FC2Parameterized & FC2Instantiate

Tomás Barros

November 30, 2005

In order to work with our parameterized formalism, we have developed a tool, named FC2Instantiate, to get instantiations from the parameterized descriptions given the finite abstract domain of their unbounded parameters. Both description and domains inputs are in FC2Parameterized format, the output of the tool is standard FC2 format.

In this section we describe the syntax of the FC2Parameterized format and the use of the tool FC2Instantiate through a guided example.

Given a system of communicating automata with parameters and the domain of its unbounded parameters, FC2Instantiate is a tool, 100% written in Java, that generates a finite system of communicating automata by translating each of the parameters to all the values in its domain.

We start by reviewing the FC2 format, necessary to a better understanding of its extension, FC2Parameterized.

1 FC2 Format

The FC2 format allows for description of labelled transition systems and networks of such. Automata are tables of states, states being each in turn a table of outgoing transitions with target indexes; networks are vectors of references to subcomponents (i.e., to other tables), together with synchronisation vectors (legible combinations of subcomponent behaviours acting in synchronised fashion). Subcomponents can be networks themselves, allowing hierarchical description.

It is important to note that this format was created as a mean of communication between several software tools in the process algebra area. The format specification defines only the syntax, and it is up to the tools to define a compatible semantics for the different format's entities. Indeed, it has been used with success in tools dealing with quite different flavours of process algebras, included timed ones.

In the following we introduce the syntax of FC2 format.

1.1 Structure of an FC2 file

An FC2 object (usually found in a file) consists of:

```
[optional]  Declarations of expression operators,
A table of nets, containing:
  the number of nets,
  global semantic tables,
  a global label,
  the nets.
  Each net (or graph) containing:
    [optional] its number,
    local semantic tables,
    the net label,
    The vertice table [optional], composed of:
      the number of vertices,
      the vertices.
      Each vertex contains:
        [optional] its number,
        the vertex label,
```

```
The edges table [optional], composed of:
  the number of edges,
  the edges.
  Each edge contains:
    [optional] its number,
    the edge label,
    the index(es) of the resulting state(s).
```

FC2 simple example

```
nets 1
hook "main">0
net 0
struct "t1" logic "initial">0
hook"automaton"
vertice 2
vertex0 struct"v0" edges2
behav"a" result 0
behav"b" result 1
v1s"v1"E2
b"a"r0
b"b"r1
```

For example, this FC2 object contains a single net (indeed an automaton, as indicated in the net hook). The automaton has two vertices with two edges each. All information attached appears here directly in place, though it could have been tabulated. The text for the first vertex appears in long form (more readable), while the second vertex is in short, compact, form.

1.2 FC2 Objects

Expressions

```
exp : CONSTANT
    | UNARY exp
    | exp INFIX exp
    | PREFIX OP exp CP
    | OP exp CP
    | STRING
    | STAR
    | ref
    ;
```

```
ref :  INT
    |  GLOBAL INT
    |  BQUOTE INT
    ;
```

An expression represents an algebraic term. Some of the operators are predefined, some are user-defined, and declared in the file header. The basic bricks are the constant operators, the strings, and integers used as references in the semantic tables.

The "*" character has a special syntax (token STAR), used for specifying idle arguments in synchronisation vectors:

* is the constant expression "*idle*"

`*i` with `i` a positive integer, is equivalent to the sequence of `i` idle constants, "`*,*,...,*`".

Types Operators are typed, and the types are used to determine in which table a reference is to be searched. Possible types are: `behav, struct, logic, hook, int, any`. The type `any` is used as a type variable.

The following table lists the predefined operators. There is no predefined PREFIX.

```
Constant:  "t" : behav    (long form "tau")
           "q" : behav    (long form "quit")
           "_" : any
Unary:     "!" : any -> any
           "?" : any -> any
           "~" : any -> any
           "#" : any -> any
```

```
Binary:    "^" : any * int -> any
           "+" : any * any -> any
           "<" : any * any -> any
           ">" : any * any -> any
           "," : any * any -> any
           ";" : any * any -> any
           "." : any * any -> any
```

All infix operators have priorities and they are left associative (e.g. `x.y.z` means `(x.y).z`). User-defined operators will be assigned a priority in the declarations as shown later.

In the following we list the priorities of predefined operators. The priorities range from 0 to 50, the lower the number, the strongest the priority.

Operator	**Priority**
+	50
> <	40
,	30
;	20
. ^	10

Example of expressions

```
tau     % the constant "tau", usually interpreted as the internal action
q       % the special action "quit" (or the "delta" of Lotos)
3       % a reference to the entry number 3 in the corresponding semantic table
!"s"    % the application of the unary operator "!" to the string "s",
        % e.g. the emission of signal "s"
1+2     % the sum of the expressions referenced 1 and 2

(t<1,!(3,?(11,1)))+(t<2,!(10,?(0)))
        % a big expression
```

1.3 Labels

Labels are used to store all kind of information attached to FC2 objects (nets, vertice, edges). They are records with 3 specific fields: **struct, behav, logic**, plus an additional field, **hook**, gathering all kind of extra information. All fields are optional. When a field is present, it contains a single expression.

