Année scolaire 1994-95

LOTOS Specification of Telephone Features

Elève:

Edouard Hibon, option RTT

Superviseurs:

Luigi Logrippo, Université d'Ottawa

Elie Najm, ENST Paris

Abstract

The project "LOTOS specification of Telephone Features" is in keeping with the general patern of the LOTOS group's research at the University of Ottawa about the detection of Telephone Feature interaction. This work has two purposes:

- to provide test exemples to apply the detection method of feature interactions presented by Mohamed Faci in his thesis.
- to provide a basis to build up a complete LOTOS specification of a Telephone System with all its associated features.

This report is organized as follows:

- chapter I is an overview of the LOTOS language
- chapter II describes the model of a Telephone System (POTS) developped by the group LOTOS at the University of Ottawa
- chapter III explains the modifications which will be added to this model for specifying Telephone Features
- chapter IV describes the LOTOS specifications of these features

Acknoledgments

- Dr. Luigi Logrippo for his trust
- Mohamed Faci for his advices
- Antoine for his precious help
- Thierry and Stephanie for their support in my work.

Table of Contents

I LOTOS: Language Of Temporal Ordering Specification

I.1 LOTOS: FDT from ISO

Formal Description Techniques are vital tools for communication system designers: they force the designer to communicate his design in precise and unambiguous terms, and they facilitate the sharing of design ideas between users, designers and managers.
LOTOS (Language Of Temporal Ordering Specification) is one of the two Formal Description Techniques developed within ISO (International Organization for Standardization) for the formal specification of open distributed systems and in particular for those related to the Open Systems Interconnection (OSI) computer network Architecture. It was developed by FDT experts in the eighties from the basic idea assuming that systems can be specified by defining the temporal relation among the interactions that constitute the externally observable behavior.

I.2 LOTOS Basic Elements: Process, Event, Synchronization

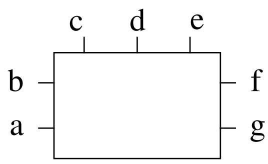
In LOTOS, a distributed, concurrent system is seen as a *process*, possibly consisting of several sub-processes. A sub-process is a process itself, so that generally a LOTOS specification describes a system via a hierarchy of process definitions. LOTOS process definitions are named *Behavior Expressions*, similar to procedures in programming languages, whereas *Process Instantiations* are similar to procedure calls.
A *process* is an entity able to perform internal, unobservable actions, and to interact with other processes which form its environment. Complex interactions between processes are built up out of elementary units of synchronization called *events* or *atomic interactions* or simply *actions*.
Events imply *process synchronization* because the processes that interact in an event (they may be two or more) participate in its execution at the same moment in time and at the same interaction point called *gate*. *Events* are atomic in the sense that they occur instantaneously, without consuming time, and in case of synchronization without data exchange, the *event* name and the *gate* name coincide.
The semantics of process synchronization are quite different in LOTOS from those found in common programming languages: it is

- *multi-way synchronization*: more than two processes can participate.
- *symmetric synchronization*: all processes that participate in a synchronization cooperate in it equally, in particular, there is no concept of a process initiating the synchronization and others responding. For instance, while it is often thought that in an

output operation, a producer process transfers information to a passive external environment, in LOTOS, one says that both the producer and the environment participate in establishing the value being output. There is no directionality in this concept, although some processes may have more information on the values to be transferred than others.

- *anonymous synchronization*: a process that is ready for a synchronization proposes the synchronization to its environment without being able to direct its proposal to a specific process. It is the structure of the system where the process is embedded that decides which processes will have to participate for the synchronization to occur. Process identification can be specified as an exchange of appropriate values.
- *nondeterministic synchronization*: often more than one synchronization is possible, but only one will be executed according to a nondeterministic choice.

The most abstract representation of a process P able to interact with its environment via gates *a* through *g* is a black box:



Process P

The process definition of P specifies its behavior by defining the sequences of observable actions that may occur at the seven gates of the process. Such a behavior is usually represented as a tree of actions where the root is the initial state of the system from which the behavior may execute one of the possible actions, in cooperation with its environment, and so move to another state.
When an action is executed, the behavior expression of the process is transformed into another behavior expression. The inference rules of the dynamic semantics determine which actions can be offered and executed by a process and how behavior expressions are transformed by effect of the execution of actions. For instance, we shall see that the behavior expression `a;B` evolves into the behavior expression `B` after executing action `a`.

I.3 Basic LOTOS and Full LOTOS

Basic LOTOS is the subset of LOTOS where processes interact with each other by pure synchronization, without exchanging values.
Full LOTOS allows processes to exchange values and to be parametrized by them.
A set of operators is used to combine actions and behavior expressions to form other behavior expressions. There are two predefined behavior expressions:

- the inaction behavior **stop**: it cannot offer anything to its environment, nor it can perform internal actions.
- the successful termination **exit**.

Other LOTOS operators are explained next.

I.4 LOTOS Operators

I.4.1 Action Prefix Operator ';'

The *action prefix* operator written as a semi-colon ';' expresses sequential composition of actions. This operator is used to sequentially order actions. For example, a;B denotes a behavior where action a must be executed before the behavior expression B.

I.4.2 Choice Operator '[]'

The *choice* operator written '[]' is used to express a choice between two alternatives. A selection between both alternatives is said to be nondeterministic if the environment cannot distinguish between them. Otherwise, the choice is said to be deterministic. For example, compare the behaviors of three vending machines:

(insert_quarter; get_coffee; **stop**) [] (insert_dime; get_milk; **stop**) (*M1*)
(insert_quarter; get_coffee; **stop**) [] (insert_quarter; get_milk; **stop**) (*M2*)
insert_quarter; ((get_coffee; **stop**) [] (get_milk; **stop**)) (*M3*)

Machine M1 offers the client (the environment) a choice between inserting a quarter and getting a coffee or inserting a dime and getting a milk; the choice is deterministic.
Machine M2 accepts quarters only and after synchronization with the environment (i.e. once the client has inserted a quarter), the behavior of the machine can evolve to either get_coffee; **stop** or get_milk; **stop** depending on a nondeterministic choice (in other words, once the client has inserted a quarter, the choice to synchronize on either get_coffee or get_-milk would no longer exist).
Machine M3 accepts quarters only like M2 but it is more democratic: once the client has inserted a quarter, the behavior of the machine evolves into (get_coffee; **stop**) [] (get_milk; **stop**), meaning that the client can still choose between coffee and milk.

Fig 1. summarizes the difference between deterministic and nondeterministic choices by showing both behavior of machines M2 and M3:

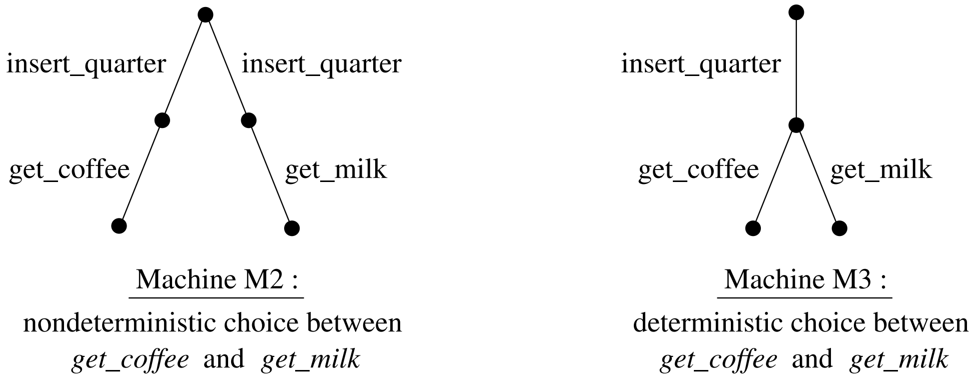


Fig 1. Nondeterministic and deterministic choices.

I.4.3 Enable Operator '>>'

The *enable* operator '>>' has a similar function as the *action prefix* operator, which expresses the sequential composition of an action with a behavior expression; the '>>' is used to express the sequential composition of two behavior expressions. For example, if P1 and P2 are two processes and P1>>P2, P1 must terminate successfully in order for P2 to be enabled. This is the only condition under which process P2 is enabled. Execution of an **exit** in P1 results in an action on a special gate δ. The enable causes δ to become an **i**, and execution to continue with P2. For example,

((a; b; **exit**) >> (c; **stop**)) is equivalent to (a; b; **i**; c; **stop**) i.e. (a; b; c; **stop**)
((a; b; **stop**) >> (c; **stop**)) is equivalent to (a; b; **stop**) because the expression on the right hand side of the enable operator cannot be executed.

I.4.4 Parallel Composition

There are three parallel composition operators in LOTOS: a basic one, the *selective composition* operator, and two derived ones, the *interleaving* operator and the *full synchronization* operator.
The *interleaving* operator denoted by '|||' is used to express the concept of parallelism between behaviors when no synchronization is required:

((a; b; **exit**) ||| (c; d; **exit**)) is equivalent to

```
(
        (a;
            (       b; c; d; exit
                []  c; (b; d; exit [] d; b; exit)
            )
        )
    []  (c;
            (       d; a; b; exit
                []  a; (b; d; exit [] d; b; exit)
            )
)       )
```

When processes must synchronize on common actions, the *selective parallel* operator denoted by '|[L]|' is used, where L is a list of actions on which synchronization must occur. For example,

((a; b; **exit**) |[a]| (d; a; c; **exit**)) is equivalent to

(d; a; (b; **exit** [] c; **exit**)) which is equivalent to

(d;a; (b; c; **exit** [] c; b; **exit**))

This is because both behavior sub-expressions execute independently until one of them reaches a common action, in which case it must wait for the other sub-expression. Once the second sub-expression reaches the same point, synchronization is possible (depending on the context in which the expression occurs, for example, participation of other processes may be necessary) and both sub-expressions proceed to offer their next actions. In this example, the first action of sub-expression 1 is a. Since action a is common to both sub-expressions, it must wait for sub-expression 2 to reach the action a, before offering action b. On the other hand, action d is not common, so no synchronization is required. After action a, they both continue independently, thus sub-expression 1 can offer action b and sub-expression 2 can offer action c, which can occur in either order.

Note that an action is not treated as common between processes composed in parallel unless it is shown in the list of gates in the parallel composition operator. Clearly, when L is the empty list, the selective parallel operator becomes identical to the interleaved operator.

A special case is provided by the action δ which is produced by **exit**s. The action δ is always considered as a common gate, for any parallel composition operator. Therefore, all behaviors composed in parallel must synchronize on theirs **exit**s. For example:

((a; **exit**) ||| (b; c; **exit**)) is equivalent to ((a; b; c; **exit**) [] b; (a; c; **exit** [] c; a; **exit**)) and both of these behaviors are ready to synchronize with the environment on any of these following sequences: a b c δ, b a c δ, b c a δ.

The *full synchronization* operator denoted by '||' is used when the processes involved in synchronization must synchronize on every observable action. Clearly, when L is the list of all gates, the *selective composition* operator becomes identical to the *full synchronization* operator. For example:

(a; b; c; **exit**) can only synchronize with the behavior (a; b; c; **exit**).
Therefore, ((a; b; c; **exit**) || (a; b; c; **exit**)) is equivalent to (a; b; c; **exit**).
((a; b; **exit**) || (d; a; c; **exit**)) is equivalent to **stop** (deadlock!) because the left hand side offers action a while the right hand side offers action d.

The parallel composition operators '||' and '|||' are commutative and associative. |[L]| is commutative and may be associative depending on the gates in L.

I.4.5 Disable Operator '[>'

The LOTOS *disable* operator denoted by '[>' models an interruption of a process by another process. So, P1[>P2 means that at any point during the execution of P1, there is a choice between executing the next action of P1 or the first action of P2. Once the first action from P2 is chosen, P2 continues executing while P1 is no longer possible. However, if P1 terminates successfully, then P2 does not start execution. For example:

((a; b; **exit**) [> (c; d; **stop**)) is equivalent to

```
(   a;
        (   b;
                (   exit
                 [] c; d; stop
                )
        [] c; d; stop
        )
)
[] c; d; stop
```

I.4.6 Hiding Operator

The *hide* operator allows abstracting from the internal functioning of a process, by hiding actions that are internal to it. In particular, when the top-down approach is used, the designer can compose the system using LOTOS operators while hiding the details of inter-process communications that are irrelevant at a higher level of abstraction. Here is an example of the *hide* operator:

((**hide** b **in** a; b; **exit**) || (a; c; **exit**)) is equivalent to
(a; **i**; c; **exit**) because b is turned into an internal action which does not synchronize.

I.5 Process Instantiation

An *Instantiation* is an invocation of an already defined process. Actual gates and values are used to instantiate the formal gates and parameters of the process definition. Recursivity can be achieved by having a process instantiate itself. For example:

Let P [fg1, fg2,..., fgn] (p1:s1, p2:s2,...,pm:sm).

If B= P [ag1, ag2,...,agn] (v1, v2,..., vn) then B is a process which behaves like P with the actual gates ag1, ag2,...,agn renaming the formal gates fg1, fg2,..., fgn and the actual values v1, v2,..., vn replacing the formal parameters p1:s1, p2:s2,...,pm:sm

II A model for specifying Telephone Systems

LOTOS has been conceived as a tool for the formal description of OSI services and protocols. But the concepts are general enough to make the language useful for a wide range of applications. In particular, LOTOS is appropriate for Telephone System (TS) specification. The present purpose is to specify a simple Plain Old Telephone System (POTS) by covering all its features but only from a service point of view. In other words, the specification describes only the end-to-end services provided to users in an abstract, concise and implementation independent fashion, and without dealing with the details of internal functioning of the POTS system.
Several concepts originally developed for the description of data communication protocols and services are also appropriate for the description of TSs. For example, in a service specification, only the external behavior of the system is captured, that is describing what the system does for the user and not how it does it. There are however important differences between TS service specification and OSI service specification. On the other hand, LOTOS at the present stage, can not deal with timing aspects: it would be impossible to exactly portray a specification element such as: “the telephone can only be off hook for a maximum of 20 seconds after which it will be disconnected.”

II.1 Issues of Specification style for TS specification

Four main styles have been identified for writing LOTOS specifications: the *monolithic* style, the *state-oriented* style, the *resource-oriented* style and the *constraint-oriented* style. Each one has its own use in TS specification and they can be mixed.

- The *monolithic* style gives explicitly all possible sequences of actions allowed by the specification. The main operator is *choice* ‘[]’ and the specification is shown as a tree of choices. This style is useful for debugging the specification and generating test sequences. A well-known basic result in LOTOS theory is the expansion theorem by which any LOTOS specification can be transformed into a monolithic one.
- In the *state-oriented* style, explicit system states are identified by using state variables. Although the state-oriented style is quite tedious if used throughout a specification, it may occasionally be useful to introduce state variables that identify the state of some devices. This may lead to increase readability of the specification in case where the informal specification uses the state concept, as it is quite common for telephone devices. It may also lead to LOTOS specifications which can be implemented directly. In this work, state-oriented style is avoided because we want to produce an abstract specification without explicit reference to device states.

