Structural Models for Specifying Telephone Systems

Mohammed Faci, Luigi Logrippo, and Bernard Stepien

University of Ottawa
Telecommunications Software Engineeering Research Group
Department of Computer Science
Ottawa, Ontario, Canada K1N 6N5
E-mail: (mfaci | luigi | bernard)@csi.uottawa.ca

Abstract. Two approaches, *resource-oriented* and *constraint-oriented*, for structuring telephone systems specifications, are presented. Both approaches express behaviour by collections of communicating processes, using the language LOTOS. However, requirements are distributed differently among processes. Examples are taken from specifications of telephone systems, first basic, and then with features. The features used as examples are *call forwarding*, *originating call screening*, and *three-way calling*. The two structuring methods are compared.

Keywords: Telephone systems, telephone features, formal specification, specification styles, design, architecture, structure, LOTOS.

1. Introduction

The structure of a specification of a distributed system should reflect the specification's purpose. If the purpose is requirement capture, the processes in the specification may represent the *constraints* of the system. If the purpose is systems design, we may wish to emphasize the *abstract* architectural components of the design. If the purpose is verification, the structure must suit the preferences of the verification methods and tools. If the purpose is implementation, the specification must exhibit a structure that corresponds to an efficient implementation, and must reflect the architectural components in the implementation. For test case generation, the specification should facilitate the extraction of relevant test cases with the available tools. It is common to end up with several specifications during the study of a given system.

Vissers et al [VSVB91] wrote a fundamental paper on the subject of *specification styles* in the LOTOS language [BB87][LFH92]. In that paper, it was shown that LOTOS can adapt itself to different expressive needs. Four basic styles were identified: constraint-oriented, resource-oriented, state-oriented and monolithic. Subsequent papers have addressed the question of transformations between styles [BFLL95][FMN95]. Over the years, our group has investigated the matter of specification styles by using as examples various types of telephony systems. We found application not only for the four basic styles[FaLS90] [FaLS91] [BL93] [FL94], but also for others, which possibly could be considered as derived ones, such as the *status-oriented* style [SL93].

From a user's point of view, a telephone system's architecture is a simple one. It has two components: the user (which, if one prefers, is the handset), and the switching system, which represents everything else. In some respect, the switching system's architecture is of no concern to the user, as long as it provides the required services according to expectations. From a designer's point of view, however, the switching system's architecture is a central issue.

In this paper, we illustrate two different structural approaches for specifying a telephone system: the *resource-oriented* style and the *constraint-oriented* style. In the resource-oriented style, the specification structure shows the architectural components of the design. By contrast, in the constraint-oriented style, one focuses on the composition of the requirements, expressed as behaviours. In this sense, one can say that the resource-oriented style is implementation-oriented and the constraint-oriented one is requirement-oriented.

The paper assumes that the reader has already been exposed to LOTOS or has familiarity with concepts such as process communication, synchronization, and composition [Hoar85] [Miln89]. The LOTOS specifications presented are partial, e.g. we may omit parameters, not consider all cases, etc., in order to keep the attention on the general structure. Similarly, we did not use G-LOTOS [BNT94] in our graphical representation in order to avoid the clutter of the details. We did, however, take inspiration from G-LOTOS.

2. The Resource-Oriented Specification Style

2.1 POTS: Plain Old Telephone Systems

In the resource-oriented style, the specification structure follows the architecture of the physical components of the system and it has an object-oriented flavor. Processes are similar to *class templates* in object-oriented terminology [CRS90][Rud92]. The system is represented by two entities, the *user* process and the *switching* system, which we will take to have a *client* role and a *server* role, respectively. Note, however, that both the user process and the switching system are part of the *switch's* software, which implies that our reference to *user* is a reference to a process inside the switch which acts on behalf of the user. So, the client entity, from which a set of users can be instantiated, expresses the observable behaviour of the system, as seen from the user. The server, of which a single instance exists for each switch, represents the behaviour of the switching system. In LOTOS, we represent this high-level architecture by a parallel composition between instances of phone users and a controller process, that handles the end-to-end aspects of connection between users. The users themselves are mutually independent, thus are described as processes in interleave with respect to each other. The following specification fragment shows how to specify *m* user processes and a switching system (note that an unbounded number of phones can be specified by using recursive interleave, this will be shown later). A *user process*, expressed by a LOTOS process *Phone*, communicates with the *switching* through gate *n* and with the user (outside world) through gate *u*.

```
(   Phone [u, n] (1)
    |||
    ...
    |||
    Phone [u, n] (m)
)
|[n]|
Switching [n]
```

In order to communicate, the users must place requests to the server. The server needs to respond concurrently to requests from many different users. Every time the server receives a request, it needs to create a new instance of a connection handler. A connection handler template is defined as a LOTOS process *Connection_Handler* which can handle one connection at a time. The template is instantiated with data specific to the user who will be part of the connection. The *Connection_Handler* instantiation can be achieved also with a recursive interleave mechanism, using a predefined action as a trigger. In this specification, the *switching* process uses *dialtone* as a trigger to create a *Connection_Handler* process. The execution of the latter is interleaved with the one of process *Switching*, and possibly with other instances of *Connection_Handler*.

```
process Switching [n]: noexit :=
    n   ?Caller: Digits     !dialtone;
    (   Connection_Handler[n](Caller)
        |||
        Switching [n]
    )
endproc
```

Connection_Handler makes a request for a connection *(conreq)*, then establishes a connection between *Caller* and *Called*.

```
process Connection_Handler [n] (Caller: Digits) : noexit :=
    n   !Caller     !conreq     ?Called: Digits;
    Establish_Connection[n](Caller, Called)
endproc
```

Note the structure of the atomic action in the first line of process *Switching*. This means that this process is ready to synchronize on gate *n* (with process *Phone)*. It is ready to do so (**!**) on a constant which represents the *dialtone* signal, and at the same time, it expects to receive (**?**) the number of the *Caller*, which is of sort *Digits*. Similarly, in the first line of process *Connection_Handler,* the process is ready to synchronize on a signal *conreq* and only on a specific (i.e. previously received) *Caller* number while expecting to receive the number of the *Called*. Note that constants are in lower cases, variables have upper case initials. Clear specifications require carefully planned action structures, which should be as uniform as possible throughout.

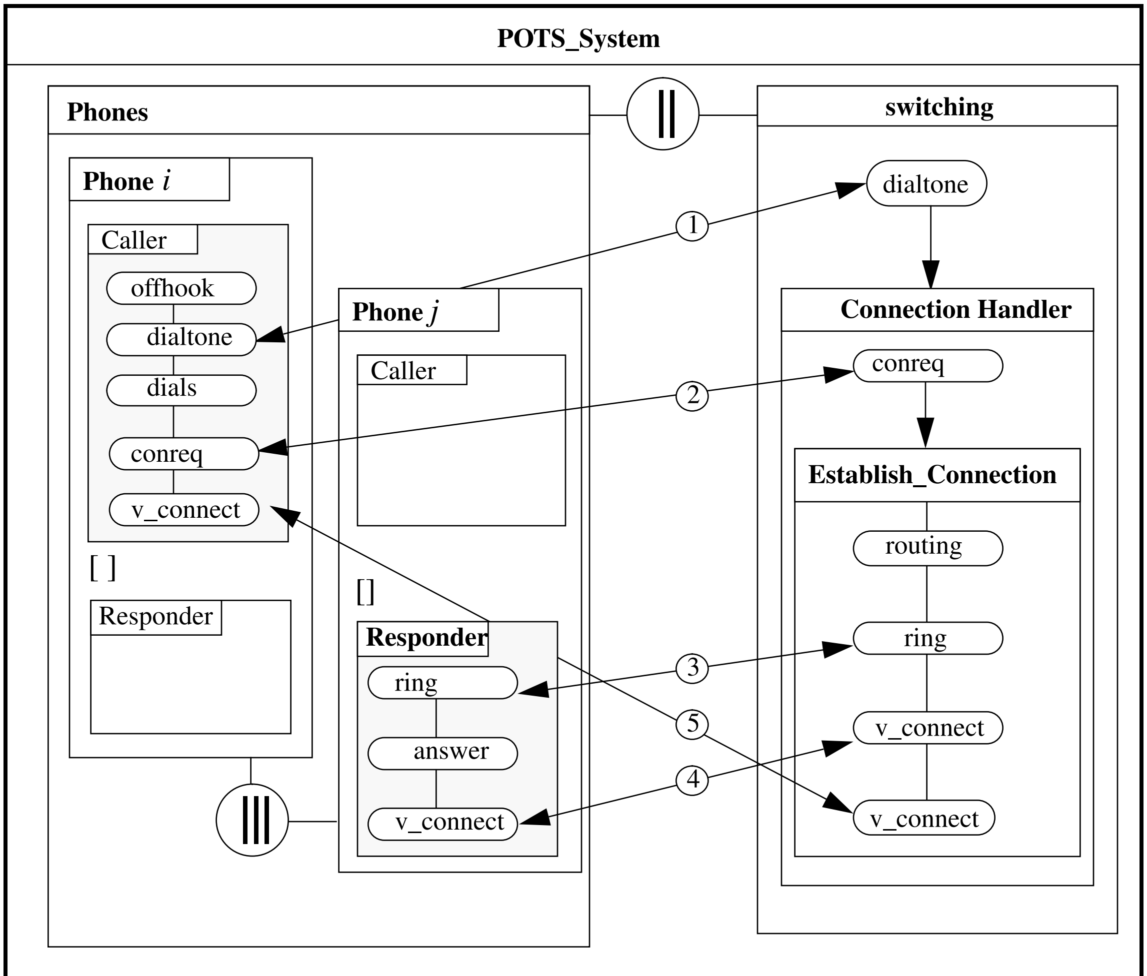

Fig. 1. Establishing a connection between two users: user *i* calls user *j*.

In the simplest model, the behaviour of a connection establisher consists in ringing the called party and if this party answers the call, starting the voice exchange phase between the two parties. After that, the server waits for a connection release event. This functionality can be represented by the sequence of actions shown in Figure 1.

From an idle state, a caller (using phone *i)* interacts with the *i* user process by lifting the handset, which we express as *offhook* action. Next, user process *i* synchronizes with the *switching* on the *dialtone* action ①. Since *dialtone* is used as detection action, an instance of *Connection_Handler* is created. Once the *user* process collects the digits of the called party through the *dials* action, the *Connection_Handler* synchronizes with the user process, on the action *conreq*②, to request the selection of a route for voice transmission. Next, if the called party (phone *j)* is idle, the *switching* process *rings* ③ phone *j*, then if the called party answers, the voice exchange phase between the two parties begins as indicated by the *v_connect* (for voice connected) actions ④ and ⑤.

The following behaviour expression shows the state of a system where user 1 is attempting to call user 3 and user 2 is attempting to call user 4:

```
(   Phone [u, n] (1)
    |||
    Phone [u, n] (2)
    |||
    Phone [u, n] (3)
    |||
    Phone [u, n] (4)
)
|[n]|
(   Establish_Connection[n](1,3)
    |||
    Establish_Connection[n](2,4)
    |||
    Switching [n]
)
```

Each of the four users is allowed to execute its specified behaviour independently of the other three users. However, the ordering of actions between two users (e.g., 1 and 3) is determined by their corresponding processes for establishing the connection (e.g., Establish_Connection[n] (1, 3)).

The mechanism for specifying busy state is explained in Section 2.6.2. The busy phone goes into a process that continually offers the action *busy* and is unable to synchronize on actions implying idle state, such as *ring*. We refer to [SL93] for a detailed discussion of this point.

Note that what we show in this paper is still a very idealized picture of the system architecture. In real systems, components will be distributed, channels will be interposed, and there won't be the direct communication between components implied by the |[n]| operator. The specification of channels, as well as of other implementation details, is the object of further refinements [BFLL95] [FMN95] and won't be given in this paper.

2.2 Specification of Features

A feature is intended to be available to users, consequently we can expect that some new sequences of operations will be available in the client specification, which needs to match corresponding sequences on the server side. Thus, a feature cannot always be represented by a single black box that can be inserted at some specific location in the specification of a phone system. Rather, it is a combination of distributed behaviours and resources that are combined to realize the desired behaviour. Therefore, we need to characterize the principles governing such distribution, to avoid scattering and lack of structure, identifying:

- Aspects dealing with basic types of functionalities;
- Aspects dealing with the activation of features; and
- Aspects dealing with implementation issues.

First, from the point of view of functionality, there are two basic kinds of features: Features that extend the system with a new type of service, which we call *technological features* and features that restrict some behaviour of the existing system or of some other existing services, which we term *policy features*.

