Design of Telephony Services In Lotos

by

José Luis Vivas
Design of Telephony Services In Lotos

José Luis Vivas

March 6, 1994

Abstract

The purpose of this thesis is to design and validate a telephony system using the formal description language LOTOS and a series of tools provided by the Lotosphere Integrated Tool Environment, LITE. The starting point is a requirements document containing a series of informal specifications of services common to modern telephony systems.

Special emphasis has been laid on choosing a design that simplifies the integration of additional telephone services in the specification. Most design decisions were thus taken with the purpose of creating an environment that could help different designers to develop distinct functionalities without the need for each one to be acquainted with other existing features in the system. To this end, the design was structured in a fashion that might enhance the extensibility of the system, as well as prevention and detection of incompatibilities between different features. A methodology is thus proposed for dealing with what is commonly known as the feature interaction problem for telephony systems.

1 Introduction

1.1 Motivation

The most common language in use today for specifying telephony systems is SDL [CC87]. However, CCITT has been considering the eventual appropriateness of LOTOS as a more advanced description language.

Currently available tools give support for production of non-LOTOS, technology specific implementations of LOTOS designs. Rapid construction of prototypes gives customers the opportunity to validate the requirements operationally at an early stage of product development, as well as to negotiate with designers in terms they may easily understand.

These facts motivated the subject for this thesis: the formulation and validation in LOTOS of a formal specification of a telephony system, with the purpose of gauging the suitability of LOTOS methodology for this kind of application.

In contrast to earlier implementation of telephony systems in LOTOS, which concentrated mainly on showing the suitability of LOTOS for the specification
of such systems, the present study is an attempt at dealing with the important feature interaction problem. The complexity of the task of introducing new features to these kinds of systems is a well-known fact, and the prevention, detection and resolution of undesirable feature interactions has become an active field of research.

Adding a new feature to a system requires changes in the specification, design and implementation. If rapid addition is desirable, the formal specification should also possess this quality. Nothing will be gained if the formal specification, which is used for validation of the specification, does not itself support rapid addition of new features. In fact, the formal specification may be viewed as a virtual implementation of the system, and may by itself introduce new interactions not contained in the informal specification, in analogy to the real implementation.

A formal specification design supporting rapid integration of new features may also be useful in the detection, classification, and even resolution of many interaction problems that might appear later in the implementation phase. Through a series of refinements of the specification, the formal specification might be of help in preventing the appearance of undesirable interactions introduced in the implementation phase. In this way, the usefulness of the formal specification could be extended to every phase of the design process.

1.2 Related Work

A formalism for describing distributed systems, applied to telephony systems, was presented in [Zav85]. Specifications of basic telephony systems using colored Petri nets was discussed by K. Jensen in [Jen87]. A different formalism for representing telephone systems, based on algebraic abstract data types, was proposed by Biebow and Hagelstein [BH83]. Tvrđy [Tvr89] published an early paper about the specification of telephony systems in LOTOS. A mobile telephone system in LOTOS have been specified by Cam and Vuong [CV:SimCel]. An object-oriented approach to the specification of telecommunication services was presented in [Sim89]. A preliminary version of a formal specification of telephone systems in LOTOS, using the constraint-oriented style, appeared in [FLS89], followed by a new version in [FLS91]. A design methodology for the description in LOTOS of telephone systems with the inclusion of additional services was published by R.Boumezbeur and L.Logrippo [BL91], followed by a Master of Sciences thesis by Boumezbeur [Bou91]. A specification of an ISDN telephone system in LOTOS, together with a semi-automatic generation of executable code from this specification, was presented in [EHM92a] and [EHM92b].

The specification presented in this study has been inspired by the work realized by Boumezber in [Bou91].
1.3 Disposition

A presentation of the stepwise refinement methodology for LOTOS, together with the related notion of specification style, is the subject of Section 2. Section 3 deals with some of the currently available tools for the analysis, simulation, verification and compilation of LOTOS, mainly those included in Lite, the Lotosphere Integrated Tool Environment. In Section 4, the sources of the informal specification are presented. An overall view of the structure of telephony systems is given in Section 5. In Section 6 we discuss the specification of the basic telephony system POTS, Plain Old Telephone System, whereas the specification of the additional services is discussed in Section 7. In Section 8 we discuss some relevant questions concerning time aspects and the disabling operator, and in Section 9 we give an assessment of our experience in working with LOTOS and its methodology. Finally, some general remarks are given in the conclusion, Section 10. An appendix contains a full discussion and documentation of the LOTOS specification.

2 Development, Design Principles and Specification Styles

Development with LOTOS may be carried out in a stepwise fashion, starting from a given requirements document and then going through a sequence of design steps, each one producing a refinement of the system. This is basically the gist of the LOTOSPHERE Design Methodology, conceived to support the development of high quality design [Pav91, QAP92].

In the LOTOSPHERE Design Methodology, the design trajectory starts with the requirements capturing phase, which yields a user requirements document, an informal description of the system, usually in terms of natural language statements. This phase is not directly supported by LOTOS, although it would be possible to include LOTOS specifications as part of the document.

After the requirements capturing phase, a sequence of design steps should follow, each one representing a transformation of the system description based upon a number of design decisions (Figure 1).

Design steps normally consists of either two or three phases: production of the design, assessment of the design, and eventually prototyping. Each step should bring forth a more refined version of the system (Figure 2).

The distinct phases of the design process are considered to be closely related to different programming styles [VShS88, QAP92, VShSB91]. The main ones are the following:

Monolithic style In the monolithic style the parallel operators are disallowed. The two most important constructs used are action prefix and choice. Only observable interactions may occur. This implies that the temporal order
Figure 1: Stepwise Refinement

Figure 2: Design Step
of events is explicitly specified and that no internal events may take place, implementation concerns being completely disregarded.

