Status-Oriented Telephone Service Specification: An Exercise in LOTOS Style[1]

Bernard Stepien and Luigi Logrippo
Telecommunications Software Engineering Research Group
University of Ottawa
Department of Compuer Science
Ottawa, Ont. Canada K1S 9B4
email: bernard{luigi}@csi.uottawa.ca

1. Motivation

The authors and other members of their group have already written some papers and research reports on specifying telephone call processing in LOTOS. Two such papers, where two differently organized specifications of the Plain Old Telephone System service (POTS) were presented, were [FLS 90][FLS 91]. The specification in the second paper was mainly in the constraint-oriented style, while the one in the first was in a combination of constraint-oriented and state-oriented style. We are now presenting yet another style of specifying telephone systems, because we feel that this new style has some definite advantages over the ones used in previous papers. In particular:

1. In the previous specifications, the identification of busy telephone numbers was done by including such numbers in a set; in this specification, busy telephones directly inform others of their state by offering the *busy* action only.

2. While in previous specifications a disconnection caused all the components of the connection (the two stations and the controller) to end up in a deadlock and thus to become "dead" and non-reusable, in this specification these components go back to initial state and are reusable in another connection.

3. Resource - oriented style: specification modules more closely model software entities such as processes that control the individual phones and the individual connections. Such processes can be dynamically created for an unlimited number of phones and connections.

4. In previous specifications, there was no mechanism for installing new stations. We have such a mechanism here.

Structurally, the main difference between this specification and previous ones are:

1. ACT ONE has almost disappeared from the specification (we are still using it to maintain the set of "installed" phones, a feature not present in previous specifications).

2. Control is realized by having processes exchange status information: therefore, we identify our specification style as a variation of the resource-oriented style, which we call *status-oriented*

1. University of Ottawa, Department of Computer Science, TR-93-07 (Feb. 1993)

style.

In our view, the main advantage of this new style is that implementors are likely to find it appealing, because it mimicks the way “black boxes” exchange status information in a physical system. Another useful characteristic of this style is that at every step of simulation a small number only of “next actions” is possible, in other words symbolic behavior trees are narrow. Needless to say, this is in sharp contrast with what happens in the case of the constraint-oriented style, where symbolic behavior trees tend to be quite wide, including many unfeasible paths. As a consequence, simulation, model-checking, and generation of test cases are expected to be all easier.

2. The Informal Specification

The Plain Old Telephone System, or POTS, is composed of a variable number of **station**s which (unless they are out of service, see below) can perform the two basic roles of call initiator and call responder, and of a **controller** that can establish the connection between any two **station**s, or can disconnect already connected **station**s. The call initiator role always starts with an **off_hook** trigger action, while the call responder role always starts with a **ring** trigger action.

Stations do not communicate directly: they are connected via the **controller** which is a distinctive physical entity, just as the **station**s are. When it is available, a **controller** can provide two types of services: connecting two **station**s upon a connection request from one of them, or disconnecting a **station**. The latter service can be offered either upon a disconnection request from another **station**, or by the **station**’s own initiative while it is in the process of being connected.

The purpose of this paper is to provide a specification of the POTS service from a user’s point of view. Therefore the switching function will not be specified explicitly.

Each entity has a life cycle that starts with an idle state and finishes by returning to the idle state. A **station**’s lifecycle can start either with an **off hook** or a **ring** and terminates either with a **hang_ up** or by stopping to **ring**. At the point of termination the **station** is again available with an **off hook** or a **ring** action.

The lifecycle of a connection control process starts with a connection request and ends when all parties involved in a connection are disconnected. To establish an initial understanding of what we are trying to specify, we describe informally the lifecycles of our entities.

***Station Lifecycle*:**

in a call initiator role

off hook -> dial tone -> dialing a number -> requesting a connection to the controller -> being connected by the controller -> talking

or in a call responder role

ringing -> answering -> being connected by the controller -> talking

Termination of a station's lifecycle:

in an initiator role:

The **station** can be hung up any time after an **off hook**. If a connection request has already been placed to the **controller**, it will have to also request a disconnection to the **controller**. The **station** could also receive a disconnection indication from the **controller** only after a connection request had been placed.

in a responder role:

The **station** can only terminate if it has rung. If it has not answered a **ring**, it can terminate by stopping to **ring**. If it has answered a **ring** it can **hang_up** and request a disconnection to the **controller** or it can receive a disconnection indication from the **controller**.

In both initiator or responder roles, the line should provide a **busy** signal to anyone attempting to ring it while it is already used in another connection.

Controller's lifecyle:

in a connector role:

receive a connection request ->
    either ring the called party -> connect the called party -> connect the caller
    or obtain a busy signal -> send a disconnection indication to the caller

in a disconnecting role:

receive a disconnection request from one station -> disconnect the other station

Termination of a controller's lifecycle:

There are two situations where the **controller** provides a disconnection:

• as a call establishment abortion.
• as an established connection termination.

The two situations are different:

in an already established connection, both parties have to be disconnected, while in the call establishment abortion case, the parties to be disconnected will depend on the phase in which the connection process is.

In the remainder of the paper, we provide a formal specification of the behavior described above. We have used the following conventions: constant values (such as values denoting status information) are identified by lower-case names, while LOTOS variables are identified by capitalized names.

3. Specification structure: Call connection

The overall highest level system architecture follows the concept of specifying the physical entities that exists in a telephone system. As we have seen in the informal specification, there are two groups of entities: the individual phone stations and the individual connection control processes.

A phone station can be specified generically in LOTOS, with a variable representing the phone number. An unlimited number of instances of phone stations can thus be produced. All of these independent stations will be in parallel with the **controller** and will interact on some actions with it.

```
collection of stations

||

controller
```

we specify a phone station using the process name **station(N: phone_number)** where the variable **N** represents the phone number assigned to the generic phone process.The collection of stations would thus be represented by a number of interleaved **station**s. The above high level design could be expanded to the following structure:

```
(
    station [b,g,n,t,tn](1234)
    |||
    station [b,g,n,t,tn](5678)
    |||
    ...
    |||
    station [b,g,n,t,tn](9999)
)

    ||

    controller[n,tn]
```

where 1234, 5678 and 9999 are phone numbers.