- In the *resource-oriented* style, the processes are chosen in such a way as to represent resources, which means implementation modules. This style is useful for implementation specification.
- The *constraint-oriented* style is the most abstract specification style since it focuses on event sequencing and logical constraints as seen from the external interaction points. In this style, processes identify constraints and usually have little or no relation with implementation processes. This style is, therefore, useful for implementation independent specifications. One can identify two main types of constraints: *event sequence* constraints and *value* constraints. The former are expressed by mechanisms of pure LOTOS, while the latter require values as well. The two main operators used to express process composition are the *interleaving* operator '|||' and the *full synchronization* operator '||'. That is, (A|||B)||C means that A and B can freely interleave, while they must synchronize with C.

II.2 The specification

In this section, we present the design method of our specification, explain its structure, and discuss our solutions to some key design decisions. But first, an informal description of POTS is in order.

II.3 The informal description

First, we introduce the terminology that will be used: *system* refers to the local telephone branch exchange. A *telephone* has two aspects: a *Caller* side and a *Called* side. Each telephone is identified by a unique number, which we call the *telephone number* (or simply *number*). A *number* may be connected to more than one telephone *extension*, but only one *extension* may be used at a time. Intuitively, telephone users communicate with each other through a black box (POTS) and they interact with POTS by using a well defined set of primitives.

Fig 2. reflects a scenario where the caller establishes a connection with a non-busy called. For further refinement of this diagram, we add more information in Fig 3. resulting in a timing diagram that shows all possible interactions from a user's point of view. In this specification, only two users per connection are considered.

Our specification allows for an arbitrary number of users to access the TS and to communicate with each other. The Telephone System works in the following manner:

The first user, the *Caller*, picks up the handset. If no other extension for the same number is in use, the network responds by sending a *Tone* signal to the user. The *Caller* is now in a position to *dial* the number of the second user, the *Called*. When the *Caller* completes

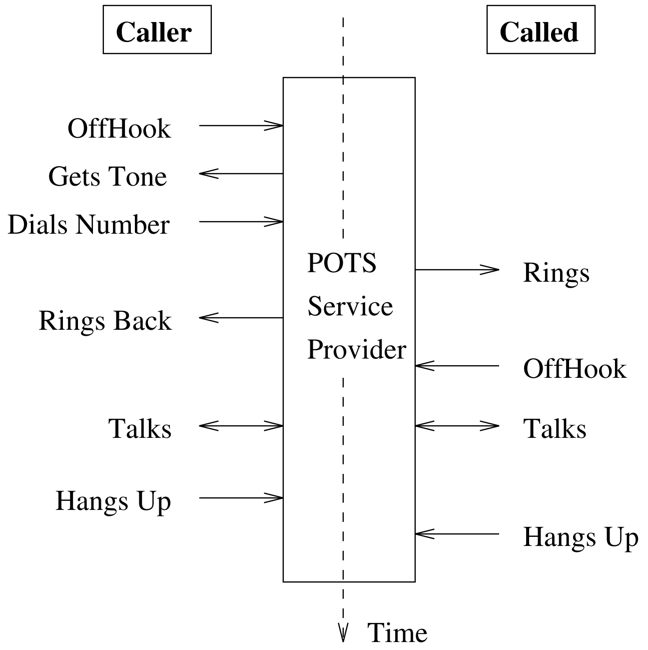


Fig 2. Structure of a POTS communication.

```
1  Caller OffHook
|  1  Caller GetsTone
|  |  1  Caller Dials Called
|  |  |  1  Caller GetsBusySignal
|  |  |  |  1  Caller HangsUp
|  |  |  2  Called Rings
|  |  |  |  1  Caller RingsBack
|  |  |  |  |  1  Called OffHook
|  |  |  |  |  |  1  Caller TalksTo Called
|  |  |  |  |  |  2  Called TalksTo Caller
|  |  |  |  |  |  3  Caller HangsUp
|  |  |  |  |  |  4  Called HangsUp
|  |  |  |  |  2  Caller HangsUp
|  |  |  |  2  Caller HangsUp
|  |  |  3  Caller HangsUp
|  |  2  Caller HangsUp
|  2  Caller HangsUp
```

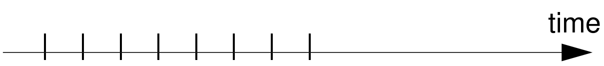


Fig 3. Sequence of events of a POTS communication.

dialing the number, the network checks if the *Called* number is free, and if so, a *Ring* signal is sent to the second telephone. Otherwise, a *BusySignal* is sent to the *Caller*. If the *Called* user does not pick up the handset to answer, the *Caller* will eventually hang up and both telephones will be free. However, if the *Called* picks up the handset, the telephone stops ringing and the two parties engage in a conversation. When the conversation is finished, either party may hang up. The first user who hangs up makes his/her telephone free to make or receive other calls. The second telephone remains busy until the user hangs up.

II.4 Structure of the specification

The graphic structure of the LOTOS specification is shown in Fig 4.

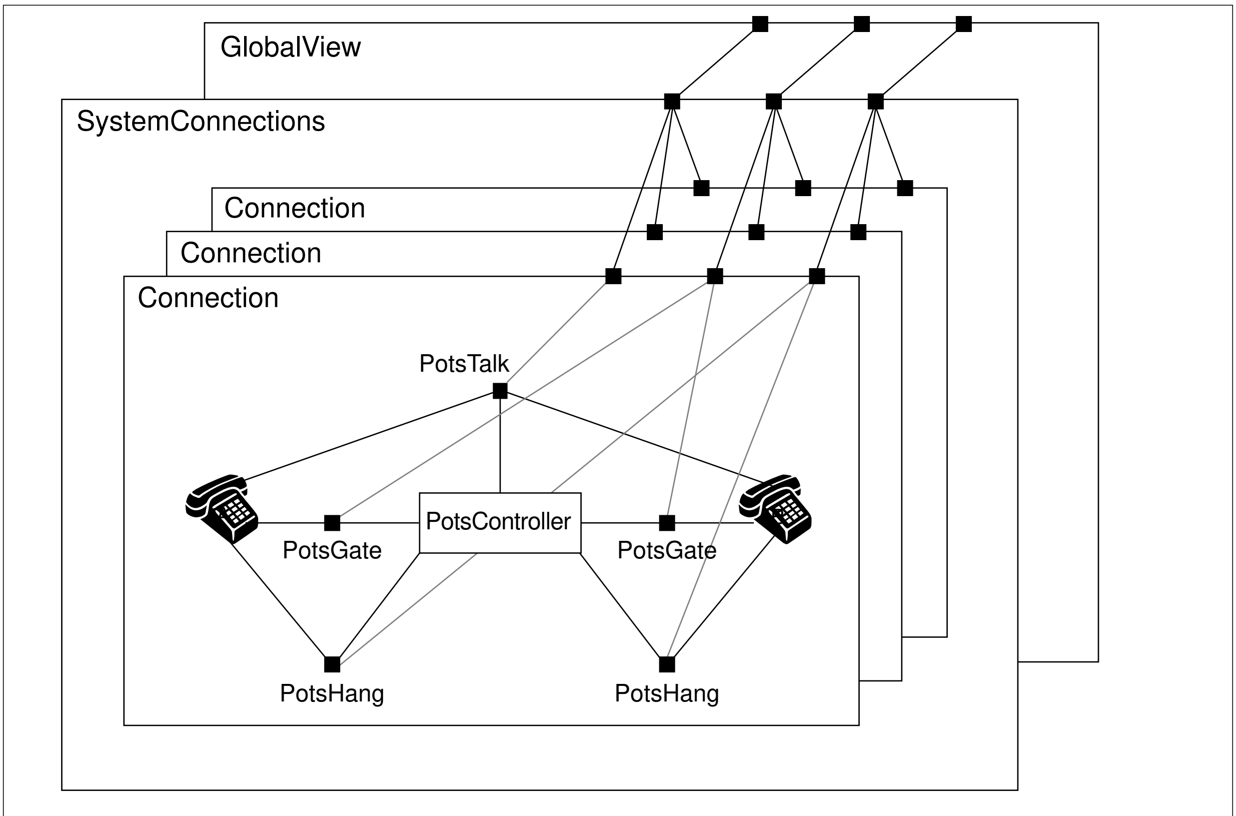


Fig 4. Graphic structure of the POTS specification.

In this figure, we identify *processes* and *gates*. We represent a process of a caller side and a called side by icons of common household telephone sets. We represent a controller process between a caller side and a called side by large white squares. LOTOS gates are identified by small black squares. Processes and gates are identified by unique names. Figure 4 shows the structure of a connection between two users *A* and *B*. Process *A* and the controller process *PotsController* synchronize on three gates: *PotsGate* is used to exchange signals for establishing a connection between user *A* and the controller, *PotsTalk* is used during the talking phase between users *A* and *B*, and finally, *PotsHang* is used during the disconnection phase. Note that *PotsTalk* is a three way synchronization point, whereas the two other gates are two way synchronization points; intuitively, the talking phase requires all three processes to participate in a 'talk' action. The connection and disconnection phases require, at any given time, the participation of only one of the two communicating sides and the controller. The composition of these three processes is labelled *Connection*. Note that the process *Connection* is embedded in process *SystemConnections* which allows the creation of as many connections as desired. *GlobalView*, which we will describe later, is a process which synchronizes with *SystemConnections* on every gate. To make the next figures simpler, the *GlobalView* process will not be shown, but it will be always implicit in the structure.

II.5 Behavior of the specification

The top levels of the LOTOS specification are shown in Fig 5.

```
behavior
(   SystemConnections [PotsGate,PotsTalk,PotsHang]
    ||
    GlobalView [PotsGate,PotsTalk,PotsHang]
)
where
    process SystemConnections [PotsGate,PotsTalk,PotsHang]:=
    (   Connection [PotsGate,PotsTalk,PotsHang]
        |||
        i; SystemConnections [PotsGate,PotsTalk,PotsHang]
    )
    endproc (* SystemConnections *)

    process Connection [PotsGate,PotsTalk,PotsHang]
    (   (   Caller [PotsGate,PotsTalk,PotsHang]
            |[PotsTalk]|
            Called [PotsGate,PotsTalk,PotsHang]
        )
        ||  PotsController [PotsGate,PotsTalk,PotsHang]
    )
    endproc (* Connection *)
```

Fig 5. Top level structure of the POTS specification.

LOTOS makes it possible to describe the behavior of systems in a stepwise fashion, moving from one abstraction level to another. This is a powerful technique since, at each level, it is possible to describe the level's architecture completely. Using this method within the framework of the constraint-oriented specification style, three types of constraints can be identified:

- *Local constraints* are used to enforce the appropriate sequences of events at each telephone, and are different according to wether the telephone is a *Caller* or a *Called*. Therefore, local constraints are represented by the processes *Caller* and *Called*, and an instance of each of these is associated with each telephone existing in the system. Because these two processes are independent of each other except for the talking phase, they are composed by the selective parallel operator '|[PotsTalk]|'.
- *End-to-end constraints* are related to each connection, and enforce the appropriate sequence of actions between telephones in a connection. For example, ringing at the *Called* must necessary follow dialing at the *Caller*. Process *PotsController* enforces these constraints. Because they must apply to both *Caller* and *Called*, we have the structure ((Caller |[PotsTalk]| Called) || PotsController). Thus, the controller must participate to every action of the *Caller*, as well as every action of the *Called*, separately.
- *Global constraints* are system-wide constraints. In our specification, one main *value constraint* is identified, which is the fact that at any time, a *number* is used at most once. This constraint is enforced by the process *GlobalView*. Because global constraints must be satisfied simultaneously over the whole system, represented by the process *SystemConnections*, we have the structure (SystemConnections || GlobalView).

II.6 Description of processes

In this section, we first give the details of the four most important processes which make up the specification: *Caller*, *Called*, *PotsController*, *Globalview*. Then we present the abstract data types used to maintain the system users. It is important to keep in mind that every action, except the **i**, is a multi-way rendezvous of these three behaviors:

- (Caller |[PotsTalk]| Called)
- PotsController
- GlobalView

II.6.1 SystemConnections

The global constraints are enforced by the composition of the processes *GlobalView* and *SystemConnections*, shown in the top level of the specification's structure. *GlobalView* has the exclusive use of the user's phone numbers. It interacts with *SystemConnections* to enforce the sequence of events within each connection as well as the use of the system values across the active connections. Each connection in the system is composed of three subprocesses (*Caller, Called* and *PotsController*).

II.6.2 Caller side

Local constraints are expressed as a sequence of events that must take place locally, either at the *Caller* side or at the *Called* side. Given the set of observable actions: *OffHook-ToCall*, *GetsTone*, *Dials*, *RingsBack*, *TalksTo*, *GetsBusySignal*, *HangsUp*, the behavior of the *Caller* is specified so that only valid sequences are executed for each instance of the process *Caller*. For example, the *Caller* cannot dial a number before the *Tone* is received. In addition, the event *HangsUp* may disrupt anywhere after the action OffHookToCall.

```
process Caller [PotsGate,PotsTalk,PotsHang]: exit(Nat):=
(   PotsGate ?Caller:DecDigit !OffHookToCall ;
    (   (   PotsGate !Caller !GetsTone ;
            PotsGate !Caller !Dials ?Called:DecDigit ;
            (
                    CallerGetsBusySignal [PotsGate] (Caller)
                []
                    PotsGate !Caller !RingsBack ;
                    UserTalks [PotsTalk] (Caller)
            )
        )
        [> UserHangsUp [PotsHang] (Caller)
    )
)
```

Fig 6. Process Caller

II.6.3 Called side

At the other end of the connection, the *Called* process is responsible for enforcing its local constraints using another set of actions, which consists of *Rings*, *OffHookToAnswer*, *TalksTo* and *HangsUp*. Again, *TalksTo* may only be offered after *OffHookToAnswer* has occurred. Also, the action *HangsUp* may occur only after *OffHooktoAnswer* has occurred.

```
process Called [PotsGate,PotsTalk,PotsHang]: exit(Nat):=
(    PotsGate ?Called:DecDigit !Rings ;
     (    PotsGate !Called !OffHookToAnswer ;
          (    UserTalks [PotsTalk] (Called)
               [> UserHangsUp [PotsHang] (Called)
          )
          [] exit (0 of Nat)        (* before reaching talk *)
     )
     [] exit (0 of Nat)             (* before reaching talk *)
)
```

Fig 7. Process Called

II.6.4 Controller

The role of the *Controller* is to enforce the-to-end constraints. It combines the independent sequences of the *Caller* and the *Called* into a single ordered sequence of events. Starting from the initial state and for all the subsequent states, the *Controller* synchronizes either with the caller side or the called side. An interesting ramification of the *HangsUp* event is that the *Controller* must deal with according to its source as well as its point of occurrence. Because of the recursion in the system, a deadlock may be unintentionally introduced if this event is not handled properly. By properly, we mean that the connection must be returned to its initial state regardless of wether the *HangsUp* occurred on one side only or both sides. As shown in figure 9, proper synchronization of the controller with the two sides ensures that the connection terminates successfully. In our specification, we distinguish three cases of the *HangsUp* event.

