A BUS INSTRUMENTATION PROTOCOL, SPECIFIED IN LOTOS

P. AZEMA - K. DRIRA - F. VERNADAT

RAPPORT LAAS N° 90191

JUIN 1990

FORTE'90

A Bus Instrumentation Protocol, specified in LOTOS.

Pierre AZEMA, Khalil DRIRA, François VERNADAT
LAAS-CNRS
7 Avenue Colonel Roche, F-31077 Toulouse cedex.

Abstract

This paper analyzes the design of a protocol for distributed process control. A bus protocol is described, and the underlying broadcast mechanism is specified within LOTOS framework. The main point concerns the study of the service with respect to user requirements.

Global assertions on actual complex configurations are introduced in such a way that only simple cases have to be verified, that is a minimal finite configuration needs to be exhaustively checked.

From requirements expressed in natural language, several constraints on execution sequences (or temporal logic assertions) are derived. These requirements are analysed with respect to observationaly equivalent reduced models.

Introduction

The formal specification of a protocol for Factory Instrumentation is presented by means of Formal Description Technique LOTOS [LOT89].

The basic service consists of remote updating of values associated with declared identifiers. The protocol deals with the updating of distributed buffers by means of broadcast operations [FIP89]. This protocol is implemented at the data link layer (OSI layer 2).

Several stations are interconnected via a bus. The protocol makes use of a bus arbiter for synchronizing the exchanges of messages. This arbiter scans periodically every buffer.

A LOTOS specification of this protocol supplies a formal description. The purpose is to focus on the analysis, in such a way a formal verification is made possible.

A crucial step is the formal interpretation of user expectations [AVL89]. Here specific behaviours are translated into constraints on execution sequences, closely related to temporal logic assertions. These assertions are interpreted on the reachable state space of the system. To be manageable, the size of this state space has to be kept limited. This point results from invariant assertions valid whatever the number of processes. In this paper, only two single entities need to be considered as active, the so-called producer and consumer. Furthermore, to be understandable, the verification will be conducted on a reduced model. This model is derived from the complete behaviour by using observational equivalence [GS90, MV89]. This allows for comparing the expected behaviour, i.e. the service, with respect to the observed behaviour, derived from the complete protocol.

This paper reports on the way to deal with initial user requirement, and how it is possible to refine successively either the specification or the requirement itself.

The basic service is introduced in section I. The global system organization and some user requirements are first informally described. LOTOS definition of each process is given in Section II. The analysis of specifications is conducted in Section III, and several versions are considered. Section IV is devoted to test.

Section I. System Organisation.

Real time process control is the main application of protocol FIP. This environment involves sensors whose data value are to be periodically sampled and actuators whose command values are to be periodically updated. The specification of communication protocols which are depicted in this paper represent a part of specification proposed to UTE by french working group [FIP]. Several devices, sensors or actuators, are interconnected through a bus. Any value, as measured by a sensor, or as assigned by an actuator, is associated with an identifier.

Let ID be a set of identifiers. The value of an identifier has at most one producer, that is a device to supply the associated value, and at least one consumer, that a (possibly many) device to make use of this value. The communications are synchronized by a bus arbiter. The global

physical organisation is depicted by Figure 1. Any normal station has two access points: the so-called drop to access the bus, and the so-called link, as a service access point for the user.

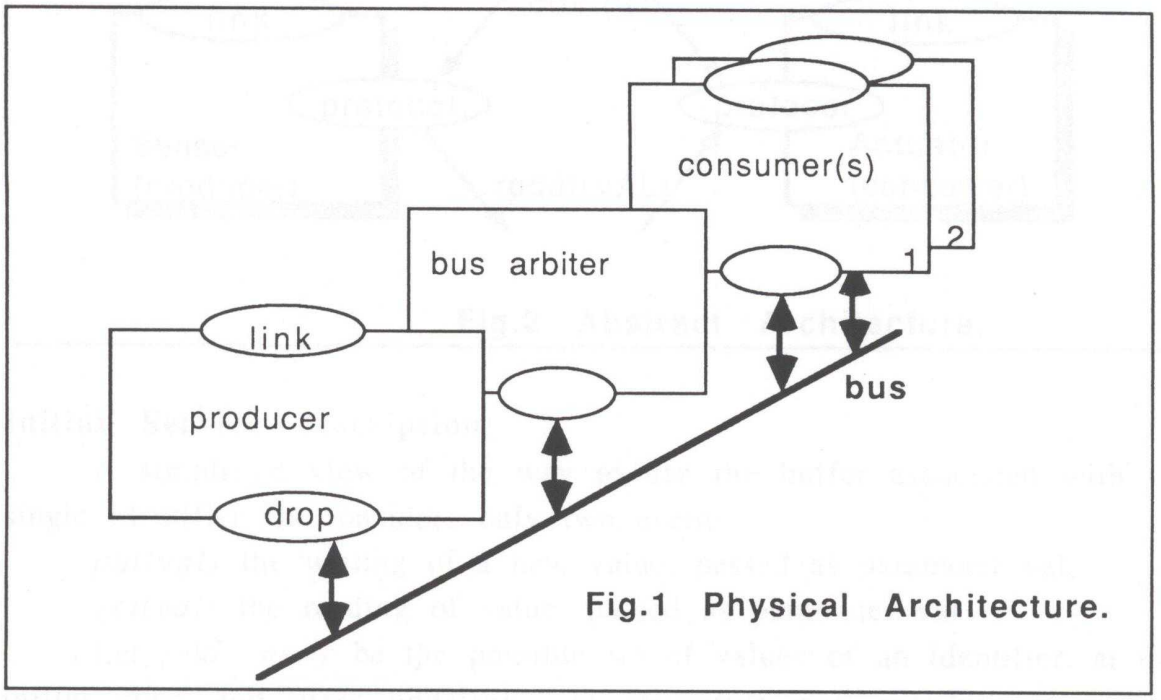

Fig.1 Physical Architecture.

Two main services are offered to users: either to put a value, or to get a value, associated with a specific identifier.

The user requests occur on interaction point *link*:
put(id, val), to write value val into a buffer associated with id, *get(id)*, to read value.

These primitives receive respective confirmations
confirm, confirm(val).

The underlying protocol is under the centralized control of a bus arbiter. This arbiter scans periodically identifiers ID, by issuing message *id-dat(ID)* for every identifier ID. The single producer becomes then ready to supply the value of the announced identifier and possibly many consumers become ready to read it. The broadcast of message *rp-dat(VAL)* performs this data exchange.

The relationship between the expected service and the implemented protocol may be interpreted on an abstract architecture, as depicted by Figure 2.