The second aspect deals with the activation of features: features can be passive or active. A passive feature will get activated by the system in a background mode, without any action from the user. *Call answer* or *voice mail* are passive because once subscribed to, they get activated by events that are not under the subscriber's control. An active feature is a feature that needs some action of the user in order to get activated. For example, the *three-way calling* feature will be activated only if the subscriber is part of an already established connection and decides to add another party to a conversation. Mixed features combine the two characteristics. For example, the *call waiting* feature is of a mixed kind because it responds passively to incoming calls while the subscriber is engaged in another call, but the subscriber decides if she wants to answer by performing a flash-hook action.

Finally, implementation issues deal with the efficient mapping of feature behaviours onto physical system resources. We have already mentioned that the refinements required to reach a real implementation architecture are beyond the scope of this paper.

2.3 Specifying Features in LOTOS

To specify features in LOTOS, we must find appropriate LOTOS constructs that express the intended behaviour and, at the same time, allow the structuring of the specification for clarity and future modifications. First, we present atomic LOTOS concepts appropriate for feature specification, and then we discuss some stylistic considerations on how to piece these atomic concepts together to obtain a readable specification.

A feature will usually be specified as an alternative or optional sequence of operations to some other behaviour. In LOTOS there are three operators that can be used for such a purpose, the interleave, the non-deterministic choice, and the disable operator.

- The interleave operator ||| is used when a feature is available concurrently with some other behaviour either to a user or to the switch. This allows both the original behaviour and the added feature to co-exist, with no side effects.

- The non-deterministic choice operator [] is used whenever a feature is a potential replacement of another behaviour.

- The disable operator [> is used whenever a feature is intended to take over from another established behaviour.

LOTOS guards ([E]->), which can be used in combination with any behaviour expression, are well suited to portray the restriction functionality of policy features. In this case the features are composed of both a behaviour and some control data. Guard constructs can also be used to turn a feature's behaviour expression *on* or *off* to reflect the fact that a user did or did not subscribe. The following two examples summarize the above described design principles for technological and policy features.

A user's behaviour having subscribed to the *call waiting* and *three way calling* technological features may be represented using interleave constructs, because these features are available concurrently to the user while in the *talk* state:

```
u   !Caller   !offhook;
n   !Caller   !dialtone;
n   !Caller   !dials            ?Called: digits;
...
(   Talk [u, n] (Caller, Called)
```

```
    |||
    Call_Waiting [u, n](Caller)
    |||
    Three_Way_Calling [u, n](Caller)
)
```

The processes *Call_Waiting* and *Three_Way_Calling* are defined as recursive processes, with the necessary logic to ensure that only one instance of the feature is active at a time, and that each feature can be activated more than once during the lifetime of the same call. For example, if three way calling is activated by a user **A**, while talking to **B**, to form a conference call between **A**, **B**, and **C**, then user **A** cannot activate a second instance of this feature until the current instance is de-activated. However, once de-activated, user **A** may form another conference call between **A**, **B**, and **D**.

Our next example illustrates the specification of a policy feature such as *Originate Call Screening* (OCS). It portrays the fact that a switch has two operational ways to process a connection request depending on the data presented by the *OCS* feature that is passed as parameters in the *conreq* action. In this case the choice operator [] is used to indicate that the two alternatives are mutually exclusive and that the guard expressions will determine which branch is chosen.

```
n   !Caller       !conreq    !Called     ?Scr_List: Screen_List;
(
    [Called NotIn Scr_List] ->
                n   !Called       !ring;
                ...
    []
    [Called IsIn Scr_List] ->
                n   !Caller       !Called       !refuse_req;
                ...
)
```

2.4 The Resource-oriented Control Mechanism

Control is achieved in LOTOS via two basic mechanisms, synchronization on actions and guards that evaluate abstract data types.

In the resource oriented style, interactions occur between two behaviour expressions, one that allows interleaved behaviours, the other that allows nondeterministic choices that are resolved by the interleaved actions of the first behaviour as it goes through its various states.

Let us illustrate this concept with an example. Figure 2 shows a user process (phone) which supports a feature. The behaviour of the feature is expressed by the sequence: *B1 := (f; g*; ...). The behaviour of basic call is expressed by the sequence: *B2 := (a; b; c*; ...). However, the requirements of this feature dictate that it can be activated only after basic call has executed its first action *a*. The resulting specification of the phone is then expressed as: *Phone := a; (B3* ||| *B1)*, where *B3 := (b; c; ...)*. A given phone may at any time be subject of requests (feature activation) from an unlimited number of other phones via the network and more precisely connection handler instances. In figure 2 we have represented two such instances of connection handlers performing requests on a single instance of a phone. The basic call sequence of the phone starts by synchronizing with the first connection handler that offers action *a*, as shown by ①. After this, this connection handler and the phone can continue the basic call sequence ②, however at any point another connection handler can start a feature sequence, which can continue in parallel with the basic call sequence ③. Many features can be specified using these principles. Guards involving simple abstract data types can be used to turn features *on* or *off* depending on the subscriber's choices, as well as to manage the invocation of features behaviours.

This approach applies to policy features as well. The main difference is that policy features will include appropriate guards to enforce the policy. There will be a side that is deterministic, driven by guards, and another side that is not. The synchronization mechanism between the two sides resolves the non-determinism and enforces the policy globally.

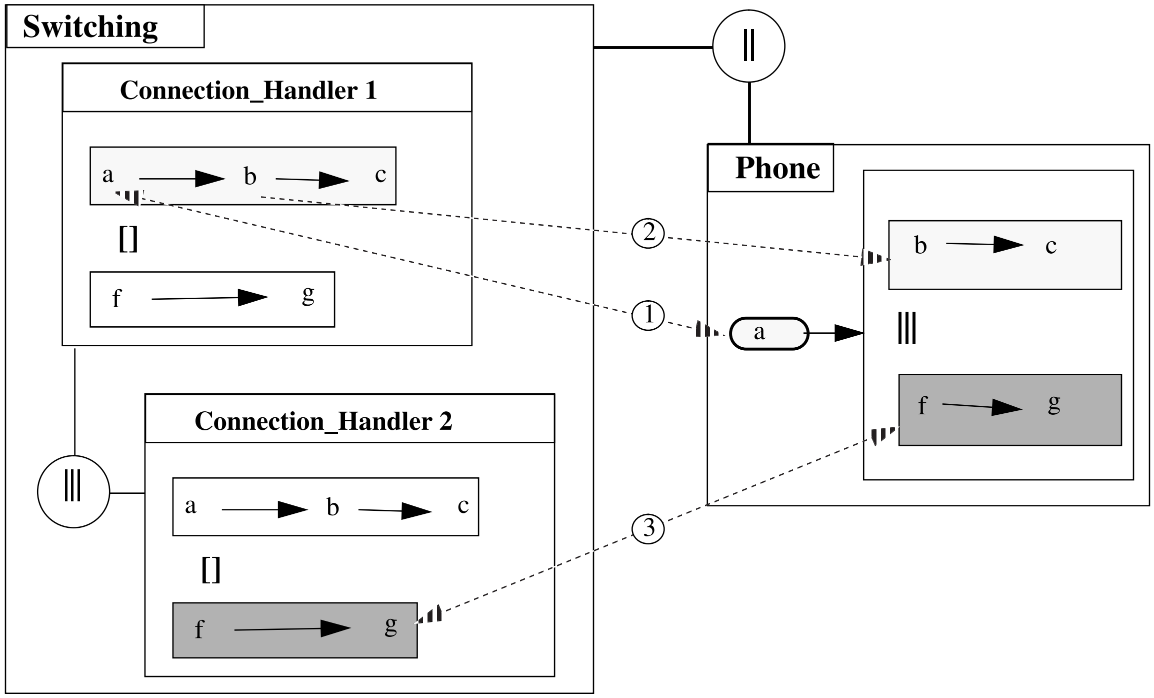

Fig. 2. Synchronization paths in the resource oriented style

The specification has to be decomposed into processes that represent individual components of the system and then these components need to be decomposed further by classes of reusable functionalities. The design of the user behaviour specification follows a different pattern from the design of a switching behaviour. This is due to the fact that a switching mechanism is basically a server (or more precisely an agent) that receives requests from many clients that are the users. We will look at the separate architectures of these two main components.

2.5 Structural Considerations for the Client

In the basic call model, a user can play two mutually exclusive roles: a call initiator or a call responder. This is represented naturally by a nondeterministic choice construct:

User := ***Call_Initiator*** [] ***Call_Responder***

Within these roles, both categories of features (i.e., technological and policy) are represented by a mix of interleave and non deterministic choice constructs that are composed with other behaviours.

2.5.1 Feature Activation

As mentioned, the activation of a feature is controlled either through synchronization with actions of a user or as passive response to signals from the switching process. Guard constructs are used only to reflect feature subscription status that will turn feature behaviour expressions *on* or *off* depending on the choices of the subscriber that are represented by values of formal parameters. But here we need to stress the fact that guarding a feature's behaviour is not enough since some features have the additional side effect of restricting other behaviours. For example, integrating *call waiting* into a basic call system requires the suppression of busy signals in some states, such as the *talking* state.

```
u   !Caller      !offhook;
(   Call_Establishment [u, n](Caller)
    |||
    (   [ CallWaiting eq on ]-> Call_Waiting [u, n](Caller)
        |||
        [ CallWaiting eq off ]-> Busy_Signal [u, n](Caller)
    )
)
```

2.5.2 Interleave Constructs

Interleave constructs are used when a feature's behaviour can be activated at any point in the basic call sequence, as a concurrent alternative. This means that a feature such as *call forward on busy* or *call answer* is offered at any point in time without distracting the current active behaviour.

```
u   !Caller      !offhook;
(
    n   !Caller      !dialtone;
    n   !Caller      !dials   ?Called: Digits;
    ...
    |||
    Call_Forward[u, n](Caller, ForwardNumber)
```

```
)
where
        process Call_Forward[u, n](Caller, AnotherNumber: Digits): noexit :=
            n   !Caller      !detect_forward;
            Establish_New_Connection [u, n](Caller, AnotherNumber)
        endproc
```

In the above behaviour expression the call forward behaviour is executed in parallel with the behaviour for establishing a call, say from **A** to **B**. Assuming that **A** forwards its calls to **D**, the *Forward_Call* process may be activated, to forward calls from **C** to **D**, when **C** attempts to call **A**, while the call from **A** to **B** continues. Both passive and active features can be represented with this construct. The forward call shown above is of a passive nature, but the three way calling feature that we call active could be represented using the same type of construct.

2.5.3 Choice Construct

Choice constructs can be used deterministically when two behaviours are mutually exclusive or nondeterministically when the resolution of non-determinism resides in the environment which in this case is the server side.

```
    ...
n   !Caller      !conreq      ?Called: Digits      ?Scr_List:ScreenList;
(
    [Called NotIn Scr_List]->
        n   !Called      !ring;
        ...
    []
    [Called IsIn Scr_List]->
        n   !Caller      !Called      !Refuse_Connection;
        ...
)
```

In the above example (showing the server side) the choice between action *ring* and *refuse_connection* will be determined by the server's evaluation of the connection request using the screening list that the client has passed to the server. The client side doesn't use guard constructs in this case because it is not in control of the decision to resolve the non-determinism.

2.6 Structural Considerations for the Server

The server responds to requests from clients using a given set of operations that satisfy them. Each instance of *connection handler* handles one request at a time. For example, if a user is already connected and decides to set up a three way call, the third party is requested via another *connection request*. This one is handled by the server which creates another instance of a *connection handler*. The server is designed in a monolithic style with choices to handle the two main situations that arise due to the presence of features, i.e. the various kinds of requests and the processing of the requests. On the server side there are no specific blocks of actions that correspond to specific features. There are only resources that can be invoked by features described on the client side. These resources are similar to SIBs (Service Independent Blocks) in Intelligent Networks [DV92][Th94].