In LOTOS there is a more elegant way to specify that there is an unlimited number of phone **station**s, which uses recursive interleave. In our case we specify a phone station **installer**, that can install any new instance of a phone **station**. The installed phone will then remain active at all times, since we have not specified a phone disconnection service.

The phone station **installer** process:

```
process installer[c,b,g,n,t,tn](PIP:phone_set): noexit:=

          c ? NewNumber: phone_number ;
          (
            station[b,g,n,t,tn](NewNumber)
            |||
             installer[c,b,g,n,t,tn](Insert(NewNumber,PIP))
        )
      []
         ..... (* to be shown later  *)
endproc
```

At the interaction point **c**, the environment can provide the number of a new instance of the **station** process. The next behavior expression consists in the new instance interleaved with the **installer** itself.

While the phone stations represent physical instances of phones that will exist permanently once installed, the instances of the **controller** have a radically different type of life cycle. The **controller** provides an instance of a control process to any station that initiates a call through a connection request. The control process is an end to end constraint that shows the protocol followed by the parties involved. Once a connection is terminated, the control process will merely die, corresponding to the fact that it represent a virtual machine, more than a physical entity.
This is the high level structure of the telephone system:

```
hide n, tn in
    (
        installer[c,b,g,n,t,tn]({} of phone_set)
     |[b,n,tn]|
       controller[b,n,tn]
    )
```

The specification of a generic station

As we have seen in the informal specification, a phone station can have two possible exclusive roles: a call initiator or a call responder. A station cannot by itself decide which role it will assume. In LOTOS we use the non deterministic choice operator **[]** to describe such a structure.

```
process station[b,g,n,t,tn](N: phone_number): noexit :=

        (
             call_initiator_station[b,g,n,t,tn](N)

             []

             call_responder_station[b,g,n,t,tn](N)
       )
        >>
            station[b,g,n,t,tn](N)
```

```
endproc
```

We also specify that a phone **station** should be able to recycle itself upon completion of either one of its roles. This is achieved with the use of the LOTOS enable operator **>>** and a recursion to process **station**.

The specification of the call initiator role of a station

The LOTOS specification follows relatively easily the informal specification using the LOTOS action prefix operator **;** to specify the sequence of events and the LOTOS disable operator **[>** to specify the fact that a termination may occur at any time after an **off hook**.

```
process call_initiator_station[b,g,n,t,tn](N:phone_number): noexit:=

    g ! N ! off_hook ;
    (
     (
      (
        g ! N ! tone ;
        g ! N ! dial ? C: phone_number ;
        n ! N ! conreq ! C ;
        n ! N ! connect ;
        talking[g](N)
      )
        |||
        continuous_busy_signal[b](N)
     )

        [> station_call_termination[t,tn](N)
    )

endproc
```

The fact that a **station** should provide a **busy** signal to any one attempting to ring it while it is already involved in a connection is portrayed using the LOTOS parallel interleave operator **|||**.
The process **continuous_busy_signal** is in parallel with the group of actions from tone to talking which indicates that the **station** is already engaged as soon as an **off hook** action is performed.

The process **continuous_busy_signal** is a process that contains only one action and is recursive. This recursion is not intended to specify the sort of cyclical beep a **busy** signal usually consists of. Rather, it is intented to allow an unlimited number of callers to attempt to connect with a busy phone and obtain a **busy** signal. In LOTOS, every synchronization on the **busy** signal will consume this action. The recursion will provide a new instance of a **busy** signal for the next caller to synchronize on.

```
process continuous_busy_signal[b](N:phone_number):noexit:=

    b ! N ! busy ;
    continuous_busy_signal[b](N)
endproc
```