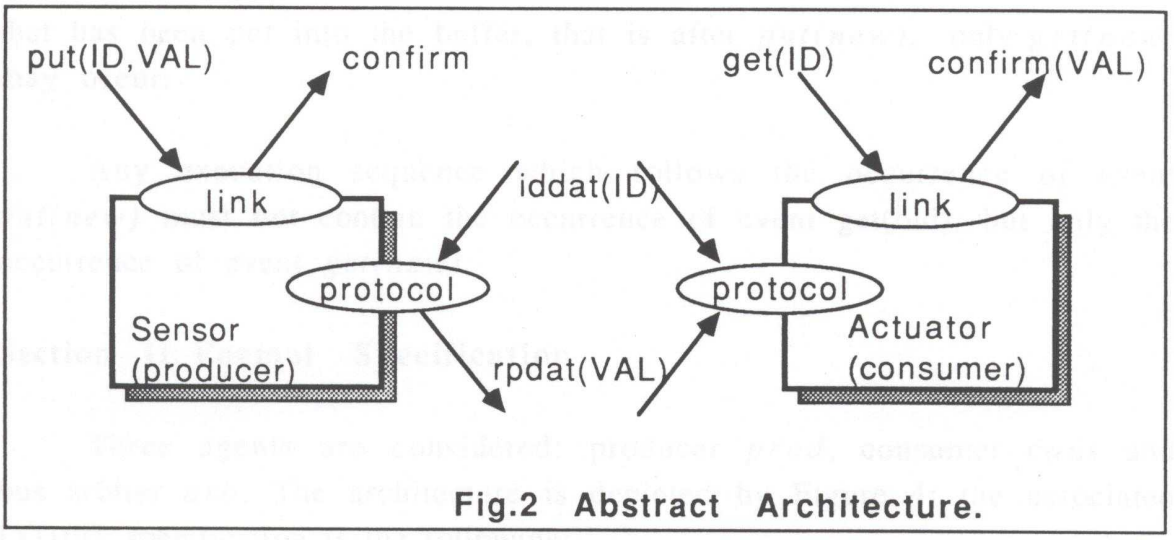

Fig.2 Abstract Architecture.

Initial Service description.

A simplified view of the way to use the buffer associated with a single identifier ID considers only two events:

put(val) the writing of a new value, passed as parameter val,

get(val) the reading of value, passed as parameter val.

Let *{old, new}* be the possible set of values of an identifier, at the initial state and after updating by primitive put, respectively. The value which results from the execution of primitive *get* is either *old*, from initial state 0, or *new*, after the occurrence of primitive *put*.

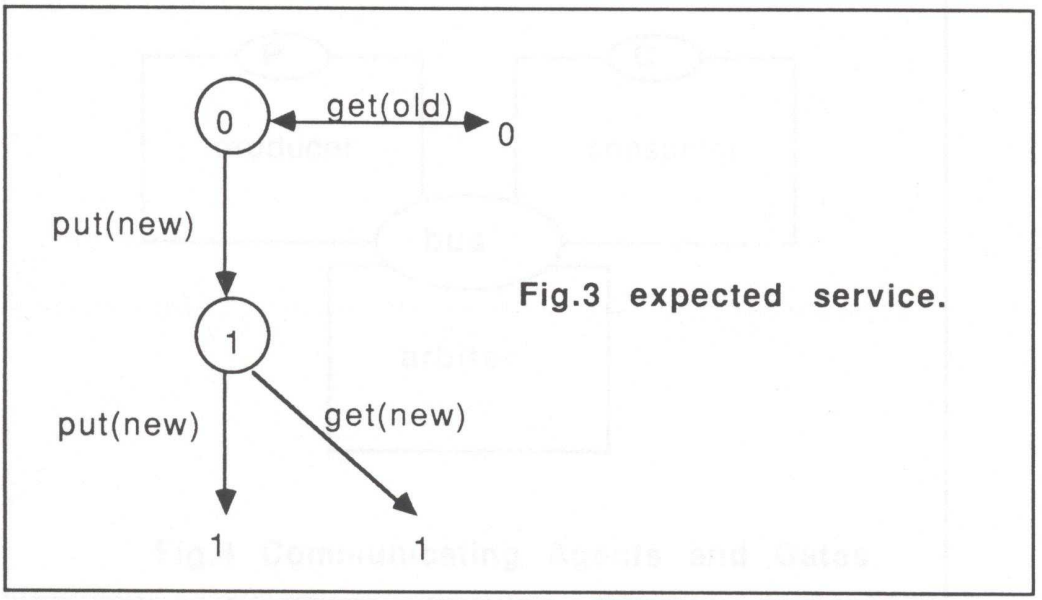

Fig.3 expected service.

The expected global behaviour may be depicted by Figure 3. In natural language, this behaviour may be phrased: you get the last value

that has been put into the buffer, that is after *put(new)*, only *get(new)* may occur.

Any execution sequence which follows the occurrence of event *put(new)* must not contain the occurrence of event get(old), but only the occurrence of event *get(new)*.

Section II. Formal Specification

Three agents are considered: producer *prod*, consumer *cons* and bus arbiter *arb*. The architecture is depicted by Figure 4; the associated LOTOS specification is the following:

prod[p,b](val)
|[b]|
cons[c,b](val)
|[b]|
arb[b](me)

A producer *prod*, with initial value val, has two gates {p, b}. The upper layer uses gate p, and lower level uses gate b. A consumer *cons*, with initial value val, has two gates {c, b}.