**Constraint-oriented style** In the constraint-oriented style we also work with observable interactions, but now the temporal ordering will not be given explicitly but implicitly by a conjunction of different constraints. This style supports modularity and may also be helpful for internal structuring. Local constraints are separated from end-to-end constraints. Orthogonality and open-endedness are also supported. The hide construct is disallowed, since only externally observable behaviour should be described. The use of parallel composition operators is extensive. This composition may be interleaved or synchronized. The processes associated with local constraints are usually interleaved or unsynchronized among themselves, whereas the process associated with the end-to-end constraints is synchronized with the others, partially or totally.

**State-oriented style** The state-oriented style allows one to define the internal state space explicitly. Only observable interactions are supported. This style is useful in the final phase of the design development. A description in this style takes the form of a process with parameters representing its state. Its behaviour expression consists of a series of choices guarded by tests associated with the actual state (the parameters of the process). The process will eventually be recursively invoked with new values for the parameters representing the state.

**Resource-oriented style** The resource-oriented style supports both observable and internal interactions. The behaviour is defined by a composition of separate resources in which the internal interactions are hidden. Intensional description with modularity and parallel structures is supported. Each resource may be considered as a separate entity with interfaces to other resources. The design principles presented above are all supported by this style. The usage of the hide construct is enforced.

**Extended automata-oriented style** The extended automata-oriented style is basically a mixture of monolithic and state-oriented styles.

The monolithic and constraint-oriented styles are used mainly in extensional descriptions, where systems are represented in terms of external observational behaviour. The other three styles are suitable for intensional descriptions of systems, which in this case are represented in terms of interacting parts showing their internal organization.

The nature of the relationship between specification styles and design steps is best understood in the light of some general principles that are fundamental for the design methodology of concurrent systems and complex architectures. These principles are known as orthogonality, generality and open-endedness. Orthogonality relates to the principle that independent requirements
should be specified by independent definitions. Generality postulates the use of
general-purpose definitions that can be instantiated, rather than special-purpose
ones. Finally, open-endedness is related to the flexibility of the design, the
possibility of modifying and extending it easily.

3 Tools

The tools that will be considered here are SMILE [CS92], LOLA [PL93] and
TOPO [VMdM+92], all of which are included in the LOTOSPHERE integrated
tool environment, LITE [CS92].

LITE is an integrated environment providing state-of-the-art tool support for
the design process with LOTOS. It provides tools for edition, report generation,
simulation, verification, compilation and testing. The tasks to be accomplished
with the help of these tools are validation, verification and compilation. A
unique objective-oriented user interface governs the coordination of the tools,
which are set up to work according to the user selection of an objective. However,
we have worked only with a subset of these tools, called MiniLITE, which are
the most mature and robust of the tools included in LITE.

The validation task may include simulation, test generation, test execution
and debugging. The symbolic simulator SMILE performs symbolic evaluation
and has an embedded narrowing functionality based upon the narrowing al-
gorithm for abstract data type evaluation [Wol91]. It provides an advanced
interface with several functionalities (Figure 3).

In the main window, both the specification and the simulation tree are vis-
ible (Figure 4). The dynamic behaviour may be requested for any process or
behaviour defined in the specification. After requesting the service of the simu-
lator, we may execute an unfolding of the corresponding behaviour in a stepwise
fashion. The unfolding depth and the narrowing power may be selected by the
user. The resulting tree may be written out to a file.

An important debugging facility offered by the simulator is the display in a
separate window of the information associated with any specified node in the
evaluation tree, which may also be written out to a file. By selecting any event
in the simulation tree we may obtain the behaviour expression corresponding to
the state that results after the event in question has occurred.

Another facility available is the computation of the extended finite state
machine for the desired specification.

An interface to the narrowing engine is given for the analysis of the abstract
data type part of the specification (Figure 5). Its main function is the solution
of goals by means of the narrowing tool. Variables may also be instantiated
here and substituted in the simulation tree.

Another tool available in LITE is LOLA, LOTos LABoratory. It is a transfor-
mational tool based on a tree representation of standard LOTOS specifications.
LOLA accepts full LOTOS language.
General Information

SMILE is a symbolic simulator for LOTOS. It allows you to browse through your specification and symbolically execute any part of it.

It contains a context-sensitive HELP function: pressing the HELP key on any function or area in SMILE will give you more detailed information on that particular function or area. This information can be saved by pressing the 'Dump to File' button. 'Dismiss' will delete the window.

SMILE consists of two main windows, the SMILE window and the ADT window. Both windows are subdivided into three parts:

SMILE window:
- contains the 'specification' window (in which the LOTOS specification is displayed);
- the 'simulation tree' window (which contains the result of executing the specification) and a 'control panel' in which various options can be set.

ADT window:

Figure 3: SMILE info

![SMILE interface]

Figure 4: SMILE interface
Several different expansions are possible. Expansions may be used in testing by synchronization with a test process or a test sequence. LOLA can be used to generate May tests from a reference specification (for details, see [PL93]). The set of May tests obtained in this way may then be composed with the system specification and executed using some test running techniques. LOLA calculates the response of a system specification to must, may, and reject tests. Specifications may also be simulated step by step, and data value expressions may be evaluated.

The third tool mentioned above, TOPO, is a LOTOS to C compiler. TOPO supports full LOTOS with the following exceptions: the functionality of the specification may not be exit; unguarded recursion in process instantiation is not allowed; and value generation is not implemented.

With the help of TOPO we have the possibility of realizing a specification. TOPO supports automatic implementation of the ADT part of a specification as a rewrite system, provides facilities for memory management of the implementation of the resulting C structures, for replacement of parts of the data types by hand-coded implementations, and for the use of partial data types; further, it implements fair behavior for non-deterministic specifications and allows the use of side effects associated with action offers. Facilities are also provided for controlling the delay time of event offers and for externally controlled event offers. Multiway synchronization is supported in the cases where data for value negotiation is explicit, that is, if there is at least one !-offer or
acceptable default or eval annotations (see below). Recursive and dynamic creation of processes are supported as well. Scheduling may be automatic or external.

