Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Multi-way to Two-way Synchronization

Chapter 8 of
"Catalogue of LOTOS Correctness Preserving Transformations"
ESPRIT Project 2304 LOTOSPHERE
task 1.2 Third Deliverable

A. Fantechi

Nota Interna B4-29
Luglio 1992

Chapter 8

Multi-way to Two-way Synchronization

8.1 Informal Description

A process P, in which an action on a gate a can be simultaneously performed by more than two subprocesses, is transformed in an "equivalent" process Q in which each action is performed at most by two subprocesses. That is, the degree of synchronization associated to QD should be at most 2. A variant of this problem imposes a bound on the synchronization degrees relative to a predefined subset of gates. Actually, the formal description of the problem and most of the solutions refer to a single multi-way gate present in P.

8.2 Motivation

In many concurrent programming languages it is not allowed to perform multi-way synchronization within a single communication construct, but only two-way communications (we intend with the term *two-way communication* a communication between two partners). It is then likely that specifications developed in the latest phases of the design trajectory will contain only two-way communications. Moreover, this transformation is useful when passing from constraint oriented specifications (where multi-way synchronizations are the norm) to resource oriented specifications.

8.3 Formal Description

8.3.1 Auxiliary Concepts

Let LTS(B) be the Labelled Transition Systems associated to the behaviour expression B; let ELTS(B) be the Labelled Transition Systems obtained by LTS(B) by substituting a label a by the label $< a, n >$, where n is the number of subexpressions which synchronize on the action a. Given a process definition PD, defined by a behaviour expression BE, we define $SyncDegree(PD) = max\{n | B \overset{<a,n>}{\rightarrow} B' \textit{ is a transition of } ELTS(BE)\}$ [7].

8.3.2 Input

PD: a process definition constituted by a finite number of processes (P1, P2,.. Pn) composed in parallel and all synchronizing on the gate a.

8.3.3 Output

QD: a process definition constituted by a finite number of processes (Q1, Q2,.. Qn, S1, S2,.. Sm) composed in parallel and synchronizing among them using gates in the set $\{a1, ..ak\}$.

Each of the Qi differs from the corresponding Pi process of PD only for what concern the communications on the ai gates, replacing the communications on the gate a of PD; the Si processes are new "service" processes introduced by the transformation: they communicate only on gates in the set $\{a1, ..ak\}$.

8.3.4 Transformation Requirements

$SyncDegree(QD) \leq 2$.

8.3.5 Correctness Preservation Requirements

hide a in PDB is observationally equivalent to *hide a1, ..., an in QDB* , where PDB, QDB are the behaviour expressions defining PD, QD.

In the formulation of this correctness preservation requirement we have considered the fact that this transformation is usually applied to a set of processes communicating on a gate a; such multi-way communication is substituted by communications on a set of gates shared by the internal processes. If the transformed gates are not visible outside the

specifications, the given correctness preservation requirement reduces to weak equivalence between PDB and QDB.

Note that the above requirement is preserved also by the transformation which, starting from a behaviour expression strongly equivalent to:

```
a;c;stop [] b;stop
```

produces a behaviour expression strongly equivalent to:

```
i; a1; a2; c; stop [] b;stop  .
```

The solutions proposed will not introduce this kind of internal nondeterminism; however, this gives the idea that the current definition of the Correctness Preserving Relation is too weak; a better definition should use an equivalence notion based on the refinement of actions (see, for example, [26]), but we feel that the current state of the art in action refinement does not give a satisfactory opportunity to improve our definition.

8.4 Example

PD:

```
process P[a]  : noexit  :=
...
a !endtransaction; exit
|[a]|
(SlaveTrans_1[a] |[a]| SlaveTrans_2[a] |[a]| SlaveTrans_3[a])
...
where

 process SlaveTrans_1[a]  : noexit  :=
....
a ?x : signal-sort ;
....
endproc

process SlaveTrans_2[a]  : noexit  :=
....
a ?x : signal-sort ;
....
endproc
```

```
process SlaveTrans_3[a]  : noexit  :=
....
a ?x : signal-sort ;
....
endproc

endproc
```

QD:

```
process Q[a]  : noexit  :=
...
a !endtransaction; a !endtransaction; a !endtransaction;
exit
|[a]|
( SlaveTrans_1[a] ||| SlaveTrans_2[a] ||| SlaveTrans_3[a] )
...
where
...
endproc
```

(in this case it is even not necessary to split the gate a in three different gates; nevertheless, the communication occurs only between pairs of processes).

8.5 Solutions

The solutions presented are given for different subsets of the formally described problem. Nevertheless, it seems that some of them can be generalized to cope with no limitation. On the other hand, it is evident that less limitations bring more complex solutions.

