Design, Specification, and Validation of Telephony Systems in LOTOS

By

Rezki Boumezbeur

**Thesis submitted
to the school of graduate studies
in partial fulfillment of the requirements for the
degree of MASTER of Science in Computer Science
under the auspices of the
Ottawa-Carleton Institute for Computer Science**

**At the
UNIVERSITY of OTTAWA
Ottawa, Ontario, Canada**

September 1991

©1991 by R. Boumezbeur, Ottawa, Canada

Acknowledgments

I am most grateful to my supervisor, Dr. Luigi Logrippo, for all the guidance and valuable advice that he has given me throughout my graduate studies.

I acknowledge the contributions made by the University of Ottawa LOTOS group; in particular J. Sincennes, M. Faci, M. Haj-Hussein, and Bernard Stepien for their help. I also like to thank Rob Caza of BNR for providing me with a set of test cases and Peter Koppstein of Bellcore for providing a set of manuals.

Special thanks to my wife, Fatiha, for her patience and support and also, many thanks to my parents, my brother, my son Abdellah, and my daughter Amina for their support.

Finally, I would like to thank with gratitude the government of Algeria, the National Sciences and Engineering Research Council of Canada, and the Telecommunications Research Institute of Ontario for the financial support.

Abstract

LOTOS (Language Of Temporal Ordering Specification) is a Formal Description Technique (FDT) based on the temporal ordering of observational behaviour. It was developed by ISO (International Organization for Standardization) for the specification of OSI (Open Systems Interconnection) services and protocols.

The topic of this thesis is first to present a *Sample Telephone System*, then formalize it using the FDT language LOTOS. The resulting LOTOS specification is then validated using an interpreter. Testing the design of the specification is also discussed.

The thesis is structured as follows: in Chapter 1, we present an introduction to formal description techniques along with a review of some relevant existing work and the objective of this thesis. The next Chapter gives an overview of the LOTOS language. In Chapter 3, we discuss concepts of design of telecommunication systems. The Chapter includes an overview of telecommunication systems along with an informal description of a *Sample Telephone System*. In Chapter 4, we discuss the formal description in LOTOS of the *Sample Telephone System* presented in Chapter 3. Validation and testing of the specification are also part of Chapter 3. The conclusions of the thesis along with a discussion of possible future work follow in Chapter 5.

Appendix A presents some technical abbreviations, and the *Sample Telephone Specification* is presented in Appendix B. In Appendix C, we present a set of test processes used to test the *Sample Telephone Specification*. Finally, some important symbolic trees of processes are presented in Appendix D.

Table of Contents

List of Figures x
List of LOTOS Specifications xii
Chapter 1 Introduction 1
Section 1.1 Background and Motivation of the Thesis 1
Section 1.2 Related Work 3
Chapter 2 An Overview of the LOTOS Language . . . 5
Section 2.1 Introduction 5
Section 2.2 LOTOS Behaviour Expressions 5
2.2.1 Introduction 5
2.2.2 Action Prefix 5
2.2.3 Enabling 6
2.2.4 Internal Action 7
2.2.5 Successful Termination 7
2.2.6 Disabling 8
2.2.7 Choice 8
2.2.8 Inaction 9
2.2.9 Process Instantiation 9
2.2.10 Parallel Composition 10
2.2.10.1 Independent Parallel Composition (Interleaving) . 10
2.2.10.2 Generalized Parallel Composition 10
2.2.11 Hiding 12
2.2.12 Guarded Behaviour Expressions 12
2.2.13 Dischoice Construct 13
2.2.14 Let Construct 14
2.2.15 Par Construct 14

Section 2.3 Scope of Variable Declarations in LOTOS 15
Section 2.4 Debugging LOTOS Specifications Using an Interpreter 18
2.4.1 Introduction 18
2.4.2 Step-by-Step Execution 18
2.4.3 Symbolic Trees of Processes 24
2.4.4 Conclusion 26
Section 2.5 Specification Styles in LOTOS 27
2.5.1 Introduction 27
2.5.2 Monolithic Style 27
2.5.3 Constraint-Oriented Style 28
2.5.4 State-Oriented Style 28
2.5.5 Resource-Oriented Style 28
Chapter 3 The Design of Telephone Systems 29
Section 3.1 An Overview of Telecommunication Systems . . . 29
3.1.1 Introduction 29
3.1.2 Distributed Systems 29
3.1.3 Structure of a Telecommunication System 32
3.1.3.1 The Centralized Structure 32
3.1.3.2 The Decentralized Structure 33
3.1.3.3 The Distributed Centralized Structure 34
3.1.4 Design of a Telephone System 34
3.1.4.1 The Integrated Perspective of a Telephone System 35
3.1.4.2 The Communication Service 36
3.1.4.3 Structure of a Connection 37

Section 3.2 Informal Description of a Sample Telephone System 38
3.2.1 Introduction 38
3.2.2 Technical Terminology 39
3.2.3 Structure of the Sample Telephone System 41
3.2.3.1 Introduction 41
3.2.3.2 An Abstract View 41
3.2.3.3 Structure of a Single Connection 43
3.2.3.3.1 The Transmitter 44
3.2.3.3.2 The Receiver 46
3.2.3.3.3 The Connection Handler 46
3.2.3.3.4 The Additional Subscribers 52
3.2.3.4 The System Network 53
3.2.4 Definition of Telephone Signals 54
3.2.4.1 Introduction 54
3.2.4.2 Busy Signal 55
3.2.4.3 Out Of Service Signal 55
3.2.4.4 Dialing Signal 55
3.2.4.5 Ringing Signal 55
3.2.4.6 Disconnect Signal 56
3.2.4.7 Forward Signal 56
3.2.5 Features of the Sample Telephone System 56
3.2.5.1 Introduction 56
3.2.5.2 Ring Again Feature 57
3.2.5.3 Forward Feature 57
3.2.5.4 Hold Feature 57
3.2.5.5 Listen on Hold 58
3.2.5.6 Transfer/Three-Way Calling Feature 58
3.2.5.7 Conference Call Feature 59

3.2.6 Call Processing Phases 59
3.2.6.1 Introduction 59
3.2.6.2 Origination Phase 61
3.2.6.3 Termination Phase 62
3.2.6.4 Disconnection Phase 65
3.2.7 Informal Description of the System Requirements . . 66
3.2.7.1 Introduction 66
3.2.7.2 Definition of the Set of Requirements 66
Section 3.3 Conclusions 68
Chapter 4 Formal Description in LOTOS of the Sample Telephone System 69
Section 4.1 Introduction 69
Section 4.2 Architecture of the Sample Telephone Specification 69
4.2.1 Introduction 69
4.2.2 Design Principles 70
4.2.3 Use of Specification Styles in the Sample Telephone Specification 71
4.2.3.1 The Extensional Description of a System 71
4.2.3.1.1 The Monolithic Style 71
4.2.3.1.2 The Constraint-Oriented Style 72
4.2.3.2 The Intensional Description of a System 73
4.2.3.2.1 The State-Oriented Style 73
4.2.3.2.2 The Resource-Oriented Style 73
4.2.4 Use of Constraints in the Sample Telephone Specification 74
4.2.4.1 Local Constraints 74
4.2.4.2 End-to-End Constraints 75
4.2.4.3 Global Constraints 75

Section 4.3 General Structure of the Sample Telephone Specification 76
4.3.1 Introduction 76
4.3.2 Top Levels of the Specification 77
4.3.3 Description of the Abstract Data Types 81
4.3.4 Informal Description of the Sample Telephone Specification 83
Section 4.4 Design Steps of the Sample Telephone Specification 86
4.4.1 Introduction 86
4.4.2 Processing Disconnection of Telephones 86
4.4.3 Creating Concurrent Connections 88
4.4.4 Design of a Single Connection 90
4.4.4.1 The Origination Side 91
4.4.4.2 The Destination Side 93
4.4.4.3 The Connection Handler 95
4.4.5 Specification of the Sample Telephone Features . . 98
4.4.5.1 Ring Again and Call Forward Features 98
4.4.5.1.1 Ring Again Feature 99
4.4.5.1.2 Call Forward Feature 101
4.4.5.2 Hold, Transfer/TWC, and Conference Features . 103
4.4.5.2.1 Formal Description of Hold Feature 106
4.4.5.2.2 Formal Description of Transfer/TWC Feature . 108
4.4.5.2.3 Formal Description of Conference Call Feature 108
4.4.5.2.4 Specification of the Invoker of a Feature . . . 110
4.4.5.2.5 The Additional Subscriber 111
4.4.5.2.6 The Party On Hold 113
4.4.5.3 The Controller of Features Processing 114
4.4.6 Formal Description of the System Network 116

Section 4.5 Debugging the Sample Telephone Specification 120
4.5.1 Introduction 120
4.5.2 Step-by-Step Execution of the Specification 121
4.5.3 Symbolic Execution Trees 126
Section 4.6 Testing the Sample Telephone Specification . . . 127
Section 4.7 Conclusion 130
Chapter 5 Conclusions & Future Work 131
Section 5.1 Conclusions 131
Section 5.2 Specification of Timing Characteristics 132
Section 5.3 Future Work 134
Appendix A Technical Abbreviations 135
Section A.1 Keys 135
Section A.2 Signals 135
Section A.3 External Tones 136
Section A.4 Other Abbreviations 137
Appendix B LOTOS Specification of the Sample Telephone System 138
Appendix C Test Processes 183
Appendix D Symbolic Execution Trees of Processes . 204
Bibliography 209

List of Figures

Figure 1	The Abstract View of a Distributed System	30
Figure 2	Refinement of a Distributed System	31
Figure 3	Refinement of an Embedded System (s3)	32
Figure 4	A Centralized Structure of a System	33
Figure 5	A Decentralized Structure of a System	33
Figure 6	A Distributed Centralized Structure of a System	34
Figure 7	The Telephone System, an Integrated Perspective	35
Figure 8	A Snapshot of a Telephone System Configuration	36
Figure 9	Decomposition of a Single Connection	37
Figure 10	The DMS-100 Business Set	39
Figure 11	An Abstract View of the Sample Telephone System	42
Figure 12	A Busy Line	44
Figure 13	A Free Line	45
Figure 14	Forwarding a Call	47
Figure 15	A Sample Evaluation Network	48
Figure 16	An Evaluation Graph for the Transmitter's Communication Scenario	50
Figure 17	An Evaluation Graph for the Receiver's Communication Scenario	51
Figure 18	An Evaluation Graph for the Disconnection's Communication Scenario	52
Figure 19	Structure of the System Network	53
Figure 20	Call Processing Flowchart for Line Origination	60
Figure 21	Flowchart for Origination Tests	61
Figure 22	Flowchart for Termination Tests	63
Figure 23	Flowchart for Ring Again Feature	64
Figure 24	Flowchart for Transmission Path Disconnection	65
Figure 25	Top Level's Graphical Representation of the Specification	80
Figure 26	Refinement of a One_Connection	81
Figure 27	Different Types of a Disconnection	87
Figure 28	Creating Multiple Connections	89
Figure 29	Refinement of the Origination Side	91

Figure 30 Refinement of the Destination Side 94
Figure 31 A Configuration of the Connection Handler 96
Figure 32 Temporal Ordering of Internal Events in a Normal Call 97
Figure 33 Flowchart of Processing a Ring Again Feature . . 99
Figure 34 Flowchart of Processing a Forward Feature . . . 102
Figure 35 Flowchart of the First Stage of Invoking a Feature 105
Figure 36 Invoking a Feature within a Conversation 107
Figure 37 The Controller of Features 115
Figure 38 Refinement of the System Network 117

List of LOTOS Specifications

Specification 1	The Topmost Level of the Specification	76
Specification 2	General Structure	77
Specification 3	Top Levels of the Specification	79
Specification 4	Process Multi_Connections Definition	89
Specification 5	Process One_Connection Definition	90
Specification 6	Process Subscriber Definition	91
Specification 7	Process Origination_Side Definition	93
Specification 8	Process Destination_Side Definition	94
Specification 9	Process Connection_Handler Definition	98
Specification 10	Process Ring_Again_Feature Definition	101
Specification 11	Process Call_Forwarded Definition	103
Specification 12	Process User_Talk Definition	104
Specification 13	Process Feature Definition	105
Specification 14	Process More_Conference Definition	109
Specification 15	Process Invoke_Feature Definition	111
Specification 16	Process Additional_Subscriber Definition	112
Specification 17	Process Party_On_Hold Definition	114
Specification 18	Process System_Network Definition	118
Specification 19	The Symbolic Execution Tree of the Process Origination_Side	127
Specification 20	A Simple Call Test Process	129
Specification 21	The Symbolic Execution Tree of the Simple Call Test Process	130

Chapter 1 Introduction

Section 1.1 Background and Motivation of the Thesis

The conventional method of *Telephone System* design and implementation does not use an effective methodology for finding design errors before the implementation is derived. Consequently, many design errors become visible only after the implementation phase. Removing such errors requires costly redesign and reimplementation. In this thesis, we propose a different methodology, involving the use of an executable specification language. In such a methodology, the specification becomes a prototype of the implementation. By executing the specification, many design errors can be caught before they are cost in the implementation.

LOTOS is an executable specification language developed by ISO for the specification of OSI services and protocols. It has a formal basis and it has been shown to be appropriate for the specification of other types of open distributed systems as well. Furthermore, it is important to note that although the Specification and Description Language (SDL) [CCI88] is the most commonly used language for the formal description of standard telephone services, the CCITT (Consultative Committee for International Telegraph and Telephone) has been studying LOTOS as a more advanced technique for the formal description of distributed systems.

The first aim of this thesis is to design a *Telephone System* that provides its users with a number of telephone features. Then, the structure of this *Sample System* will be reflected in its formal description using the FDT language LOTOS. The resulting *Telephone Specification* is validated and its design is tested by use of an interpreter. By this application, we show that the LOTOS language is appropriate for formally specifying not only Plain Old Telephone systems (POTS) but also more complex ones. We show also that LOTOS concepts allow users to design a system in a stepwise fashion. The telephone features provided by the *Sample System* include *Call Forward*, *Ring Again*, *Call Transfer/Three-Way Calling*, *Hold a Call*, and *Conference Call* features.

Our LOTOS specification of the *Sample Telephone System* allows for an arbitrary number of subscribers to use the *Telephone Service* simultaneously.

Therefore, multiple connections may be processed in parallel. Another useful feature of this *Sample Telephone Specification*, is that it allows the environment to update the system at any time during its functioning[1]. Because of the complexity of the system designed, our specification uses a mixture of the different styles known in the domain of LOTOS specifications. The use of any style may depend on the level of abstraction of the system.

An important particularity of *Telephone Systems* is that they are governed by the use of signals. The type of a signal being exchanged depends on the status of the requested line. This has led to a design of a specification based on the use of constraints. Constraints include the functioning of a specific entity in real world *Telephone Systems* and of the system requirements on the use of a line. The possibility of well structuring a specification in LOTOS helps us to design the *Sample Telephone System* in a stepwise refinement approach and may lead to a more precise description of the system. This method may help designers to get a complete specification of telephone features at the design stage, and therefore it enables early detection of design errors. All these aspects are discussed in detail in this thesis.

For the purpose of readers acquainted with the area of software engineering and real time systems design, we must mention that some technical terms, such as 'design', 'architecture', and 'structure' are used in this thesis with meanings that may differ from the ones encountered in these areas. Our description of the telephone system in LOTOS uses the features of this language to describe the system's externally visible behaviour. It does not extend to describe the structural modules of the system in an implementation architecture. This distinction is mainly due to the concepts of the LOTOS language as a specification oriented language. So, we have used a specification structuring style that aims at maximum expressiveness in view of what can be done using the various LOTOS constructors, rather than aiming at structuring the specification in view of an actual system structure. In other words, the terms mentioned above refer to specification structuring, rather than to physical system structuring. If our aim had been to show a system structure, we would have had to use a different, and probably more complicated, structure.

---

[1] Updating the system refers to adding new lines to the telephone system or removing some old lines from that system.

Similarly, we use the term ‘component’ to denote a specification component or a part of it. A component here is represented as a process which interacts with the other components via gates. So, the remarks above are also valid for the term ‘component’ as it is used here and where the philosophical meaning differs from its use in ‘real time systems design’.

LOTOS allows to describe systems in a modular way. This has helped us in the construction (design) of the system in a step wise fashion. A system may be decomposed into subsystems where each subsystem may be viewed as a process in a specification. A process in LOTOS can be a constraint, but it may also represent a component which sometimes represents an independent part of the overall system (specification). Therefore, from the design point of view this construction can be viewed as at least a step in the design of a system (its behavioural part).

Although there may be some disagreements over the appropriateness of the use of these terms, we have used them consistently in the thesis. We hope that these clarifications will help the reader not to confuse their use in the two distinct domains of LOTOS and real time systems design.

We conclude this thesis with a discussion of the specification debugging method which involved the use of an interpreter [HH88], along with a presentation of some interesting test sequences taken from real world functioning of *Telephone Systems* and used as a basis to test the design of the specification in LOTOS.

Section 1.2 Related Work

During the last few years, our research group has been investigating different applications of the LOTOS language. This includes the formal specification of some OSI services and protocols [Gue89], distributed systems [HH88], and basic telephone systems [FLS90][FLS91].

The latter study was the subject of an early work leading to the specification of a simple telephone system based on the concepts of POTS (Plain Old Telephone Systems). The specification given in [FLS90] was used as the basis of our work. As mentioned in the previous section, this specification included only the basic operations of POTS.

A number of different techniques have been used in the formal description of *Telephone Systems*. In [Zav85], a formalism for describing distributed systems, applied to *Telephone Systems*, is presented. Biebow and Hagelstein [BH] have given a different formalism to write specifications by using algebraic abstract data types. The latter was applied on a *Telephonic* example. In 1989, Tvrdy [Tvr89] published a paper showing the application of the LOTOS language to the formal specification of *Telephone Systems*. Also, Cam and Vuong [CV90] have specified a mobile telephone system in LOTOS. In late 1989, an object-oriented approach [Sim89] was applied on the specification of a *Telecommunication Service* using the MLog programming language [Kar89]. It includes the basic telecommunication service and also a number of supplementary services and their combinations.

Chapter 2 An Overview of the LOTOS Language

Section 2.1 Introduction

LOTOS is a formal specification language developed by ISO for the specification of OSI services and protocols. The basic idea that LOTOS developed from was that distributed systems can be described by defining the temporal relation between events in the externally observable behaviour of a system [ISO88][ISO89].

LOTOS specifications consist of two components: 1) the control component, based on Milner's CCS [Mil80] and Hoare's CSP [Hoa85], deals with the description of process behaviours and interactions, and 2) the data component that describes the data structures and value expressions used in that system. This part of LOTOS is based on the formal theory of algebraic abstract data types. Most concepts in this component were inspired by the abstract data type technique ACT ONE.

In the next sections we give a brief overview of the LOTOS language, we discuss the execution of LOTOS specifications using an interpreter, and then we discuss the use of specification styles in LOTOS.

Section 2.2 LOTOS Behaviour Expressions

2.2.1 Introduction

In this section we present the semantics of some LOTOS operators. The examples used in the explanation of operators are taken from our LOTOS *Telephone Specification*. This is to facilitate understanding of the telephone systems specifications in LOTOS. Fore more details on LOTOS language see [BB87]. Behaviour expressions are built out of actions and behaviour operators.

2.2.2 Action Prefix

This operator is used to prefix a behaviour expression with an event. An event is an action composed of a gate, an optional list of expected or offered

values and an optional predicate (condition on values). It is also called “action” or “action denotation”.

process *Caller [user,line]* :**noexit**:=
user !*OffHook* ?*N:phone* **;**
Calling [user,line] (N)
endproc

This can be translated as, *Calling* process is preceded by an *OffHook* event from the user originating a call.

2.2.3 Enabling

This operator is used to sequence two behaviour expressions or processes. The first process or behaviour expression has to successfully terminate in order to enable the execution of the second one.

process *Calling [user,line] (N:phone)* :**noexit**:=
Dialing [user,line] (N)
>>
Termination_Phase [user,line] (N)
endproc

The termination of the process *Dialing* enables the execution of process *Termination_Phase*’s events.

Another type of enable operator is the sequential composition with value passing.

process *Calling [user,line] (N:phone)* :**noexit**:=
Dialing [user,line] (N)
>>
accept *N:phone* **in**
line !*N* !*CONN* ;
User_Talking [user,line] (N)

endproc

The first process "*Dialing*" terminates and enables the second process to be executed with passing to it the telephone numbers renamed by *N* and *CN* in the **accept** construct.

2.2.4 Internal Action

An internal action, denoted "**i**", is an action that a process can execute independently, i.e. without collaboration with the environment. It can be used to represent an internal event such as a shutdown or internal synchronization between processes.

process *Multi_Connections [user,line]* :**noexit**:=
One_Connection [user,line]
|||
i ;
Multi_Connections [user,line]
endproc

The internal action "**i**" in this process represents the internal events performed by the system before creating a new connection. It is also used to make it possible to execute the specification on a simulator, because without it, an infinite number of copies of process "*One_Connection*" would become simultaneously enabled (unguarded recursion).

2.2.5 Successful Termination

A process or behaviour expression terminates successfully if it performs all the actions and then executes the LOTOS action **exit** which offers a special event and then behaves as **stop**.

process *Dialing [user,line] (N:phone)* :**noexit**:=
line !*N* !*DITO* ;
user !*N* !*DialKs* ?*CN:phone* ;
line !*CN* !*CORE* ;

exit (N)

endproc

On executing the action **exit***(N)* the process *Dialing* is said to be successfully terminated. If *Dialing* enables another process, we have an internal action followed by the first action of the second process.

2.2.6 Disabling

This operator is used to express the situations where a process or behaviour expression can be killed by an exceptional event during its normal functioning. And as our goal is to specify communication systems, such situation can be seen as the disconnection process that may occur at any time to interrupt the communication.

process *User_Talking [user,line] (N:phone)* :**noexit**:=
Conversation [user,line] (N)
[>
User_Termination [user,line] (N)
endproc

Note here that the process *User_Termination* may occur and thus interrupt the communication path, even before the conversation starts.

2.2.7 Choice

This operator allows the user to define different alternatives for a given process depending on its actual state.

In the process *User_Termination* (see below) we define two alternatives for the user terminating a call. The user may go *OnHook*, or it may receive a disconnection indication from the other side after which time it will be forced to go *OnHook*.

process *User_Termination [user,line] (N:phone)* :**noexit**:=
Hang_Up [user,line] (N)
[]

line !*N* !*DSIN* ;

Hang_Up [user,line] (N)

endproc

Note that the user cannot hang up the telephone and receive a disconnection indication (*DSIN*) at the same time.

2.2.8 Inaction

An inaction is denoted by a "**stop**" and represents a deadlock. It is used to terminate (unsuccessfully) the sequence of actions of a behaviour expression or a process.

process *Hang_Up [user,line] (N:phone)* :**noexit**:=

user !*N* !*HangUp* ;

line !*N* !*DISC* ;

stop

endproc

2.2.9 Process Instantiation

When instantiating a process, its formal variable parameters are substituted by the actual ones and its formal gates are relabelled by the actual gates at instantiation time.

In telephone systems users can stay on the telephone for as long as they wish (for a single call). In LOTOS, this can be achieved by recursively instantiating the same process corresponding to the sequence of events for a specific user.

process **Conversation** *[user,line] (N:phone)* :**noexit**:=

user !*N* !*SendVo* ;

line !*N* !*VOCE* ;

Conversation *[user,line] (N)*

endproc

2.2.10 Parallel Composition

There are three parallel composition operators in LOTOS: interleaving, full parallelism, and selective parallelism.

2.2.10.1 Independent Parallel Composition (Interleaving) The interleaving operator " ||| " is used to allow behaviour expressions or processes to be executed independently and in parallel.

```
process Telephones [user,line] :noexit:=
    ( user !OffHook ?N:phone ;
      line !N !DITO ;
      user !N !DialKs ?CN:phone ;
      line !CN !CORE ;
      stop
    )
    |||
    ( line ?CN:phone !COAT ;
      line !CN !RING ;
      user !CN !OffHook ;
      stop
    )
endproc
```

This represents a pure interleaving of two behaviour expressions. First either one of the first actions in the two behaviour expressions may be offered and then any one of the remaining actions in the behaviour expressions respectively may be offered and so on.

2.2.10.2 Generalized Parallel Composition The parallel composition operator " || " is used to force events on some specific gates to synchronize.

In our previous example, a user's line may get a connection attempt (*COAT*) signal before initiating a call from another side. To avoid such an anomaly, one may add in parallel a third process that synchronizes with the other two on gate "*line*"

to force a sequence of events on that gate to happen according to the telephone system's functioning.

```
process Connection [user,line] :noexit:=
  ( ( user !OffHook ?N:phone  ;
      line !N !DITO ;
      user !N !DialKs ?CN:phone ;
      line !CN !CORE ;
      stop
    )
    |||
    ( line ?CN:phone !COAT ;
      line !CN !RING ;
      user !CN !OffHook ;
      stop
  ) )
  |[line]|
  ( line ?N:phone !DITO ;
    line ?CN:phone !CORE ;
    line !CN !COAT ;
    line !CN !RING ;
    stop
  )
endproc
```

This forces the connection attempt (*COAT*) signal to occur after a connection request (*CORE*) has been sent from the other side. The *CORE* signal is sent to the destination only if the origination side gets dial tone (*DITO*) first and then dials the destination number.

In our telephone specification we specified the process describing the controller in parallel with the processes defining the caller and called parties. The controller synchronizes with the others on gate "*line*", thus forcing a correct sequencing of the actions performed from different sides (telephones) participating in a single connection.

2.2.11 Hiding

The operator " **hide** <*gatelist*> **in** " is used to hide actions (gates) which are internal to the behaviour of a system.

In telephone systems the actions performed by the switching module are internal to the system and not visible to the environment. In specifying telephone systems in LOTOS all actions offered on gate "*line*" are internalized by hiding that gate. Therefore, they become internal actions as explained in section 2.2.4. However actions performed by users on telephones are not hidden and occur at gate "*user*" in our specification. In this example, gate "*line*" is not known outside process Connection, and therefore there is no need to wait for synchronization from outside.

process *Connection [user,line]* :**noexit**:=
hide *line* **in**
(*Caller [user,line]*
|||
Called [user,line]
)
|[*line*]|
Controller [line]
endproc

2.2.12 Guarded Behaviour Expressions

In LOTOS one may state conditions on behaviour expressions by using guards. If the expression in the guard is evaluated to true then the following behaviour expression can be executed. However, if the expression in the guard is evaluated to false then the following behaviour expression is equivalent to stop. In the latter case, the remaining alternatives (if any) are explored.

```
process Hang_Up [user,line] (N1,N2:phone) :noexit:=
  user !N1 !HangUp ;
  line !N1 !DISC ;
  ( [ N1 eq N2 ] —>
      stop
    []
    [ N1 ne N2 ] —>
      line !N2 !DISC ;
      stop
  )
endproc
```

In this example, *N2* stands for the user to whom the call was destined. *N1* stands for the user to whom the call was forwarded by *N2*. Once user *N1* goes *OnHook* his or her line is disconnected. Then, if the call has been forwarded (*N1* ne *N2*) disconnection of the user N2 is also required.

2.2.13 Dischoice Construct

With the choice operator "*[]*" we can only model a choice between a finite number of alternatives. As shown in section 2.2.2, let *Calling [user,line] (N)* be a behaviour expression that depends on a variable *N* (of sort telephone). We can now specify the choice between the processes *Calling [user,line] (N)* for all telephone numbers (available) by writing:

```
process Caller [user,line] :noexit:=
  choice N:phone [] Calling [user,line] (N)
  where
    process Calling [user,line] (N:phone) :noexit:=
      user !OffHook !N ;
      Dialing [user,line] (N)
      >>
        Termination_Phase [user,line] (N)
    endproc
```

endproc

Note that the event "*user* !*OffHook* ?*N:phone* **;**" has been replaced by the event offer "*user* !*OffHook* !*N* **;**". This type of dischoice is called "*summation on values*". Another type of dischoice construct is the so called "*summation on gates*". In this type a set of gate identifiers is used to replace specific gates for different alternatives. Therefore, a behaviour expression containing a dischoice construct can be interpreted as a set of alternatives using the choice operator.

2.2.14 Let Construct

In LOTOS, it is possible to substitute value expressions by variable identifiers in a behaviour expression. A concrete example of the use of this construct would be the following behaviour expression,

process *Caller [user,line]* :**noexit**:=
  let *N:phone = 10 , Of:Key = OffHook* **in**
    user !*Of* !*N* **;**
    Dialing [user,line] (N)
    >>
      Termination_Phase [user,line] (N)
endproc

The variable identifiers *N* and *Of* are (locally) used instead of the values *10* and *OffHook* respectively.

2.2.15 Par Construct

Like the choice and let constructs, the generalized parallel operator "**par**" was defined to compact specifications. It is also possible to interpret this generalized parallel operator in terms of parallel compositions of the behaviour expression that follows the parallel operator.

As example of this construct, take the following behaviour expression,

par *g* **in** *[a,b,c]* **|[***d***]|**
  Beh [g,d]

This is equivalent to,

Beh [a,d]
  |[*d*]|
Beh [b,d]
  |[*d*]|
Beh [c,d]

Section 2.3 Scope of Variable Declarations in LOTOS

A variable declaration in LOTOS has the form **?x:t** , where **x** is the name of the variable and **t** is its sort identifier that indicates the domain of values over which **x** ranges. Each variable declaration in a specification has an associated *scope*.

The scope of a variable declaration **x** includes all inner behaviour expressions up to and excluding the behaviour expression where **x** is redeclared.

As in block-structured programming languages, the scope of a variable may include the scope of another variable by the same name. A variable declaration in LOTOS can be defined in five different ways:

1. A variable declaration *as a formal parameter* of a process,

process *User_Termination [user,line] (N:phone)* :**noexit**:=

  Hang_Up [user,line] (N)
  []
  line !*N* !*DSIN* ;
  Hang_Up [user,line] (N)

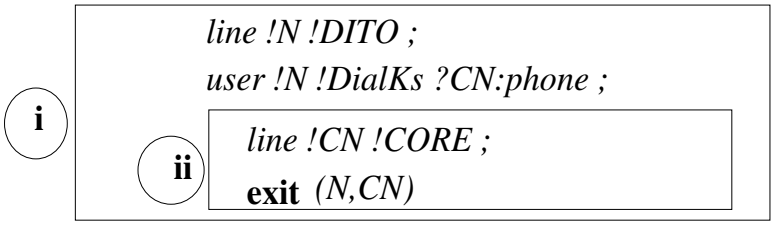
endproc

The scope of the variable declaration *N* is the definition block of the process *User_Termination*.

2. A variable declaration *in an action denotation*,

process *Origination_Side [user,line]* :**exit**(*phone,phone*):=

user !OffHook ?N:phone ;



endproc

The scope of the variable declaration *N* that appears in the first action denotation is the behaviour expression (**i**) following that action in an action prefix construct. The scope of the variable declaration *CN* that appears in the third action denotation is the behaviour expression (**ii**) following that action.

3. A variable declaration *in an enable* construct,

process *Destination_Side [user,line]* :**noexit**:=

line ?*CN:phone* !*COAT* ;
exit (*CN*)
>>
accept *CN:phone* **in**

line !*CN* !*RING* ;
user !*CN* !*OffHook* ;
line !*CN* !*CONN* ;
User_Talk [user,line] (CN)

endproc

The scope of the variable declaration *CN* (first occurrence) terminates at the action **exit** (end of action prefix construct). However, a new scope is defined for the second occurrence of the variable declaration *CN* in the **accept** operator of an

enable construct. It includes all the inner behaviour expressions following that operator where the variable declaration *CN* is not redeclared.

4. A variable declaration *in a dischoice construct*,

```
process Hang_Up [user,line] :exit:=
  choice N:phone []

    user !N !HangUp ;
    line !N !DISC ;
    exit

endproc
```

The scope of the variable declaration *N* defined by the dischoice construct is the inner behaviour expressions following the operator "[]" .

5. A variable declaration *in a let construct* as a local definition,

```
process Origination_Side [user,line] :noexit:=
  let N:phone = 14 , CN:phone = 15 in

    user !N !OffHook ;
    line !N !DITO ;
    user !N !DialKs !CN ;
    line !CN !CORE ;
    exit
    >>
     line !N !CONN ;
     User_Talk [user,line] (N)

endproc
```

The scope of the variable declarations *N* and *CN* is the local definition of the behaviour expression following the let construct in an inner level. If a variable is redeclared a new scope for that variable declaration is defined.