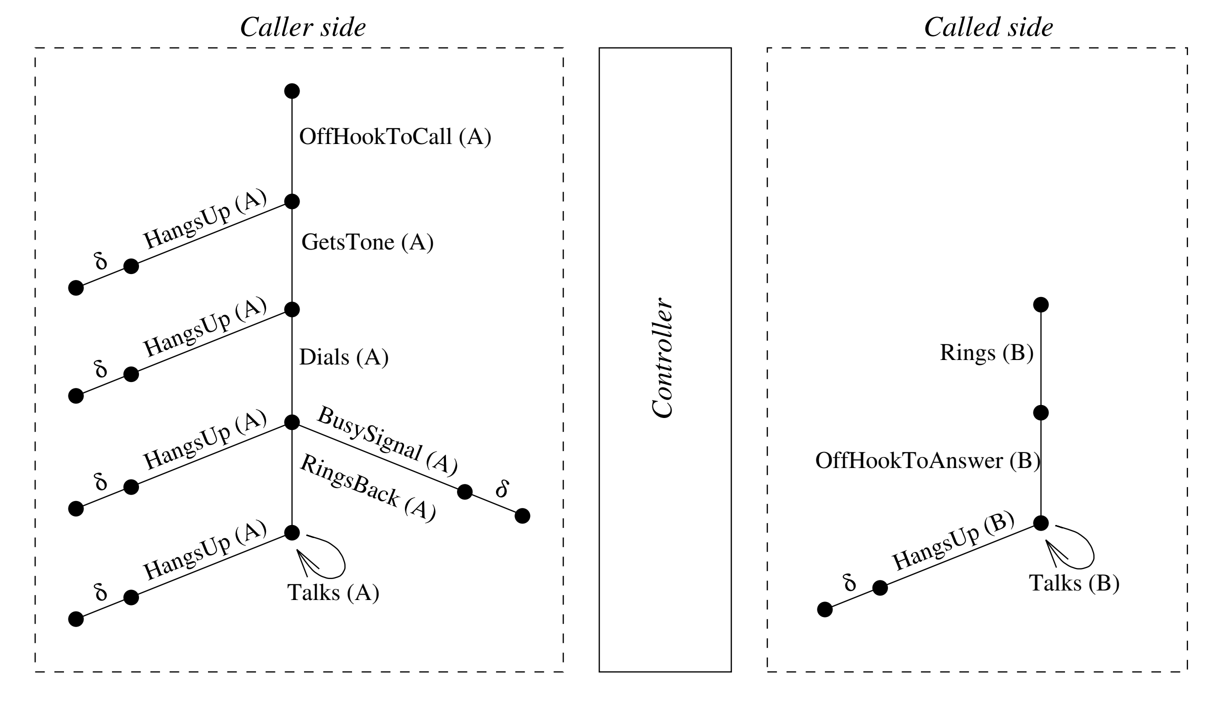


Fig 8. Formal specification of *Caller* and *Called*.

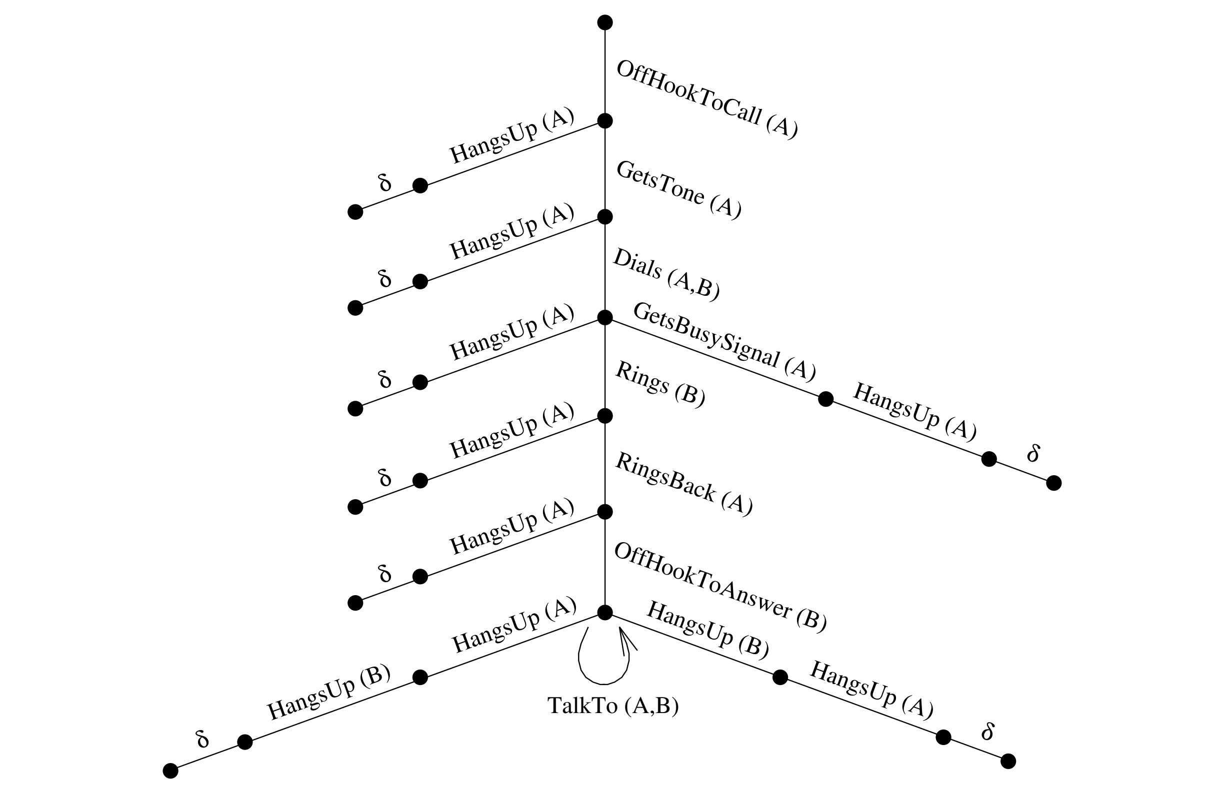


Fig 9. Formal specification of *Controller*.

II.7 GlobalView

The process *GlobalView* manipulates the data structures of the specification. It synchronizes with *SystemConnections* in order to update the list of busy numbers.
There is a trade-off between how much information should be exchanged, through processes synchronization, and how much of it should be kept as part of the abstract data types. Choosing the 'right' data structure is not a simple task. For this specification, our choice is influenced by our objective to focus the reader's attention on the constraint-oriented style, by keeping the processes simple while exchanging the minimum information required through the service interactions. For example, the only event that requires both the caller number and the called number to be exchanged in the service interaction is the *Dial* event. All other events are associated with either the caller number or the called number. This has the advantage of simplifying the three major processes *Caller*, *Called*, and *Controller*. But on the other hand, it complicates the data structure that has to be used. For instance, in order for the caller to receive a *BusySignal*, we must check wether the called number is busy or not. Since the interaction *GetsBusySignal(Caller)* does not contain the number of the called user, we must keep this information inside the process *GlobalView*, by means of a set of pairs. So, our first data structure *EngagedSet* (or *E* for short) of sort *SetOfPairs* is a set of pairs (X,Y) where X equals Caller or Called, Y equals Called or None, and (X,Y) is different from (Called,Called). In other words, the first element of a pair is a number which has executed the *OffHook* event. If it is a Caller, then the pair (Caller,None) is added to the set. If it is a Called, then the pair (Called,None) is added to the set. The difference between these two pairs is that the first one will be modified to become (Caller,Called) when the *Dials* event is executed, while the second pair remains (Called,None) until it is removed as a result of the *Called* executing the *HangsUp* event.
If several callers execute the *Dials* event while attempting to call the same number, only one of them will succeed to make the *Called* ring. Therefore, we must remember which numbers have executed the *Rings* event so that they may not ring again for another caller. To do so, we define a second data structure *BusySet* (or *B* for short) of type *DecSet* which is a set of phone numbers of the called users. So, if a called user is not busy, the *Rings* event is executed and the associated number is added to *BusySet*. However, if the called number is busy, then the caller must receive a busy signal. Again, a called number is busy if it has executed the *Rings* event, as just stated, or if it is the first element of a pair in *EngagedSet*.
Finally, there is the *HangsUp* event for which several cases must be distinguished:
First, if the event is associated with a called number, then both the *Rings* event and the *OffHookToAnswer* event must have been executed. Consequently, the called number must appear both in *EngagedSet* as a pair (Called,None) and in the *BusySet* as Called. The called

number is removed from both sets. Otherwise, if the event is associated with a caller number, we must distinguish the cases where the caller may hang up:

- before the *Dials* event is executed, the called number is not identified and therefore removing the pair (Caller,None) from *EngagedSet* is sufficient.
- after the *Dials* event is executed, but before the *Rings* event is executed, the called number is already identified and removing the pair (Caller,Called) from *EngagedSet* is sufficient.
- after the *Rings* event is executed, but before the *OffHookToAnswer* event is executed, the called number associated with the caller number which has hung up is already in the *BusySet* and it is not sufficient to remove the pair (Caller,Called) from the *EngagedSet*, but we must also remove the called number from the *BusySet*.
- after the *OffHookToAnswer* event is executed, the called number appears three times in the two sets as (Caller,Called) and (Called,None) in *Engagedset*, and as Called in *BusySet*. Therefore, when (Caller,Called) is removed from *EngagedSet* and Called is removed from *BusySet*, there is still an occurrence of (Called,None) in *Engagedset* which guarantees that the called number is still busy.

III Basic Concepts for Specifying Telephone Features

The Telephone System specification described in chapter II is the result of the work of the LOTOS research group from the University of Ottawa. It represents my first contact with the specification language LOTOS and it constituted the foundation of my work, the model to follow. I started in this way specifying Telephone Features by applying the specification's concepts developed in this former work, but I realized gradually that some rules, some choices made in this model were quite complicate, tedious for specifying Telephone Features, or even useless. So, I present in the next sections my reflection's results about the model for specifying Telephone Systems, which constitute the basic concepts I will use for the specification of Telephone Features described in chapter IV.

III.1 The specification's main structure

The structure ((Caller |[PotsTalk]| Called) || PotsController) used to specify a connection in a LOTOS constraint-oriented style modelizes in a comprehensive way the interactions between the users and the Telephone System: every user is represented by a process (wether *Caller* or *Called*) and the TS is represented by the process *PotsController* which makes the link between both users in a connection in order to synchronize their actions.
But we can observe that the execution of this structure is exactly identical to the execution of the process *PotsController* alone. So, in the next feature's specifications, the process *Connection* will be assimilated to the process *PotsController*, and the processes *Caller* and *Called* will never exist. The advantages are the following;

- less work: only one process will be specified instead of three.
- no synchronization's problem between the three processes *Caller*, *Called*, *PotsController*, especially on their **exit**s.

III.2 Structure of a POTS communication

Even if the high levels of the specification will be simpler and the processes *Caller* and *Called* will never exist, we will keep the graphic representation used in Fig 4. which describes in a clear way the LOTOS specification's structure of a connection. In this modelisation, the gate *PotsTalk* is a 3 way synchronization point, whereas the gates *PotsGate* and *PotsHang* are 2 way synchronization points. Because these two latter gates have the same role, we will only keep the gate *PotsGate* and remove the gate *PotsHang* from the specification. Therefore, every action for establishing and disconnecting the communication will occur on the gate *PotsGate* except for the talking phase which will occur on the gate *PotsTalk*.

On the other hand, we can remark that the gates in Fig 4. are divided: *PotsGate* and *Pots-Hang* appear twice in process *Connection* whereas in fact, every gate is unique in the whole system, and it is common for all processes' instantiations. This can imply unintentional interactions between instantiations which have common gates, but which shouldn't synchronize together. So we must be very careful of the structure of actions in order to make them unique and discernible in the whole system and to avoid unintentional interactions. (see next paragraph)

III.3 Structure of actions and exchange of data

In paragraph II.7, the choice for specifying the Telephone System was to keep the processes simple while exchanging the minimal information required through the service interactions. So the only event that required both the caller number and the called number was the dial event. All other events were associated with either the caller number or the called number. The problem in this choice is that the actions may be not unique in the whole system and we can easy get some interaction's problems. For example, let assume that users A and B try to call the same user C at the same time.We can have in this way the action *Rings(C)* twice (from A and from B) and the choice between both is non deterministic. If we transform this action into *RingsFrom(C,A)* and *RingsFrom(C,B)*, both actions become discernible and we will never have any problem of interaction.
Completing an action by adding values like the telephone numbers of the users implied in this action allows to exchange more information between the synchronizing processes and it does not complicate their structure: it makes the LOTOS actions more comprehensible and it makes the manipulation of the abstract data types easier in the process *GlobalView*: every element of a list which must be updated after the execution of an action refers to users which telephone numbers are communicated to process *GlobalView* through this action.

III.4 Process GlobalView and abstract data types

GlobalView synchronizes on every action with *SystemConnections* and it is used to update the lists of users which describe the state of the Telephone System at any moment. The lists defined for TS specification are of two types:

- type *DecSet* which defines a list of telephone numbers (*DecDigit*s)
- type *SetOfPairs* which defines a list of number's pairs (*PairOfDecDigits*)

For manipulating these lists, we must first define some operators which allow to add or remove an element from a list and to avoid dangerous situations such as having the same element several times in the same list. To do so, we need three operators (*Insert*, *InsertAtTheEnd*, *Remove*) for manipulating the lists of type *DecSet*, and three operators (*InsertPair*, *InsertPairAtTheEnd*, *RemovePair*) which are similar, for manipulating the lists of type *SetOfPairs*. Both sets of operators are defined as follows:

```
for all x,y : elements
        L   : list

Remove (x,{}) = {} ;
Remove (x, Insert (x,L)) = Remove (x,L) ;
(x≠y) ⇒ Remove (x, Insert (y,L)) = Insert (y, Remove (x,L)) ;

InsertAtTheEnd (x,{}) = Insert (x,{}) ;
InsertAtTheEnd (x, Insert (x,L)) = InsertAtTheEnd (x,L)) ;
(x≠y) ⇒ InsertAtTheEnd (x, Insert (y,L)) = Insert (y, InsertAtTheEnd (x,L)) ;
```

Such a definition allows to have only one occurrence of the element *x* in the list *L* after having executed *InsertAtTheEnd (x,L)* (if the element is already in the list, all its occurrences are removed from the list and it is added at the end), and to remove all the occurrences of the element *x* from the list *L* by executing the single operation *Remove (x,L)*. These two operators will so allow us to update properly the lists of users in the process *GlobalView*.

On the other hand, the description of this process in paragraph II.7 and the data type structure associated (list *EngagedSet* and list *BusySet*) present two main disadvantages:

- First, the treatment of the *HangsUp* event is quite complicate, because it depends on wether the user is a caller or a called, and if it is a caller, four cases must be considered! When we will specify Telephone Features, which allow for example a user to put another user on hold, or to be recall automatically and so on, we can imagine the complexity and the number of cases that we will have to resolve if we keep this data structure for updating the state of the system.
- Secondly, if the caller hangs up after the *Rings* event but before the *OffHookToAnswer* event, the process *GlobalView* must remove the called number from *BusySet*. In this case, an action executed by a user (the caller) must update a list (*BusySet*) by removing an other user (the called) without having a direct link between both users in the list. This technique may lead to cases very complicate to resolve and we will try to avoid it in the next specifications, what means that an action executed by a user will update some lists only by adding or removing elements where this user is present.

Such considerations lead us to define two new lists for describing the state of the POTS Telephone System:

- *BusyList*, of type *DecSet*, is a list of the telephone numbers which are off hook.
- *WaitPairs*, of type *SetOfPairs*, is a list of pairs (X,Y) where X is ringing from Y.