To give general guidelines to help the transformer in selecting the more convenient solution, we should consider that the solution to this problem is heavily influenced by the following issues:

- 1) the multiway communication may imply an agreement between the processes on the values to be passed (the nondeterministic choice implicit in the question mark should be resolved);

- 2) an agreement on performing a particular action among the possible actions for the processes may be implied; this is the case when the multi-way gate a is present in a nondeterministic choice;

- 3) moreover, we can distingush two cases when a is present in a nondeterministic choice: a) this agreement is limited to the set of processes interacting on a, and, transitively, on gates present in the nondeterministic choice (the precise definition of the set of such gates, called $G_{min}(a)$, is given by means of the transitive closure on the alternatives to a in the nondeterministic choice - see solution e below); in this case, no intervention from this group of processes is possible, that is, a properly placed *hide* $G_{min}(a)$ construct embeds the parallel composition of the interested processes; b) the agreement is open to processes from outside, that is, gates in the $G_{min}(a)$ set are not hidden.

- 4) the multi-way communication may be used with the purpose of synchronizing all the processes involved in the communication; that is, after an action on the gate a the processes assume that all the partners have reached a certain point in their execution. This is always the case when an agreement on the actions (see point 2 above) is implied, but this use of multi-way synchronization can be done also when no agreement on actions is involved; in this case the action on a should be considered as an atomic action; in the following, we will call this requirement *complete synchronization* of the processes;

- 5) the agreement among the participating processes may be realized by a centralized entity, or distributed among the partners; this choice should be done on the basis of architectural considerations.

For these reasons we have preferred to propose six different solutions, ranging from the simplest case to the most complex one, implementing different kinds of agreements and employing different techniques, so as to show their potential in matching the desires of the transformer.

The two distributed solutions are based on a ring arrangement of the partner processes. This, together with the fact that the processes configuration is static, permits to base the solutions on a simple basic algorithm which involves a round of token passing over a ring. The completion of a round gives to the starter process the knowledge of the state of the partner processes, on which it can base its decisions.

Solutions for different topological arrangements of partner processes may be derived by substituting the basic ring algorithm with a "total" algorithm (that is, an algorithm giving to a process a complete knowledge of the state of the partners) suitable for that topology (see [45]).

In conclusion, architectural choices and the criteria listed before (agreement on values, internal/external agreement on choice) have first to be assessed for the problem at hand, in order to guide the selection of the most convenient solution.

The presented cases assume that there is no mixing of question and exclamation marks in the same action prefix; solutions c,d,e,f can however be generalized in this sense.

Obviously, in all these transformations the process functionalities and the set of gates on which the processes synchronize must be rearranged accordingly with the transformation; we have not described such rearrangement.

a) broadcast

A multi-way gate can be used to model broadcast communication: there is a single "output" command *a!value* in the "sender process" and several "input" commands *a?x:t* , one for each "receiving" process, as in the example before (Figure 8.1).

Condition: the input/output commands on *a* should not occur in a choice.

Solution: substitute `a!value` in the sender process with the sequence: `a1!value; ...; an!value; a1!ack; ...; an!ack` and substitute in each receiving process Ri the command `a?x:t` with `ai?x:t; ai!ack`.

Usually, in the case of broadcast communication complete synchronization is not required, since a receiver process is not interested in the state of the other receiver processes. We can therefore use the following simpler solution.

Solution: substitute `a!value` in the sender process with the sequence: `a1!value; ...; an!value` and substitute in each receiving process Ri the command `a?x:t` with `ai?x:t`.

However, note that the simpler solution does not satisfy the correctness preservation requirement. This means that a weaker correctness preservation requirement should be defined for this specific transformation case. Note also that the simpler solution has been used in the example; this means that the transformation used there does not preserve the correctness requirement. However, still the simpler solution is acceptable for this broadcast case, and makes not necessary to use different gates for each communicating pair. On the other hand, the use of distinct gates helps to identify the different communicating pairs and is therefore closer to implementation.

b) agreement on a value performed by a single receiver

The second case is analogous to the previous one, but the sign of the input-output operations are reverted; that is, in PD there is a single "input" command *a?x:t* in a single "receiver" and several "output" commands *a!value*, one for each "sender" process. In this case, however, an agreement on the values to be passed is implied (Figure 8.2).

Condition: the input/output commands on *a* should not occur in a choice; therefore, if there is no agreement, the processes simply block.

