Formal Specification of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach

Mohammed Faci, Luigi Logrippo, Bernard Stepien

University of Ottawa
Protocols Research Group
Department of Computer Science
Ottawa, Ontario, Canada K1N 6N5
E-mail: lmlsl@uottawa.bitnet

Abstract. The LOTOS constraint-oriented style allows the design of well-structured, implementation-independent specifications of distributed systems. As an example, we provide a small, didactically-oriented specification of a simple telephone service. The design of the specification is based on three types of constraints, i.e. global constraints, end-to-end constraints and local constraints. The structure of the specification, as well as its design method, are described in some detail. We conclude with a discussion of the specification debugging method.

Keywords: Telephone Systems, Formal Specification, LOTOS

1. Introduction and Motivation

LOTOS [ISO1][VVD], the Language of Temporal Ordering Specifications, has been conceived in the framework of OSI standardization as a tool for the formal description of OSI services and protocols. In fact, most examples of LOTOS specifications found in the literature today relate to this type of application. We claim, however, that the concepts of LOTOS are general enough to make the language useful for a wide range of applications.

In this paper we show that LOTOS is appropriate for telephone systems (TS) specification. We feel that this is practically important, given the greater market importance of telephone networks with respect to packet-switching networks, and the fact that, in the long range, these two types of networks are expected to become integrated. In this context, it is also important that the

---

A preliminary version of this paper was presented at the IX IFIP WG 6.1 Symposium on Protocol Specification, Testing, and Verification (June 1989).

CCITT has been studying LOTOS as a more advanced technique than its Specification and Description Language (SDL) [SDL], which today is the most commonly used language for the formal specification of standard telephony procedures.

Over the years, our research group has produced a number of TS specifications using different styles. The example system chosen for this paper is a simple Plain Old Telephone System (POTS). It covers all features of POTS, but only from a service point of view. The intention here is to separate the service description from the details of internal functioning of the POTS system. In other words, the specification describes only the end-to-end service provided to users, in an abstract, concise and implementation-independent fashion. Most notably, no attempt was made to describe the switching function, for which we refer to [QA]. The size of our specification is kept intentionally small, to make it possible to explain it in the paper. Since the concept of POTS service is generally well understood, the example can be used as an introduction to LOTOS. A somewhat different LOTOS specification of the POTS service can be found in [FLS].

The subject of formally specifying TSs is attracting increasing attention from the research community. Zave [Z] has written a well-known paper presenting a formalism for describing distributed systems, especially protocols and TSs. Jensen [J] discusses POTS specifications using the formalism of colored Petri nets. Biebow and Hagelstein [BH] have shown how such specifications can be written by using algebraic abstract data type formalism. Tvrdy published an early paper on the application of LOTOS to telephone system specification in [TV].

2. Specification of Telephone Systems in LOTOS

A predictable observation that we made at the beginning of our work was that several concepts originally developed for the description of data communications protocols and services are also very appropriate for the description of TSs. One such concept is that of service provider [VL]. In a service specification, only the external behavior of the system is captured, that is, describing *what* does the system do for the user and not *how* does it do it.

There are, however, important differences between TS service specifications and OSI service specifications, especially of the type that have been specified in LOTOS so far. For example, more than OSI service specifications, TS specifications are dominated by the connection and disconnection phases, where the connection is established or released and a number of housekeeping functions is performed, such as the functions of maintaining the list of busy numbers, of checking whether the telephone is in use or not, and of generating busy signals. The “data exchange” phase has a minimal importance, and in fact it is even impossible to portray faithfully, because in POTS data is not discrete.

Another element that cannot be dealt with completely in LOTOS at the present stage is timing aspects. For example, 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", although a similar result might be obtained by specifying disconnection after an unspecified amount of time. Some research has been done in this area, and some proposals exist [QF][BR][VTZ]; however in this paper we wanted to use only standard LOTOS. We are looking forward to progress in this important issue. However, from a pragmatic point of view it should be recognized that timed LOTOS will be a more complex language than the existing LOTOS, and that for many purposes it is not necessary to formally specify exact time delays.

3. Issues of Specification Style for TS Specifications

Vissers, Scollo and van Sinderen [VSV] identify four main styles for writing LOTOS specifications, viz. the monolithic style, the state-oriented style, the resource-oriented style and the constraint-oriented style. Each one of these styles has its own uses in TS specifications, and they can be mixed (although of course arbitrary mixture can be counterproductive).

The monolithic style gives explicitly all possible sequences of actions allowed by a specification. The main operator is *choice* [], and the specification is shown as a tree of choices. Therefore, this style is useful for debugging the specification and generating test sequences. A well-known basic result in LOTOS theory is the expansion theorem [M][ISO1], by which any LOTOS specification can be transformed into a (possibly infinite!) monolithic one. Although expansion may not terminate, it can yield finite initial subtrees of an infinite monolithic specification equivalent to the given one. Such subtrees can be used for the purposes mentioned above. This "partial expansion" process can be carried out by an interpreter (see the "symbolic execution trees" of [GHL][GL]) or by specialized tools [QPF]. Symbolic execution trees were used in the design process of our specification.

In the state-oriented style, explicit system states are identified, e.g. 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 increased readability of the specification in cases where the informal specification uses the state concept, as is quite common for telephone devices. It may also lead to LOTOS specifications which can be implemented directly. In this work, we avoided the state-oriented style because we wanted 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.

If it is desired to produce an implementation of the specification presented below, it would be appropriate to translate the specification into some combination of resource-oriented and/or state-oriented style.