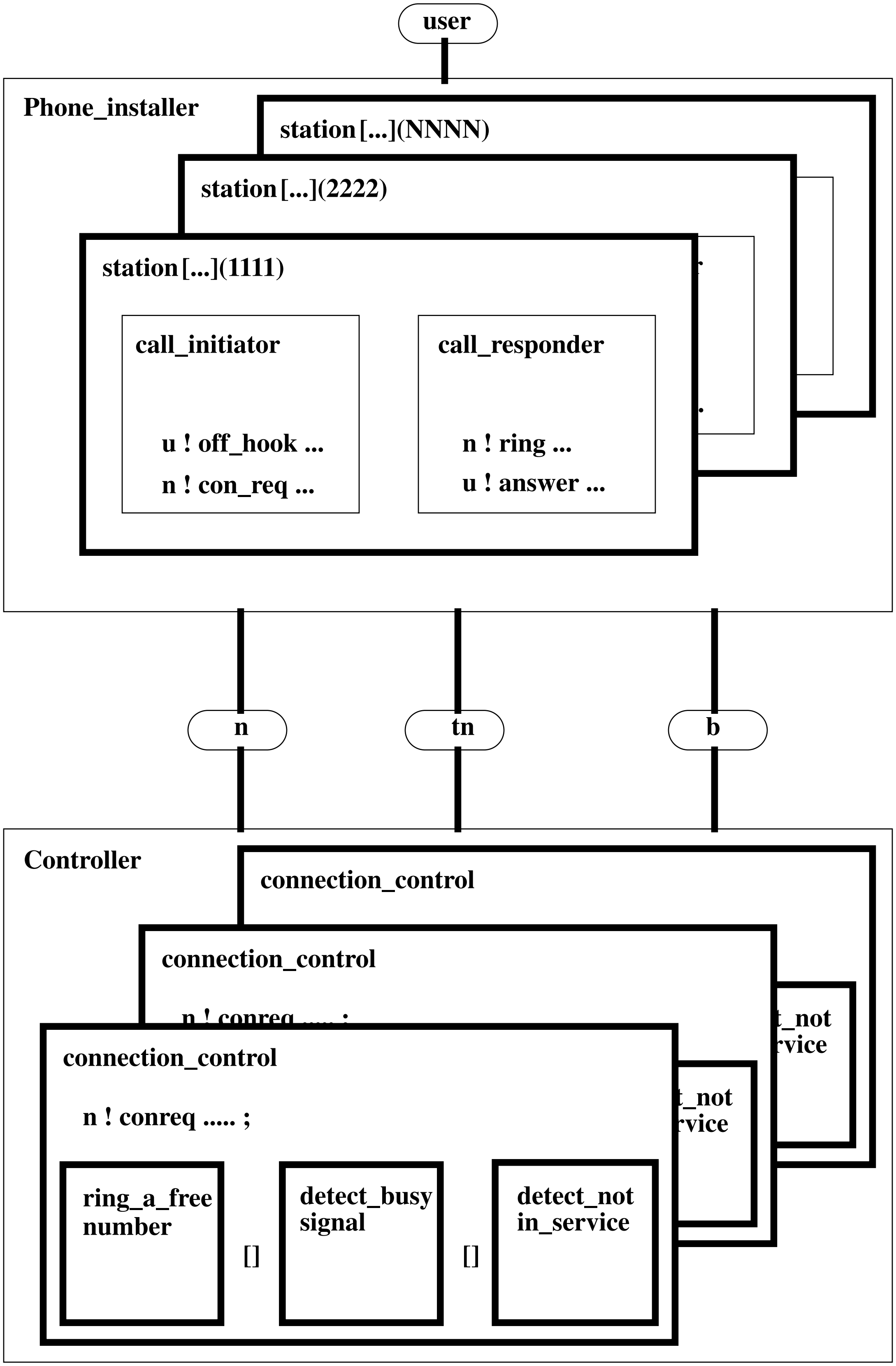

Figure 1. Overall Specification Structure

.

The process **talking** on the other hand is recursive to portray the concept of dialogue between the two parties.

```
process talking[g](N:phone_number):noexit:=

  g ! N ! voice ; talking[g](N)

endproc (* talking *)
```

The specification of the call responder role of a station

The LOTOS specification uses the same LOTOS features and the same overall structure as in the call initiator role

```
process call_responder_station[b,g,n,t,tn](C:phone_number): noexit:=

  n ? N:phone_number !  ring  ! C ;
   (
     (
       (
         g ! C ! answer ;
         n ! C ! connect ;
         talking[g](C)
       )
       |||
       continuous_busy_signal[b](C)
     )
       [> station_call_termination[t,tn](C)
   )

endproc
```

The call termination can occur any time after the **ring** has been executed and also the **station** provides a **busy** signal as soon as it rings, since the line is no longer available to another connection.

Common features between the call initiator and responder role of a station

In both cases we have a role trigger action: the **off_hook** action in the **call initiator** process or the **ring** action in the **call responder** process. Once executed they will determine the subsequent behavior of a station, including providing a **busy** signal or terminating.

Abstract interaction points.

There are two classes of interaction points: general and specialized interaction points

general interaction points:
g is the interaction point between the environment and the **station**.
n is the interaction point between the **station** and the **controller**.

specialized interaction points:
b is the interaction point between the **station** and the **controller** in the case of a **busy** signal.
t is the interaction point between the environment and the **station** in case of termination.

tn is the interaction point between the **station** and the **controller** in the case of termination.

The specification of the controller process

The **controller** is specified as capable of providing an unlimited number of control processes, one for each connection. This is achieved in a similar way as the phone **installer**, using the recursive interleave construct. Each instance of the **connection control** process is instantiated with the appropriate parameters for its connection as determined by its unique trigger action, the connection request, which upon synchronization with a **call initiator station** process will obtain these parameters by value passing. The **controller** can offer only one connection request at a time. As soon as a connection request has been processed, a **connection control** process is created to handle the connection, and a new instance of the **controller** is also created in order to offer a connection request action to the next candidate caller.

The LOTOS specification mirrors in part the **station** specification. There is however a basic difference with the **busy** signal specification. While for a **station** the **busy** signal is a status that is offered continuously at any point of the connection lifecyle, the **controller** will offer it only once as an alternative to a ring. This corresponds to the fact that a **controller** can either ring a **station** because it is free or obtain a **busy** signal if it is not free.

```
process controller[b,n,tn]:noexit:=

      n ? N: phone_number ! conreq ? C: phone_number ;

    connection_control[b,n,tn]

endproc


process connection_control[b,n,tn](N,C:phone_number):noexit:=

      (
      (
       ring_a_free_number[n](N,C)

       []

       detect_busy_signal[b,tn](N,C)

       []

       detect_not_in_service[b,tn](N,C)
      )
        [> control_termination[tn]
     )
    |||

    controller[b,t,tn]
```

```
endproc
```

The **connection_control** process can perform three different types of functions:

- ring a free number
- detect a busy signal
- detect a number that is not in service

The process **ring_a_free_number** is the behavior of a successful completion of a connection, unless it is aborted.

```
process  ring_a_free_number[n](N,C:phone_number):noexit:=

            n ! N ! ring ! C ;
            n ! C ! connect ;
            n ! N ! connect ;
            stop
endproc
```

The **detect_busy_signal** process will occur when the called **station** is busy.

```
process detect_busy_signal[b,tn](N,C:phone_number):noexit:=

            b ! C ! busy ;
            (
             tn ! N ! disind  ? S:status ; stop
              []
             tn ! N ! disreq ? S:status ; stop
            )
endproc
```

Finally if the control process cannot synchronize with either a **ring** or a **busy** signal, this means that there is no phone with such a number at all. It will however synchronize with the **not_in_service** action of the **installer** (see below).

```
process detect_not_in_service[b,tn](N,C:phone_number):noexit:=

      b ! C ! not_in_service ;
            (
             tn ! N ! disind ? S:status; stop
              []
             tn ! N ! disreq ? S:status; stop
            )
endproc
```

As mentioned, the **controller** connects the two parties if the responder is not busy and then dies when the connection is terminated.