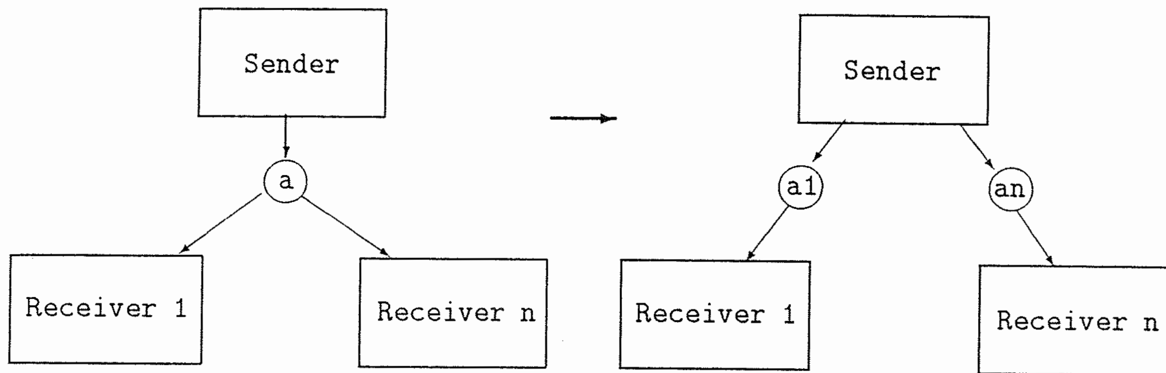


Figure 8.1: Solution a

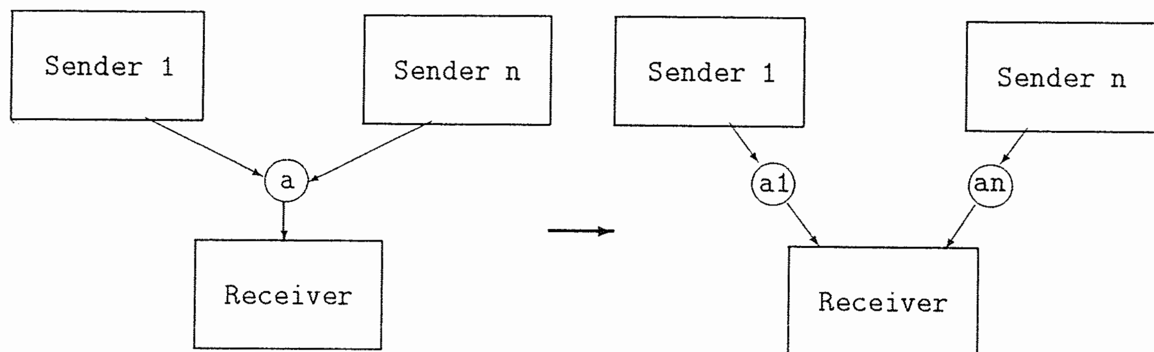


Figure 8.2: Solution b

Solution: substitute `a?x:t` in the receiver process with the sequence: `a1?x:t ; a2!x; ...; an!x; a1!ack; ...; an!ack` and substitute in each sender process Si the command `a!valuei` with `ai!valuei; ai!ack`.

Also in this case, we can give a simpler solution (but not satisfying the correctness preservation requirement), in the casecomplete synchronization is not required:

Solution: substitute `a?x:t` in the receiver process with the sequence: `a1?x:t ; a2!x; ...; an!x` and substitute in each sender process Si the command `a!valuei` with `ai!valuei`.

c) agreement on a value through a master

The input/output commands on a can occur in any subprocess of PD. The following solution is given for processes communicating on the sort Nat; we consider predicates on the input values of the form:

$[lb \leq x \leq ub]$.