TOPO expects some compiler information from the user representing implementation decisions. This is achieved by means of annotations, a special class of LOTOS comments that are interpreted by the compiler. An annotation may be a program fragment that is executed whenever the machine makes the transition associated with it. Several fragments may be associated with any event. These will be executed either in parallel or in random order.

4 The Sources of the Informal Specification

4.1 The System Requirements

No written informal specifications were given for the system requirements. We have thus decided to implement a simple standard POTS (Plain Old Telephone System). The set of requirements of a POTS are those commonly known from most real telephone systems. Such systems show little variation, but certain functionalities may differ, e.g. whether a call is released if the called subscriber clears.

4.2 The Services

The informal specifications of the ISDN-services included in this work were taken from a PM produced at ELELEMTL [Ell90]. It is a carefully conceived and well-structured document, produced in a telephone company by people with experience in the field, written in a concise style with special emphasis on the use of clear and unambiguous terms. It is a pure natural language specification, without charts or diagrams.

Ten out of a total of eighteen function specifications were selected for implementation, basically those exhibiting the most relevant properties with regard to synchronization issues. The functions that were left out are mainly those involving only a static checking of certain properties, e.g. Abbreviated Dialling, Outgoing Call Barring and Hotline, as well as those associated with actions performed by the operator or with redirection to recorded announcements, e.g. Don't Disturb Service, Automatic Casual Alarm Service, Absent Subscriber Service and Malicious Call Tracing. Finally, Charging Analysis, involving time considerations outside the scope of LOTOS, was also discarded.

Two types of telephone sets are described in the PM: push-button and rotary dial. The subscriber procedures affecting the use of the telephone sets are presented in detail. However, these procedures have been disregarded in favour of an abstract modelling of the subscribers actions, which have thus been specified.
as indivisible and instantaneous operations, in order to keep the specification as general as possible.

The presentation of each function consists usually of four parts: a definition, a description of the function, the administration procedures and capabilities of the service.

The definition describes shortly the functionality of the service. The description of the function is a more detailed presentation of the function, including the handling of calls and clearing situations, the type of calls related to the service, the handling of exception situations, the category marking of the subscriber and kind of equipment required, the subscriber procedures and charging. The subscriber procedures include procedures for registration, verification, interrogation, erasure, activation, deactivation, etc., in the cases where they apply. The administration procedures specify how the service is provided and withdrawn. The capabilities describes the connection and subscriber types (individual or group number, connection in SSN, etc.).

In our case, the most relevant points are the definition and description of the function, the handling of calls and clearing situations. Requirements related to subscriber category, number zones, capabilities and similar considerations were left out.

The ten services selected may be classified into three distinct groups, according to certain dependency relations among them:

The first group comprises six services, which are not mutually orthogonal (Figure 6): Full Three Party Service with Transfer on Busy, Full Three Party Service, Add-on Conference Service, Hold for Enquiry with Transfer, Hold for Enquiry and Call Waiting. Both Hold for Enquiry with Transfer and Add-on Conference are extensions of Hold for Enquiry, and Full Three Party Service is a combination of both. Full Three Party Service with Transfer is in its turn an extension of Full Three Party Service, and Call Waiting of Hold for Enquiry with Transfer.

The second group contains three services that are orthogonal to those in the first group (Figure 7): Call Diversion, Call Diversion on Busy and Call Diversion on No Reply. The pertinent relation in this case is priority: Call Diversion has priority over both Call Diversion on Busy and Call Diversion on No Reply. This means that a subscriber with e.g. the call transfer function activated will have his incoming calls transferred even if some or both of the other services were activated.

The last group contains a single service, Registered Call, which is orthogonal to all the other services (Figure 8).

The names of these services were taken directly from the PM and are not standard. A description of them is given in appendix A.
Figure 6: Group 1: six services related by extension.

Figure 7: Group 2: three services related by priority.

Figure 8: Group 3: a single service.
5 The Structure of Telephony Systems

In contrast to data and computer communication systems, the main issues in telephone systems, at least at the level of abstraction we are concerned here, are connection and disconnection. In this way, questions involving message format and the procedure rules guaranteeing the consistency of message exchanges, essential for protocol specification, are of no concern for us. In our case, communication takes place among human agents whose behaviour in this aspect is best left unspecified. Their behaviour is relevant for us only in relation to those acts associated with connection establishment, disconnection and service requests, which always imply some form of manipulation of the telephone set. We may thus regard our communication system as a “dataless” system.

A telephony system may be regarded as a system with local and remote or end-to-end requirements, and may be structured as a set of subsystems interacting both among themselves and with the environment. The system environment of the telephone system is composed of the users of the system. Each connection involves at least one user.

Four distinct phases may be distinguished in a connection (Figure 9): initiation, establishment, utilization, and termination or disconnection. In the first phase, the initiation, a user lifts the handset, upon which he normally will dial a number. In the second phase, the establishment phase, a connection attempt, composed mainly of internal events, will take place. Thereafter, the utilization or conversation phase may follow if the connection has been successfully established. Both subscribers may then engage in conversation. The last phase, the termination phase, will be initiated by any of the users after execution of a clearing action. If a second subscriber is associated with the connection, he must also clear.

The first three phases mentioned are sequential, whereas disconnection may take place at any instant. The disconnection phase is thus responsible for a great portion of the complexity of telephone systems.

In POTS, at most two users may be associated with a connection. The possibility for several users to be engaged at the same time in one connection introduces a great deal of complexity into this picture.

The environment is represented by the users of the system, normally the subscribers. From the user's point of view, a telephone system is a black box interacting with its environment (Figure 10). Subscribers interact with the system in the context of connections, through such actions as lifting or hanging the handset, and tangent or rotary dial manipulation. A connection requires at least one user, the initiator of the call.
DISCONNECTION

Figure 9: The Phases of a Connection

Figure 10: Telephony system
6 The POTS specification

6.1 Design Principles

The design principles presented above, namely orthogonality, open-endedness and generality, are relevant for telephony systems in different ways.