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, therefore, is useful for implementation-independent specification [T]. 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 predicates as well. The two main operators used to express process (i.e. constraint) composition in our specification are ||| and ||, i.e. the *interleave* and *synchronization* operators. That is, *(A|||B)||C* means that *A* and *B* can freely interleave, while synchronizing with *C*. For example:

(a; b; stop ||| c; d; stop) || (a; d; stop ||| c; b; stop)

is equivalent to (in monolithic style)

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

where the latter expression is the *expansion* of the former.

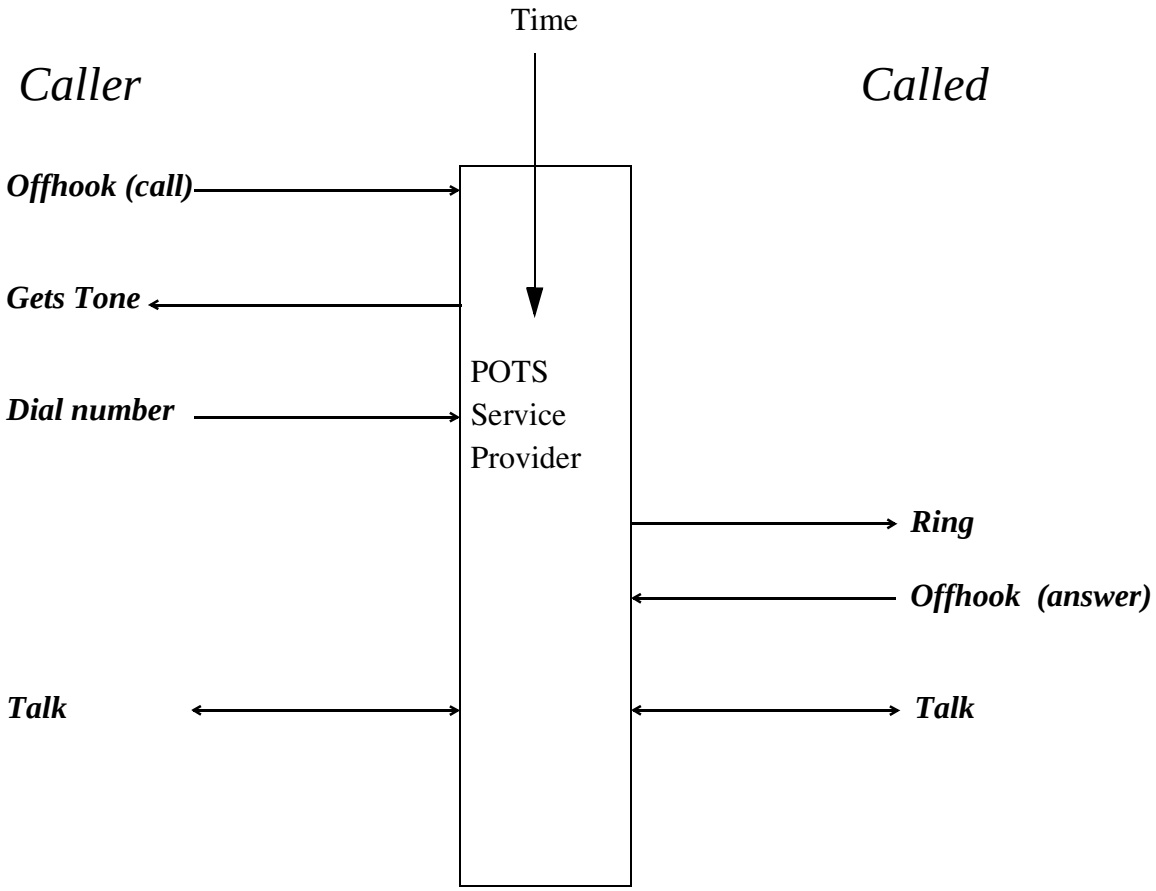
4. The Specification

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. The complete LOTOS specification is presented later in Appendix A.