Operations that correspond to features are inserted as nondeterministic alternatives among the steps of the *Connection_Handler* process described for the basic call model. For example, the call forward feature is specified as an alternative to the *ring* or *busy* operations. Here we need to restructure the original behaviour expression of the *Connection_Handler* because a new process *Attempt_Call* has emerged from the introduction of a new operation that handles the call forward feature.

```
process Connection_Handler[n]: noexit :=
    n   ?Caller: Digits     !conreq     ?Called: Digits;
    Attempt_Connect [n](Caller, Called)
where
        process Attempt_Connect[n](Caller, Called: Digits):noexit:=
            n   !Called     !ring;
            Connect_parties...
        []
            n   !Called     !busy;
            ...
        []
            n   !Caller     !detect_forward    ?AnotherNumber: Digits;
            Attempt_Call [n](Caller, AnotherNumber)
        endproc
endproc
```

2.6.1 Feature Policy Data Passing Considerations

Policy features use data bases to help the server decide on the operations to execute in order to provide their functionalities. These data bases are represented by abstract data types, and are not shown in this paper. There will be different kinds of request handling actions where we will find the usual value of the terminating number but with some critical data that represent the conditions that trigger policy features. In fact, the switching process is strictly an execution device that by itself does not know about the user's subscribed features unless it is told. The user process is where the knowledge resides and this knowledge has to be passed on to the switch when making a connection request or when being solicited by a switch, depending on whether the user is in an active or passive state (caller or responder). For example, in the case of *Originate Call Screening* feature, the switch needs to receive the screening list in the connection request, while in the case of a *Terminating Call Screening*, the switch needs to receive the screening list from the called user to decide whether it will establish the connection or refuse it. Consequently, the connection handler will have to contain some non-deterministic choices of requesting data, depending on the type of request to be handled:

```
process Connection_Handler[n]: noexit :=
    n   ?Caller: Digits    !conreq    ?Called: Digits;
    Establish_Connection[n](Caller, Called, NoData)
    []
    n   !Caller    !conreq    ?Called: Digits    ?PolicyData: PolicyType;
    Establish_Connection[n](Caller, Called, PolicyData)
endproc
```

2.6.2 Feature Control and Busy State

In the case of the server, control is achieved by both synchronization and evaluation of guard expressions. The process *Establish_Connection* will be composed mainly of choices between behaviours that are resolved either by the evaluation of the feature policy data received in the request, or by the state in which the client is because of its activated features. For example, the decision to actually complete a connection by ringing the requested party may be taken on the basis of originate call screening data received by the switch. In this case, guard constructs are necessary:

```
[Called  NotIn Scr_List] ->    n    !Called      !ring; ...
[]
[Called IsIn Scr_List] ->      n    !Caller      !Called      !refuse_req; ...
```

Another example illustrates the case of a decision taken in the switch on the basis of the state of the called party, that can be either free, leading to a *ring*, or *busy*:

```
(   n    !Called      !ring ; ...
    []
    n    !Called      !busy ; ...
)
```

This is resolved by the called party, which is represented as follows:

```
    n ! Called ! ring;
(
    u ! Called ! answer;
    n ! Called ! voice_connect;
    Talking[u]

    |||

    n ! Called ! busy ; Busy_signal[n](Called)
 )
```

Where *Busy_signal* is a process that does nothing but offer *busy* recursively. After there has been synchronization between client and server on *ring*, another connection attempt by another instance of the switch can only synchronize on the *busy* action that is in interleave with the normal connection path of the client.

As can be seen in the two above examples, the control of a feature can be internal or external. The internal control results from the evaluation of guard expressions, while the external control results from synchronization with actions of the client.

2.7 Control via Client-Server Interactions

In the two previous sections we have considered the structure of the two main components separately. Now we need to demonstrate how they interact to achieve the functionality of a given feature. We will look at two examples illustrating respectively the technological and the policy feature cases. These are illustrations of the principles shown in Figure 2.

2.7.1 Example of Technology Features: Call forward

From a user's point of view, we distinguish between *Call Forward Always (CFA)* and *Call Forward on Busy (CFB)*. The first feature allows a user to forward all his/her incoming calls to a second (predetermined) number; the second one forwards incoming calls only if the

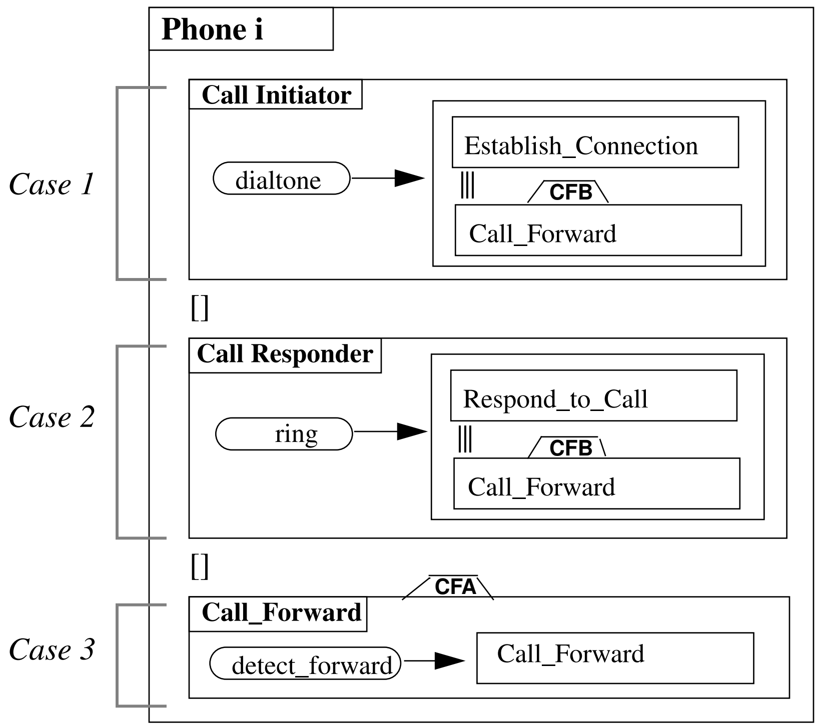

Fig. 3. Structure of a phone design supporting Call Forward

user is busy. From a structural point of view, both *CFA* and *CFB* can be designed to make use of a single resource represented by a *Call_Forward* process, whose functionality is to establish a connection between two users: the one to whom the call is originally directed and the one to whom the call is forwarded. However, the different instances of this process need to be guarded appropriately in order to reflect the user's subscription choices. Figure 3 shows the structure of a phone which is designed to support both *CFA* and *CFB*, although a user may not want to activate both of them at the same time. Since a phone can play the roles of call initiator or call responder, then we must make provisions to support both *CFA* and *CFB* for each of the roles. A structure of a well designed phone must consider the following cases:

2.7.1.1 Support for the *CFA*

This is shown as case 3 in Figure 3. It is expressed as a third alternative to the *call initiator* and *call responder* roles. This alternative is guarded by a *detect_forward* condition, which indicates whether or not the feature is activated. If it is, this alternative is chosen regardless of whether the user is idle or busy. Note that process *Call_Forward* is recursive, meaning that a forwarded phone offers continuously the *forward* signal. This is all process *Call_Forward* does. As shown below, the actual forwarding is done by the switch.

2.7.1.2 Support for the *CFB* in a call initiator role

In a call initiator role, a phone becomes busy as soon as user **A** sends an *offhook* signal to the switch (i.e., **A** picks up the handset) and receives a *dialtone*. This suggests that, structurally, the *Call_Forward* process must be placed within the *call initiator* alternative, and behaviourally, its execution is triggered by the *dialtone* action, as soon as another user calls **A**. This is shown as case 1 in the figure. Also, note that the *Call_Forward* process may execute in parallel with the *Establish_Connection* process (*Call_Forward* ||| *Establish_Connection*), which means that (1) a call directed to **A**, during any stage of the call processing phases after **A** has become busy, is forwarded to another predetermined number, and (2) user **A** proceeds with its normal call processing behaviour.

2.7.1.3 Support for the *CFB* in a call responder role

In a call responder role, a phone (**B**) becomes busy as soon as a connection path (circuit) is reserved for communication between the calling user **A** and the called user **B**. This suggests that, structurally, the *Call_Forward* process must be placed within the *call responder* alternative, and behaviourally, its execution is triggered by the *ring* action, as soon as another user calls **B**. This is shown as case 2 in the figure. Again, note that the *Call_Forward* process may execute in parallel with the *Respond_to_Call* process (*Call_Forward* ||| *Respond_to_Call*), which means that (1) a call directed to **B**, during any stage of the call processing phases after **B** has become busy, is forwarded to another predetermined number, and (2) **B** proceeds with its normal call processing behaviour.

2.7.1.4 A Call Scenario in the Context of Call Forward

Figure 4 shows the invocation of the different instances of the *Call_Forward* process in the context of a typical call from user **A** to user **B**. We assume that both phones (*A* and *B*) are instances of the template phone of Figure 3. Also, we assume that **A** is the caller and **B** is the called. In addition, we assume that **A** and **B** are in idle state and have activated *CFB*, but not *CFA*. User **A** starts the call scenario by sending an *offhook* signal to the switch (not shown in the Figure). The switch marks **A**'s state as busy and sends a *dialtone* to **A**, as shown by the synchronization point ①. After a connection request (*conreq* action in the Figure) by the switch and the selection of a communication path, the switch sends a *ring* to user **B**. This is shown by ② in the figure. Now, suppose that while **B** is ringing, a third user **C** (not shown in the Figure) attempts to call **B**. Once the connection handler for **C** detects that **B** is busy and that **B** has activated *CFB*, synchronization occurs between the action *detect_forward* (executing in the context of the **C** connection handler) and same action in process *Call_Forward* in the behaviour (*Call_Forward* ||| *Respond_to_Call*) of **B**, as shown by ③. **C**'s connection handler will then take over the execution to establish a connection with the user to whom the call is forwarded.

Other scenarios can be easily constructed by making different assumptions about users' states and activation of features.

2.7.1.5 Server design

The design of the server is considerably simpler. First of all, the call forward detection action is an alternative to the ring operation. Then if a call forward has been detected, the switch merely attempts to establish a new call using the new number received from the client to which the call is forwarded. This results in creating a new instance of process *Attempt_connect* with the forward number as a formal parameter. There is no need to distinguish between CFA or CFB since the result is the same. Note that the *detect_forward* action in the new instance of process *Attempt_connect* could apply to the new number in case this one also has activated its call forward feature. This process is truly reusable.

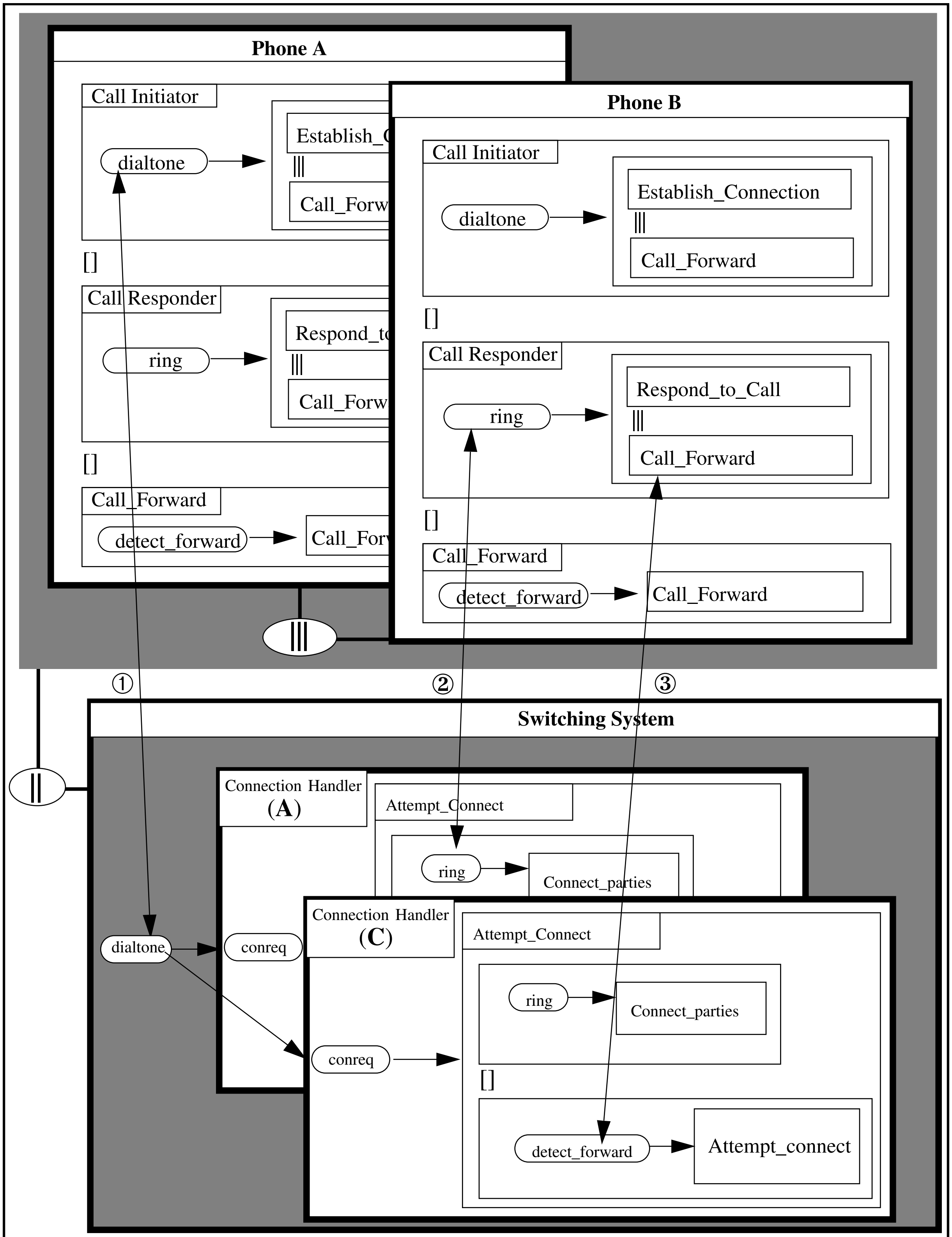

Fig. 4. Structure of a phone design, which support Call Forward Busy

2.7.2 Example of Policy Features: Originating Call Screening

Originating Call Screening is a feature which allows a subscriber to prevent outgoing calls to be made to a predefined set of numbers. It is an interesting example of the policy features class. First, the policy data is located in the client entity. When a request for this feature