4. Specification Structure: Call Termination

The need for different termination patterns

Unfortunately, we are now obliged to complicate considerably our so far relatively straightforward specification in order to take into consideration all possible termination patterns.

For the whole system, there are four termination patterns depending on the stage where the terminating **station** is in a connection. For instance, before a **station** places a connection request to the **controller**, a termination can be performed by simply hanging up. If a connection request has been sent, then a disconnection request has to be placed with the **controller** in order to release the **controller** as well. If the responding **station** was ringing, it has to stop ringing. Finally, when both stations are connected, either one of them can become disconnected. Informally, we have shown all these possibilities in Fig. 2. Boxes at the left-hand-side of the figure show the different stages of the connection. Disconnection sequences, leading back to the initial state, are shown on the right. For example, by reading this diagram we see that if a call initiator hangs up after any of **offhook**, **tone**, or **dial**, the system can return to the initial state without any further action. However, if a call initiator or call responder **hang up** after any of **answer**, **connect**, or **talk**, appropriate disconnection requests or indications have to be exchanged.

Since our system specification consists of two main components, the **station** and the **controller**, we must now specify termination sequences in LOTOS for each one of these two components.We shall first discuss in detail the termination sequences for the **station**. Note that so far we have considered separately the *initiator* and *responder* roles. However, since termination sequences are almost the same for the two roles, it it convenient to combine them into one specification.

The problem that presents itself at this point is how to represent in LOTOS a control structure such as the one presented in Fig. 2. It is a situation of disable, which therefore must be represented as **S:= P [> D**, where P is the normal connection sequence (on the left in the diagram), and D is a choice construct, representing all possible termination sequences (on the right). This structure is shown in process **call_initiator_station** shown above. Which choice must be taken in D, however, depends on where the disable occurred in P. We can think of P as consisting of a series of *stages*, each stage being a sequence of LOTOS action offers. All actions in a stage have in common the fact that if a disable occurs after any of them, the same disconnection sequence must be used. The disable operator, which allows transferring control from P to D at any point within P, is not capable at the same time of transmitting knowledge of the stage where the disabling occurred. In order to save and transmit this information , we use a "memory" process M that synchronizes with P[>D, in the form **(P [> D) || M**. This structure is presented in Fig. 3, where P [> D and M are shown on the left and right respectively. M itself is made of two parts: a state recording process R and a termination logic T, which are in alternative. The following structure results:

(P [> D) || (R [] T).

M is recursive, it reinstates itself after each interaction with P. Before the disable in S, at each reinstantiation, R is executed and the current state of P is passed as a parameter. By synchronizing with P, R keeps track of its progress. We shall see that in order to inform R of its progress, P passes to R what we call *primitive values*. R keeps track of the stage reached within P by generating parameter values, called *status values.* When D takes over, it will no longer synchronize with R, rather it will synchronize with T. T will pass back to D the last reached state of P. Now D and T can synchronize to carry out the appropriate termination procedure. Thus T is also a choice between several termination sequences. Note that T and D must keep synchronizing after the trigger action during the termination sequence, otherwise deadlock will occur.

Station Termination

Applying this concept to process **call_ initiator_station**, seen above, we obtain the following modified structure (using a process **connection _behavior** which will be a process of the form P[>D, an appropriate variation of process **call_initiator_station** seen above):

```
process call_initiator_station[b,g,n,t,tn](N:phone_number):exit:=

        (* connection _behavior *)

        |[b,g,n,t,tn]|

        station_processor_memory[b,g,n,t,tn]

endproc
```

the same structure applies to the **call_responder_station.**

The disabling process, corresponding to D in the discussion above, is as follows:

```
process station_call_termination[t,tn](N:phone_number):exit:=

        station_terminate_set_up[t](N)                      (*1*)
        []
        station_terminate_during_con_req[t,tn](N)           (*2*)
        []
        station_disconnect_once_connected[t,tn](N)          (*2'*)
        []
        station_terminate_after_rung[tn](N)                 (*3*)
        []
        station_terminate_once_connected[t,tn](N)           (*4*)
        []
        station_disconnect_during_con_req[t,tn](N)          (*4'*)

endproc
```

Following is a description of the use of each one of these termination sequences:

1. This termination sequence applies after an **off-hook**, **tone**, or **dial**. Therefore, it only applies to the initiator.
2 and 2'. These termination sequences apply after a **conreq**. Therefore, they apply to both initiator and responder.

3. This termination sequence applies to the responder only after a **ring**.

4. and 4'. These sequences apply to either an initiator or a responder after an **answer**, **connect**, or **talk**.

Each termination sequence is now further specified. Note that each such sequence triggers an exit which enables the return to initial state.

The termination sequence in the set_up stage is very simple.

```
process station_terminate_set_up[t](N:phone_number):exit:=

        t ! N ! hangup ! set_up ; exit

endproc
```

The two following termination sequences reflect the possibility of collision. A collision occurs when both parties **hang_up**, each before having received notification of the other's **hang_up**. Collision is handled by the **controller** who informs each party of the disconnection, instead of forwarding a disconnection request from a party to another. Therefore, these sequences are composed of a termination action, which acts as a trigger, followed by a choice of disconnection request or disconnection indication.

```
process station_terminate_once_connected[t,tn]
                              (N:phone_number):exit:=

        t ! N ! hangup ! conctd ;
        (
         tn ! N ! disreq ? PT: phone_number ? S: status  ;
         exit
          []
         tn ! N ! disind ? S: status ; exit
        )

 endproc

 process station_terminate_during_con_req[t,tn]
                              (N:phone_number):exit:=
        t ! N ! hangup ! recvd_req ;
        (
         tn ! N ! disreq ? PT: phone_number ? S: status  ;
         exit
          []
         tn ! N ! disind ? S: status ; exit
        )
  endproc

  process station_terminate_after_rung[tn]
                              (N:phone_number):exit:=
```

```
    tn ! N ! stopring ! rung ; exit

endproc

process station_disconnect_once_connected[t,tn]
                    (N:phone_number):exit:=

    tn ! N ! disind ! conctd ;
    t !  N ! hangup ! conctd ; exit

endproc


process station_disconnect_during_con_req[t,tn]
                    (N:phone_number):exit:=

    tn ! N ! disind ! recvd_req ;
    t !  N ! hangup !  recvd_req ; exit
endproc
```

Specification of the memory process

As already mentioned, the memory process keeps track of the stage reached within the **station** process by using synchronization. The memory is represented by a formal parameter, the *status value*.