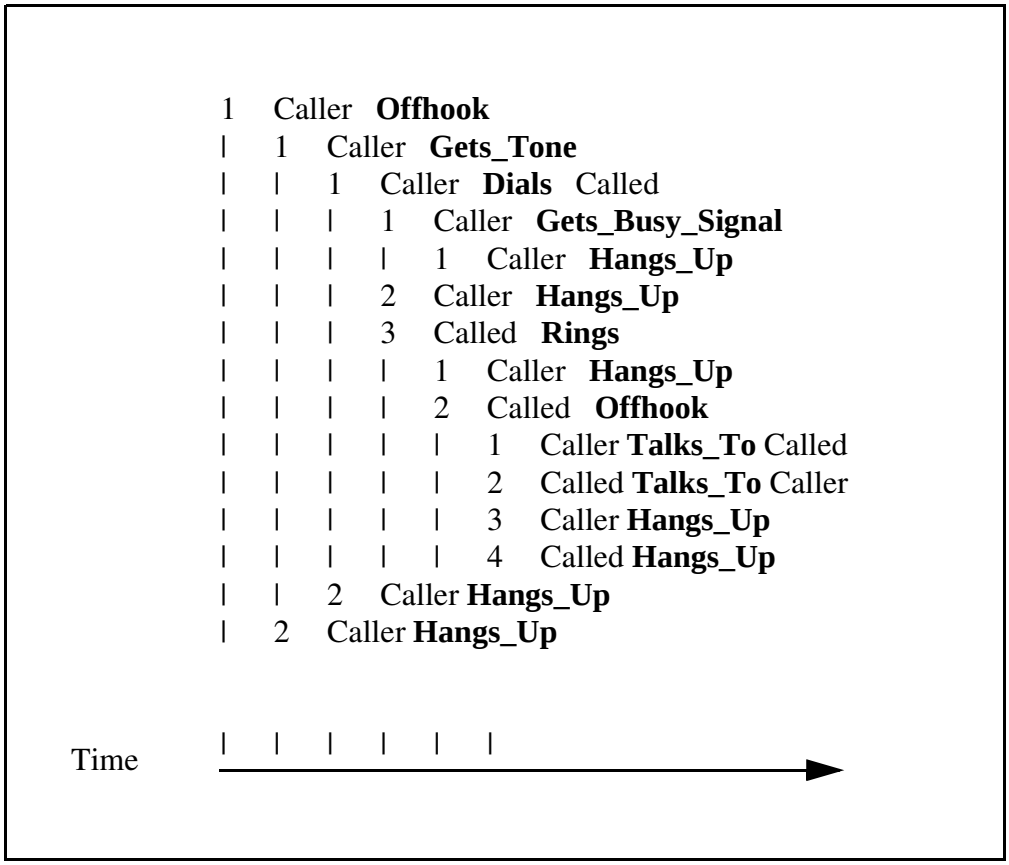
4.1 The Informal Description

First, we introduce the terminology that will be used throughout this paper. *System r*efers to the local telephone branch exchange. A *telephone c*onsists of a *Caller* side and *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*. Each extension has a *handset* which the user picks up before dialing the number of another user, or answering a ring. This is the case where we may find one telephone extension in the kitchen, another one in the bedroom, and possibly a third one in the garden. Only one extension may be used at a time. Intuitively, telephone users communicate with each other through a black box (POTS). They interact with POTS by using a well defined set of primitives.

The following diagram reflects a scenario where the caller establishes a connection with a non-busy called. This is the kind of diagram that can be found to be useful as a first step in the design process. The diagram is not complete, at this stage, because *hang ups* and *busy signals* are not shown. They will be added in the next design stage.



For further refinement of this diagram we add more information, resulting in a timing diagram that shows all possible interactions, from a user's point of view. Only two users are considered. Alternatives are shown at the same level of indentation.



Our specification allows for an arbitrary number of users to access the TS and communicate with each other. The TS 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 dialling the number, the telephone network checks if the *Called* number is free, and if so, a *Ring* signal is sent to the second telephone. Otherwise, a *Busy_Signal* 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 are 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 telephone free to make or receive other calls. The second telephone remains busy until the user hangs up.

4.2 The Structure of the 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, we were able to identify three types of constraints:

(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.* 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, they are composed by the interleaving operator |||. Another possible choice at this level is to specify only one telephone process, provided with a parameter indicating a *Caller* or a *Called* role for each instantiation [ISO2].

(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, w*e 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.

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

The notions of local, end-to-end and global constraints are well-known in the area of service specifications [BO][ISO2][VSV]. As in [ISO2], the top level architecture of the specification is obtained by composing in parallel the processes representing the constraints.

The top levels of the LOTOS specification are shown in Fig. 1. A graphical representation is shown in Fig. 2. Intuitively, interleaved processes are drawn on top of each other, such as *Caller* and *Called*; parallel processes are drawn next to each other, such as *Users_List_Handler* and *Establish_Connections*; and subprocesses are drawn inside the processes by which they are invoked. Dashed lines represent parentheses.

```
84  behaviour
85    (
86      Establish_Connections[S_User]
87        ||
88      Users_List_Handler [S_User](empty, {} of DecSet)
89    )
90
91  where
92
93    process Establish_Connections[S_User]:noexit:=
94    (
95        Single_Connection[S_User]
96         |||
97        i; Establish_Connections[S_User]
98    )
99    where
100
101     process Single_Connection[S_User]:noexit:=
102     (
103       ( Caller[S_User]
104           |||
105         Called [S_User]
106       )
107        ||
108         Controller[S_User]
109     )
110     where ...
```

Figure 1.
The top levels of the specification

The top-level behavior is composed of two processes, *Establish_Connections* and *Users_List_Handler*. These two processes synchronize through gate *S_User*. Stated informally, we want to create as many connections as desired provided that neither the calling nor the called

number is already in use. *Users_List_Handler*, which we will describe later, enforces global constraints by keeping track of free and busy numbers and synchronizing with *Establish_Connections* to exchange values.

Process *Establish_Connections* is composed of two processes: *Single_Connection* interleaved with the process *Establish_Connections* itself. This creates the desired effect of being able to have an arbitrary number of connections existing simultaneously. Note the action ***i*** (line 97 in the specification) before the recursive call to *Establish_Connections*. This can be taken to mean that the creation of a new connection follows internal actions by the system, such as allocation of necessary resources (more technically, it should be considered that if no internal action was specified at this point, the recursion would be unguarded [M], which would make the specification impossible to execute on a simulator). The process *Single_Connection* is viewed as the composition of three processes: *Caller*, *Called* and *Controller*. The conceptual notion of modeling the call initiator (*Caller*) side and the *Called* side by two interleaved processes is quite natural; it reflects the distributed nature of the architecture, in that local constraints apply to separate portions of behavior. *Caller* (lines 126 to 141) and *Called* (lines 143 to 152) exchange information by synchronization with the *Controller* (line 154 to 191).