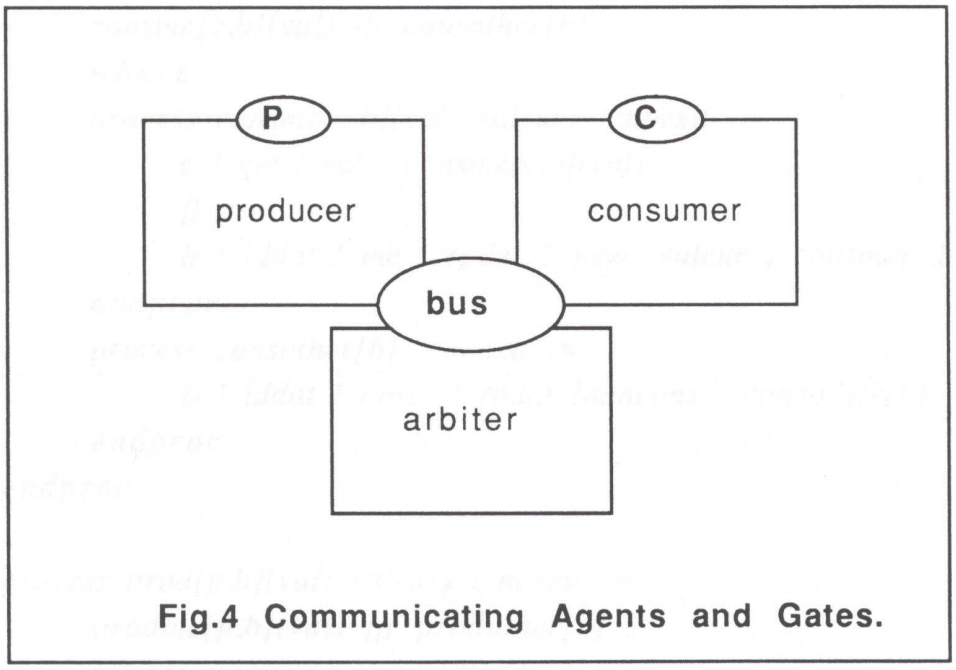

Fig.4 Communicating Agents and Gates.

The behaviour of the arbiter consists of cyclically scanning every identifier.

```
process arb[b](id : ident) : noexit :=
        b ! iddat ! id ! rpdat ? v : valeur; arb[b] (suc(id))
endproc
```

The general definition of function *suc* defines a cyclic permutation on the list of identifiers, in such a way each identifier will be successively considered.

However, from the point of view of a single identifier, only two cases have to be considered, whatever the number of identifiers. The behaviour of processes prod and cons are specified in LOTOS by considering mainly two cases: the arbiter wants to update either the identifier under concern, i.e. *me*, or another one, that is *other*.

In a similar manner, each agent may either perform a communication with upper layer, via primitive *put* (resp. *get*), or perform a communication with the bus. This decomposition is represented by the following interleaving of two processes: *consme* and *consother*, *prodme* and *prodother* on the other hand.

```
process cons[c,b](val: valeur) : noexit :=
        consme[c,b](val) ||| consother[b]
        where
        process consme[c,b](val: valeur) : noexit :=
                c ! get ! val ; consme[c,b](val)
                []
                b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
        endproc
        process consother[b] : noexit :=
                b ! iddat ! other ! rpdat !noncons ; consother[b]
        endproc
endproc

process prod[p,b](val: valeur) : noexit :=
        prodme[p,b](val) ||| prodother[b]
        where
        process prodme[p,b](val: valeur) : noexit :=
                p ! put ! new ; prodme[p,b](new)
```

```
        []
        b ! iddat ! me ! rpdat ! val ; prodme[p,b](val)
    endproc
    process prodother[b] : noexit :=
        b ! iddat ! other ! rpdat !noncons ; prodother[b]
    endproc
endproc
```

In the general case, type *valeur* may be integer.

However, for analysis purpose only two values will be considered: { *old, new*}.

Section III Analysis.

With respect to single identifier *me*, only two states of arbiter have to be considered: either the next identifier to be scanned is *me*, or the next is not me that is *other*. Consequently, the definition of type *ident* and operation *suc* is the following.

```
type ident is
    sorts ident
    opns
        me , other :-> ident
        suc : ident -> ident
    eqns
        ofsort ident
        suc(me) = other;
        suc(other) = me;
endtype
```

The three agents {*prod, cons, arb*} are composed by means of parallel operator */[b]/*, that is they are strongly synchronized with respect to bus events.

Process producer *prodme*, from initial state 0, may either perform action *p!put*, (exclusively) or action *b!me*, that is update by receiving *iddat(me)* followed by sending *rpdat(val)*. A transition system is associated with this behaviour.

On the left side of Figure 5, the local transition systems of producer and consumer are represented. From state 0, the occurrence of event (*b!me*) does not change the state, as far as the current data value is not considered.

This (finite) behaviour may be enumerated and reduced, by using algorithm for the derivation of minimal automaton with respect to observational equivalence: the observed events are only events which occur at gates *p* or *c*, that is *put* or *get*. The result of this analysis produces the automaton depicted by Figure 5.

On the right side of Figure 5, three events have been made visible: { *put, get(old), get(new)* }.

By inspection, it may be checked that it is possible to get the old value, even after operation put, that is from state 1, although the value written by operation put is a new value. That contradicts the requirement.

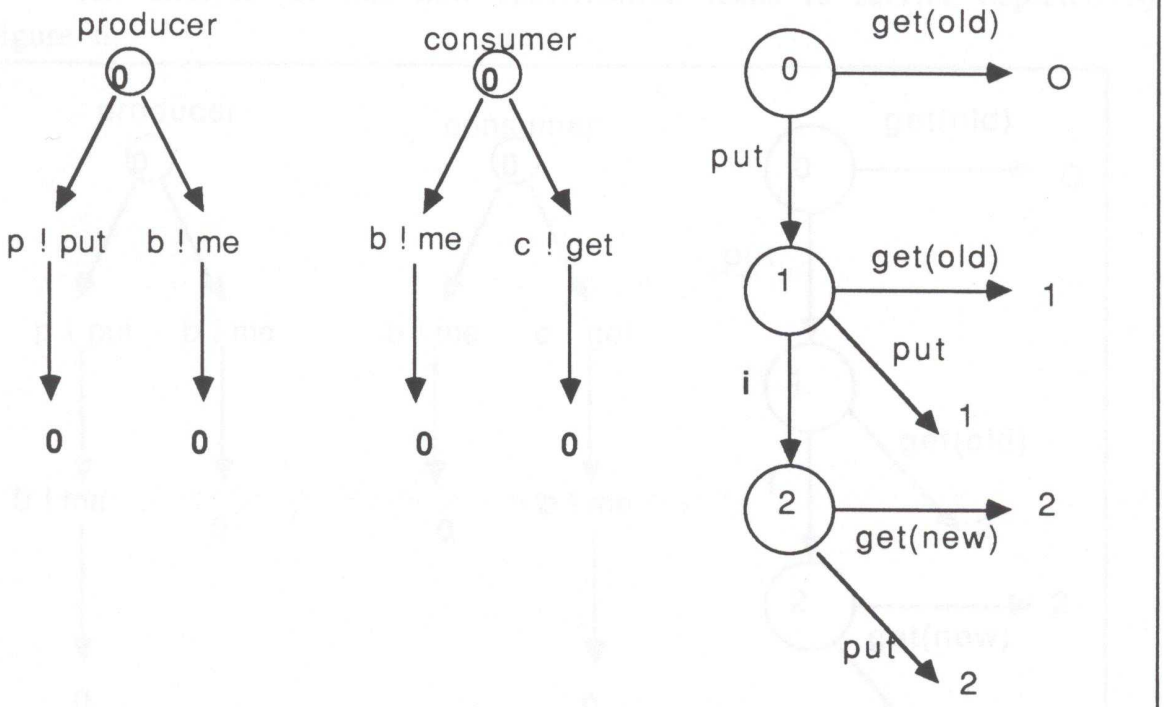

Fig.5 Protocol and service (first version).