Hooks will usually contain information private to a tool, or to a small set of tools. In addition, some of the conventions defined in this document also make use of hooks. Hook information need not be defined by the format, so it will not be parsed: it appears as mere strings in the hook value. Each tool will decide whether or not to decode this value, depending on the name of the hook. On the other hand, some tools may agglomerate hooks when building new objects, so hook values (with same name) will be composed from string with the usual expression operators.

Example of labels

```
struct "name0"
behav "act1"+"act2"
                        % value can be explicite expressions
logic "initial"
logic (~0 + 1).2        % or use references

                        % To give a hook a name, one usually use the ">" operator
hook "colour">"red"
hook "coord">"x=134,y=24"+"x=35,y=124"
```

Convention : In a given label, the Struct, Behav, and Logic fields should appear at most once. There can be several Hook fields, though, and multiple hooks will be interpreted as conjunctions (as if connected by a "." operator).

1.4 Semantic Tables

Expressions may appear at many places in the file, but usually their values are tabulated, both for compact and semantic reasons. Thus, expressions are gathered in tables, sorted by type; there are four semantic table types, corresponding to the four label fields:

behavs, structs, logics, hooks

A *semantic table* has a type, a size, and a number of entries. Each entry has an optional index, and contains an expression. The size is a positive integer, bigger than all entry indexes. An entry without an index gets its index from the preceding entry, plus one.

Example of Semantic Table

```
behavs 3
:0 tau
:1 "a1"
:2 !1

logics 2
:0 "initial">0
:1 may (1)      % refers to behav :1, if "may" is behav -> logic
:2 diverge . must (2)
```

Tables can be found either inside a net (*local* table), or attached to the top level FC2 object (*global* table, shared by all nets)

Semantic interpretation: The size indication is intended to ease the creation/filling of tables at parse time: the "size" specified is the effective size of the table, and not the number of entries. Indexes start at 0. Indexes appearing explicitly in entries are –real– indexes; this implies that there can be holes in a table.

1.5 Nets

The nets table is the top level structure of an FC2 file.

```
net_table :  /* EMPTY */
          |  NETS INT tables labels net_list
    ;
```

The integer following the **nets** (or **N**) keyword is the number of nets in the file (mandatory). The "table" and "labels" non-terminals contain the global tables, and the global label (information concerning the whole network).

Convention: When needed, the root of the nets tree is indicated by a conventional hook of the form: `hook "main">0,` in which the right hand side argument of `>` is of type `net`

Each individual net contains local tables, a label, and a vertice table:

```
net :   NET opt_index tables labels vertice_table
```

The index of the net (integer immediately following the *net* keyword) is optional.

Convention: The hook of the net usually contains an indication of its *kind*. The kinds currently foreseen are: `"transducer", "synch_vector", "partition", "automaton"` The kind of a net carries information needed to resolve some ambiguities. The default kind is "automaton".

Convention: The initial vertex (or vertice) of an automaton or of a transducer may be indicated either in the logic field of the net label, or in the logic field of the vertex (resp. vertice) itself. Some tools (e.g. FC2Tools) require a single initial vertex to be defined.

Initial indication

```
...
net 0 hook "automaton" struct "foo"
      logic "initial">1
      vertice 3
v s"st_0" edges 1
b"a" r1
v s"st_1" edges 1
b"b" l"initial" r2
v s"stop"
```

The **struct** field of the net label is used to encode the structure of the network (which net contains which other nets), using a "<" operator, with the name of the node as first argument, and a comma-separated list of net references as second argument:

Example of Structured Net

```
nets 2
hook "main" "0"

net 0 hook "transducer" struct "system"<1,1,1
vertices ...
                    % the main net is called "system", and has
                    % three identical sons, copies of net 1.

net 1 hook "automaton" struct "cell"
                    % this one is a simple automaton named "cell"
                    % (a net with no argument)
```

1.6 Vertice and Edges

A vertice table is the main component of a net. The number of vertice in the table is mandatory.

```
vertice_table : /* Empty */
              | VERTICE INT vertex_list
              ;
...
vertex : VERTEX opt_index label edge_table
```

A Vertex contains an optional index, a label, and an edges table. The number of edges in the edges table is mandatory.

```
edge_table : /* Empty */
           | EDGES INT edges
           ;
...
edge : EDGE opt_index label target_vertice
           | label target_vertice
           ;
target_vertice : result
           | result exp
               ;
result : ARROW | RESULT /* -> or r */
       ;
```

In each **edge** entry:

- The **edge** (or **e**) keyword is optional.
- The edge index is optional.
- The result keyword is mandatory (either **r**, **result**, or `->`), and is followed by an expression of type vertex, usually a single vertex index, or a "+"-separated list of vertex indexes (e.g. "`1+2+5+6`").
- When a net has only one vertex (usual for synchronisation vectors) this index (but not the preceding **result** keyword) can be omitted.

Example of Edges

```
v s"state1" E2       % vertex with 2 edges
e0 b?0 r1            % edge 0 has behaviour ?0 and one target vertex (number 1)
e1 b?1 r 1+2+3       % edge 1 has 3 target vertice, number 1, 2, and 3
                     %
vertex 2 struct "state2" edges 2    % another vertex, in expanded syntax
edge 0 behav ?0 -> 1
edge 1 behav !1 result 1+2+3
```

1.7 Declarations

The declaration section allows a specific tool to use more operators than those predefined in FC2. The syntactic class, the lexical representation, and the type of the operator must be specified.