These two lists are updated by process *GlobalView* as follows:

- When user X picks up to call or to answer a call, he is added to *BusyList*. When he hangs up, he is removed from *BusyList*.
- When the action *RingsFrom(X,Y)* happens, the pair (X,Y) is added to *WaitPairs*, for explaining the fact that Y is waiting for speaking with X. This pair is removed from *WaitPairs* only if X picks up to answer Y (action *OffHookToAnswer(X,Y)*) or if Y hangs up before (action *HangsUp(Y)*).

These two rules are now very simple to implement and they won't have to be modified for the next Telephone Features specifications. On the other hand, some actions can be executed only if some conditions about the user lists are verified. These conditions are named 'guards':

– the action *OffHookToCall(X)* can be executed by process *GlobalView* for synchronizing with *SystemConnections* only if X is not in *BusyList* (if not, he would be already off hook) and if there is no pair (X,Y) in *WaitPairs*, what would mean that X is ringing from Y.
– The action *RingsFrom(X,Y)* can be executed only if X is not in *BusyList* and if there is no pair in *WaitPairs* where X is the first element (what would mean that X is already ringing from another user). In the opposite case, the action *GetsBusySignalFrom(Y,X)* is executed.

III.5 Treatment of Telephone Signals

In LOTOS, two ways for specifying a continuous signal have been identified. Their temporal representation are shown in Fig 10.
The first way modelizes the signal by repeating the same action (the action 'signal ') as long as the signal is on. The second one uses an action for moving from state *off* to state *on* (action of signal starting) and an other action for moving from state *on* to state *off* (action of signal ending). The second modelisation is better for two reasons:

- the first modelisation uses events which are in LOTOS atomic and instantaneous by definition (see paragraph I.2) for representing a continuous signal (odd?) whereas the second one uses two events for moving from a state to another state (more rational).
- the tree of actions shown in Fig 10. is more complicated to implement in the first modelization than in the second one.

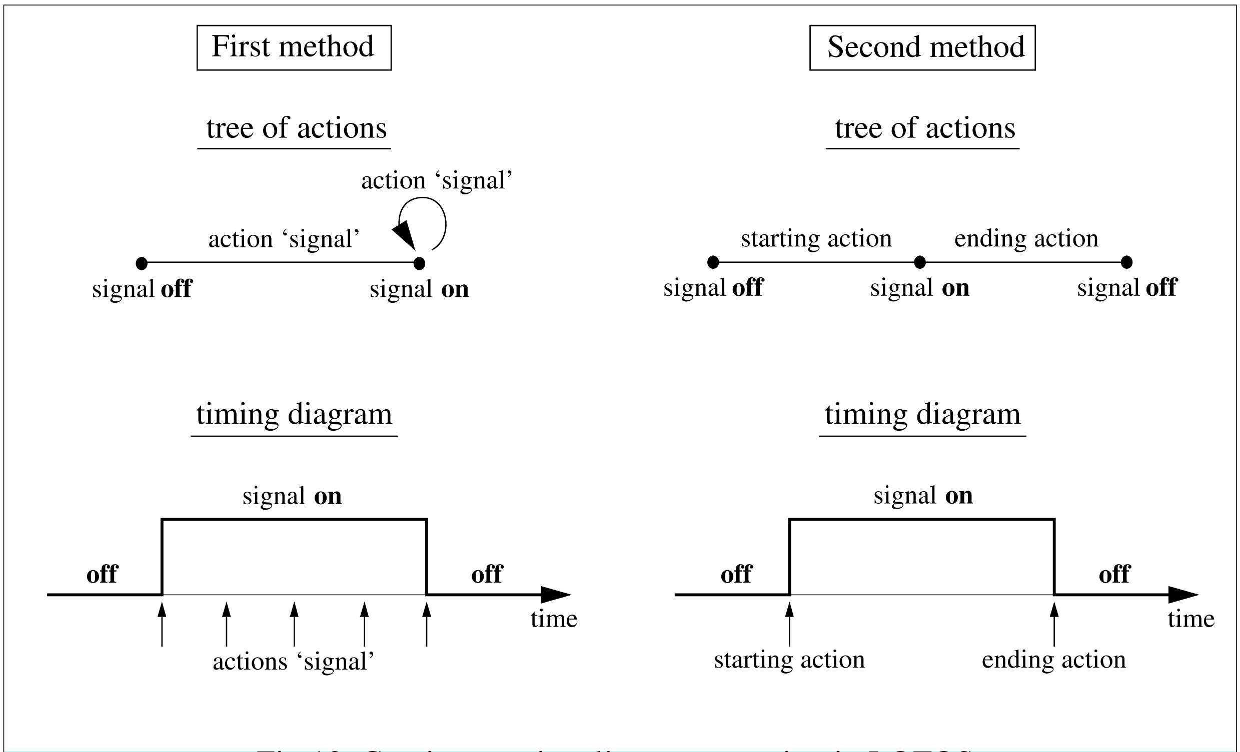


Fig 10. Continuous signal’s representation in LOTOS.

So we will keep the second way for representing the different continuous signals existing in the Telephone System, and table 1 shows the different actions that can start or stop these signals.

Table 1: Telephone Signals and their associated actions.

Signals	Starting actions	Ending actions
Tone Signal	GetsTone (Caller)	HangsUp (Caller) Dials (Caller,Called)
Ringing Signal	RingsFrom (Called,Caller)	HangsUp (Caller) OffHookToAnswer (Called,Caller) StopsToRingFrom (Called,Caller)
Back Ringing Signal	RingsBackFrom (Caller,Called)	HangsUp (Caller) OffHookToAnswer (Called,Caller)
Busy Signal	GetsBusySignalFrom (Caller,Called)	HangsUp (Caller)
Call Waiting Signal (only for Call Waiting Feature, see § IV.2)	GetsCwToneFrom (Called,Caller)	HangsUp (Caller) HangsUp (Called) FlashHook (Called)

III.6 Conventions for the representation of trees

The trees described in chapter IV will respect the next drawing conventions:

- A black circle ● represents a state of the system from which several actions can be executed, while a black square ■ represents an internal step between two events of a sequence which execution can't be interrupted. The first event is generally an action executed by the system whereas the second event is an hidden event which is used to communicate internal information between processes of the specification. So a black square is an internal, hidden step, not a state of the system, and if several events are possible just after, this is the synchronization between the involved processes which determines automatically which event will be executed (see § IV.1.4 or § IV.3.5 for more details)

- Every action or hidden event in the specification is parametrized at least by one of both users involved in the action. This technique makes them unique in the whole system and it avoids interferences between several instantiations of the same process. On the other hand, the pair (Caller,Called) becomes (X,Y) after the states where both have an identical status in the connection, and may execute the same set of actions. For example, caller side and called side play the same role when they are in talking state: both may hang up so the next action in the tree is *HangsUp (X)* where X is the first of the caller or the called who hangs up. Y represents then the second user. The pair (X,Y) may become (X',Y') and so on, if such a situation happens again. This technique makes the tree of actions simpler because two identical sub-trees, the first one parametrized by (Caller,Called), the second one by (Called,Caller), can be represented by only one sub-tree parametrized by (X,Y).

- Every branch of a tree of actions ends with either the action δ or with a state which is already present in the tree. In the first case, δ represents the internal action following an exit, used to synchronize the process with its environment, and followed by the behavior expression **stop** which specifies the fact that there is no more executable action left. In the second case, the state already present in the process is labelled with a letter in a gray square like A .

IV LOTOS Specification of Telephone Features

The two first Telephone features, Three Way Calling and Call Waiting, represent 80% of my work because they map a big modification of the POTS telephone logic and so they involve deep changes in the POTS specification structure. However, they allowed me to ask the good questions and so to establish the right foundations for the next features.

IV.1 Three Way Calling (TWC)

IV.1.1 Informal Description.

Three Way Calling is a feature which allows a user who is already involved in a conversation with a second user, to add a third user to the conversation. The subscriber of the feature must put the second user on hold by flashing the switchhook to establish the communication with the third user. He receives in this way a *Tone* signal, dials the third party's telephone number, and once the communication is established, he may flash the switchhook a second time to bring the second user back and to form the three way conversation. During this three way communication, the user with the activated feature may disconnect the last party added by a new single switchhook. Otherwise, every user can hang up, what returns the three way conversation to a two party connection between the remaining parties.
The customer with the activated feature can flash the hook a second time to bring back the second user, before talking with the third user. In this case, his TWC feature becomes free and the connection between the two users gets back to a basic two way communication. This user may also forget the second user in holding state, and hang up before flashing the switchhook to bring him back in conversation. In this case, the Telephone System will automatically recall this customer to establish again the communication between him and the user he let in holding state.

IV.1.2 Formal description of TWC

We modelize a 3 way communication by synchronizing two 2 way communications together. The static structure of this model is shown in Fig 11.
The First connection between the users A and B is specified by process *PotsConnection*. It is a basic POTS communication where user A is subscriber of the TWC feature, what means that A may flash the switchhook when he is talking to B in order to add the third user C to the conversation. The second connection between user C and the TWC feature of user A is specified by process *TwcConnection*. Both processes *PotsConnection* and *TwcConnection* synchronize on gate *TwcLink* for every common action between user A and his feature.

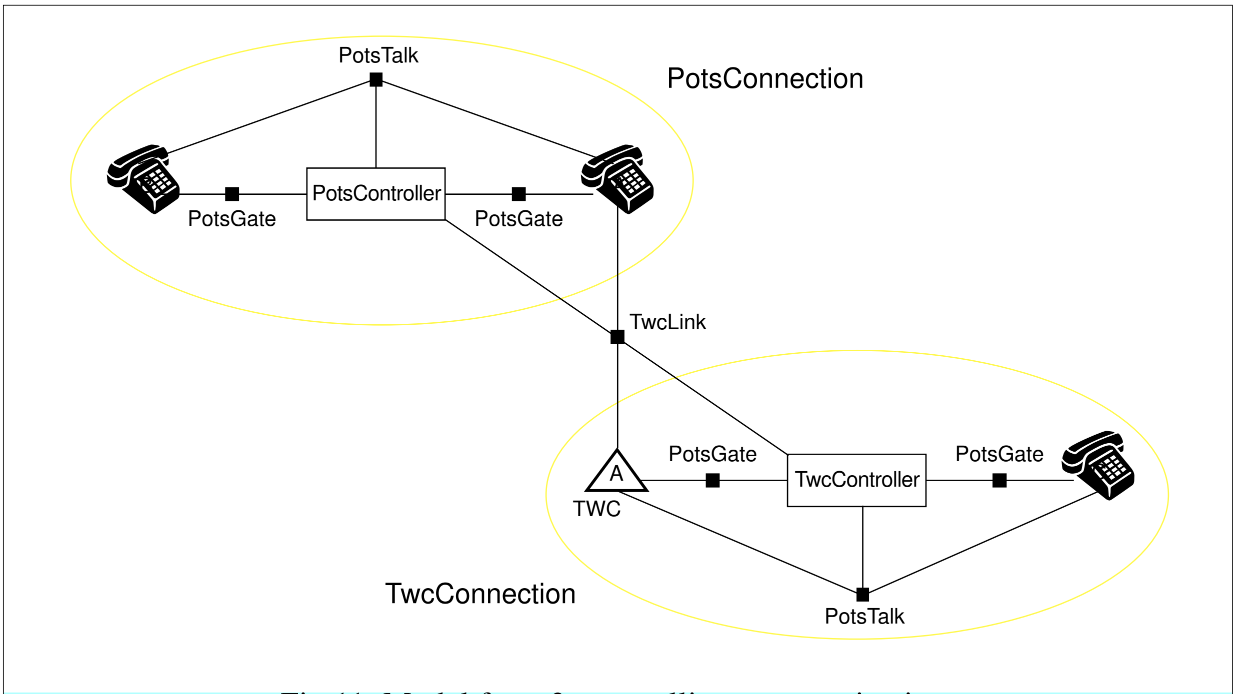


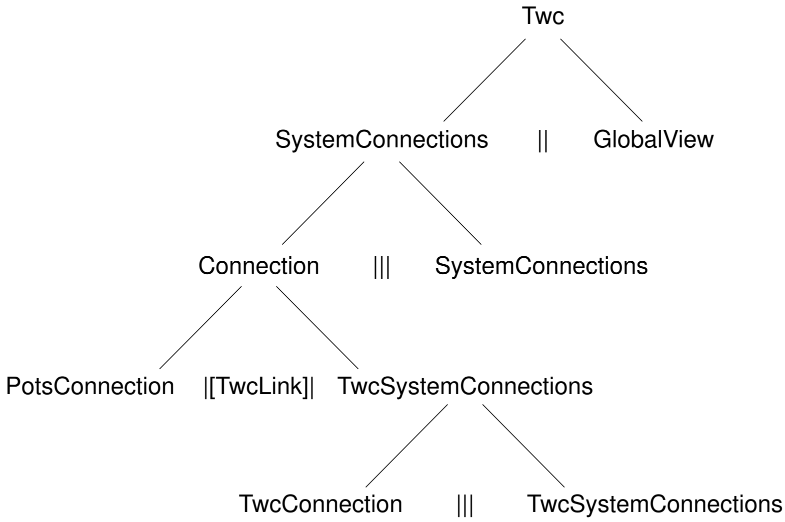
Fig 11. Model for a 3 way calling communication.

IV.1.3 Structure of the Specification

The high levels of the specification are shown in Fig 12.

We can see that the new process *Connection* is composed by process *PotsConnection* which enforces the sequences of events between both users of a POTS communication, and process *TwcSystemConnections* which allows one of both users to call a third user under condition that this user is subscriber of the TWC feature (see § IV.1.5) and that the second user didn't flash the hook before.

The process *TwcSystemConnections* is composed by process *TwcConnection* interleaved with process *TwcSystemConnections* itself. This structure allows the POTS users to call several third users during the same POTS communication. For example, when user A is talking to user B, he may flash the hook to add user C to the conversation. During the 3 way conversation, user C may hang up and user A can flash the hook again to call another third user D. We have in this way one instantiation of process *PotsConnection* parametrized by A and B, and which synchronizes successively with two instantiations of process *TwcConnection*, the first one parametrized by A and C, the second one by A and D. The structure of this execution is shown in Fig 13.




```
process Twc [PotsGate,PotsTalk,TwcLink] :noexit:=
(   SystemConnections [PotsGate,PotsTalk,TwcLink]
    ||
    GlobalView [PotsGate,PotsTalk,TwcLink]
)
endproc (* Twc *)

process SystemConnections [PotsGate,PotsTalk,TwcLink] :noexit:=
(   Connection [PotsGate,PotsTalk,TwcLink]
    |||
    i; SystemConnections [PotsGate,PotsTalk,TwcLink]
)
endproc (* SystemConnections *)

process Connection [PotsGate,PotsTalk,TwcLink]
(   PotsConnection [PotsGate,PotsTalk,TwcLink]
    |[TwcLink]|
    TwcSystemConnections [PotsGate,PotsTalk,TwcLink]
)
endproc (* Connection *)

process TwcSystemConnections [PotsGate,PotsTalk,TwcLink]
(   TwcLink ?X:DecDigit !FlashHook ?Y:DecDigit ;
    (   TwcConnection [PotsGate,PotsTalk,TwcLink] (X)
        |||
        TwcSystemConnections [PotsGate,PotsTalk,TwcLink]
)   )
endproc (* TwcSystemConnections *)
```

Fig 12. Structure of the TWC specification.

```
(   PotsConnection (A,B)
    |[TwcLink]|
    (       TwcSystemConnections
        |||  TwcConnection (A,C)
        |||  TwcConnection (A,D)
    )
)
||  GlobalView
```

Fig 13. Execution's example of the TWC specification.