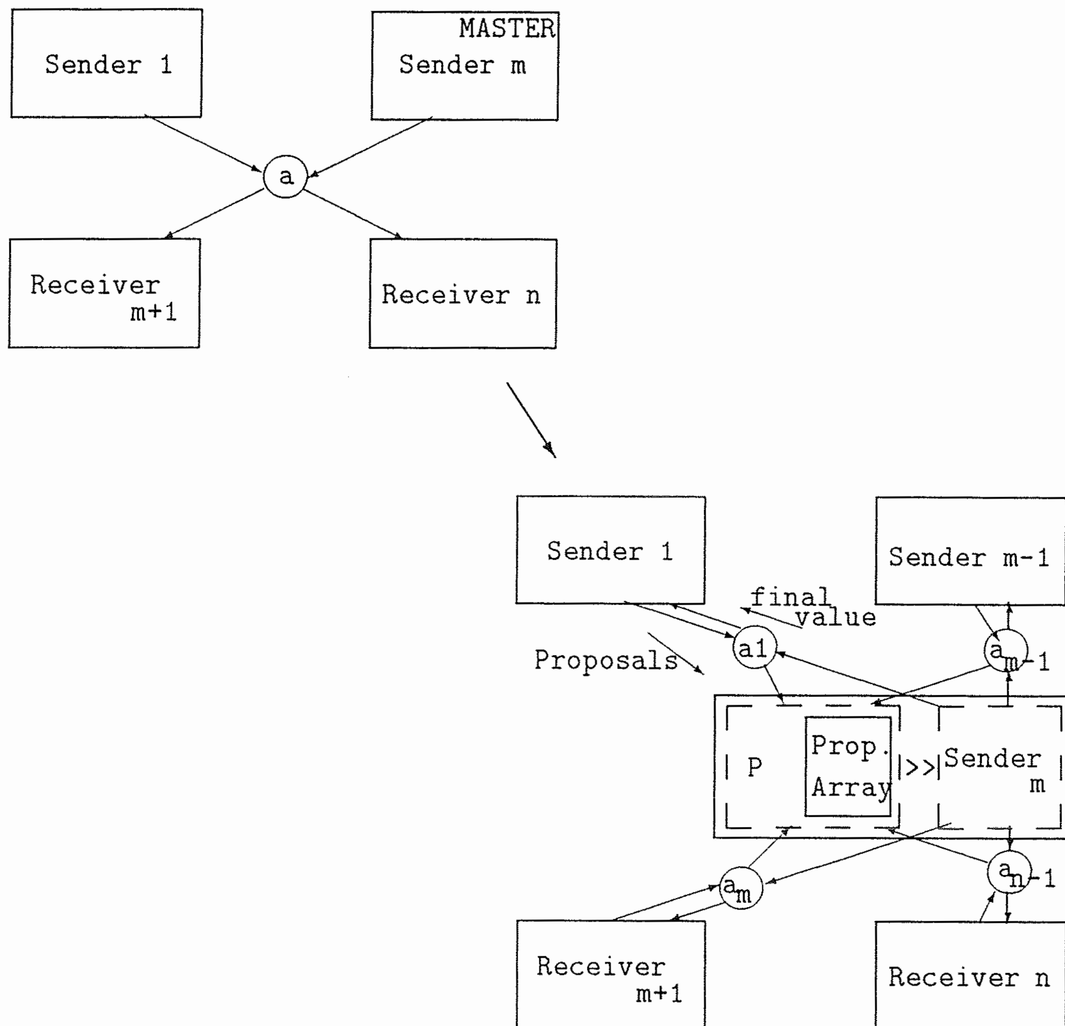


Figure 8.3: Solution c

The solution can be adapted, by hand, to different sorts and different predicates. A general solution in this respect could be given only if we had in LOTOS the possibility of passing sorts and predicates as first class values in the communications (polymorphism).

Condition: the input/output commands on a should not occur in a choice.

Solution: first identify a "master" subprocess (this choice is completely arbitrary).

Add to the master the following data specification, here given informally for conciseness:

an array of n elements which are records with three fields (value, lower bound, upper bound) with values in Nat+nil, and with the following operations:

- three modification operations, *modifyi(array, index, value)*, with i=1..3, which mod-

ify the related field of the indexed element

- creation operation of the initial array with all fields equal to nil, named *createarrayNIL*
- *agreement predicate* (says whether a value satisfies all the constraints requested by the communicating partners)

The array is used to store the proposed value or interval sent by each partner (Figure 8.3 shows the case in which there are m "senders" and n-m "receivers", and the m-th sender has been chosen as the master). When all the proposals are arrived, the process can decide the final value. Moreover, we assume that a predicate *ininterval(x,lb,ub)* exists, which is true if x is in the closed interval (lb,ub).

Add to the master the following process definitions:

```
process P(array:AR)[a1, a2,....an]  : exit AR:=
Pslice(array,1) [a1] >> accept array in Pslice(array,2) [a2] >>
...
accept array in Pslice(array, n-1) [an-1]
endproc

process Pslice(array:AR, index:Nat)[b]  : exit AR :=
     b?val:Nat; exit (modify1 (array, index, val))
  [] b ?lb:Nat ?ub:Nat;
          exit (modify3(modify2(array, index, lb), index, ub))
endproc
```

Substitute in the master every occurrence of `a! value; B` with the following sequence of operations:

```
P(createarrayNIL) >> accept array in
      ([agreed(array,value)] -> a1!value; a2!value;....an-1!value;B)
```

Substitute in the master every occurrence of

```
a? x: Nat [ininterval(x,lb,ub)]; B
```

with the following sequence of operations:

```
P(createarrayNIL) >> accept array in
choice x:Nat [] ([agreed(array,x)] -> [ininterval(x,lb,ub)] ->
  a1! x; a2! x; ....an-1!x; B)
```

in every other process Pi (i=1..n-1), perform the following substitutions:

every occurrence of `a!value; B` is substituted by: `ai!value; ai!value; B`

every occurrence of

```
a? x:Nat [ininterval(x,lb,ub)]; B
```

is substituted by:

```
ai! <lb,ub>; ai?x:Nat ; B
```

This solution may use a simplified array if all the input commands have no predicate. Note that this solution introduces a data structure, that is, it is directed towards the implementation of the communication. However, it still employs the rather "abstract" *choice* construct. Obviously, in an implementation the choice construct would be substituted by a more concrete, though still arbitrary, choice. The implementation of this choice will probably be influenced by the data type actually involved in the case at hand (e.g., default values could be preferred).

The final phase of communication of the agreed value to all the partners ensures their complete synchronization.

d) distributed agreement on a value, by token passing

We present a distributed solution to the case c) above; the solution is given with the same restrictions on sorts and predicates used for the previous case. The same comments on generalization apply.

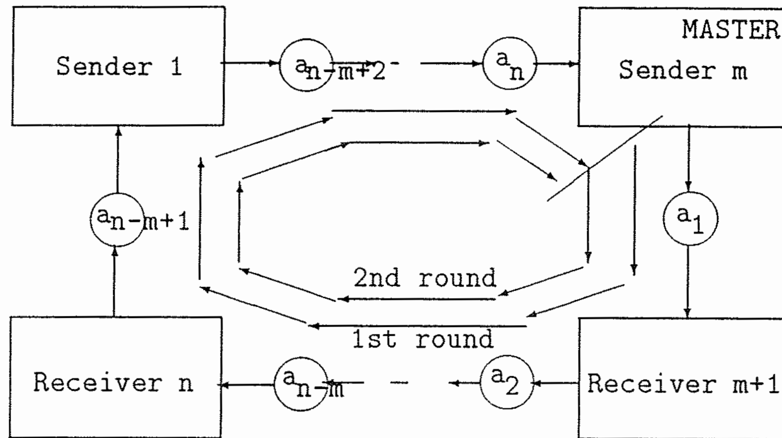
In this solution, the new gates substituting the single *a* gate connect the processes in a ring fashion; two rounds of token passing through the network are needed to complete the agreement. The first round brings proposals of values, which are refined as the token completes the round. At the end of the first round, the "master" process decides about the value and starts a second round to communicate to the other processes the agreed value (see Figure 8.4, where only the transformed processes are shown, being the original processes the same as Figure 8.3).