The syntax class is one of **constant, unary, prefix, infix**. The difference between **unary** and **prefix** is that the latter requires parenthesis surrounding its arguments. Locally-declared **infix** operators should be assigned to a priority between 0 and 50.

The lexical representation must be a legal token for the FC2 scanner, that is either a SYMBOL (single character symbol) or an IDENT (fully alphabetic identifier).

The type must be coherent with the syntactic class.

```
declaration : decl_class decl_token decl_type
            ;
decl_class : PREFIX_DECL | UNARY_DECL | INFIX_DECL priority INT | CONSTANT_DECL
           ;
decl_token : IDENT | SYMBOL
           ;
decl_type : OP types CP ARROW type
          ;
```

Example of Declarations

```
constant diverge () -> logic
infix $ (logic logic) -> logic priority 45
prefix & (struct behav behav) -> struct
```

2 Specification of Parameterized System

The FC2Parameterized format allows the systems specification using parameterized networks of communicating automata. It is an extension of the standard FC2 format with additional operators and user defined variables. This extension is done using the declarations section of FC2 format. We use a graphical syntax for a better understanding.

2.1 Example description

We use the well known Consumer-Producer system as an example. The example is shown in Fig. 1, it is composed of a single bounded buffer, with a maximal capacity of Max elements, and a bounded quantity of consumers ($\#consumer$) and producers ($\#producer$) running in parallel.

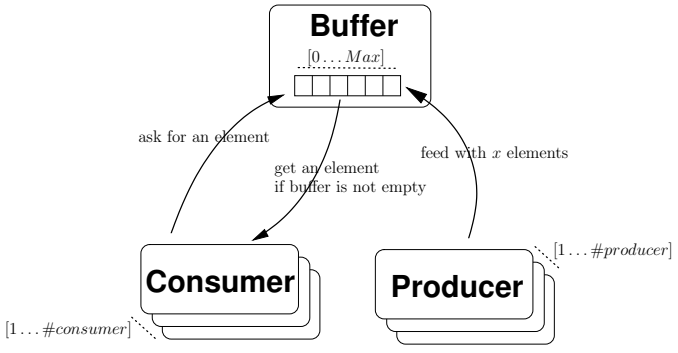

Figure 1: Consumer-Producer system interaction

Both producers and consumers have a single functionality. The producers keep feeding the buffer with an arbitrary quantity (x) of elements without waiting for a response, while the consumers request an element from the buffer and wait for its response.

On the contrary, the buffer has two functionalities. On one side it keeps a bounded stock where elements are added and taken, but it also manages a queue where the request from the consumers are enqueued until elements become available.

We model each functionality of the system as a pLTS, and the hierarchical composition of this functionalities (pLTS) as pNets. We chose to describe the example behaviour in two FC2 files: `SystemParameterized.fc2` describes the synchronisation network (i.e. the composition) between the producers, consumers and the buffer; both the behaviour of a consumer and the behaviour of a producer is included in the same file, while the behaviour of the buffer is described in the file `BufferParameterized.fc2`.

2.2 The System

Let's take a look back to the system, it is shown in Figure 2. The arbitrary numbers of consumers and producers are respectively represented by the exponents c and p in the figure.

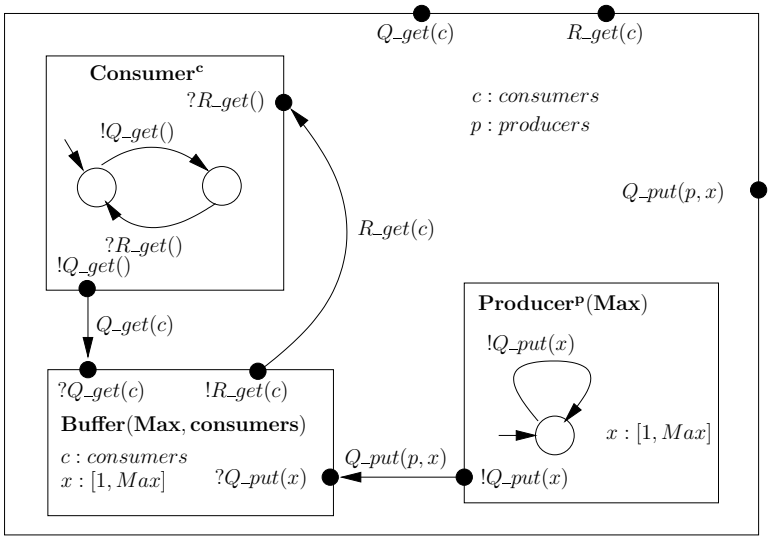

Figure 2: Parameterized consumer-producer system

In the FC2Parameterized format, the system is described by three nets (consumer, producer and buffer) and a fourth net defining the synchronisation between them (having the synchronisation vectors) as follows:

SystemParameterized.fc2

```
declarations
constant consumers() ->any
constant producers() ->any
infix & (any any ) -> any priority 8
... % other necessary declarations
nets 4
   hook"main" > 0
   struct"Consumer-Producer"
   net 1
      struct "Buffer"
... % the rest of the net definition (semantic table)
   net 2
      struct "Consumer"
... % the rest of the net definition (semantic table & vertices table)
   net 3
      struct "Producer"
... % the rest of the net definition (semantic table & vertices table)
   net 0
      hook "synch_vector"
      struct _< 1,2&consumers,3&producers
... % the rest of the net definition (including synchronisation vectors)
```

When writing a parameterized system, all the variables and operators not predefined in the standard FC2 format should be declared in the `declarations` section of the file. In our example, the file begins declaring the variables `consumers` and `producers`, encoding respectively the set of consumers and producers. The keyword to declare a variable in FC2Parameterized is `constant`. The type of the variables (after the arrow) can be any of the FC2 Types. However, the FC2Instantiate tool does not make type checking yet, meanwhile we mainly use the type "any".