IV.1.4 Processes PotsConnection and TwcConnection.

The tree of actions of process *PotsConnection* is shown in Fig 14., process *TwcConnection*'s one is shown in Fig 15.

- Process *PotsConnection* starts exactly like a basic POTS communication: the sequence of actions *OffHookToCall(Caller), GetsTone(Caller), Dials(Caller,Called), RingsFrom (Called,Called) [] GetsBusySignalFrom(Caller,Called), RingsBackFrom(Caller,Called)* and *OffHookToAnswer(Called,Caller)* can be interrupted by the action *HangsUp(Caller)* at any moment. After the called user has answered the caller, the connection is in the talking state A where both users may hang up or flash the hook if they are subscriber of the TWC feature, what is the most general case. If one of both users hangs up, he becomes user X while the second one becomes user Y and can only hang up too.
- If one of both users flashes the hook, he becomes user X and puts the other user Y in holding state. From this time, the processes *PotsConnection* and *TwcConnection* must synchronize on every action which is gray tinted in both trees.
- X may forget Y in holding state and hang up before flashing the switchhook to bring him back in conversation. In this case, X is automatically recalled by the Telephone System (action *RingsFrom(X,Y)*), even if Y has hung up too. If Y is still on line when X picks up to answer, they both return in the talking state A . If Y has already hung up, the connection moves into state B where X is ringing while Y is not on line anymore. X may also answer and get nobody on line (so he can only hang up) or the Telephone System may stop the ringing signal with the action *StopsToRingFrom(X,Y)* after a time that can't be specified in LOTOS (see § I).
- In process *PotsConnection*, X can keep in mind Y in holding state and flash the hook a second time to bring him back in conversation. The state just after the action *FlashHook (X)* is a hidden step ■ because this action is compulsorily followed by one of the four events *YNotOnLine(X), TNotOnLine(X), Y&TNotOnLine(X)* or *Y&TOnLine(X)* which are used to

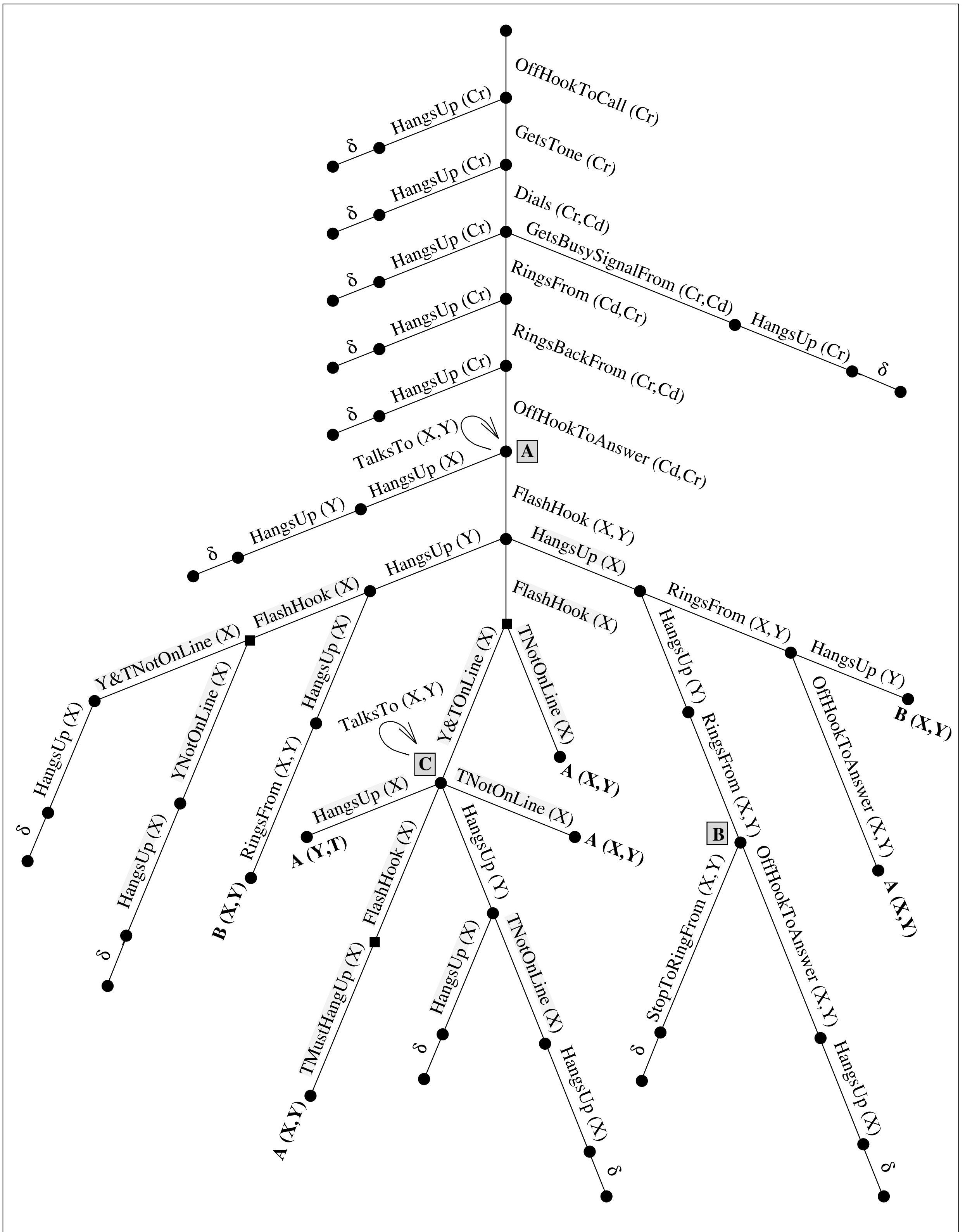


Fig 14. Tree of process PotsConnection.

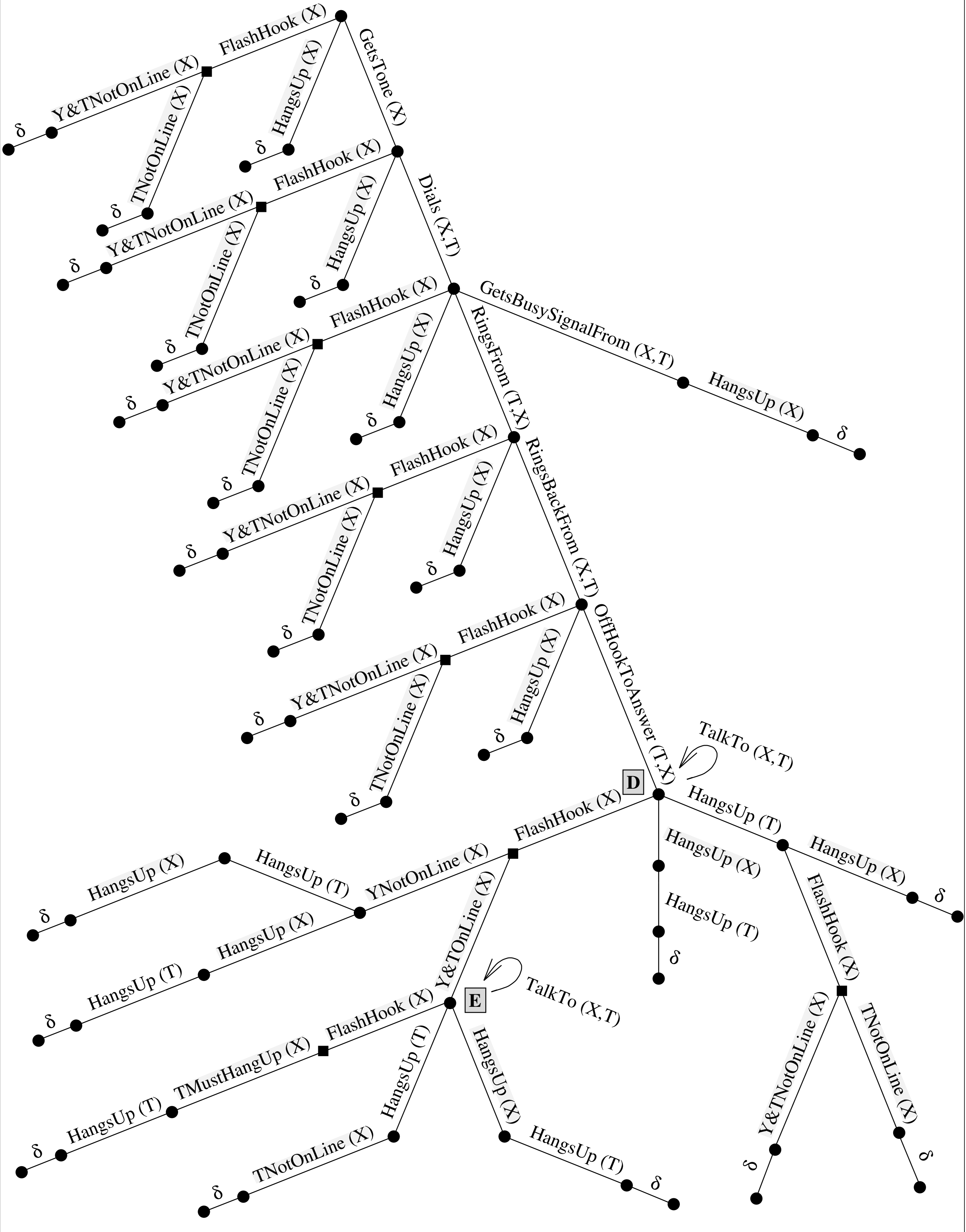


Fig 15. Tree of process TwcConnection.

exchange information between *PotsConnection* and *TwcConnection*. Actually, process *PotsConnection* “knows” if Y is still on line or not when X flashes the hook a second time, and process *TwcConnection* “knows” if the third user T is still on line or not; but both processes need information about the three users to move either into the state of 3 way conversation (state $\boxed{C}$ in *PotsConnection*, state $\boxed{E}$ in *TwcConnection*) or into the state of a single 2 way conversation if one of the three users is not on line any more. So, if Y is still on line, *PotsConnection* proposes for synchronization the events *Y&TOnLine(X)* or *TNotOnLine(X)* (and Y on line). If Y has already hung up, *PotsConnection* proposes *Y&TNotOnLine(X)* or *YNotOnLine(X)* (and T on line). For its part, *TwcConnection* proposes the events *Y&TOnLine(X)* or *YNotOnLine(X)* (and T on line) if T is still on line, and *Y&TNotOnLine(X)* or *TNotOnLine(X)* (and Y on line) if T has already hung up.
The synchronization between the two processes involves only one possible event which leads to the right states.

- If the three users X,Y,T are on line when X flashes the hook a second time, process *PotsConnection* moves into state $\boxed{C}$ of 3 way conversation. In this state, X may flash the hook a third time to remove T from the conversation. So this action is followed by the event *TMustHangUp(X)* and *PotsConnection* returns to the basic POTS conversation state $\boxed{A}$ with the users X and Y.
If X hangs up first during the 3 way conversation, *PotsConnection* returns to state $\boxed{A}$ with the users Y and T who are still on line and who participate now in a POTS communication.
If T hangs up first, *PotsConnection* synchronizes with *TwcConnection* on the event *TNotOnLine(X)* and it moves into state $\boxed{A}$ with X and Y.
At last, if Y hangs up first, *PotsConnection* waits for synchronizing with *TwcConnection* on either *HangsUp(X)* or *TNotOnLine(X)* according to X hangs up before T or not, and then it synchronizes on the exit.

- Process *TwcConnection* starts like a POTS communication, except that the first action *OffHookToCall(Caller)* is missing because it is replaced by the action *FlashHook(X,Y)* in the process *TwcSystemConnections*. At any moment before T picks up to answer X, X can disconnect the communication by hanging up (and he forgets Y on hold) or by flashing the switchhook a second time without having T on line. So the action *FlashHook(X)* in *TwcConnection* is followed either by *Y&TNotOnLine(X)* or *TNotOnLine(X)*.

- When they are in the talking state $\boxed{D}$, both users X or T may hang up for disconnecting the communication, or T may hang up and X may flash the hook to bring back Y in conversation. At last, X may flash the hook before T hangs up, what is followed by either *YNotOnLine(X)* if Y has already hung up or *Y&TOnLine(X)* which leads to the 3 way conversation state $\boxed{E}$.

- In this state, X may flash the hook a third time to remove T from the conversation or they can both hang up. If T hangs up first, the action *HangsUp(X)* doesn't appear next in *TwcConnection* for synchronizing with *PotsConnection* because the TWC feature of X is already free (see for more details § IV.1.5).

IV.1.5 Process GlobalView

The whole Telephone System is updated in process *GlobalView* by using the structure presented in § III.4 (list *BusyList* and list *WaitPairs*) and completed by three new lists related to the TWC feature functioning:

- As seen as before, *BusyList* is the list of users off hook.
- *WaitPairs* is the list of pairs (X,Y) where X is ringing from Y.
- *TwcList*, of type *DecSet*, is the list of subscribers of the TWC feature. So a telephone user X can execute the action *FlashHook(X,Y)* to call a third user only if he is in the list *TwcList*. This list is static because it is defined in the specification and it can not be modified during the execution.
- *TwcUsers*, of type *DecSet*, is the list of users which TWC feature is activated.
- *HoldPairs*, of type *SetOfPairs*, is a list of pair (X,Y) where X has put Y in holding state by flashing the switchhook, to call a third user T.

These lists are updated as follows:

- User X is added to *TwcUsers* when he executes the action *FlashHook(X,Y)* which mapped the first time X puts Y on hold to call a third user (action just after the talking state $\boxed{A}$). X is removed from *TwcUsers* when he hangs up or if he flashes the hook a second time without having the third user on line (action *FlashHook(X)* followed by *TNotOnLine(X)* or *T&YNotOnLine(X)*) or if he flashes the hook a third time, during the 3 way connection, for removing T from the conversation and making his TWC feature free (action *FlashHook(X)* followed by *TMustHangUp(X)*).

- The pair (X,Y) is added to *HoldPairs*, by the action *FlashHook(X,Y)*, when X puts Y on hold to call a third user. Y stays on hold as long as X does not flash the hook a second time to bring him back in conversation. So if X forgets Y in holding state and hangs up instead of flashing the hook, the pair (X,Y) is not removed from *HoldPairs* and X is recalled automatically by the Telephone System. The pair (X,Y) will be removed only if X picks up to answer (action *OffHookToAnswer(X,Y)*) or if the Telephone System decides to stop the ringing signal after a predefined time (action *StopsToRingFrom(X,Y)*). Even if Y hangs up while he is in holding state, the pair (X,Y) is removed only if X tries to bring him back in conversation by flashing the switchhook a second time, or by picking up to answer the automatic

recall: the pair (X,Y) is updated independently of the behavior of Y.
HolPairs is a very important list to build up the guards on the actions in *GlobalView*:

- user X can execute *OffHookToCall(X)* only if:
  - X is not in *BusyList*; if not he would be already off hook.
  - X is not the first element of a pair in *WaitPairs*; if not, he would be ringing.
  - X is not the first element of a pair in *HoldPairs*; if not, he would have forgotten Y in holding state and he could not pick up to call before the Telephone System has recalled him automatically.
- user X rings from Y (action *RingsFrom(X,Y)*) only if:
  - X is not in *BusyList*.
  - X is not already the first element of a pair (X,Z) in *WaitPairs*.
  - X is not the first element of a pair (X,Z) in *HoldPairs*: he would have forgotten an other user Z in holding state and he could not receive a new call.