It would be possible to specify an upper bound on the number of possible simultaneous connections, for example by using an additional *counter* and appropriate guards.

Specialists involved in implementation and simulation of TS might at first be taken aback by this abstract structure. We should emphasize again that our primary objective is to produce a clear and concise specification of the service provided, as made possible by the characteristics of the specification language, and no attempt is made to reflect a possible implementation architecture. A different specification style will have to be used for that purpose. And, the resulting specification will be larger and less clear than the one provided here, one of the reasons being that it will have to be dependent on a specific implementation architecture.

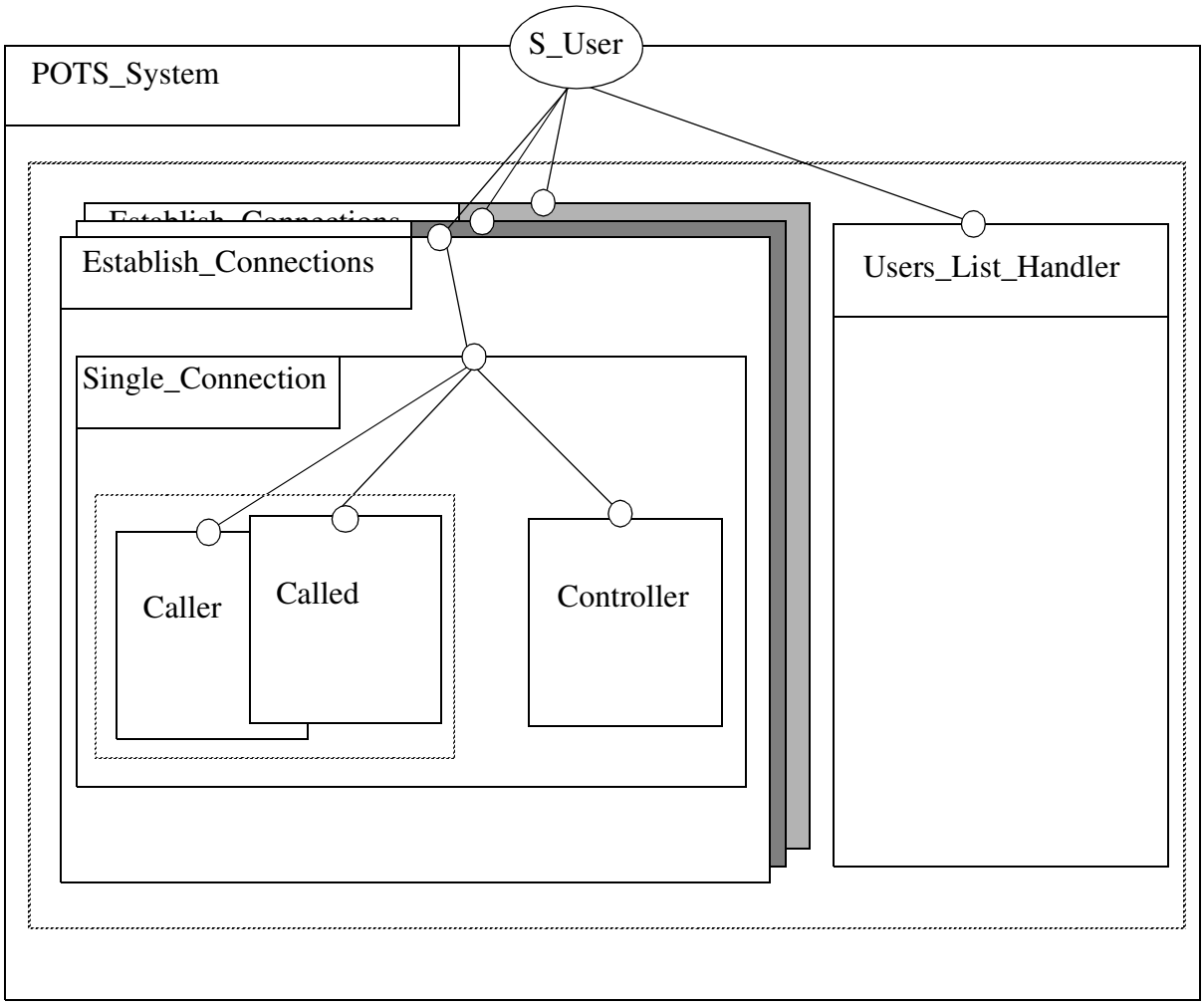


Figure 2. A graphical representation of the specification's top levels

4.3 Descriptions of Processes

In this section we first give the details of the four most important processes which make up the specification. As mentioned earlier, these are *Caller, Called, Controller,* and *Users_List_Handler.* And then we present the abstract data types used to maintain the system users. It is important for the reader to keep in mind that every action, except the ***i*** at line 97, is a multiway rendezvous, on the gate *S_User*, of three behaviors:

(*Caller* ||| *Called*), *Controller* and *Users_List_Handler*.

4.3.1 Establish_Connections

The global constraint is enforced by the composition of two processes, *Users_List_Handler* and *Establish_Connections*, shown in the top level of the specification's structure (lines 85 to 89). *Users_List_Handler* has exclusive use of the users' phone numbers. It interacts with *Establish_Connections* and its subprocesses *(Caller, Called* and *Controller)* for synchronizing on the appropriate events.