A correction has to be made. A possible one is to enable operation *put* and *get* only if a transfer of value has been executed, that is to force a transfert of value after any *put* or *get* operation.

Only processes prodme and consme are modified, and a second version is now considered.

```
process consme[c,b](val: valeur) : noexit :=
        c ! get ! val ;
        b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
        []
        b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
    endproc
process prodme[p,b](val: valeur) : noexit :=
        p ! put ! new ; b ! iddat ! me ! rpdat ! new ;
        prodme[p,b](new)
        []
        b ! iddat ! me ! rpdat ! val ; prodme[p,b](val)
    endproc
```

The analysis of this new specification leads to service depicted by Figure 6.

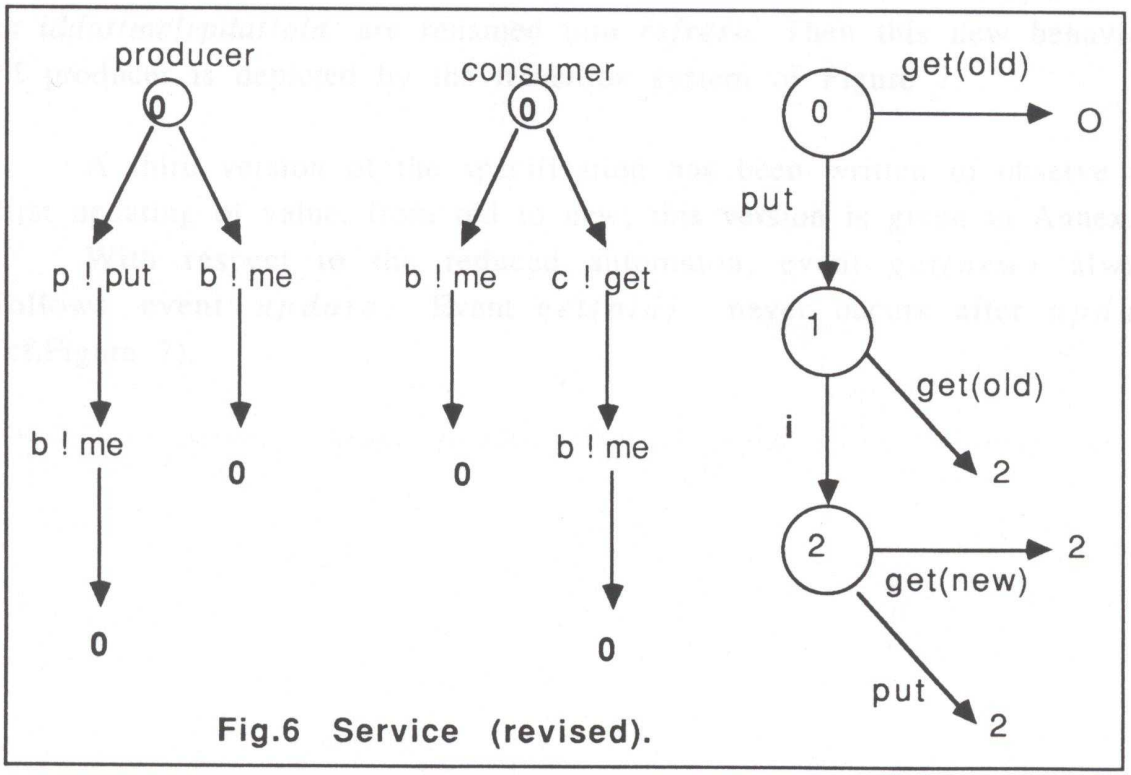

Fig.6 Service (revised).

The analysis of the observational equivalent automaton shows that after a put, that is from state 1, *get(old)* may occur at most once, because from state 2, only *get(new)* may occur.

The result is a better approximation of the expected service, however the service, as stated in section I, is not yet fulfilled.

In fact, the service as expressed in section I can not be implemented, as long as the data transfer is explicitely considered in the model but not in the requirement, because the actual exchange occurs by means of bus primitives. The relative ordering of occurrences of primitives *put* and *get* cannot be observed at two distinct locations, that is, on Figure 5, the transition from state 1 to state 2, labelled by event *get(old)*, does not implies that primitive *get* has been issued after primitive *put*. The ordering is valid only if the considered events occur on the same site, at a given level of abstraction.

The events occuring on the bus need to be observed, in particular with respect to the producer. Let the first occurrence of *iddat !me !rpdat !new* be renamed into *update*. Other occurrences of *iddat!me!rpdat!new* or *iddat!me!rpdat!old* are renamed into *refresh*. Then this new behaviour of producer is depicted by the transition system of Figure 7.

A third version of the specification has been written to observe the first updating of value, from old to new; this version is given in Annex.

With respect to the reduced automaton, event *get(new)* always follows event *update*. Event *get(old)* never occurs after *update* (cf.Figure 7).