Orthogonality This principle postulates that independent architectural requirements should be specified by independent definitions, emphasizing functional independence.

In the case of telephony systems, this question has two facets. The first one relates to locality aspects, which are enforced by the adoption here of the constraint-oriented specification style. The second one is related to the notion of supplementary or additional service. The finite-state machine-oriented style used extensively in the definition of the processes representing local and end-to-end constraints has been motivated by the need to keep apart independent functions in an effort to simplify the integration of new additional services.

Generality This principle expresses the convenience of having generic and parameterized definitions instead of a collection of special-purpose ones. The main advantages of this approach is that it enhances reusability and enforces abstraction.

LOTOS supports parameterized process definitions, which is essential for the specification of telephony systems. The number of connections in the system is arbitrary, as well as the number of users, a feature that cannot be specified without parameterization.

Open Endedness This principle expresses the requirement that designs be maintainable, that is, easy to extend and modify functionally. Open-ended specifications require modular decomposition of a design into small and rearrangeable units that are reusable, small, and orthogonal.

Open-endedness is an important feature in relation to insertion of new additional services or modification of old ones. The relevance of this principle needs scarcely be emphasized. It is nevertheless the hardest to satisfy in such applications as telephony systems.

6.2 The overall design

The adoption of a connection-oriented approach for the design of a telephony system seems natural enough. A telephone system is a dynamic system which at any time is composed of an arbitrary number of active connections. The system interacts with an environment composed of an arbitrary number of users, entities that regard the system as a black box which they may utilize to communicate
with each other. The system may also communicate with system administrators. LOTOS multi-way synchronization allows us to define a central agency responsible for monitoring the system according to whatever information about the state of the system is available at any given time. It may appear that we are enforcing a centralized system. However, as long as data is structured by phone number it would not be hard to decentralize it. For example, in an eventual refinement this information could be associated with a process attached to every user in the system, an extreme case of decentralization.

Four distinct phases may be distinguished in a connection, as explained above.

In the first phase a process is initiated, called the Initiator, which is responsible for the local constraints associated with the subscriber that started the connection.

In the second phase, the establishment phase, a process called the Controller, initiated at the same time as the Initiator and denoting the end-to-end constraints of the connection, will perform a connection attempt. This will give rise to a new process, the Responder, responsible for the local constraints associated with the subscriber that is being called.

The third phase begins if the connection has been successfully established.

In this phase, the two processes representing local constraints, the Initiator and the Responder, become symmetrical. In Figure 11, these are represented by the objects with the label user. Some additional services require that more than two users may participate in a connection.

In the last phase, the termination phase, every process involved in the connection comes to a halt.

The system requirements are enforced by a process called the Exchange. It is responsible for updating the Catalogue, presented below, and for rejecting events that are not compatible with the overall state of the system at any given instant. For example, a ringing tone should not be applied at a subscriber who is already engaged in an established call.

The Initiator and Responder need each two communication lines (gates): one for the external events associated with the user, and the other for the internal events associated with the Controller and the Exchange (signals). A third gate is used for events (tones) conveying information to the user about the state of the connection. These gates are called user, line and tone, respectively.

We have decided to synchronize also the actions of the user with the actions of the Controller and the Exchange. This appears to be a flaw, but without such notions as time and priority, lacking in LOTOS, a correct implementation would become too complex if the Exchange does not get instantaneous information about the actions of the user (see section 8 for a clarification of this point).
6.3 The data types

In order to achieve an open, extendible, modular and general data structure, a strategy towards the data type part of the specification has been chosen that differs from earlier ones [Bou91, EHM92a]. Basically, it is an orthogonalization of the data structures used in these implementations. In order to avoid the necessity of defining new sets for each new function that is included in connection with new services (e.g. a set for busy numbers, a set of pairs of telephone numbers indicating which subscribers are forwarded to which numbers, etc.), a structure ordered by telephone number, not by function, has been adopted. It is called the Catalogue. In this way, the insertion of new parameters in the argument list of such processes as the Exchange, which may be needed in connection with the introduction of new additional services, is no longer necessary. Moreover, a simple interface for manipulating the Catalogue has been made available. In our opinion, this design enhances both modularity and open endedness.

All data types have been defined by equations that can be interpreted as left to right rewrite rules. The rules are (hopefully) both confluent and terminating.

The Catalogue is a general data structure displaying all information needed by the system at any given instant. It consists of a list of Subscribers, which are structures containing information about any registered subscriber.

For each Subscriber there is a corresponding data structure containing the information required by the Exchange to steer the whole system. A Subscriber
is thus a pair of elements: the first one is the identifying key, the phone number, and the second is a list of data structures called Service.

A Service is also a pair whose first element is a label denoting the kind of information contained in the second element, called the ServiceInfo. The structure of the ServiceInfo is left unspecified, and the sole constraint associated with it is that it must belong to a fixed predefined sort. The specifier of a service is supposed to give it the structure that will best serve his needs.

Seven interface operators are available for manipulating the Catalogue: four constructors, two for insertion/removal of subscribers in the Catalogue and two for insertion/removal of services; one selector for retrieving the ServiceInfo associated with any subscriber; and two predicates for assessing the existence of a subscriber in the Catalogue or of a service associated with any registered subscriber.