4.3.2 Caller, Called and Controller

Local constraints are expressed as a sequence of events that must take place locally, either at the *Caller* side or the *Called* side. The *Caller* (lines 126 to 141) insures that the sequence *Offhook (*To Call*), Gets_Tone, Dials, Talks_To (*or *Busy) i*s executed in that order for each instance of the *Caller*. For example, the *Caller* cannot dial a number before the tone is received. In addition, the event *Hangs_Up*, line 121, may disrupt anywhere after *Offhook* has occurred. At the other end of the connection, *Called* (lines 143 to 152) is responsible for enforcing its local constraints consisting of the sequence *Rings, Offhook (*To Answer*)* and *Talks_To* (line 114). Again, *Talks_To* may only be offered after *Offhook (*To Answer) has occurred. The role of the *Controller* is to enforce the end-to-end constraints. The *Controller (*lines 154 to 191) combines the independent sequences of the *Caller* and the *Called* into a single ordered sequence of events: *Offhook (*To Call*), Gets_Tone, Dials, Rings (*or *Busy), Offhook (*To Answer*)* and *Talks_To.* Similar to the previous case, a *Hangs_Up* event may occur anytime after *Offhook* from the *Caller* and anytime after *Offhook* from the *Called*.

4.3.3 Users_List_Handler

The process *Users_List_Handler* (lines 195 to 229) manipulates the data structures of the specification. It synchronizes with *Establish_Connections* in order to update the list of busy numbers. The mechanics of updates are best described with respect to the data structure itself, which is the subject of the next section.

4.3.4 The Data Structures: Engaged_Set and *Busy_Set*

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 no 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 a caller to receive a *Busy_Signal*, we must check whether the called number is busy or not. Since the interaction *S_User? Caller: DecDigit ! Busy_Signal*, at line 221, does not contain the number of the called, we must keep this information inside the process *Users_List_Handler*, by means of a set of pairs.

So, our first data structure *Engaged_Set* (or *E* for short, line 195) of sort *Set_Of_Pairs* (line 32), is a set of pairs which has the form: {(*X, Y*)| *X* in {*Caller, Called*} and *Y* in {*Called, None*} and *(X, Y)* is different from *(Called, Called)}*. In other words, the first element of a pair in the set is a number which has executed the *Offhook* event (line 197). This could have occurred only if the number was not already the first element of a pair in *Engaged_Set* (line 198). If the number is a caller, then the pair (*Caller, None*) is added to the set (line 199). If it is a called, then the pair (*Called, None*) is added to the set (again, line 199). The difference between these two pairs is that the first one will be modified, to become (*Caller, Called*), when the *Dials* event is executed (lines 204 to 206), while the second pair remains, in the set, in that form until it is removed as a result of the *Called* executing the *Hangs_Up* event (lines 212 to 219).

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 *Busy_Set* (or *B* for short, line 195) of type *Dec_Set* (line 64), which is a set of phone numbers and has the form: {*X*| *X* in {*Called*}}. So, if the *Called* is not busy, the *Rings* event is executed and the associated phone number is added to *Busy_Set* (lines 208 to 210). However, if the called number is busy then the caller must receive a busy signal (lines 221 to 224). Again, a called number is busy if it has executed the *Rings* event, as just stated, (it is then in *Busy_Set*) or if it is the first element of a pair in *Engaged_Set.*

Finally, there is the *Hangs_Up* event (line 212), for which several cases must be distinguished.

First, if the event is associated with a called number (line 149) then both the *Rings* event and the *Offhook* (To Answer) event must have been executed. Consequently, the called number must appear both in *Engaged_Set* as a pair (*Called, None)* and in the *Busy_Set* as *Called*. The called number is removed from both sets (lines 213 to 214).

Otherwise, if the event is associated with a caller number, we must distinguish the cases where the caller may hang up

(1) Before the *Dials* event is executed. Then the called number is not identified yet, and therefore removing the pair (*Caller, None)* from the *Engaged_Set* is sufficient (lines 216 to 217: note that the *Remove* of line 218 will have no effect in this case because *None* is not in *Busy_Set*).
(2) After the *Dials* event is executed, but before the *Rings* event is executed. This case is similar to case (1), except that the called is already identified and removing the pair (*Caller, Called*) from the *Engaged_Set* is sufficient as well (lines 216 to 217).
(3) After the *Rings* event is executed but before the *Offhook* (To Answer) event is executed. This means that the *Called* number associated with the *Caller* number which has hung up is already in the *Busy_Set* and it is not sufficient to remove the pair (*Caller, Called)* from the *Engaged_Set*, but we must also remove the *Called* from the *Busy_Set* (line 218 takes effect).
(4) After the *Offhook* (To Answer) event is executed. This is similar to case (3) except that the *Called* number appears three times in the two sets: as (*Caller, Called)* and *(Called, None)* in the *Engaged_Set* and as a single element in the *Busy_Set.* Therefore, when *(Caller, Called)* is removed from the *Engaged_Set* and *Called* is removed from *Busy_Set* (lines 217 to 218), there is still an occurrence of *(Called, None)* in *Engaged_Set,* which guarantees that the *Called* is still busy.

Note that some data type operators are overloaded. To help the reader, we have capitalized the operators on the set *B*, namely *Insert*, *Remove*, and *IsIn*, whose definitions are to be found in the standard library.

5. Debugging the Specification

LOTOS is based on formal semantics, by which specifications could be (in principle) proven correct according to certain criteria. Unfortunately, however, the proof techniques available today are of limited power and do not allow verification of specifications of this size. Debugging is the other option.