Section 2.4 Debugging LOTOS Specifications Using an Interpreter

2.4.1 Introduction

A useful feature of a specification language is executability. For an executable specification language it is possible to build an interpreter, with whose help specifications can be executed and tested.

At the University of Ottawa, an interpreter for the LOTOS language has been implemented. This interpreter supports many functions useful for simulating the execution of specifications and performing other related tasks. For more details about this interpreter, called ISLA, see [HH88][LOBF88][GL89b].

In the following we discuss with examples some of ISLA's functions that were needed for debugging and testing our specification.

2.4.2 Step-by-Step Execution

One of the basic features that ISLA offers is the execution of specifications by steps. At each step, the user chooses the next action to be taken among all possible actions that are offered at that level and may be asked to enter the data needed to perform this action (if any). This method is very useful for checking the conformance of a system (defined informally) to its formal definition in LOTOS.

This may be realized by any of the following:

1. checking if given test sequences are accepted by the formal specification.

2. checking if the test sequences obtained by executing the specification are specified by the informal definition of the system.

3. checking if test sequences that are not specified by the system are also not accepted by the formal specification.

The following example taken from our specification shows the step-by-step execution. Since it is not practical to show it on the whole specification we took only the process that simulates the conversation between two users after their connection has been established. In order to be able to execute the process separately, the telephone numbers of the users are given in a let construct. At any time during the conversation any participating user may interrupt

it (by going *OnHook*). The conversation is controlled by a special process *Control_Conversation*.

```
 58 process Users [user,line] :noexit:=
(* Generation of telephone numbers *)
 59  let N:phone = 2222 , CN:phone = 1111 in
 60  ( ( Conversation [user,line] (N)
 61      [>
(* The conversation may be disabled by any party *)
 62        User_Termination [user,line] (N) )
(* Interleaving of the telephones *)
 63    |||
 64    ( Conversation [user,line] (CN)
 65      [>
 66        User_Termination [user,line] (CN) )  )
(* Synchronization of the Controller with the telephones *)
 67  |[line]|
 68  ( Control_Conversation [line] (N,CN)
 69    [>
(* The Control of the disconnection of telephones *)
 70      ( Control_Termination [line] (N,CN)
 71        []
 72        Control_Termination [line] (CN,N) ) )
 73
 74  where
 75   process Conversation [user,line] (M:phone)
                :noexit:=
 76    user !M !SendVo ;
 77    line !M !VOCE ;
 78    Conversation [user,line] (M)
 79   endproc (* Conversation *)
 80
 81   process Control_Conversation [line] (M1,M2:phone)
                :noexit:=
 82    line !M1 !VOCE ;
 83    Control_Conversation [line] (M1,M2)
```

```
 84     []
 85     line !M2 !VOCE ;
 86     Control_Conversation [line] (M1,M2)
 87    endproc (* Control_Conversation *)
 88
 89    process User_Termination [user,line] (M:phone)
              :noexit:=
 90     user !M !HangUp ;
 91     line !M !DISC ;
 92     stop
 93     []
(* Receiving a disconnection indication *)
 94     line !M !DSIN ;
 95     user !M !HangUp ;
 96     line !M !DISC ;
 97     stop
 98    endproc (* User_Termination *)
 99
100    process Control_Termination [line] (M1,M2:phone)
              :noexit:=
(* Any party may be disconnected *)
101     line !M1 !DISC ;
(* A disconnection indication is sent to the other party *)
102     line !M2 !DSIN ;
103     line !M2 !DISC ;
104     stop
105    endproc (* Control_Termination *)
106   endproc (* Users *)
```

At the beginning of the execution, the following four actions are offered to the user by the interpreter. Note that the line number in the specification of an offered action is shown on the right side of that action and its sequencing number is shown on its left side. This offer can be explained by stating that any user whose number is either 1111 or 2222 may talk (*SendVo*) or go *OnHook* at this stage.

==========

<1>- user !2222:phone !*SendVo*:action —> bh1 [76]

<2>- user !2222:phone !*HangUp*:action —> bh2 [90]
<3>- user !1111:phone !*SendVo*:action —> bh3 [76]
<4>- user !1111:phone !*HangUp*:action —> bh4 [90]
==========
ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 1
Passed evaluated value ==> 2222
Passed evaluated value ==> *SendVo*

The first action is chosen to show that the user on line 2222 is now talking and goes to next stage where it listens (*VOCE*).

==========
<1>- user !2222:phone !*HangUp*:action —> bh1 [90]
<2>- user !1111:phone !*SendVo*:action —> bh2 [76]
<3>- user !1111:phone !*HangUp*:action —> bh3 [90]
<4>- i (hiding: line !2222:phone !*VOCE*:Signal) —> bh4 [77,82]
==========
ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 4
Internal event is executed
Passed evaluated value ==> 2222
Passed evaluated value ==> *VOCE*

The user on line 2222 is now listening (action 4). Any user may terminate the call (actions 1 and 3), or user on line 1111 may send voice (action 2).

==========
<1>- user !2222:phone !*SendVo*:action —> bh1 [76]
<2>- user !2222:phone !*HangUp*:action —> bh2 [90]
<3>- user !1111:phone !*SendVo*:action —> bh3 [76]
<4>- user !1111:phone !*HangUp*:action —> bh4 [90]
==========
ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 3
Passed evaluated value ==> 1111

Passed evaluated value ==> *SendVo*

At this stage both users are on conversation. The action number 3 is chosen to show the next step where the user on line 1111 finishes talking.

```
=========================================================
  <1>- user !2222:phone !SendVo:action —> bh1 [76]
  <2>- user !2222:phone !HangUp:action —> bh2 [90]
  <3>- user !1111:phone !HangUp:action —> bh3 [90]
  <4>- i (hiding: line !1111:phone !VOCE:Signal) —> bh4 [77,85]
=========================================================
  ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 3
  Passed evaluated value ==> 1111
  Passed evaluated value ==> HangUp
```

While listening to the user on line 2222 who is talking, the user on line 1111 goes *OnHook* (action 3). As soon as a party goes *OnHook* its disconnection is required, although it is also possible for the other party to talk or go *OnHook*.

```
=========================================================
  <1>- user !2222:phone !SendVo:action —> bh1 [76]
  <2>- user !2222:phone !HangUp:action —> bh2 [90]
  <3>- i (hiding: line !1111:phone !DISC:Control_Signal) —> bh3 [91,101]
=========================================================
  ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 3
  Internal event is executed
  Passed evaluated value ==> 1111
  Passed evaluated value ==> DISC
```

In this step we choose action 3 that corresponds to the disconnection (*DISC*) of the user who goes *OnHook*, causing a disconnection indication (*DSIN*) signal to be received by the other side.

```
=========================================================
```

<1>- user !2222:phone !*SendVo*:action —> bh1 [76]

<2>- user !2222:phone !*HangUp*:action —> bh2 [90]

<3>- i (hiding: line !2222:phone !*DSIN*:Signal) —> bh3 [94,102]

==========

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 3

Internal event is executed

Passed evaluated value ==> 2222

Passed evaluated value ==> *DSIN*

At this level the action number 3 is chosen for disconnection indication (*DSIN*) signal

==========

<1>- user !2222:phone !HangUp:action —> bh1 [95]

==========

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 1

Passed evaluated value ==> 2222

Passed evaluated value ==> *HangUp*

In this step only one action is offered and the user is forced to choose it. After disconnection of the user on line 1111, the user on line 2222 is now forced to go *OnHook*.

==========

<1>- i (hiding: line !2222:phone !*DISC*:Control_Signal) —> bh1 [96,103]

==========

ACT: la[ctions][<N>] | ? | h[elp] | <action number> | <command> ==> 1

Internal event is executed

Passed evaluated value ==> 2222

Passed evaluated value ==> *DISC*

This is the last action to be offered and executed by the user. It corresponds to the disconnection (*DISC*) signal from the user on line 2222. After this, the system will stop.

2.4.3 Symbolic Trees of Processes

The step-by-step execution allows the user to interact with the execution of the specification at each step and to choose one action among the offered ones. Its execution is slow. The interpreter (ISLA) also allows the user to obtain the symbolic tree of a specific process or of the whole specification where all paths are symbolically executed and shown up to a maximum depth specified by the user [GL89b]. Symbolic execution means that values to be provided by the environment are represented by symbolic names. The predicates that involve such values are assumed to be true.

The symbolic tree of execution of the last example is given below. Each action in the tree is followed by its line number in the specification and the depth number of each action is given by the number of bars that precede that action. For reasons of space, we have limited ourselves to a depth of three.

```
1 user !2222:phone !SendVo:Key [76]
| 1 user !2222:phone !HangUp:Key [90]
| | 1 user !1111:phone !SendVo:Key [76] ==> continue
| | 2 user !1111:phone !HangUp:Key [90] ==> continue
| | 3 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101] ==> continue
| 2 user !1111:phone !SendVo:Key [76]
| | 1 user !2222:phone !HangUp:Key [90] ==> continue
| | 2 user !1111:phone !HangUp:Key [90] ==> continue
| | 3 i (hiding: line !2222:phone !VOCE:Signal) [77,82] ==> continue
| | 4 i (hiding: line !1111:phone !VOCE:Signal) [77,85] ==> continue
| 3 user !1111:phone !HangUp:Key [90]
| | 1 user !2222:phone !HangUp:Key [90] ==> continue
| | 2 i (hiding: line !2222:phone !VOCE:Signal) [77,82] ==> continue
| | 3 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101] ==> continue
| 4 i (hiding: line !2222:phone !VOCE:Signal) [77,82]
| | 1 user !2222:phone !SendVo:Key [76] ==> continue
| | 2 user !2222:phone !HangUp:Key [90] ==> continue
| | 3 user !1111:phone !SendVo:Key [76] ==> continue
```

| | 4 user !1111:phone !HangUp:Key [90] ==> continue
2 user !2222:phone !HangUp:Key [90]
| 1 user !1111:phone !SendVo:Key [76]
| | 1 user !1111:phone !HangUp:Key [90] ==> continue
| | 2 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101] ==> continue
| | 3 i (hiding: line !1111:phone !VOCE:Signal) [77,85] ==> continue
| 2 user !1111:phone !HangUp:Key [90]
| | 1 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101] ==> continue
| | 2 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101] ==> continue
| 3 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101]
| | 1 user !1111:phone !SendVo:Key [76] ==> continue
| | 2 user !1111:phone !HangUp:Key [90] ==> continue
| | 3 i (hiding: line !1111:phone !DSIN:Signal) [94,102] ==> continue
3 user !1111:phone !SendVo:Key [76]
| 1 user !2222:phone !SendVo:Key [76]
| | 1 user !2222:phone !HangUp:Key [90] ==> continue
| | 2 user !1111:phone !HangUp:Key [90] ==> continue
| | 3 i (hiding: line !2222:phone !VOCE:Signal) [77,82] ==> continue
| | 4 i (hiding: line !1111:phone !VOCE:Signal) [77,85] ==> continue
| 2 user !2222:phone !HangUp:Key [90]
| | 1 user !1111:phone !HangUp:Key [90] ==> continue
| | 2 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101] ==> continue
| | 3 i (hiding: line !1111:phone !VOCE:Signal) [77,85] ==> continue
| 3 user !1111:phone !HangUp:Key [90]
| | 1 user !2222:phone !SendVo:Key [76] ==> continue
| | 2 user !2222:phone !HangUp:Key [90] ==> continue
| | 3 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101] ==> continue
| 4 i (hiding: line !1111:phone !VOCE:Signal) [77,85]
| | 1 user !2222:phone !SendVo:Key [76] ==> continue
| | 2 user !2222:phone !HangUp:Key [90] ==> continue
| | 3 user !1111:phone !SendVo:Key [76] ==> continue
| | 4 user !1111:phone !HangUp:Key [90] ==> continue
4 user !1111:phone !HangUp:Key [90]
| 1 user !2222:phone !SendVo:Key [76]
| | 1 user !2222:phone !HangUp:Key [90] ==> continue
| | 2 i (hiding: line !2222:phone !VOCE:Signal) [77,82] ==> continue
| | 3 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101] ==> continue
| 2 user !2222:phone !HangUp:Key [90]

| | 1 i (hiding: line !2222:phone !DISC:Control_Signal) [91,101] ==> continue

| | 2 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101] ==> continue

| 3 i (hiding: line !1111:phone !DISC:Control_Signal) [91,101]

| | 1 user !2222:phone !SendVo:Key [76] ==> continue

| | 2 user !2222:phone !HangUp:Key [90] ==> continue

| | 3 i (hiding: line !2222:phone !DSIN:Signal) [94,102] ==> continue

2.4.4 Conclusion

A specification may be debugged by using an interpreter. This is a very important task in the design stage. It may help the designer to detect design errors, which can therefore be fixed before generating the final description of a system.

The step-by-step execution of a specification is very slow, and therefore it is convenient at the first stages of the design only. However, the symbolic execution method may be used to check the different possible paths allowed by the specification for a specific case, in particular when that case is too far to be reached by the step-by-step execution approach. Note here that for big specifications, the generation of the symbolic execution trees may also explode. Therefore for such specifications, the use of symbolic execution trees is rather recommended for specific processes instead. The method may also be used for the whole specification if the user specifies an appropriate depth of execution, as discussed in Section 2.4.3.

Therefore, both methods are needed in the debugging process of a specification. One starts with the step-by-step execution to test the possible paths, and once this method becomes too slow, one may think of generating symbolic tree of specific stages of the specification, then checking the different branches of the tree.

Another method that is found to be appropriate for checking big specifications, is the composition in parallel of sequences of actions with the whole specification. Then, we generate the symbolic execution tree for the combined specification. For more details on this method, refer to Section 4.6.

Section 2.5 Specification Styles in LOTOS

2.5.1 Introduction

A specification is an abstract definition of a set of the system's requirements, and can be used to derive various implementations. For comprehensibility and to efficiently express its functionality, a specification needs to be well structured, even if it introduces implementation-oriented elements.

LOTOS supports four well-defined specification styles [VSvS88], called a *monolithic*, a *constraint-oriented*, a *state-oriented*, and a *resource-oriented* style. These specification styles support the complete design trajectory from architecture to implementation. Each one of these specification styles has its own uses in telephone systems specifications, and depending on the goal of the specification designer, the specification may use any one or a mixture of these styles. Following, is a brief discussion of the different styles and their uses in LOTOS specifications.

A more detailed discussion of the different specification styles along with their use in the *Sample Telephone Specification* are presented later in Section 4.2.3.

2.5.2 Monolithic Style

In the monolithic style all possible sequences of actions are presented explicitly, and thus the specification appears as a tree of choices. In this style, the specification shows explicitly the temporal ordering of actions.

This style allows only sequential composition, choice, guards, and recursion constructs, and thus prohibits the parallel composition of behaviour expressions. Therefore, it is useful for debugging specifications and generating test sequences. Milner's expansion theorem [Mil80], asserts that any LOTOS specification can be transformed into a monolithic style specification. Although the expansion may not terminate, it can yield finite initial subtrees of the full expansion of the specification.

Although the monolithic style is very useful for the design of simple specifications, it is of little practical use for the design of more complex ones because of its lack of structure, which causes difficulty for human comprehension.

2.5.3 Constraint-Oriented Style

The constraint-oriented style is the most popular for specifications that are wanted to be implementation-independent. In this style, only observable interactions are defined by a conjunction of separate constraints.

This style allows modularity and a logical structuring of the specification. It also makes the design of the specification flexible, that is, easy to extend and modify.

However, this style may make specifications hard to implement or to understand.

2.5.4 State-Oriented Style

In the state-oriented specification style, explicit system states are identified by using state variables. This style is analogous to the monolithic style in the presentation of dynamic behaviour of a specification, and therefore is not suitable for specifications where abstractness is needed. However, this style is useful for specifications where the informal definition uses the state concept, that is, in these cases, the state-oriented style increases readability. This style is also useful for specifications that are wanted to be relatively close to an implementation.

2.5.5 Resource-Oriented Style

In the resource-oriented style, the system is described by processes that represent different resources. The resources interact among themselves through interfaces. Each resource is defined by a temporal ordering of both internal and external interactions. Interactions among internal modules are hidden.

This style allows modularity and parallel structures. Therefore, it is useful for implementation specifications.

Chapter 3 The Design of Telephone Systems

Section 3.1 An Overview of Telecommunication Systems

3.1.1 Introduction

Telecommunications are among the most complex applications for hardware and software systems. Telecommunication applications have evolved from two main technological fields: the computer field and the communication field [Spr91]. *Telecommunication Systems* are one particular form of *Distributed Systems* because of their distributed applications. *Entities* on different sites use *Interfaces* to communicate with each other. The set of *Interfaces* involved in the communication process within a *Telecommunication System* is called an *Interaction System*.

In the next sections we briefly discuss the concepts of telecommunication system structures, as derived from standard references such as [VSA+89][Spr91][Tan89][Mar72][COR81]. First we give an overview of the concepts of *Distributed Systems* and structures of a telecommunication system. Then, a description of the methodology used throughout this work to design a *Telecommunication System* is given. The resulting structure of the *Telecommunication System* will be the subject of a formal description using the FDT language LOTOS in Chapter 4.

3.1.2 Distributed Systems

The most abstract view of a *System* is that of a black box, that is, it provides its users with a service in terms of their interactions with the *System*, see Figure 1. In this view, users should not be interested in the internal structure of a *System* but, instead, all the focus will be on the input/output of the interface functions between a *System* and its environment.

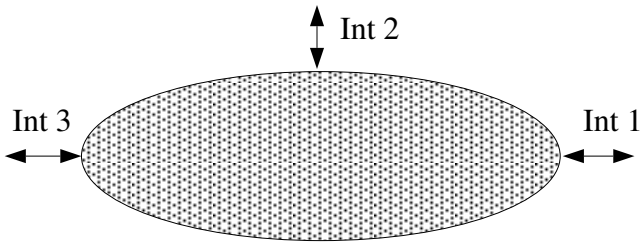


Figure 1. The Abstract View of a Distributed System

A *System* is thus a group of interacting items forming a unified whole. It consists of a set of interacting *Subsystems*. That is, a *Subsystem* represents an item of a *System* whose *Function* is a *Subfunction* of the overall *System*'s *Function* .

In order to communicate with each other, entities (*Subsystems*) on different locations must be interconnected. The set of all the interconnected entities along with the communication medias form a *System*. Such entities are said to be *embedded* in the *system*. The resulting system is called a *Distributed System*.

A *Distributed System* is thus, a collection of parts that interact like a single *System* with the environment. In other words, the users are not aware of the existence of multiple independent parts. Each part performs a specific *Function*, for example, handling a *Connection*.

A *Distributed System* is also characterized by a partitioned presentation of its *Function*, which appears to consist of a collection of geographically separate embedded *Subfunctions*.

For each *Distributed System* one can distinguish two types of *System* requirements: 1) *System* requirements that are applied on the execution of *Subfunctions* at each embedded *System* considered in isolation (Local), and 2) *System* requirements that are applied on the execution of *Subfunctions* at each embedded *System* depending upon execution of *Subfunctions* at other embedded *Systems* (Remote or End-to-End).

Therefore, a first refinement of a *Distributed System* results in a set of interacting *Systems* and further decompositions of the overall system will be applied on the embedded *Systems*, see Figure 2.

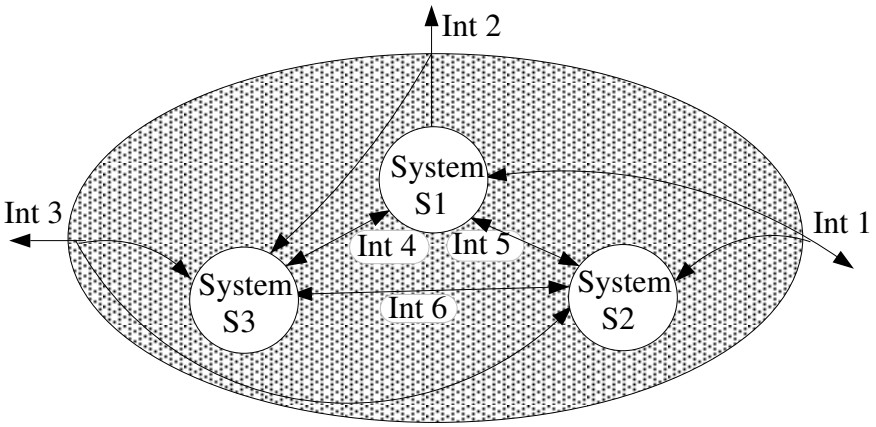


Figure 2. Refinement of a Distributed System

A *System* can itself be an embedded *System* within a *Distributed System*, that provides users[2] and the other embedded *Systems* with a service in terms of its interactions with them. It may also be further decomposed leading to a set of embedded *Subsystems*, see Figure 3.

This process of refinement of a *System* may be repeated on the resulting *Subsystems* until no further decomposition is possible. The resulting *Functions* give a description of the behaviour of the original *System*'s parts. Therefore, a design methodology of repeated decompositions provides a means for achieving a clear and readable (final) representation of a *Distributed System*.

[2] Users here are part of the environment of the initial *System*.

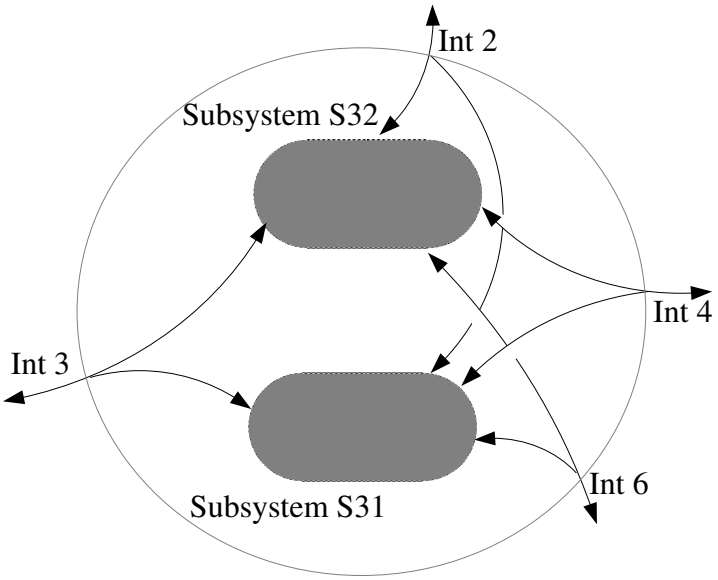


Figure 3. Refinement of an Embedded System (s3)

The *Functions* of the resulting parts are grouped together to give a description of the *Function* of an embedded *System*. The resulting *Functions* of embedded *Systems* are also unified together and the process of grouping together the *Functions* is repeated until the description of the *Function* of the global system (*Distributed System*) is achieved.

3.1.3 Structure of a Telecommunication System

In fact, one may distinguish three different functional structures for *Distributed Systems*. These are the centralized structure, the decentralized structure and the distributed centralized structure.

3.1.3.1 The Centralized Structure In this structure, all the entities of a system are connected to the same central processor. Let's call it a *Handler*. Information exchange is handled by this processor; it is sent from the transmitting entity to the handler which is in its turn sends it to the destination entity.

In *Telecommunication Systems*, this type of structure corresponds to the local switching network, called the *End Office*. All local lines are connected to the same *End Office*. To be able to communicate with each other, an entity first sends

a request to the *End Office*. The *End Office* then searches for the destination entity and establishes a line between the communicating entities, as shown in Figure 4.

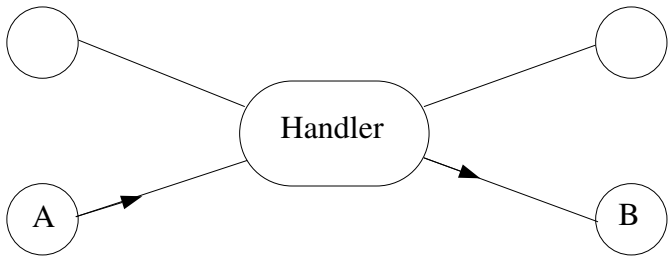


Figure 4. A Centralized Structure of a System

3.1.3.2 The Decentralized Structure In this structure, similar to what existed in early age telephone systems, all entities in the *System* are strongly connected to each other and there is no need for a handler. This structure is represented by an undirected and strongly connected graph, as shown in Figure 5. Therefore, this type of structure is not suitable for large *Distributed Systems*.

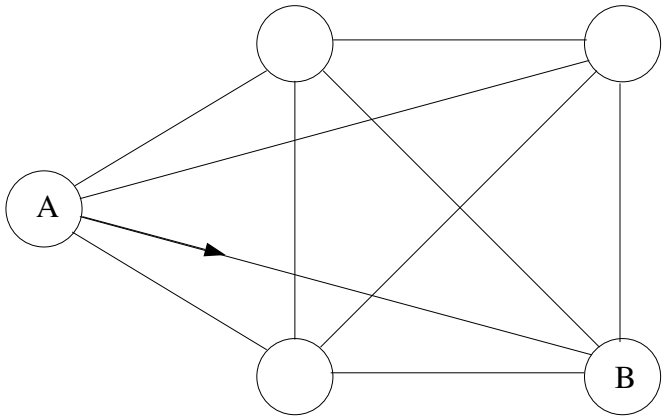


Figure 5. A Decentralized Structure of a System

3.1.3.3 The Distributed Centralized Structure In this structure, a *Telecommunication System* is viewed as a *Distributed System*. The *Subsystems* of this *Distributed System* are connected to each other. Each *Subsystem* consists of a set of embedded systems in a centralized structure.

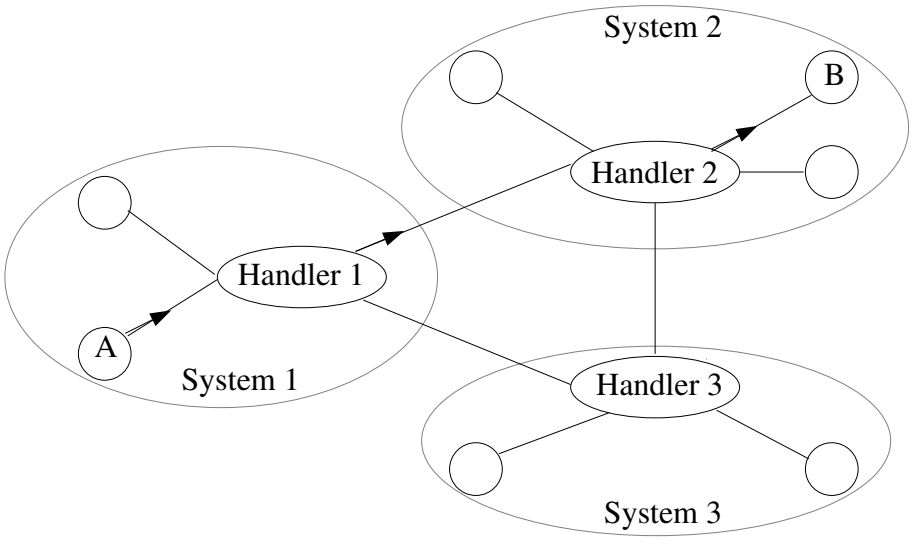


Figure 6. A Distributed Centralized Structure of a System

Therefore, this structure is very suitable for large systems and where distance is concerned.

3.1.4 Design of a Telephone System

In the design of complex systems such as *Telecommunication Systems*, it is generally hard and possibly impractical to deal with all the concerns of the *System* in a single design step. Rather, a design should be applied on concerns in the well known "stepwise refinement" way, as discussed in Section 3.1.2.

Each decomposition of the *System* gives a new step in the design process, which is guided by the criterion of separating design concerns.

Therefore, for a successful design of a complex *System*, structuring methods are the basic conceptual tools that the *System* designer must possess.

Telephone Systems are the very well known means of communication between entities on different sites. This type of *Systems* is dominated by use of *Connections*. Therefore, such a *Telephone System* may be modeled using a *Connection-Oriented* approach. This means that at connection time a communication path is established between the users involved, path which will be used for information transfer until disconnection. To talk to someone, a user picks up the handset of his telephone, dials the other side's telephone number, talks to him, and then hangs up his telephone. Similarly, to use a *Connection-Oriented Network* service, the service user first initiates a *Connection*, establishes the *Connection*, uses the same *Connection*, and then terminates this *Connection*.

3.1.4.1 The Integrated Perspective of a Telephone System In general there is a multiplicity of users around a *Telephone System* (TS), the totality of them is referred to as the "*System's Environment* (SE)". A *Telephone System* consists of a *Communication Service* (CS) that is interacting (I) with several connections being handled simultaneously, see Figure 8. A *Connection*, however, is a local processor that handles only one call at a time.

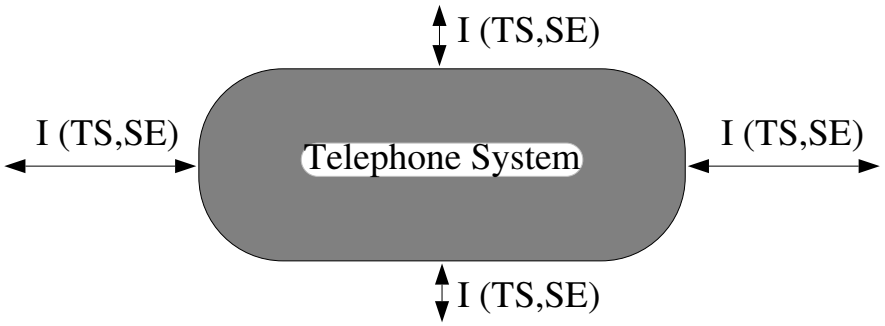


Figure 7. The Telephone System, an Integrated Perspective

Definitions;

I : Interaction.

SE : System's Environment.

TS : Telephone System

I(TS,SE) : Interaction of the Telephone System "TS" with the System's Environment "SE".

The integrated perspective of a *System*, Figure 7, is the initial view of the *System*'s design, when only the purpose of the design in known.

3.1.4.2 The Communication Service The *Communication Service* is the processing unit of a *Telephone System*. It is responsible for maintaining connections between communicating entities on different sites, regardless of the distance between entities.

The *Communication Service* interacts with all the entities and is capable of keeping track of all the existing lines, and of their status. Furthermore, it replies to queries from different entities by sending back appropriate signals and establishing connections. Once a connection is released, the *Communication Service* disconnects all the lines involved in it.

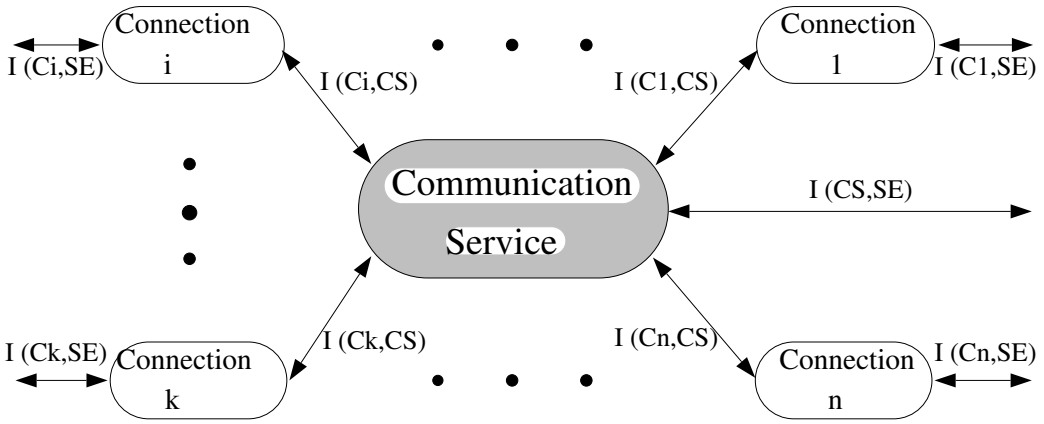


Figure 8. A Snapshot of a Telephone System Configuration

Definitions;

I (C_n,CS) : Interaction of Connection "n" with the Communication Service "CS".

I (C_n,SE) : Interaction of Connection "n" with the Telephone System's Environment "SE".

The Environment of a Connection "C_n" is the Telephone System's Environment "SE".

I (CS,SE)[3]: Interaction of the Communication Service "CS" with the Telephone System's Environment "SE".

[3] In Figure 7, "TS" represents either a Connection "Cn" or the Communication Service "CS".

The Environment of the Communication Service “CS” is the set of Connections and the Telephone System’s Environment “SE”.