Condition: the input/output commands on a should not occur in a choice.

Solution: first identify a "master" subprocess -

in the master: every occurrence of `a!value; B` is substituted by:

```
a1!value;
(an!value; a1!value; an!value; B
```



1st round: proposed values
2nd round: agreed value

Figure 8.4: Solution d

```
[] an ?lb:Nat ?ub:Nat;
              [ininterval(value,lb,ub)] -> a1!value; an!value; B)
```

every occurrence of

```
 a? x:Nat [ininterval(x,lb,ub)]; B
```

is substituted by:

```
a1 !lb !ub;
(an?x:Nat; [ininterval(x,lb,ub)] -> a1!x; an!x; B
[] an ?lbp:Nat ?ubp:Nat; choice x:Nat [][ininterval(x,lbp,ubp)] ->
                                                    a1!x; an!x; B)
```

in every other process Pi: every occurrence of `a!value; B` is substituted by:

```
(ai-1!value; ai!value; ai-1!value; ai!value; B
[] ai-1 ?lb:Nat ?ub:Nat; [ininterval(value,lb,ub)] ->
                         ai!value; ai-1!value; ai!value; B)
```

every occurrence of

```
a? x:Nat [ininterval(x,lb,ub)]; B
```

is substituted by:

```
(ai-1?x:Nat; [ininterval(x,lb,ub)] -> ai!x; ai-1!x; ai!x; B
[] ai-1 ?lbp:Nat ?ubp:Nat;
      ai !max(lb,lbp) !min(ub,ubp); ai-1?x:Nat ; ai!x; B)
```

where we have assumed the existence of two *max* and *min* operations, with the obvious meaning.

The second round ensures the complete synchronization of the partner processes.

e) internal agreement on performing actions, through a centralized scheduler

In this case the presence of input/output commands on the multiway gate *a* in "closed" choice constructs is admitted. A choice in a subprocess P of S is called "closed" if for each gate g present in the choice, there exists in S an embedding operator of the form:

```
hide g in ...P |[...g..]| Q..... ,
```

that is , if g is not visible outside the specification.

Let *CC(G)* be the set of closed choices containing gates in G. Let *Gates(CC(G))* be the set of all the gates contained in CC(G). Let $G_{min}(a)$ be the minimal set including *a* and closed with respect to the operation $Gates(CC(\bullet))$.

Condition This solution assumes that each gate in $G_{min}(a)$ appears only in closed choices.

This solution uses a centralized scheduler which takes every decision on the actions to be performed by the relevant closed choices; the scheduler process, therefore, implements both the agreement on the value and the agreement on the actions to be performed; n gates (a1,..,an) connect directly each process with the scheduler (Figure 8.5).

In order to make this solution more readable we do not allow the presence of predicates in input commands. Nevertheless, there is nothing that prevents the use of the technique of case c for taking in account predicates. For the same reason, we do not indicate the sort of variables in action prefixes.

Solution. In every process Pi, every closed choice construct in the set *CC(Gmin(a))* of the form:

```
(b1!v1; B1[]....[]bk!vk; Bk[] c1?x; Cn []...[]cm?x;Cm)
```

is substituted by:

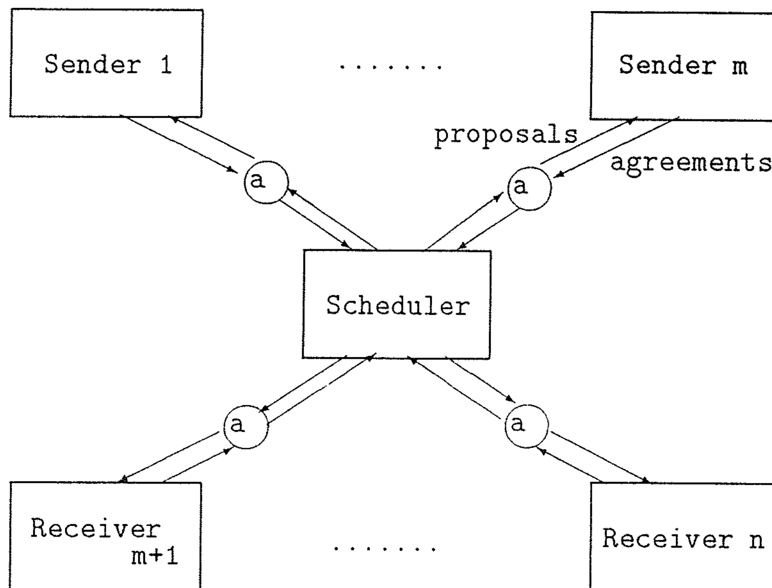


Figure 8.5: Solution e

```
ai! <<b1,v1>,...<bk,vk>,<c1,...cm>>;
ai? <p,x>;
(   [p=b1] ->B1
..
[][p=bk] ->Bk
[][p=c1] ->C1
..
[][p=cm] ->Cm   )
```

where the notation `<x1, x2,...xn>` is from now on used to represent concisely n-uples of values, and *bi*, *ci*, are string values corresponding to the identifiers of gates in the set $G_{min}(a)$.

