
           r.at[j++] = q.at[i];
       }
   }
   r.len = q.len;
   return r;
}
```

4 Processes

4.1 Contents of a process

Processes describe sequential behaviors. A process is defined by a set of control states and a set of process transitions each expressing a state change by a statement built from deterministic constructs (assignments, conditionals, loops, and sequential composition), nondeterministic constructs (nondeterministic choice and nondeterministic assignments), interaction statements and jump statements. In addition, processes can be parameterized by values, value locations (shared variable addresses) and communication ports (Interaction labels).

A distinction must be clearly made between the transitions of a process and those of its underlying transition system (its behavior, as expressed by the semantics of FIACRE). A single process transition may correspond in general with several behavior transitions, according to the execution path taken in the process transition.

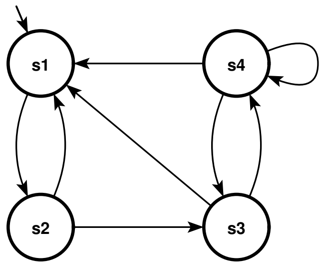
Another important notion related to transition is atomicity: a process transition leads to a behavior transition only if some execution path of that transition can be taken that holds a jump (**to**) statement. Behavior transitions are performed totally or not started at all. If some condition along some execution path of a process transition is not fullfilled, then that path does not yield a behavior transition (we say the path *blocks*, or that it is *blocking*).

Finally, it is assumed that no computation along any path fails; it could fail because some arithmetic or other primitives are partially only defined. The outcome of a (dynamic) failure is unspecified.

4.2 Representing automata

The simplest example of a FIACRE process is an automaton. The following process represents the automaton depicted on the right side:

```
process A is
    states s1, s2, s3, s4
    from s1  to s2
    from s2  select to s1 [] to s3 end
    from s3  select to s1 [] to s4 end
    from s4  select to s4 [] to s1 [] to s3 end
```

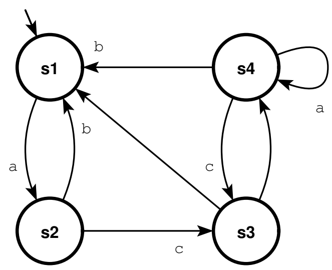


The states must be declared in the header of the process. Then follow the descriptions of process transitions, at most one per declared state. Each process transition (introduced by the keyword **from**) hold a *statement* that may correspond with several transitions of the automaton. Here, the statements are simple jump statements **to**, and nondeterministic choice statements **select**.

4.3 Labelling transitions

The transitions can be labelled. The purpose of labels is to identify some transitions of a process for a later synchronization with transitions from other processes. The next example represents the previous automaton labelled over alphabet $\{a, b, c\}$ as indicated in the picture:

```
process B [a,b,c:sync] is
    states s1, s2, s3, s4
    from s1  a; to s2
    from s2  select b; to s1 [] c; to s3 end
    from s3  to s4
    from s4  select a; to s4 [] b; to s1 [] c; to s3 end
```



Labels (interaction labels, or ports, in FIACRE terminology) must appear as parameters of the process. They are here declared with type **sync**[1], a predefined profile meaning that they are not used to communicate values. Within transitions, labelling is achieved using "synchronization" statements (making precise a label) preceding the jump statements.

4.4 Synchronization and Communication

A *profile* is either the predefined **sync** profile or a series of types separated by `#`. Profiles can be given names using **channel** declarations.

In addition to synchronization, labels may be used to express communications between processes. In that case, they must be declared in the process header with as profiles the type(s) of the value(s) passed and, optionally, with a **input** or **output** attribute restricting the use of that label in the process. By default, communication labels have both the **input** and **output** attributes; meaning that they can serve both to emit and receive values.

An interaction statement is either:

- A synchronization statement, constituted of a label, e.g. `a` in the previous example;
- An emission statement, of general shape `a!e1,...,en`, in which `a` is a label and the `e1,...,en` are expressions of the types appearing in the profile declared for label `a` in the process header;
- A reception statement, of general shape `a?p1,...,pn` **where** `Q`, in which the `p1,...,pn` are destination patterns (soon to be described) and `Q` is an optional predicate that restricts the values to be received. The patterns must obey the profile declared in the process header for label `a`.

As a simple example, the following process reads pairs of integers of type `0..7` on port `a` (restricted to input) then sends their product over port `b`.