3.1.4.3 Structure of a Connection A *Connection* consists of the *Transmitter* that initiates a call and possibly of a *Receiver* if a call terminates successfully to its destination. Once a *Connection* is established, many other subscribers may be added to the same *Connection*, see Figure 9. This is the case when a participating subscriber invokes a feature to call a new subscriber. The communication *scenario* between subscribers in a single connection is controlled by a local processor, called *Connection Handler*.

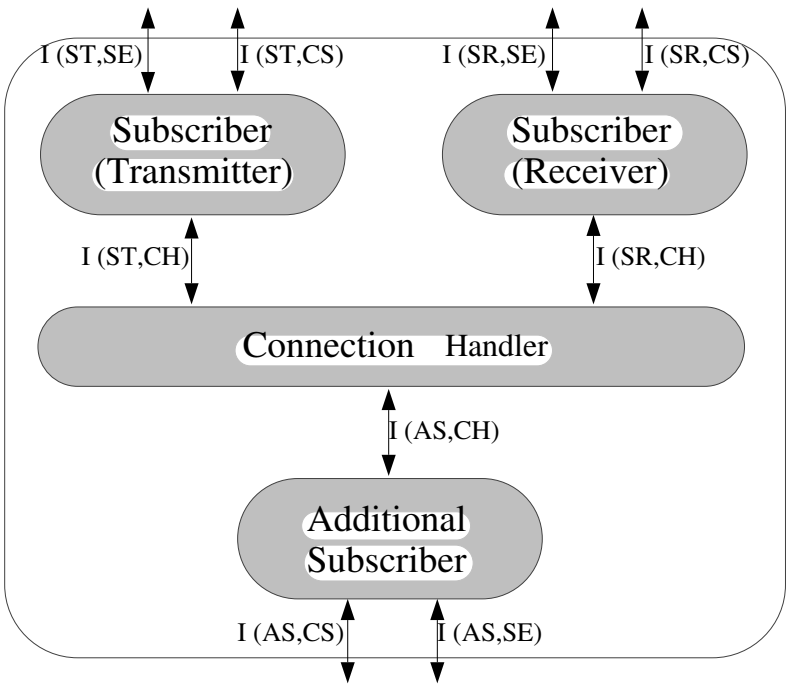


Figure 9. Decomposition of a Single Connection

All these components will be described in detail when presenting a *Sample Telephone System* in Section 3.2.3.

Definitions;

ST : A Subscriber as the Transmitter.

SR : A Subscriber as the Receiver.

CH : Connection Handler (in a Single Connection).

I (ST,CH) : Interaction of the Transmitter "ST" with the Connection Handler "CH".

I (SR,CH) : Interaction of the Receiver "SR" with the Connection Handler "CH".

I (AS,CH) : Interaction of an Additional Subscriber "AS" with the Connection Handler "CH".

I (ST,SE) : Interaction of the Transmitter "ST" with the Environment "SE".

I (SR,SE) : Interaction of the Receiver "SR" with the Environment "SE".

I (AS,SE) : Interaction of the Additional Subscriber "AS" with the Environment "SE".

Note 1 : An interaction of a Connection with other components results from the interactions of subscribers participating in the same Connection with those components.

Note 2 : ST, SR, and AS in Figure 9 correspond to Ci in Figure 8.

Section 3.2 Informal Description of a Sample Telephone System

3.2.1 Introduction

In this section, the design concepts introduced in Section 3.1.4 are employed to present the first steps in the design methodology for a *Sample Telephone System*.

First, we give a definition of the technical words introduced in the description of the sample model. Then, we present the structure of the *Sample Telephone System* along with an informal description of its functionality.

The sample model is subject to a formal description in the LOTOS language later in Chapter 4. Next, is an informal description of the telephone signals involved in the sample model and of an informal definition of some modern telephone features provided by this system along with their functionalities. It is followed by a description of the different phases of a call processing. We end this

section by giving an informal list of user requirements for our *Sample Telephone System*.

3.2.2 Technical Terminology

A *Connection* in this *Sample Telephone System* refers to the engagement of a *Subscriber* in an attempt to initiate a call. It consists of a *Subscriber* using its telephone to initiate a call denoted *Origination Side* and possibly a receiver of an incoming call to its telephone denoted *Destination Side*. The *Destination Side* may be involved in a *Connection* only if the *Origination Side* does not abort the call before its telephone becomes connected.

While in conversation, if any participating *Subscriber* requests a specific feature allowed by the *Telephone System*, some other *Subscriber(s)*, called *Additional Subscribers*, may be involved within the same *Connection*.

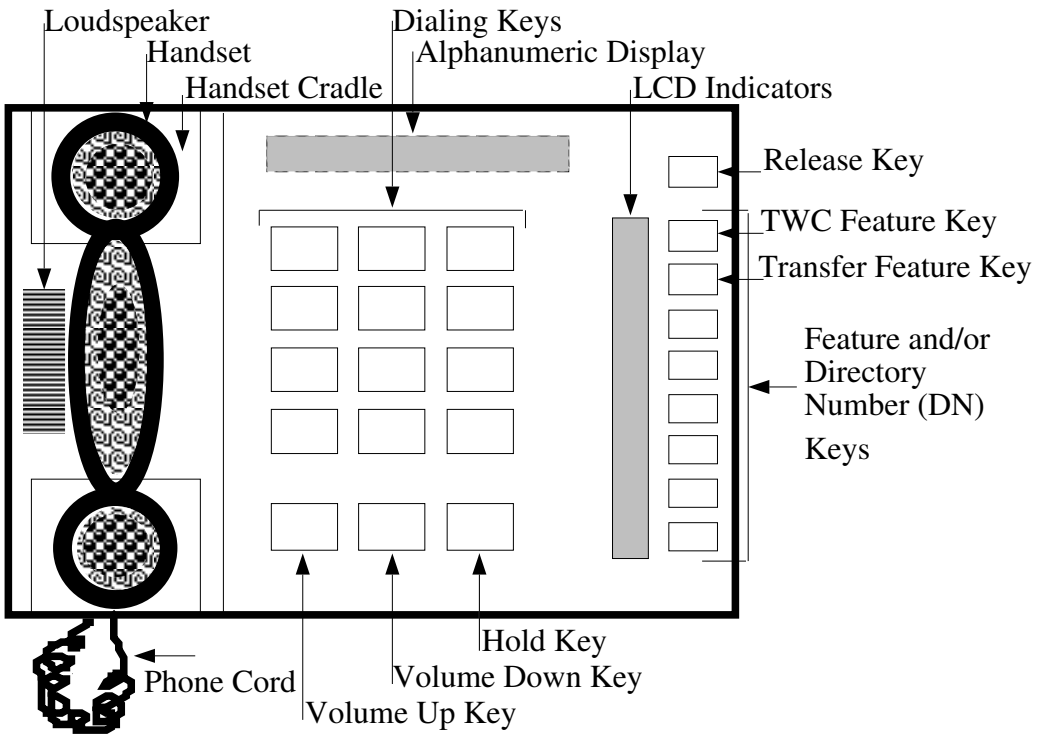


Figure 10. The DMS-100 Business Set

Each user owns a telephone which is known by an associated number, called *telephone number*. For each number there is a telephone, called *Main Extension*, and there may be other telephones, called *Extensions*, associated with the same number. A *line* is the logical connection of the telephone number to the *Telephone System*. A *telephone number* gives the identity of the line being used.

Since our goal is to design a *Telephone System* that allows for an arbitrary number of *Subscribers* to communicate with each other via their telephones within the same *Connection*, any *Subscriber* participating in this *Connection* has the possibility of invoking any feature among the ones available within this *Telephone System*. The sample *Telephone*, called *Business Set*, consists of a handset for user's communication and a keyboard containing *Dialing Keys* and some other additional keys, called *Directory Number (DN)* keys, used to invoke different modern features.

There are two basic features that are permanently assigned to the first two *DN* keys on the *Business Set*. These are *Three-Way Calling* and *Call Transfer* features. The *Business Set* may also have access to features that are not permanently assigned to *DN* keys. These features can be activated by pressing the *Three-Way Calling* or *Call Transfer* feature key and then dialing the appropriate feature code. The sample *Business Set* is shown in Figure 10 [Bel88]. In addition to *DN* keys, the sample *Business Set* offers to its users a set of other keys that allow them to process a call accordingly. These are mainly dialing keys and *Release* key. The latter allows a user to leave or abort a call without hanging up the handset of his telephone.

While in conversation, a participating *Subscriber* may decide to invoke a specific feature available on its telephone. To do so, this *Subscriber* must first press a *Directory Number (DN)* key that is in use (*idle*) then proceeds to activate that feature, as specified above. The selected *DN* key will be occupied during the use of the feature. So, the *DN* keys allow *Subscribers* to invoke modern features that are allowed by the *Telephone System*. This sequence of operations will be taken as being as an atomic action of feature invocation in the rest of this thesis.

An initiator of a call or an invoker of a feature may get any one of different types of tones, called *Audio Services*, depending on the status of the line being requested. A *Status Test* on a line refers to the process of checking that line and

providing an appropriate signal to the sender of the request.

A *Connection Handler* within a *Connection* is the local controller of signal exchanges between telephones and the *System Network*. It controls the logical sequencing of signal exchanges between the *System Network* and an arbitrary number of *Subscribers* involved in a single *Connection*.

However, a *System Network* is the switching device that gets requests from telephones involved in different *Connections* being handled in parallel and sends back to them appropriate signals. This is achieved by making some status tests on the requested lines as discussed above. A requested line may be in use within some *Connection* (*Busy*), not connected to the *Telephone System* (*Out Of Service*) or free for use (*Idle*). The *System Network* checks also whether a call terminates to this line or it is to be forwarded to another line. In the latter case, all further status tests are applied on the line to whom incoming calls have been forwarded.

3.2.3 Structure of the Sample Telephone System

3.2.3.1 Introduction As our objective in this work is to show that telephone systems can be formally specified in LOTOS, the *Sample Telephone System* presented in this section corresponds to a telephone service where time-out concerns are ignored. This is due to the fact that the standard FDT language LOTOS cannot represent the time faithfully, although some research has been done in this area and proposals exist [QF87][Bri88a].

In this section, we informally describe in detail the structure of our *Sample Telephone System* along with the functionality of its different components and of the overall system.

First, the general structure of the *Sample Telephone System* is presented, then each of its components is described along with the role that it plays in the functioning of the overall system. The relation between each component and the remaining components within the *Telephone System* is also described in this section.

3.2.3.2 An Abstract View

From the user's perspective, a *Telephone System* is viewed as a black box that is interacting with its *Environment*. The general structure of the *Sample Telephone System* is presented in Figure 11. This structure is similar to the one presented in Section 3.1.4.1, except that here we show the *System*'s *Environment* that consists of an arbitrary number of *Subscribers* connected to a particular service provider called *Telephone System*. The *Subscribers* use their telephones to communicate with each other.

We note here that the dots in the figure shown below indicate the existence of some *Subscribers* whose lines are not connected to the *Telephone System* but they may want to use this service provider to communicate with other *Subscribers*. This is explained in detail later in this section.

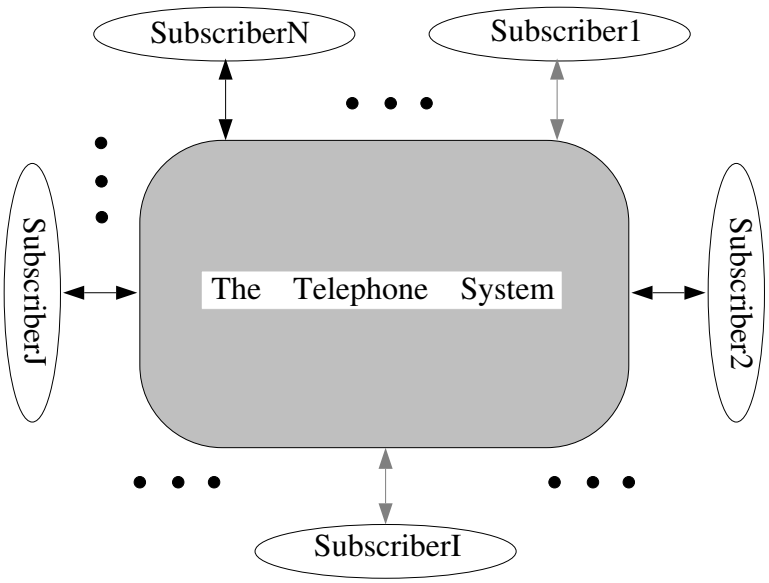


Figure 11. An Abstract View of the Sample Telephone System

Each *Subscriber* is connected to the *Telephone System* by a single *line* that can be used either to initiate outgoing calls or to answer incoming calls. A *Subscriber* represents a single component that may interact with other *Subscribers* via the *Telephone System* by using its telephone. However, the *Telephone System* consists of different components needed for handling the communication between different *Subscribers*.

Note that the structure of the *Sample Telephone System* is dynamic. This is due to the fact that at a given time, the *Telephone System* consists of an arbitrary number of *Connections* (calls) being handled in parallel. A snapshot of a typical configuration the *Telephone System* is shown in Figure 8. The figure shows a set of *Connections* being handled simultaneously by the *Telephone System*. All these *Connections* must synchronize with the *Communication Service* on signal exchanges. In the rest of this thesis the *System Network* will denote the *Communication Service*.

Therefore, the *System Network* plays the role of the medium between different *Connections*. It is capable of keeping track of the lines in service, of the lines being used by the *Subscribers*, and of the lines to whom some incoming calls are to be forwarded. We note here that each *Connection* is handled separately. As already mentioned, the environment of the *System Network* at a given time is the set of *Connections* being processed within the *Telephone System* and the overall *Telephone System*'s environment.

The description of each component of the *Telephone System* will be presented later in the next sections.

3.2.3.3 Structure of a Single Connection A *Connection* in our model refers to the engagement of a *Subscriber* in order to initiate a new call. It lasts until the caller aborts the call or the communication path between the participating subscribers is released. Such a *Connection* may also terminate if the *line* of a *Subscriber*, as the initiator of a call, is busy[4] or its line is not connected to the *Telephone System*.

Therefore, a *Connection* in this context requires the participation of at least one *Subscriber*. This is the case when a call is aborted before establishment of the communication path.

The number of *Connections* being handled in parallel at a given time is equal to the number of subscribers who have initiated new calls but have not terminated it yet. Note that no new call is created when a participating *Subscriber* within a *Connection* consults privately with an *Additional Subscriber*[5]. The structure

[4] The line of a caller is busy, if it is being used by an *Extension* within another *Connection*.

[5] This may occur when a *Subscriber* invokes a feature to call an *Additional Subscriber* while it is connected to another *Connection*.

of a single *Connection* is presented in Figure 9, Section 3.1.4.3. In this figure, we show the interactions of a component within a *Connection* with the other components and the environment.

A *Connection* is thus composed of at least an initiator of the call, called *Transmitter*, and a local controller associated with the call, called *Connection Handler*. A responder to the call, called *Receiver*, may be added to this *Connection* if the current call terminates successfully to its line. Some other subscribers, called *Additional Subscribers*, may also be added to the same *Connection*. These *Additional Subscribers* result from invoking some features by either the *Transmitter*, the *Receiver* or any other participating subscriber in the current *Connection*[6].

3.2.3.3.1 The Transmitter The *Transmitter* is the basic component that provokes the creation of a new *Connection*. When a subscriber, the *Transmitter*, picks up the handset of its telephone to initiate a call, a new *Connection* is automatically created within the *Telephone System*[7].

Furthermore, when the *Transmitter* picks up the handset of its telephone to initiate a call, it will be immediately identified by the *System Network* that sends back to him a special *Audio Service*. The type of the *Audio Service* that the *Transmitter* can get depends on the status of its line at that time. Mainly, there are three types of *Audio Services* that may be applied on the line of the initiator of a new call.

1.1. The line of the *Transmitter* is being used by an *Extension* on the same line and therefore the *Transmitter* gets a *Busy* tone. This is presented in Figure 12 below.

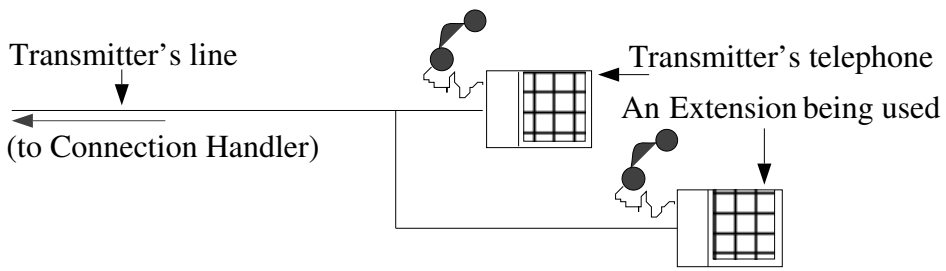


Figure 12. A Busy Line

[6] *Transmitter* and *Receiver* denote the *Origination Side* and the *Destination Side* respectively.

[7] This is not the case when a subscriber picks up the handset of its telephone to answer an incoming call to its line.

In real world, this is not a *Busy* tone but the *Transmitter* can recognize the busy status of its line. In our model, we consider this situation like a *Busy* tone received by the *Transmitter* from the *System Network* since the former cannot initiate a call while an *Extension* of its telephone is being used.

1.2. The line of the *Transmitter* is not in service and therefore it cannot be used to either initiate new calls within the *Telephone System* nor to receive incoming calls to its associated telephone(s). Therefore, when it picks up the handset of its telephone to initiate a new call, the *Transmitter* gets an *Out Of Service* tone.

In real world, such a telephone is not connected to the *Telephone System*. Therefore, when a *Subscriber*, as a *Transmitter*, picks up the handset of such a telephone, it cannot get any tone but the *Transmitter* can realize that this line is not connected to the *Telephone System*. In our model, we take such a situation as an *Out Of Service* tone sent by the *System Network* to the *Transmitter* on that line.

1.3. The line of the *Transmitter* is connected to the *Telephone System* and free for use (*Idle*) for either incoming or outgoing calls, and therefore when it picks up the handset of its telephone to initiate a new call, the *Transmitter* gets a *dial* tone. The *Transmitter* is now ready to dial the *Receiver*'s telephone number, see Figure 13 below.

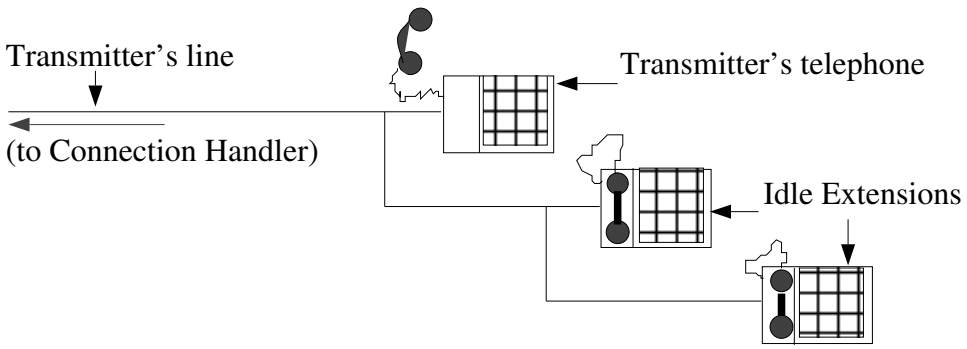


Figure 13. A Free Line

Then, after dialing the *Receiver*'s telephone number and on termination of the current call to the other side, the *Transmitter* gets an *Audio Service*. The type of *Audio Service* that the *Transmitter* can get depends on the status of the line of the *Receiver*. This can be summarized by the following statements:

2.1. The line of the *Receiver* is being used by another *Connection*, and therefore the *System Network* sends back to the *Transmitter* a *Busy* tone. Then the latter must *Hang Up* its telephone to be disconnected and be available within the *Telephone System*.

As we will see in the next sections, when the *Transmitter* gets a *Busy* tone, it may invoke a *Ring Again* feature and therefore it will be notified as soon as the other side becomes available (*Idle*).

2.2. The *Receiver*'s line specified by the *Transmitter* is not connected to the *Telephone System*, and therefore the *System Network* sends back to the *Transmitter* an *Out Of Service* tone. The *Transmitter* then must also *Hang Up* its telephone to be disconnected and be available (*Idle*) within the *Telephone System*.

2.3. The line of the *Receiver* exists and it is *Idle*, and therefore the *System Network* sends back to the *Transmitter* an *Audible Ring* tone. The *Receiver*'s telephone is now ringing. The *Transmitter* may then wait for an answer from the *Receiver* or abort the call and both telephones are disconnected.

3.2.3.3.2 The Receiver A *Receiver* is the subscriber to whom a call is destined. It may also be the subscriber to whom a call has been forwarded by any other subscriber to whom the current call is supposed to be destined.

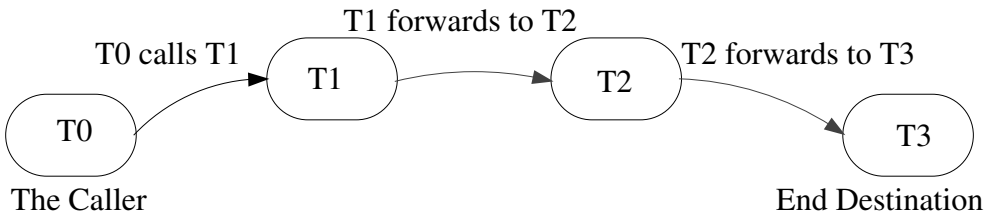
Depending on the status of its line, the *Receiver* may not be involved in a *Connection*. This is the case when an incoming call to its telephone is aborted by its *Transmitter* before an *OffHook* from the *Receiver* is detected.

If a *Connection* has successfully been terminated to its line, the *Receiver* then will have the same priority as the *Transmitter* to invoke any feature among the ones provided by the *Telephone System* and that fulfills the requirements related to the functionality of that feature, see Section 3.2.7.2.

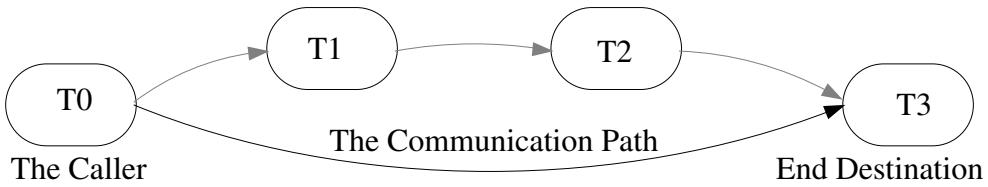
Although the *Receiver* is the subscriber that answers a call, it may also be the first participating subscriber to terminate it. Therefore, during a conversation any participating subscriber may leave it for good or invoke any feature allowed by the *Telephone System*.

3.2.3.3.3 The Connection Handler For each newly created *Connection*, there is an associated *Connection Handler*. The *Connection Handler* plays the role of

the manager of different scenarios between the telephones that are communicating with each other within the same *Connection*. Furthermore, the *Connection Handler* uses a temporary set of line identities whenever the current call is forwarded to another pre-selected line. This set contains the identities of all the lines that have forwarded this call and will be kept busy until establishment of the communication path between the caller and the end destination, or until the call is aborted before its termination to the latter, (1) of Figure 14.



1. A Call Before its Termination to End Destination



2. Real Connection Between the Caller and the End Destination

Figure 14. Forwarding a Call

Whenever the communication path between the caller and the end destination is established, (2) of Figure 14, all the lines (T1 and T2) that have forwarded this call are disconnected.

Some of the communication scenarios that may be found in a simple call are presented in this section by using the formalism of *evaluation networks*. Later on we shall see how LOTOS can represent the same scenarios, as well as much more complicated ones, in a much more complete way.

The graphs presented in Figures 16, 17 and 18 below represent three different communication scenarios between subscribers and the *Telephone System* within a simple call. These scenarios handle only the basic telephone communication between two subscribers where there is no request for any feature. We assume also that only *Idle* telephones may be involved in this sample call[8].

An *evaluation network* is a directed graph with three different types of nodes [COR81], see Figure 15.

– states, represented by a circle,
– queries, represented by a rectangle,
– and transitions, represented by a horizontal bar.

A sample graph is,

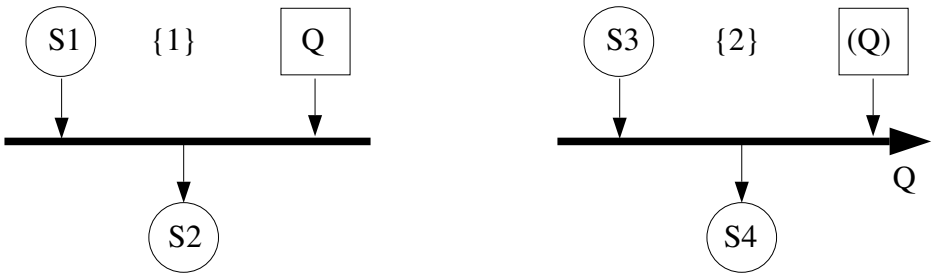


Figure 15. A Sample Evaluation Network

In graph {1}, the (sub) system is currently in state S1. When a query Q arrives from the other side it creates a transition that changes the system from state S1 to state S2. In graph {2}, the system is currently in state S3. Whenever it sends a query Q, a new transition is created and will change the state of the system from S3 to S4.

Following is the description of the evaluation graphs for some communication scenarios between *Subscribers* and the *System Network* that may exist in a simple *Connection*.

The first graph (Figure 16) represents a communication scenario between the *Transmitter* of a call and the *Telephone System*, while the second graph (Figure

[8] No status test is applied on any line.

17) represents a communication scenario between the *Receiver* of a call and the *Telephone System*.

However, the last graph represents a communication scenario between either subscriber, the *Transmitter* or the *Receiver* of a call, and the *Telephone System* when attempting to terminate a call.

First, we give a definition of some notations used to identify the states and queries that represent the nodes of these graphs:

Queries

```
Ofh    : Other side goes OffHook
(Ofh)  : This side goes OffHook
Onh    : Other side goes OnHook
(Onh)  : This side goes OnHook
Dial   : Other side dials number
(Dial) : This side dials number
```

States

```
Idle  : Idle state "OnHook"
Rin   : Transmitter's ring
Rout  : Receiver's ring
Est   : Connection established
Tone  : Gets dial tone
```

1. The Transmitter Side scenario Initially, the *Transmitter*'s telephone is in an *Idle* state. As soon as it picks up the handset of its telephone to initiate an outgoing call, *(Ofh)*, the *Transmitter* gets a dial tone, *Tone*.

The *Transmitter* may now start dialing the *Receiver*'s telephone number, *(Dial)*, or decide to abort the call, *(Onh)*. In the latter situation the *Connection Handler* changes the state of the *Transmitter* from *Tone* to *Idle* again and terminates the call, while in the former one, the *Connection Handler* changes the state of the *Transmitter* from *Tone* to *Rout* (gets an *Audible Ring*). The *Receiver*'s telephone is now ringing.

Then, if the *Transmitter* decides to go *OnHook*, *(Onh)*, before the *Receiver*

answers the call, *Ofh*, the *Connection Handler* changes its state to *Idle* again and terminates the call. However, if the *Receiver* answers the call, *Ofh*, before an *OnHook*, *(Onh)*, from the *Transmitter* is detected, the *Connection Handler* changes the state of the *Transmitter* to *Est*.

At this stage the communication path between both subscribers is established from the *Transmitter*'s side.

The corresponding communication scenario between the *Transmitter*'s side and the *Telephone System* is given by the evaluation graph shown in Figure 16 below.

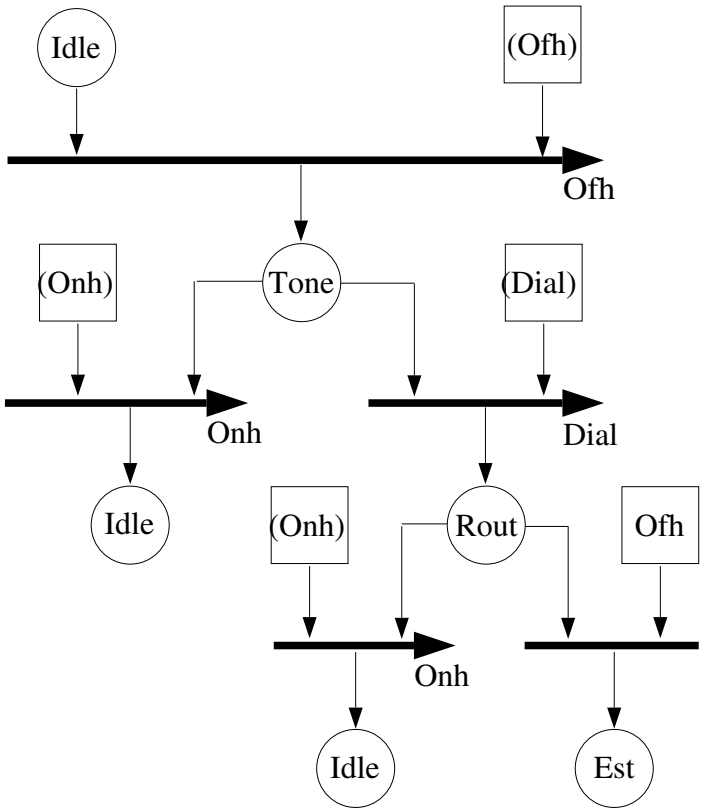


Figure 16. An Evaluation Graph for the *Transmitter*'s Communication Scenario

2. The Receiver Side scenario Initially, the *Receiver*'s telephone is in an *Idle* state. When its telephone number is dialed, *Dial*, by any other side (the *Transmitter*), the *Connection Handler* changes its state to *Rin*. The *Receiver*'s telephone is now ringing.

Meanwhile, the *Receiver* may go *OffHook*, *(Ofh)*, and the communication path between the *Receiver* and the *Transmitter* is established, *Est*, from its side or the call is aborted by the *Transmitter*, *Onh*, and the state of the *Receiver* is changed to *Idle* again.

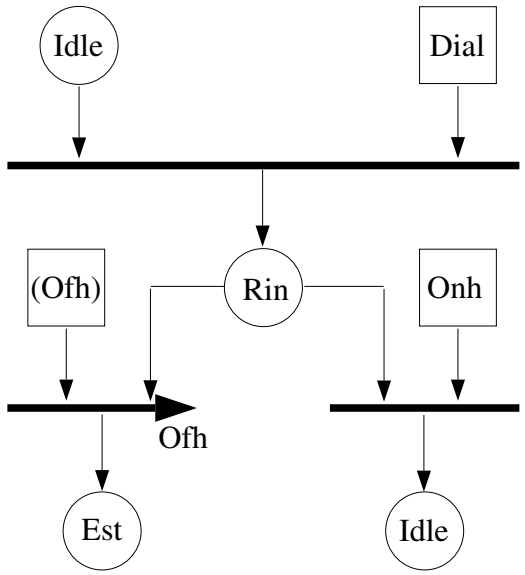


Figure 17. An Evaluation Graph for the *Receiver*'s Communication Scenario

With *Idle* state, the *Receiver* may play the role of the *Transmitter* in a new call or answers any further incoming call to its telephone.

The corresponding communication scenario between the *Receiver*'s side and the *Telephone System* is given by the evaluation graph presented in Figure 17 above.

3. The Disconnection Scenarios: While connected to each other, *Est*, either subscriber[9] (the *Transmitter* or the *Receiver*) may go *OnHook*, *(Onh)*, and its

[9] This communication scenario handles one side at a time.

state is changed to *Idle* again. Any subscriber with a new state *Idle* is now ready to initiate a new call or receive any incoming call to its telephone.

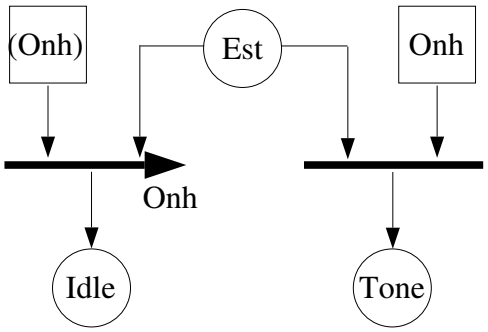


Figure 18. An Evaluation Graph for the *Disconnection*'s Communication Scenario

If instead, an *OnHook*, *Onh*, is detected from the other side while this side is still *OffHook*, the latter will then get a *Dial* tone, *Tone*, and therefore it is ready now to only initiate a new call (as a *Transmitter*).

The corresponding communication scenario between either subscriber's side and the *Telephone System* is given by the evaluation graph presented in Figure 18 above.