Catalogue: List of Subscriber

Subscriber: SubscriberEntry(PhoneNumber, ServiceList)
   PhoneNumber: DecString
   ServiceList: List of ServiceSort

ServiceSort: Service(Category, ServiceInfoSort)
   Category: CharSet
   ServiceInfoSort: defined by service implementator

INTERFACE:

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertSubscriber</td>
<td>PhoneNumber, Catalogue</td>
<td>Catalogue</td>
</tr>
<tr>
<td>RemoveSubscriber</td>
<td>PhoneNumber, Catalogue</td>
<td>Catalogue</td>
</tr>
<tr>
<td>InsertService</td>
<td>PhoneNumber, ServiceSort, Catalogue</td>
<td>Catalogue</td>
</tr>
<tr>
<td>RemoveService</td>
<td>PhoneNumber, Category, Catalogue</td>
<td>Catalogue</td>
</tr>
<tr>
<td>IsInCatalogue</td>
<td>PhoneNumber, Catalogue</td>
<td>Bool</td>
</tr>
<tr>
<td>HasService</td>
<td>PhoneNumber, Category, Catalogue</td>
<td>Bool</td>
</tr>
<tr>
<td>GetServiceInfo</td>
<td>PhoneNumber, Category, Catalogue</td>
<td>ServiceInfoSort</td>
</tr>
</tbody>
</table>

As an example, the entry for a subscriber with phone number 11, who is busy and has the phone number diverted to phone number 55 for incoming calls, may look like the following:

SubscriberEntry{11, {Service(diversion, DivertedTo(55)), Service(engaged, NoInfo),{}}}

The specifier of the Call Diversion service may retrieve the number to which phone number 11 is diverted by using the predefined interface selector GetServiceInfo as follows:

GetServiceInfo(11, diversion, Catalogue).

The result will be DivertedTo(55), an element of sort ServiceInfo. A selector may then be defined which will return 55.
6.4 The behaviour

For the extensional description of a telephony system, we have adopted what may be best described as a mixture of two specification styles: the constraint-oriented style and the extended automata oriented style. Nevertheless, features of other specification styles are also present in the specification.

The constraint-oriented style seems natural for this kind of application and at this abstraction level. This style promotes the separation of local and end-to-end constraints and identification of orthogonal functions by separate constraints.

The top level of the behaviour part of the specification consists of two processes, Network and Exchange (Figure 12), composed in parallel, and synchronizing through gates user and line:

The processes Network and Connection are defined as follows:

```
hide line,tone in
  Network[user,line,tone]
  [user,line]]
  Exchange[update,user,line]\{() of Catalogue\}
```

![Diagram](image-url)

Figure 12: Graphical representation of the top levels of the specification

Here we may observe a typical feature of the resource-oriented style: hiding of gates and abstract resources such as the Exchange and, as we shall see below, the Controller, the Initiator and the Responder.
process Network[u,l,t]: noexit:=
    Connection[u,l,t]
    ||
    |; Network[u,l,t]
endproc

process Connection[u,l,t]: noexit:=
    Subscribers[u,l,t]
    [(u,l,t)]
    Controller[u,l,t]
endproc

The recursive definition of the Network allows an arbitrary number of connections to run in parallel. The process Connection is defined in the constraint-oriented style. The processes Subscriber and Controller synchronize fully. The Controller represents the end-to-end constraints of the connection. The process Subscribers is defined as follows:

process Subscribers[u,l,t]: noexit:=
    Initiator[u,l,t]
    ||
    Responder[u,l,t]
endproc

The processes Initiator and Responder are interleaved, since both represent only local constraints.

The Exchange is responsible for the general systems requirements. Its function is to allow or reject the occurrence of certain events according to the state of the system embodied in the Catalogue. It is thus natural to define the Exchange in the state-oriented style, which typically consists of a choice between several guarded or unguarded alternatives and recursive calls with updated parameters (in this case only the Catalogue). As an example we show a simplified version of the Exchange:

process Exchange[update,u,l](Cat:Catalogue): noexit:=
    u ?N:PhoneNumber !HangUp;
    Exchange[update,u,l](GetsFree(N,Cat))
    |
    u ?N:PhoneNumber !OffHook;
    Exchange[update,u,l](GetsBusy(N,Cat))
    |
    u ?AN:PhoneNumber !DialKs ?BN:PhoneNumber; Exchange[update,u,l](Cat)
    |
    l ?N:PhoneNumber !RING [IsInCatalogue(N,Cat) and not(IsBusy(N,Cat))];
    Exchange[update,u,l](GetsBusy(N,Cat))
    |
    l ?N:PhoneNumber !DISC; Exchange[update,u,l](GetsFree(N,Cat))

19
[]
ExchangeS[update,u,l](Cat)
endproc

In contrast, the processes Initiator, Responder, and Controller are implemented in a style that may be depicted as finite-state machine oriented. The reason for this is to simplify the insertion of new choices in connection with the integration of new functions in the system.

An example may be in place. After lifting the handset and receiving a dial tone indication, the Initiator will be in a state described by the following process:

process Dial[u,l,t](AN:PhoneNumb3:er): noexit:=
  u !AN !DialKs ?BN:PhoneNumb3:er; Arin3:ging[u,l,t](AN) []
  u !AN !HangUp; stop []
  DialS[u,l,t](AN)
endproc

At this point, the subscriber that initiated the call may either dial a phone number or hang up. After the occurrence of any of these events, a new process will be invoked, each one denoting a new state for the process Initiator. If the subscriber chooses to hang up, this process is simply stop.

The process ControlDial matches Dial on the part of the Controller:

process ControlDial[u,l,t](AN:PhoneNumb3:er): noexit:=
  u !AN !DialKs ?BN:PhoneNumb3:er; ControlResponder[u,l,t](AN,BN) []
  u !AN !HangUp; stop []
  ControlDialS[u,l,t](AN)
endproc

If the event representing a dial operation takes place, the process Control-Responder is invoked. It synchronizes with the Responder:

process Responder[u,l,t]: noexit:=
  l ?BN:PhoneNumb3:er !COAT; Bringing[u,l,t](BN) []
  ResponderS[u,l,t]
endproc

process ControlResponder[u,l,t](AN,BN:PhoneNumb3:er): noexit:=
  l !BN !COAT; ControlRing3:ing[u,l,t](AN,BN) []
  ControlResponderS[u,l,t](AN,BN)
endproc

We may observe that the last choice in each process is an invocation of a process named after the invoker process, but extended with an S, which stands for Service. Initially, it is only a dummy process, defined in the same way as DialS below:
process DialS[u,l,t](AN:PhoneNumber): noexit:=
stop;
endproc

The purpose of these processes will be explained below.