For example when an **off hook** occurs:

```
g ! CIN ! off_hook  ;  station_processor_memory[b,g,n,t,tn](CIN,set_up,CRN)
```

the primitive value **off_hook** is passed to the memory process, which generates the status value **set up**. This is a parameter for the reinstantiation of the memory process. At the same time, the numbers of the initiator and responder stations are passed on.

The connection request action generates the **recvd_req** state in the same way:

```
g ! CIN ! conreq ! CRN  ;  station_processor_memory[b,g,n,t,tn](CIN,recvd_req,CRN)
```

Both the **tone** and **dial** actions are represented using the same construct.

Each of the choices that constitute the termination logic starts by an action offer that synchronizes with one of the choices of termination actions of **station_call_ termination**. An appropriate value offer ensures that the choice is deterministic.

For example, we can specify the proper termination sequence in the **set_up** state as:

```
t ! N ! hang_up ! set_up ; exit
```

which informally means that a hang up is followed by an **exit** (with return to *idle*) in the set up stage. The **controller** is not involved because it has not received a connection request.

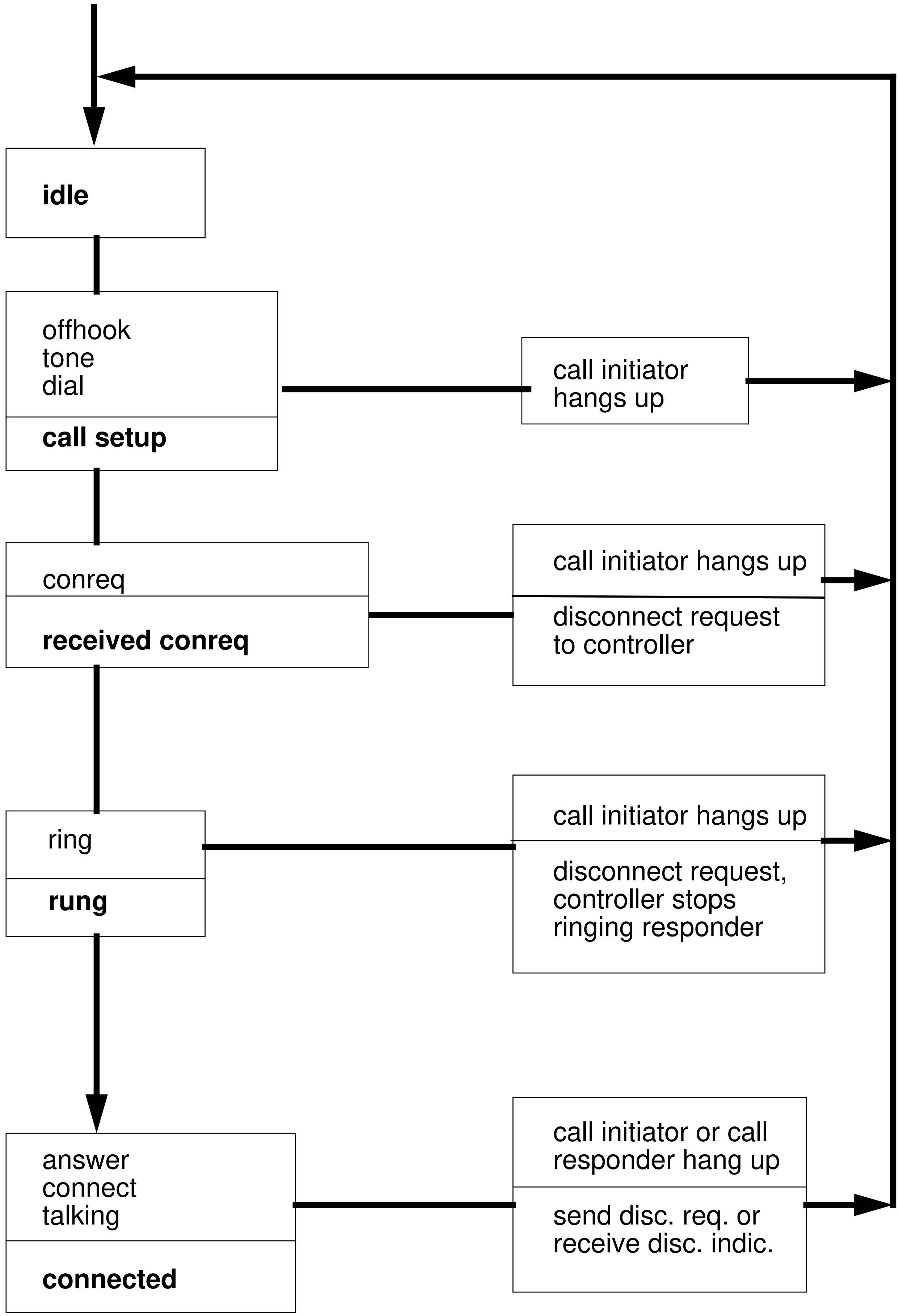

Figure 2: End-to-end Termination Sequences

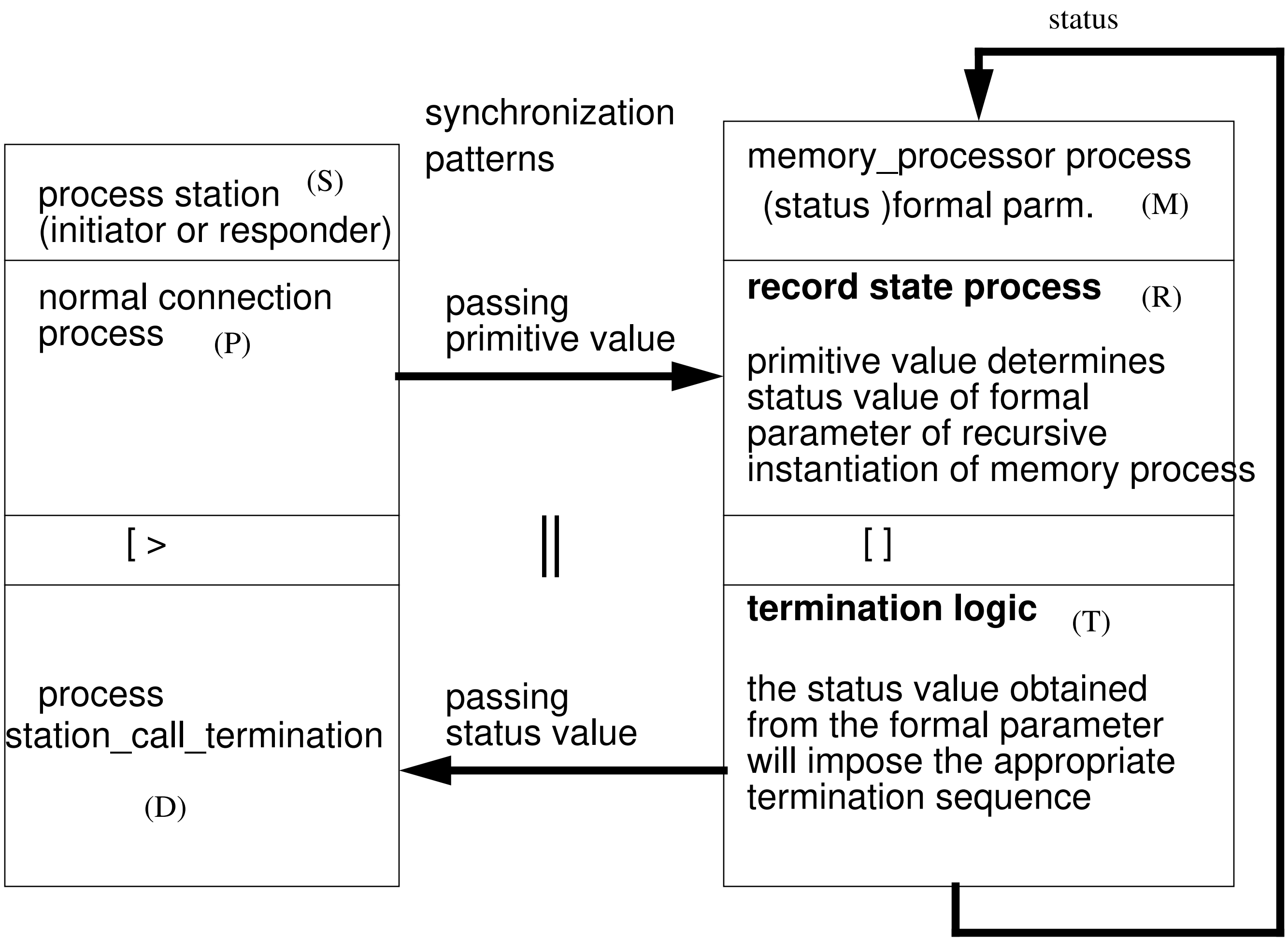

Fig. 3. Structure of a state memory processor

If the connection request has been received by the **controller**, we use the following termination sequence:

t ! N ! hang_up ! recvd_req ;
tn ! N ! disreq ! C ! recvd_req ; exit

where the second action forces termination of the **controller**.These termination actions are specified in the **station_call_termination** process. The processor with memory would have similar actions but with one difference, the formal parameter that acts as a memory instead of a state value.

```
t ! N ! hang_up ! Status ;
 (
   exit
   []
   tn ! N ! disreq ! C ! Status ; exit
 )
```