3.2.3.3.4 The Additional Subscribers At the initial stage, only the *Transmitter* and the *Receiver* are involved in a *Connection*. However, an *Additional Subscriber* may be involved in a *Connection* if either the *Transmitter* or the *Receiver* decides to call them within the same *Connection*. An *Additional Subscriber* that is participating in a conversation may also add other *Additional Subscribers* to join them within the same *Connection*.

To do so, a participating *Subscriber* must invoke an appropriate feature, provided it is available in the *Business Set* being used and then calls a new *Subscriber* without leaving the current conversation.

Once a new *Subscriber* is added to a *Connection*, it will play the same role as any other participating *Subscriber* within this *Connection*. Therefore, a new *Subscriber* may leave for good the conversation , invoke any (possible) feature or add any other *Additional Subscriber* to the same *Connection*.

3.2.3.4 The System Network In our model, the *System Network* is the global handler of message exchanges between the *Telephone System* and its *Environment*. The *Environment* represents all the *Subscribers* that may use this *Telephone System* to communicate with each other.

Therefore, the *System Network* controls all the queries sent by *Subscribers* participating in different *Connections* and replies to these queries by appropriate signals. It has access to a database containing the set of identities of all the telephones that are connected to the *Telephone System* and the set of identities of all the telephones that are currently being used by *Subscribers (busy)*.

The database may also contain the list of identities of the telephones that have forwarded incoming calls to their telephones to other pre-selected telephones and of the telephones to whom these calls were forwarded. The *System Network* may also have access to temporary sets of identities being used by *Connection Handlers*. Each *Connection Handler* uses a temporary set to keep track of the lines that have forwarded the current call to another line, until the end destination is reached. This situation is explained in Sections 3.2.3.3.3 and 3.2.5.3.

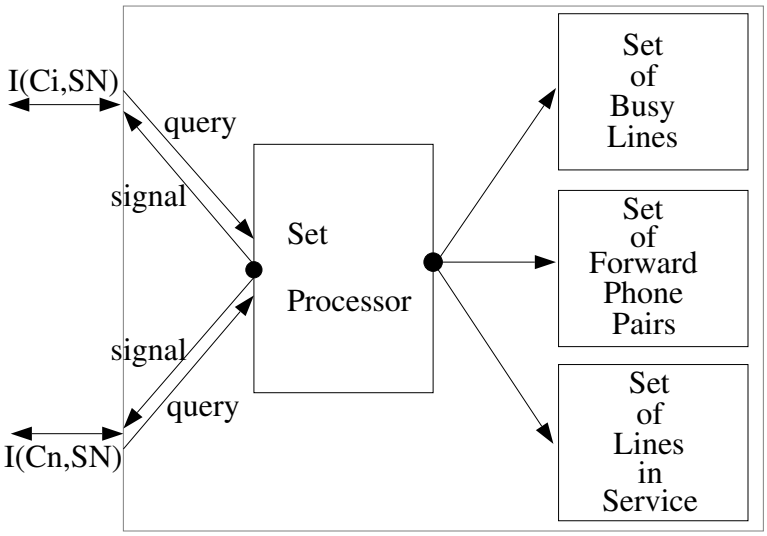


Figure 19. Structure of the System Network

Note: I(Ci,SN) denotes the interaction between a connection "i" being processed and the *System Network*.

When a *Forward* feature is invoked, the *System Network* adds to the set of *ForwardPhonePairs* the pair consisting of the invoker of the *Forward* feature and the telephone to whom incoming calls are to be forwarded. This pair is removed from the set whenever the *Forward* feature is cancelled from the same telephone.

Depending on the type of a request sent by a *Subscriber*, the *System Network* checks the status of the associated line on that telephone and then sends back to that *Subscriber* an appropriate signal.

As soon as a line is released by a Subscriber, the *System Network* makes it available *(free)* for use by any *Subscriber*. It removes the released line from the associated set.

The structure of the *System Network* is presented in Figure 19 and a snapshot of a *Telephone System* configuration is presented in Figure 8, Section 3.1.4.2. Such a configuration may take place at any time during the functioning of the *Telephone System* and consists of a set of *Connections* being processed simultaneously and a special manager referred to as the *System Network* that controls the lines requested by *Subscribers* from within these *Connections*.

3.2.4 Definition of Telephone Signals

3.2.4.1 Introduction Telephone systems are mainly governed by use of signals. Among these telephone signals, some signals can be distinguished for their special use by the handler of connections.

In our *Sample Telephone System*, six signals are of special interest. These are *Busy*, *Out Of Service*, *Dial*, *Ring*, *Disconnect* and *Forward* signals. We refer to these signals as *Control Signals*, since they are used by the *System Network* to control the lines requested by different connections and to be able to reply to a query by an appropriate signal. The remaining signals used in our model are of little interest and used only to fulfill the scenarios between *Subscribers* and the *System Network*.

Therefore, when a *Subscriber* requests a line, the *System Network* first checks its status and then provides its sender with an appropriate *Control Signal*, that is, a special tone heard by the *Subscriber*. A *Control Signal* represents the status of the requested line at that time.

A description of each of these *Control Signals* is given in the following sections.

3.2.4.2 Busy Signal A *Busy* signal may be detected by a *Subscriber* (1) when it picks up the handset of its telephone to originate a call while an extension of its telephone on the same line is currently being used[10], (2) after having called a line that is in a *Busy* state, (3) when attempting to call an *Additional Subscriber* from within a *Connection* and the *System Network* finds out that the requested line is *Busy*, and (4) if its call terminates to a line after it has been forwarded and where this line is among the ones that have forwarded it. These lines are still kept busy until establishment of the communication path.

3.2.4.3 Out Of Service Signal An *Out Of Service* signal is detected by a *Subscriber* whose line is not connected to the *Telephone System*, but who picks up the handset of its telephone in an attempt to initiate a call[11]. Such a signal may also be detected by a *Subscriber* after calling a line which is currently not connected to the *Telephone System*.

Furthermore, an *Out Of Service* signal is detected by a *Subscriber* when attempting to call an *Additional Subscriber* from within a *Connection* and the *System Network* finds out that the requested line is not connected to the *Telephone System*.

3.2.4.4 Dialing Signal A *Dial* signal is detected by a *Subscriber* when it picks up the handset of its telephone to initiate a new call. It is detected only by a telephone whose line is *Idle* (no other *Extension* on the same line is being used).

Therefore, when it goes *OffHook* to initiate a call, a *Subscriber* may get a *Dial* signal only if the status tests described above on its line have successfully been completed.

3.2.4.5 Ringing Signal *Ringing* should be applied to the terminating line if it is found to be *Idle*[12] and no *Forward* feature is in effect on that line. Following an *OffHook* from the terminating side, *Ringing* should be removed from its telephone.

---

[10] This is the case where there is more than one telephone connected to the same line.

[11] In real world, this is the case when a *Subscriber* picks up the handset but it gets no tone.

[12] The status tests 3.2.4.2 and 3.2.4.3 on that line have been successfully completed.

Ringing should also be removed from the telephone of the terminating side if the call is aborted[13] by its originator.

3.2.4.6 Disconnect Signal The communication path between the *Transmitter* and the *Receiver* should be removed as quickly as possible following initial detection of an *OnHook* from either side. The *Disconnect* signal is detected by the *System Network* from a line after an *OnHook* has been initiated by the *Subscriber* using that line.

3.2.4.7 Forward Signal This additional signal results from the use of a *Forward* feature by a *Subscriber* whose telephone is *Idle*.

When a call terminates to a line, the *System Network* checks first its status. If it finds that incoming calls to that line are to be forwarded to another line, no *Ringing* is applied to it but, instead, it sends back a *Forward* signal and then forwards that call to the pre-selected line. All further status tests are applied on the line to whom a call has been forwarded.

Note that even the line to whom a call was forwarded, may also have forwarded incoming calls to another pre-selected line and so on. The *Forward* feature remains in effect until it is canceled from the same line.

3.2.5 Features of the Sample Telephone System

3.2.5.1 Introduction The *Business Set* presented in Figure 10, is designed in such a way as to provide the *Subscribers* with a convenient means to access to a number of more sophisticated operations[14]. These are mainly, *Hold*, *Ring Again*, *Transfer / Three-Way Calling (TWC)*, *Forward* and *Conference* call features.

An informal description of each feature, together with its functionality, is presented in the following sections.

---

[13] *Disconnect* signal from the calling line is detected by the *System Network* before an *OffHook* from the terminating side.

[14] To activate other features than *Transfer* or *TWC* feature, press the *Transfer* or *TWC* feature key then dial the appropriate feature code.

3.2.5.2 Ring Again Feature When you initiate a call and you find out that the *Destination Side*'s line is *busy*, the *Ring Again* feature allows you to be notified when that line becomes free and will redial automatically the number for you.

To cancel this option simply press a second time the *Ring Again* feature key and the call will be released.

Functionality:

You *dial* a number and you get a *busy* signal (that *Subscriber* is already engaged in another call). Press the *Ring Again* feature key then *Hang Up* the telephone. When the *Destination Side*'s line becomes free you will hear a special ring (*Ring Again Notification (RANO)*) on your telephone. Lift the handset (*OffHook*) of your telephone and the call will be completed for you. You may, of course, cancel this feature at any time by pressing a second time the *Ring Again* feature key.

3.2.5.3 Forward Feature This feature allows you to forward incoming calls from your telephone to another pre-selected telephone.

Functionality:

Without lifting the handset of the telephone, press the *Forward* feature key and dial the telephone number to whom you want your further incoming calls to be forwarded to, then press the associated key again. While the *Forward* feature is in effect, the telephone will not ring, for any incoming call, until the feature is cancelled. This sequence of actions will be represented in our LOTOS specification by a single atomic action.

To cancel the *Forward* feature, press the associated feature key once and you will be notified whenever a call comes in to your telephone.

Note here that whenever a call is forwarded from one line to another pre-selected line, the former will be kept busy until the communication path between the caller and the end destination is established or the call is aborted before its termination. Note also that, if there is a cycle in forwarding a call, calling any number in the cycle will result in a *Busy* signal.

3.2.5.4 Hold Feature This feature allows you to place a new call from within a *Connection* while putting on hold the other *Subscriber* to whom you are connected.

Functionality:

You are talking to someone and you decide to call another *Subscriber* while maintaining the first one connected. Press the *Hold* feature key then press an *Idle DN* key and start dialing the required number for a new call. The other side is automatically put on hold and that *DN* key is no longer *Idle*.

To return to the first call (on hold) press the same *DN* key. This will automatically release the second call and re-connect you with the first one. To cancel this feature simply press the same *DN* key a second time.

3.2.5.5 Listen on Hold[15] If the *Subscriber* you are talking to places you on hold, you are able to hear through the telephone speaker when that *Subscriber* re-establishes the call.

Functionality:

You are talking to someone and he decides to put you on hold. Remain *OffHook* and as soon as this *Subscriber* becomes free from the other call you will be automatically re-connected with him. While on hold, you may terminate the call at any time.

3.2.5.6 Transfer/Three-Way Calling Feature This feature allows you to transfer incoming calls to a third *Subscriber* or to set up a *Three-Way Calling (TWC)* conversation. You may, of course, consult privately with the *Subscriber* to whom you are transferring the call then return to the first *Subscriber*.

Functionality:

i . This is useful if, for some purpose, you decide to transfer an incoming call to a third *Subscriber*: answer the call and then press the *Transfer* feature key. *Dial* the required number and after the third *Subscriber* goes *OffHook* you will be connected to a new call (privately), while the first *Subscriber* is still on hold. Press a second time the *Transfer* feature key to connect together both *Subscribers* and then leave the conversation (*HangUp*).

ii. To set up a TWC, simply complete step **i** but do not *HangUp* and join the conversation. Now all three of you are connected together in a *Three-Way Calling* conversation. You may go *OnHook* at any time and the other parties can continue the conversation.

[15] In fact, this is not a feature but a consequence of invoking the *Hold* feature by the other side.

To cancel this option, press the *Transfer* feature key before the connection has been established with the third *Subscriber*.

3.2.5.7 Conference Call Feature This feature allows you to set up a conversation between yourself and *N* other *Subscribers*. At any time during the *Conference* conversation any participating *Subscriber* can leave the conversation for good or consult privately with another party and then return to the *Conference*. Additional *Subscribers* may be added at any time to the *Conference* conversation.

Functionality:

You are in a conversation with someone and you decide to consult (privately) with another party or to add it to the conversation. Press the *Conference* key and dial the required telephone number. After that you may either consult privately with this party or you may bring him to the *Conference*, in which case you press the *Conference* key again. If the new party remains *OffHook*, you are all connected to the *Conference* conversation.

To leave (for good) the *Conference* conversation simply hang up your telephone.

3.2.6 Call Processing Phases

3.2.6.1 Introduction A *Connection* proceeds in three main phases that can be distinguished by the *Control Signals*. A *Connection* begins with the *Origination Phase* to end with either the *Termination Phase* or the *Disconnection Phase* depending on the status of the lines being used. This can be summarized by the flowchart presented in Figure 20 [Bel87].

Within a *Connection* a sequence of tasks, referred to as *Call Processing*, is performed. It starts with an *Idle* telephone and ends by making *Idle* all participating telephones. Any task can be performed if the preceding ones have been successfully completed.

The flowchart presented in Figure 20 shows a sequence of steps of a *Call Processing*. Note that this sequence may not be completely performed if a call is aborted by the calling line or fails when using a line. This depends on the status of the line(s) being used by that call.

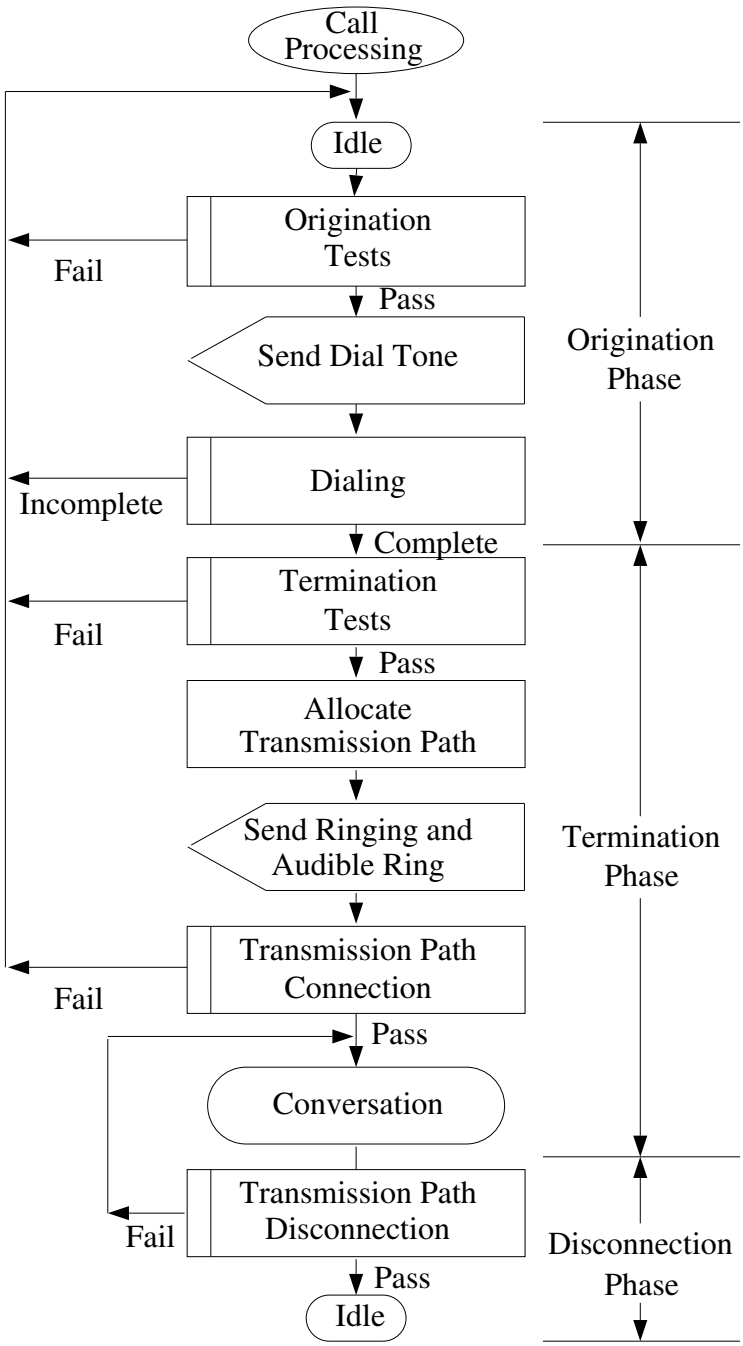


Figure 20. Call Processing Flowchart for Line Origination

3.2.6.2 Origination Phase The *Origination Phase* includes the processing of a call from the detection of a tone until the call is abandoned or terminates to the destination line[16]. The *Origination Phase* may also terminate on the detection of a *Busy* or an *Out Of Service* tone by the line initiating the call. If the *Origination Phase* is not completed successfully, no further phases will take place, see Figure 20.

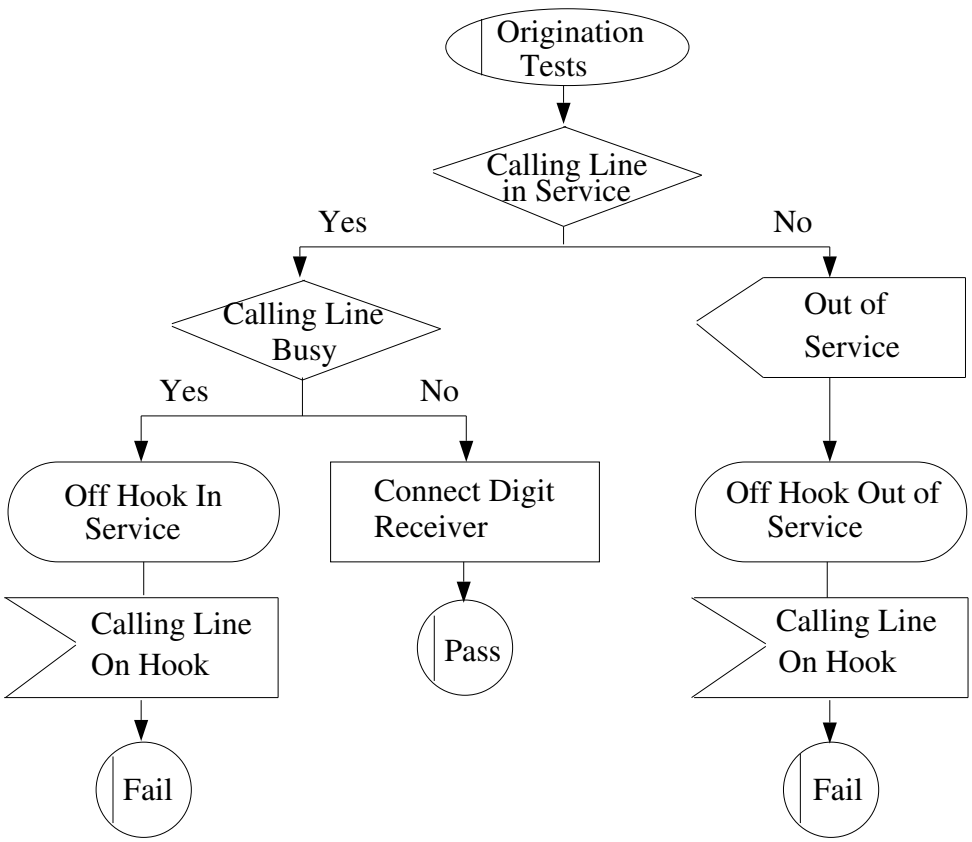


Figure 21. Flowchart for Origination Tests

The most important task in this phase are the *Origination Tests*, which consist in testing the status of the line being used to initiate a call. They are represented by the flowchart in Figure 21. In this process all status tests are first applied on

[16] A call terminates to the destination line if the *System Network* receives a connection request from the line initiating the call.

the line of the *Origination Side*. Once this task passes, the *Origination Side* may continue with the dialing of a telephone number.

When *Dialing* is successfully completed, the *Termination Phase* begins. However, if *Dialing* is not completed the call terminates and the *Connection* is released, see Figure 20.

3.2.6.3 Termination Phase The *Termination Phase* includes the processing of a call from the point at which the *System Network* determines the destination line until the communication path is established. The *Termination Phase* may terminate on the detection of a *Busy* or an *Out Of Service* tone by the line initiating a call.

It starts by applying some status tests on the *Destination Side*'s line, *Termination Tests*, Figure 22, and ends when either side decides to terminate the current call, see Figure 20.

Checking the status line of the *Destination Side* is explained by the flowchart shown in Figure 22. These tests result in different situations:

First, if the *Destination Side*'s line is not in service an appropriate tone is sent to the calling line that must go *OnHook*, otherwise, the *System Network* proceeds in checking the line for a *Busy* status.

Second, if the line is neither *Out Of Service* nor *Busy*, the *System Network* then verifies if incoming calls have been forwarded to another pre-selected line, a *Forward* feature is in effect. If not, the line is *Idle* and therefore the communication path may be established. If the last terminating line is one of the lines that have forwarded the current call, a Busy signal is returned to the calling line.

However, if the *Destination Side*'s line is *Busy* an appropriate tone is sent to the calling line that may go *OnHook* and terminates the call, or invoke a *Ring Again* feature and then *Hangs Up* its telephone. In the former case, the calling line goes *OnHook* to release the *Connection*, while in the latter case, the *System Network* keeps looping on the *Destination Side*'s line until it becomes free. Processing *Ring Again* feature is presented in Figure 23.

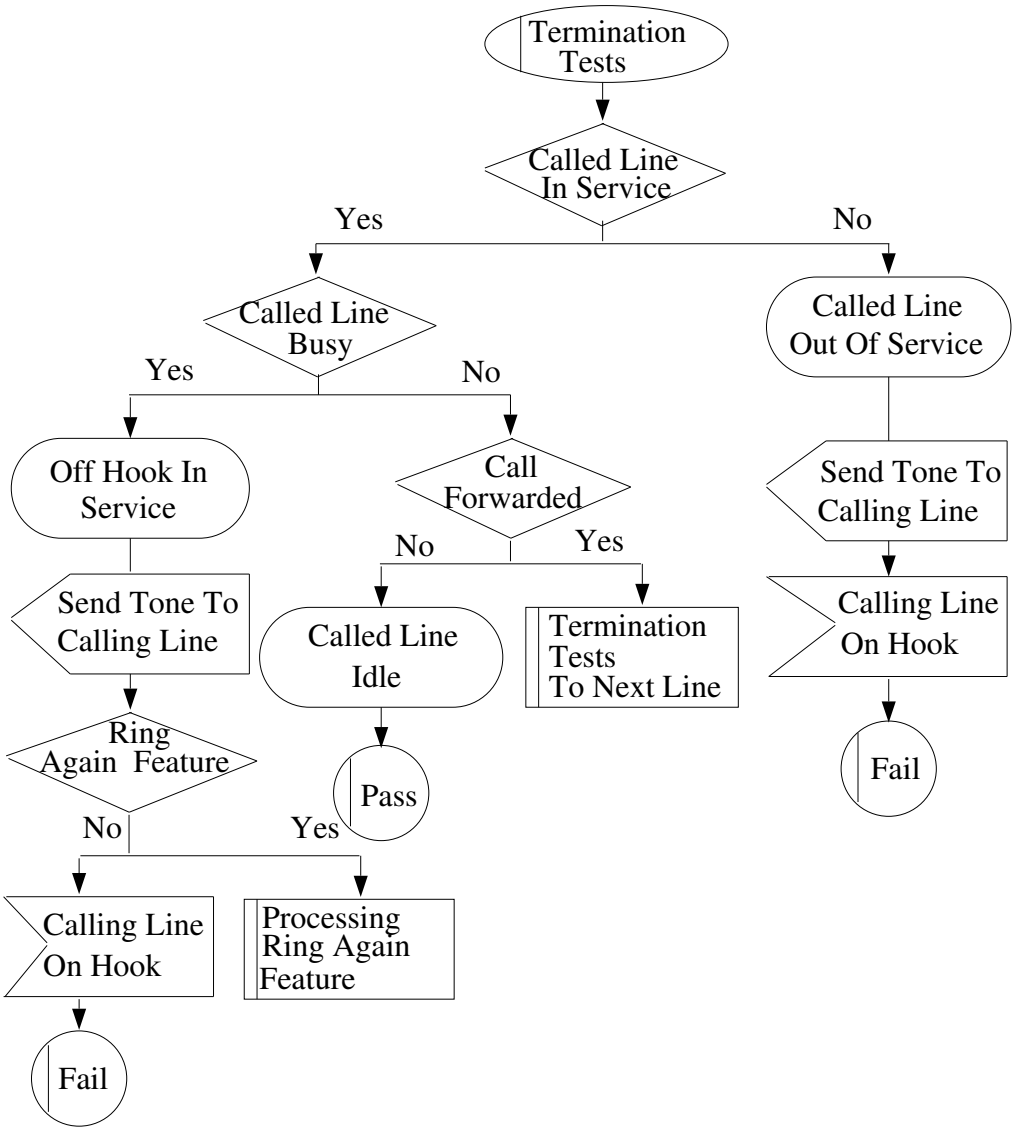


Figure 22. Flowchart for Termination Tests

Meanwhile, the *Ring Again* feature may be canceled and the *System Network* stops looping on the *Destination Side*'s line, and therefore the *Connection* is released.

While looping on the *Destination Side*'s line, as soon as the latter becomes free, the *System Network* informs the invoker of the feature by sending him a *Ring Again Notification* signal (*RANO*). The calling line may then go *OffHook* and wait for an answer from the *Destination Side* or cancel the *Ring Again* feature and release the current *Connection*.

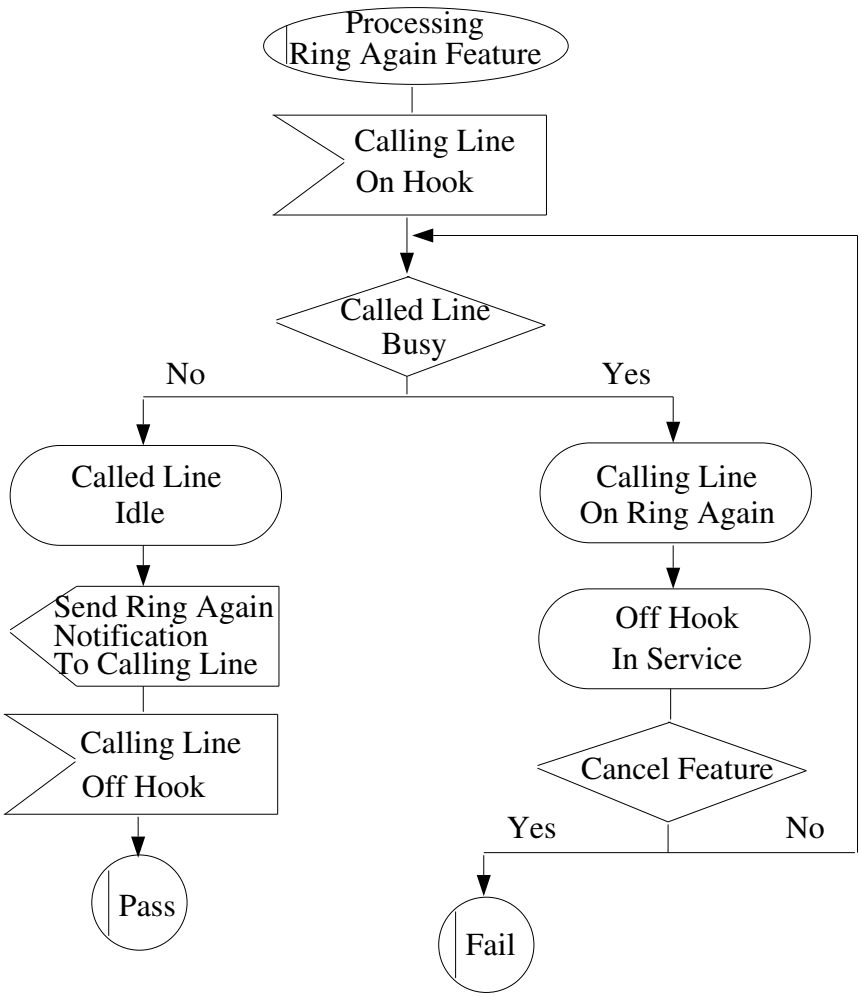


Figure 23. Flowchart for Ring Again Feature

3.2.6.4 Disconnection Phase The *Disconnection Phase* includes the processing of a call from the detection of a *Disconnect* signal (from either line) until the communication path is released and the facilities are idled. Since it is one part of a single *Connection*, this phase lasts until all participating telephones go *OnHook*. This is explained by the flowchart presented in Figure 24.

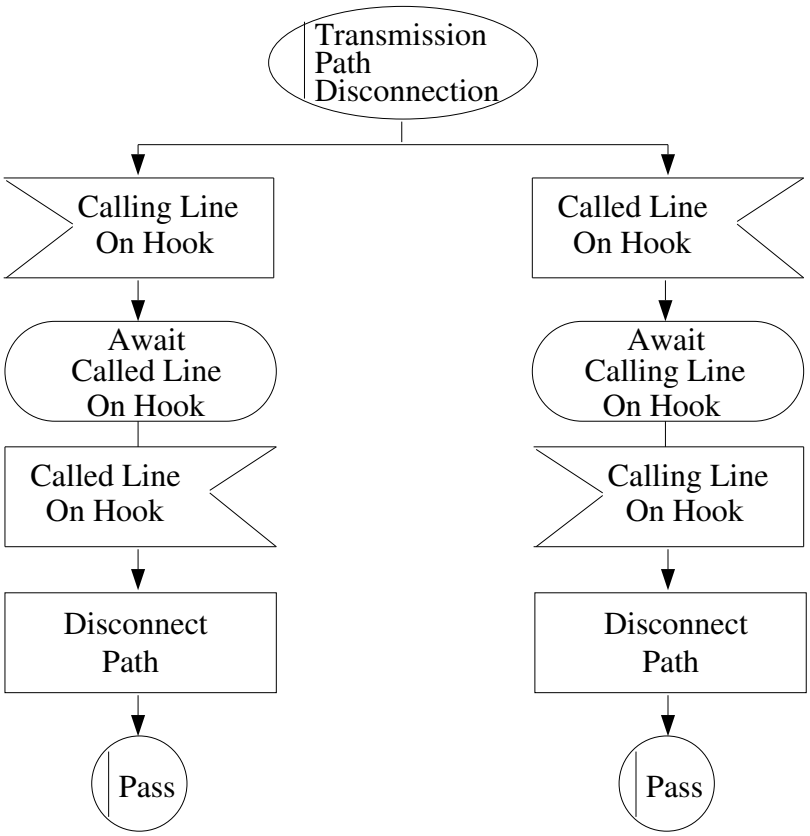


Figure 24. Flowchart for Transmission Path Disconnection

Note that the remaining features introduced in our *Sample Telephone System* may be invoked by any participating *Subscriber* in a *Connection* while it is in conversation. Each additional call from within a Connection is similar to a normal

call, except that it may use some other additional actions to inform the other sides and to return to the initial *Connection.*

3.2.7 Informal Description of the System Requirements

3.2.7.1 Introduction In this section we give an informal definition of a set of requirements that are taken from real telephone systems functionalities and used as the basis for the functioning of our *Sample Telephone System*. The type of requirements listed below are mainly related to the use of telephone lines and also the use of the features provided by the sample *Business Set* presented in Figure 10. Some general requirements on the overall system functioning are also presented in this section.

There may be other situations that are not considered in this work but the listed requirements will be reflected in the formal specification of the typical *Sample Telephone System* in LOTOS.

3.2.7.2 Definition of the Set of Requirements

1. The *Telephone System* must allow for an arbitrary number of *Connections* to be established simultaneously.
2. All lines that are in service are initially disconnected and the associated telephones are in an *OnHook* state.
3. A line that is not in service is *Out Of Service* (state).
4. Many telephones may use the same line alternately, so any telephone may have *Extensions*.
5. A telephone is *Busy* if the associated line is connected to some other connection through the same telephone or some other *Extension*.
6. A telephone is ready to initiate or receive a call if it is not *Out Of Service* and not *Busy*.
7. A line must be in service or *Out Of Service* but not both.
8. A line that is in service must be free (ready) or *Busy* but not both.
9. A telephone must be *OnHook* or *OffHook* but not both.
10. A telephone that is ringing is *Busy*.
11. While a telephone is *OnHook*, the *Subscriber* cannot:
    a. receive *Audio Services* (*Dial* tone, *Busy* tone, ...),

b. *Dial* a number,
c. invoke a feature among *Hold*, *Conference*, *Transfer* and *Ring Again* features using that telephone.