```
channel bytepair is byte # byte

process C [a:nat, b:input bytepair] is
   states .....
   var x,y: byte
   from s0  b?x,y; to s1
   from s1  a!x*y; to s0
```

The *single communication* rule: A central rule in FIACRE processes is that at most one interaction label is found along any execution path of any process transition. E.g. the transition **from** `s0  b?x,y; a!x*y; to s0` would be illegal.

There are no syntactical restrictions on the use of shared variables in transitions, in particular, one may read or write shared variable in a transition performing a synchronization or communication. However, a conservative non-interference check is performed that rejects programs potentially exhibiting concurrent writes or concurrent read and write on shared variables.

[1]Replaces **none** that was used in Fiacre V2; frac still tolerates **none** however.

4.5 Parameterized processes

The transitions of a process can be parameterized from parameters passed as arguments to the process, as in the next example below, or obtained by communication with another process.

In the following example showing a parameterized automaton using conditional statements: The automata transitions originating at state `s2` are no more nondeterministic but follow from the value of parameter x, while the set of automata transitions of source `s4` depend upon parameter y.

```
process B [a,b,c:sync] (x:nat,y:bool) is
   states s1, s2, s3, s4
   from s1  a;to s2
   from s2  if x > 4 then to s1 else to s3 end
   from s3  to s4
   from s4  if y then
               select a;to s4 [] b;to s1 [] c;to s3 end
            else
               b;to s2
            end
```

4.6 Assignements

Processes can have local variables, initialized statically or from the process parameters. Computations can be carried out with these variables, using assignement statements and various control structures described in the sequel.

For instance, the following process encodes an automaton that, for any $0 \leq n \leq 4$, has n transitions labelled a followed by $2 * n$ transitions labelled b. In both cases the local variable x is used as a counter:

```
process D [a,b:sync] (n:nat) is
   states s1, s2
   var x:nat := n
   from s1  if x > 0 then a; x := x-1; to s1 else x:=2*n; to s2 end
   from s2  if x > 0 then to b; x := x-1; s2 else x:=n; to s1 end
```

Fiacre assignements statements support a rich set of capabilities. They can be:

- simple, as above or in: `x := e`

  The destination `x` is a single variable; the contents `e` is an expression admitting as type that of variable x.

  There are no type restrictions on assignable variables. In particular, array or records variables can be assigned.

- multiple, as in: `x,y,z := 3,x,y+z`

  Several destinations are provided, separated by comas. A matching number of contents must be provided on the right hand side. Assignement of all destinations are simultaneous.

- random, as in: `x,y :=` **any where** `x > y`

  It can be specified using the **any** contents that a set of destinations is randomly assigned any value belonging to their types. An optional **where** clause restricts the permitted values to those satisfying some predicate. Random assignement is restricted to destinations of numeric types.

  If no candidate contents obeys the predicate in a random assignement with **where** constraint, then the assignement *blocks*, or is *blocking*: no transition is possible through the execution path that holds the assignement.

- with complex destinations, as in: `a[3].f := 4`

  Destinations can be particular fields of record variables or particular positions of array variables. `a[3].f := 4`, for example, stores integer `4` in field `f` of the fourth component of array `a` (arrays elements are indexed from 0).

- with matching constraints, as in: `u(v(z)),4 := true,x`

  Destinations can also be constants, or union constructions, like `u(v(z))` or `4`.

  If the destination holds no variable then, obviously, no destination is assigned, but the content must still match the destination; the assignement simply asserts then a matching constraint. If some destination holds some variable (as in `u(v(z))` or `A(b[3].f)`), then the content much match the destination for the constructors surrounding it. If this is the case, then the destination is assigned the matching value in the contents. Such assignement express then both a matching constraint on a content and, possibly, an effective assignement. A more general **case** construction for achieving such effects will described in Section 4.9.

  As for random assignements with **where** predicates, assignements with matching constraints such that contents and destination do not match are blocking. Such assignements can be used to implement "guards" in process transitions: they restrict the possible transitions to those obeying the **where** or matching constraints.

4.7 Conditionals, loops, sequences

Contrarilly to assignements with constraints, conditional statements **if** _ **then** _ **else** _ **end** are never blocking. If the optional **else** branch is omitted, then control is passed to the statement following the conditional.

For convenience, a **null** statement is provided, which constitutes a "neutral-element" for sequences of statements. Dangling **else**'s are also given a semantics in terms of **null**. The following equivalences hold:

`s;` **null** ≡ **null**`; s` ≡ `s`
if `c` **then** `s1` **end** ≡ **if** `c` **then** `s1` **else null end**

Two constructs are provided for iterative computations: **while** statements and **foreach** statements; the later significantly simplifying loop expressions when iteration variables have interval types. Examples of such constructions include the following in which the while loop computes in variable `f` the factorial of `n` (assumed initialized) and the foreach loop computes in variable `a` the sum of elements of array `b`:

```
Process P is
   states s1, s2, ...
   var n,f,j : int, b : array 8 of int, i : 0..7, a : int
   ...
   from s1  f:=n; j:=n-1; while j > 0 do f:=f*j; j:=j-1 end; ...
   from s2  a:=0; foreach i do a:=a+b[i] end; ...
   ...
```

4.8 Shared variables

The process parameters can be passed by value (cf. Section 4.5) or by reference. The variables passed by reference can be shared among several processes. In process headers, they are distinguished from variables passed by value by an ampersand prefix `&`.

The following process implements a simple busy-waiting mutual exclusion, using a shared boolean variable called `lock` in the process. The exact specification of the task is omitted; the process idles until the `lock` shared variable is false, then it performs some work (the task) and releases the lock:

```
process K (&lock: read write bool) is
   states idle, cs, free
   from idle   if lock then to idle else lock := true; to cs end
   from cs     /* unspecified task; */ to free
   from free   lock := false; to idle
```

Note: When several such processes are run concurrently, the mutual exclusion property follows from the fact that transitions paths are atomic (executed totally or not at all). An implementation of the specification should guarantee atomicity of transition paths, and hence implement the first transition by some test-and-set mechanism rather than a conditional. More elaborated mutual exclusion mechanisms will be discussed in Sections 5 and 6.

Shared variable arguments can have attributes **read** and/or **write**. They have both by default but if only one is specified, then the usage of the shared variable into the process is restricted accordingly.

4.9 Case and pattern matching

Case statements allow one to match a value against various patterns and, in case of match, to assign some variables to the corresponding contents in the value.

case statements have the following general form, in which `v` is the value to be matched, the `pi` are patterns, `si` is the statement to be executed if `v` matches `pi` and **any** is a special pattern matched by any values:

```
case v of p1 -> s1 | ...  | pn -> sn | any -> s0 end
```

Les us call *destination* a variable followed by record and/or array subscripts (e.g. `y` or `x[3].f[5]`).

A pattern is either a literal value, a union constant, a destination or some construction made from those and 1-ary union constructors. For instance, if type `ty =` **union** `A | B of` **int end** has been declared, as well as variables `x:` **int** and `c: ty`, then `A`, `B(5)`, `c`, and `B(x)` are patterns of type `ty`. In addition a particular pattern is provided, overloaded at all types, written **any** (not to be confused with the random assignement construction).

A value `v` matches a pattern `p` if:

- `p` is **any**, a variable or a destination;
- `p` is a literal value or a union constant and `v` = `p`;
- `p` has shape `c (p')`, in which `c` is some 1-ary union constructor, `v` has shape `c (v')` and `v'` matches `p'`.

If pattern `p` includes a destination and value `e` matches `p`, then there is a unique subvalue `e'` of `e` that matches the destination in `p`; that destination is assigned that unique `e'`.

The following is a typical example of use of a **case** statement. If the message received over port `rq` is a status, then boolean **true** is sent over port `out1`. If that message is packed with the `value` constructor, then the value encapsulated is assigned to variable `key` and the value `b[key]` is send over port `out2`.