7 The additional services

The specification of POTS may be extended to integrate additional services. These services may require connections with more than two subscribers participating in it. We decided to allow several local and control processes, that is, processes denoting local and end-to-end constraints, respectively, to run in parallel with the initial ones, the Initiator, Responder and Controller. We may thus describe a connection as a binary structure consisting of two top processes denoting local and end-to-end constraints, respectively. Each of these processes are composed of an arbitrary number of interleaved subprocesses.

Processes denoting local constraints are always associated with a single user, whereas those denoting end-to-end constraints may be associated with any number of subscribers. Direct communication among processes belonging to the same group is impossible. This feature allows new processes to be defined in a modular fashion, without the need to define new gates or include spurious synchronizations between otherwise unrelated processes.

To exemplify this, let's suppose that the service Hold For Enquiry is available. This service implies that subscriber A can put subscriber B in a passive state, and proceed to ring subscriber C. If the new call is completed, subscriber A may engage in conversation with subscriber C and also switch back and forth between B and C, with full secrecy towards the passive subscriber. If A should clear while someone is in the held call state, that is, passive but still on line, A will be rerung and a new connection will be established between A and the subscriber which was in the held call state when A cleared. The third subscriber is released.

This service may be implemented by creating new local processes that run in parallel with the Initiator and Responder, as well as new control processes that do the same with the Controller.

In our solution, upon request of the service, two new local processes are initiated: a new version of the Responder, this time associated with subscriber C, and a specially defined process responsible for reringing subscriber A if it becomes necessary. The initial Responder will still be associated with subscriber B. Corresponding control processes are also necessary.

As an illustration, let's look at the definition of Talking, the local process responsible for the communication phase of the connection:

process Talking[u,l,t](N:PhoneNumber): noexit:=
u IN IVOICE; Talking[u,l,t](N) []
Interruption[u,l,t](N) []
TalkingS[u,l,t](N)
endproc

process TalkingS[u,l,t](AN:PhoneNumber): noexit:=
stop;
endproc

In order to allow a subscriber to perform a service request, we should add new choices to TalkingS:

process TalkingS[u,l,t](N:PhoneNumber): noexit:=
  u IN ?CN:PhoneNumber !HoldK;
    (HoldRequest[u,l,t](N) || HolderHangUp[u,l,t](N)) []
  u IN !SwitK; SwitchRequest[u,l,t](N) []
  u IN !RiseK; ReleaseRequest[u,l,t](N) []
  !N !HOLD; HoldIndication[u,l,t](N) []
  !N !AN:PhoneNumber !SWITCH; SwitchIndication[u,l,t](N) []
  !N !RLSE; ReleaseIndication[u,l,t](N)
endproc

The first three choices in TalkingS denote invocation of the service, switching and releasing, respectively (the last two are for switching back to the subscriber in the held call state while either putting the other subscriber in this state or else releasing him). The last three choices are initiated by internal events associated with the subscriber that will be put in the held call state.

ControlTalkingS is changed accordingly:

process ControlTalkingS[u,l,t](AN,BN:PhoneNumber): noexit:=
  u !AN !BN !HoldK; (ControlHoldRequest[u,l,t](AN,BN)
    || ControlHolderHangUp[u,l,t](AN)) []
  u !BN !AN !HoldK; (ControlHoldRequest[u,l,t](BN,AN)
    || ControlHolderHangUp[u,l,t](BN)) []
  u !AN !SwitK; ControlSwitchRequest[u,l,t](AN,BN) []
  u !AN !RiseK; ControlReleaseRequest[u,l,t](AN,BN)
endproc

The process HoldRequest is eventually split into two new processes: a new version of Responder for the new subscriber, and a process associated with the subscriber in the held call state. ControlHoldRequest will likewise split into two processes. The processes HolderHangUp and ControlHolderHangUp will monitor the invoker of the service for reringing. The service specifier may define new signals to control the operation of these processes.

The resulting configuration is shown schematically in Figure 13.

As a further illustration of these principles, the process DialS, presented above, must also be modified. It must allow the service requester to interrupt
the operation if he does not want to complete it. The connection with the subscriber in the held call state will then be reestablished.

The example shown above illustrates how the integration of new services in the telephony system may take place. The service specifier works directly with the POTS, which may be regarded as the backbone of the system. He need not care about other services that may have been already defined. He will get a fresh version of the POTS with the dummy service processes intact. New data types and processes may be defined. Changes in the original code are done only inside the dummy functions. The sole exception is the process Exchange, which may require some modification in the form of added extra guards. To produce the final specification integrating several separately defined additional services, the data types and processes defined specially for each service are simply inserted in the code. Further, the choices in the service processes defined for each service are simply added, as well as extra guards eventually inserted into Exchange. Dependencies among services may need special treatment.

The method followed for inserting new additional services may also be interpreted as follows: addition of new services may be done by appending new choices to each node in the trees representing the basic processes: the Initiator, the Responder and the Controller. As illustration, let’s see what happens to the process Initiator. Its tree, as specified for the POTS, but slightly simplified, may be seen in figure 14.