12. While *OffHook*, a telephone cannot:

    a. *Ring*,
    b. notify a *Ring Again*.

13. A *Subscriber* can request a *Forward* feature only if its telephone is *OnHook*.
14. A *Subscriber* can request a *Forward* feature only if its line is in service and not busy.
15. A *Subscriber* on any line can request a *Forward* feature only if that feature is not currently active on the same telephone.
16. At any time a *Subscriber* can request a *Forward* feature provided that requirements 13, 14 and 15 are fulfilled. Therefore for any incoming call to its telephone and thus to the telephone to whom these calls have been forwarded, its telephone remains busy until the communication path is established or the call is aborted.
17. A telephone can accept at most one *Audio Service* at a time.
18. A telephone cannot *Ring* unless someone dials it.
19. *Subscriber1* cannot dial *Subscriber2* unless it has *Dial* tone.
20. Once the connection is established a participating *Subscriber* can:

    a. talk,
    b. listen,
    c. go *OnHook*,
    d. put the current call on *Hold*,
    e. make a *Conference* call,
    f. *Transfer* the current call.

21. The features *Hold*, *Transfer/TWC* and *Conference* are not possible unless the connection has been established.
22. A *Subscriber* can invoke the *Ring Again* feature only if the *Destination Side*'s line is *Busy*.
23. While waiting for a *Ring Again Notification* or being put on *Hold*, a telephone is *Busy*.
24. A *Subscriber* that dials its own telephone number will get a *Busy* tone.

25. A *Subscriber* can get *Dial* tone only if its telephone is:
    a. *OffHook*,
    b. not *Busy*,
    c. in service.
26. A *Subscriber* that has invoked the *Ring Again* feature must *HangUp* his telephone.
27. A *Subscriber* that has invoked a feature can at any time cancel it.
28. With the *Transfer* feature a *Subscriber* may:
    a. consult privately with a third party,
    b. make a *Three-Way Calling* conversation,
    c. *Transfer* the call to a third party and then goes *OffHook*.
29. The *Conference* feature allows a *Subscriber* to set up a *Connection* between himself and N other *Subscribers* , for $N > 1$.
30. During a *Conference* conversation, any participating *Subscriber* may:
    a. consult privately with a new *Subscriber*,
    b. allow an *Additional Subscriber* to join them in the *Conference* conversation,
    c. goes *OffHook*.

Section 3.3 Conclusions

The structure of the *Sample Telephone System* described in Section 3.2 corresponds to the structure presented in Section 3.1.4 where we discussed the design of a telephone system in general. However, the functional structure of this *Sample Telephone System* is described in such a way to reflect the requirements defined in [Bel88].

The first aim in the design of this sample model is to allow the users to process multiple connections in parallel. Another advantage of using this system is that it provides its users with a number of telephone features.

The functioning of the *Sample Telephone System* is the subject of a formal definition using the LOTOS language in the next Chapter.

Chapter 4 Formal Description in LOTOS of the Sample Telephone System

Section 4.1 Introduction

A specification is the abstract definition of a system. It is written in a specification language that can be either formal (e.g. LOTOS) or informal (e.g. English). Informal specifications are easier to read and write than formal ones, but a substantial advantage of formal specifications is that they have a precise meaning. In formulating the system's requirements in an informal language, there is often ambiguity. One of the reasons for which the language LOTOS has been chosen to be the specification language of our *Sample Telephone System*, is that it can resolve such ambiguities in a precise way.

As mentioned in Chapter 3, to formally specify a system one needs a precise definition of each of its parts and the way these parts are interconnected to each other and to the system's environment. In the following sections, the system described in Section 3.2 of Chapter 3 is the subject of a formal definition using LOTOS. The complete LOTOS specification of the *Sample Telephone System* is presented in Appendix B.

First, we introduce some design methodologies used to specify the sample telephone system in LOTOS, then we present the general structure of the *Sample Telephone Specification* along with an informal description of its functioning and of the abstract data types as well. Next is the specification of the different system's components along with the features provided by this system. Finally, we discuss the debugging of our specification using an interpreter, including a presentation of a method to test a LOTOS specification.

Section 4.2 Architecture of the Sample Telephone Specification

4.2.1 Introduction

An architecture is a general structure for the representation of a system corresponding to the user requirements. When a satisfactory architecture is

achieved, one may derive from it more concrete definitions of the system. A good architecture needs to be well structured for human comprehension and to facilitate the complete expression of the functionality of a system.

In the next sections we present some design principles to be respected by a designer in order to get a better representation of the description of a system. This will be followed by a presentation of some well-known specification styles that allow the designer to structure formal specifications and that can be used as a basis in a satisfactory design. Finally, we show how these styles were used in the formal description of the *Sample Telephone System*

4.2.2 Design Principles

The first aim of the system design process is to derive an implementation – a concrete instance of a system realization – that fulfils the user requirements. But due to the complexity of some distributed systems, the design process may become a more complicated task. In order to achieve a better – clear and readable – representation of a system, the design process may be carried out in steps, where each step represents a different level of abstractness of the system. In LOTOS, the complete specification of the behaviour of a (complex) system may be achieved in a stepwise fashion. First, the system is described as a set of processes that represent distinct objects within the system, then each resulting process is decomposed into sub-processes. The process of refinement of the system is repeated until we end up with simpler single entities where no further decomposition is possible. Each stage of the refinement process represents a level of abstraction of the formal description of the system.

This method, the stepwise refinement, helps the designer to achieve a clearer description of the behaviour of each separate component, and yields a well structured and readable behaviour of the overall system.

In the design of our *Sample Telephone Specification* we used a mixture of the different specification styles discussed in Section 2.5. The use of a specification style in the design of a system must respect some general principles of the design. These design principles are mainly Orthogonality, Generality, and Open-endedness [VSvS88]. Orthogonality preserves locality aspects of (sub)systems, Generality suggests the use of general-purpose parameterized definitions, and

Open-endedness supports flexibility of a design to ease the modification of the system functionality.

4.2.3 Use of Specification Styles in the Sample Telephone Specification

In our design of the *Sample Telephone Specification* in LOTOS different specification styles have been used. Each of these styles has its own role to play in the formal description of the behaviour of a specific (sub)system depending on its functionality and on the level of abstractness of the overall system. In this section we give a brief discussion on these styles and show their relation to the design of the *Sample Telephone Specification*.

In order to achieve a satisfactory design, the use of specification styles in the design stage of a system must fulfill the design principles discussed above. The (formal) description of a system may have as its prime concern the description of observable behaviour, like a black box, or it may have as a prime concern the description of the internal behaviour of the system, like a white box. The former description is called "*extensional*" while the latter is called "*intensional*" [VSvS88]. For each type of description some specification styles have been defined to guide the design process.

4.2.3.1 The Extensional Description of a System In this type of description only observable interactions are shown and their temporal ordering relationship is defined in terms of alternatives of sequences, or parallel composition of constraints.

For extensional descriptions two specification styles have been defined [VSvS88]; these are the *monolithic style* and the *constraint-oriented style*.

4.2.3.1.1 The Monolithic Style In the monolithic style the temporal ordering relationship of interactions is defined in terms of alternatives of sequences. This style is more suitable for a small system where only observable behaviour is of concern, like a *black box* description. For complex (distributed) systems, however, this type of style is impractical (to some extent), and if the description of such a system is achieved all design principles discussed above are violated and the description is more or less unreadable, because of lack of structuring.

The monolithic style prohibits the use of parallel composition operators. Therefore, the monolithic style cannot be used for the top levels of our specification but it is still useful in the local description of a single component of the system, e.g. the local description of the behaviour of the *System_Network*, presented in Section 4.4.6.

The *Sample Telephone System* described in Section 3.2 states that an arbitrary number of connections may be handled simultaneously. In addition, each connection is independent from the others in a sense that its subscribers use different telephones. This is the idea of separation of concerns in the design strategy of our *Sample Telephone Specification*. An arbitrary number of connections may be handled separately and concurrently. In LOTOS, this type of representation may be achieved by using the interleave operator "|||".

4.2.3.1.2 The Constraint-Oriented Style In the constraint-oriented style the temporal ordering relationship of interactions is defined in terms of separate constraints composed in parallel. Each constraint is defined on a subset of interactions within a separate entity. If the subsets of interactions that are relevant to some entities are disjoint, i.e. the entities do not synchronize on any action, we use the "interleave" operator in LOTOS. If, however, there exist some non-disjoint interactions in the subsets, the parallel composition is synchronized and we use the "generalized" parallel operator in LOTOS.

The constraint oriented style supports parallel structures and thus it provides designers with tools for modularity of description by separation of concerns. The structuring of the specification may also help a designer in the description of the internal behaviour of a system. In this style, all design principles discussed above are fully preserved.

The use of such a style in the specification of a distributed system helps the user in distinguishing between different types of constraints that may be applied on the interactions within a system. A constraint may be applied on the overall (distributed) system and in this case we say it is a *remote* constraint, it may be applied between interactions of different (sub) systems within a system and in this case is known as *end-to-end* constraint, or it is applied on the subset of interactions within the local behaviour of an entity in the system and in this case we say that

it is a *local* constraint. The use of different constraints in our *Sample Telephone Specification* is discussed later in Section 4.2.4.

4.2.3.2 The Intensional Description of a System With the intensional description, both observable and internal interactions of a system are presented. Two main specification styles have been defined for such a description, the *state-oriented* and the *resource-oriented* styles.

4.2.3.2.1 The State-Oriented Style In this style the system is regarded as a single entity whose internal behaviour is explicitly defined. Only observable interactions are represented. Like in the *monolithic* style, a description in the *state-oriented* style is represented in terms of a collection of alternative sequences on interactions in a single process definition. State variables are present, and the alternatives are guarded by tests on such variables.

Since this style gives the explicit definition of a system, it can be used to transform the formal specification into the final implementation of the system. The *state-oriented* style is therefore particularly useful in the end-phase of the system design trajectory when the resource can be defined and understood as an object that can be implemented by a single implementation authority [VSvS88].

In our *Sample Telephone Specification*, this style has been avoided in the first stages of the design methodology because of its lack of structuring. The *Sample Telephone System* considered in this work supports distribution of tasks and therefore its design needs to be well structured for surveyability and human comprehension. The description of the process *System_Network* in the *Sample Telephone Specification* uses the *state-oriented* style. As a single entity, it is represented by a set of alternative sequences of interactions. With each alternative, a predicate is associated and the alternative is chosen only if the predicate is evaluated to true.

4.2.3.2.2 The Resource-Oriented Style This style makes extensive usage of parallel composition operators, but, unlike the *constraint-oriented* style, it allows the presentation of both observable and internal interactions of a system. A substantial advantage of using this style in the description of (particularly complex) distributed systems is that it permits the usage of the *hiding* construct, which

is used to hide internal interactions, by making them invisible in the external behaviour of a system.

Because of the complexity of telephone systems, their formal description requires a style that allows modularity of the design. The functioning of telephone systems is dominated by the use of signals. In the real world, these signals are not visible to the environment, and therefore a means for hiding signal exchanges between different entities is needed. We need also a specification style that allows parallelism and synchronization to be able to compose in parallel different connections and also to allow the *System Network* and the *Connection Handler* to synchronize with the telephones. All these concepts are, however, provided by the *resource-oriented* style, except that we introduced the idea of constraints everywhere in the *Sample Telephone Specification* as it will be discussed later in Section 4.2.4.

Note that in the *resource-oriented* style, the structuring of the formal description of a system is based on distinguishing between separate embedded resources by separating "intensional concerns" of the overall behaviour. The structure obtained is similar to the one described when we discussed the *constraint-oriented* style, however the description here is extensional. Note that the formal description of a single resource by itself may use any specification style among the *monolithic*, the *constraint-oriented*, and the *state-oriented*.

4.2.4 Use of Constraints in the Sample Telephone Specification

In the design of the *Sample Telephone Specification* three types of constraints have been introduced [FLS91]. These constraints are applied on the temporal ordering of events on the gates, which represent lines. A constraint may be applied within the behaviour of a process (*Local*), between two processes (*End-to-End*), or on the overall system behaviour (*Global*).

4.2.4.1 Local Constraints The *Local* constraints control the temporal ordering of events at each telephone. They are represented by using both parallel or sequential compositions within the process that describes the behaviour of a single entity of the system. As an example, consider the events involved in a normal call[17]:

[17] In this example, we assume that the telephone of the caller is free and the call is not aborted.

•the constraints on initiating a call are:

- *OffHook* is followed by a *Dial Tone*,
- *Dial Tone* is followed by *Dialing*,
- *Dialing* is followed by a *Connection Request* . . .

Therefore, *Local* constraints are forced by the local definition of a single entity of the system.

4.2.4.2 End-to-End Constraints The *End-to-End* constraints control the interaction between communicating entities within the same connection. For instance, within a single connection different subscribers may communicate with each other. To control the sequence of events performed by a subscriber with respect to the others, we introduced a special process, *Connection Handler*, which controls the sequencing of signals between different telephones participating in a single connection.

As an example, consider the internal events involved in a normal call:[18]

•the constraints on initiating a call in relation to being answered are:

- *Dial Tone* is the first signal in a connection,
- *Dial Tone* (caller side) is followed by a *Connection Request* (caller side),
- *Connection Request* is followed by a *Connection Attempt* (receiver side),
- *Connection Attempt* is followed by *Ringing* (callee side),
- *Ringing* is followed by a *Connection Establishment* (both sides) . . .

Therefore, *End-to-End* constraints are enforced by the process *Connection_Handler*. This process is composed in parallel with process *Subscribers*. The two processes synchronize with each other on gate *line* for internal signal exchanges only.

4.2.4.3 Global Constraints The *Global* constraints are system-wide constraints. In the *Sample Telephone Specification* these constraints control the use of lines by telephones within different connections. For instance, a line can be in use by at most one telephone at any given time. Also, ringing cannot be applied on a telephone that is currently being used within another connection. These constraints

[18] In this example, we assume that both telephones (the caller and the callee) are free, and the callee must answer the call.

are usually achieved by the use of *Abstract Data Types* (ADTs), namely sets of telephone numbers in the case of our specification. Process *System_Network*, that has access to all the lines and is capable of appropriately responding to a query from any subscriber, is responsible for manipulating such ADTs.

Processes *Multi_Connections* and *System_Network* synchronize with each other on gate *line*, see Specification 1. In this case, the *System Network* is able to control all the requested lines, and provides their senders with appropriate signals.

```
behaviour
hide line in
 Multi_Connections [user,line]
 |[line]|
 System_Network[user,update,line](Empty,Empty,Empty,
        NoPair)
```

Specification 1. The Topmost Level of the Specification

Section 4.3 General Structure of the Sample Telephone Specification

4.3.1 Introduction

As already mentioned, the specification of the *Sample Telephone System* uses a mixture of different styles. Because it describes the system at a very abstract level and therefore it is implementation independent, our formal description (in LOTOS) uses mostly the constraint-oriented style, Section 4.2.3.1.2. Three different types of constraints have been specified, see Section 4.2.4.

The *Sample Telephone Specification* is structured as follows:

specification *Sample_Telephone [user,update]* :**noexit**
library
NaturalNumber, Boolean
endlib
< Abstract Data Type Definitions >
behaviour

< Processes Definitions >

endspec

Specification 2. General Structure

It can be accessed through gate '*user*'. For example:

user ?*N:Phone* !*OffHook* ;

represents the only possible action that allows a subscriber to initiate a call. A subscriber is identified by the first data item (number *N*)[19] and the associated action is given by the second value (*OffHook*) on the same gate *user*. The gate *line* is used for the transmission and reception of signals between the telephones and the *System Network*. The line being used is given by the associated number on the same gate *line*. Visible signals to the environment are transmitted on gate *line* then sent to the environment on gate *user*. Signals that are internal to the *Telephone System* are, however, handled only on gate *line*. In LOTOS, this type of messages can be specified by hiding the associated gate (see Section 2.2.11).

In the following sections we give an informal description of our specification, including its abstract data types part, and we explain how the functionality of the *Sample Telephone System* can be formally specified using LOTOS. Then we present the design of the whole specification in LOTOS.

4.3.2 Top Levels of the Specification

The top levels of the *control part* of the *Sample Telephone Specification* consist of two main processes, *Multi_Connections* and *System_Network*, composed in parallel. The process *Multi_Connections* creates a single connection, process *One_Connection*, and then gives the environment the possibility of invoking new connections. An arbitrary number of connections may be processed simultaneously and without synchronization, see Section 4.4.3.

All the connections must synchronize with process *System_Network* on internal signal exchanges. The gate *line* is used in the specification for handling all such events. Therefore, processes *Multi_Connections* and *System_Network* must synchronize on gate *line*. We explicitly hide the gate *line* because in the real world the first stage of processing a signal exchange is an internal event to the

[19] The number *N* is offered by the environment. It represents the telephone number of the *Subscriber* going *OffHook*.

telephone system. Once a signal is sent to its destination, if it is visible to the environment, the same primitive will be sent again on gate *user* to make it visible to the environment.

In order to be able to control the lines and provide telephones with appropriate signals, the process *System_Network* uses four sets of numbers, *BusySet*, *ServiceSet*, *ForwardSet*, and *ForwardPairSet*, see Section 4.3.3. Initially, all the sets are empty. To update the set *ServiceSet* (add a new line or remove an old one), the environment may at any time provide the appropriate line number using a special gate "*update*", see Section 4.4.6. Therefore, the gate *update* is used for the interactions of the *System Network* with the environment to update the sets of line numbers.

The behaviour of a single connection is described within process *One_Connection*. Any subscriber may participate in the same connection and with each connection is associated a local controller whose behaviour is described by the process *Connection_Handler*. This process synchronizes on gate *line* with the processes describing the behaviour of subscribers participating in the same connection. This is to be able to control the temporal ordering of events (signal exchanges) performed within the same connection.

The top level of process *Subscribers* consists of two processes composed in parallel. These processes do not synchronize with each other, because each process describes the behaviour of a subscriber in an independent way from the other participating subscribers. So, each *subscriber* may start a new connection by behaving as an *Origination Side*. Other subscribers that may be involved in the same connection are invoked from within this connection by any participating subscriber. The control of protocols between subscribers is handled by process *Connection_Handler*.

```
specification Sample_Telephone [user,update] :noexit
  (* ... Abstract Data Types Definitions ... *)
  behaviour
  hide line in
    Multi_Connections [user,line]
    |[line]|
    System_Network[user,update,line](Empty,Empty,Empty,
           NoPair)
```

```
    where
      process Multi_Connections [user,line] :noexit:=
        One_Connection [user,line]
        |||
        i ;
        Multi_Connections [user,line]
        where
          process One_Connection [user,line] :noexit:=
            Subscribers [user,line]
            |[line]|
            Connection_Handler [line]
            where
              process Subscribers [user,line]
                      :noexit:=
                Origination_Side [user,line]
                |||
                Destination_Side [user,line]
                where
                 (*... Other Processes
                   Definitions ...*)
              endproc (* Subscribers *)
               (*... Process Connection_Handler
                 Definition ...*)
          endproc (* One_Connection *)
      endproc (* Multi_Connections *)
       (*... Process System_Network
         Definition ...*)
endspec (* Sample_Telephone *)
```

Specification 3. Top Levels of the Specification

The top levels of the specification are shown in Specification 3, and the corresponding graphical representations are given by Figures 25 and 26. In our graphical representation, parallel processes are represented next to each other while interleaved ones are drawn on top of each other. Processes defined within another process definition are represented inside the box provided for that process. The gate over which processes synchronize is shown in the upper left corner of

the box that includes those processes. Lines associated with small circles show the access of a process to a specific gate.

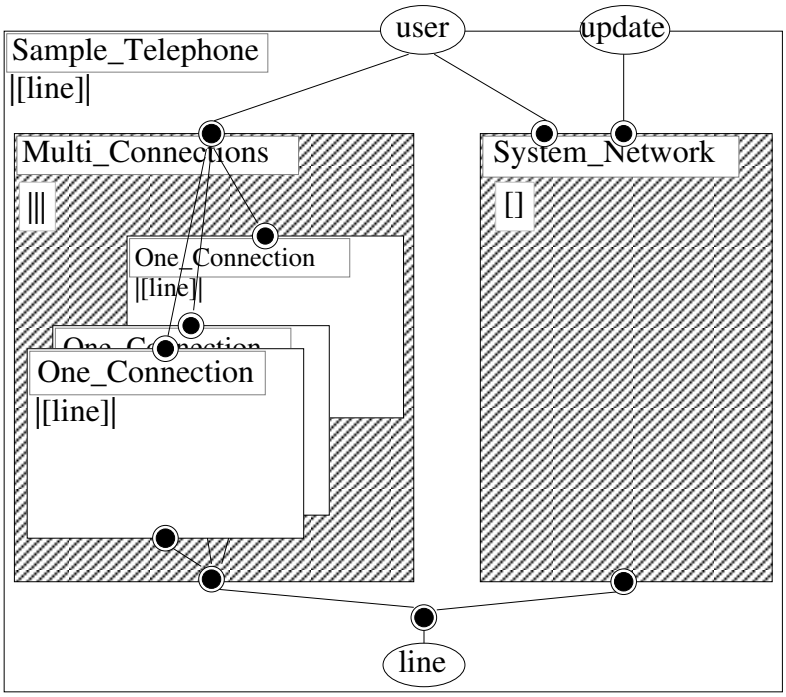


Figure 25. Top Level's Graphical Representation of the Specification

In Figure 25, gates *user* and *update* are drawn on the border line of the box provided for the *Telephone System*, to show that they are the access point for the environment. Gate *line*, however, is drawn inside that box to show that it is hidden from the environment.

Further refinement of the *Telephone System* gives the structure presented in Figure 26. The process *One_Connection* consists of process *Subscribers* that is composed in parallel with process *Connection_Handler*.

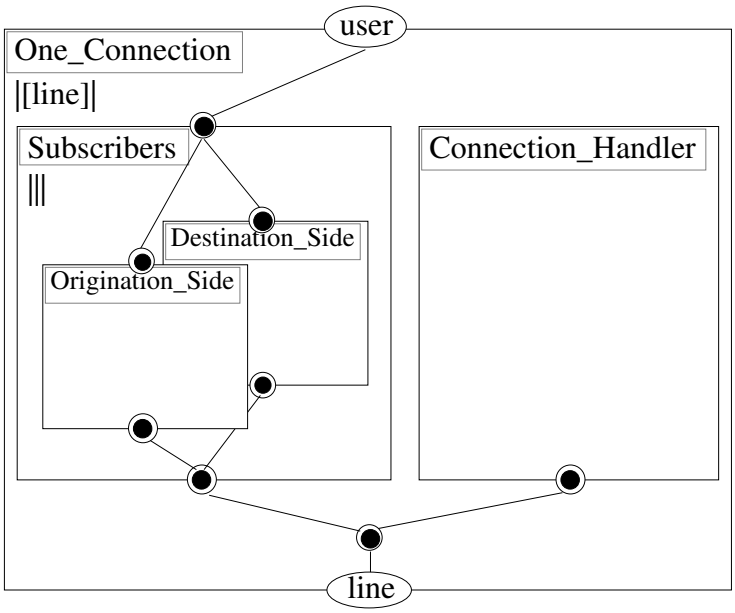


Figure 26. Refinement of a One_Connection

4.3.3 Description of the Abstract Data Types

In LOTOS a standard library of data types is defined, but these are types of general use. In order to be able to use other types than the existing ones, LOTOS allows users to define their own data structures and value expressions in the *data part* of a specification.

In our *Sample Telephone Specification* and in addition to the *NaturalNumber* and *Boolean* types defined in LOTOS library, we defined a set of types that are required by the control part. The numbers between brackets refer to the line numbers where that type is defined within the *Sample Telephone Specification* presented in Appendix B.

First, since we are dealing with telephone systems, we need to define *telephone numbers* (lines 5–77) and some operations on them. The operations == and <> are used to compare telephone numbers[20]. Operation *To_Nat* is used to

[20] The comparison is made within a predicate.

map real telephone numbers to *Natural Numbers*. In type *Phone_Num* definition, operation *Num* is defined to allow the environment entering real telephone numbers, provided it follows the format:

Num(d1, d2, . . . , d7) , where ***di*** are digits, for $1 \leq i \leq 7$.

A telephone number, in our specification, may consist of up to 7 digits, however it's also possible to use numbers with larger numbers of digits. This requires some changes in the definition of *Phone_Num*.

We need also to define a generic *set* type (*PhoneSet*, lines 79–110) to be actualized to define the sets *BusySet*, *ServiceSet*, and *ForwardSet*. The set *BusySet* contains the identities of all the lines currently being used, while the set *ServiceSet* contains the identities of all the currently existing lines in the telephone system[21]. The remaining sets of numbers are used by the process *System_Network* to manage the forwarding of calls. *ForwardSet* contains the numbers of the lines where a *Forward* feature is active. In order to be able to manage these sets, many useful operations have been defined within the *PhoneSet* type. These are: operation *Empty* that denotes an empty set, operation *Insert* to add a telephone number to a set, operation *Remove* to remove a telephone number from a set, operation *IsIn* to check the existence of a telephone number in a set, operation *NotIn* to check the non existence of a telephone number in a set, operation *Is_Empty* to check if a set is empty, operation *Head* to get the first element of a set, and operation *Tail* to get the subset of the remaining numbers of a set after removing its first element. A second type of sets is the *ForwardPairSet* (lines 124–149) that is needed for the *Forward* feature. This set contains pairs of telephone numbers of type *PhonePair* (lines 112–122). Such a pair consists of the telephone number that has forwarded a call and the telephone number to whom a call is forwarded by the telephone given by the first number, e.g. ***ForPair (N, FN)*** means that ***N*** has forwarded a call to ***FN***.

Another important data item needed by the *control part* of the specification are *keyboard keys* and *signals*. The *Key* type is defined in lines 151–169 and uses two operations == and <> to check for a specific key being used by a subscriber. Only three of these keys (*HoldK*, *TransK*, and *ConfK* keys) may be involved in the comparison process. *Signal* types are defined in lines 178–204 using also two

[21] The identity of a line is the number of the telephone associated with that line.

operations == and <> to check for a specific signal being sent by the *System Network*. Among these signals some are of special use. These are *RING, BUSY, DITO, DISC*, *OOFS*, *FORW*, and *RANO*. These signals differ from the others on two accounts. First, an event with such a signal occurs only if specific conditions are satisfied. Second, these signals may affect the set *BusySet*.[22] These are defined of type *Control_Signal*. The remaining signals are not constrained to any condition and have no effect on any set of telephone numbers. Also the different tones that a subscriber may get on its telephone are defined of type *Tone*, lines 171–176.

4.3.4 Informal Description of the Sample Telephone Specification

The LOTOS specification of the *Sample Telephone System* consists of two main parts: *data part* and *control part*.

In the *data part*, we define all the data used by the different processes of the specification, Section 4.3.3, while in the *control part* we describe the behaviour of the components of the system in terms of processes and/or behaviour expressions, Section 4.4.

The specification must be written in such a way as to allow for an arbitrary number of *Subscribers* to use the *Telephone System* simultaneously, see Section 3.2.7.2. It must also describe both the interactions of the *Telephone System* with its *Environment* and signal exchanges between different components of the *Telephone System*. To do so, three different gates are used in our LOTOS specification; gates *user* and *update* to handle the interactions between the *Telephone System* and its *Environment*, and gate *line*, used as a means to exchange signals between the telephones and the *System Network*. Within an event, different types of data may be associated with the same gate. The first data item is the identity of the line being used while the second one defines the action being handled on that line (a primitive) and possibly an additional data item to determine the destination line of the current action.

The *System Network* represents the manager of all the lines used by the *Telephone System*. It synchronizes with telephones on any internal action where the associated predicates are satisfied. On gate *user*, it synchronizes with the environment for *Forward* feature purposes. On gate *update*, however, it synchronizes with the environment when for updating the sets of line numbers.

[22] This is the case when a free line becomes busy or a busy line gets released.

Within a single connection, *One Connection*, the *Telephone System* works in the following manner:

The subscriber initiating a call, *Origination Side*, picks up the telephone handset (*OffHook*). It then gets: (1) an out of service signal (*OOFS*) if its telephone is not connected to the *Telephone System*[23], (2) a busy signal (*BUSY*) if an extension of its telephone is being used, or (3) a dial tone signal (*DITO*) if its line is free. For more details on status line see Section 3.2.3.3.1.

In the first two cases, the *Origination Side* is forced to hang up (*HangUp*) the telephone. In the third one, the *Origination Side* may start dialing the callee's telephone number, *Destination Side*. When it finishes dialing, the control is passed to the other side.

Once the callee's number is dialed, the *Origination Side* gets one of the following three signals: (1) an out of service signal (*OOFS*) if the line specified by the dialed number is not connected to the *Telephone System*, (2) a busy signal (*BUSY*) if the callee's line is being used, or (3) an audible ring if the callee's line is currently free. In (1), the *Origination Side* must hang up the telephone (*HangUp*). In (2), the *Origination Side* may hang up the telephone and terminate the connection or it may invoke a *Ring Again* feature and then hangs up the telephone. If the *Ring Again* feature is invoked, the *System Network* keeps looping on the *Destination Side*'s line until it becomes free, then it notifies the *Origination Side*, which is waiting for a ring again notification *(RANO)*. As soon as the *Origination Side* gets a *RANO* signal, it may pick up the telephone handset and the communication path is established. Note that while waiting for a *RANO* signal, the *Origination Side* may cancel the *Ring Again* feature.

However in situation (3), the *Origination Side* may wait for an answer from the *Destination Side*, or hang up the telephone and release the connection. In the former case, the callee's telephone may start ringing or the *Destination Side* may have forwarded incoming calls to another telephone, and therefore *Destination Side* now denotes the party to whom these calls were forwarded.

If the *Destination Side* answers the call (*OffHook*) before an *OnHook* from the *Origination Side* is detected, then its telephone stops ringing and the communi-

[23] In real world this is not a signal, however the subscriber can recognize it. So, in our specification it's taken as a signal received by a subscriber.

cation path between both subscribers is established. Thereafter, the participating subscribers may enter in a conversation. Meanwhile, either subscriber may invoke any feature among *Hold*, *TWC*, *Transfer* and *Conference* features.

Invoking *Hold* Feature

If a subscriber decides to invoke a *Hold* feature, it must first inform the callee and then press the appropriate key. The callee is now on hold. While on hold, the callee may go *OnHook* at any time.

The process of calling a new party within a connection is similar to the one found in a simple connection except that the *Origination Side*'s telephone is already *OffHook* and to terminate or abort a call, it simply presses the *Hold* feature key again and will be re-connected to the first party. However if the first party has left the connection, the *Origination Side* must hang up the telephone after it returns to the first call and the connection is released.

Invoking *Transfer* Feature

When a call comes in to its telephone, the *Destination Side* presses the *Transfer* key then dials the telephone number of the party to whom this call is to be transferred. When the new party answers the call, the *Destination Side* leaves for good the conversation and this new party becomes the new *Destination Side* that will be connected to the *Origination Side* in a normal call.

Invoking *TWC* Feature

When a call comes in to its telephone, the *Destination Side* answers it then presses the *Transfer* key to (privately) consult with a third party. The *Origination Side* is automatically put on hold. After the new party answers the call, the *Destination Side* then presses again the *Transfer* key and all three parties are connected together in a conference like conversation. The feature is then called a *TWC* feature.

Invoking *Conference* Feature

While in conversation, a participating subscriber presses the *Conference* feature key, consults (privately) with a third party then presses the *Conference* key a second time to bring this new party to the conference. Note that an arbitrary number of subscribers may be involved in a single connection and any one may leave (for good) the conversation at any time.

Section 4.4 Design Steps of the Sample Telephone Specification

4.4.1 Introduction

In this section we show how the components of the *Sample Telephone System* presented in Chapter 3 can be formally described using LOTOS.

First, we discuss some particularities of *Telephone Systems* and how they can be described in LOTOS, then we present the structure of some important processes that can be a basis for a formal description in LOTOS.

4.4.2 Processing Disconnection of Telephones

The process *Connection_Handler* plays the role of a controller within a single connection. It is responsible for handling the connection of the lines to establish calls, and also their disconnection to terminate calls. There are several ways in which a call may be terminated.

– A call may be aborted by the *Origination Side* before its termination to the end destination.