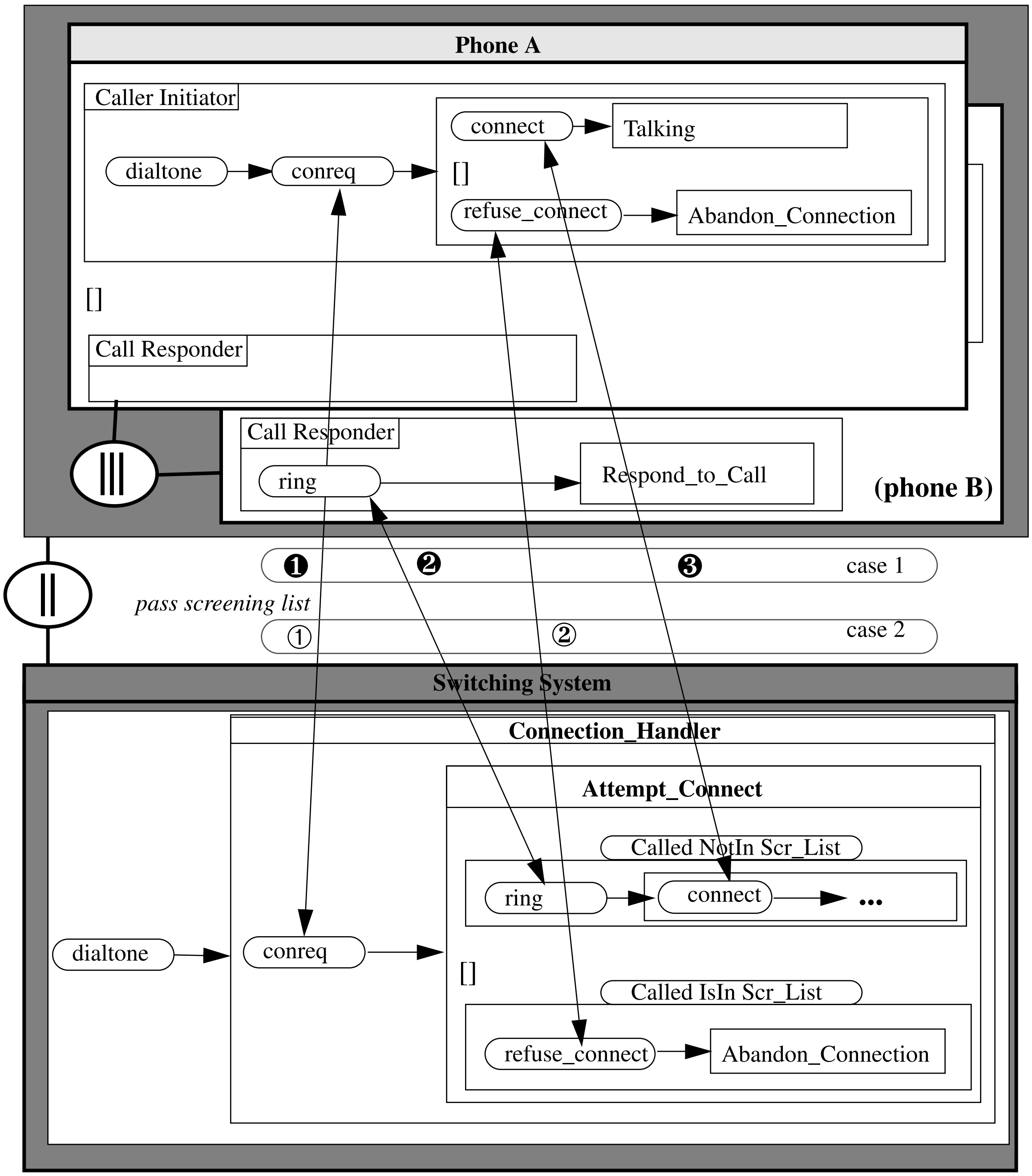

Fig. 5. Originate Call Screening Example

is made to the switch, it will include the screening list in the data passed to the switching process. Using this data, the switch will evaluate the request of the client and choose only one of the two alternatives of ringing the requested number, or sending back a connection refusal to the client. Here nondeterminism is resolved essentially by guard constructs. The client itself is ready to synchronize on either the *connect* or the *refuse_connect* action depending on what becomes available on the server side. However in order to synchronize on the *connect* operation with the originating client, the switch first needs to synchronize on the *ring* operation of the terminating client. Here the guards are used to enforce a policy, rather than to signal availability of the feature.

Let us trace two typical call scenarios, cases 1 and 2 in Figure 5, from **A** to **B** in the context of *OCS*.

Assuming that **A** is the call initiator, and that **B** is not in the screening list of **A**, when a connection request (*conreq*) is made, the list of numbers to be screened is passed from the phone process to the switch ❶. Assuming that the called number **B** is idle, a *ring* is sent from the switch to user **B** ❷. Finally, the voice exchange phase begins as shown by ❸.

Now let assume that **B** is in the screening list of **A**, shown as case 2 in the Figure. As soon as a connection request is made ①, the switching system will refuse the connection ②, and proceeds to abandon the call and release the system resources.

3. The Constraint-oriented Specification Style

3.1 POTS again

Many of the specification concepts we have seen for resource-oriented style also apply to resource-oriented style, however there is a change in perspective. While in the resource-oriented style processes play the role of physical system components, in the constraint-oriented style they express logical constraints that must be satisfied by the system [Boch80][ISO88b][TrVs95][VSVB91]. The parallel composition operators acquire a logical meaning in this style. For example, the expression $P_1 \parallel P_2 \parallel P_3$ means that every action in the system must be the result of synchronization, or 'agreement' of all three processes (as well as of the environment) on that action. In other words, it means that the requirements or 'constraints' of all three processes must be simultaneously satisfied by the environment. The expression $P_1 \mid\mid\mid P_2 \mid\mid\mid P_3$ means instead that each action must satisfy the constraints of at least one of the three processes. As we have just implied, there are no hidden gates in a constraint-oriented LOTOS specification, all gates are external and all actions of all processes require participation of the environment. Thus, specifications written in this style describe externally observable behaviour only.

Using this perspective, we identified three types of constraints for a POTS specification [FaLS91]:

(1) *Local constraints* are used to enforce the appropriate sequences of events at each telephone, and are different according to whether the telephone is a *Caller* or a *Called*. For example, on the caller side, *dials* must proceed *talk*. Therefore local constraints are represented by 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, they are composed by the interleaving operator |||. Typical behaviours are shown in Figure 7.

(2) *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 necessarily follow dialling at the *Caller*. Process *Controller* enforces these constraints. Because they must apply to both *Caller* and *Called,* we have the structure *(Caller* ||| *Called)* || *Controller.* Thus the controller must participate in every action of the *Caller*, as well as in every action of the *Called,* separately. Figure 6 shows an instance of this structure. Its behaviour is shown in Figure 8. Each such structure constitutes a *connection*. An arbitrary number of connections is created by recursive interleaving in a process *SystemConnections*, see below.

(3) *Global constraints* are system-wide constraints. In our specification we identified one main such constraint, which is the fact that at any time, a number is used at most once. This constraint is enforced by the process *GlobalConstraints*. Because global constraints must be satisfied simultaneously over the whole system, represented by process *SystemConnections,* we have the structure *SystemConnections* || *GlobalConstraints.*

In Figure 6, local constraints are expressed by **A** and **B**, end-to-end constraints are expressed by C_1, while global constraints are implied. LOTOS gates (represented by dark squares) are used to structure the specification according to the three different phases of a telephone communication: connection, talking, and disconnection. The reader unfamiliar with process algebra should note that lines denote shared gates, and not busses. Assuming that **A** plays a caller's role and **B** plays a called role, then for the connection phase, process C_1

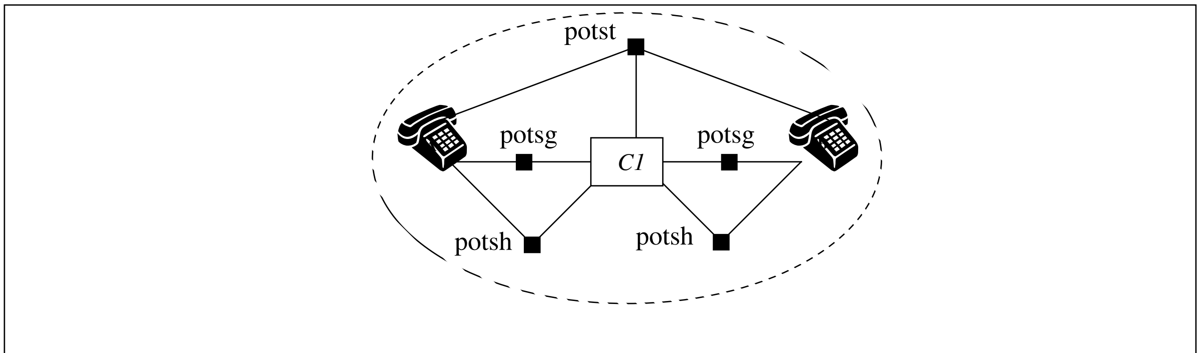

Fig. 6. Structure of a POTS connection.

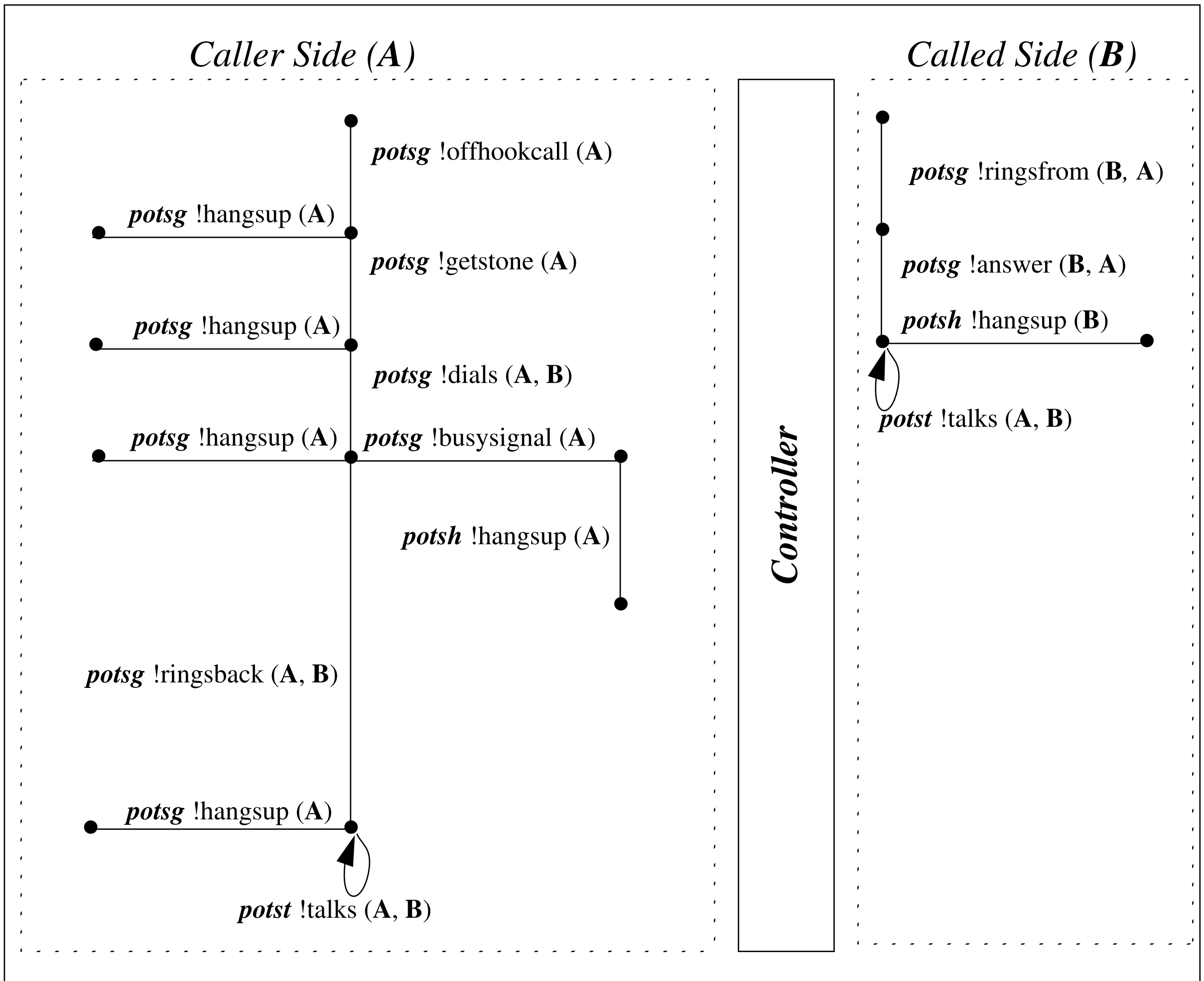

Fig. 7. Behaviour of *Caller* and *Called* with respect to *Controller*

synchronizes with either **A** or **B** to exchange signalling information such the *offhook* and *dialtone* signals; this type of synchronization is achieved through the gate *potsg*. Once the connection is established, the voice exchange phase is achieved through a three-way communication between **A**, **B**, and C_1. This synchronization is represented by the LOTOS gate