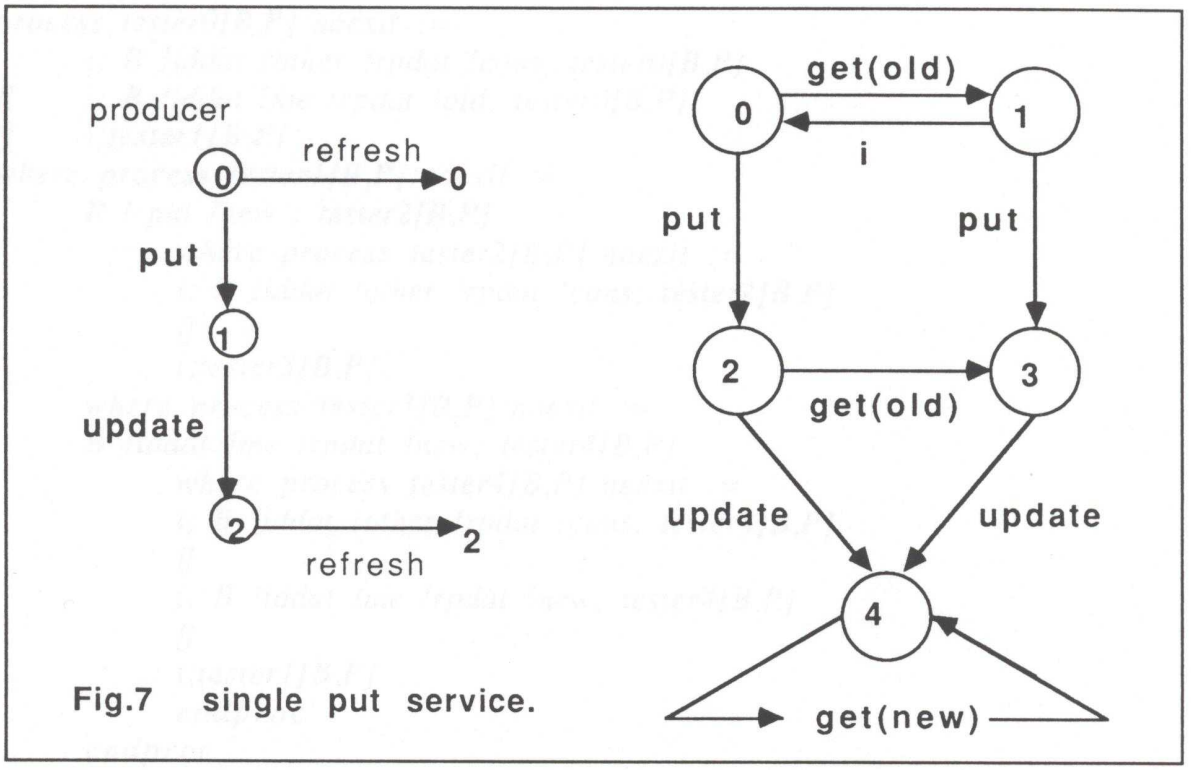

Fig.7 single put service.

Section IV. System Test.

This section deals with test generation following the methodology of refusal testing [Phil87] and more particularly conformance test introduced in [BR89]. Given a (finite) process S and its implementation I, the purpose is to verify that the implementation conforms to the specification.

(*I conf S*) whenever the following holds:
for every sequence σ potentially accepted by S (i.e σ in Trace(S)),
if S after σ can not refuse the action "*a*" then I after σ must accept "*a*" .

The idea is then to generate a process T(S) (called tester of S) which accepts the same language as S (in terms of external actions);
which performs the maximum of internal choice (*i*); and when composed in parallel , must never deadlock with I.

To illustrate the methodology, the tester of process *producer* is given. The unfolded behaviour has been obtained by using caesar [GS90]. The behaviours of Producer and its tester are represented by a transition system (Figure 8).

```
process tester0[B,P]:noexit :=
        i; B !iddat !other !rpdat !cons; tester0[B,P]
[]      i; B !iddat !me !rpdat !old; tester0[B,P]
[]      i;tester1[B,P]
where process tester1[B,P]:noexit :=
        P ! put !new ; tester2[B,P]
                where process tester2[B,P]:noexit :=
                i; B !iddat !other !rpdat !cons; tester2[B,P]
                []
                i;tester3[B,P]
        where process tester3[B,P]:noexit :=
        B !iddat !me !rpdat !new; tester4[B,P]
                where process tester4[B,P]:noexit :=
                i; B !iddat !other !rpdat !cons; tester4[B,P]
                []
                i; B !iddat !me !rpdat !new; tester4[B,P]
                []
                i;tester1[B,P]
                endproc
        endproc
endproc
endproc
```

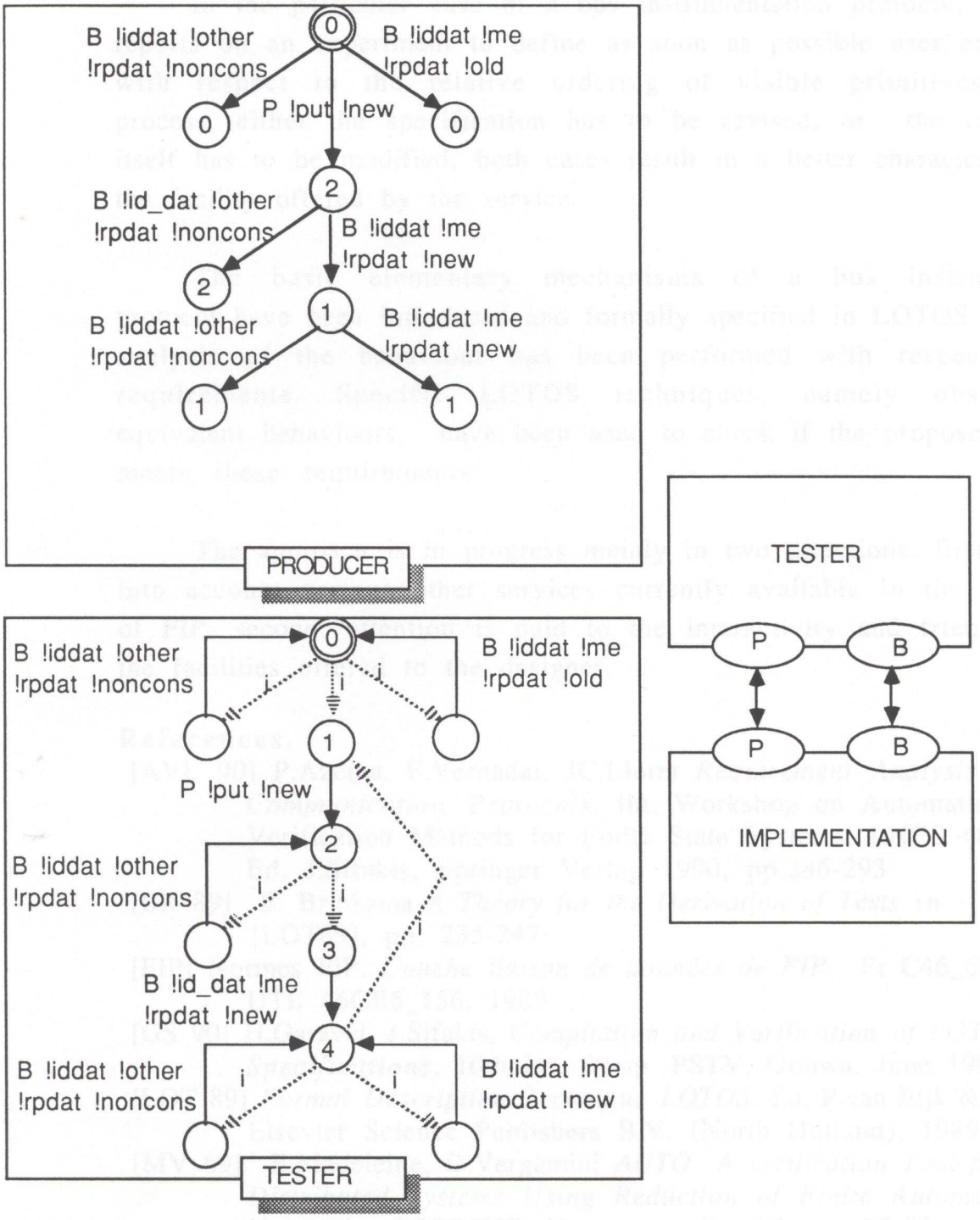

Figure 8. Producer and its tester.