– A call may not terminate successfully to the end destination.

– A call terminates successfully to the end destination and any participating subscriber may terminate it in a normal way.

When the *System Network* receives a disconnection request signal (*DISC*), it may send a disconnection indication signal (*DSIN*) to the other side or no signal is exchanged. This depends on the stage of the current call and the status of the lines being used. These are summarized by Figure 27.

To abort or terminate a call we use the LOTOS disable operator "**[>**". This operator transfers control from the current behaviour to the behaviour following the operator.

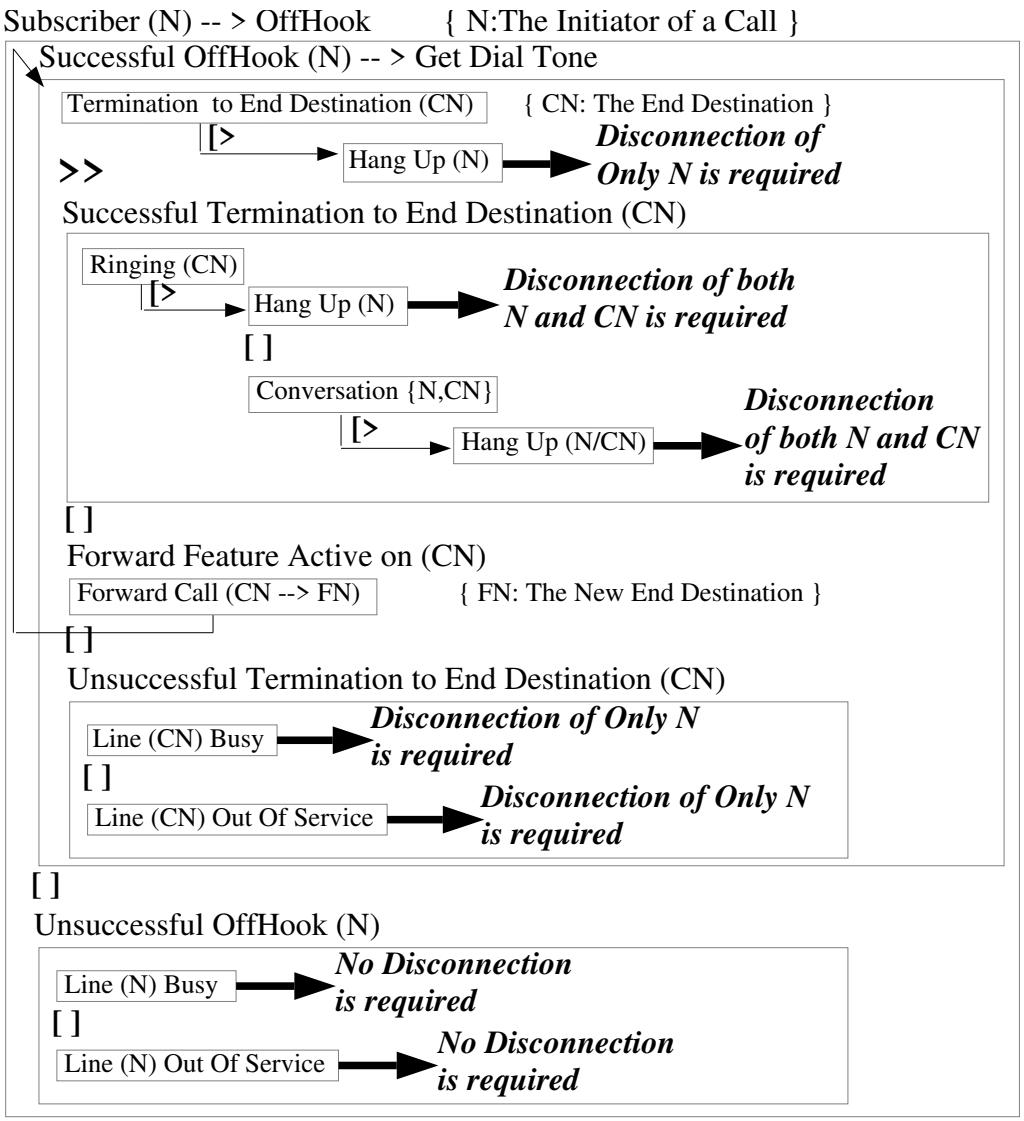


Figure 27. Different Types of a Disconnection

A *Subscriber* as the *Origination Side* goes *OnHook* after an unsuccessful *OffHook*[24], and therefore no disconnection is required.

[24] Its line is busy or out of service.

A *Subscriber* as the *Origination Side* goes *OnHook* after a successful *OffHook* but before the *Destination Side*'s telephone starts ringing[25]. Therefore, disconnection of only the *Origination Side*'s line is required. Note here that because the lines that have forwarded a call are kept busy until establishment of the communication path, they will be released upon disconnection of the *Origination Side*'s line[26].

If following a successful *OffHook* from the *Origination Side* and an unsuccessful termination of its call to the *Destination Side*'s line, the *Origination Side* must go *OnHook* and disconnection of only its line is required.

If a call terminates to a busy line and the *Origination Side* invokes a *Ring Again* feature, then: (1) if the *Origination Side* cancels the feature while its telephone is *OnHook*, the disconnection of only its line is required, (2) if after it gets a *Ring Again Notification* signal (*RANO*), the *Origination Side* goes *OffHook*, this automatically makes the *Destination Side*'s telephone ringing, and therefore, if afterward the *Origination Side* goes *OnHook*, the disconnection of both side's lines is required, (3) if the *Destination Side* answers the call, this situation will be similar to the one discussed below.

If following a successful termination of a call to the *Destination Side* whose telephone starts ringing, the *Origination Side* goes *OnHook*, then disconnection of both side's lines is required and the *Destination Side*'s telephone stops ringing.

If following a successful termination of a call to the *Destination Side* who has answered it, either side goes *OnHook*, then disconnection of its line is required. Once this line is disconnected, a disconnection indication signal (*DSIN*) is sent to the other side. The latter will be disconnected as soon as it goes *OnHook*.

Within a *Connection*, if any participating *Subscriber* invokes a feature to call any other subscriber, disconnection of the latter's line is required only after its telephone starts ringing. Note that in a *Conference* conversation, whenever a participating subscriber leaves the call its line is disconnected while the other subscribers remain connected to the same call.

4.4.3 Creating Concurrent Connections

In the real world, an arbitrary number of connections may be handled simul-

[25] The *Destination Side* here refers to the end destination if the current call was forwarded.
[26] The current call is aborted.

taneously, where each connection is processed separately. In LOTOS, this can be modelled by using the *Interleaving* operator "|||", see Specification 4 below. For more details on this operator refer to Section 2.2.10.1.

```
process Multi_Connections [user,line] :noexit :=
  One_Connection [user,line]
  |||
  i ;
  Multi_Connections [user,line]
endproc
```

Specification 4. Process Multi_Connections Definition

The process *Multi_Connections* may generate an arbitrary number of connections by recursive instantiations of the process *One_Connection*[27].

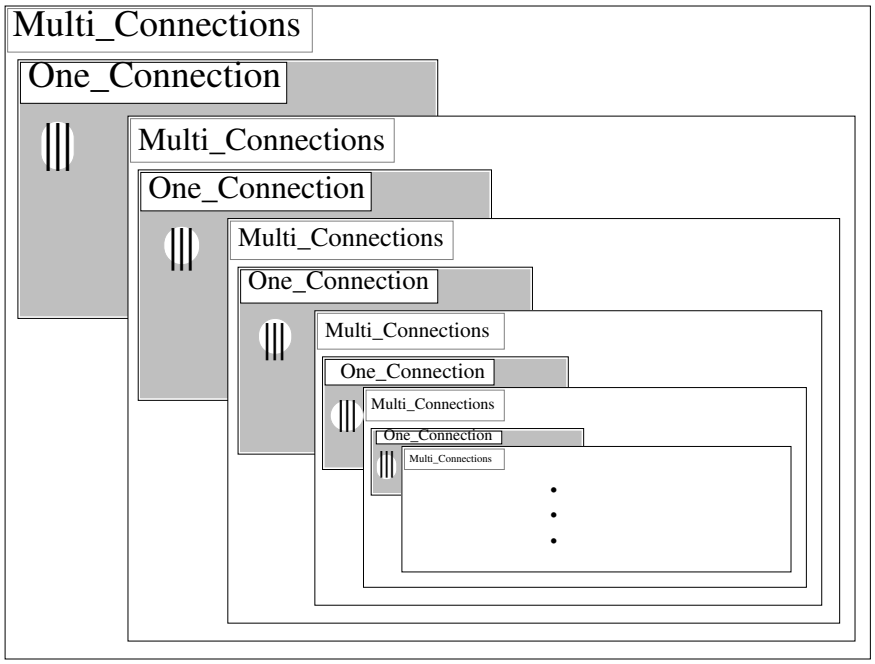


Figure 28. Creating Multiple Connections

A snapshot representation of process *Multi_Connections* is given in Figure 28. The dots indicate that there exist other copies of the process *Multi_Connections*.

[27] The internal action "i" following the *Interleaving* operator is used to prevent the LOTOS interpreter from looping endlessly, by attempting to provide an infinite number of instantiations of process *One_Connection*.

4.4.4 Design of a Single Connection

As shown in Section 3.2.3.3, a single connection consists of a set of subscribers and a connection handler. The connection handler controls the temporal ordering of internal events within a single connection. All internal events are handled on gate *line*.

In LOTOS, this can be represented by composing in parallel the process describing the behaviour of subscribers, process *Subscribers*, with the process describing the behaviour of the connection handler, process *Connection_Handler*.

Since the connection handler has access to only internal events, process *Connection_Handler* must synchronize with the process *Subscribers* on gate *line*. Therefore, we use the LOTOS generalized parallel composition "|[*L*]|", where *L* denotes the set of gates on which processes synchronize. In our representation, *L* is a set that consists of only one gate *line*. The corresponding process is shown in Specification 5 below.

```
process One_Connection [user,line] :noexit:=
  Subscribers [user,line]
  |[line]|
  Connection_Handler [line]
endproc
```

Specification 5. Process One_Connection Definition

Note that gate *user* is used for the interactions of the telephone system with the environment, and therefore it is not subject to a declaration in process *Connection_Handler*.

The process *Subscribers* used in Specification 5 is further refined. It consists of a set of subscribers that are participating in a single connection simultaneously. The behaviour of each subscriber is independent from the others, and therefore we use the LOTOS interleaving operator "|||" to formalize such a behaviour.

Once a subscriber goes *OffHook*, it becomes the originator within a new connection. Its actions are handled in parallel with the actions performed by any specified receiver of that call, see Specification 6 below.

```
process Subscribers [user,line] :noexit:=
  Origination_Side [user,line]
  |||
  Destination_Side [user,line]
endproc
```

Specification 6. Process Subscriber Definition

4.4.4.1 The Origination Side The behaviour of the subscriber initiating a call is described by the process *Origination_Side* presented in Specification 7. After an *OffHook*, the *Origination Side* may get any signal among *DITO*, *BUSY*, and *OOFS*, see Figure 29. This depends on the current status of its line. In LOTOS description, and with respect to the requirements on the use of telephones, see requirement 17 in Section 3.2.7.2, we use the choice operator "**[]**" to select one alternative at a time.

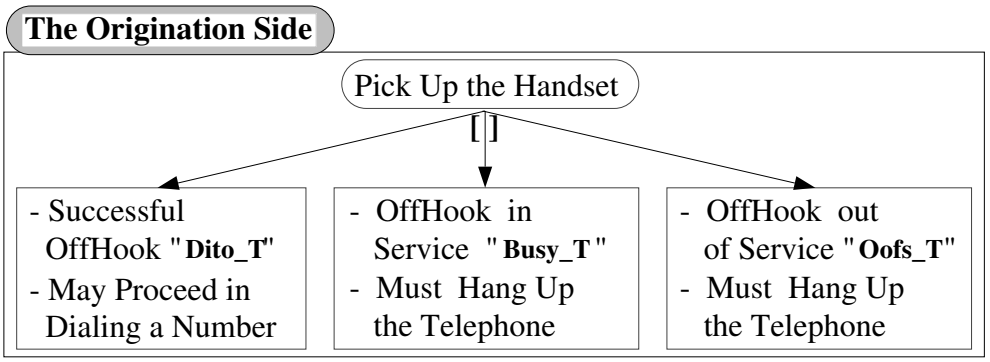


Figure 29. Refinement of the Origination Side

The *Origination Side* may at any time abort the call before establishment of the connection. In LOTOS this can be achieved by using the disabling operator "**[>**", because in the real world, once a call is aborted it cannot be retrieved. This is another advantage of using LOTOS for the formal description of *Telephone Systems*.

As stated in Section 3.2.6, a call processing proceeds in phases. Therefore, we may think of passing from one phase to another in the formal description. In LOTOS, this can be achieved by using the enabling operator "**>>**".

First, each phase is described separately (as a process) and after a successful termination of the first phase, the next phase may begin. Between the two processes we use the enabling operator which provides the specifier with a means to pass information from one process to the other. The first process terminates by an "**exit**(*L*)" with some information (*L*) and the next process begins by an "**accept** *L* **in**" to accept the same information.

In Specification 7, after a successful termination of the first phase (the *Origination Phase*), the *Termination Phase* starts by a connection *CONN*, a detection of a busy signal *BUSY*, or an out of service signal *OOFS*. No information is passed from the first phase. If the *Destination Side*'s line (*RN*) is busy, the *Ring Again* feature may be invoked in this phase. The *Ring Again* feature will be discussed later in Section 4.4.5.1.1.

```
process Origination_Side [user,line] :noexit:=
  user ?N:Phone !OffHook ;
       (* line of initiator is currently free *)
  ( line !N !DITO ;
    (  ( . . .
         [>
              (* initiator aborts the call *)
           Hang_Up [user,line] (N)
       )
       >>
         (* successful termination of the origination phase *)
          (* connection being established *)
       line !N !CONN ;
       User_Talk [user,line] (N)
       []
          (* destination line is currently out of service *)
       line ?RN:Phone !OOFS ;
       . . .
       []
          (* destination line is currently busy *)
       line ?RN:Phone !BUSY ?Set:PhoneSet ;
       user !N !Busy_T ;
```

```
            (* initiator hangs up *)
          ( Hang_Up [user,line] (N)
            []
            (* initiator invokes ring again feature *)
            . . .
            [N <> RN] ->
             Ring_Again_Feature [user,line] (N,RN)
      )   )
      []
          (* line of initiator is currently busy *)
      line !N !BUSY !Empty ;
      . . .
      []
          (* line of initiator is currently out of service *)
      line !N !OOFS;
      . . .
    )
endproc
```

Specification 7. Process Origination_Side Definition

After establishment of a connection *CONN*, the conversation may begin. The conversation is described by process *User_Talk*, shown in Specification 12. However, if a *BUSY* or *OOFS* signal was detected in the origination phase, the *Origination Side* must hang up the telephone.

4.4.4.2 The Destination Side The *Destination Side* is the subscriber that may be involved in a connection once its telephone starts ringing. Its identity is the number dialed by the *Origination Side*, "*CN*" in Specification 7, and its behaviour is described by the process *Destination_Side* shown in Specification 8. Note that during the first phase of a call processing if the *Origination Side* aborts the call, disconnection of the *Destination Side*'s line is not required, we use a **stop** after the disabling operator to terminate the behaviour.

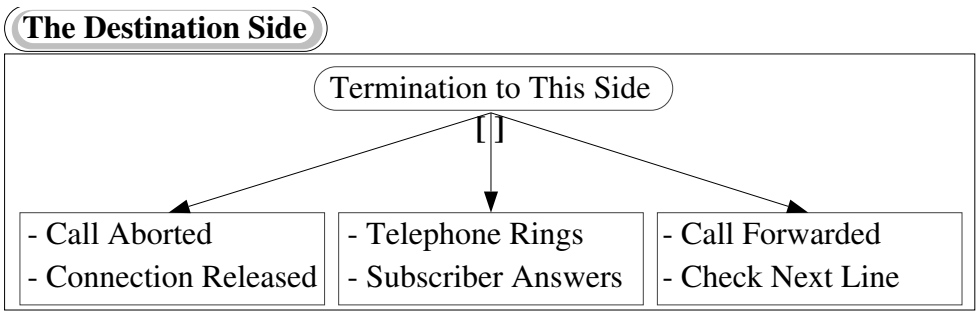


Figure 30. Refinement of the Destination Side

```
process Destination_Side [user,line] :noexit:=
  ( line ?CN:Phone !COAT ;
    exit(CN)
    [>
      stop
  )
  >>
  accept CN:Phone in
     (* termination to destination line *)
  Destination_Phone_Ring [user,line] (CN)
  []
     (* a forward feature is active on this destination line *)
  Call_Forwarded [user,line] (CN)
endproc
```

Specification 8. Process Destination_Side Definition

If the call is not aborted, the control is then passed to the termination phase. The number *CN* is passed by the operator **exit** and is received by the following behaviour through the **accept** construct, see Specification 8. The *Destination Side*'s telephone may then start ringing, "instantiation of process *Destination_Phone_Ring*", unless a *Forward* feature is active on its telephone, in which case the next checks must be applied on the line to whom this call is forwarded. In the latter situation, the number *CN* is passed to process *Call_Forwarded* for special purposes as we will see in Section 4.4.5.1.2. Note the use of the choice

operator **[]** in the second behaviour. This is to force only one alternative at a time, because in the real world, a telephone may ring or the call is forwarded but not both at the same time.

Once it starts ringing, if the *Origination Side* aborts the call or the *Destination Side* answers the call, disconnection of the *Destination Side*'s line is required . After establishment of the connection, the behaviour of the *Destination Side* is described by process *User_Talk*, see Specification 12. Processing *Forward* feature is discussed in Section 4.4.5.1.2.

4.4.4.3 The Connection Handler As discussed above, for each connection an additional process is needed to handle the temporal ordering of events performed at different telephones within that connection. A sample configuration is given in Figure 31. Only internal events are presented within the *Connection Handler* and some of them are shown in the figure "in **bold** type style". To simplify the figure, only some control signals are presented and the remaining sequences of events are omitted.

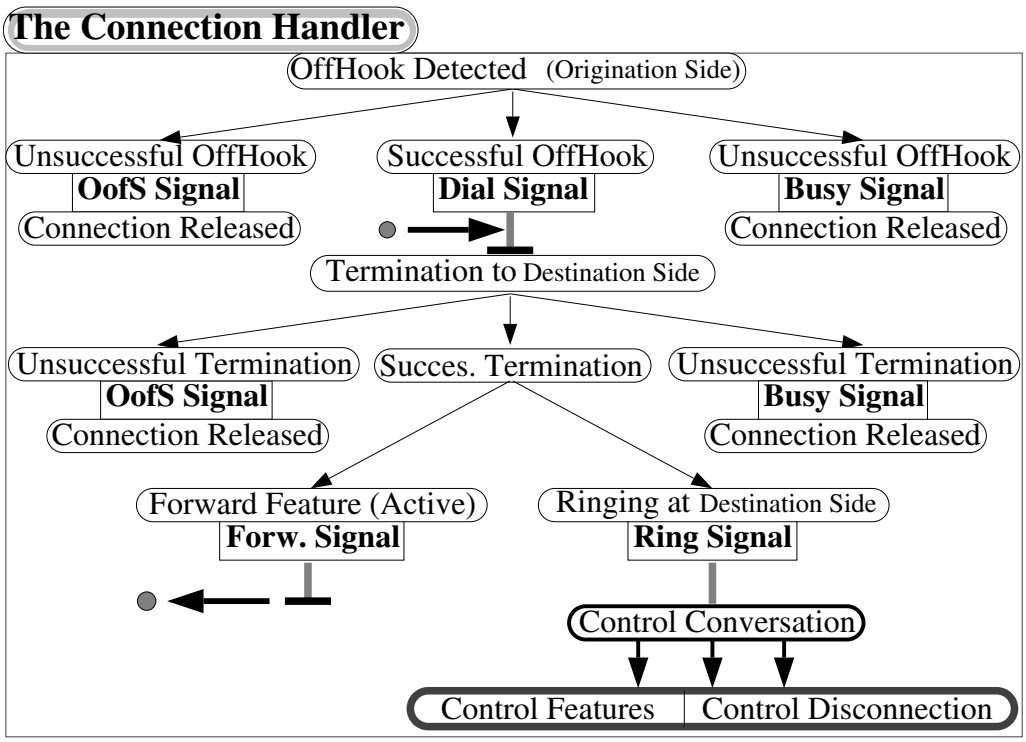


Figure 31. A Configuration of the Connection Handler

In Figure 32 below, we present an example of the temporal ordering of events involved in a simple call where the *OffHook* from the *Origination Side* and the termination of the call are assumed to be successful. Note that the *Connection Handler* synchronizes with the telephones on hidden events only.

The *Connection Handler* synchronizes with the *Origination Side* on event **Dial Tone** and then with the *Destination Side* on event **Ring**. Since the *Origination Side* does not synchronize with the *Connection Handler* on event *OffHook*, it cannot execute the event **Dial Tone** until it has executed the event *OffHook*. Then, the event **Dial Tone** may take place followed by the sequence, *Dial*, **Conn. Request**, and **Conn. Attempt** before the event **Ring** takes place, and so forth.

The same scenario continues with the remaining events. A synchronization forces an internal event to take place, e.g. the internal event **Ring** must be preceded by the internal events **Dial Tone**, **Conn. Request**, and **Conn. Attempt**.

It also forces an observable event to take place if there is no synchronization, e.g. the event *OffHook*[28].

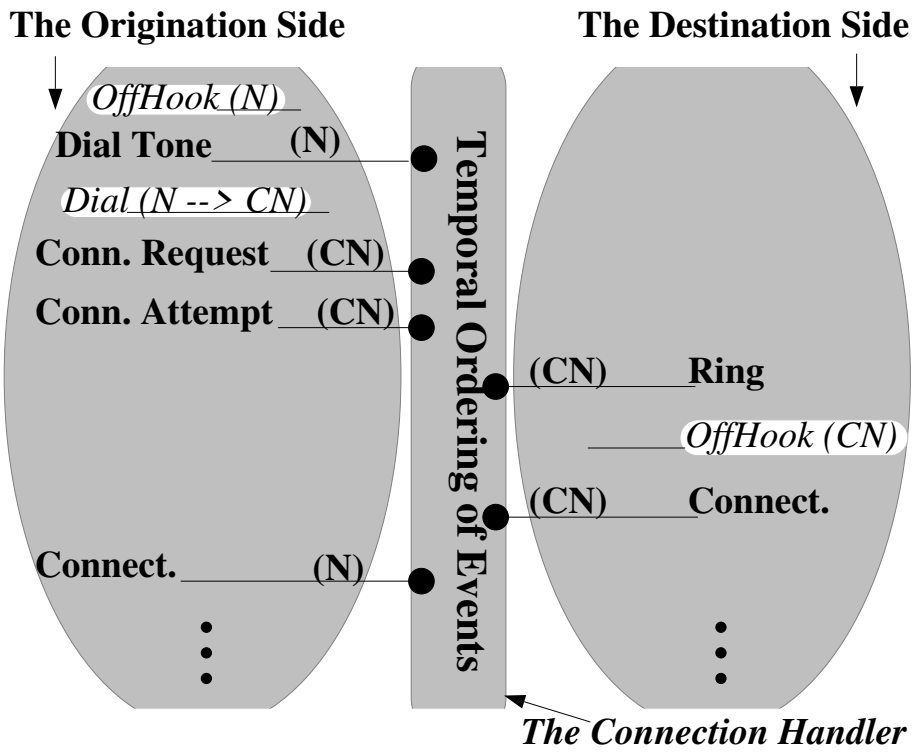


Figure 32. Temporal Ordering of Internal Events in a Normal Call

In our specification, a process called *Connection_Handler*, see Specification 9, is associated with every new connection. During the first stage of a call processing, the *Connection Handler* forces the first (internal) event to be one of **DITO**, **BUSY**, and **OOFS**. It specifies also that this event must take place at the *Origination Side* (*N*).

[28] This is not always true. However, in our sample specification, synchronization is possible only on events internal to the *Telephone System*.

```
process Connection_Handler [line] :noexit:=
  line ?N:Phone !DITO ;
  Control_Termination [line] (N,Empty)
  []
  line ?N:Phone !BUSY !Empty ;
  stop
  []
  line ?N:Phone  !OOFS ;
  stop
endproc
```

Specification 9. Process Connection_Handler Definition

In this figure, only (the internal part of) the first phase of a call processing is presented. The remaining internal behaviour of a call is, however, presented in Appendix B, process *Control_Termination*, lines 558–588.

4.4.5 Specification of the Sample Telephone Features

For modularity of the specification and to facilitate its modification, the features were specified separately, each by a different process. The first stage of the design of the specification is presented using the resource-oriented style.

In the next sections, we will present the formal description, in LOTOS, of the telephone features discussed in Section 3.2.5 with respect to the requirements on their usage, see Section 3.2.7. First, we describe the features *Ring Again* and *Call Forward* which may be invoked during the early stage of a call processing.

The remaining features, *Hold*, *TWC*, *Transfer* and *Conference*, are possible only after establishment of the communication path. Therefore, we prefer to describe them later in a different section.

4.4.5.1 Ring Again and Call Forward Features In this section we focus on two main features, *Ring Again* and *Call Forward*. *Ring Again* feature may be invoked by the initiator of a call, while *Call Forward* feature is to be invoked by the receiver of a call.

4.4.5.1.1 Ring Again Feature The *Ring Again* feature may be invoked by the *Origination Side* when attempting to call a subscriber whose line is currently busy. The formal description of its behaviour is given by process *Ring_Again_Feature* that is instantiated from the process *Origination_Side*, see Specification 7. This feature may be invoked by a subscriber provided that the dialed number is different from its number, "[*N* <> *RN*] —>" where *RN* is the dialed number (the *Destination Side*). However, if on the detection of a busy signal "*BUSY*", the *Origination Side* decides to hang up its telephone, the process *Ring_Again_Feature* is not instantiated, and therefore the connection is released (process *Hang_Up*).

Processing the Feature

To invoke a *Ring Again* feature, the *Origination Side* must first press the appropriate feature key *RiAgK* and then hang up its telephone "*HangUp* ".

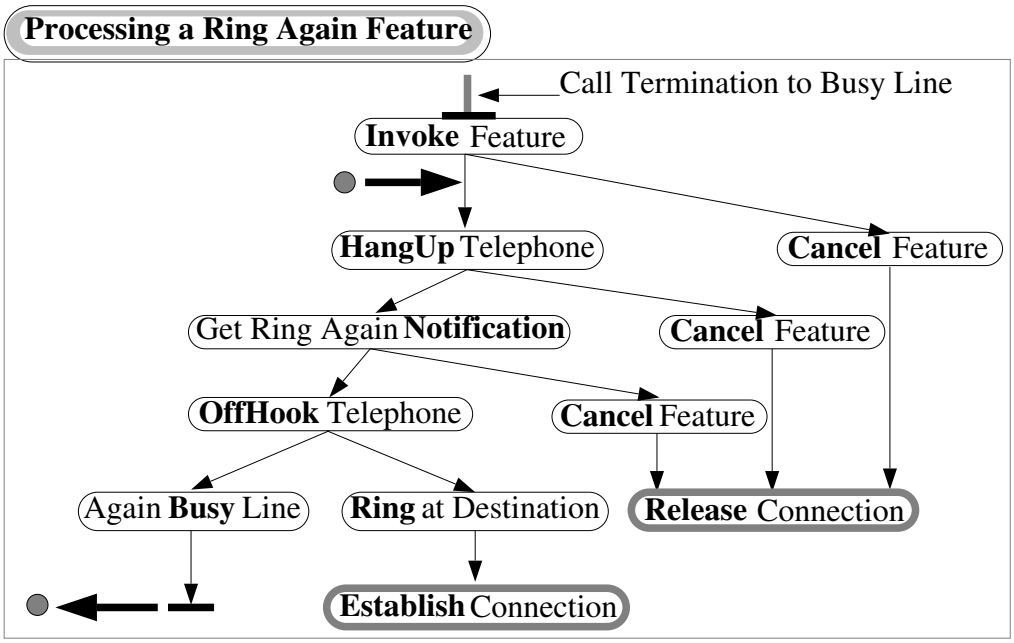


Figure 33. Flowchart of Processing a Ring Again Feature

The *System Network* keeps checking the *Destination Side*'s line until it becomes free. It then notifies the *Origination Side*, by sending to it a *RANO* signal. Meanwhile, the *Origination Side* may, at any time, cancel this feature by pressing

again the same feature key *RiAgK* and the connection will be released *DISC*. The flowchart of invoking and processing the *Ring Again* feature is presented in Figure 33 and the LOTOS description of invoking it is given in Specification 10. Note that this specification starts at the point indicated in Figure 33 by an arrow.

If the *Origination Side* does not cancel the feature and the *Destination Side*'s line becomes free, the *Origination Side* will get a ring again notification "*RANO*". The *Origination Side* may then go *OffHook* and be connected or cancel the feature and the connection is released.

```
process Ring_Again_Feature [user,line] (N,RN:phone)
        :noexit:=
  user !N !HangUp ;
     (* initiator cancels the ring again feature *)
  ( user !N !RiAgK ;
    line !N !DISC ;
    stop
    []
     (* initiator gets a ring again notification *)
    line !N !RANO !RN ;
    user !N !Rano_T ;
       (* initiator picks up telephone after notification *)
    ( user !N !OffHook ;
      line !N !RIAG ;
  (* connection being established (ring at destiantion side) *)
      ( line !N !CONN ;
        User_Talk [user,line] (N)
        []
         (* again destination line is currently busy *)
        line ?RN:Phone !BUSY ?Set:PhoneSet;
        user !N !Busy_T ;
        Ring_Again_Feature [user,line] (N,RN)
      )
      []
       (* initiator cancels feature after notification *)
      user !N !RiAgK ;
```

```
        line !N !DISC ;
        stop
    ) )
endproc
```

Specification 10. Process Ring_Again_Feature Definition

If, after it gets a notification "*RANO* signal", the *Origination Side* goes *OffHook* and finds out that the *Destination Side*'s line is again busy, then it must *HangUp* the telephone and wait for another notification.

4.4.5.1.2 Call Forward Feature The *Forward* feature may be invoked only by the *Destination Side*, and therefore it is instantiated from the process *Destination_Side* as presented in Specification 8. This feature allows a user to forward automatically all further incoming calls to another pre-selected telephone. A *Forward* feature remains active on a telephone until it is cancelled from the same telephone.

When a call terminates to a line, the *System Network* first checks its status. Then, it either applies a *Ring* signal on that telephone if the line is free, by instantiating the process *Destination_Phone_Ring*, or it *Forwards* the call to a pre-selected telephone if the *Forward* feature is active on that telephone, by instantiating the process *Call_Forwarded*.

To invoke the *Forward* feature, the requirements 13, 14, and 15 must be fulfilled, see Section 3.2.7.2. Note that whenever a call is forwarded by a telephone, the line of the latter must be kept busy until establishment of the call or until this call is aborted, see Section 3.2.7.2, requirement 16.

Processing the Feature

Without lifting the handset of the telephone, press the appropriate feature key "*ForK*" then dial the number of the line to whom you want further incoming calls to be forwarded to. The description in LOTOS of invoking a *Forward* feature is given by process *Forward_Feature* presented in Appendix B (lines 1050–1067). Note that a *Forward* feature may be cancelled at any time.