The operator `&` declared as infix in the file, when is used within the `struct` label of a `net` instantiates the referenced net (its left argument) to the size of the given set (its right argument). Then `struct _< 1,2&consumers,3&producers` indicates that the network is composed by one single instance of the network 1, #`consumers` instances of the network 2 and #`producers` instances of the network 3. For instance if `consumers = {"cons1", "cons2"}` and `producers = [2, 4]`, then using the tool `struct _< 1,2&consumers,3&producers` is expanded to `struct _< 1,2,2,3,3,3`.

Before given the complete `SystemParameterized.fc2` file contents, we analyse each one of its networks.

2.3 Consumer

The parameterized automaton modelling the Consumer behaviour is shown in Figure 3.

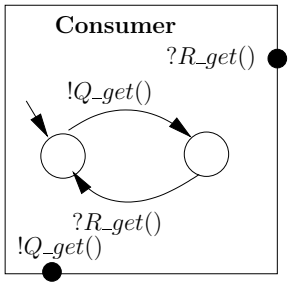

Figure 3: Parameterized Consumer

SystemParameterized.fc2

```
...
   net 2
      struct "Consumer"
      behavs 2
         :0 "Q_get()"
         :1 "R_get()"
      logic "initial">0
      behav !0+?1
      hook "automaton"
      vertice 2
         vertex 0
            edges 1
               edge0
                  behav !0 -> 1
         vertex 1
            edges 1
               edge0
                  behav ?1 -> 0
...
```

Since the consumer does not use parameters, its definition is done using the standard FC2 format as shown next to the figure.

2.4 Producer

The parameterized automaton modelling a Producer behaviour is shown in Figure 4. The transition's alphabet is specified in the behaviour's semantic table of the net. The semantic table for the Producer is shown next to the figure.

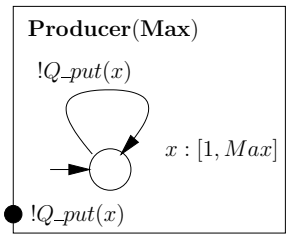

Figure 4: Parameterized Producer

SystemParameterized.fc2

```
declarations
...
  infix & (any any ) -> any  priority 8
  prefix in (any any) -> any
...
  net 3
    behavs 1
      :0 "Q_put"&in(1,Max)
...
```

The `&` operator, when inside a semantic table, takes a FC2 Expression in its left side and a list of sets (each set separate by "`,`") in its right side (in our example there is only one set). This operator generates, during instantiation, an entry in the semantic table for each member of the Cartesian product of the sets in its right side.

The `in` operator takes two integers and generates the set of all the integers between those integers inclusive ($in(1,3) = \{1,2,3\}$). For instance, when running the FC2Instantiate tool with $Max = 3$, the semantic table above is extended to:

SystemInstantiatied.fc2

```
...
  net 3
    behavs 3
      :0 "Q_put(1)"
      :1 "Q_put(2)"
      :2 "Q_put(3)"
...
```

Within states (vertices) and transitions (edges), their local variables should be assigned using a hook label (one per variable). The variables are assigned with the infix operator `=`, which takes a variable name in its left side and a set in its right side. When using the tool, the variable is assigned to each element of the set.

In the producer the state has no variables. On the contrary, the only transition contains the variable x which will range in the set $[1, Max]$. Then the vertices table of the Producer is:

SystemParameterized.fc2

```
declarations
...
  infix = (any any ) -> any priority 8
  infix & (any any ) -> any priority 8
...
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
```

The left parameter of the operator `&`, when used inside an automaton's edge (which is not a synchronisation vector), is a reference to an action in the corresponding semantic table of the `net`; and the right parameter is an expression. When using the tool, an expression of the form `0&x` (inside an automaton edge) will be replaced by a reference to the entry `0` of the semantic table (note that the corresponding entry in the table must be also parameterized by `x` or equivalent) for each instantiation of `x`.

The complete parameterized description of a producer and its instantiation when $Max = 3$ are:

SystemParameterized.fc2

```
declarations
  constant Max() -> int
  constant x() -> int
  infix & (any any ) -> any priority 8
  infix = (any any ) -> any priority 8
  prefix in (any int) -> any
...
  net 3
    behavs 1
      :0 "Q_put"&in(1,Max)
    logic "initial">0
    hook "automaton"
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
...
```

SystemInstantiated.fc2

```
...
  net 3
    behavs 3
      :0 "Q_put(1)"
      :1 "Q_put(2)"
      :2 "Q_put(3)"
    logic "initial">0
    hook "automaton"
    vertice 1
      vertex 0
        edges 3
          edge0
            behav !0 -> 0
          edge1
            behav !1 -> 0
          edge2
            behav !2 -> 0
...
```

2.5 Buffer

The buffer communicates with consumers and producers through three actions: *Q_get* to receive a request for element from a consumer, *R_get* to give the answer to the consumer and *Q_put* to receive feeds from producers. Since the behaviour of the buffer is given in a separate file, in the file

SystemParameterized.fc2 only is specified its behaviour semantic table (i.e. its alphabet of actions) as follows:

SystemParameterized.fc2

```
declarations
...
  constant Max() -> int
  infix & (any any ) -> any priority 8
...
   net 1
      struct "Buffer"
      behavs 3
         :0 "Q_get"&consumer
         :1 "R_get"&consumer
         :2 "Q_put"&in(1,Max)
      behav ?0+!1+?2
      hook "synch_vector"
```

2.6 Synchronisation vectors and complete SystemParameterized.fc2 file

Finally in the `net 0` we give the synchronisation vectors that define the actions from the producers, consumers and the buffer which should be synchronised.