In the other cases, the action *GetsBusySignalFrom(Y,X)* is executed by *GlobalView* instead of *RingsFrom(X,Y)*.

- X can execute the action *FlashHook(X,Y)* to put Y on hold only if he is subscriber of the TWC feature (if he is in *TwcList*) and if Y is not already in *TwcUsers* what would mean that Y has an activated TWC feature and he already tries to establish a 3 way communication. This last condition resolves the interaction problem between two TWC features in the same POTS connection, by preventing the second one from being activated if the first one is already in use.

IV.2 Call Waiting (CW)

IV.2.1 Informal Description

Call Waiting is a feature which allows a user in talking state, to be alerted by a Call Waiting Signal that another incoming call is waiting for being answered. The Call Waiting Signal is only audible to the line with the activated feature, while the new line receives a ringing signal. The service also provides a hold feature that is activated by a switchhook flash: consecutive flashes allow the customer with the activated feature to alternatively talk to either the original or the newly calling party, while the other is put in holding state. If this customer hangs up while one party is on hold, the system calls him back automatically, and upon answer he is connected to the held party.

IV.2.2 Formal Description

The dynamic structure used to modelize a Call Waiting feature is shown in Fig 16.

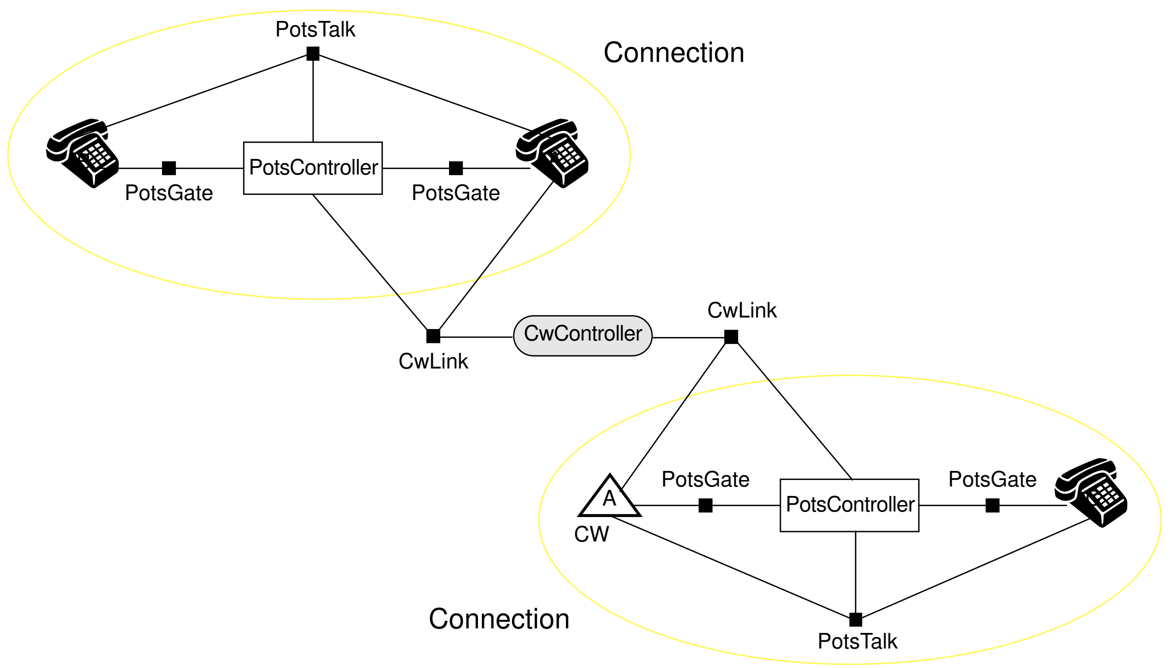


Fig 16. Model for specifying a Call Waiting Feature.

The CW feature allows two 2 way communications to have one common user, and so to synchronize together. In the first connection, the common user may be either the caller or the called side. In the second connection, the common user, or the feature which represents him, is necessarily the called side.

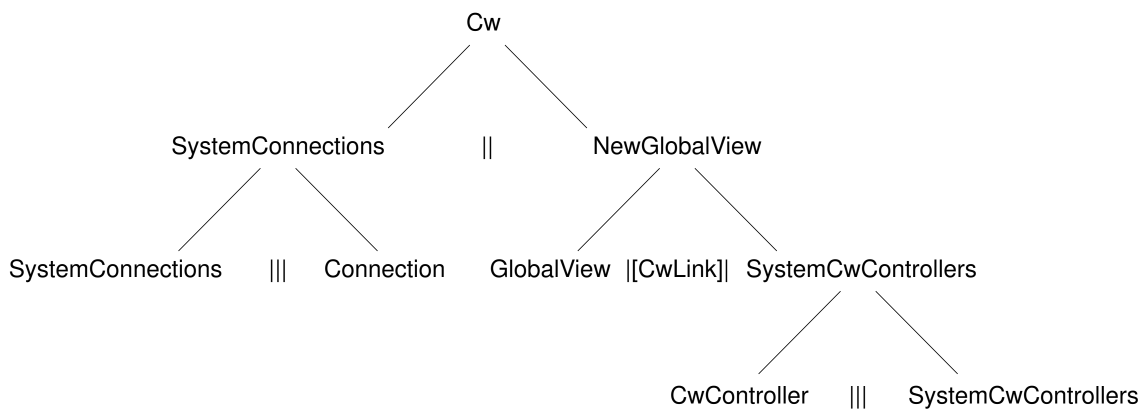
In the TWC specification seen in paragraph III.1.3, the synchronization between the two 2 way connections enforced by the processes *PotsConnection* and *TwcConnection* was static, what means that it was provided by the static structure of the specification: *PotsConnection |[TwcLink]| TwcConnection*. In the CW specification, the synchronization between the two 2 way connections is dynamic because they first are enforced by two interleaved instantiations of process *Connection*, and we decide during the execution to synchronize them together. For instance, let assume that users A and B are talking together in a POTS communication enforced by instantiation number 1 of process *Connection*. If user C picks up to call, we create the instantiation number 2 of *Connection* which is independent from instantiation number 1, and so both interleave. When user C dials, two cases must be considered: either user C dials a different number of users A and B and the two instantiations remain interleaved, or user C dials the number of A or B who is subscriber of the CW feature. In this case, we must create a dynamic link between both instantiations. Because LOTOS does not allow the creation of dynamic gates during the execution, the dynamic link for the synchronization is a new instantiation of process *CwController* which synchronizes on gate *CwLink* with alternatively instantiation number 1 and instantiation number 2 of process *Connection*. This process *Cwcontroller* enforces in fact a global constraint between two POTS communications, so its place is naturally inside the process *GlobalView*.

IV.2.3 Structure of the specification

The high levels of the CW specification are shown in Fig 17.
Process *SystemConnections* allows to create an infinite number of independent communications. The global constraints are now enforced by process *NewGlobalView* which is composed by the processes *GlobalView* and *SystemCwControllers* which synchronize together on gate *CwLink*. *SystemCwControllers* allows to create a new instantiation of process *CwController* each time a new Call Waiting feature is activated in order to synchronize two instantiations of *Connection* and to create a 3 way communication. The structure of an execution's example is shown in Fig 18. In this example, user C calls user A in talking state with user B.

```
(       SystemConnections
    ||| Connection (A,B)
    ||| Connection (C,A)
)
||
(   GlobalView
    |[CwLink]|
    (       SystemCwControllers
        ||| CwController (A)
)   )
```

Fig 18. Execution's example of the CW specification.




```
process Cw [PotsGate,PotsTalk,CwLink] :noexit:=
(   SystemConnections [PotsGate,PotsTalk,CwLink]
    ||
    NewGlobalView [PotsGate,PotsTalk,CwLink]
)
endproc (* Cw *)

process SystemConnections [PotsGate,PotsTalk,CwLink]
(   Connection
    |||
    i; SystemConnections [PotsGate,PotsTalk,CwLink]
)
endproc (* SystemConnections *)

process Connection [PotsGate,PotsTalk,CwLink]
(   PotsGate ?Caller !OffHookToCall ;
    ...
)
endproc (* Connection *)
...
process NewGlobalView [PotsGate,PotsTalk,CwLink]
(   GlobalView [PotsGate,PotsTalk,CwLink]
    |[CwLink]|
    SystemCwControllers [CwLink]
)
endproc (* SystemCwControllers *)

process SystemCwControllers [CwLink]
(       CwLink ?X:DecDigit !GetsCwToneFrom ?T:DecDigit ;
        CwLink !X !GetsTone ;
        CwController [CwLink] (X)
    |||
        SystemCwControllers [CwLink]
)
endproc (* SystemCwControllers *)
```

Fig 17. Structure of the Call Waiting Specification.

IV.2.4 Process Connection

The formal description of process *Connection* is shown in Fig 19. Fig 20. and Fig 21. This tree can be decomposed into three parts:

- from the initial state to state $\boxed{\text{B}}$, the connection is a basic POTS communication between a caller and a called.
- from state $\boxed{\text{C}}$ to state $\boxed{\text{J}}$, one of both users, denoted by X, has his Call Waiting feature activated.
- from state $\boxed{\text{K}}$ to state $\boxed{\text{Q}}$, both users X and Y have their Call Waiting feature activated. This part of the specification resolves the interaction's problems between the two features ('MUSE' interaction).

• As in the TWC tree, some states represented by ■ are internal steps between two events which execution can not be interrupted. For example, the action *FlashHook(X)* is always followed by either *OnLine(?,X)* or *NotOnLine(?,X)* on which process *Connection* synchronizes with its environment to know if the third user represented by '?' is still on line or not when X flashes the hook to talk to Y.
On the other hand, every gray tinted action is an event of synchronization between an instantiation of *Connection* and process *SystemCwControllers* or an instantiation of *CwController* (see the next example).

• In a former version, the process *Connection* was decomposed into two independent subprocesses which mapped the two cases after the dialing event: the first sub-tree started with the action *RingsFrom(Called,Caller)* and its execution mapped the case of a basic POTS communication between a caller and a called. The second sub-tree started with the action *GetsCwToneFrom(Called,Caller)* and it mapped the case where the called is subscriber of the Call Waiting feature and he is already in communication with a third user. But I realized that both sub-trees had similar sequences of actions and they could be integrated into an unique and more compact tree. So after the event *GetsCwToneFrom(Called,Caller)*, the tree moves into states which are common with the tree following the action *RingsFrom(Called,-Caller)*.
Every state of process *Connection* maps a state of the pair (Caller,Called) in this connection:

- State $\boxed{\text{A}}$ follows the dialing event *Dials(Caller,Called)*. The Telephone System must decide which action between *RingsFrom(Called,Caller)*, *GetsBusySignalFrom(Caller,Called)* or *GetsCwSignalFrom(Called,Caller)* it must execute according to the state of the called user. This choice is made by process *GlobalView* (see § IV.2.5).

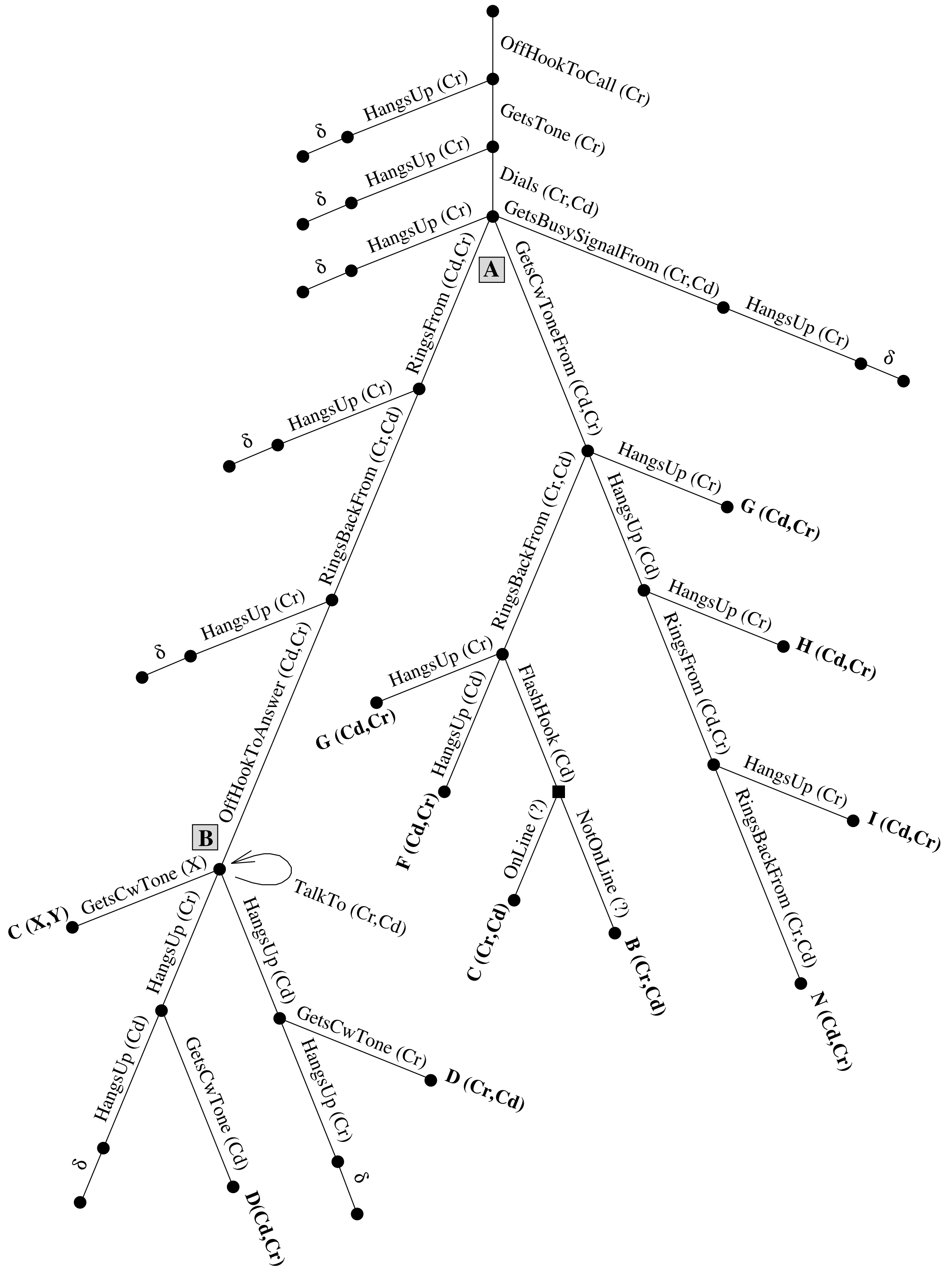


Fig 19. Formal specification of *Connection*. Part 1.