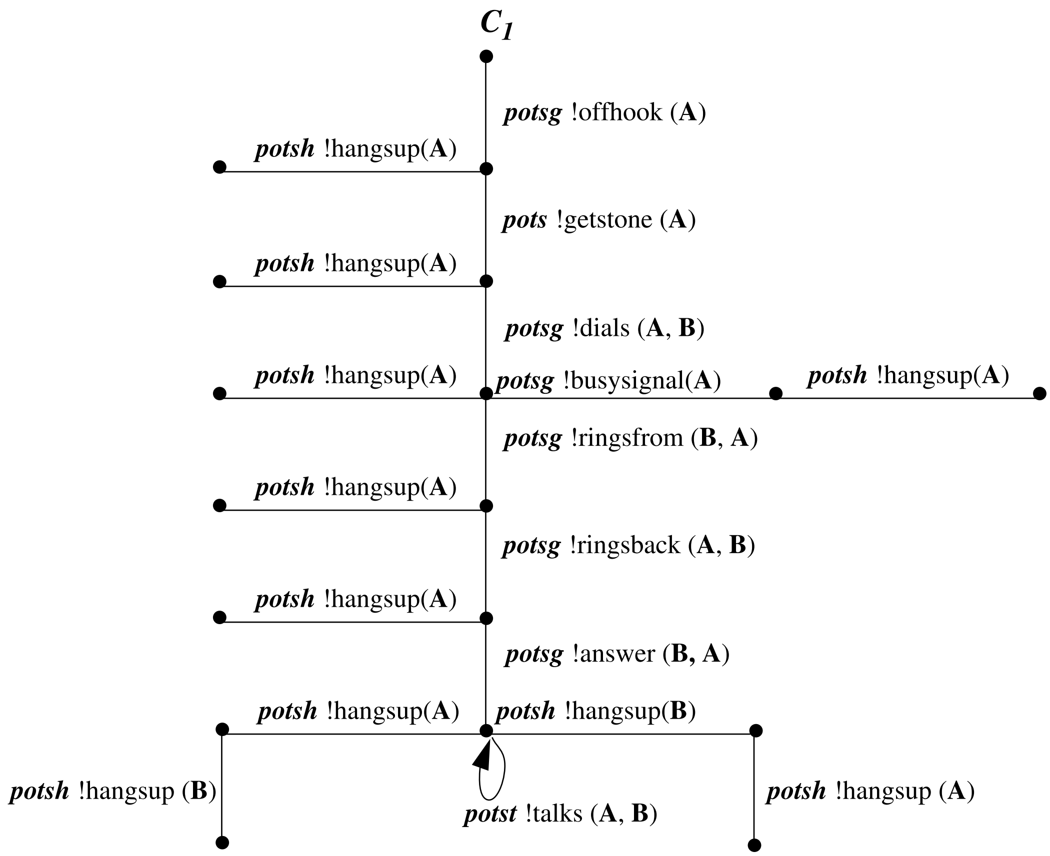

Fig. 8. Behaviour of *Controller* with respect to *Caller* and *Called*

potst (which expresses the talk phase in a connection). Finally, the hang up phase is expressed by the LOTOS gate *potsh* (which expresses the hang up phase in a connection), where C_1 has a two-way synchronization with either **A** or **B**. By restricting the synchronization to two-way communication, we allow for the independent hang up from either **A** or **B**. As an example, synchronization between **A** and C_1 may occur on *potsg* only if **A** in not busy. In fact, taking the global constraints into consideration, the above synchronizations become three-way (for the gates *potsg* and *potsh*) and four-way (for *potst)*. Figures 7 and 8 show the relevant behaviour trees.

We now provide the details of the general structure already introduced above. The basic structure of a constraint oriented description of a telephone system is illustrated in Figure 9.

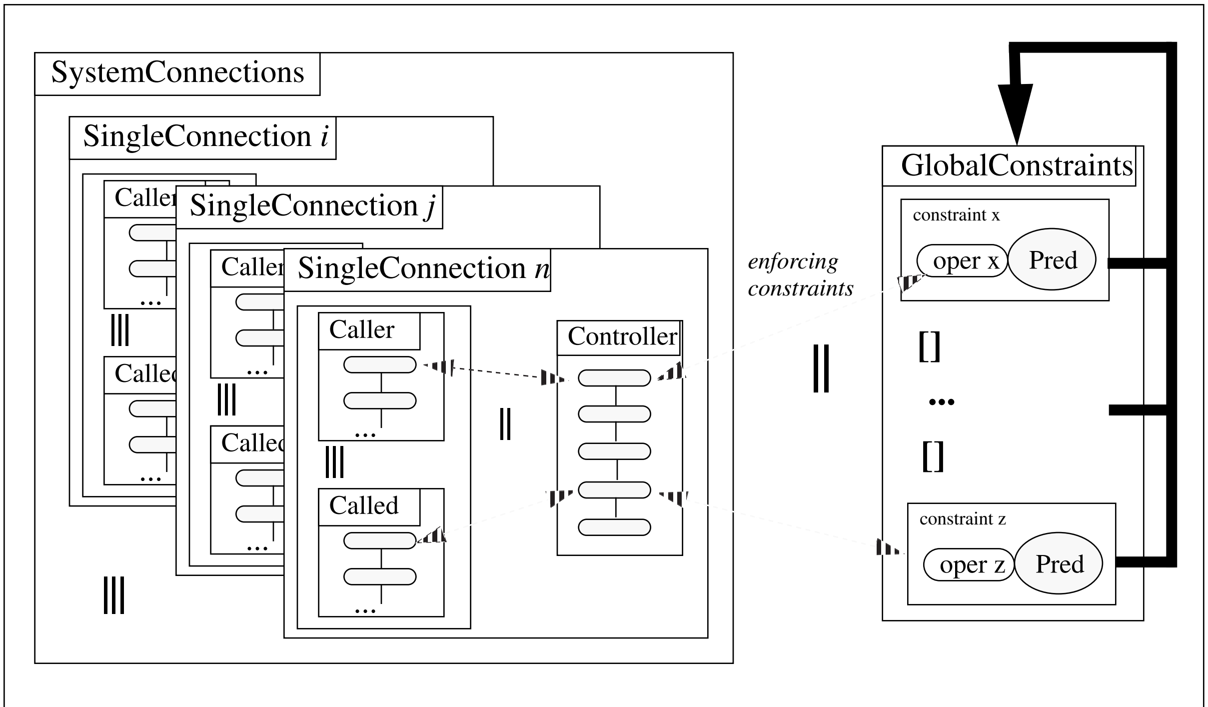

Fig. 9. Abstract view of a POTS connection

The top-level behaviour is composed of two processes, *SystemConnections* and *GlobalConstraints*. Stated informally, we want to create as many connections as desired provided that neither the calling nor the called number is already in use. *GlobalConstraints*, which we will describe later, enforces global constraints by keeping track of free and busy numbers and synchronizing with *SystemConnections* to exchange values:

SystemConnections[potsg, potsh, potst]

||

GlobalConstraints [potsg, potsh, potst](BusySet)

The parameter to the Global Constraints process is the set of busy numbers, which is empty at the beginning.

Process *SystemConnections* is composed of two processes: *SingleConnection* interleaved with *SystemConnections* itself. This creates the desired effect of being able to have an arbitrary number of connections existing simultaneously. At the initiation of a connection, the identity of the called party is unknown but its template behaviour is ready to be instantiated.

process ***SystemConnections*** [potsg, potsh, potst] : **noexit** :=

(

SingleConnection [potsg, potsh, potst]

```
        |||
        i; SystemConnections[ potsg, potsh, potst]
      )
  endproc
```

The process *SingleConnection* is viewed as the composition of three processes: *Caller*, *Called* and *Controller*. The conceptual notion of modeling the call initiator (*Caller*) side and the call responder (*Called)* side by two interleaved processes is quite natural; it reflects the distributed nature of the system, in that local constraints apply to separate portions of behaviour. *Caller* and *Called* exchange information by synchronization with the *Controller*.

```
process SingleConnection [potsg, potsh, potst] : exit :=
(   (
        CallerHandler [ potsg, potsh, potst]
        |[potst]|
        CalledHandler[potsg, potsh, potst]
    )
    ||
    Controller [potsg, potsh, potst]
)
endproc
```

3.2 The Constraint-oriented Control Mechanism

As in the resource-oriented style, control is achieved not only by the temporal ordering of the actions of the constraint processes, but also via predicates attached to actions. Predicates involve the evaluation of abstract data types, which mimic data structures which are maintained to record the states of phones. They are consulted for each operation occurring in the system to determine if an action is available or visible, thus allowing or disallowing potential synchronizations which cause triggering of behaviours. The process *GlobalConstraints* manipulates the data structures of the specification. It synchronizes with *SystemConnections* in order to update the list of busy numbers. The mechanics of updates are best described with respect to the data structure itself.

Our first data structure *EngagedSet* is a set of pairs which records the engaged pairs, where two users are engaged if the called user is the ringing state. The caller is inserted in the

EngagedSet when an offhook is executed. The use of *EngagedSet* is illustrated in Figure 10.

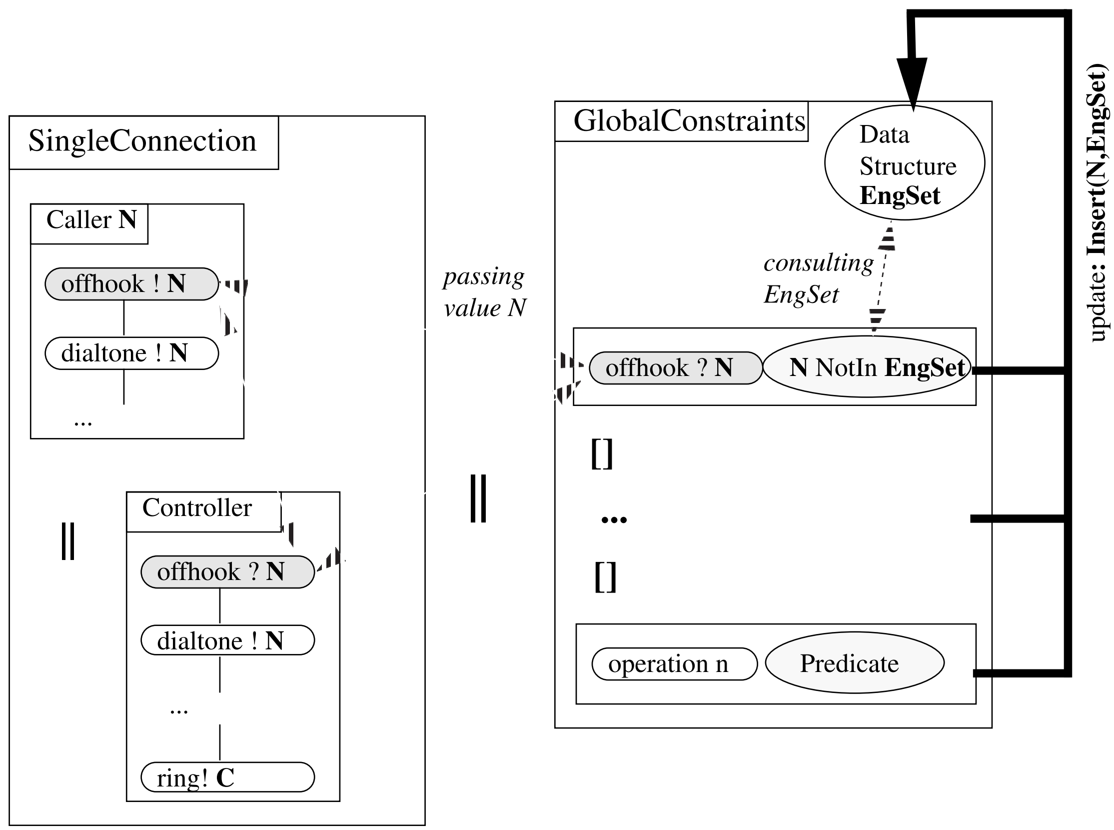

Fig. 10. Selecting *offhook* requires consultation of *EngSet* (engaged set)

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*. If the *Called* is not busy, the *Rings* event is executed and the associated phone number is added to the set *BusySet*. If the called number is busy then the caller must receive a busy signal and the *BusySet* is not modified. Figure 11 presents a diagrammatic view of the use of this data structure.