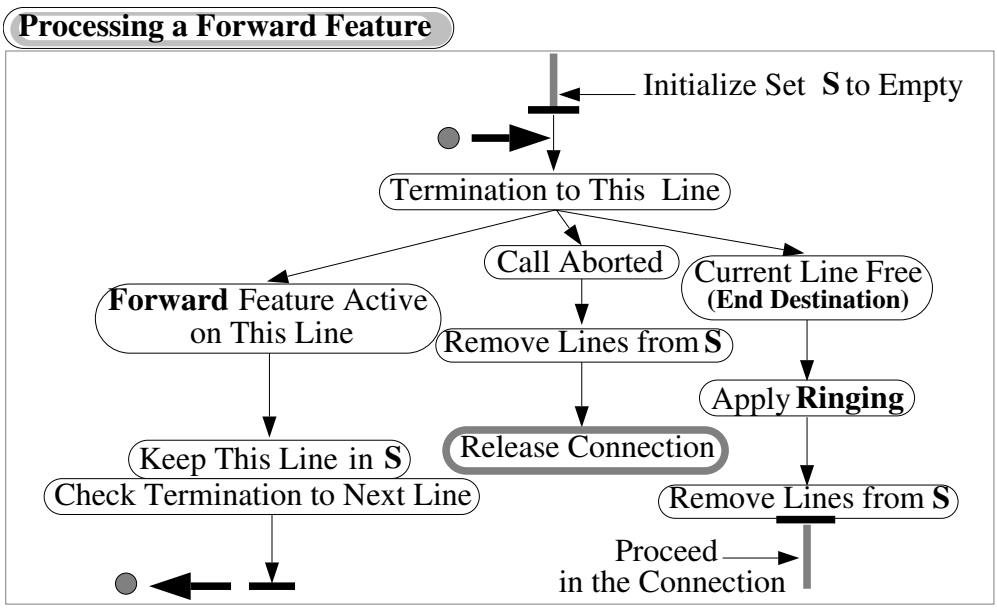


Figure 34. Flowchart of Processing a Forward Feature

Whenever a call terminates to a line, a new temporary set "*S*" is created and initialized to empty. If a *Forward* feature is currently active on an associated telephone[29], its number is added to the set *S*, and further status checks are applied on the line to whom this call is to be forwarded. The use of the set *S* allows us to keep busy all the lines where a *Forward* feature is active for the same call.

If the current call is aborted by the *Origination Side*, all the numbers are removed from the set *S*, and therefore the associated lines are released, see Figure 34. However, if the current call is not aborted, those numbers must be kept busy until establishment of the communication path between the *Origination Side* and the *End Destination Side*[30]. Afterward, all the lines, except the *End Destination Side*'s line, are released. The lines involved in the forwarding are kept busy to be able to detect any cycle when forwarding the same call and also to refuse further incoming calls while a call is being forwarded.

[29] More than one telephone may be associated with the same line.

[30] The *End Destination Side* denotes the line where no *Forward* feature is currently active and an associated telephone rings on termination of the current call to its line.

```
process Call_Forwarded [user,line] (CN:Phone)
        :noexit:=
  line !CN !FORW ;
    (* call being forwarded from destination line CN to FN *)
  line ?FN:Phone !FORW ;
  ( line !FN !CORE ;
    line !FN !COAT ;
    exit(FN)
    [>
       (* call aborted by initiator *)
      stop
  )
  >>
  accept FN:Phone in
    (* current destination line is free, therefore successful
     termination to this destination (End Destination) *)
  Destination_Phone_Ring [user,line] (FN)
  []
    (* call forwarded to another destination (a Forward
     feature is active on the current destination FN) *)
  Call_Forwarded [user,line] (FN)
endproc
```

Specification 11. Process Call_Forwarded Definition

The formal description in LOTOS of forwarding incoming calls is presented in Specification 11. The number *CN* denotes the *Current Destination* while the number *FN* denotes the *Next Destination*, *Call_Forwarded [user,line] (FN)*. If a telephone starts ringing, *Destination_Phone_Ring [user,line] (FN)*, the number *FN* then denotes the *End Destination*.

4.4.5.2 Hold, Transfer/TWC, and Conference Features After establishment of the communication path between subscribers, any subscriber may invoke a feature among *Hold*, *Transfer/TWC*, and *Conference Call*. When a subscriber invokes such a feature, the conversation with the other side is temporarily disabled (by process *Feature*), then the former may proceed in calling a new side. The problem presents itself, of how to represent such temporary disable by using

the LOTOS disable operator. This has been done by reinstantiating the disabled process after the disable operator. If a subscriber goes *OnHook*, the conversation is disabled for good, by process *User_Termination*. This is described by the process *User_Talk* presented in Specification 12.

```
process User_Talk [user,line] (N:Phone) :noexit:=
  Conversation [user,line] (N)
  [>
       (* a hold feature is being invoked on line N *)
     Feature [user,line] (N,HoldK,HOLD,HOLDN)
     []
       (* a transfer/TWC feature is being invoked on line N *)
     Feature [user,line] (N,TransK,TRAN,TRANN)
     []
       (* a conference feature is being invoked on line N *)
     Feature [user,line] (N,ConfK,CONF,CONFN)
     []
       (* terminating this connection *)
     User_Termination [user,line] (N)
endproc
```

Specification 12. Process User_Talk Definition

A process *Feature* is instantiated whenever a feature is invoked. The alternative to be taken depends on the feature key being used (*HoldK*, *TransK*, or *ConfK*).

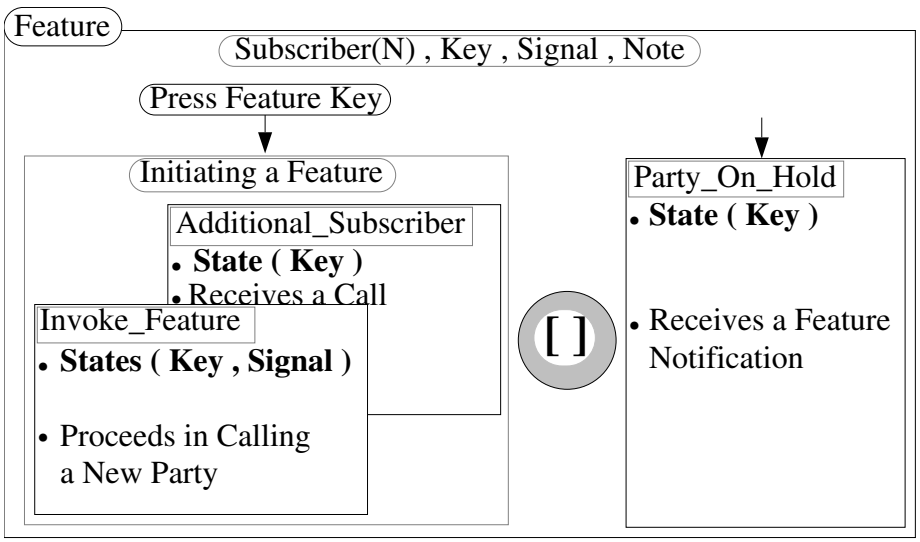


Figure 35. Flowchart of the First Stage of Invoking a Feature

Within the formal description of invoking a feature, the specification is mostly written in the state-oriented style.

```
process Feature [user,line] (N:Phone,K:Key,
        Sign,Note:Signal) :noexit:=
  user !N !K ;
  ( Invoke_Feature [user,line] (N,K,Sign)
    |||
    Additional_Subscriber [user,line] (K)
  )
  []
  Party_On_Hold [user,line] (N,Note)
endproc
```

Specification 13. Process Feature Definition

Process *Feature* describes the behaviour of a subscriber when a feature is invoked. A subscriber may be the invoker of the current feature (*Invoke_Feature*), or the receiver of a feature notification (*Party_On_Hold*). The corresponding LOTOS specifications are shared by the features *Hold*, *Transfer/TWC*, and *Conference Call*. These are given in Sections 4.4.5.2.4, and 4.4.5.2.6.

The process *Invoke_Feature* is interleaved with the process *Additional_Subscriber* whose description is given in Section 4.4.5.2.5. The process *Additional_Subscriber* describes the behaviour of the *New Side* that may be involved in the new call, see Specification 16. When it gets a feature notification, the *New Side* remains on hold until it is re-connected again to the original call. It may also leave the call without informing the invoker of the current feature. The control of events between different subscribers involved in a specific feature within a single connection is handled by the process *Control_Feature* (lines 644–896) in Appendix B. In Sections 4.4.5.2.1, 4.4.5.2.2, and 4.4.5.2.3, we briefly describe each of these features.

4.4.5.2.1 Formal Description of Hold Feature To invoke a *Hold* feature, a subscriber *N* must first press the *HoldK* key, then start dialing the *New Side*'s telephone number *SN*, see Specification 15[31]. If the invoker of the feature doesn't abort the new call, it then gets an audio service depending on the current status of the *New Side*'s line. If the termination of the new call to the *New Side*'s line is unsuccessful, the invoker of the feature may then press the *HoldK* key to be reconnected to the first party, see Figure 36. At this stage, disconnection of the *New Side*'s line is not required. To abort a call, press the *HoldK* key.

[31] The associated action will be "*user* **!N !HoldK ;**" and then process *Invoke_Feature [user,line] (N,***HoldK**,**HOLD***)* is instantiated.

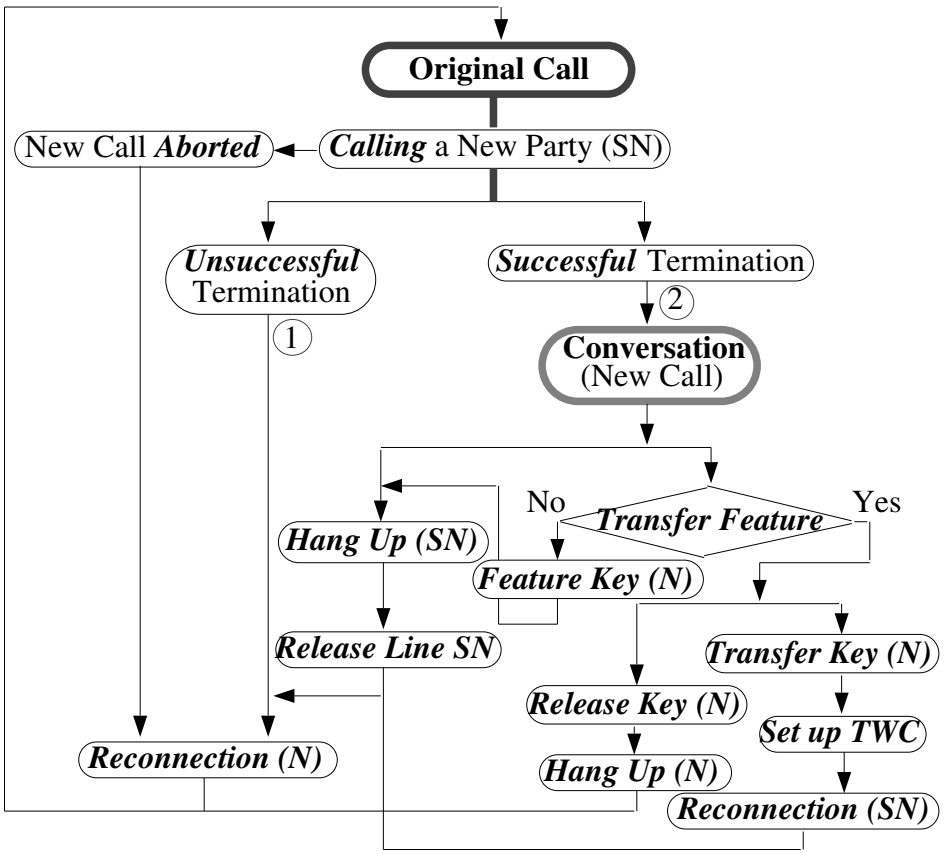


Figure 36. Invoking a Feature within a Conversation

As shown in Figure 36, within a connection, a new call may be initiated. While calling a new party, the invoker of the feature may abort the call before its termination to the new party's line.

If the new call is not aborted, the type of its termination depends on the current status of the new party's line. In (1) in Figure 36, the termination is not successful, and therefore the invoker of the feature is reconnected to the original call "*Reconnection (N)*". In (2), however, the termination is successful and the conversation with the new party may begin. Meanwhile, the new party may go *OnHook* "*Hang Up (SN)*". Afterward, the new party's line is released and the invoker of the feature is reconnected to the original call "*Reconnection (N)*". The

invoker of the feature may also terminate the new call "*Feature Key (N)*" and be reconnected to the original call. The new party "*SN*" must then go *OnHook* and its line is released "*Release Line SN*".

4.4.5.2.2 Formal Description of Transfer/TWC Feature To invoke a *Transfer* feature, a subscriber *N* must first press the *TransK* key, then start dialing the *New Side*'s telephone number *SN*. If a subscriber invokes a *Transfer* feature and remains on the line "*Transfer Key (N)*", then all three parties are connected together in a *TWC* conversation. This feature may also be used to transfer the current call to a third party then leave the conversation "*Release Key (N)*", see Figure 36.

For some incoming call, the *Destination Side* may answer this call or transfer it to a third party "*New Side*" and then leaves the original call. While in conversation, the *New Side* may at any time go *OnHook*, see Specification 16.

If the *Destination Side* transfers an incoming call without leaving it, the transfer feature is then called *TWC* feature. With this feature, all three subscribers are connected together in a conference-like conversation, see Specifications 15, 16, 17, and Figure 36. During such a conversation, any subscriber may go *OnHook*. However, when two subscribers leave the conversation, the third subscriber is then forced to go *OnHook* and terminate the call.

4.4.5.2.3 Formal Description of Conference Call Feature With the *Conference* feature, a subscriber must press the associated feature key, before dialing the *New Side*'s telephone number.

After establishing a connection between the *Origination Side* and the *Destination Side*, either subscriber may invoke a conference feature. When a feature is invoked by one side, the other side is temporarily put on hold "process *Party_On_Hold*". In our specification, this feature allows an arbitrary number of subscribers to be involved in the same connection.

The invoker of a conference feature is specified by the process *Invoke_Feature*, Specification 15. While processing a new call, its initiator may abort it at any time "*Feature Key (N)*", see Figure 36.

If the termination of the new call is unsuccessful, its initiator must press the *ConfK* key to be reconnected to the first call. However, if the call terminates

successfully to the *New Side*'s line, a connection path between the invoker of the feature and the *New Side* is established, then both subscribers may enter in a private conversation.

To bring the *New Side* to a conference conversation, the invoker of the feature must press the *ConfK* key.

The behaviour of each subscriber participating in a conference call is described by process *More_Conference* presented in Specification 14 below.

```
process More_Conference [user,line] (N:Phone)
        :noexit:=
     (* conference conversation *)
   Conversation [user,line] (N)
   [>
        (* conference feature on line N *)
     Feature [user,line] (N,ConfK,CONF,CONFN)
     []
        (* subscriber on line N leaves the conversation *)
     Hang_Up [user,line] (N)
endproc
```

Specification 14. Process More_Conference Definition

At any time during a conference conversation, any participating subscriber may consult privately with a new subscriber, instantiation of the process *Feature* in Specification 14.

Furthermore, any subscriber participating in a conference conversation may leave it for good, by going *OnHook*.

4.4.5.2.4 Specification of the Invoker of a Feature The invoker of a feature is the side that, while connected to a call, may decide to initiate a new call. The formal description of this side is given by the process *Invoke_Feature*, lines 378–426 of the specification presented in Appendix B. A less detailed description of this process is given in Specification 15 below.

In Specification 15, the invoker of a feature is denoted by the number "*N*", while the new party "*End Destination*" involved in the new call is denoted by "*SN*". The corresponding flowchart for this specification is given in Figure 36, and is shared by all the features.

```
process Invoke_Feature [user,line] (N:Phone,
          K:Key,Sign:Signal) :noexit:=
       (* calling a new party *)
  line !N !Sign ;
  ( user !N !DialKs ?SN:Phone ;
    . . .
    [>
          (* the new call aborted *)
      User_Recon [user,line] (N,K)
  )
  >>
  . . .  (* unsuccessful termination of the new call *)
  []
  line !N !CONN ; (* successful termination *)
  ( Conversation [user,line] (N)
    [>
      line !N !DSIN ; (* new party hang up *)
      ( [ K == TransK ] ->
          . . .
        []
        [ K <> TransK ] ->
            (* reconnection of the invoker *)
         User_Recon [user,line] (N,K)
      )
      []
```

```
        [ K == TransK ] ->
              (* the invoker establishes a TWC *)
         ( user !N !TransK ;
           . . .
           []
              (* the invoker transfers a call *)
           user !N !ReleaseK ;
           Hang_Up [user,line] (N)
         )
        []
        [ K <> TransK ] ->
           (* reconnection of the invoker and hang up from
            the new party *)
         User_Recon [user,line] (N,K)
   )
endproc
```

Specification 15. Process Invoke_Feature Definition

To initiate a new call, a subscriber proceeds as in a normal call. After establishing the communication path, the conversation may begin. However, if the *New Side*'s line is *busy* or *out of service*, the invoker must press the same feature key to be reconnected again to the original call.

While in conversation, within the new call, any subscriber may terminate it. To return to the original call, the description depends on the feature used to initiate a new call.

In LOTOS, we associate a special key to each feature and we use it as a state in the specification. The description of the processes *Invoke_Feature*, *Additional_Subscriber*, and *Party_On_Hold* uses mostly the state-oriented style.

4.4.5.2.5 The Additional Subscriber A subscriber that is added to a connection by a *Basic Subscriber*[32] is called an *Additional Subscriber*. This situation may take place when a subscriber invokes any feature among *Hold*, *Transfer/TWC* and *Conference Call*.

[32] A *Basic Subscriber* is the subscriber currently connected to a call. It can be either the *Origination Side* or the *Destination Side*. It can also be any other subscriber already participating in a conference conversation.

If the invoker of a feature aborts the new call before its termination to the other side's telephone, disconnection of the latter's line is not required. Otherwise, if its telephone starts ringing, the *Additional Subscriber* may go *OffHook* and then it will be connected to the invoker of the feature. If the *Additional Subscriber* doesn't go *OffHook*, its telephone keeps ringing until the invoker of the feature aborts the call, and therefore disconnection of the *Additional Subscriber*'s line is required, see process *New_Phone_Ring* (lines 459–486) of the specification presented in Appendix B.

As it is the case in a normal call, the *Additional Subscriber*'s telephone (*SN*) may ring or a *Forward* feature is active on its telephone, and therefore that call is forwarded to the next destination, see Specification 16. For more details, see processes *New_Phone_Ring* (lines 459–486) and *New_Subscriber_Forward* (lines 442–457) of the *Sample Telephone* specification presented in Appendix B.

```
process Additional_Subscriber [user,line] (K:
          Key) :noexit:=
  ( line ?SN:Phone !COAT ;
    exit (SN)
    [>
       stop  (* new call aborted *)
  )
  >>
  accept SN:Phone in
    (* ringing at this side *)
  New_Phone_Ring [user,line] (K,SN)
  []
    (* new call being forwarded *)
  New_Subscriber_Forward [user,line] (K,SN)
endproc
```

Specification 16. Process Additional_Subscriber Definition

While in conversation, an *Additional Subscriber* may, at any time, leave it for good, regardless of the type of the current feature.

If the feature is a *Conference Call* and its invoker presses the same feature key, the *Additional Subscriber* is automatically connected to the original call.

However, if it is a *Transfer* feature and its invoker presses the same feature key, the *Additional Subscriber* is connected to the subscriber already on hold in a normal conversation. Furthermore, if the feature is a *Hold* and its invoker presses the same feature key, the private consultation terminates and the *Additional Subscriber* must go *OnHook*, see Appendix B (lines 468–480).

4.4.5.2.6 The Party On Hold While in conversation, a participating subscriber may invoke a feature, to privately consult with a new party. The party waiting for the completion of the feature is called the *Party On Hold*.

While waiting to be reconnected to the original call, the *Party On Hold* may go *OnHook* without notifying the invoker of the feature. It may also wait until the completion of the new call, then it will be reconnected "*RECO*" again to the original call, see Specification 17.

```
process Party_On_Hold [user,line] (N:
          Phone,Notify:Signal) :noexit:=
  line !N !Notify ;  (* gets notification *)
  ( line !N !RECO ;  (* being reconnected *)
    ( [ Notify == TRANN ] ->
       ( Conversation [user,line] (N)  (* TWC *)
         [>
             line !N !DSIN ;
                (* back to normal call *)
             User_Talk [user,line] (N)
             []
                (* leaves for good the TWC *)
             Hang_Up [user,line] (N)
       )
       []
       [ Notify == CONFN ] ->
            (* conference conversation *)
        More_Conference [user,line] (N)
       []
       [ Notify == HOLDN ] ->
            (* reconnection after a hold feature *)
        User_Talk [user,line] (N)
```

```
      )
      []
      line !N !DSIN ;
         (* the invoker leaves the conversation *)
      line !N !RECO ;
      User_Talk [user,line] (N)
      []
         (* this party leaves the conversation *)
      Hang_Up [user,line] (N)
    )
endproc
```

Specification 17. Process Party_On_Hold Definition

When reconnected to the original call, one of the following situations may occur[33]:

1. if the feature was a *Transfer*, the *Party On Hold* enters in a *TWC* conversation and may leave it at any time. The invoker of the feature may have left the call, and therefore the *Party On Hold* enters in a normal conversation with the *New Party*. The latter case is a transfer of an incoming call.

2. if the feature is a *Conference Call*, the *Party On Hold* is reconnected to a call in a conference conversation and may leave it at any time, see Specification 14.

3. finally, if the feature is a *Hold*, the *Party On Hold* is reconnected again with the invoker of the feature in a normal call and no new party is involved in the current conversation.

4.4.5.3 The Controller of Features Processing The use of each feature is controlled by a process instantiated from within the process *Connection_Handler* that is associated to the current connection. The former controls the order of internal events on all the telephones involved in the same connection, as shown in Figure 37.

[33] When reading these statements, please refer to Specification 17.

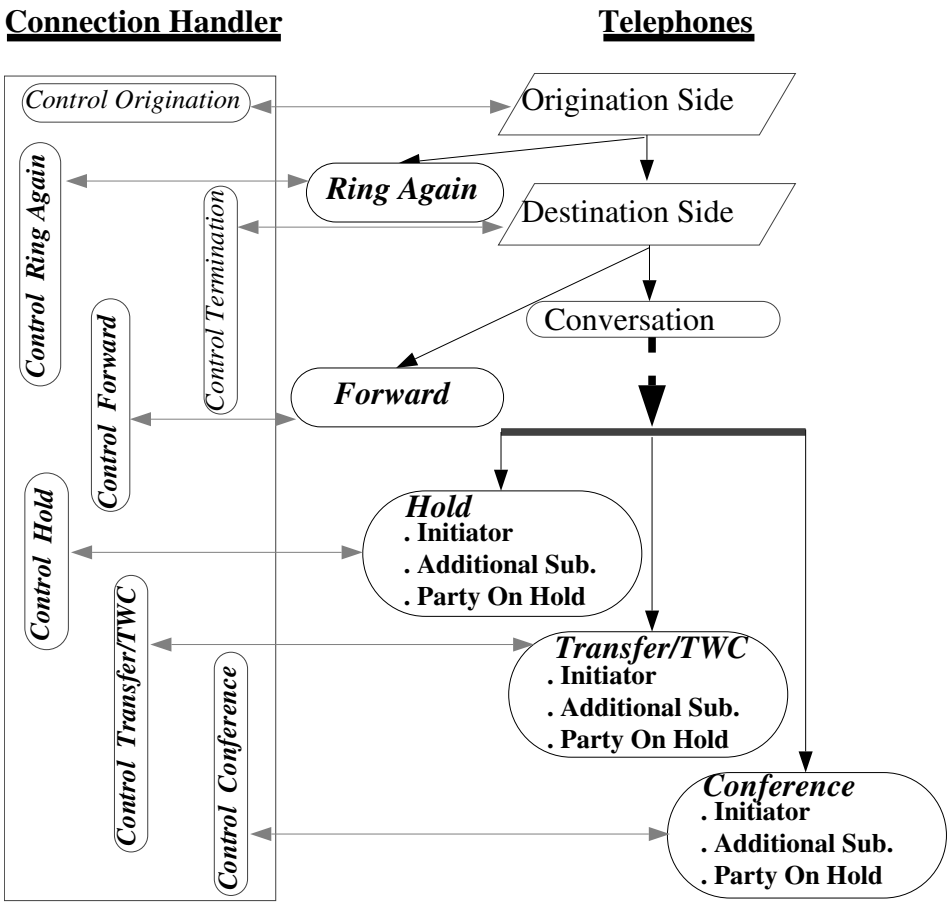


Figure 37. The Controller of Features

1. Instantiating process *Control_Termination* (line 585)[34] to control the events when a call is to be forwarded to another destination.

2. Process *Control_RingAgain* (lines 604–631) controls the events involved when a *Ring Again* feature is invoked by the *Origination Side*.

3. Process *Control_Hold* (lines 655–711) controls the events when a *Hold* feature is requested by either subscriber participating in a call.

[34] The associated numbers refer to the line numbers of the *Sample Telephone Specification* presented in Appendix B.

4. Process *Control_Transfer* (lines 713–786) controls the events when a *Transfer* feature is invoked by the *Destination Side*. It includes also the establishment of a *TWC* conversation.

5. Finally, process *Control_Conference* (lines 795–855) controls the sequence of events involved in a call when a *Conference Call* feature is requested by any subscriber participating in a call.

4.4.6 Formal Description of the System Network

The *System Network*, see Figure 38, interacts with both the environment and the connections. The interactions with the environment on gate *user* allow the *System Network* to keep track of the telephones where a *Forward* feature is active. The *Forward* feature is described separately within the definition of process *System_Network*. In addition, the interactions with the environment on gate *update* allow the latter to add or remove line numbers to/from the set *ServiceSet*.

The second role of the *System Network* is to control the internal events. It checks the status of the associated line and then sends back an appropriate signal. These are handled on gate *line*. Predicates are associated to only the signals of type "*Control_Signal*", see Section 4.3.3. An event with such a signal is possible only if the associated predicate is evaluated to *true*.

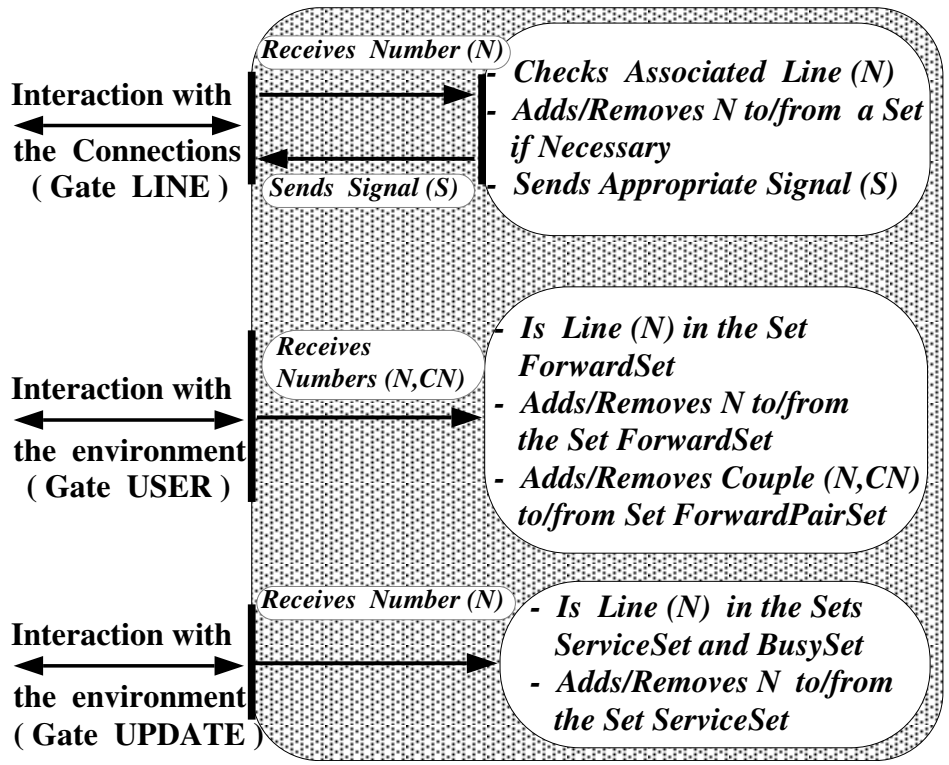


Figure 38. Refinement of the System Network

The corresponding LOTOS description of the process *System_Network* is presented in Specification 18. In this Specification, a predicate is associated to each event. The control signals are not presented in this Specification, but their descriptions along with the associated predicates are presented below.

```
process System_Network [user,update,line]
    (ServiceSet,BusySet,ForwardSet:PhoneSet,
    ForwardPairs:ForwardPairSet) :noexit:=
  update ?N:Phone;
  ( [ NotIn(N,ServiceSet) ] ->
      System_Network [user,update,line] (Insert(N,
      ServiceSet),BusySet,ForwardSet,ForwardPairs)
    []
```

```
    [ IsIn(N,ServiceSet) and NotIn(N,BusySet) ] ->
      System_Network [user,update,line] (Remove(N,
       ServiceSet),BusySet,ForwardSet,ForwardPairs)
    []
    [ IsIn(N,BusySet) ] ->
      System_Network [user,update,line]
       (ServiceSet,BusySet,ForwardSet,ForwardPairs)
  )
  []
  Forward_Feature [user,update,line] (ServiceSet,
   BusySet,ServiceSet,ForwardSet,ForwardPairs)
  []
  line ?N:Phone ?sign:Signal ;
  System_Network [user,update,line] (ServiceSet,
   BusySet,ForwardSet,ForwardPairs)
  []
  line ?N:Phone ! (* control signal *)
  []
  . . .
endproc
```

Specification 18. Process System_Network Definition

The interaction with the environment is defined by the first two actions. In the action "*update* **?***N:Phone*;", a number is added to the set *ServiceSet* "*Insert(N,ServiceSet)*" only if it is not currently in that set "*NotIn(N,ServiceSet)*". A number may be removed from the set *ServiceSet* only if it is in service and is not currently busy "*IsIn(N,ServiceSet) and NotIn(N,BusySet)*". In the second action, a *Forward* feature is invoked and therefore process *Forward_Feature* is instantiated. For more detail on the formal description of invoking a *Forward* feature see the specification presented in Appendix B (lines 1050–1067).

A **control signal** is one of *DITO*, *RING*, *BUSY*, *OOFS*, *DISC*, *RANO*, and *FORW* signals. For each of these signals, we give the predicate to be satisfied in order for that signal to be applied on the associated line[35]. The effect of the corresponding event on the sets of numbers is also presented.

[35] A line is identified by the associated number.

```
DITO signal:
  Associated Predicate
    [ NotIn(N,BusySet) and IsIn(N,ServiceSet) ]
  Effect of the Event:  Insert(N,BusySet)
RING signal:
  Associated Predicate
    [ NotIn(N,BusySet) and IsIn(N,ServiceSet)
      and NotIn(N,ForwardSet) ]
  Effect of the Event:  Insert(N,BusySet)
BUSY signal:
  Associated Predicate
    [ IsIn(N,BusySet) or IsIn(N,TempSet) ]
  Effect of the Event:  no effect
OOFS signal:
  Associated Predicate
    [ NotIn(N,ServiceSet) ]
  Effect of the Event:  no effect
DISC signal:
  Associated Predicate
    [ IsIn(N,BusySet) ]
  Effect of the Event:  Remove(N,BusySet)
RANO signal:
  Associated Predicate
    [ NotIn(TN,BusySet) ]
  Effect of the Event:  no effect
FORW signal:
  Associated Predicate
    [ IsIn(N,ServiceSet) and NotIn(N,BusySet)]
      and IsIn(N,ForwardSet)
  Effect of the Event:  Insert(N,TempSet)
```

Following is a description of the predicates associated with the *control signals*:

A subscriber may get a "*DITO*" signal, only if its line "*N*" is free, and connected to the *Telephone System*. The occurrence of such an event makes the line "*N*" busy.

A telephone may start ringing "*RING*", only if it is connected to the *Telephone System*, free, and no *Forward* feature is active on it. When the event occurs, the associated line becomes busy.

A line "*N*" is busy "*BUSY*" for a given incoming call, if it is currently being used, or another call is currently being forwarded via that line. To keep track of the lines involved in the forwarding process of a call, we use an additional "temporary" set of numbers "*TempSet*". This set is released whenever the communication path is established, or that call is aborted. There are no effects following the occurrence of such an event.

A telephone is out of service "*OOFS*", if the associated line is not connected to the *Telephone system*. There are no effects following the occurrence of such an event.

A line may be disconnected "*DISC*", only if it is currently being used. The occurrence of such an event affects the set of busy lines. The line "*N*" is removed from the set *BusySet*.

A subscriber "*N*" may get a "*RANO*" signal, only when the other side's line "*TN*" becomes free. There are no effects following the occurrence of such an event.

A call may be forwarded "*FORW*" from a telephone whose line is "*N*", only if that line is connected to the *Telephone System*, not being used, and no *Forward* feature is active on that telephone. The associated line is temporarily kept busy "set *TempSet*".