in particular, the degenerate cases `b!v ; B` and `c?x; C` can be substituted respectively by:

```
ai!<<b,v>>; ai!<b,v>; B      and     ai!<<c>>; ai?<c,x>; B
```

The following data definition is added to the specification:

a table containing information on processes suspended on gates; the exported operations are:

- a creation operation, *Initialtable*. The initial table contains the information about which processes must synchronize on each gate (this information can be statically

derived by the structure of the parallel composition operators. The table contains initially the information that no process is suspended at any gate;

- a couple of modification operations: *suspend*$(i, < b, v >, A)$ to record that the process i is suspended (with offer v) on gate b; *reset(g,A)* to delete suspension information on gate g;
- a predicate, *suspendedon*, which says whether a process is suspended on a certain gate;
- an agreement predicate, *agreed*, which says whether a value satisfies all the constraints requested by the communicating partners on a certain gate;
- a *ready* predicate, which is true when all the processes that must synchronize on a gate are suspended on that gate.

The following process definition is added (and properly composed in parallel) to the specification:

```
process InitialScheduler[a1,...an](A: schedulingtable) noexit :=
    Scheduler(Initialtable)
endproc
where
process Scheduler [a1,...an] (A: schedulingtable) noexit :=
  ( a1? <<b1,v1>,...<bk,vk>,<c1,...cm>>;
      Scheduler [a1,...an] (suspend(1,<b1,v1>,
                            ...
                            suspend(1,<bk,vk>,
                            suspend(1,<c1,any>,
                            ...
                            suspend(1,<cm,any>, A)..))...)
 ...
  [] an? <<b1,v1>,...<bk,vk>,<c1,...cm>>;
      Scheduler [a1,...an] (suspend(n,<b1,v1>,
                            ...
                            suspend(n,<bk,vk>,
                            suspend(n,<c1,any>,
                            ...
                            suspend(n,<cm,any>, A)..))...)
  []  choice  x:G [] [ready (x,A)] ->
        choice v: Nat  [] [agreed(v,x,A)] ->
          ( [suspendedon(1,x,A)] -> a1! <x,v>; exit
          [][not(suspendedon(1,x,A))] ->  exit ) >>
          ...
          ( [suspendedon(n,x,A)] -> an! <x,v>; exit
```

```
          [][not(suspendedon(n,x,A))] ->  exit ) >>
Scheduler(reset(x,A))
endproc
```

where G is the set of identifiers of gates in $G_{min}(a)$. This solution is evidently oriented to the implementation; note that this solution is able to transform not only the single a gate of the original problem, but any multiway gate included in $G_{min}(a)$. This solution could be considerably simpler if dynamic channels would be available in LOTOS. The final phase of communication of the agreed value ensures the complete synchronization of the partner processes.

f) distributed agreement on performing actions, with possibility of external intervention

We admit now the possibility that the other gates of a nondeterministic choice in which a is present can be accessed from outside the specification. In this case, in a solution similar to the case e) above we could not maintain in a table the complete knowledge of the processes communicating on a, since they are not known a priori. To solve this problem a *cancellation* mechanism is needed, which is activated when a suspended process is reactivated (on a different choice) by an external intervention). The cancellation mechanism permits also to treat each gate separately, since it no more needed to consider the transitive closure of alternatives to a, as done in solution e), because alternatives can be all considered as external.

We have chosen to show this cancellation mechanism within a distributed solution; a similar cancellation mechanism is however possible also in a centralized solution. To each communication partner on a a new scheduling process is associated; the scheduler processes are connected in a ring fashion (Figure 8.6). Since there is now the possibility for a process to be simultaneously suspended on a and on other gates, possibly accessible from outside the specification, the communications on the ring employ three phases (rounds of token passing through the ring) in order to complete the agreement (Figure 8.7):

- alert phase, initiated by one of the processes, chosen as the master, when it enters the relevant choice construct;
- agreement phase, which communicates to each partner that an agreement is possible
- acknowledge phase, which communicates that an agreement has been reached and carries the agreed value.