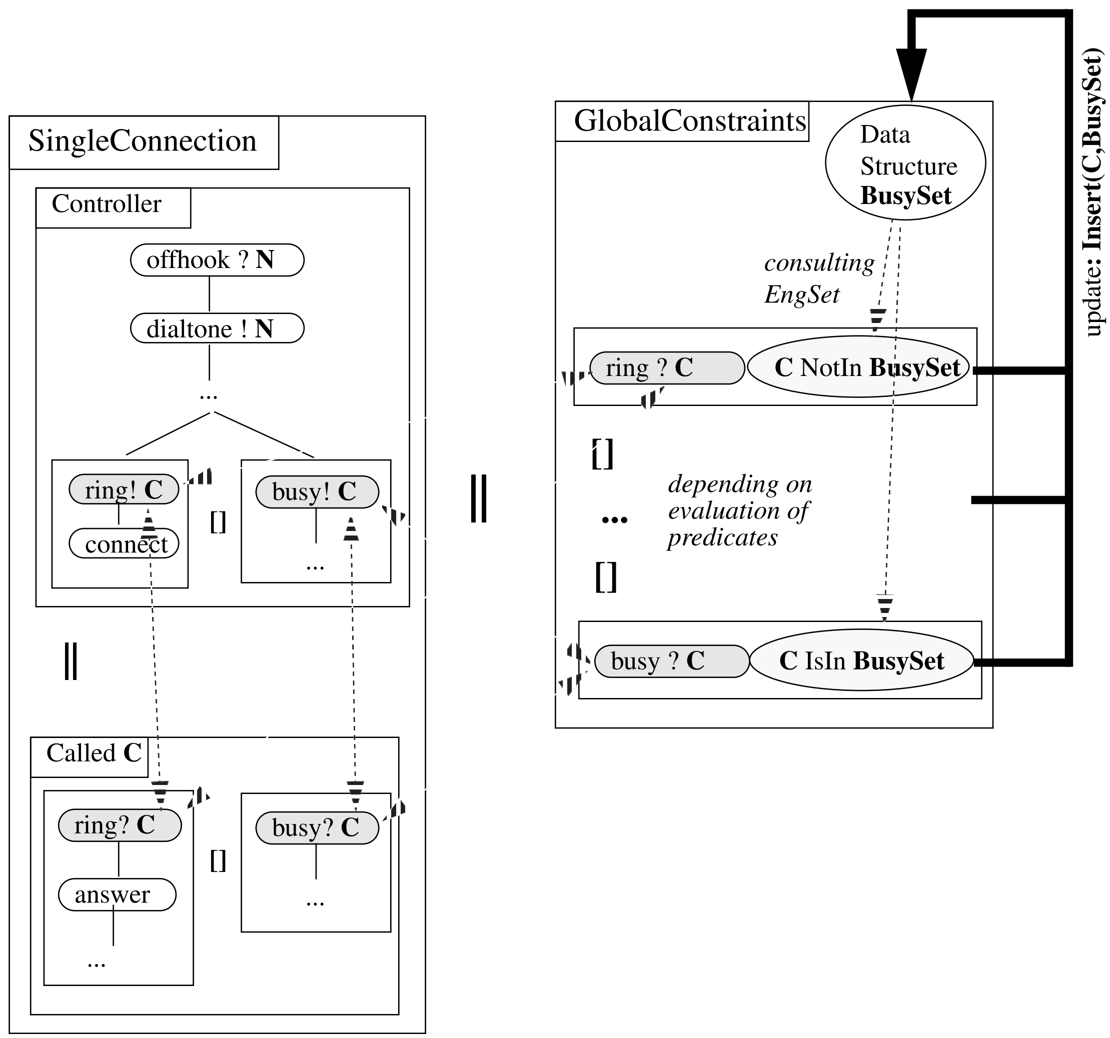

Fig. 11. Selecting *ring* or *busy* based on *BusySet*

Additional details about a complete POTS specification in LOTOS are given in [FLS91].

3.3 Specification of Features in the Constraint-oriented Style

With some simplification, we define a feature as an extension of the functionality of an existing telephone system. In general, a feature extends either the calling side or the called side, or both. To extend a system with a new functionality, we first decide on the *role* of the feature, which can be derived from its informal description. The integration of the feature's behaviour into the system is accomplished by making the appropriate modifications to the user on which the feature is to be activated, as well as to its controller.Thus, in the constraint-oriented style, features are themselves constraints. In the most basic terms, if one adds a feature **F** to a POTS system **A** (playing a caller or called role), the local constraints of the combined system will be specified as **A** |[L]| **F**, for some set of gates L, thus allowing a pleasing modularity. Of course, the processes representing the other constraints may also need to be modified in order to properly synchronize with **F.**

Two examples are provided to illustrate the two basic categories of features, the technological and policy features.

3.4 Technological Feature: Three Way Calling

The *three way calling* feature (*Twc*) is specified as an instance of the connection entity. When a user adds a third party to a call, another connection entity is instantiated. The mechanism for achieving a three way communication in the context of a POTS connection is the following.

Figure 12 shows the static structure of the POTS connection (a) and of the *Twc*

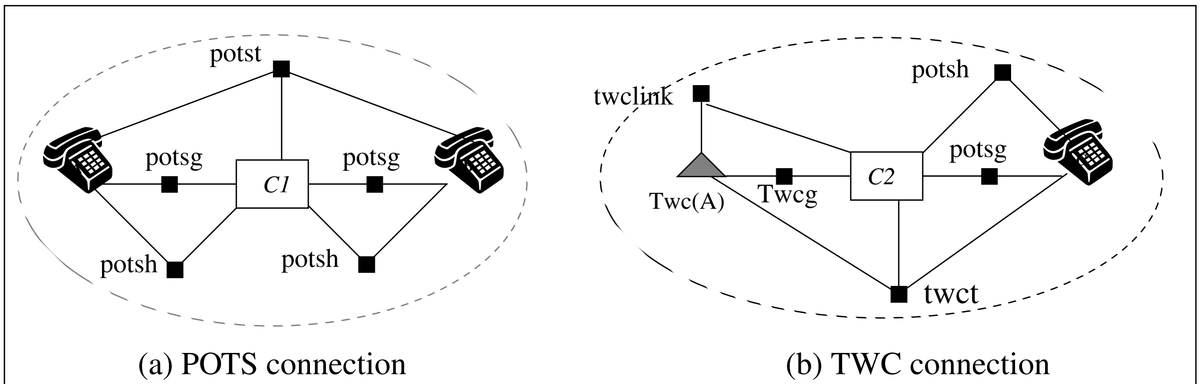

Fig. 12. Structure of a POTS and TWC connections in isolation

Connection (b). The *Twc* structure of (b) is very similar to the POTS structure of (a), except for the calling side of the connection which now represents the local constraints of the *Twc* feature.

From the analysis of the informal description of the feature, we deduce that *Twc* has a calling role, meaning that the first action of the feature requires synchronization with the caller.

Before we give the complete behaviour of the extended specification, let us identify the local and end-to-end constraints, which express the formal specification of *Twc*, in isolation.

To define the constraints on this feature we analyse the sequence of actions that can be exchanged between the feature and its environment, namely the switching system and subscriber **A**. A behavioural representation of the feature is shown in Figure 13. Assuming that the feature executes in a context where **A** is talking to **B**, **A** starts the feature with a *flashhook* signal, then continues in a similar fashion as in POTS, until a talking state between **A** and **C** is reached. After the first *flashhook*, **A** may cancel the invocation of *Twc*, by sending a second *flashhook* before the talking state is reached. If the feature is cancelled, it returns to its initial state; the next *flashhook* from the initial state invokes a new instance. If the talking state

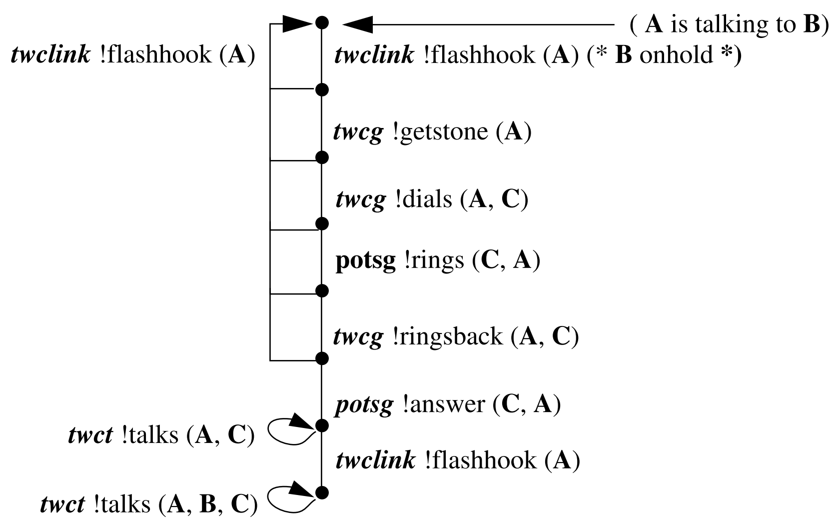

Fig. 13. Simplified LTS of the three way calling feature

between **A** and **C** is reached, the feature allows for another flashhook, which permits the system to re-activate the connection between **A** and **B** while maintaining **A** and **C** in a talking state.

The structural integration of the subcomponents (i.e., POTS and *Twc*) is shown in Figure 14. However, the structure of the connections is only part of the solution for providing the three way calling feature in the POTS context. The other part is the integration of the two structures so that the desired behaviour is achieved. If we assume that **A** has a calling role and both **B** and **C** have a called role, then the two processes **A** and ***Twc(A)*** must have a common synchronization point which allows user **A**, while in a talking state, to flash the hook and transfer control to the Twc feature. This is done by identifying the *potsh* gate of process **A** with gate *twclink* of process ***Twc(A)***, by using the LOTOS gate relabeling feature. The behaviour of the resulting structure is shown in Figure 15. Note that we have abstracted from LOTOS gates in the figure. For example, the *flashhook* signal results from synchronization of the two sub-connections and the global constraints process with the user on gate *TwcLink* but this gate is not shown in the figure.

Note that the Twc behaviour of Figure 13 is integrated in that of Figure 15, although the

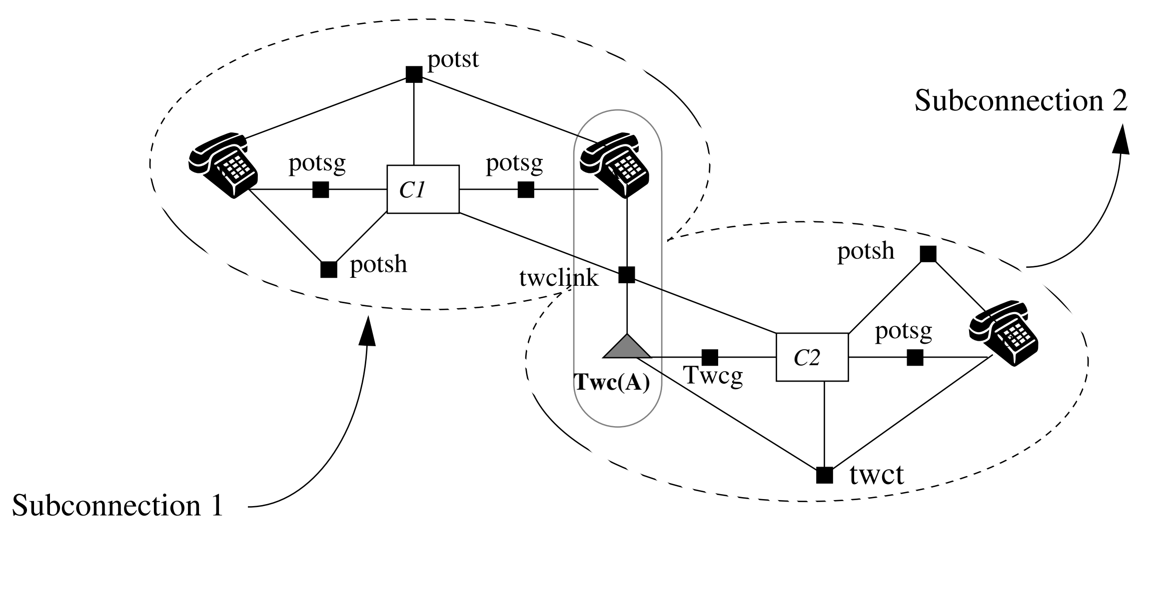

Fig. 14. Integration of *Twc* into POTS: Top level structure

arrow back to initial state is not shown.