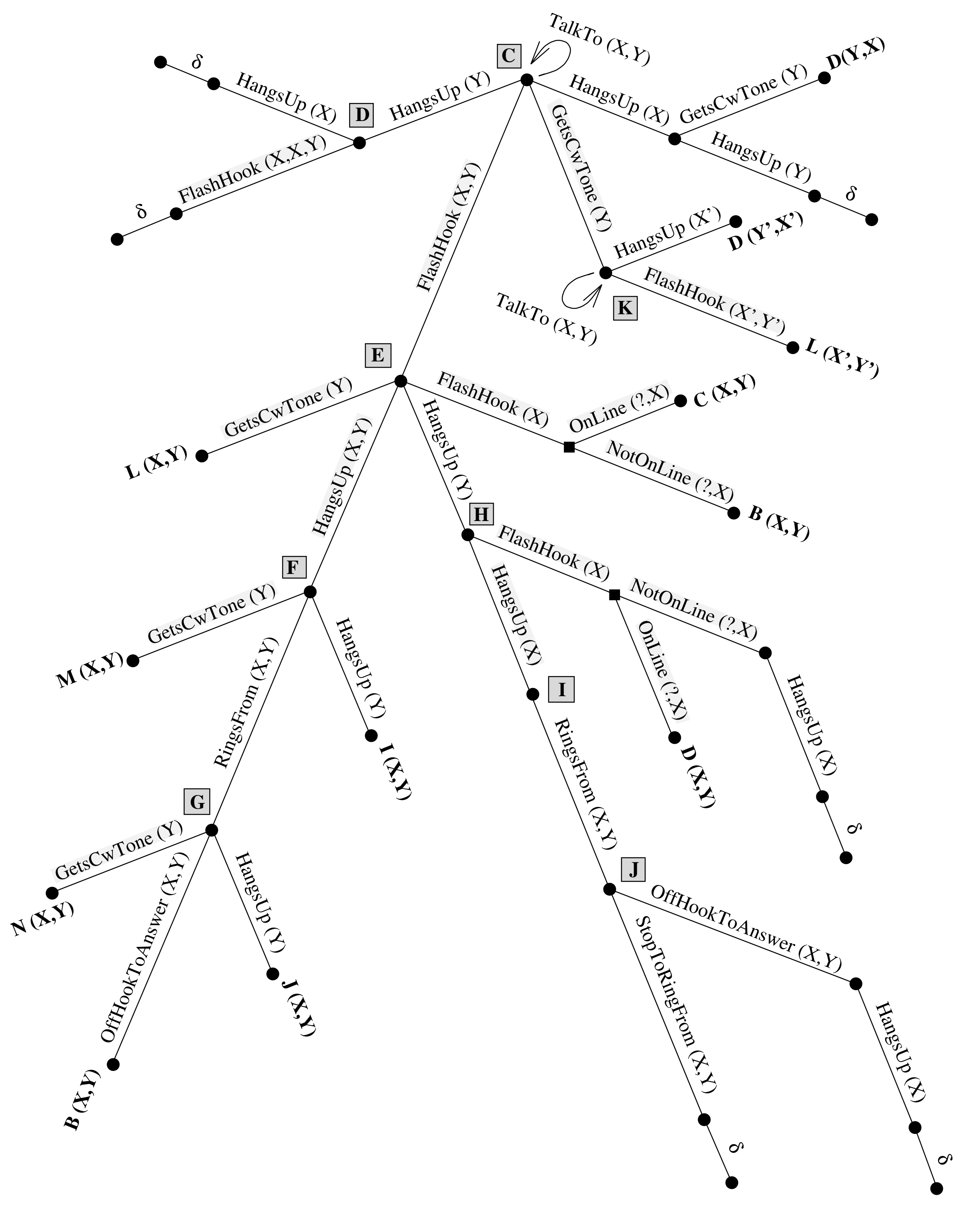


Fig 20. Formal specification of *Connection*. Part 2.

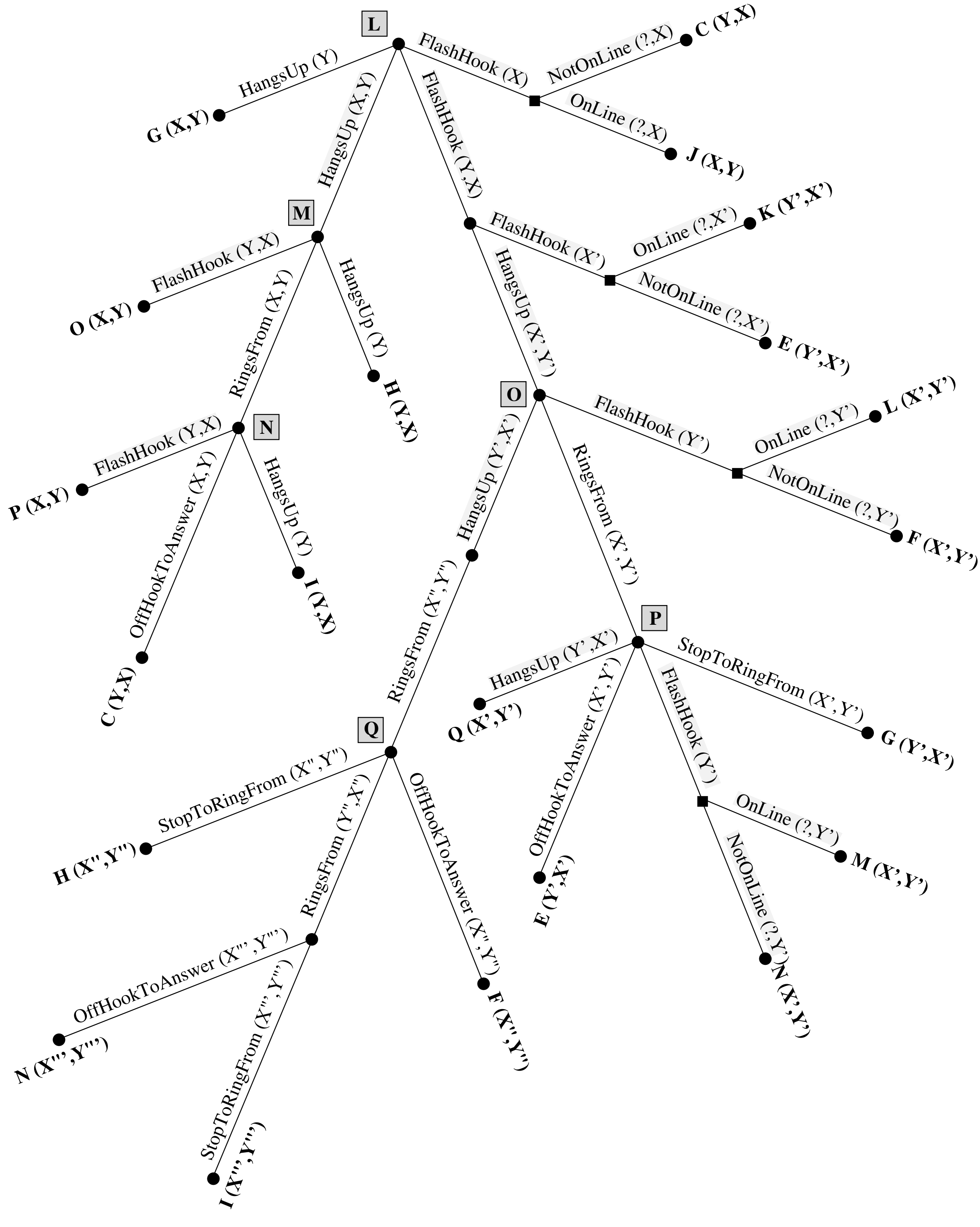


Fig 21. Formal specification of *Connection*. Part 3.

- State B maps the case where the caller and the called are talking together in a POTS communication. If both are subscribers of the Call Waiting feature, their CW features are free and they may receive a CW signal (action *GetsCwToneFrom(X,Y)*). The first one who receives a CW tone becomes user X while the second one becomes user Y.
- In state C user X has received a CW tone while he is speaking with Y. So his CW feature is now busy. If Y hangs up, process *Connection* moves into state D where X may flash the hook to answer the CW signal or X may hang up and he will be automatically recalled by the Telephone System to answer the CW signal.
- State E maps the case where X has put Y in holding state to speak with a third user. So Y is on hold, but he may receive a CW tone if he is subscriber of the CW feature.
- In state F, user X has forgotten Y in holding state and he has hung up instead of flashing the switchhook. So he is recalled automatically and in state G X is ringing from Y while Y is still on hold.
- State H maps the case where user Y has hung up while he was in holding state. If X flashes the hook now, he can't bring back Y in conversation.
- If X forgets Y on hold, process *Connection* moves into the states I and J where X is recalled automatically while Y is not on line any more.
- In state K, the users X and Y are talking together while both have received a CW tone. The first one who flashes the hook to answer his incoming call becomes X' and he puts Y' on hold.
- In state L, X is answering his incoming call while Y is on hold. If X hangs up, he is recalled automatically while Y may answer his CW signal (states M and N).
- If Y flashes the hook, both users have put each other on hold. If one of them hangs up, process Connection moves into state O where he is recalled automatically.(state P)
- If both users hang up while they have put each other on hold, we move into state Q where both users must be recalled. They may talk together again if both pick up to answer the automatic recall at the same time.

• When user A is talking alternatively to user B and to user C, the two instantiations of process *Connection*, the first one parametrized by (A,B), the second one by (C,A) execute the same sequences of actions in the tree but their current positions are shifted and both synchronize consecutively with the process *CwController*. The example of Fig 22. illustrates the synchronization between these processes.

In this example, user C calls user A who is subscriber of the Call Waiting feature and who is already talking to user B. In Fig 22. the time is represented by a vertical descendent axis, the first column shows the sequence of actions executed by the first instantiation of process *Connection*, parametrized by (A,B). The second column shows the sequence of actions executed by the instantiation number 2 parametrized by (C,A). Both instantiations synchronize

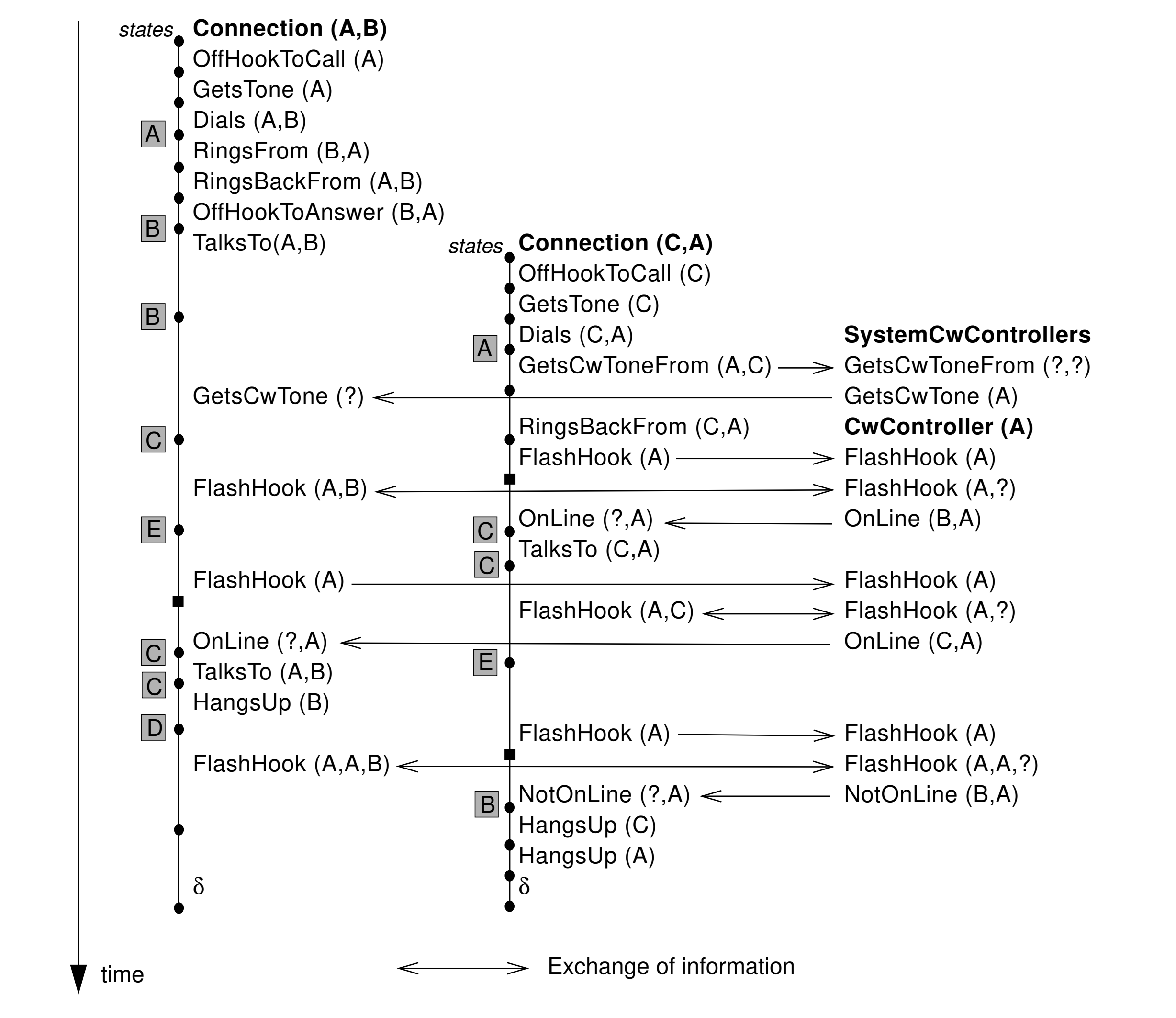


Fig 22. Execution's example of CW specification.

with the third column which shows the actions executed by the process *SystemCwControllers* and then by the instantiation of *CwController* parametrized by (A). A synchronization between two of these processes is materialized by an arrowed line which shows the sense of information's exchange.

The execution starts with a basic sequence of events where user A calls user B (instantiation number 1 of *Connection*). Once they are in talking state B, user C picks up to call user A, what maps the creation of the instantiation number 2 of process *Connection*. After the dialing event, the only event allowed by the process *GlobalView* (see § IV.2.5) is the action *GetsCwToneFrom(A,C)* which synchronizes with *Connection(C,A)* and *SystemCwControllers*. This last process forces then *Connection(A,B)* to synchronize on the action *GetsCwTone(?)* with ?=A and so to move into state C . *Connection(C,A)* may then execute

RingsBackFrom(C,A) and A may flash the hook to answer C. This action *FlashHook(A)* is executed by *CwController(A)* which obliges just after *Connection(A,B)* to execute *FlashHook (A,B)*. But during this action, *Connection(A,B)* communicates *CwController(A)* the fact that user B is still on line when A flashes the hook. In the opposite case, *Connection(A,B)* would have proposed the action *FlashHook(A,A,B)* for synchronization, what means that B would have already hung up. After the action *FlashHook(A,B)*, *CwController(A)* communicates *Connection(C,A)* through the event *OnLine(?,X)* with ?=B and X=A the fact that the third user B has been put in holding state by user A.
When user A talks to user C, *Connection(A,C)* is in the talking state E while *Connection (A,B)* is in the holding state C . If user A flashes the hook once again with user C on line, we have the same sequence of events as seen just above, and *Connection(A,B)* moves into the talking state C while *Connection(C,A)* moves into the holding state E .So both instantiations move on the same tree of actions but their current positions are shifted.