The first action in this tree corresponds to a lifting of the handset by a subscriber. An internal event follows, indicating that the subscriber may receive a dial tone (line $Switch$), or else that the phone is out of service. In the latter case, a tone is sent indicating out of service, tone $OutOfService$, upon which the subscriber must hang up (user $HangUp$). In the former case, a dial tone
Figure 14: The Initiator's process tree in POTS.
Figure 15: The Initiator's process tree with the Call Waiting service.
is sent, tone !Dial. The subscriber may then choose either to clear or to dial a phone number (user !Dial). At this point there are three possibilities: i) the called number is out of service (tone !BoutOfService), and the subscriber must clear; ii) the called number is busy (tone !Busy), and the subscriber must again clear; iii) the called number is ready and ringing (tone !Ring). In the latter case, the subscriber may also choose to clear before getting an answer. If he gets an answer, the connection will be established (line !Connection) and conversation may begin, indicated by the event user !Voice. At this point, the subscriber may also choose to hang up.

Now suppose the service Call Waiting is introduced. This means that, upon receiving a busy tone, the subscriber may choose to request this feature instead of clearing (according to the informal requirements, in our specification this happens automatically if the called number has the feature activated). This fact appears as a new choice in the corresponding node, as shown in figure 15. If the subscriber requests the call waiting feature, the process Initiator goes into a state where two things may happen: either the connection is finally established, indicated by the internal event line !Switch, upon which the process will get into the same state it would be after a normal connection establishment, or else the subscriber clears and the connection is terminated.

Suppose now that the supplementary service Hold for Enquiry is introduced. This service implies that the subscriber may, if he is engaged in conversation, request this feature (indicated by user !Hold) and, if the new call is established, switch back and forth between both subscribers (user !Switch). See figure 16.

When both services are introduced, the resulting situation is shown in figure 17.

We have simplified matters a little bit, since nothing is said about what happens to the subscriber that receives a call waiting indication or is put in the held call state. But the method will be the same: new choices are added at appropriate nodes in the tree, the process Exchange being responsible for the control of the flow of events.

The relevant question here is: is this method applicable for any possible additional service? Our answer is that it is quite general, since new functions usually imply new choices in the event tree or new control agencies, and the normal invariance of the four connection phases guarantees reusability of the basic processes defined in POTS. If the method is also suitable is harder to answer. We believe that this design enhances modularity and open-endedness, adding flexibility to earlier models. Each service may be implemented and validated independently of other services, as long as no dependencies exist between them. Furthermore, we believe that the programming style chosen also enhances both the surveyability and readability of the code.
Figure 16: The Initiator's process tree with the Hold for Enquiry service.
Figure 17: The Initiator's process tree with the Hold for Enquiry and Call Waiting services.
8 Remarks on Time and the Disabling Operator

One of the objectives in this work has been the development of a correct specification in LOTOS. The disabling operator, which seems natural to apply in disconnections, has been avoided on the ground that it may lead to nondeterministic and incorrect behaviour, as may be seen in [Bou91], where some corrections were introduced in this way.

An inspection of the rules associated with the disabling operator might make things clearer:

\[ B_1 \overset{\mu}{\to} B'_1 \Rightarrow B_1[\succ B_2 \overset{\mu}{\to} B'_2] \succ B_2 \]  
(1)

\[ B_1 \overset{\delta}{\to} B'_1 \Rightarrow B_1[\succ B_2 \overset{\delta}{\to} B'_1] \]  
(2)

\[ B_2 \overset{\mu}{\to} B'_2 \Rightarrow B_1[\succ B_2 \overset{\mu}{\to} B'_2] \]  
(3)

Rules (1) and (3) show that nondeterministic behaviour is possible under certain circumstances, namely if both conditions

\[ B_1 \overset{\mu}{\to} B'_1 \text{ and } B_2 \overset{\mu}{\to} B'_2 \]  
(4)

are possible at the same time.

Often, there is some parallel process waiting to synchronize with the disabling event of \( B_2 \). A typical situation would be:

\[ (B_1[\succ B_2] \| [\beta]) B_3 \]

where

\[ B_1 = \alpha; B'_1 \]  
(5)

\[ B_2 = \beta; B'_2 \]  
(6)

\[ B_3 = \gamma; \beta; \text{stop} \]  
(7)

Now,

\[ (B_1[\succ B_2] \| [\beta]) B_3 \overset{\mu}{\to} (B_1[\succ B_2] \| [\beta]) \beta; \text{stop}. \]

Then, applying rule (3),

\[ (B_1[\succ B_2] \| [\beta]) \beta; \text{stop} \overset{\mu}{\to} B'_2. \]

This is maybe what we intuitively expect. But applying rule (1) we may also obtain

\[ (B_1[\succ B_2] \| [\beta]) \beta; \text{stop} \overset{\mu}{\to} (B'_1[\succ B_2] \| [\beta]) \beta; \text{stop} \]

Rule (3) has no priority over rule (1), nor is there any liveness property associated with it. If process \( B_3 \) were signalling abortion of a connection, we would
like to interrupt immediately the course of action associated with process \( B_1 \).
In the worst case, e.g. if

\[ B'_1 = \delta \gg C, \]

the abortion signal might be missed altogether.

This is what happens in the specification developed in [Bou91]), where the following sequence of events is possible:

```
user ?n.0:Phone !OffHook
line !n.0 !DITO
user !n.0 !Dito.T
user !n.0 !DialKs ?CN.0:Phone
line !CN.0 !CORE (* connection request *)
user !n.0 !HangUp (* caller hangs up *)
line !CN.0 !COAT
I(exit) !CN.0
I(exit) !CN.0
line !CN.0 !RING
user !CN.0 !Ring.T
user !CN.0 !OffHook (* connection established *)
line !CN.0 !CONN
user !CN.0 !SendVo (* called subscriber talking *)
```

Although the subscriber that initiated the connection cleared previously, a ringing tone is applied to the called subscriber, who may even lift the handset and in this way establish the connection; he may then talk to a subscriber that cleared before the connection attempt was performed!

Another reason for avoiding the disabling operator is that it does not fit into the finite-state machine oriented style used throughout the specification.