In the synchronisation vector we use the $ operator to indicate which of the instances of a network is being to be referenced.

In a synchronisation vector a reference such as `behav 0&c < ?(0&c)$1, !0$(2&c) -> 0` indicates that the action labelled with the entry 0 (for an instantiation of c) in the net 1, is synchronised with the entry 0 of the net 2 (where the net 2 is instantiated for an evaluation of c). This synchronisation is observable and produces a global action labelled with the entry 0 (for an instantiation of c) in the behaviour semantic table of the net having the synchronisation vector.

The complete `SystemParameterized.fc2` file is shown bellow:

SystemParameterized.fc2

```
declarations
constant Max() -> int
constant x() -> int
constant consumers() ->any
constant producers() ->any
constant c() ->any
constant p() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix $ (any any ) -> any priority 8
infix = (any any ) -> any priority 8
prefix in (any int) -> any
nets 4
  hook"main" > 0
  struct"Consumer-Producer"
  net 1
    struct "Buffer"
    behavs 3
      :0 "Q_get"&consumers
      :1 "R_get"&consumers
      :2 "Q_put"&in(1,Max)
    behav ?0+!1+?2
    hook "synch_vector"
  net 2
    struct "Consumer"
    behavs 2
      :0 "Q_get()"
      :1 "R_get()"
    logic "initial">0
    behav !0+?1
    hook "automaton"
    vertice 2
      vertex 0
        edges 1
          edge0
            behav !0 -> 1 vertex 1
      vertex 1
        edges 1
          edge0
            behav ?1 -> 0
  net 3
    struct "Producer"
    behavs 1
       :0 "Q_put"&in(1,Max)
     logic "initial">0
    behav !0
    hook "automaton"
    vertice 1
      vertex 0
        edges 1
          edge0
            hook x=in(1,Max)
            behav !(0&x) -> 0
net 0
  struct _< 1,2&consumers,3&producers
  hook "synch_vector"
 behavs 3
   :0 "Q_get"&consumers
    :1 "R_get"&consumers
    :2 "Q_put"&(producers,in(1,Max))
  behav 0+1+2
   vertice 1
      vertex 0
       edges 3
         edge 0
           hook c=consumers
            behav 0&c < ?(0&c)$1, !0$(2&c) -> 0
          edge 1
           hook c=consumers
           behav 1&c < !(1&c)$1, ?1$(2&c) -> 0
         edge 2
           hook p=producers
           hook x=in(1,Max)
           behav 2&(p,x) < ?(2&x)$1, !(0&x)$(3&p) -> 0
```

As we mention before, if `consumers = {"cons1", "cons2"}` and `producers = [2, 4]`, then using the tool the expression `struct _< 1,2&consumers,3&producers` is expanded to `struct _< 1,2,2,3,3,3`. Then when `c="cons2"` and `p=3`, the vector `behav 0&c < ?(0&c)$1, !0$(2&c) -> 0` becomes `behav 1 < ?1,*, !0, *,*, * -> 0`.

2.7 Buffer's behaviour

The buffer's behaviour is described through a synchronisation network between two components: **stock** and **queue**. **stock** takes care of keeping the actual number of elements (up to Max) in the buffer's stock and receive the feeds from the producers. **queue** receives the requests from the consumers and put them in a queue. The answers to the consumers are given in FIFO order from the request queue until elements are available in the buffer's stock.

The network describing the buffer behaviour is shown in Figure 5.

The behaviour of the buffer is specified in a separate file because FC2Instantiate does not support nested synchronisation networks in a single file.

In the stock, its only state is parameterized with the variable **stock**. The variable **stock** encodes the quantity of elements actually available in stock. When an element is taken, this variable is decreased