where the identifier Status will contain the status information. If that value was **set_up** the first action of the processor memory would synchronize with the **t ! N ! hang_up ! set_up** action in the **station_call_termination**. That process would then offer an exit that would resolve the non deterministic choice between the exit or the disreq action in the **memory_processor** process.

At the highest level, our phone memory processor has two main functions:

• Record the stage where the system is.
• Indicate the termination pattern depending on the stage the phone process was in.

```
process station_processor_memory[b,g,n,t,tn](N:phone_number,S: status, C:phone_number):
                                                                                noexit:=

   station_record_status[b,g,n,t,tn](N,Status,C)
   []
   station_indicate_termination_sequence[b,g,n,t,tn](N,Status,C)

endproc
```

Each sequence for recording the station's stage starts by an action which determines a stage change. Process **station_processor_memory** is then reinstantiated with the updated stage. There are five groups of stages for which the termination patterns differ, as indicated in the **station_record_status** process.

```
process station_record_status[b,g,n,t,tn]
              (N:phone_number, PT:phone_number):exit:=
  is_not_reachable[b,g,n,t,tn](N,PT)
  []
  is_in_set_up[b,g,n,t,tn](N,PT)
  []
  is_in_con_req[b,g,n,t,tn](N,PT)
  []
  has_rung[b,g,n,t,tn](N,PT)
  []
  is_fully_connected[b,g,n,t,tn](N,PT)
endproc
```

Each one of these processes responds to an action that is possible in the connection phase. It records a change of stage, and this information is then used to direct (choose?) the termination sequence. The first group of actions deals with a non reachable phone, either because it is busy or because it doesn't exist.

```
process is_not_reachable[b,g,n,t,tn](N,PT:phone_number):exit:=

   b ! N ! busy ;
   station_processor_memory[b,g,n,t,tn](N,sbusy,PT)
  []
   b ! N ! not_in_service ;
   station_processor_memory[b,g,n,t,tn](N,sbusy,PT)
endproc
```