Conclusion

In the particular case of a bus instrumentation protocol, this paper reports on an experiment to define as soon as possible user expectations with respect to the relative ordering of visible primitives. In this process, either the specification has to be revised, or the requirement itself has to be modified, both cases result in a better characterization of the facility offered by the service.

The basic elementary mechanisms of a bus instrumentation protocol have been introduced and formally specified in LOTOS. A careful analysis of the behaviour has been performed with respect to user requirements. Specific LOTOS techniques, namely observational equivalent behaviours, have been used to check if the proposed protocol meets these requirements.

The approach is in progress mainly in two directions: firstly to take into account various other services currently available in the framework of FIP; second, attention is paid to the interactivity and friendlyness of the facilities offered to the designer.

References.

[AVL 90] P.Azema, F.Vernadat, JC.Lloret *Requirement Analysis for Communication Protocols,* Int. Workshop on Automatic Verification Methods for Finite State Systems, LNCS 407, Ed. J.Sifakis, Springer Verlag 1990, pp.286-293

[BR 89] E. Brinksma *A Theory for the Derivation of Tests* in [LOT89], pp. 235-247

[FIP] Normes FIP: *Couche liaison de données de FIP*, Pr C46_603 UTE 46GE6_156, 1989

[GS 90] H.Garavel, J.Sifakis, *Compilation and Verification of LOTOS Specifications*, 10th Int. Symp. PSTV, Ottawa, June 1990.

[LOT 89] *Formal Description Technique LOTOS*. Ed. P.van Eijk & al, Elsevier Science Publishers B.V. (North Holland), 1989.

[MV 89] E.Madeleine, D.Vergamini *AUTO: A verification Tool for Distributed Systems Using Reduction of Finite Automata Networks,* FORTE'89, Vancouver, Dec.89, pp. 77-83.

[Phil 87] I. PHILIPS *Refusal Testing* Theorical Computer Science 50 (1987) 241-284, North-Holland.