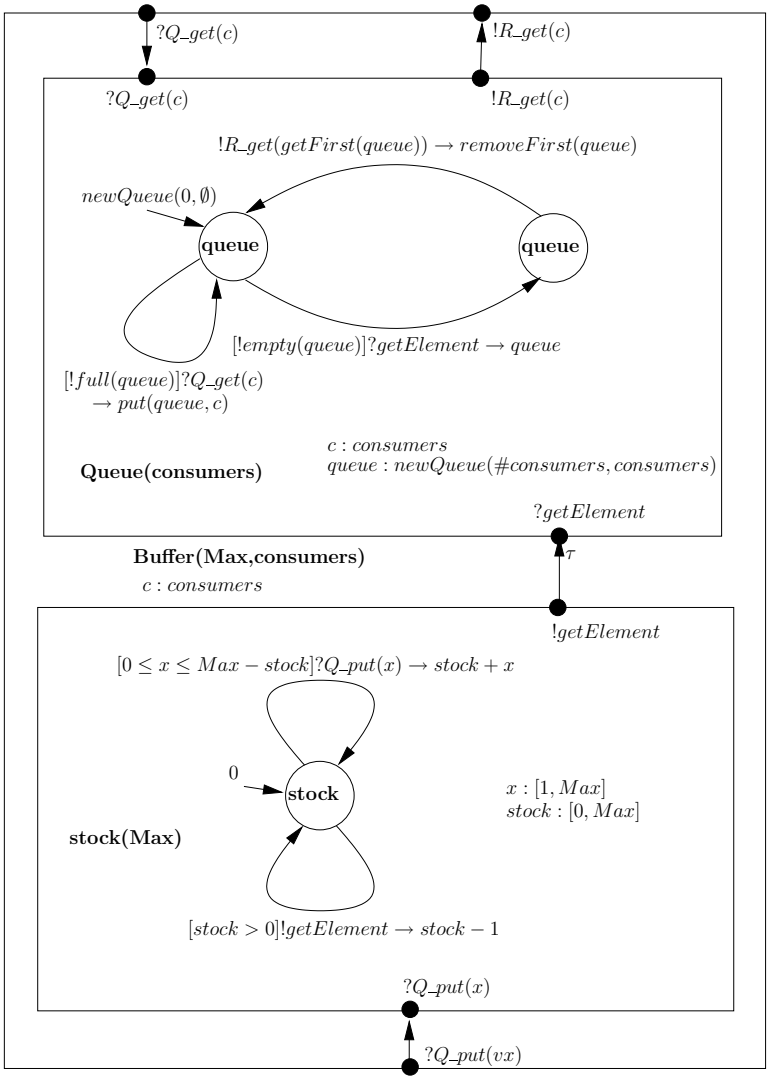

Figure 5: Parameterized Buffer

by 1, which is expressed by a transition to the state encoding the stock having one element less, in the figure correspond to the transition labelled $[stock > 0]!getElement \rightarrow stock - 1$. When feeding the stock (action Q_put) this variable is incremented by the quantity of elements received, transition labelled as $?Q\_put(x)$ ($[0 \leq x \leq Max - stock]?Q\_put(x) \rightarrow stock + x$ in the figure. Note that both transitions are guarded to avoid taken an element from an empty stock or to overfill it.

The FC2 file describing the stock is:

BufferParameterized.fc2

```
declarations
constant Max() ->any
constant stock() ->any
constant x() ->any
infix & (any any ) -> any priority 8
infix = (any any ) -> any priority 8
infix - (any any ) -> any priority 8
prefix in (any int) -> any
prefix greaterThan (any any) -> any
prefix when (any) -> any
...
   net 2
      struct "stock"
      behavs 2
         :0 "Q_put"&in(0,Max)
         :1 "getElement"
      logic "initial">0
      behav ?0+!1
      hook "automaton"
      vertice 1
         vertex 0
            hook stock=in(0,Max)
            edges 2
               edge0
                  hook x=in(1,(Max-stock))
                  behav ?(0&x) -> 0&(stock+x)
               edge1
                  hook when(greaterThan(stock,0))
                  behav !1 -> 0&(stock-1)
```

As you can see, new operators supported by the tool are introduced: the conditional operator `when`, the comparison operator `greaterThan`, and the arithmetic operators `-` and `+`.

In the queue, the states are parameterized by the state variable **queue** which encodes the states of a queue. A queue is characterised by its contents.

When a request for one element is received ($?Q\_get(c)$), the caller id (c) is appended to the end of the queue $put(queue, c)$. As soon as an element is available in the stock ($?getElement$), the first caller from the queue is taken ($getFirst(queue)$) and a response to it is given ($!R\_get(c)$). At the same time the caller is removed from the queue ($removeFirst(queue)$).

Figure 5 introduces several operators supported by the FC2Instantiate tool to manipulate queues. Their complete descriptions are given in section 5.1.

The FC2 file describing the Queue is:

BufferParameterized.fc2

```
declarations
constant consumers() ->any
constant c() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix = (any any ) -> any priority 8
prefix when (any) -> any
prefix instantiateQueue (any) -> any
prefix getFirst (any) -> any
prefix removeFirst (any) -> any
prefix fullQueue (any) -> any
prefix emptyQueue (any) -> any
prefix putQueue (any) -> any
prefix size (any) -> any
...
   net 1
```

```
      struct "queue"
      behavs 3
         :0 "Q_get"&consumers
         :1 "R_get"&consumers
         :2 "getElement"
      logic "initial">0
      behav ?0+!1+?2
      hook "automaton"
      vertice 2
         vertex 0
            hook queue=instantiateQueue(size(consumers),consumers)
            edges 2
               edge0
                  hook c=consumers
                  hook when(!fullQueue(queue))
                  behav ?(0&c) -> 0&putQueue(queue,c)
               edge1
                  hook when(!emptyQueue(queue))
                  behav ?2 -> 1&queue
         vertex 1
            hook queue=instantiateQueue(size(consumers),consumers)
            edges 1
               edge0
                  hook when(!emptyQueue(queue))
                  behav !(1&getFirst(queue)) -> 0&removeFirst(queue)
...
```