During the alert phase it is still possible for a process to choose independently another alternative, by sending a cancellation message to the associated scheduler, which triggers a cancellation phase. This possibility is blocked by the agreement phase. The cancellation phase may be triggered also by a missed agreement on the values, carried on by the alert

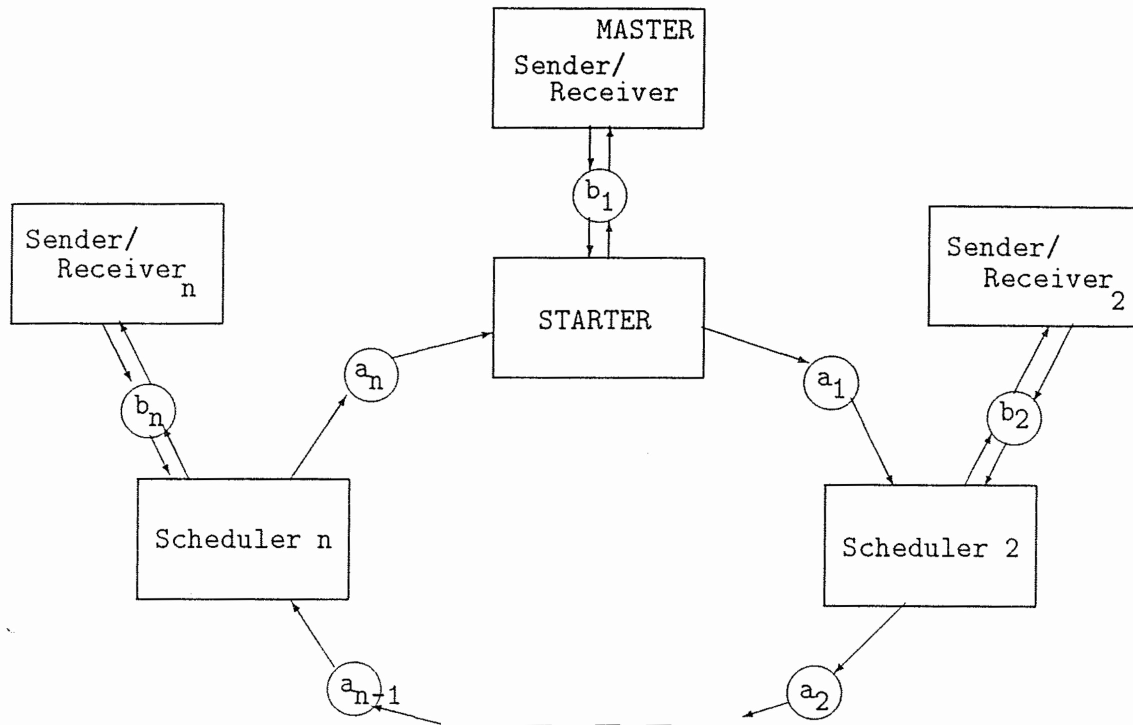


Figure 8.6: Solution f

messages. The cancellation message is propagated to all the schedulers, by completing the interrupted round and by another subsequent round of cancel messages (Figure 8.8). Cancel messages reaching a scheduler may be ignored if they are not interesting for the current state of the associated process, or passed to the process, where they can interrupt the wait for the acknowledge phase.

Solution. In each process Pi substitute the choice:

```
(a?x; Rest-a [] others; Rest-others)
```

with the process instantiation `P[others, bi]`, where

```
process P[others, bi]: functionality as Rest-a and Rest-others :=
  ( others; Rest-others
  []bi!alert ; ( bi!OK; ( bi!Accepted ?x; Rest-a
                        []bi!Cancelled; P[others, bi])
               []others; bi!Cancel; Rest-others)
endproc
```

and where "others" stands for a choice of actions or for the related set of gates, as clear from the context, and, correspondingly, "Rest-others" stands for a set of behaviour expres-