LOTOS specifications, if written in an executable style, can be simulated by means of an interpreter [GHL]. This allows debugging the specification to increase the designer's confidence that the specification reflects the requirements. The University of Ottawa LOTOS interpreter allows the designer to execute a specification in two ways: step-by-step and composing test processes in parallel with the specification.

5.1 Step-by-step Execution

In step-by-step execution mode the user can debug a specification by executing it one action at a time. At each step, the interpreter presents the user with the list of all possible next actions. Users are responsible for choosing the next action and providing appropriate data values. When using this execution mode, one proceeds in a bottom-up fashion. In other words, each leaf

process is debugged separately, their parent processes are then debugged, and so on until reaching the root, which is the specification itself. Space requirements do not allow us to show the execution sequences of the specification. Exhaustive debugging was of course impossible, however many important paths were tested. Note that test sequences are chosen according to our intuitive understanding of how the specification is expected to behave.

5.2 Test Processes

Once the most obvious errors are removed, a more efficient way to detect errors in a specification is to compose a non-branching test process in parallel with it, and then obtain the execution tree of the resulting specification [GHL]. If the execution reaches the last action in the test process, then the specification accepts the sequence of the test process. Appendix B shows one such test process.

```
behaviour
    (
        Establish_Connections ...
      ||
        Users_List_Handler ....
    )
    ||
    Test_Process
where
    process Establish_Connections ....
    process Users_List_Handler ...
    process Test_Process ...
```

Note that the testing process cannot be a nondeterministic process as it should be by LOTOS theory [BR1], because our interpreter is deterministic. Therefore the internal action that guards the starting of a new connection (line 97) has been replaced by an additional gate *Relay*.

6. Conclusions

We have shown that LOTOS is appropriate for specifying the observable behavior of TSs. Of course today's TS are much more complex than the one described in this paper, however features such as Call transfer, Hold, Conference call, Call forward, etc. can be treated by extending the basic mechanism described here. In fact, this extension is the subject of our current research.

By writing and debugging LOTOS specifications of such systems during the design phase, TS designers can give precise descriptions and validate their designs before the implementation

stage. Several types of specifications may exist, some having the goal of describing behavior only, and others having the goal of describing implementation architecture as well. Implementors are then presented with a validated and precise specification of the system's behavior, which can contribute greatly to the quality of the final product. In the final steps, the behavior of the implementation can be compared with the behavior of the specification, by using formal or informal testing methods. The LOTOS methodology will help in this respect as well [GL].

Acknowledgments. Funding sources for our work include the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario (Design of Validation Environments project), Bell-Northern Research, and the Canadian Department of Communications. We are indebted to Raymond Aubin and Rezki Boumezbeur for preliminary work leading to this paper. Also, we are grateful to the four referees whose comments led to significant improvements in the style and content of this paper. Last, but not least, many thanks to Jacques Sincennes for continuous technical assistance and useful discussions.