As specified in Figure 5, the complete Buffer behaviour is done by synchronising the *getElement* action of **stock** and of **queue** (the queue request elements from the stock). The full FC2 file describing this synchronisation is following:

BufferParameterized.fc2

```
declarations
constant Max() ->any
constant consumers() ->any
constant stock() ->any
constant x() ->any
constant c() ->any
constant queue() ->any
infix & (any any ) -> any priority 8
infix $ (any any ) -> any priority 8
infix = (any any ) -> any priority 8
infix - (any any ) -> any priority 8
prefix in (any int) -> any
prefix greaterThan (any any) -> any
prefix when (any) -> any
prefix instantiateQueue (any) -> any
prefix getFirst (any) -> any
prefix removeFirst (any) -> any
prefix fullQueue (any) -> any
prefix emptyQueue (any) -> any
prefix putQueue (any) -> any
prefix size (any) -> any
nets 3
   hook"main" > 0
   struct"Buffer"
   net 1
      struct "queue"
      behavs 3
         :0 "Q_get"&consumers
         :1 "R_get"&consumers
         :2 "getElement"
```

```
   logic "initial">0
   behav ?0+!1+?2
   hook "automaton"
   vertice 2
      vertex 0
         hook queue=instantiateQueue(size(consumers),consumers)
         edges 2
            edge0
               hook c=consumers
               hook when(!fullQueue(queue))
               behav ?(0&c) -> 0&putQueue(queue,c)
            edge1
               hook when(!emptyQueue(queue))
               behav ?2 -> 1&queue
      vertex 1
         hook queue=instantiateQueue(size(consumers),consumers)
         edges 1
            edge0
               hook when(!emptyQueue(queue))
               behav !(1&getFirst(queue)) -> 0&removeFirst(queue)
net 2
   struct "stock"
   behavs 2
      :0 "Q_put"&in(0,Max)
      :1 "getElement"
   logic "initial">0
   behav ?0+!1
   hook "automaton"
   vertice 1
      vertex 0
         hook stock=in(0,Max)
         edges 2
            edge0
               hook x=in(1,(Max-stock))
               behav ?(0&x) -> 0&(stock+x)
            edge1
               hook when(greaterThan(stock,0))
               behav !1 -> 0&(stock-1)
net 0
   behavs 3
      :0 "Q_get"&consumers
      :1 "R_get"&consumers
      :2 "Q_put"&in(1,Max)
   struct _< 1,2
   behav ?0+!1+?2
   hook "synch_vector"
   vertice 1
      vertex 0
      edges 4
        edge 0
          hook c=consumers
          behav ?(0&c) < ?(0&c)$1 -> 0
        edge 1
          hook c=consumers
          behav !(1&c) < !(1&c)$1 -> 0
        edge 2
          hook x=in(1,Max)
          behav ?(2&x) < ?(0&x)$2 -> 0
        edge 3
          behav tau < ?2$1, !1$2 -> 0
```

3 Instantiation File

The domain of the global variables, i.e. the variables visible all over the parameterized system definition, is defined in a instantiation file. The instantiation file is given in FC2 format having a single net. The variables are assigned using the operator = in the hooks of that net (they can be assigned equally to a set or a value). For our consumer-producer example, we instantiate the system with 2 producers, 2 consumers and a maximal buffer capacity of 3. The instantiation file is:

InstantiationDomains.fc2

```
declarations
constant Max() ->any
constant consumers() ->any
constant producers() ->any
infix = (any any ) -> any priority 8
prefix in (any int) -> any
prefix set (any) -> any
nets 1
   net 0
     hook Max=3
     hook consumers=set("cons1","cons2")
     hook producers=in(1,2)
```

4 Using the tool

The command to run FC2Instantiate is:

```
JAVA_CMD -cp FC2Instantiate.jar:FC2Parser.jar:jargs.jar\
fr.inria.oasis.fc2.FC2Instantiate [-o <net_file_output>]\
-d <definition.fc2> [-v] <instantiations.fc2>+
```

where JAVA_CMD is the Java runtime command and

- <net_file_output> is an optional file to print the result. If it is not given the result will be print to the standard output.
- <definition.fc2> is the file describing the parameterized system (in FC2 parameterized format)
- <instantiations.fc2>+ is a list of files defining the domain of the global variables. If a variable is defined in more than one file, it takes the value defined in the last file where it is defined.
- -v for extra information (debugging)

5 FC2Parameterized reference manual

While the FC2Parameterized format is not "human-friendly", it is a powerful language to describe behaviour of systems. Even when humans may directly use this language to model distributed system, its main target is to be the intermediate format produced by our automatic tools in order to do verification.

This section contains a preliminary version of the FC2Parameterized reference manual. It is expected that the tool (FC2Instantiate) and the format will evolve quickly in the short term, and a separate reference manual document will follow.