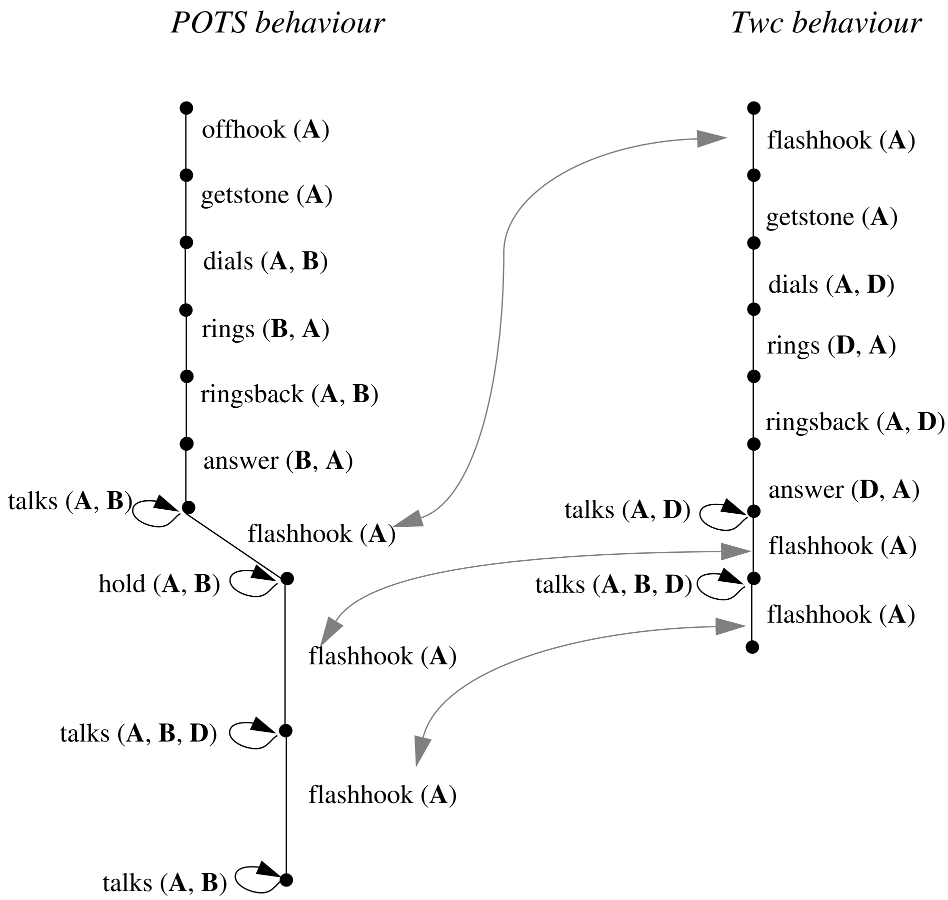

Fig. 15. LTSs of *Twc* in the context of POTS.

Thus, the integration of this feature into POTS requires the specifications of the following two subconnections:

- Subconnection 1 := (***Pots***(**A**) |[potst]| ***Pots***(**B**)) || $\boldsymbol{C_1}$
- Subconnection 2 := (***Twc***(**A**) |[twct]| ***Pots***(**C**)) || $\boldsymbol{C_2}$

As mentioned, each subconnection has the same general structure as a single POTS connection. This is illustrated in Figure 16. However some adaptations are necessary. We need to modify the calling side of a POTSConnection process so that it communicates with the process *TwcConnection*, whose behaviour is expressed by subconnection 2. We also need to respecify the corresponding behaviour of the *POTSController* (C1) and the *TwcController* (C2) processes so that they relate to each other, and we need to compose the two subconnections so that they synchronize on their common actions. Synchronization between the subconnections occurs on gate *TwcLink*, which is used for exchanging the signals *flashhook* and *hangsup* which occur on **A**.

Finally, each modification of the local or end-to-end constraints requires a modification of the global constraints to support the additional behaviour. In addition to the *BusyList* and *EngagedList* that we described previously, three additional lists are required to maintain the global view of the system, in the presence of *Twc*:.

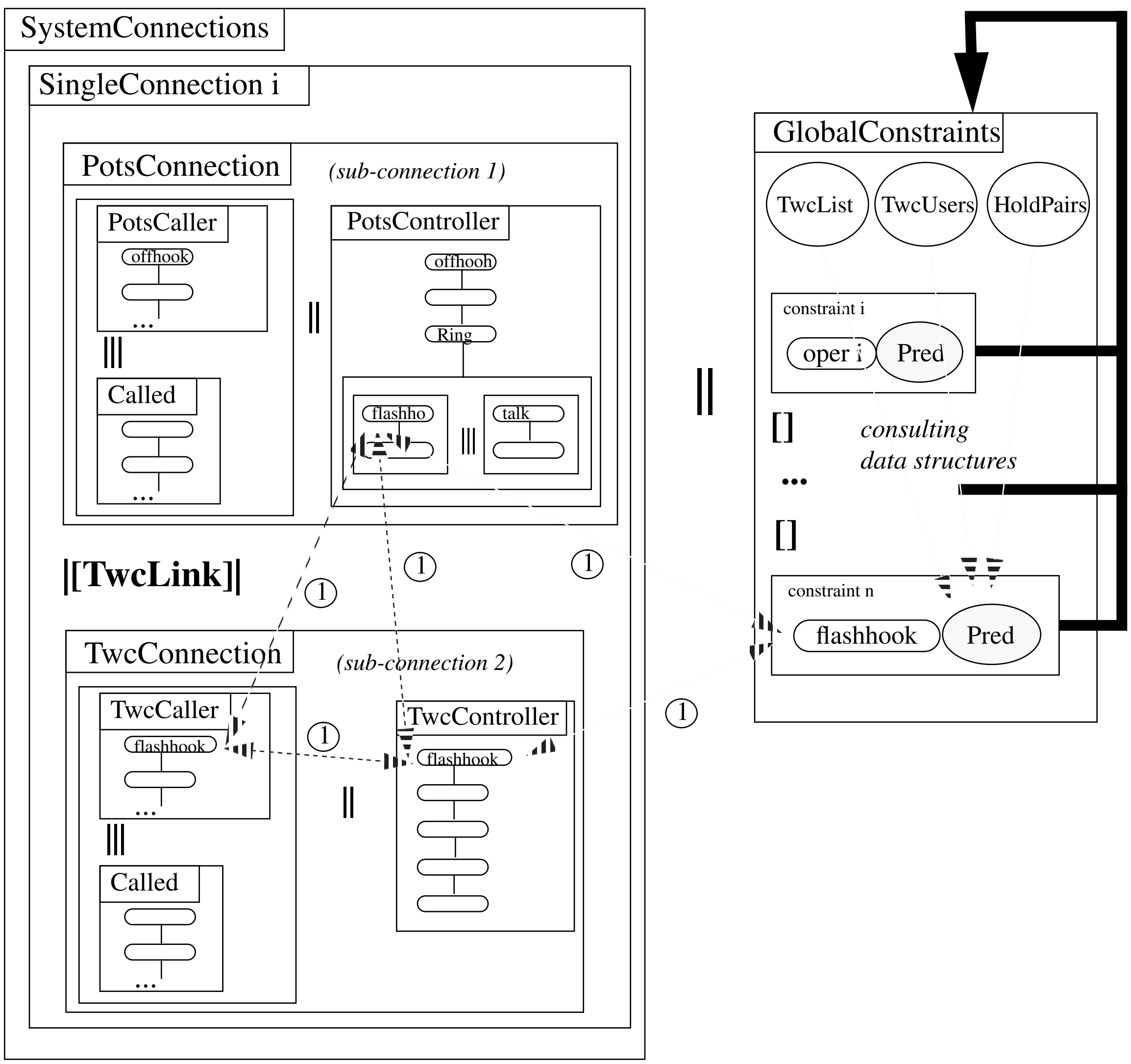

Fig. 16. Abstract representation of three way calling feature

- *TwcList*: A list of single elements, where each element is a *Twc* subscriber; this is a static list. A user can execute a flash hook signal only if he/she is in this list.

- *TwcUsers:* The list of users who have activated the feature. This is a subset of *TwcList.* A user (**A**) is inserted in this list when synchronization occurs on the first flash hook signal. If **A** reaches a talking state with **C**, the second flash hook has no effect on the list. If **A** abandons the call before **C** rings, or if **C** does not answer, the second *flashhook* has the effect of removing **A** from the list. If **C** rings, **A** is removed from the list when the third *flashhook* occurs, or when **A** hangs up.

- *HoldPairs:* List of pairs (**A**, **B***)*, where **A** has put **B** on hold, by way of the *flashhook*. If the second *flashhook* occurs before the talking state is reached, the pair is used to identify the user to which a ring reminder is to be sent. If it occurs after, the pair is removed from the list.

3.5 Policy Feature: Originating Call Screening

This feature has a caller role, yielding the structure shown in Figure 17. It imposes

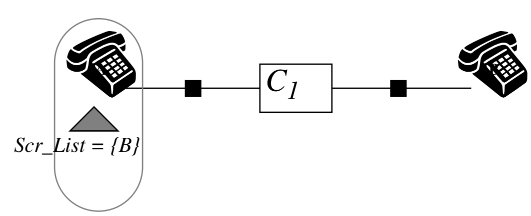

Fig. 17. Structure of the specification which integrates *Oc*s into POTS

further constraints on the caller side with respect to the POTS connection. One such constraint allows a called party **B** to ring for **A** only if **B** is not in the *Scr_List* of **A**. So, after the *dials(**A**, **B**)* action in the LTS, the system now offers a new alternative, the action *refuse(**A**,**B**)*, to indicate that a connection from **A** to **B** is not possible if **B** is in the *Scr_List*. If **A** dials a user who is not in the list, then the system's behaviour is reduced to that of a normal POTS call.

This feature can be naturally specified by adding the *Scr_List* to the data structures of the global constraints process. Consequently we need to add another formal parameter to the *GlobalConstraints* process, modify the predicates associated with the *ring* action and add another action to indicate that a refusal has occurred. The new data structure is somewhat different from the *Busy* or *EngagedSet* data structures because it is set once for all and is not updated via events as we have seen before. On the *SingleConnection* side we need also to add a choice construct to indicate that a connection can be refused.