```
type request is union status | value of 0..7 end
process P [inp:request, out1:bool, out2:int] (b:array 8 of int) is
   states s1, s2
   var key :  0..7, rq :  request
   from s1 inp?rq; to s2
   from s2 case rq of
         status -> out1!true
       | value(key) -> out2!b[key]
   end; to s1
```

The different clauses of a **case** statement are considered in the order they are written, from first to last. If no match is found, then the construction is blocking: no transition is possible though the **case** statememt.

4.10 Initialization of variables

The variables locally declared in processes can be initialized by three methods:

- Statically in the **var** declaration, as in:

  ```
  process P (b:int) is
     states ...
     var c : array 8 of int := [0,1,2,3,4,5,6,7]
     from  ....
  ```

- Statically in an initialization statement. That optional statement is introduced by the **init** keyword and placed before the first process transition. Initialization statements may not write shared variables and may not contain interaction labels. Each execution path of the init statement must contain a **to** statement.

```
process P (b:int) is
   states s0, s1, s2
   var d:byte, c : array 8 of int
   init  foreach d do c[d] := d end;
        if b<4 then to s1 else to s2 end
   from  s1 ....
```

In absence of **init** statement, the initial state of the process if the first state for which a transition is defined. Note that the **init** statement allows one to parameterized the initial state.

- Dynamically in some transition, before the first transition that reads it.

4.11 Subtyping

All variables (whether local, argument, shared) and interaction labels are declared of some type. As seen in Section 2, expressions can have in general several types, though, in any context, they will be assigned a single one: the largest allowed by the context. This section discusses the subtyping rules of FIACRE.

A *subtyping* relation is defined on FIACRE types. Intuitively, we have `ty' < ty` if any value having type `ty'` also has type `ty`. The subtyping relation is formally defined in [2], page 12. In particular, any interval type is a subtype of **int** and of any larger interval type, and **nat** < **int**.

If `ty' < ty`, then an expression of type `ty'` can be used everywhere an expression of type `ty` is expected. In particular:

- If variable `x` has been declared with type `ty`, then it may be initialized or assigned by any expression of a smaller or equal type. E.g. the following process is legal:

  ```
  process P (a: 0..3) is
     states s
     var x : 0..7 := a, y : nat := a
     from s  x := a; y := x; to s
  ```

  Both variable `x` and `y` may be initialized with the value of `a`; `x` and `y` can be assigned the value of `a`, and and `y` can be assigned the value of `x` as well, since `0..3` < `0..7` < **nat**.

  If some variable has declared type `ty`, then its content is always assumed to have type `ty`, even though it could also have some smaller type. Hence the assignement `x := y` is ill-typed.

- If the interaction label `q` has been declared with profile `ty`, then any value of a smaller or equal type can be sent over `q`, but the values received on `q` may only be stored in variables of type larger or equal than `ty`.

By construction, any value computed is ultimately stored at some destination, sent over some port or appears in some condition. In all cases, a largest acceptable type can be determined from it, from either the declared type of the assigned variable, the declared type of the port or the generic type of the primitives involved. From this upper "type-bound" for expressions can be determined recursively a largest acceptable type for all primitives occurring in the expression.

4.12 Time constrained silent transitions, wait, loop statements

wait statements allowed in silent transitions [in general short hand for ...]
loop statement versus **to** self ... [clock resets]

4.13 Priority constraints within processes

unless clauses in **select** statements ... [in general short hand for ...]

5 Components

5.1 Purpose and contents

Components describe interactions between processes or components, in a hierarchical manner, and possibly constrain these interactions with timing and/or priority requirements. Components also create and initialize shared variables, if any.

As processes, components may be parameterized by interaction labels, value parameters and shared variables. A component description may include (all optional except the body):

- Local variable declarations: Those variables may be used for computing the arguments passed to the instances in the body; they may also be shared among instances;
- Local port declarations: These create interaction labels, each associated with a profile and possibly a timing constraint;
- Priority declarations over interaction labels;
- An initialization statement (**init**);
- A body, which is some composition of process or component instances.

5.2 Instances, Compositions

Instances: An instance is a process or component name, together with the parameters passed to it, if any. They have the following form in which both argument lists are optional, the `li` are interaction labels and the `ai` are arguments either constituted of an expression (for those passed by value) or of a variable prefixed by `&` (for passing shared variables by reference):