Appendix A: The LOTOS specification

In this appendix we show the complete POTS specification in LOTOS.

```
1   specification POTS_System [S_User]:  noexit
2
3   library Set, Boolean, DecDigit, NaturalNumber endlib
4
5   type DigitPair is DecDigit, Boolean
6
7     sorts DigitPair
8     opns
9        None: -> DecDigit
10       None: -> Nat
11       Pair: DecDigit,DecDigit -> DigitPair
12       _eq_: DigitPair, DigitPair -> Bool
13
14    eqns
15
16       forall f1,f2,l1,l2:DecDigit, f3:Nat
17
18       ofsort Bool
19          f1 eq None     = false;
20          None     eq f1 = false;
21          Pair(f1,l1) eq Pair(f2,l2) = (f1 eq f2) and (l1 eq l2);
22
23       ofsort Bool
24          f3 eq None     = false;
25          None     eq f3 = false;
26
27       ofsort Nat
28          NatNum(None) = None
29  endtype
30
```

```
type Set_Of_Pairs  is DigitPair
   sorts Set_Of_Pairs
   opns
      empty: -> Set_Of_Pairs
      add:   DigitPair,Set_Of_Pairs  -> Set_Of_Pairs
      remove:   DigitPair,Set_Of_Pairs  -> Set_Of_Pairs
      isin: DigitPair,Set_Of_Pairs  -> Bool
      Second_Element:DecDigit,Set_Of_Pairs  -> DecDigit
      notin: DigitPair,Set_Of_Pairs -> Bool

   eqns
      forall x,y:DigitPair, s:Set_Of_Pairs, f1,f2,l1,l2:DecDigit

      ofsort Set_Of_Pairs
        add(x,add(x,s)) = add(x,s);
        remove(x,add(x,s)) = s;
        not(x eq y) =>
                        remove(x,add(y,s)) = add(y,remove(x,s));

      ofsort Bool
        isin(x,empty) = false;
        isin(x,add(y,s)) = (x eq y) or (isin(x,s));
        isin(x,remove(x,s)) = false;
        not(x eq y) =>
                isin(x,remove(y,s)) = isin(x,s);
        notin(x,s) = not(isin(x,s));

      ofsort DecDigit
        Second_Element (f1,empty) = None   ;
        Second_Element (f1,add(Pair(f1,l1),s)) = l1;
        f1 ne f2 => Second_Element (f1,add(Pair(f2,l2),s)) = Second_Element (f1,s);
endtype

type DecSet is Set
   actualizedby DecDigit
   using
   sortnames  DecDigit  for Element
              Bool      for FBool
              DecSet    for Set
endtype
```

```
71
72  type signal is Boolean
73      sorts  Signal
74      opns
75          Offhook,
76          Hangs_Up,
77          Talks_To,
78          Rings,
79          Dials,
80          Gets_Tone,
81          Busy_Signal    :-> Signal
82  endtype
83
84  behaviour
85      (
86       Establish_Connections[S_User]
87         ||
88       Users_List_Handler [S_User](empty, {} of DecSet)
89      )
90
91  where
92
93    process Establish_Connections[S_User]:noexit:=
94      (
95          Single_Connection[S_User]
96           |||
97          i ; Establish_Connections[S_User]
98      )
99      where
100
101       process Single_Connection[S_User]:noexit:=
102       (
103          ( Caller[S_User]
104            |||
105            Called [S_User]
106          )
107         ||
108            Controller[S_User]
109       )
110       where
```

```
111
112         process User_Talks [G_User] (This_Side : DecDigit) :noexit:=
113         (
114           G_User ! This_Side ! Talks_To ? That_Side : DecDigit;
115           User_Talks [G_User](This_Side)
116         )
117         endproc (* User_Talks *)
118
119         process User_Hang_Up [Any_User](User : DecDigit) : noexit :=
120         (
121           Any_User ! User ! Hangs_Up;
122           stop
123         )
124         endproc (* User_Hang_Up *)
125
126         process Caller[S_User]:noexit:=
127             S_User  ?Caller:DecDigit  !Offhook;
128             ( (
129                 S_User ! Caller ! Gets_Tone;
130                 S_User ! Caller ! Dials ? Called: DecDigit;
131                 (
132                   S_User  !Caller     ! Busy_Signal;
133                   stop
134                  []
135                   User_Talks [S_User](Caller)
136                 )
137               )
138               [>
139                 User_Hang_Up[S_User](Caller)
140             )
141         endproc (* Caller *)
142
143         process Called [S_User] : noexit:=
144         (
145           S_User  ?Called:DecDigit  !Rings;
146           S_User ! Called     ! Offhook;
147           (
148               User_Talks [S_User](Called)
149             [> User_Hang_Up [S_User](Called)
150           )
151         )
152         endproc (* Called *)
153
```

```
154         process Controller[S_User]: noexit:=
155         (
156           S_User  ?Caller:DecDigit  !Offhook;
157           ( (
158             S_User ! Caller ! Gets_Tone;
159             S_User ! Caller ! Dials ? Called: DecDigit ;
160             (
161               S_User  !Called  ! Rings;
162               S_User ! Called     ! Offhook;
163               Talk_To_Both[S_User](Caller, Called)
164               []
165               S_User  !Caller     !Busy_Signal;
166               stop
167             )
168            )
169            [> Controller_Hang_Up[S_User]
170           )
171         )
172         where
173
174           process Talk_To_Both [S_User](Caller, Called : DecDigit) :noexit:=
175           (
176              S_User ! Caller !Talks_To !Called;
177              Talk_To_Both [S_User](Caller, Called)
178             []
179              S_User ! Called !Talks_To !Caller;
180              Talk_To_Both [S_User](Caller, Called)
181           )
182           endproc
183
184           process Controller_Hang_Up [Any_User] : noexit :=
185           (
186             Any_User ? User: DecDigit ! Hangs_Up;
187             Controller_Hang_Up [Any_User]
188           )
189           endproc (* Controller_Hang_Up *)
190
191         endproc (* Controller *)
192       endproc (* Single_Connection *)
193     endproc (* Establish_Connections*)
```

```
194
195   process Users_List_Handler [S_User](E : Set_Of_Pairs, B: DecSet ) :noexit:=
196   (
197       S_User  ?User:DecDigit  !Offhook
198                                                [notin (  Pair(User, Second_Element (User, E)), E)];
199       Users_List_Handler [S_User]( add( Pair(User, None ), E), B)
200    []
201        S_User ? Caller: DecDigit ! Gets_Tone;
202        Users_List_Handler [S_User](E, B)
203    []
204        S_User ? Caller: DecDigit ! Dials ? Called: DecDigit ;
205        Users_List_Handler [S_User]
206                                    (add( Pair( Caller, Called), remove( Pair(Caller, None), E)), B)
207    []
208         S_User  ?Called: DecDigit  !Rings
209                     [not(isin( Pair(Called, Second_Element (Called, E)), E) or  ( Called IsIn B))];
210         Users_List_Handler [S_User](E, Insert(Called, B))
211    []
212         S_User   ?User: DecDigit !Hangs_Up;
213         (   [ (User IsIn B)] ->
214                Users_List_Handler [S_User] (remove(Pair (User, None),E), Remove (User, B))
215          []
216             [ (User NotIn B)] ->
217               Users_List_Handler [S_User] (remove(Pair (User,Second_Element (User, E)),E),
218                                              Remove(Second_Element (User, E), B))
219          )
220    []
221          S_User ? Caller: DecDigit ! Busy_Signal
222          [isin( Pair(Second_Element (Caller, E), Second_Element
223          (Second_Element (Caller, E), E)), E)  or ( Second_Element (Caller, E) IsIn B) ];
224          Users_List_Handler [S_User](E, B)
225    []
226           S_User ? Caller: DecDigit !Talks_To ?Called : DecDigit;
227           Users_List_Handler [S_User](E, B)
228   )
229   endproc (* Users_List_Handler *)
230 endspec
```