We have also tried to develop a specification that is correct independently of any time considerations, specially in the context of single connections, which might explain certain abstruse points in it. This should eliminate the possibility that an implementation might produce unexpected behaviour as a consequence of unusual time delays, maybe in connection with an uncommon sequence of events. Nevertheless, in the wider context of system behaviour, it is for example possible for several connections to be started and terminated during the time a subscriber is waiting for tone indication. This unrealistic but otherwise correct behaviour could be eliminated in the design of the implementation. Time aspects are very important for telephony systems, and as long as the notion of time is absent from LOTOS, any specification of telephony systems will probably contain several weird features.
9 Pragmatic assessment of working with LOTOS

For program developers with experience in conventional languages, FDTs pose several new problems. Here they are confronted with a language where semantics overshadows pragmatics. The language is intentionally thin, to keep it as general as possible. The familiar constructs that are part of conventional languages are missing. New constructs may be hard to introduce, since their syntax and semantics must be defined formally and completely before they may be integrated in the language, in contrast to conventional languages, where pragmatic considerations are given precedence in this respect. This lends LOTOS simplicity, but also a certain nakedness characteristic of e.g. assembler languages.

As a consequence, it is far from obvious how to model a real world application in the language. Questions about e.g. the trade off between information coded as data structures or in the form of processes, the function of gates and value offers, the most adequate styles, etc., are not easy to answer, and the implications of design decisions for the future development of the specification are hard to gauge. Too many possibilities are open, a fact which is rather intimidating for beginners.

Compared to conventional languages, it is harder to hide the internals of LOTOS from the programmer. The problem is compounded by the nature of the tools, many of which require acquaintance with the theory underlying their construction. Moreover, the programmer is confronted with not one but two new paradigms: abstract data types and process algebra.

Adequate tools are also missing, although many seems to be under way. Even when the functionality of some of the available tools is acceptable, realistic size problems does not run well in them.

Lack of adequate tools, emphasis on theory, uncertain methodology, combined with the inherent complexity of the subject matter, makes it specially hard to gain confidence in the use of LOTOS.

Notwithstanding the formal character of the subject, it requires a great deal of experience and intuition to get sensible specifications. Large training periods are certainly necessary. In this context, we believe that questions related to the pragmatics of the language should receive more attention.

10 Conclusion

It is apparent that general concepts such as orthogonality, generality and open-endedness are relevant for this kind of application in very specific terms.

Generality is an essential feature since one of the system requirements associated with telephone systems is the possibility of an arbitrary number of connections to develop concurrently. A connection must thus be defined as a general object that may be instantiated in order to become an actual connection.
Open endedness is a desirable feature to simplify adding new functionalities to the system, here in the form of new additional services.

Finally, orthogonality is needed if these services, which often are mutually independent, are to be implemented in a flexible manner, maybe by different specifiers at distinct times.

Analogously, the application of different specification styles has been motivated by factors that are specific to our subject matter, as well as to the overall objectives of the work. This may imply that these styles have a rather general value as specification disciplines for structuring different aspects of a system, independently of the requirements usually associated with the design trajectory of the stepwise refinement methodology. Considerations that were inherent to distinct aspects of our system elicited the application of distinct styles to different parts of the system. In fact, all five specification styles presented above were used. The relation of specification styles to different stages of the design trajectory might thus not be as general as it is supposed.

Even the implementation of the system in a pure state oriented style might have its advantages, notwithstanding the view that this style may produce unstructured specifications and should thus be reserved to the later stages of the design trajectory. In the context of telephony systems, these requirements may be shifted to the data type part, which can be defined in different ways and should also be well structured. We claim that no possibility should be summarily dismissed. For instance, even if the design goal is only an extensional description of the system, internal events may be suitable for several reasons. If modularity, open-endedness, surveyability and the maintenance of a programming environment with a reusable library are essential concerns, then a description of a system involving several internal events might in the end turn out to be a good alternative. In our case, the description of a telephone system in terms of local and end-to-end constraints, with events representing internal, non-observable signals, might be intuitively more appealing and in this way enhance the comprehensibility and surveyability of the resulting specification.

An extensive validation of the specification was performed, following the guidelines presented in [MAQM92]. Simulation and state exploration were performed mainly with the LOTOSPHERE tool SMILE [CS92]. Must and may tests were executed with LOLA [PL93]. Validation of data types was done mainly by evaluation of expressions and with the completion tool in LITE.

The length of the resulting LOTOS specification is approximately 2000 lines, of which 1100 were dedicated to the data type part.

Getting acquainted with the tools and the syntax and semantics of full LOTOS took about 8 weeks of work. Most problems in this stage were related to the fact that many tools are still in a development phase, some being plagued with bugs, incomplete, not well documented or with primitive interfaces towards the user.

The time used in developing the the data type part of the POTS specification was about 2 weeks. This part of the work ran smoothly, since it was conceived
as a series of constructors and selectors in a clear-cut manner, and the rules were to be interpreted as a left-to-right rewriting system. Debugging was done manually and presented few surprises. It turned out that none of the bugs encountered later in the development of the system could be ascribed to the data type part.

Development of the behaviour part was a more arduous task. Several design decisions were made at the beginning which proved to be blind alleys. When the final form for the POTS part of the specification was ready, which took about 4 weeks, the development of the additional services could be carried out in a more straightforward manner. Some of them took only one or two days to develop and test. The total time used for the development of the ten additional services was about four weeks, including a lot of testing.

Integrating all the services in the final specification, testing it, changing many aspects of some features which were found to be less satisfactory, and correcting some bugs took another 3 weeks, since testing got more time consuming at this stage.

Testing of the final specification with LOLA was also carried out, a process that took little more than one week. In this phase only one bug was discovered, which might be interpreted as an indication that the conception of the whole system is sound.

As might be expected, the most demanding and time consuming task was documentation.

Acknowledgements

I would like to thank Lars-åke Fredlund and Patrik Ernberg for their help and for many useful suggestions.

References