Additionally to the predefined operators in the FC2 format, the FC2Parameterized defines the following (all of them supported in the FC2Instantiate):

5.1 General Operators

- & : $Exp \times Exp \rightarrow Exp$, where Exp is an FC2 expression. Its semantic depends on the context where is used:
  - *In a semantic table.* The & operator, when inside a semantic table, takes a FC2 Expression in its left side and a list of sets (each set separate by ,) in its right side. This operator generates an entry in the semantic table for each member of the Cartesian product of the sets in its right side.
  - *In an automaton's edge.* The left parameter of the operator &, when used inside an automaton's edge (which is not a synchronisation vector), is a reference to an action in the corresponding semantic table (for instance, the behaviour label in the edge/vertex, reference the behaviour table of the net). The right side contains an expression. When using the tool, an expression of the form `0&x` (inside an automaton edge) will be replaced by a reference to the entry `0` of the semantic table (note that the entry in the table should be also parameterized by `x` or equivalent) for each instantiation of `x`.
  - *In a synchronisation vector.* When it is present at the left side of the $ operator, it semantic is the same that when is inside an automaton's edge (defined above). When is in the right side of the $ operator is analogous to the edge case, but it references to an evaluation of a net instead of a semantic table.
  - *In the struct label of a net.* In this case the & operator indicates the number of instantiations to be done for a net. In its left side is the referenced net, and in its right side a set. The number of instantiations of the nets will be the same as the number of elements of the set. For instance `struct _< 1,2&consumers,3&producers` indicates that the network is composed by one single instance of the network 1, #consumers instances of the network 2 (consumers is a variable encoding a set) and #producers instances of the network 3.
- $: $Exp \times Exp \rightarrow Exp$, where Exp is an FC2 expression. In the synchronisation vectors, the $ operator is used to indicate which of the instances of a network is being referenced.
- *when* : *boolean* is used in a hook to indicate a condition. When the condition is not true, the instantiation stops for the vertex/edge holding the hook.
- =: $varName \times Set$ is for assignment of values to a variable. When instantiating, the variable on the left side will be assigned to each element in the set of the right side.

5.2 Set operators

- *set* : $List(any) \rightarrow Set$. This operator receives a list of elements and constructs an ordered set containing them. Ex: `var=set("a",3,"b","other")` ($var = \{"a", 3, "b", "other"\}$).
- *in* : $int \times int \rightarrow Set$. The operator receives two integers and generates the set of all the integers between them inclusive. Ex: `var=in(2,5)` ($var = \{2, 3, 4, 5\}$).
- *size* : $Set \rightarrow int$. The size operator receives a set and returns the number of elements in it. Ex: `size(var)=4` ($var = \{"a", 3, "b", "other"\}$).
- *merge* : $Set \times Set \rightarrow Set$. The merge operator receives two sets and returns an unique set with the disjoint union of elements in both sets. Ex: `merge(set("a","b"),merge(in(1,2),set("b")))` ($= \{"a", "b", 1, 2, "b"\}$).

5.3 Queue Operators

Often in distributed systems, because their asynchronous communication nature, it is needed to model *queues*, sometimes named *channels*. FC2Parameterized provides the following set of queue manipulation:

- *instantiateQueue* : $Set \times int \rightarrow queue$. This operator receives a set of the potentially elements that could be added to the queue, and a maximal capacity of the queue. It generates the queue (queue is an internal type for FC2Parameterized). Ex: `queue=instantiateQueue(var,6)` ($var = \{"a", 3, "b", "other"\}$).

- *putQueue* : $queue \times any \rightarrow queue$. This operator receive a queue and an element to be added at the end of the queue. It returns the queue with the element already added. Ex: `queue=putQueue(queue,"newElement")`.

- *getFirst* : $queue \rightarrow any$. This operator returns the first element of a queue. Ex: `var=getFirst(queue)`.

- *getFirstFilter* : $queue \times Set \rightarrow any$. This operator returns the first element of a queue that is in the set *Set*. Ex: `var=getFirstFilter(queue,set("a",2))`.

- *removeFirst* : $queue \rightarrow queue$. This operator returns the queue *queue* without it first element. Ex: `queue=removeFirst(queue)`.

- *removeFirstFilter* : $queue \times Set \rightarrow queue$. This operator returns the queue *queue* without the first element in the queue which is also in the set *Set*. Ex: `queue=removeFirstFilter(queue,set("a",2))`.

- *fullQueue* : $queue \rightarrow boolean$. This operator checks whether the queue has reach the maximal capacity. Ex: `fullQueue(queue)`.

- *emptyQueue* : $queue \rightarrow boolean$. This operator checks whether the queue is empty. Ex: `emptyQueue(queue)`.

- *emptyQueueFilter* : $queue \times Set \rightarrow boolean$. This operator returns true if the queue has no elements from the set *Set*. It returns false otherwise. Ex: `emptyQueueFilter(queue,set("a",1))`.

- *hasElement* : $queue \times any \rightarrow boolean$. This operator returns true if the element *any* is in the queue *queue*. It returns false otherwise.

5.4 Arithmetic operators

- $+ : int \times int \rightarrow int$. Addition between integers. Ex: `var=3+var`.

- `^`: $int \times int \rightarrow int$. Power. Ex: `var^3` ($= var^3$).

- $- : int \times int \rightarrow int$. Subtraction between integers.

- $\times : int \times int \rightarrow int$. Multiplication between integers.

- $/ : int \times int \rightarrow int$. Integer division between integers. The result is the integer part of the division.

- *mod* : $int \times int \rightarrow int$. It gives the common residue of the arguments.

- *greaterThan* : $int \times int \rightarrow boolean$. It returns true if the first argument is greater than the second one, false otherwise.

- *lessThan* : $int \times int \rightarrow boolean$. It returns true if the first argument is less than the second one, false otherwise.

- *equal* : $int \times int \rightarrow boolean$. It returns true if the first argument is equal than the second one, false otherwise.