The second group of actions are for the set up stage.

```
process is_in_set_up[b,g,n,t,tn](N,PT:phone_number):exit:=
   g ! N ! offhook ;
   station_processor_memory[b,g,n,t,tn](N,set_up,PT)
  []
   g ! N ! tone ;
   station_processor_memory[b,g,n,t,tn](N,set_up,PT)
  []
   g ! N ! dial ? NPT: phone_number ;
   station_processor_memory[b,g,n,t,tn](N,set_up,NPT)
endproc
```

The next group of actions records the fact that a connection request has been performed. This means that both a phone and a **controller** process have been activated. This implies that both processes will have to be terminated in case of hang up.

```
process is_in_con_req[b,g,n,t,tn](N,PT:phone_number):exit:=
   g  ! N ! answer ;
   station_processor_memory[b,g,n,t,tn](N,recvd_req,PT)
 []
   n  ! N ! conreq ! PT ;
   station_processor_memory[b,g,n,t,tn](N,recvd_req,PT)

  []
   g  ! N ! voice ;
   station_processor_memory[b,g,n,t,tn](N,recvd_req,PT)
endproc
```

The next process is a single action process that records that a **ring** has occurred.

```
process has_rung[b,g,n,t,tn](N,PT:phone_number):exit:=

   n  ? NPT: phone_number ! ring ! N ;
   station_processor_memory[b,g,n,t,tn](N,rung,NPT)
endproc
```

Finally the last status recorder process records the fact that a phone station is connected.

```
process is_fully_connected[b,g,n,t,tn](N,PT:phone_number):exit:=
```

```
    n ! N ! connect ;
    station_processor_memory[b,g,n,t,tn](N,conctd,PT)

 endproc
```

the **indicate_termination_sequence** process will now impose sequences through the value of the status that it inherits after each recursion.

There are three group of actions corresponding to three possible terminations of a phone process.
• The user hangs up. This may lead to a straight exit if the **controller** was not involved yet (set up stage).
• The **station** was a responder and a **ring** had already been performed.
• The **station** has received a disconnect indication from the **controller** and the user has no other choice but hanging up.

The following process synchronizes with process **station_call_termination** given above.

```
process station_indicate_termination_sequence[t,tn]
            (CIN:phone_number,Status:status,CRN:phone_number):exit:=

    t ! CIN ! hang_up ! Status ;
     (
        exit
         []
         tn ! CIN ! disreq ! CRN ! Status ; exit
     )

    []

    tn ! CIN ! stopring ! Status ; exit

   []

    tn ! CIN ! disind ! Status ;
    t ! CIN ! hang_up ! Status ; exit

endproc
```

This process exits to process **station** shown above. In other words, we are ready to restart all over again. In a way the memory and termination sequence processes are mirror images of the **station** and **station_call_termination** processes, but they are different and they don't indicate the temporal ordering of these actions relative to each other. The **indicate_termination_sequence** gives a temporal ordering, but without specifying where in the connection.

The formal parameters CIN and CRN mean Call Initiator Number and Call Responder Number respectively.
The updated **station_call_termination** process would now show a status value for each of its termination sequences.

Controller Termination

Controller termination will be discussed in less detail, because the method should be clear by now.

For the **controller**, there are only the last three termination patterns shown in Fig. 2, because clearly the **controller** is not involved at all in the first one. When a **controller** receives a disconnection request during the connection establishment phase, it will have to determine if it can go straight back to idle state or also act upon the responder **station**. If it is causing the responder **station** to ring, it will have to stop the ringing. If the responder has already answered the call, the **controller** should send it a disconnection indication. The **controller** should also be able to detect cases of collision, i.e. independent disconnection from both sides.

The **controller** will have its own memory and processor mainly to decide if after receiving a disconnection request from the call initiator, it should stop ringing the responder, send a disconnection indication if the responder has answered, or merely stop if the responder has not been involved yet.

Consequently, we will have a similar structure as in the **station** processor. The same design applies for the **connection control** processes. Each **connection control** process is in parallel with a **control_memory_processor**.

```
process connection_control[b,n,tn]:noexit:=

    (* connection control behavior, see above *)

    |[b,n,tn]|

   control_processor_memory[b,n,tn](idle,idle)

endproc

  process control_termination[tn]:noexit:=

        tn ? N: phone_number ! disreq ! C: phone_number ;
       (
        tn ! C ! stopring ! rung ; stop
       []
        tn ! C ! disind ! conctd ; stop
       []
        tn ! C ! disind ! recvd_req ; stop
       []
        tn ! C ! disreq ! N ! conctd ; stop
       )
  endproc
```

There are two functions of the processor:

```
 process control_processor_memory[b,n,tn]
        (CIN:phone_number,CIS:status,CRN:phone_number,CRS:status):noexit:=)

   control_record_status[b,n,tn](CIN,CIS,CRN,CRS)
```

```
[]
control_indicate_termination_sequence[b,tn](CIN,CIS,CRN,CRS)

endproc
```

The first process records the status of a connection action by forwarding it to the next instantiation of the process. There are separate Call Initiator **(CIN)** and Call Responder (**CRN**)status recording actions.

```
process control_record_status[b,n,tn]
        (CIN:phone_number,CIS:status,CRN:phone_number,CRS:status):noexit:=)


   n ? CIN: phone_number ! conreq ? NCRN: phone_number  ;
   control_processor_memory[b,n,tn](CIN,recvd_req,CRN,CRS)

   []

   n ? CIN: phone_number ! ring ? CRN: phone_number ;
   control_processor_memory[b,n,tn](CIN,CIS,CRN,rung)

   []

   n ! CIN ? SG:signal  ;
   control_processor_memoryb,n,tn](CIN,conctd,CRN,CRS)

   []

   n ! CRN ? SG:signal  ;
   control_processor_memory[b,n,tn](CIN,CIS,CRN,conctd)

endproc
```

The **controller_indicate_termination_sequence** process has to be able to perform only one major important decision, i.e. wether or not to get the responder involved. This depends strictly on the **ring** action. If the **ring** has been performed, this means that the responder is active. To know this, we use the memory (formal parameters). The phone numbers are initialized to “none” when the **controller** is idle. A connection request will fill the call initiator memory position, and a **ring** will fill the call responder memory position. When a disconnection request has been received from the call initiator, a simple check using a LOTOS guard on the memory call responder number is enough to decide wether to stop or continue the process.