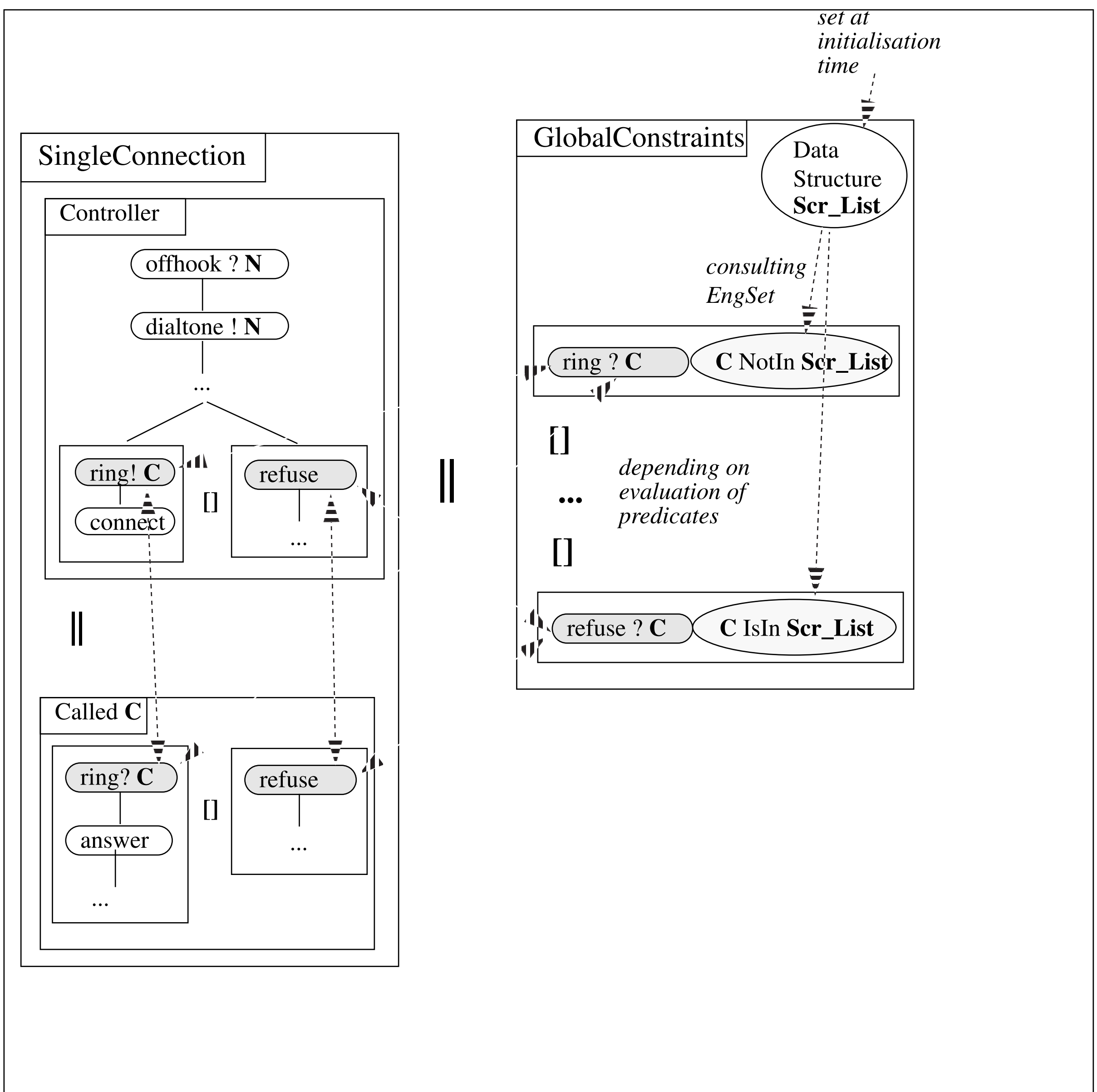

Fig. 18. Originating Call Screening Feature

In Figure 18 we observe how the nondeterministic choices of the *Called* and the *Controller* processes are resolved with the evaluation of the predicates of the *ring* and *refuse* operations in the global constraints process. The nature of these predicates makes the associated operations mutually exclusive.

4. Comparison of the Resource and Constraint oriented specification styles

The two specification styles we have presented have each its own applications and advantages. The LOTOSPHERE software development methodology [BvdLV] suggests a specific role for each specification style in a software development trajectory. The constraint-oriented style is closer to the requirements level, while the resource-oriented style is closer to the implementation level. Our results support this view.

4.1 General Structure

The resource-oriented style allows a structuring of the specification that can be visualized easily in terms of action sequences. The features are strongly associated to states in both the client and the server. Control code of features can easily be located in relation to the basic call model.

The constraint-oriented approach allows the designer to define the system as a black box which interacts with its environment via a well defined set of primitives. From this perspective, the structure is appealing. Once the observable global behaviour is captured, the black box can be decomposed into several types of constraints which help the designer to achieve a clear separation of concerns. Local constraints localize design issues so that both the caller entity and the called entity are expressed in terms of their allowed sequences independent of each other. End-to-end constraints allow the designer to establish a certain dependency between the local constraints so that only valid sequences are selected. Finally, the global constraints allow the designer to impose constraints reflecting the global behaviour of the system. The features themselves become added constraints. They play a local role, but may require changes in end-to-end and global constraints.

4.2 Control Mechanism

In the resource-oriented approach, the control mechanism involves at most two entities at a time.

In the constraint-oriented style, control is expressed incrementally at several levels, three in our example. Processes representing local, end-to-end, and global constraints must synchronize together. The advantage of the control mechanism in the constraint-oriented approach is that restrictions on the set of valid system sequences are incremental. In other words, the designer starts with the set of all potentially valid sequences for each component, and restrictions on the set of all possible shuffles of these are imposed at different levels, which makes the design more manageable. The drawback of this approach is that at the global constraint level, the data management becomes complex, because at that level all constraints become composed.

4.3 Incrementality

In constraint-oriented style, an added feature is simply an added constraint. Unfortunately, however, adding features in this style is more difficult than doing so in resource-oriented style. Because of the fact that it is not allowed to hide actions in the constraint-oriented style, changes to several processes are usually required in order to take into account the new actions belonging to added features. In resource-oriented style, the hiding mechanism makes it easier to add processes locally, without affecting other existing processes.

4.4 Practical considerations

The constraint-oriented style is rather conceptual, and requires a good level of expertise and sophistication of the user. The use of the LOTOS process synchronization mechanism, as well as the use of abstract data types, are much more sophisticated in this style. In addition, the style is very implementation-independent, and it appears to be more conducive to thinking in terms of abstract requirements. The structure is not an implementation structure, rather it is a requirement structure.

On the other hand, the resource-oriented style can be learned quickly and is much more implementation-oriented.

It is not by chance that our discussion of the constraint-oriented style was dominated by high-level processes, while our discussion of the resource-oriented style was dominated by low-level action sequences.

It should not be surprising that resource-oriented specifications tend to be much longer than constraint-oriented ones, especially if an attempt is made to describe distribution to a greater extent than we did in this paper. The description of distributed components, and of processes representing channels, with the related mechanics, can be quite lengthy. The difference between the two styles becomes much more obvious, since all these details have no place in the constraint-oriented style.

5. Conclusions and Research Directions

For conventional telephony systems, the typical time requirements for introducing a new feature to the market was reported to be in the range of three years [Mart88]. Ideally, this should be in the 2-3 months range.

Several conceptual frameworks are being proposed to shorten this development time. Formal techniques have a place because they allow logical modeling, prototyping, and validation of features at the design stage. Possible problems, such as feature interaction, can be detected at this stage.

This paper presents and contrasts two structural paradigms for formally specifying telephone systems and their features. The two paradigms view the systems from different perspectives, and highlight different aspects of interest at the initial stages of the system development process. These approaches have been successfully used to extend POTS with a number of features [B91][BL93][SL94][SL95][Faci95] such as Call Waiting, Call Forward on Busy, Call Forward Always, Automatic Recall, Automatic Callback, Originating and Terminating Call Screening, Distinctive Ringing, Calling Number Delivery, and Unlisted Numbers.

The subject of feature interactions was not addressed in this paper. A *feature interaction* is defined as the interference of the functionality of one telephone feature with the functionality of another telephone feature, meaning that the invocation of the first feature modifies the functionality of another active feature, or even prevents its functionality altogether. This problem has become a major obstacle for the extension of telephone systems with new services. There is a considerable body of literature on the subject. Some references are [BDCG89][BL93] [CGLN94] [SL94] [Fits95] [SL95] [Faci95] and [FaLo96].

The Intelligent Network model [DV92][Th94] provides architectural solutions for rapid feature introduction. We are studying specification structures corresponding to the various aspects of this model. Our initial results, which include a method for feature interaction detection, are reported in [Kam96].

Acknowledgment

We would like to acknowledge the invaluable comments made the anonymous referees, that lead to a significant improvement of the paper, both in content and style. This research was funded in part by Bellcore, the National Institute of Standards and Technology (USA Dept. of Commerce), and the Telecommunications Research Institute of Ontario.

References

[Boch80] G. V. Bochmann. A General Transition Model for Protocols and Communication Services. IEEE Trans. Comm., 28 (1980), 643-650.

[BB87] B. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25-59.

[BFLL95] T. Bolognesi, D. DeFrutos, R. Langerak, D. Latella. Correctness Preserving Transformations for the Early Phases of Software Development. In: T. Bolognesi, J. v.d. Lagemaat, and C. Vissers. *LOTOSPHERE: Software Development with LOTOS.* Kluwer, 1995.

[BNT94] T. Bolognesi, E. Najm, and P.A.J. Tilanus. G-LOTOS, a Graphical Language for Distributed Systems. Computer Networks and ISDN Systems 26 (1994) 1101-1127.

[BvdLV] T. Bolognesi, J. v.d. Lagemaat, and C.Vissers. *LOTOSPHERE: Software Development with LOTOS.* Kluwer, 1995.

[B91] R. Boumezbeur. Design, Specification and Validation of Telephony Systems in LOTOS. Master thesis, University of Ottawa, 1996. Available by ftp on lotos.csi.uottawa.ca.

[BL93] R. Boumezbeur, L. Logrippo, Specifying Telephone Systems in LOTOS, IEEE Communications Magazine, Aug. 1993, 38-45.

[BDCG89] T.F. Bowen, F.S. Dworak, C.H. Chow, N. Griffeth, G.E. Herman, and Y-J. Lin. The Feature Interaction Problem in Telecommunications Systems. *7th International Conference on Software Engineering for Telecommunication Switching Systems*, 1989, 59-62.

[CGLN94] E. J. Cameron, N. Griffeth, Y. Lin, M. E. Nilson, W. K. Schnure, H. Velthuijsen. A Feature Interaction Benchmark for IN and Beyond. IEEE Communication 31, 3 (1993), 64-69. Also reprinted in [Fits94].

[CRS90] E. Cusack, S. Rudkin, and C. Smith. An Object-Oriented Interpretation of LOTOS. In: S.T. Vuong (Ed.) *Formal Description Techniques, II.* North-Holland, 1990, 211-226.

[DV92] J.M. Duran and J. Visser. International Standards for Intelligent Networks. IEEE Communications Magazine, Feb. 1992, 34-42.

[Faci95] M. Faci. Detecting Feature Interactions in Telecommunications Systems Designs. Ph. D. Thesis, University of Ottawa, 1995. Available by ftp on lotos.csi.uottawa.ca

[FaLo96] M. Faci and L. Logrippo. An Algebraic Framework for the Feature Interaction Problem. Proc. of the 3rd AMAST Workshop on Real-Time Systems, Salt Lake City, 1996, 280-294.

[FaLS90] M. Faci, L. Logrippo and B. Stepien. Formal Specifications of Telephone Systems in LOTOS. In E. Brinksma, G. Scollo, and C. Vissers, eds., *Protocol Specification, Testing, and Verification IX.* North-Holland, 1990.

[FaLS91] M. Faci, L. Logrippo and B. Stepien. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach. Computer Networks and ISDN Systems, 21, North Holland, 1991, 52-67.

[FL94] M. Faci and L. Logrippo. Specifying Features and Analysing their Interactions in a LOTOS Environment. *Second International Workshop on Feature Interactions in Telecommunications Software Systems*, eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994, 136-151.

[Fits94] *Second International Workshop on Feature Interactions in Telecommunications Software Systems*. Eds. L. G. Bouma and H. Velthuijsen, IOS Press 1994.

[FMN95] A. Fantechi, B. Mekhanet, E. Najm, P. Cunha, J. Queiroz. Correctness Preserving Transformations for the Late Phases of Software Development. In: T. Bolognesi, J. v.d. Lagemaat, and C. Vissers. *LOTOSPHERE: Software Development with LOTOS.* Kluwer, 1995.

[Hoar85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[Kam96] J. Kamoun. Formal Specification and Feature Interaction Detection in the Intelligent Network. Master thesis, University of Ottawa, 1996 . Available by ftp on lotos.csi.uottawa.ca.

[LFH92] L. Logrippo, M. Faci and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples, Computer Networks & ISDN Systems, Vol. 23, No. 5, 1992, 325-342.Errata in 25 (1992) 99-100.

[Mart88] R.L. Martin. Future Telecommunications Services. IEEE Global Telecommunications Conference 1988, 721-725.

[Miln89] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[Rud92] S. Rudkin. Inheritance in LOTOS. In: K.R. Parker and G.A. Rose (Eds.) *Formal Description Techniques, IV.* North-Holland, 1992, 409-424.

[SL93] Stepien, B., and Logrippo, L. Status-Oriented Telephone Service Specification. In: T.Rus and C.Rattray (eds.) *Theories and Experiences for Real-Time System Development*. AMAST Series in Computing, Vol. 2, World Scientific, 1994, 265-

286.

[SL94] B. Stepien and L.Logrippo. Representing and Verifying Intentions in Telephony Features Using Abstract Data Types. In: *Third International Workshop on Feature Interactions in Telecommunications Software Systems*, eds. K.E.Cheng and T.Ohta, IOS Press 1994, 136-151.

[SL95] B. Stepien and L.Logrippo. Feature interaction detection by using backward reasoning with LOTOS. In: S.T. Vuong and S.T. Chanson. *Protocol Specification, Testing and Verification XIV*. Chapman & Hall, 1995, 71-86.

[Th94] J. Thorner. *Intelligent Networks*. Artech House, 1994.

[TrVs95] K. J. Turner, and M. van Sinderen. LOTOS Specification Style for OSI. In: T. Bolognesi, J. v.d. Lagemaat, and C. Vissers. *LOTOSPHERE: Software Development with LOTOS*. Kluwer, 1995.

[VSVB91] C. A. Vissers, G. Scollo, M. van Sinderen, and E. Brinksma. Specification Styles in Distributed Systems Design and Verification. Theoretical Computer Science 89 (1991) 179-206.