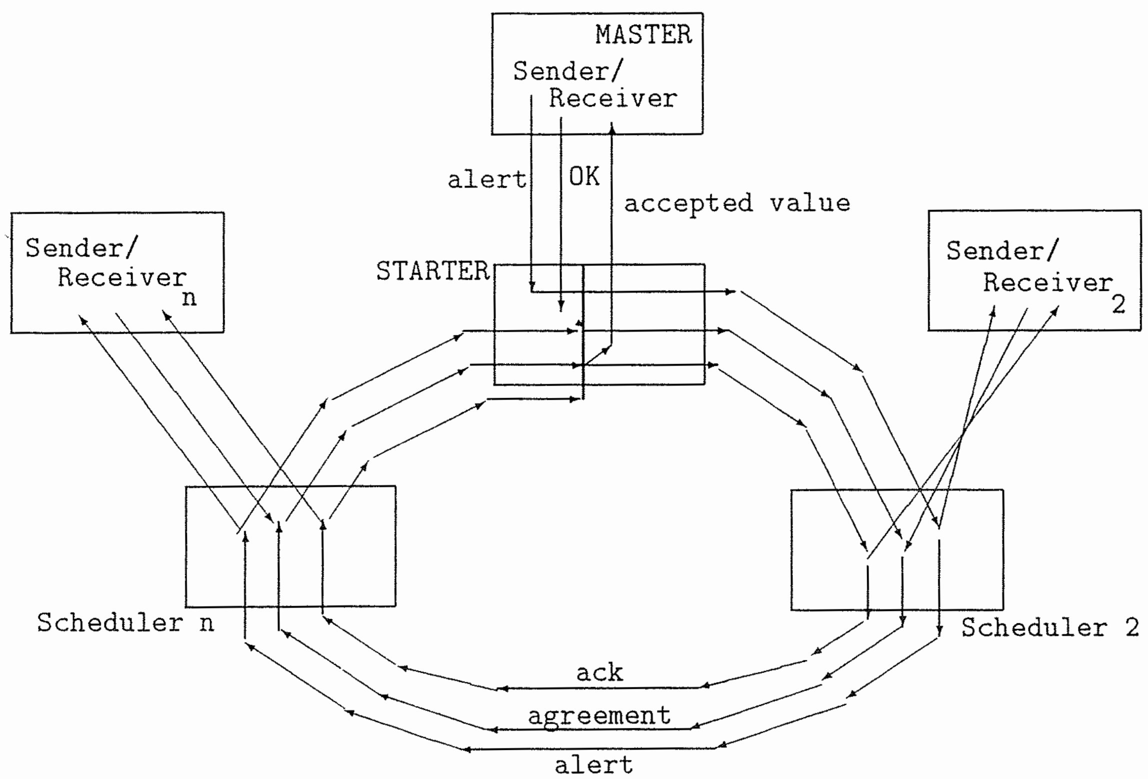


Figure 8.7: The three rounds

Figure 8.8: A case of cancellation

sions consequent to the "others" alternatives). Note that *a1...an* are the gates connecting in a ring fashion the scheduler processes, while *b1...bn* are the gates connecting the processes with the associated scheduler; therefore, the Correctness Preservation Requirement should be modified to include the *bi* in the set of hidden gates. As in solution e), we admit in offers only either variables of (not indicated) sort Nat or values in Nat, referring to the techniques of solutions c) and d) for more complex value agreements.

The following process definitions are added - and properly composed in parallel - to the specification (there is one "starter" process, which is the scheduler process associated to the process chosen as the master; there are n-1 instantiations Scheduler[ai-1,ai,bi], for i= 2..n):

```
process Starter [a1,an,b1] noexit :=
( b1!alert !any; a1!alert !any;
    ( an!alert ?value; Starter-2 [a1,an,b1](value)
    []an!alert !any; choice x:Nat [] Starter-2 [a1,an,b1](x)
    []an!Cancel;a1!Cancel;an!Cancel;(Starter[a1,an,b1]
                                    [] b1!Cancel;Starter[a1,an,b1]
                                    )
    [] b1!Cancel; an!alert; Cancelling
    )
[]b1!alert ?value; a1!alert!value;
    ( an!alert ?value; Starter-2 [a1,an,b1](value)
    []an!Cancel;a1!Cancel;an!Cancel;(Starter[a1,an,b1]
                                    [] b1!Cancel; Starter[a1,an,b1]
                                    )
    [] b1!Cancel; (  an!alert !any; Cancelling
                  [] an!alert ?value; Cancelling
                  )
    )
)
endproc
where
process Starter-2 [a1,an,b1](x:Nat) noexit :=
( b1!Cancel; Cancelling
[]b1!OK; a1!Agreed; (an!Agreed; B1!Accepted !x;
                              a1!Ack !x; an!Ack !x;
                              Starter[a1,an,b1]
                    [] an!Cancel;b1!Cancelled; Cancelling
                    )
)
endproc
process Cancelling [a1,an,b1] noexit :=
a1!Cancel;an!Cancel;Starter[a1,an,b1]
```

```
endproc



process Scheduler [left,right,up] noexit :=
( left!alert !any; ( up!alert !any; ( right!alert !any;
                                        Scheduler [left,right,up]
                                  []up!Cancel;S-Cancelling[left,right,up]
                                  )
                  []up!alert ?value;( right!alert !value;
                                        Sched2 [left,right,up]
                                   []up!Cancel;S-Cancelling[left,right,up]
                                   )
                  )
[]left!alert ?value;( up!alert !any; ( right!alert !value;
                                        Sched2 [left,right,up]
                                   []up!Cancel;S-Cancelling[left,right,up]
                                   )
                   []up!alert ?value2;
                         ( [eq(value,value2)]->
                                ( right!alert !value;
                                        Sched2 [left,right,up]
                                []up!Cancel;S-Cancelling[left,right,up]
                                )
                         [][ne(value,value2)]->
                              ( right!Cancel;
                                   ( Scheduler [left,right,up]
                                   []up!Cancel;
                                            Scheduler [left,right,up]
                                   )
                              []up!Cancel;S-Cancelling[left,right,up]
                              )
                         )
                   []up!Cancel;S-Cancelling[left,right,up]
                   )
[]left!Cancel; S-Cancelling[left,right,up]
)
endproc

where
process Sched2 [left,right,up] noexit :=
( left!Agreed; ( up!OK;right!Agreed;( left!Ack ?x;
                                        up!Accepted !x;
                                        right!Ack !x;
```

```
                                        Scheduler[left,right,up]
                                 []left!Cancel; up!Cancelled;
                                        S-Cancelling[left,right,up]
                                 )
             []up!Cancel; S-Cancelling[left,right,up]
             )
[]left!Cancel; S-Cancelling[left,right,up]
[]up!Cancel; S-Cancelling[left,right,up]
)
endproc

process S-Cancelling[left,right,up] noexit :=
  right!Cancel; Scheduler[left,right,up]
endproc
```

The third acknowledge round ensures the complete synchronization of the partner processes.