The details are left to the reader.

```
process control_indicate_termination_sequence[b,tn]
  (CIN:phone_number, CIS:status,CRN:phone_number, CRS:status):exit:=

 indicate_disreq_initiator_to_responder[tn] (CIN, CIS,CRN, CRS)
  []
 indicate_disreq_responder_to_initiator[tn] (CIN, CIS,CRN, CRS)
  []
```

```
  indicate_not_reachable[b,tn] (CIN, CIS,CRN, CRS)
   []
  stop

endproc (* control_indicate_termination_sequence *)

process indicate_disreq_initiator_to_responder[tn]
   (CIN:phone_number, CIS:status,CRN:phone_number, CRS:status):noexit:=


  tn ! CIN ! disreq ? NCRN:phone_number  ! CIS  ;
  (
   tn ! CRN ! stopring ! CRS ; stop
   []
   [CRN ne none]->  (
                   tn ! CRN ! disind ? CRS: status ;  stop
                  []
                   tn ! CRN ! disreq ! CIN ? CRS: status  ; stop
                 )
   []
   [CRN eq none] -> stop
  )
endproc

process indicate_disreq_responder_to_initiator[tn]
   (CIN:phone_number, CIS:status,CRN:phone_number, CRS:status):noexit:=

  tn ! CRN ! disreq ! CIN ? CRS: status  ;
  (
   tn ! CIN ! disind ? CIS:status;  stop
   []
   tn ! CIN ! disreq ! CRN ? CIS:status  ; stop
  )

endproc

process indicate_not_reachable[b,tn]
   (CIN:phone_number, CIS:status,CRN:phone_number, CRS:status):noexit:=

    b ? CRN: phone_number ! busy ;
    (
     tn ! CIN ! disind ! CIS;  stop
     []
     tn ! CIN ! disreq ! CRN ! CIS  ; stop
    )

   []

    b ? CRN: phone_number ! not_in_service ;
    (
     tn ! CIN ! disind ! CIS;  stop
     []
     tn ! CIN ! disreq ! CRN ! CIS  ; stop
    )
```

```
endproc
```

5. Installing Stations

The status oriented specification style solves two well-known constraints easily: the busy number detection and the number activation (an off hook is possible only if the system offers an off hook). However, there remains the problem of specifying what happens when one dials a number that has not been installed. The specification that we have shown so far would merely deadlock, thus preventing any further simulation.

In an old-fashioned telephone system a number not in service could be detected through some grounding mechanism at some point of the hierarchy between exchanges. In a more computer-oriented system, however, one would have a data base of numbers in service. In LOTOS we can use abstract data types to specify such a data base. Using the standard LOTOS library data type Set, we can build a set of installed phone stations. This set of numbers can then be used by a separate process, which will be offered as an alternative to all instances of active numbers. The process will offer a **not_in_service** action, which could be taken as an abstraction for the announcement usually obtained whenever a dialed number is not installed.

This action will have a mirror action in the **controller** process. The latter will now be able to either ring, get a **busy** signal, or a not in service announcement after receiving a connection request.

This is the revised process **installer**:

```
process installer[c,b,g,n,t,tn](Prev_SetofPhones:phone_set): noexit:=

     c ? NewNumber: phone_number ;
      (
          station[b,g,n,t,tn](NewNumber)
          |||
           installer[c,b,g,n,t,tn](Insert(NewNumber,Prev_SetofPhones))
      )
     []
     not_in_service[c,b,g,n,t,tn](Prev_SetofPhones)

    where

    process not_in_service[c,b,g,n,t,tn](SetofPhones:phoneset):noexit:=
       b ? CRS: phone_number ! not_in_service [ CRS NotIn SetofPhones] ;
       installer[c,b,g,n,t,tn]SetofPhones)
   endproc

endproc
```

6. Conclusions

We have presented a style of LOTOS specification based on synchronization between processes to exchange status information. This style takes full advantage of the capabilities of the control part of LOTOS, so the importance of the data part can be reduced significantly.

It remains to be seen to what extent this style can be used to advantage in order to specify more complex entities than the simple POTS service described in this paper. Also the significance of this style with respect to other aspects of the LOTOS-based software development process, such as validation, testing, and implementation, are yet to be clarified.

Acknowledgment. This work was supported in part by grants and contracts of Bellcore, Bell-Northern Research, and the Telecommunications Research Institute of Ontario.

References

[B 91] R. Boumezbeur. Design, Specification, and Validation of Telephony Systems in LOTOS. Master of Computer Science Thesis, University of Ottawa, 1991 (Obtainable by ftp on lotos.csi.-uottawa.ca).

[BL 92] R. Boumezbeur, and L. Logrippo, Specifying Telephone Systems in LOTOS and the Feature Interaction Problem. International Workshop on Feature Interactions in Telecommunications Systems (St.Petersburg, Fla., Dec. 1992), 95-108.

[CV 90] R.C. Cam and S.T. Vuong. A Formal Specification, in LOTOS, of a Simplified Cellular Mobile Communication System. In: S.T. Vuong (ed.) Formal Description Techniques, II. North-Holland, 1990, 485-499.

[DCB 92] L. Drayton, A. Chetwind, and G. Blair. Introduction to LOTOS through a Worked Example. Computer Communications, 15 (1992), 2 70-85.

[FLS 90] M. Faci, L. Logrippo, and B. Stepien. Formal Specification of Telephone Systems in LOTOS. In: E. Brinksma, G. Scollo, and C. Vissers (eds), Protocol Specification, Testing, and Verification, IX North-Holland, 1990, 25-34.

[FLS 91] M. Faci, L. Logrippo, and B. Stepien. Formal Specification of Telephone Systems in LOTOS: the Constraint-Oriented Style Approach. Computer Networks and ISDN Systems, 21 (1991), 53-67.