Appendix B: Test Process Sample

The sample test process given below verifies that a number becomes busy once it has executed the ring action.

```
process Test_Process [S_User, Relay]: noexit :=
(
 S_User !1 !Offhook;
 S_User !1 !Gets_Tone;
 S_User !1 !Dials !2;
 S_User !2  !Rings;
 Relay;
 S_User !3 !Offhook;
 S_User !3 !Gets_Tone;
 S_User !3 !Dials !2;
 S_User !3 !Busy_Signal;
 stop
)
endproc
```

The results of the execution is a unary tree whose root is the first action in the test process and whose leaf is the last action. Predicates have been manually edited and replaced by their evaluated value [true], to enhance readability. Line numbers of synchronized actions are given in square brackets. The first number comes from either process *Caller* or process *Called,* the second from process *Controller,* the third from process *Users_List_Handler* and the last from the process *Test_Process*, which is not shown in the specification given in appendix A. For the Relay action, the first number comes from the specification's behaviour expression, the second from process *Users_List_Handler* and the third from *Test_Process*.

```
S_User !1:DecDigit !Offhook:Signal      [true]      [127,156,197,239]
S_User !1:DecDigit !Gets_Tone:Signal                [129,158,201,240]
S_User !1:DecDigit !Dials:Signal !2:DecDigit        [130,159,204,241]
S_User !2:DecDigit !Rings:Signal        [true]      [145,161,208,242]
Relay                                               [97,231,243]
S_User !3:DecDigit !Offhook:Signal      [true]      [127,156,197,244]
S_User !3:DecDigit !Gets_Tone:Signal                [129,158,201,245]
S_User !3:DecDigit !Dials:Signal !2:DecDigit        [130,159,204,246]
S_User !3:DecDigit !Busy_Signal:Signal  [true]      [132,165,221,247]
```

REFERENCES

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

[BR] Brinksma, E. *On the Design of Extended LOTOS, A Specification Language for Distributed Systems.* Doctoral Dissertation, Universiteit Twente (NL), 1988.

[BR1] Brinksma, E. A Theory for the Derivation of Tests. In: Aggarwal, S., and Sabnani, K., (Eds.) *Protocol Specification, Testing, and Verification, VIII,* North-Holland, 1988, 63-74.

[BH] Biebow, B. and Hagelstein, J. Algebraic Specification of Synchronization and Errors: A Telephonic Example. *Lectures Notes in Computer Science, Vol. 186*, 294-308.

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

[GHL] Guillemot, R., Haj-Hussein, M., and Logrippo, L. Executing Large LOTOS Specifications. In Aggarwal, S., and Sabnani, K., (eds.) *Protocol Specification, Testing and Verification, VIII,* North-Holland, 1988, 399-410.

[GL] Guillemot, R., and Logrippo, L. Derivation of Useful Execution Trees from LOTOS by Using an Interpreter. In: Turner, K. J. (ed.), *Formal Description Techniques,* North-Holland, 1989, 311-325.

[ISO1] International Organization for Standardization, Information Processing Systems, Open Systems Interconnection. IS 8807: *LOTOS: A Formal Description Technique Based on the Temporal Ordering of Observational Behavior* (1988).

[ISO2] ISO/IEC JTC1/SC6 N4870. Formal Description of ISO 8072 (Transport Service) in LOTOS (working draft), 1988.

[J] Jensen, K. Coloured Petri Nets, *Lecture Notes in Computer Science, Vol. 254,* 1987, 248-299

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

[QA] Quemada, J., and Azcorra, A. A Constraint-Oriented Specification of Al's Node. In [VVD], 83-88.

[QF] Quemada, J., and Fernandez, A. Introduction of Quantitative relative Time in LOTOS. In: Rudin, H., and West, C.H. (eds.) *Protocol Specification, Testing, and Verification, VII.* North-Holland, 1987, 105-121.

[QPF] Quemada, J., Pavon, S., and Fernandez, A. Transforming LOTOS Specifications with LOLA - The Parameterized Expansion. In: Turner, K. J. (ed.), *Formal Description Techniques,* North-Holland, 1989, 45-54.

[SDL] CCITT Recommendation Z.100. *Specification and Description Language,* 1988.

[T] Turner, K. Constraint-oriented style in LOTOS. Proceedings BCS Workshop on Formal Methods in Standards, Didcot, UK, April 1988.

[TV] Tvrdy, I. Formal Modelling of Telematics Services Using LOTOS. *Microprocessing and Microprogramming, Vol. 25,* 1989, No. 1-5, 313-317.

[VL] Vissers, C. A., and Logrippo, L. The Importance of the Service Concept in the Design of Data Communications Protocols. In Diaz, M. (ed.) *Protocol Specification, Testing, and Verification, V.* North-Holland, 1986.

[VSV] Vissers, C. A., Scollo, G., and van Sinderen, M. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In: Aggarwal, S., and Sabnani, K., (eds.) *Protocol Specification, Testing and Verification, VIII*, North-Holland, 1988, 189-204

[VTZ] van Hulzen, W., Tilanus, H., and Zuidweg, H. LOTOS Extended with Clocks, to appear in: Vuong, S., (ed.) *Formal Description Techniques*, North-Holland, 1990.

[VVD] van Eijk, P.H.J., Vissers, C. A., and Diaz, M. (eds.). *The Formal Description Technique LOTOS.* North-Holland, 1989.

[Z] Zave, P. The Distributed Alternative to Finite-State-Machine Specifications. *ACM Trans. On Prog. Lang. and Systems, 7*, No. 1, Jan. 1985, 10-36.