```
p [l1,...,ln] (a1,...,an)
```

The composition operator: Compositions have the following form, in which the `lseti` are lists of labels (e.g. `a,b,c`) and the `insti` are instances or embedded compositions:

```
par lset1 -> inst1 || ...|| lsetn -> instn end
```

To ease writing compositions with large label sets, these may be factorized: If `lset0` is included in all `lseti`, then the above composition can also be written as follow, in which `lseti' = lseti-lset0`:

```
par lset0 in lset1' -> inst1 || ...|| lsetn' -> instn end
```

Label sets are optional. If some is empty, then the arrow following it can be omitted.

Sorts and the universal label set $*$: The *sort* of `insti` in the above composition is the set of labels "known" by `insti`, that is:

- If `insti` is an instance `p [l1,...,ln] (a1,...,an)`, then it is the set of labels among label parameters `[l1,...,ln]`;
- If `insti` is a composition **par** `insti1 || ...|| instin` **end**, then it is the union of the sorts of the `instij`.

Conventionally, the *universal label set*, written `*`, denotes the set of all labels known to the composition element it precedes, that is its sort. In addition, the two following forms are considered equivalent:

```
par * -> inst1 || ...|| * -> instn end
par * in inst1 || ...|| instn end
```

Label sets specify interactions: The label sets in compositions specify the interactions between the instances or compositions involved in the composition, as follows.

- If no label is specified, as in:

  ```
  par inst1 || ...|| instn end
  ```

  Then the behavior of the composition is simply the interleaving, or *shuffle*, of the behaviors of the instances or compositions involved.

- If all label sets are universal, as in:

  ```
  par * -> inst1 || ...|| * -> instn end
  ```

  Then the behavior of the composition is the *synchronous product* of the behaviors of the instances or compositions involved: for each `i`, every labelled path of `insti` must be synchronous with some identically labelled path from all components `instj` having that label in their sort.

  From the definition of `*`, The above notation is equivalent to the following, in which, for each `i`, `sorti` = *sort*(`insti`):

  ```
  par sort1 -> inst1 || ...|| sortn -> instn end
  ```

- Otherwise, if label sets are made explicit as in:

  ```
  par lset1 -> inst1 || ...|| lsetn -> instn end
  ```

  Then, every path of `insti` labelled by some `l` $\in$ `lseti` must be synchronous with an identically labelled path from all components `instj` such that `l` $\in$ `lsetj`.

Here are two simple example of components. The first creates a `lock` variable shared by four instances of the `K` process ensuring mutually exclusive access found in Section 4.8:

```
component Mutex is
   var lock : bool := false
   par K (&lock) || K (&lock) || K (&lock) || K (&lock) end
```

The second component synchronizes two copies of the automaton defined in Section 4.3 on ports `a` and `b`, while leaving all labelled paths open for further synchronizations:

```
component Sync [a,b,c:sync] is
   par a,b -> B [a,b,c] || a,b -> B [a,b,c] end
```

Graphical representation of compositions ?

5.3 Local variables

Variables may be declared locally in components, using the same notations as in processes except that their initialization is mandatory. Local variables can be used for computations (e.g. of instance arguments), or holders for the arguments passed to instances. Initialization can be done in the **var** declaration or in an initialization statement. Initialization statements for components are similar to those used in processes except that they may not contain **to**.

The variables locally declared in components can be shared among the instances occurring in the component body; passing a shared variable to an instance requires to give as argument of the instance the name of the variable prefixed by `&`. This will only be legal if the corresponding component or process expects a variable passed by reference at that position.

By construction, a process may not communicate to another the location (address) of a shared variable, hence the scope of a shared variable is the body of the component in which it is declared.

The use of shared variables is illustrated by the previous `Mutex` component, together with the `K` process in Section 4.8.

5.4 Local interaction labels

Local label declaration create interaction labels, to be passed as "label arguments" to the instances in the body of the component.

The scope of a label is the body of the component it is declared within. Hence, if a locally declared label is passed to an instance, then interaction offers are closed on that label. Interactions in some body component that should remain open should make use of labels appearing as argument of the component.

As an example, consider the following component `C`. It pipes two instances of process `P`. The two instance communicate via a local port `tmp`, while ports ii and oo of the component can further interact:

```
process P [ii:int,oo:int] is
   states s1, s2
   var x : int
   from s1  ii?x; to s2
   from s2  oo!x; to s1
```

```
component C [ii:int,oo:int] is
   port tmp : int
   par tmp -> P [ii,tmp] || tmp -> P [tmp,oo] end
```

5.5 Time constraints

Closed interactions may be assigned a time interval: Intervals are associated with a label and apply to all interactions using that label. If interval $[\alpha, \beta]$ is associated with label `p`, then, from the instant at which it was last enabled, any interaction labelled `p` must wait at least α units of time but may not be delayed more than β units of time. Fiacre does not make precise the exact unit of time, but it is assumed that all components in a fiacre specification make use of the same unit.

To make the words "last enabled" more precise, we need to define when interactions are conflicting. An interaction is some set of synchronous transitions belonging to different process instances. With each interaction one can associate the set of source states of the process transitions involved. Two interactions are *in conflict* when they share one of these states.

Fiacre interactions are computed as follows:

- Assume state `s` enables some set of interactions, each with their current time interval. One interaction among those is choosen and performed;
- Then the interactions enabled at the target state are computed. In this set, those that were not enabled at state s or were enabled at s but were in conflict with the interaction taken start with their initial timing constraints; all other preserve their current constraint.

The first example is the same as the previous pipe example except that communications between subcomponents can be delayed between 2 and 5 units of time, and that they have priority over input and output events (priorities will be addressed in the next Section):

```
component C [ii:int,oo:int] is
   port tmp : int in [2,5]
   priority tmp > oo | ii
   par tmp -> P [ii,tmp] || tmp -> P [tmp,oo] end
```

The next example exhibits two conflicting interactions. In process `Q`, the actions labelled `a` and `b` are always enabled simultaneously, but cannot be performed simultaneously (processes express sequential behaviors). This property remains true when these transitions are synchronized with the action of process `R`. Hence, performing one of the two possible interactions labelled `c` in the component disables then restarts the other interaction.

```
process Q [a, b:none] is
   states s
   from s  select a [] b end; to s
process R [z:none] is
   states s
   from s  z; to s
component Z is
   port c : sync in [1,3]
   par c -> R [c] || Q [c,c] end
```

5.6 Priority constraints

Component descriptions may include priority declarations between interaction labels. Priority constraints apply to all interactions in the body of the component, and are inherited in further compositions. For instance, if the priority relation in the component holds `a < b`, then any interaction labelled `b` in the body of the component has priority over any interaction labelled `a`. If these interactions are open and further synchronized with others, then the later inherit the priority constraints of the former.

After compositions are performed, the resulting priority relation (more precisely its transitive closure) must be a partial order.

A simple illustration of priorities on open interactions was shown in the last `C` example. The next `C2` component is similar to `C` except that it pipes two instances of component `C` rather than two instances of process `P` and that it does not make explicit any priority constraint. However, since `C` specified `oo > ii`, the ports in the `C2` body inherit constraints `oo2 > tmp2` and `tmp2 > ii2` (as well as `oo2 > ii2`, by transitivity of `>`).