Annex.1 LOTOS source

```
Specification fip0[p, c, b]:noexit

behaviour (*first specification*)
hide b in
(     prod[p,b](old)
      |[b]|
      cons[c,b](old)
      |[b]|
      arb[b](me) )
where
process arb[b](id : ident) : noexit :=
      b ! iddat ! id ! rpdat ? v : valeur;
            arb[b] (suc(id))
endproc


process cons[c,b](val: valeur) : noexit :=
      consme[c,b](val) ||| consother[b]
where
process consme[c,b](val: valeur) : noexit :=
      c ! get ! val ; consme[c,b](val)
      []
      b ! iddat ! me ! rpdat ? new: valeur ;
                  consme[c,b](new)
endproc
process consother[b] : noexit :=
      b ! iddat ! other ! rpdat !noncons ;
                  consother[b]
endproc
endproc



process prod[p,b](val: valeur) : noexit :=
      prodme[p,b](val) ||| prodother[b]
where
process prodme[p,b](val: valeur) : noexit :=
      p ! put ! new ; prodme[p,b](new)
      []
      b ! iddat ! me ! rpdat ! val ;
                  prodme[p,b](val)
endproc
process prodother[b] : noexit :=
      b ! iddat ! other ! rpdat !noncons ;
                  prodother[b]
endproc
endproc
endspec
```

```
specification fip1[p, c, b]:noexit

behaviour (*revised specification*
hide b in
(     prod[p,b](old)
      |[b]|
      cons[c,b](old)
      |[b]|
      arb[b](me) )
where
process arb[b](id : ident) : noexit :=
      b ! iddat ! id ! rpdat ? v : valeur;
                  arb[b] (suc(id))
endproc



process cons[c,b](val: valeur) : noexit :=
      consme[c,b](val) ||| consother[b]
where
process consme[c,b](val: valeur) : noexit :
      c ! get ! val ;
      b ! iddat ! me ! rpdat ? new: valeur ;
                  consme[c,b](new)
      []
      b ! iddat ! me ! rpdat ? new: valeur ;
                  consme[c,b](new)
endproc
process consother[b] : noexit :=
      b ! iddat ! other ! rpdat !noncons ;
                  consother[b]
endproc
endproc

process prod[p,b](val: valeur) : noexit :=
      prodme[p,b](val) ||| prodother[b]
where
process prodme[p,b](val: valeur) : noexit
      p ! put ! new ; b ! iddat ! me ! rpdat ! r
                  prodme[p,b](nev
      []
      b ! iddat ! me ! rpdat ! val ;
                  prodme[p,b](val
endproc
process prodother[b] : noexit :=
      b ! iddat ! other ! rpdat !noncons ;
                  prodother[b]
endproc
endproc
endspec
```

```
specification fip3[p, c, bu, br, bo]:noexit
      (* bupdate,brefresh, bother *)
type prim is
      sorts prim
      opns iddat, rpdat, put, get , conf:-> prim
endtype
type valeur is
      sorts valeur
      opns old, new, noncons  :-> valeur
endtype
type ident is
      sorts ident
      opns        me, other :-> ident
                  suc: ident -> ident
      eqns ofsort ident
            suc(me) = other;
            suc(other) = me;
endtype
behaviour (* only update is visible *)
hide br,bo in
(     prod[p,bu,br,bo](old)     |[bu,br,bo]| cons[c,bu,br,bo](old)
                                |[bu,br,bo]  arb[bu,br,bo](me) )
where
process arb[bu,br,bo](id : ident) : noexit :=
      bu ! iddat ! id ! rpdat ? v : valeur; arbo[bu,br,bo] (suc(id))
      []
      br ! iddat ! id ! rpdat ? v : valeur; arbo[bu,br,bo] (suc(id))
endproc
process arbo[bu,br,bo](id : ident) : noexit :=
      bo ! iddat ! id ! rpdat ? v : valeur; arb[bu,br,bo] (suc(id))
endproc

process cons[c,bu,br,bo](val: valeur) : noexit :=
      consme[c,bu,br](val)  |||  consother[bo]
where
process consme[c,bu,br](val: valeur) : noexit :=
c ! get ! val ;
      (bu ! iddat ! me ! rpdat ! new ;
consnew[c,br]
      []
      br  ! iddat ! me ! rpdat ! old ;
consme[c,bu,br](val))
[](
      bu ! iddat ! me ! rpdat ! new ; consnew[c,br]
      []
      br ! iddat ! me ! rpdat ! old ;
                  consme[c,bu,br](val)  )
endproc
process consnew[c,br] : noexit :=
c ! get ! new ; br ! iddat ! me ! rpdat ! new ;
                  consnew[c,br]
      []
      br ! iddat ! me ! rpdat ! new ; consnew[c,br]
endproc
process consother[bo] : noexit :=
      bo ! iddat ! other ! rpdat !noncons ;
                  consother[bo]
endproc
endproc

process prod[p,bu,br,bo](val: valeur) : n
:=
      prodme[p,bu,br](val)  |||  prodother[b
where
process prodme[p,bu,br](val: valeur) : n
:=
p ! put ! new ; bu ! iddat ! me ! rpdat ! new
                  prodnew[br](new)
      []
      br ! iddat ! me ! rpdat ! val ;
                  prodme[p,bu,br](val)
endproc

process prodnew[br](val: valeur) : noexit
      br ! iddat ! me ! rpdat ! val ;
                  prodnew[br](val)
endproc

process prodother[bo] : noexit :=
      bo ! iddat ! other ! rpdat !noncons ;
                  prodother[bo]
endproc
endproc
endspec
```

```
Received: from imag.imag.fr by bauges.imag.fr (4.0/5.17)
        id AA07392; Thu, 12 Jul 90 11:11:38 +0200
Received: by imag.imag.fr (5.54/5.17)
        id AA17193; Thu, 12 Jul 90 11:12:03 +0200
Received: from inria.inria.fr by mirsa.inria.fr with SMTP
        (5.61+++/IDA-1.2.8) id AA07988; Thu, 12 Jul 90 11:11:54 +0200
Received: by inria.inria.fr (5.61+/89.0.8)
        via Fnet-EUnet id AA08970; Wed, 11 Jul 90 16:49:45 +0200 (MET)
From: khalil@gina.laas.fr
Received: from gina.laas.fr by laas.laas.fr, Wed, 11 Jul 90 16:40:09 +0200
Return-Receipt-To: khalil@gina.laas.fr
Received: by gina.laas.fr, Wed, 11 Jul 90 16:40:42 +0200
Date: Wed, 11 Jul 90 16:40:42 +0200
Message-Id: <9007111440.AA11513@gina.laas.fr>
To: hubert@bauges
Subject: FIPLOTOS
Cc: hubert@imag.imag.fr
Status: R

Salut hubert,

Je t'envoie par mail le sources lotos de fip. Nous avons essaye de
simplifier au naximum les specs pour qu'elles soient comprehensibles.
Je t'ai envoye aussi l'article par poste. Il y a trois specs fip0.lotos,
fip1.lotos et fip3.lotos (et leur fip.h commun). Ils correspondent
aux trois categories de service analysees dans l'article.

Tes remarques et corrections sont les bien venues.

                                        Amicalement
                                          khalil
------------------------------fip0.lotos-----------------------------
specification npac[p, c, b]:noexit

type prim is
        sorts prim (*! implementedby PRIM comparedby CMP_PRIM
                                    printedby PRINT_PRIM *)
        opns
        iddat (*! implementedby IDDAT *),
        rpdat (*! implementedby RPDAT *),
        put   (*! implementedby PUT *),
        get   (*! implementedby GET *),
        conf  (*! implementedby CONF *) :-> prim
endtype
type valeur is
        sorts valeur (*! implementedby VALEUR comparedby CMP_VALEUR
                         printedby PRINT_VALEUR *)
        opns
                old (*! implementedby OLD constructor *),
                new (*! implementedby NEW constructor *),
                noncons (*! implementedby NONCONS constructor *) :-> valeur
                succ   (*! implementedby SUCC *) :valeur -> valeur
        eqns
                ofsort valeur
                succ(new) = old;
                succ(old) = new;
endtype

type ident is
        sorts ident (*! implementedby IDENT comparedby CMP_IDENT
                                            printedby PRINT_IDENT *)
        opns
                me    (*! implementedby ME *),
                other (*! implementedby OTHER *) :-> ident
                suc   (*! implementedby SUC *) :ident -> ident
        eqns
                ofsort ident
                suc(me) = other;
                suc(other) = me;
endtype

behaviour
hide b in
(       prod[p,b](old)
        |[b]|
        cons[c,b](old)
        |[b]|
        arb[b](me)      )
where

process arb[b](id : ident) : noexit :=
        b ! iddat ! id ! rpdat ? v : valeur; arb[b] (suc(id))