- If user B hangs up when he is talking to user A, user A may flash the hook to bring user C back in conversation. But this time, *Connection(A,B)* communicates *CwController(A)* the fact that B has hung up by synchronizing on the event *FlashHook (A,A,B)*. So C*wController (A)* obliges *Connection(C,A)* to execute *NotOnLine(?,X)* with ?=B and X=A, and so to move into state B instead of state C. (the Call Waiting feature of user A is free in state B while it is busy in state C)

This example has shown the synchronization between two interleaved connections via a CwController, and the exchange of information during the synchronization phases. The formal description of process *CwController* is shown in Fig 23.

IV.2.5 Process GlobalView

For describing the state of the Telephone System, process *GlobalView* updates the lists of the POTS specification (*BusyList* and *WaitPairs*) completed by three new lists related to the Call Waiting feature functioning:

- *CwList*, of sort DecSet, is the list of subscribers of the CW feature. So user X may receive a Call Waiting tone only if he is present in this list which is defined in the specification and which can not be modified during the execution.
- *HoldPairs*, of sort SetOfPairs, is the list of pairs (X,Y) where X has put Y in holding state to speak with a third user T.
- *CwPairs*, of sort SetOfPairs, is the list of pairs (T,X) where user T calls user X who is subscriber of the CW feature and who is already in conversation with user Y. The pair (T,X) in *HoldPairs* maps the fact that the CW feature of X is busy.

These lists are updated as follows:

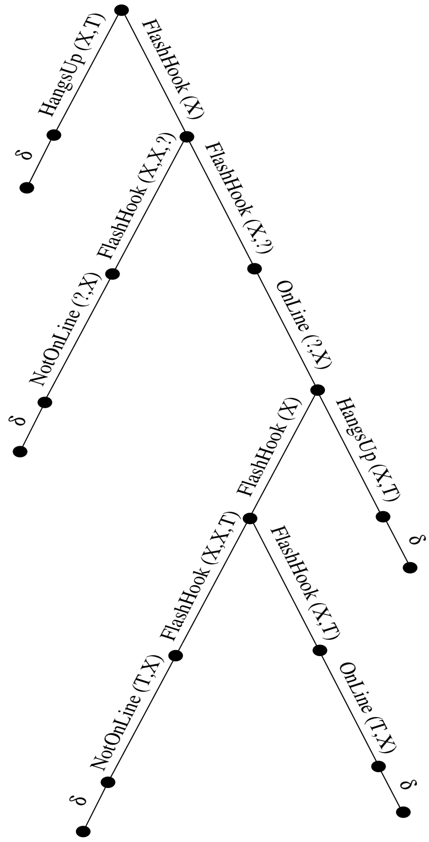


Fig 23. Formal specification of process *CwController*.

• X is added to *BusyList* when he picks up the handset. He is removed from *BusyList* when he hangs up.

• As in the former specifications, the pair (X,Y) is added to *WaitPairs* when X is ringing from Y and it is removed when X picks up to answer. But this time, the list *WaitPairs* is used to differentiate two new user states: when X picks up to call and he is subscriber of the CW feature, he can't receive a CW tone before speaking with an other user Y; we must differentiate the two states: before and while speaking with an other user. So when X picks up to call, we add the pair (X,X) to *WaitPairs*, which is replaced by the pair (Y,X) when Y is ringing from X. The pair (Y,X) is removed only when Y picks up to answer X. So now X can receive a CW signal only if he is not the second element of a pair in *WaitPairs*. The list *WaitPairs* is now used to differentiate four sates:

- when X is off hook, he may be calling Y or he may be speaking with Y.
- when X is on hook, he may be ringing or not.

• The pair (X,Y) is added to HoldPairs when Y is on line and X flashes the hook to put him on hold (action *FlashHook(X,Y)*) or when X gets a CW tone from Y (action *GetsCwTone-*

From(X,Y)). *HoldPairs* is tested when a user hangs up: if this user is the first element of a pair in *HoldPairs*, he has forgotten somebody in holding state, and he must be recalled automatically. The pair (X,Y) is removed from *HoldPairs* only if X flashes the hook to bring Y back in conversation or when X picks up to answer the automatic recall from Y. If X does not answer this recall, the Telephone System may decide to stop the ringing signal (action *StopsToRingFrom(X,Y)*) and the pair (X,Y) is removed from *HoldPairs*.

• The pair ((X,Y) is added to *CwPairs* when X receives a CW signal from Y (action *GetsCwToneFrom(X,Y)*) what means Y grasps the Call Waiting feature of X. The pair (X,Y) is removed from *CwPairs* only if X hangs up while he is speaking with Y or if X flashes the hook to speak with a third user T while Y has already hung up. If X forgets Y on hold and hangs up, the pair (X,Y) is removed only when X picks up to answer the automatic recall or when the Telephone System decides to stop the ringing signal.

These lists are used to make the guards on the following actions:

- X can pick up to call (action *OffHookToCall(X)*) only if:
  - X is not in *BusyList* (if not, he would be already off hook)
  - there is no pair (X,Y) in *WaitPairs* (if not, X would be ringing from Y)
  - there is no pair (X,Y) in *HoldPairs* (if not, X would have forgotten Y in holding state and he could not pick up the handset to call before the automatic recall)

After the dialing event *Dials(Y,X)* where Y tries to call X, there is three possibilities:

- X receives a ringing signal (action *RingsFrom(X,Y)*) only if:
  - X is not off hook (X is not in *BusyList*)
  - X is not ringing from an other user (X is not already the first element of a pair in *WaitPairs*)
  - X has forgotten nobody on hold or X has forgotten Y and he is automatically recalled (so if X is the first element of a pair in HoldPairs, the second element must be Y)
- X receives a Call Waiting signal (action *GetsCwToneFrom(X,Y)*) only if:
  - X is subscriber of the Call Waiting feature (X is in *CwList*)
  - Y does not equal X what means X can not call himself via the Call Waiting feature
  - X is off hook (X is in *BusyList*) and X is speaking with a third user (X is not the second element of a pair in *WaitPairs*)
  - the Call Waiting feature of X is free (X is not the first element of a pair in *CwPairs*)

• If X does not respect the conditions to receive a ringing signal or a Call Waiting tone, Y gets a busy signal from X (action *GetsBusySignalFrom(Y,X)*).

IV.3 Call Forward on Busy Line (Cfbl)

The Call Forward on Busy Line feature allows a call incoming to a telephone number to be automatically transferred to an other predefined number if this called number is already busy. The specification of this feature presents the same sequences of actions as the POTS specification seen in Fig 9. The main differences are about process *GlobalView*:

• The list *CfblPairs*, of type SetOfPairs, is added to the former lists *BusyList and WaitPairs*. The pair (X,Y) is added to *CfblPairs* only if X is subscriber of the Cfbl feature and if Y is the predefined number where the calls must be transferred if X is busy. This list is static so it can not be modified during the execution.

• After the dialing event *Dials(Caller,Called),* process *GlobalView* communicates process *Connection* via a hidden event the number X associated to the called number in *CfblPairs* if the pair (Called,X) is in this list. If not, X equals None. So process *Connection* proposes four possible actions for synchronizing with *GlobalView*:

- *XInsteadOfYRingsFromZ(Called,Called,Caller)*
- *XInsteadOfYRingsFromZ(X,Called,Caller)* if X≠None
- *XGetsBusySignalFromYInsteadOfZ(Caller,Called,Called)*
- *XGetsBusySignalFromYInsteadOfZ(Caller,X,Called)* if X≠None

The choice between these four events is made in process *GlobalView*:

• The action *XInsteadOfYRingsFromZ(X,Y,Z)* is possible only if:

```
        (X is free)
and     (X is not ringing)
and     (       (X=Y)
            or  (           (X= second element of Y in CfblPairs)
                    and     (       (Y is busy)
                                or  (Y is ringing)
        )       )           )
```

• The action *XGetsBusySignalFromYInsteadOfZ(X,Y,Z)* is possible only if:

```
        (       (Y is busy)
            or  (Y is ringing)
        )
and     (       (           (Y=Z)
                    and     (Y is not subscriber of the Cfbl feature)
                )
            or  (           (Y= second element of Z in CfblPairs)
                    and     (       (Z is busy)
                                or  (Z is ringing)
        )       )           )
```

IV.4 Call Forward Always (Cfa)

The Call Forward Always feature allows a user to transfer automatically all his calls to an other predefined telephone number. As in the Cfbl feature, process *Globalview* uses a list *CfaPairs*, of sort SetOfPairs. The pair (X,Y) is added to this list only if X is subscriber of the Cfa feature and if Y is the telephone number where every call must be transferred. After the dialing event *Dials(Caller,Called)*, process *GlobalView* communicates process *Connection* through a hidden event the number X associated to the Called number in *CfaPairs* if the pair (Called,X) is in this list. If not, X equals None.

- If X=None, process *Connection* proposes the two actions *RingsFrom(Called,Caller)* or *GetsBusySignalFrom(Caller,Called)*.

- If X≠None, process *Connection* proposes the two actions *RingsFrom(X,Caller)* or *GetsBusySignalFrom(Caller,X)*.

Process *GlobalView* synchronizes on one of both actions proposed by process *Connection*.

IV.5 Automatic Recall (Arc) and automatic Call Back (Acb)

If the pair (Y,X) is subscriber of the Automatic Recall feature, Y will be automatically recalled by X if he is busy when X tries to call him.

- The formal description of process *Connection* shown in Fig 24. is identical to the POTS specification shown in Fig 9. except for the busy signal treatment. Every gray tinted action in the tree is an internal action executed by the Telephone System alone (not visible from a user point of view).

- When user X receives a busy signal from Y, he must hang up and process *Connection* moves into state $\boxed{A}$. If the pair (X,Y) is not subscriber of the Arc feature, process *Connection* synchronizes with process *GlobalView* on the hidden event *HangsUp(X)* and its execution stops. In the opposite case, both processes execute the hidden action *Redials(X,Y)* what maps a new try to join Y. If Y is still busy, the Telephone System executes the internal event *GetsBusySignalFrom(X,Y)* and process *Connection* returns to state $\boxed{A}$ to try again. If Y is free, the Telephone System executes the hidden action *RingsFrom(Y,X)* followed by the external action *RingsFrom(Y,X)*.
If X is free, the Telephone System continues with *RingsFrom(X,Y)* and it waits the users to pick up to answer. On the other hand, it may decide to stop the ringing signals after a predefined time.
If X is busy, the Telephone System can not execute *RingsFrom(X,Y)*, and it may stop the ringing signal of user Y or Y may pick up to answer and have nobody on line.

Arc process Connection MISSING

Fig 24. Formal specification of process *Connection* in the Arc feature.

• Process *GlobalView* updates the former lists *BusyList* and *WaitPairs*, and the two new lists related to the Arc feature:

- *ArcPairs*, of sort SetOfPairs, is the list of pairs (X,Y) where X must be recalled automatically by Y if he is busy when Y tries to call him.
- *RcPairs*, of sort SetOfPairs, is the list of pairs (X,Y) where X will be recalled automatically by Y when he will be free.

ArcPairs is a static list while *RcPairs* is updated as follows:

- When Y gets a busy signal from X (action *GetsBusySignalFrom(Y,X)*) and if the pair (X,Y) is in *ArcPairs*, the pair (X, second element of X in RcPairs) is removed from *RcPairs* and (X,Y) is added.
- When the internal action *RingsFrom(X,Y)* is executed (when X can be recalled by the Telephone System), the pair (X,Y) is removed from *RcPairs*.

Automatic Call Back is a feature which works like Automatic Recall but the caller user is here the subscriber of the feature instead of the called user.

IV.6 Originating Call Screening (Ocs) and Terminating Call Screening (Tcs)

Originating Call Screening is a feature which prevents a subscriber from calling an other user Y. So the list *OcsPairs*, of sort SetOfPairs, is the list of pairs (X,Y) where X can not call Y.
The formal description shown in Fig 25. is identical to the POTS specification of Fig 9. except after the dialing event *Dials(X,Y)*:

- If (X,Y) is in *OcsPairs*, the action *GetsRefusedCallSignalFrom(X,Y)* is executed.
- If Y is busy or already ringing from an other user Z, X receives a busy signal.
- If (X,Y) is not in *OcsPairs*, Y is not busy and Y is not ringing, the action *Rings-From(Y,X)* is executed.

Terminating Call Screening is a feature identical to Originating Call Screening except that it prevents a subscriber X to be called by a user Y.

IV.7 Calling Number Delivery (Cnd) and Unlisted Number (Un)

Calling Number Delivery is a feature which allows a subscriber X to read on his telephone screen the number of user Y who is calling him. The list *CndList*, of type DecSet, is the list of the subscribers of the Calling Number Delivery feature. So when Y tries to call X who is in *CndList*, the action *XRingsFromYDisplayed(X,Y)* is executed. If X is not in *CndList*, the action *XRingsFromYNotDisplayed(X,Y)* is executed.

OCS process MISSING

Fig 25. Formal specification of process *Connection* in the Ocs feature.

The feature Unlisted Number allows the subscriber X not to be displayed on the screen of user Y when he tries to call him. The list *UnList*, of type Decset, is the list of subscribers of the Unlisted Number feature. So when Y is calling X, the action *XRingsFromYBlocked(X,Y)* is executed if Y is in *UnList*. If Y is not in *UnList*, the action *XRingsFromYNotBlocked(X,Y)* is executed.

IV.8 Different Ringing

Different Ringing is a feature which allows a telephone set to have several telephone numbers and to associate one ringing signal to each number. So if X has two numbers X1 and X2, the pairs (X,X1) and (X,X2) are added to the list *DrPairs*, of type SetOfPairs.
When user Y tries to call X2, the action *XRingsfromYWithRingNumberZ(X,Y,X2)* is executed. If Y tries to call Z who is not subscriber of the Dr feature, the action *XRingsFromY(Y,Z)* is executed. Otherwise, the specification of this feature presents the same sequence of actions as the POTS specification seen in Fig 9.

V Conclusion

We have presented in § III and § IV the specification of 12 Telephone Features. Because all of them follow the general rules about the modelisation of a Telephone System (the structure of the high levels is generic and common for all the specifications), because the structure of actions is unique and always respected and because we have defined a general data structure which is usable in all the specifications and which allows to update the system in a clear and concise way, we are now able to integrate these features two by two for detecting their interactions. This step has been realized but I don't describe it in my report because I consider that its interest is insufficient.
However, a next interesting work could be the specification of a complete Telephone System by integrating all the specifications developed during this internship, and in which we could integrate any new Telephone Feature.

Bibliography

- *Detecting Feature Interaction in Telecommunication Systems.*
  Thesis presented to the University of Ottawa, Mohamed Faci, 1995.
- *Formal Description of Telephone System in LOTOS: the Constraint oriented Style Approach.* M. Faci, L. Logrippo, B. Stepien.
- *The Formal Description Technique LOTOS.*
  P.H.J. van Eijk, C.A. Vissers, M. Diaz, 1989.