```
component C2 [ii2:int,oo2:int] is
   port tmp2 : int in [2,5]
   par tmp2 -> C [ii2,tmp2] || tmp2 -> C [tmp2,oo2] end
```

6 Examples

See `http://www.laas.fr/fiacre/documents.php` for some example Fiacre specifications.

References

[1] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Fréderic Lang, and François Vernadat. Fiacre: an intermediate language for model verification in the TOPCASED environment. In J.-C. Laprie, editor, *Proceedings of the 4th European Congress on Embedded Real-Time Software ERTS'08 (Toulouse, France)*, 2008.

[2] Bernard Berthomieu, Jean-Paul Bodeveix, Mamoun Filali, Hubert Garavel, Fréderic Lang, Florent Peres, Rodrigo Saad, Jan Stoecker, and François Vernadat. The Syntax and Semantics of Fiacre – Version 2.0. 2007.

Appendix: The syntax of Fiacre

A1. Notations

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.

A2. 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
```

Comments: A Fiacre 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 loop nat none sync not null of 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:

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

A3. Types and Channels

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

channel_id ::= IDENT
channel ::= **sync** | type$^{+}_{\#}$ | channel_id
channel_decl ::= **channel** channel_id **is** channel

A4. Expressions

unop ::= '-' | '+' | '$' | **not** | **full** | **empty** | **dequeue** | **first**
binop ::= **enqueue** | **append**
infixop ::=
      or
    | **and**
    | '=' | '<>' |
    | '<' | '>' | '<=' | '>='
    | '+' | '-'
    | '*' | '/' | '%'

Infixes 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

| '[' exp$^+_,$ ']'
| '{' (field '=' exp)$^+_,$ '}'
| '{|' exp$^*_,$ '|}'
| constr [atomexp]
| var '(' exp$^+_,$ ')'
| unop atomexp
| binop '(' exp ',' exp ')'
| exp infixop exp
| exp '?' exp ':' exp
const_decl ::= **const** var ':' type **is** exp

A5. Functions

name ::= IDENT
farg_dec ::= (var)$^+_,$ ':' type
var_dec ::= var$^+_,$ ':' type [':=' exp]
atompatt ::=
any | literal | var | constr
| atompatt '[' exp ']' | atompatt '.' field
| '(' pattern ')'
pattern ::= atompatt | constr [atompatt]
fstatement ::=
null
| pattern$^+_,$ ':=' exp$^+_,$
| **while** exp **do** fstatement **end** [**while**]
| **foreach** var **do** fstatement **end** [**foreach**]
| **if** exp **then** fstatement (**elsif** exp **then** fstatement)* [**else** fstatement] **end** [**if**]
| **case** exp **of** (pattern '->' fstatement)$^+_|$ **end** [**case**]
| fstatement ';' fstatement
| **return** exp
function_decl ::=
function name '(' farg_dec$^+_,$ ')' ':' type **is**
[**var** var_dec$^+_,$]
[**begin** fstatement **end** | **return** exp]

A6. Processes

state ::= IDENT
port := IDENT
left ::= '[' DECIMAL | ']' DECIMAL
right ::= DECIMAL ']' | DECIMAL '[' | '...' '['
time_interval ::= left ',' right
port_dec ::= port$^+_,$ ':' [**in**] [**out**] channel
arg_dec ::= ([&] var)$^+_,$ ':' [**read**] [**write**] type
transition ::= **from** state statement
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^+_[] (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^+_,
process_decl ::=
    process name
       ['[' port_dec^+_, ']']
       ['(' arg_dec^+_, ')']
    is  states state^+_,
       [var var_dec^+_,]
       [init statement]
       transition^+
```

A7. Components

```
arg ::= exp | '&' var
instance ::= name ['[' port^+_, ']']  ['(' arg^+_, ')']
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
```

In priority declarations, $\mathtt{a}_1|\dots|\mathtt{a}_n > \mathtt{b}_1|\dots|\mathtt{b}_m$ is a shorthand for $(\forall i \in \{1,\dots,n\})(\forall j \in \{1,\dots,m\})(\mathtt{a}_i > \mathtt{b}_j)$.

A8. Programs

```
declaration ::=
      type_decl
    | channel_decl
    | const_decl
    | function_decl
    | process_decl
    | component_decl
program ::=
    declaration+
    name
```