endproc

process cons[c,b](val: valeur) : noexit :=
        consme[c,b](val) ||| consother[b]
where

process consme[c,b](val: valeur) : noexit :=
        c ! get ! val ; consme[c,b](val)
        []
        b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
endproc
```

```
process consother[b] : noexit :=
        b ! iddat ! other ! rpdat !noncons ; consother[b]
endproc
endproc

process prod[p,b](val: valeur) : noexit :=
        prodme[p,b](val) ||| prodother[b]
where

process prodme[p,b](val: valeur) : noexit :=
        p ! put ! new ; prodme[p,b](new)
        []
        b ! iddat ! me ! rpdat ! val ; prodme[p,b](val)
endproc

process prodother[b] : noexit :=
        b ! iddat ! other ! rpdat !noncons ; prodother[b]
endproc
endproc

endspec
------------------------------fip1.lotos-----------------------------
specification atom[p, c, b]:noexit

type prim is
        sorts prim (*! implementedby PRIM comparedby CMP_PRIM
                                    printedby PRINT_PRIM *)

        opns
        iddat (*! implementedby IDDAT *),
        rpdat (*! implementedby RPDAT *),
        put   (*! implementedby PUT *),
        get   (*! implementedby GET *),
        conf  (*! implementedby CONF *) :-> prim
endtype
type valeur is
        sorts valeur (*! implementedby VALEUR comparedby CMP_VALEUR
                         printedby PRINT_VALEUR *)

        opns
                old (*! implementedby OLD constructor *),
                new (*! implementedby NEW constructor *),
                noncons (*! implementedby NONCONS constructor *) :-> valeur
                succ   (*! implementedby SUCC *) :valeur -> valeur

        eqns
                ofsort valeur
                succ(new) = old;
                succ(old) = new;
endtype

type ident is
        sorts ident (*! implementedby IDENT comparedby CMP_IDENT
                                            printedby PRINT_IDENT *)

        opns
                me    (*! implementedby ME *),
                other (*! implementedby OTHER *) :-> ident
                suc   (*! implementedby SUC *) :ident -> ident

        eqns
                ofsort ident
                suc(me) = other;
                suc(other) = me;
endtype

behaviour
hide b in
(       prod[p,b](old)
        |[b]|
        cons[c,b](old)
        |[b]|
        arb[b](me)      )
where

process arb[b](id : ident) : noexit :=
        b ! iddat ! id ! rpdat ? v : valeur; arb[b] (suc(id))
endproc

process cons[c,b](val: valeur) : noexit :=
        consme[c,b](val) ||| consother[b]
where

process consme[c,b](val: valeur) : noexit :=
        c ! get ! val ;
        b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
        []
        b ! iddat ! me ! rpdat ? new: valeur ; consme[c,b](new)
endproc

process consother[b] : noexit :=
        b ! iddat ! other ! rpdat !noncons ; consother[b]
endproc
endproc

process prod[p,b](val: valeur) : noexit :=
        prodme[p,b](val) ||| prodother[b]
where
```

```
rocess prodme[p,b](val: valeur) : noexit :=
       p ! put ! new ; b ! iddat ! me ! rpdat ! new ;
       prodme[p,b](new)
       []
       b ! iddat ! me ! rpdat ! val ; prodme[p,b](val)
endproc

process prodother[b] : noexit :=
       b ! iddat ! other ! rpdat !noncons ; prodother[b]
endproc
endproc

endspec
------------------------------fip3.lotos-----------------------------
specification fip3[p, c, bu, br, bo]:noexit (* bupdate,brefresh, bother *)

type prim is
       sorts prim (*! implementedby PRIM comparedby CMP_PRIM
                              printedby PRINT_PRIM *)
       opns
       iddat (*! implementedby IDDAT *),
       rpdat (*! implementedby RPDAT *),
       put   (*! implementedby PUT *),
       get   (*! implementedby GET *),
       conf  (*! implementedby CONF *) :-> prim
endtype
type valeur is
       sorts valeur (*! implementedby VALEUR comparedby CMP_VALEUR
                        printedby PRINT_VALEUR *)
       opns
              old (*! implementedby OLD constructor *),
              new (*! implementedby NEW constructor *),
              noncons (*! implementedby NONCONS constructor *) :-> valeur
              succ   (*! implementedby SUCC *) :valeur -> valeur
       eqns
              ofsort valeur
              succ(new) = old;
              succ(old) = new;
endtype

type ident is
       sorts ident (*! implementedby IDENT comparedby CMP_IDENT
                                    printedby PRINT_IDENT *)
       opns
              me     (*! implementedby ME *),
              other (*! implementedby OTHER *) :-> ident
              suc    (*! implementedby SUC *) :ident -> ident
       eqns
              ofsort ident
              suc(me) = other;
              suc(other) = me;
endtype

behaviour
hide br,bo in
(      prod[p,bu,br,bo](old)
       |[bu,br,bo]|
       cons[c,bu,br,bo](old)
       |[bu,br,bo]|
       arb[bu,br,bo](me)       )
where

process arb[bu,br,bo](id : ident) : noexit :=
       bu ! iddat ! id ! rpdat ? v : valeur; arbo[bu,br,bo] (suc(id))
       []
       br ! iddat ! id ! rpdat ? v : valeur; arbo[bu,br,bo] (suc(id))
endproc

process arbo[bu,br,bo](id : ident) : noexit :=
       bo ! iddat ! id ! rpdat ? v : valeur; arb[bu,br,bo] (suc(id))
endproc

process cons[c,bu,br,bo](val: valeur) : noexit :=
       consme[c,bu,br](val) ||| consother[bo]
where

process consme[c,bu,br](val: valeur) : noexit :=
       c ! get ! val ;
              (bu ! iddat ! me ! rpdat ! new ; consnew[c,br]
              []
              br  ! iddat ! me ! rpdat ! old ; consme[c,bu,br](val))
       [](
              bu ! iddat ! me ! rpdat ! new ; consnew[c,br]
              []
              br ! iddat ! me ! rpdat ! old ; consme[c,bu,br](val) )
endproc

process consnew[c,br] : noexit :=
       c ! get ! new ; br ! iddat ! me ! rpdat ! new ; consnew[c,br]
       []
       br ! iddat ! me ! rpdat ! new ; consnew[c,br]
endproc

process consother[bo] : noexit :=
```

```
        bo ! iddat ! other ! rpdat !noncons ; consother[bo]
endproc
endproc

process prod[p,bu,br,bo](val: valeur) : noexit :=
        prodme[p,bu,br](val) ||| prodother[bo]
where

process prodme[p,bu,br](val: valeur) : noexit :=
        p ! put ! new ; bu ! iddat ! me ! rpdat ! new ; prodnew[br](new)
        []
        br ! iddat ! me ! rpdat ! val ; prodme[p,bu,br](val)
endproc

process prodnew[br](val: valeur) : noexit :=
        br ! iddat ! me ! rpdat ! val ; prodnew[br](val)
endproc

process prodother[bo] : noexit :=
        bo ! iddat ! other ! rpdat !noncons ; prodother[bo]
endproc
endproc

endspec
-------------------------------fip.h---------------------------------
typedef enum {iddat, rpdat, put, get, conf} PRIM;
#define IDDAT() iddat
#define RPDAT() rpdat
#define PUT() put
#define GET() get
#define CONF() conf
#define CMP_PRIM(T1,T2) ((T1) == (T2))
char *TEXT[] = {"iddat","rpdat","put","get","conf"};
#define PRINT_PRIM(F,T) fprintf (F, TEXT[(T)])

typedef unsigned char VALEUR;
#define OLD() 0
#define NEW() 1
#define NONCONS() 2
#define CMP_VALEUR(X1,X2) ((X1) == (X2))
#define ENUM_VALEUR(X) for ((X) = OLD(); (X) <= NEW(); ++(X))
#define SUCC(T) (((T)==0) ? 1 : 0)
char *TEXTVAL[] = {"old","new","noncons"};
#define PRINT_VALEUR(F,T) fprintf (F, TEXTVAL[(T)])

typedef enum {me, other} IDENT;
#define ME() me
#define OTHER() other
#define CMP_IDENT(T1,T2) ((T1) == (T2))
#define SUC(T) (((T)==me) ? other : me)
#define PRINT_IDENT(F,T) fprintf (F, ((T)==me) ? "me" : "other")


-------------------------------FIN DES SOURCES-----------------------

N.B. IL ne manque pas fip2.lotos


(Il se peut que tu recoive 2 fois ce meme message)
```