Section 4.5 Debugging the Sample Telephone Specification

4.5.1 Introduction

Using the LOTOS interpreter of the University of Ottawa [HH88], we were able to extensively debug our specification of the *Sample Telephone System*. As mentioned earlier, there are two ways to execute a specification: (1) the specification may be processed by steps, where at each step all the possible actions are presented, and the environment may choose any action by giving its number that appears on the left of that action, as presented in Section 4.5.2, (2) it may also be executed by giving a symbolic tree for the whole specification or for only a specific process, see Section 4.5.3.

4.5.2 Step-by-Step Execution of the Specification

In the Step-by-Step approach, Section 2.4.2, the debugging focuses on specific stages of processing a call. It may also be used to check the consistency of a particular feature by validating at each step the sequence of events.

The Step-by-Step execution helps the specifier to check the specification for specific stages by trying all the possible alternative at that step. It is an interactive execution of the specification, where the environment may be prompted for some data in order for that specific action to be executed.

Following, is an example of executing the *Sample Telephone Specification* using the step by step approach. It tests one functionality of the specification that, if the subscriber whose number is (731–6979) dials the number (741–9729), and the latter's line is free, the associated telephone will ring. This is the simplest case of a call processing described in our specification.

```
                          ISLA
          Interactive System for LOTOS Applications
The available processes are:
[1] Telephone_System[user,update]()
[2] Multi_Connections[user,line]()
[3] One_Connection[user,line]()
[4] Subscribers[user,line]()
    ......................
[12] Feature[user,line](N:Phone,K:Key,Sign,Note:Signal)
    ......................
[22] Connection_Handler[line]()
    ......................
[27] Control_Feature[line](N:Phone,RN:Phone)
    ......................
[49] System_Network[user,update,line](ServiceSet,BusySet,
     ForwardSet:PhoneSet,ForwardPairs:ForwardPairSet)
PROC: lp[rocesses]|?|h[elp]|<process number>|<command> ==> 1
 Enter actual process gates for execution
 ==> Telephone_System
 Applying inferences rules !
+ Choose the whole specification.
================================================================
  <1>- user ?N:Phone !OffHook:Key  ---> bh1   [236]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 3
```

```
 Enter a value for N:Phone => Num(7,3,1,6,9,7,9)
+ Add the line number (7,3,1,6,9,7,9) to the Telephone System.
=====================================================================
  <1>- user ?N:Phone !OffHook:Key  ---> bh1   [236]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- user !Num(7,3,1,6,9,7,9):Phone !ForK:Key ?FN:Phone
  ---> bh4 [1060]
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 3
 Enter a value for N:Phone => Num(7,4,1,9,7,2,9)
+ Add the line number (7,4,1,9,7,2,9) to the Telephone System.
=====================================================================
  <1>- user ?N:Phone !OffHook:Key  ---> bh1   [236]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
  <5>- user !Num(7,3,1,6,9,7,9):Phone !ForK:Key ?FN:Phone
  ---> bh5   [1060]
+ Any telephone among (7,4,1,9,7,2,9) and (7,3,1,6,9,7,9) may
invoke a Forward feature.
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 1
 Enter a value for N:Phone => Num(7,3,1,6,9,7,9)
 Passed evaluated value ==> OffHook
+ The Origination Side goes offhook to initiate a call.
=====================================================================
  <1>- i (specified explicitly)  ---> bh1   [219]
  <2>- update ?N:Phone  ---> bh2   [1004]
  <3>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh3   [1060]
  <4>- user !Num(7,3,1,6,9,7,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
  <5>- i (hiding: line !Num(7,3,1,6,9,7,9):Phone !DITO
  :Control_Signal   [and(NotIn(Num(7,3,1,6,9,7,9),Empty),
  IsIn(Num(7,3,1,6,9,7,9),Insert(Num(7,4,1,9,7,2,9),
  Insert(Num(7,3,1,6,9,7,9),Empty))))]) ---> bh5 [237,548,1020]
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 5
   Internal event is executed
 The existing predicate(s) for this action :
   [and(NotIn(Num(7,3,1,6,9,7,9),Empty),
 IsIn(Num(7,3,1,6,9,7,9),Insert(Num(7,4,1,9,7,2,9),
 Insert(Num(7,3,1,6,9,7,9),Empty))))]
```

```
 evaluated to true
 Passed evaluated value ==> Num(7,3,1,6,9,7,9)
 Passed evaluated value ==> DITO
=====================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !Dito_T:Tone ---> bh1 [238]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 1
 Passed evaluated value ==> Num(7,3,1,6,9,7,9)
 Passed evaluated value ==> Dito_T
+ The Origination Side Gets a dial tone.
=====================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !DialKs:Key ?CN:Phone
  ---> bh1   [239]
  <2>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh2 [541]
  <3>- i (specified explicitly)  ---> bh3   [219]
  <4>- update ?N:Phone  ---> bh4   [1004]
  <5>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh5   [1060]
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 1
 Passed evaluated value ==> Num(7,3,1,6,9,7,9)
 Passed evaluated value ==> DialKs
 Enter a value for CN:Phone ==> Num(7,4,1,9,7,2,9)
+ The Origination Side dials the number (7,4,1,9,7,2,9).
=====================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
  <5>- i (hiding: line !Num(7,4,1,9,7,2,9):Phone !CORE:Signal)
  ---> bh5   [240,559,1017]
=====================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 5
   Internal event is executed
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> CORE
=====================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
```

```
  <4>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
  <5>- i (hiding: line !Num(7,4,1,9,7,2,9):Phone !COAT:Signal)
  ---> bh5   [302,560,1017]
==================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 5
   Internal event is executed
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> COAT
==================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- user !Num(7,4,1,9,7,2,9):Phone !ForK:Key ?FN:Phone
  ---> bh4   [1060]
  <7>- i (hiding: line !Num(7,4,1,9,7,2,9):Phone
  !RING:Control_Signal  [and(and(NotIn(Num(7,4,1,9,7,2,9),
  Insert(Num(7,3,1,6,9,7,9),Empty)),IsIn(Num(7,4,1,9,7,2,9),
  Insert(Num(7,4,1,9,7,2,9),Insert(Num(7,3,1,6,9,7,9),Empty)))S ()),
  NotIn(Num(7,4,1,9,7,2,9),Empty))]) ---> bh7   [333,570,1024]
==================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 7
   Internal event is executed
 The existing predicate(s) for this action :
  [and(and(NotIn(Num(7,4,1,9,7,2,9),Insert(Num(7,3,1,6,9,7,9),
 Empty)),IsIn(Num(7,4,1,9,7,2,9),Insert(Num(7,4,1,9,7,2,9),
 Insert(Num(7,3,1,6,9,7,9),Empty)))),NotIn(Num(7,4,1,9,7,2,9),
 Empty))]   evaluated to true
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> RING
==================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- user !Num(7,4,1,9,7,2,9):Phone !Ring_T:Tone ---> bh2 [334]
  <3>- i (specified explicitly)  ---> bh3   [219]
  <4>- update ?N:Phone  ---> bh4   [1004]
==================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 2
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> Ring_T
+ The Destination Side's telephone is ringing.
==================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- user !Num(7,4,1,9,7,2,9):Phone !OffHook:Key ---> bh2 [337]
  <3>- i (specified explicitly)  ---> bh3   [219]
  <4>- update ?N:Phone  ---> bh4   [1004]
```

```
  <5>- i (hiding: line !Num(7,4,1,9,7,2,9):Phone !ONRI:Signal)
  ---> bh5   [997,997,1017]
=================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 2
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> OffHook
+ The Destination Side answers the call.
=================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- i (specified explicitly)  ---> bh2   [219]
  <3>- update ?N:Phone  ---> bh3   [1004]
  <4>- i (hiding: line !Num(7,4,1,9,7,2,9):Phone !CONN:Signal)
  ---> bh4   [338,573,1017]
=================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 4
   Internal event is executed
 Passed evaluated value ==> Num(7,4,1,9,7,2,9)
 Passed evaluated value ==> CONN
=================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh1 [541]
  <2>- user !Num(7,4,1,9,7,2,9):Phone !SendVo:Key ---> bh2 [528]
  <3>- user !Num(7,4,1,9,7,2,9):Phone !HoldK:Key ---> bh3 [369]
  <4>- user !Num(7,4,1,9,7,2,9):Phone !TransK:Key ---> bh4 [369]
  <5>- user !Num(7,4,1,9,7,2,9):Phone !ConfK:Key ---> bh5 [369]
  <6>- user !Num(7,4,1,9,7,2,9):Phone !HangUp:Key ---> bh6 [541]
  <7>- i (specified explicitly)  ---> bh7   [219]
  <8>- update ?N:Phone  ---> bh8   [1004]
  <9>- i (hiding: line !Num(7,3,1,6,9,7,9):Phone !CONN:Signal)
  ---> bh9   [247,574,1017]
=================================================================
ACT: la[ctions][<N>]|?|h[elp]|<action number>|<command> ==> 9
   Internal event is executed
 Passed evaluated value ==> Num(7,3,1,6,9,7,9)
 Passed evaluated value ==> CONN
=================================================================
  <1>- user !Num(7,3,1,6,9,7,9):Phone !SendVo:Key ---> bh1 [528]
  <2>- user !Num(7,3,1,6,9,7,9):Phone !HoldK:Key ---> bh2 [369]
  <3>- user !Num(7,3,1,6,9,7,9):Phone !TransK:Key ---> bh3 [369]
  <4>- user !Num(7,3,1,6,9,7,9):Phone !ConfK:Key ---> bh4 [369]
  <5>- user !Num(7,3,1,6,9,7,9):Phone !HangUp:Key ---> bh5 [541]
  <6>- user !Num(7,4,1,9,7,2,9):Phone !SendVo:Key ---> bh6 [528]
  <7>- user !Num(7,4,1,9,7,2,9):Phone !HoldK:Key ---> bh7 [369]
  <8>- user !Num(7,4,1,9,7,2,9):Phone !TransK:Key ---> bh8 [369]
  <9>- user !Num(7,4,1,9,7,2,9):Phone !ConfK:Key ---> bh9 [369]
 <10>- user !Num(7,4,1,9,7,2,9):Phone !HangUp:Key ---> bh10 [541]
```

```
 <11>- i (specified explicitly)  ---> bh11   [219]
 <12>- update ?N:Phone  ---> bh12   [1004]
```
+ *The communication path is established between subscribers and any may invoke a feature among (Hold, Transfer/TWC, and Conference Call).*

4.5.3 Symbolic Execution Trees

The symbolic execution method [GL89b] is used to generate a tree of symbolic sequences of events involved in a specification or in a particular part of it. This gives all the possible paths, allowed by the specification or by the selected parts. In the *Sample Telephone Specification*, a symbolic tree represents the behaviour of a particular phase of a call processing or of a particular feature. Note that due to the complexity of the specification, the generation of the whole symbolic execution tree is not practical. However symbolic execution trees for some important processes of the specification were generated using a small depth of execution. The depth is chosen as small as possible provided it covers the main part of the process description.

In this section we present one sample symbolic execution tree. This symbolic tree, see Specification 19, describes the behaviour of process *Origination_Side*, which is the behaviour of a call processing at its first stages. The conversation may begin at level 7. Invoking *Hold*, *Transfer/TWC*, and *Conference Call* features may also take place at this level. A level in this case is the depth of execution of the tree and it is represented by the number of bars displayed on the left of an action. For example, the action corresponding to an *OffHook* event is at level 1. It is followed at level 2 by an internal event, then at level 3 a subscriber may get a dial tone *Dito_T*. At the same level (3), a subscriber may also get a busy tone *Busy_T* or an out of service tone *Oofs_T*. Note that only observable behaviour is presented here "*gate user*".

```
1 user ?Phone@1:Phone !OffHook:Key [236]
| | 1 user !Phone@1:Phone !Dito_T:Tone [238]
| | | 1 user !Phone@1:Phone !DialKs:Key
        ?Phone@4.1:Phone [239]
| | | | | | 1 user !Phone@1:Phone !SendVo:Key [528]
              ==> continue
| | | | | | 2 user !Phone@1:Phone !HoldK:Key [369]
```

```
                ==> continue
| | | | | | 4 user !Phone@1:Phone !TransK:Key [369]
                ==> continue
| | | | | | 6 user !Phone@1:Phone !ConfK:Key [369]
                ==> continue
| | | | | | 8 user !Phone@1:Phone !HangUp:Key [541]
                ==> continue
| | | | | | 1 user !Phone@1:Phone !Oofs_T:Tone [251]
| | | | | | | 1 user !Phone@1:Phone !HangUp:Key [541]
                  ==> continue
| | | | | | 1 user !Phone@1:Phone !Busy_T:Tone [255]
| | | | | | | 1 user !Phone@1:Phone !HangUp:Key [541]
                  ==> continue
| | | | | | | 2 [<>(Phone@1,Phone@6)]user !Phone@1:
                  Phone !RiAgK:Key [259] ==> continue
| | | | | 4 user !Phone@1:Phone !HangUp:Key [541]
              ==> continue
| | | | 2 user !Phone@1:Phone !HangUp:Key [541]
            ==> continue
| | | 2 user !Phone@1:Phone !HangUp:Key [541]
          ==> continue
| | | 1 user !Phone@1:Phone !Busy_T:Tone [265]
| | | 1 user !Phone@1:Phone !HangUp:Key [266]
          DEADLOCK
| | | 1 user !Phone@1:Phone !Oofs_T:Tone [270]
| | | 1 user !Phone@1:Phone !HangUp:Key [271]
          DEADLOCK
```

Specification 19. The Symbolic Execution Tree of the Process Origination_Side

Some symbolic execution trees of different processes are presented in Appendix D. For each of these trees, a brief description of its behaviour is given.

Section 4.6 Testing the Sample Telephone Specification

Testing a specification is an appropriate way to detect errors at the design stage. This can be achieved by composing in parallel a test sequence with the

whole specification, then execute the combined specification. The test sequence is written as a separate process, see Specification 20 below.

Using the tools provided by the interpreter ISLA [HH88][GHHL88], some test sequences have been run on our *Sample Telephone Specification*. The execution of the composition will result in an execution tree [GL89b], from which either the test sequence (process) may reach its last action (in this case we say that it is accepted by the specification), or the test sequence is refused by the specification. Furthermore, the execution tree allows users to see exactly where the error occurs in the specification. This is possible, because in the symbolic tree, for each action are associated action offers participating in that action.

Note that a test sequence must be deterministic in order for the designer to be able to judge its acceptance [Bri88b] by a specification. However, LOTOS concepts allow the specification of non-deterministic systems. Note also that the internal action "**i**" introduced in the specification to prefix the instantiating of process *Multi_Connections* when creating new connections and to prevent the interpreter from looping endlessly, is replaced by the gate *line*. This is to avoid the occurrence of multiple **i**'s in the execution tree and also to allow synchronization of a test process with the specification if the former has the purpose of testing multiple calls in parallel. This method of testing is, of course, a 'white-box' testing [Mye79] in the sense that it presupposes complete knowledge and control of the internal structure of the system.

Test sequences that we feel interesting in our system have been run in parallel with the *Sample Telephone Specification* and have helped us to fix errors at the design stage. Some of these test sequences are presented in Appendix C as individual processes.

The purpose of the test sequence presented in this section, see Specification 20 below, is to verify that, if the subscriber whose telephone number is U1 dials a telephone number U2, and the line on U2 is free, the telephone whose number is U2 will ring. Note that we test only the user interface.

```
process Test_Sequence1 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;      (* Add line U1 to the system *)
  update  !U2 ;      (* Add line U2 to the system *)
  user  !U1 !OffHook ;
        (* Subscriber on U1 goes OffHook *)
  user  !U1 !Dito_T ;
        (* Subscriber on U1 gets a Dial Tone *)
  user  !U1 !DialKs !U2 ;
        (* Subscriber on U1 dials the number U2 *)
  user  !U2 !Ring_T ;
        (* Telephone on U2 rings *)
  user  !U2 !OffHook ;
        (* Subscriber on U2 answers the call *)
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  exit
endproc (* Test_Sequence1 *)
```

Specification 20. A Simple Call Test Process

The corresponding symbolic tree is presented below, in Specification 21. A subscriber may start talking at level 15 (count the number of bars that precede the action associated with *SendVo*).

```
1 update !Num(7,3,1,6,9,7,9):Phone [1025,223]
| 1 [NotIn(Num(7,3,1,6,9,7,9),Empty)]update
    !Num(7,4,1,9,7,2,9):Phone [1025,224]
| | 1 user !Num(7,3,1,6,9,7,9):Phone
      !OffHook:Key [257,225]
| | | | 1 user !Num(7,3,1,6,9,7,9):Phone
          !Dito_T:Tone [259,226]
| | | | | 1 user !Num(7,3,1,6,9,7,9):Phone !DialKs:Key
            !Num(7,4,1,9,7,2,9):Phone [260,227]
| | | | | | | | | 1 user !Num(7,4,1,9,7,2,9):Phone
                    !Ring_T:Tone [355,228]
```

```
| | | | | | | | | | 2 user !Num(7,4,1,9,7,2,9):Phone
                        !OffHook:Key [358,229]
| | | | | | | | | | | | | 1 user !Num(7,3,1,6,9,7,9):
                          Phone !SendVo:Key [549,230]
| | | | | | | | | | | | | | 2 user !Num(7,4,1,9,7,2,9):
                            Phone !SendVo:Key
                            [549,231] ==> continue
```

Specification 21. The Symbolic Execution Tree of the Simple Call Test Process

Once we generate the corresponding symbolic tree, we may check its status. The last event in our test sequence is **SendVo** from U2. In the symbolic tree, the last event is also **SendVo** from U2, and therefore the test sequence is accepted by the specification. The same process of testing the specification is repeated with the remaining test processes.

Section 4.7 Conclusion

In this Chapter, the *Sample Telephone System* described in Section 3.2 is specified formally using LOTOS. Some of the particularities of telephone systems are discussed and the way that can be described in LOTOS is presented. All the concepts introduced in the specification are explained and examples are given. The resulting specification is validated using an interpreter.

The debugging of a specification is a very important task to validate a design. It helps the designer to check its design at each stage, and fix design errors at early stages before the final realization of a system is achieved.

Chapter 5 Conclusions & Future Work

Section 5.1 Conclusions

A detailed description of a *Sample Telephone System* has been given, after presenting an overview of telecommunication systems structure and design. We have also presented a design methodology to specify *Telephone Systems* in the FDT language LOTOS. By this experience, LOTOS was found to be appropriate for the specification of the service provided by telephone systems. We have also shown how more advanced telephone features can be specified and included in the initial specification of a system. Furthermore, we found that LOTOS specifications display a clear and intuitive structure, and therefore LOTOS is appropriate for the design of complex systems.

LOTOS has formal semantics, allowing verification of certain aspects of a design. The availability of tools provided by the University of Ottawa interpreter allowed us to extensively debug the *Sample Telephone Specification* and to test its design at each stage. Note that the execution trees provided by LOTOS Toolkit enable design validation, and therefore design errors can be caught before the implementation phase.

A method to debug LOTOS specifications was presented and test sequences were validated using an interpreter. By precisely specifying the features of the system at the design stage, the development team can achieve a better view of the system's functionality. The specification can then be validated by various debugging, testing, and verification methods, and therefore costly design errors can be detected at the design stage. Finally, an implementation can be obtained from the formal specification. Test cases for the implementation can also be derived from the specification. The latter topic is not covered in this thesis, however the related methodology is the subject of active research.

As discussed above, the step-by-step execution (Section 2.4.4) of a specification is very slow, and therefore it is more likely to be used at the first stages of the design. However, the symbolic execution tree method (Section4.5.3) may be used to check the different possible paths allowed by the specification for a spe-

cific case, in particular when that case is too far to be reached by the step-by-step execution approach.

The final stages, however, are more likely to be tested by composing in parallel test processes with the specification and then checking for their acceptance by the specification, see Section 4.6. This is because it is hard to check large sequences in the generated symbolic tree or to execute them using the step-by-step method.

The LOTOS specification of the *Sample Telephone System* is given in Appendix B, and test sequences are presented in Appendix C. Also, some symbolic execution trees of processes are presented in Appendix D.

Section 5.2 Specification of Timing Characteristics

The specification of the *Sample Telephone System* presented in this thesis does not describe 'real-time' characteristics such as time-out, and similar. This is due to the fact that facilities for the specification of real-time concepts are not (yet) part of the standard FDT language LOTOS (in Section 3.2.3, we mention some proposed extensions of LOTOS to deal with this problem). In spite this fact, there are different ways to explicitly specify the time within a specification. One way is to use a clock as a process composed in parallel with the processes of the specification. This process synchronizes with every action and offers to this action the current time. It then increments the time by one unit. This is possible by using an additional variable (time) over the gate carrying the action. Another simpler way is to specify timing events such as 'start', 'expired', and 'reset' as actions of a process and then compose this process in parallel with the behaviour expression where timing is considered. As an example, we take the process *Origination_Side* that describes the actions performed by the initiator of a call. In this process, the initiator goes *OffHook* and after a certain period of time if it does not complete dialing a number, the given allowed time, represented by the internal event **i**, may elapse. So, in process *Timer*, the first action is to start the timer 'start'. Then after a while, the time may elapse 'expired'. If not, the initiator has either completed dialing the called number or aborted the call, and therefore the timer has to be reset 'reset' for further use by any other behaviour expression.

```
process Origination_Side [user,line] :noexit:=
  hide t in
  ((user ?N:Phone !OffHook ;
    ( line !N !DITO ;
      user !N !Dito_T ;
      t !start ;
      ( user !N !DialKs ?CN:Phone ;
        line !CN !CORE ;
        exit
        [>
           ( t !expired ;
             Hang_Up [user,line] (N)
             []
             ( Hang_Up [user,line] (N)
               >>
               t !reset ;
               stop
      )   ) )
      >>
      t !reset ;
      . . .
    ))
    |[t]|
    Timer [t]
  )
endproc
```

The process *Timer* may be used by several other behaviour expressions or processes. It states that timing starts and expires after a certain period of time. This may be handled by the environment while simulating the execution of the specification. If the time expires, the internal action '**i**' is executed before the action associated with the constant 'expired'. This triggers the disable in the timed-out process. However, if the time does not expire, the corresponding behaviour expression terminates within the required time and the action associated with the constant 'reset' is executed.

```
process Timer [t] :noexit:=
  t !start ;
  ( i ;
    t !expired ;
    stop
    [>
      t !reset ;
      stop
  )
endproc (* Timer *)
```

So, the time-out 'expired' event can occur at any time after the timer has been started. The timer may also be reset before the required time elapses. This method of specifying time-out was used in [Gue89] for the specification of the LAP-B protocol.

In this work, we have chosen not to include specification of time, in order to keep the complexity to a minimum. We recognize, however, that time would have to be specified in order to portray faithfully the system behaviour.

Section 5.3 Future Work

Since timing is very important in telephone systems, a good project would be to write a similar LOTOS specification where timing concepts are introduced as mentioned in the previous section. However, as mentioned, if the introduction of explicit real time is desired, this will have to wait until LOTOS and related tools are enhanced.

In our thesis, a number of telephone features have been specified. Many other existing features have not been included, but it is still possible to add them with little change. This is because the specification is written in a very structured way to allow its future modification.

Once a specification has been debugged and its design is tested[36], we can proceed to code the implementation. Translators from LOTOS to C are available [MdMvT89]. The specification can also be used to derive test cases for the implementation.

[36] Testing the design of the specification, refers to testing only the user interface of the specification.

Appendix A Technical Abbreviations

Section A.1 Keys

Following is a list of the keys available on the *Sample Keyboard* **DMS-100** shown in Figure 10 [Bel88]. Interactions of the *Environment* "*Subscribers*" with the *Telephone System* are achieved by means of these keys. These keys allow also the subscribers to invoke the features described in section 3.2.5.

Basically, there are two features keys available on the *Business Set* keyboard. These are *Three-Way Calling* and *Call Transfer* feature keys. However, any other feature may be activated by pressing either the *Three-Way Calling* or the *Call Transfer* feature key and then dialing the appropriate feature code. For more details on the keys available on the **DMS-100** keyboard, see Section 3.2.2.

ConfK : Conference Feature Key
DialKs : Dial Keys Available on the Keyboard
ForK : Forward Feature Key
HangUp : Put Handset in its Cradle, it causes Telephone to go OnHook
HoldK : Hold Feature Key
OffHook : pick up Handset
ReleaseK : Release Key, Similar to a HangUp but with Handset out of its Cradle
RiAgK : Ring Again Feature Key
SendVo : Talk Through the Telephone
TransK : Transfer Feature Key

Section A.2 Signals

The following list groups the signals sent/received by the *System Network* to/from the telephones. These signals correspond to the internal events within the *Telephone System*.

BUSY : Busy Signal
COAT : Connection Attempting Signal
CONF : Conference Signal
CONFN : Conference Notification Signal
CONN : Connection Signal
CORE : Connection Request Signal
DISC : Disconnection Signal
DITO : Dial Tone Signal
DSIN : Disconnection Indication Signal
FORW : Forward Signal
HOLD : Hold Signal
HOLDN : Hold Notification Signal
ONRI : Repeat Ring Signal
OOFS : Out Of Service Signal
RANO : Ring Again Notification Signal
RECO : Reconnection Signal
RIAG : Ring Again Signal
RING : Ring Signal
TRAN : Transfer Signal
TRANN : Transfer Notification Signal
VOICE : Gets Voice on the Telephone

Section A.3 External Tones

These are special tones heard by subscribers through their telephones. Such a tone results from an appropriate signal detected by the same telephone. Depending on the current status of a line, first an internal signal, see Section A.2, is sent, then if it can be detected by a subscriber, the same signal is sent again to the subscriber via its telephone[37].

[37] To make it clear, a different operation is used on the gate "*user*". For example, when a subscriber gets a dial tone, first the signal "*DITO*" is exchanged via gate *line*, then an external tone "*Dito_T*" is sent to the environment via gate *user*.

Busy_T : Busy Tone Heard by a Caller
Dito_T : Dial Tone Heard by a Caller
Oofs_T : An Out of Service Tone Heard by a Caller
Rano_T : A Ring Applied on the Caller's Telephone When the Callee's Telephone Becomes Free
Ring_T : A Ring Applied on the Callee's Telephone if it is Found Free

Section A.4 Other Abbreviations

Following is a list of some other abbreviations used in this thesis.

CCITT : Consultative Committee for International Telegraph and Telephone
CCS : A Calculus of Communicating Systems
CSP : Communicating Sequential Processes
FDT : Formal Description Technique
ISLA : An Interactive System for LOTOS Applications
ISO : International Organization for Standardization
LOTOS : Language Of Temporal Ordering Specifications
OSI : Open Systems Interconnection
POTS : Plain Old Telephone Systems
SDL : Specification and Description Language

Appendix B LOTOS Specification of the Sample Telephone System

Following is the LOTOS specification of the *Sample Telephone System*. The associated numbers are not part of the LOTOS specification, but are used only to show the line numbers referred to from the specification.

```
   1  specification Sample_Telephone [user,update]
                    :noexit
   2     (* Abstract Data Types Definitions *)
(* Use of the data types NaturalNumber and Boolean from the LOTOS
   standard library.
*)
   3  library NaturalNumber,Boolean endlib
   4
(* Define the operations == and <> on NaturalNumber type. *)
   5  type Enriched_Nat is NaturalNumber
   6       opns
   7           _ == _ : Nat, Nat -> Bool
   8           _ <> _ : Nat, Nat -> Bool
   9       eqns
  10           forall m,n:Nat
  11           ofsort Bool
  12             m == n = m eq n;
  13             m <> n = not(m eq n)
  14  endtype (* Enriched_Nat *)
  15
(* Define digits to be used by (phone) numbers. To_Nat is the
   operation to map to natural numbers defined in the library.
*)
  16  type Digits is Enriched_Nat,Boolean
  17       sorts Digit
  18       opns
  19           0,1,2,3,4,5,6,7,8,9 : -> Digit
  20           To_Nat : Digit -> Nat
  21           _ == _ : Digit, Digit -> Bool
```

```
  22            _ <> _ : Digit, Digit -> Bool
  23        eqns
  24            forall N,M:Digit
  25            ofsort Nat
  26              To_Nat(0) = 0;
  27              To_Nat(1) = Succ(To_Nat(0));
  28              To_Nat(2) = Succ(To_Nat(1));
  29              To_Nat(3) = Succ(To_Nat(2));
  30              To_Nat(4) = Succ(To_Nat(3));
  31              To_Nat(5) = Succ(To_Nat(4));
  32              To_Nat(6) = Succ(To_Nat(5));
  33              To_Nat(7) = Succ(To_Nat(6));
  34              To_Nat(8) = Succ(To_Nat(7));
  35              To_Nat(9) = Succ(To_Nat(8));
  36            ofsort Bool
  37              M == N = To_Nat(M) == To_Nat(N);
  38              M <> N = To_Nat(M) <> To_Nat(N);
  39  endtype (* Digits *)
  40
(* Define a type Phone_Num and some operations on it. *)
  41  type Phone_Num is Digits,Boolean
  42    sorts Phone
  43    opns
(* A phone number (type Phone_Num) consists of seven digits. *)
  44      Num : Digit,Digit,Digit,Digit,
                Digit,Digit,Digit -> Phone
  45      Num : Digit,Digit,Digit,Digit,
                Digit,Digit -> Phone
  46      Num : Digit,Digit,Digit,Digit,Digit -> Phone
  47      Num : Digit,Digit,Digit,Digit -> Phone
  48      Num : Digit,Digit,Digit -> Phone
  49      Num : Digit,Digit -> Phone
  50      Num : Digit -> Phone
  51      _ == _ : Phone, Phone -> Bool
  52      _ <> _ : Phone, Phone -> Bool
  53    eqns
```

```
  54    forall D11,D12,D13,D14,D15,D16,D17,D21,D22,
  55           D23,D24,D25,D26,D27 :Digit, M,N:Phone
  56    ofsort Bool
(* The comparison of two phone numbers requires the comparison
   of all their digits respectively.
*)
  57     (D17 == D27) and (D16 == D26) and (D15 == D25)
  58     and (D14 == D24) and (D13 == D23) and
  59     (D12 == D22) and (D11 == D21) =>
  60     Num(D17,D16,D15,D14,D13,D12,D11) ==
  61     Num(D27,D26,D25,D24,D23,D22,D21) = true;
  62
  63     (D17 <> D27) or (D16 <> D26) or (D15 <> D25)
  64     or (D14 <> D24) or (D13 <> D23)
  65     or (D12 <> D22) or (D11 <> D21) =>
  66     Num(D17,D16,D15,D14,D13,D12,D11) ==
  67     Num(D27,D26,D25,D24,D23,D22,D21) = false;
  68
  69     M <> N = not(M == N);
  70    ofsort Phone
  71     Num(D17) = Num(0,0,0,0,0,0,D17);
  72     Num(D16,D17) = Num(0,0,0,0,0,D16,D17);
  73     Num(D15,D16,D17) = Num(0,0,0,0,D15,D16,D17);
  74     Num(D14,D15,D16,D17) = Num(0,0,0,D14,D15,
             D16,D17);
  75     Num(D13,D14,D15,D16,D17) = Num(0,0,D13,D14,
             D15,D16,D17);
  76     Num(D12,D13,D14,D15,D16,D17) = Num(0,D12,D13,
             D14,D15,D16,D17);
  77  endtype (* Phone_Num *)
  78
(* Define a set type PhoneSet on the phone numbers and some
   useful operations on it.
*)
  79  type PhoneSet is Phone_Num,Boolean
  80     sorts PhoneSet
  81     opns
```

```
 82      Empty : -> PhoneSet
 83      Insert: Phone,PhoneSet -> PhoneSet
 84      Remove: Phone,PhoneSet -> PhoneSet
 85      IsIn  : Phone,PhoneSet -> Bool
 86      NotIn : Phone,PhoneSet -> Bool
 87      Is_Empty : PhoneSet -> Bool
 88      Head : PhoneSet -> Phone
 89      Tail : PhoneSet -> PhoneSet
 90     eqns
 91      forall x,y:Phone, s:PhoneSet
 92      ofsort PhoneSet
 93       Remove(x,Empty)        = Empty ;
 94       Insert(x,Insert(x,s)) = Insert(x,s);
 95       Remove(x,Insert(x,s))    = s;
 96       Remove(x,Insert(x,Empty))= Empty ;
 97       x <> y => Remove(x,Insert(y,s)) =
                    Insert(y,Remove(x,s));
 98       Tail(Insert(x,s))  = s;
 99       Tail(Empty)        = Empty;
100      ofsort Bool
101       IsIn(x,Empty)      = false;
102       IsIn(x,Insert(y,s))=(x == y) or(IsIn(x,s));
103       IsIn(x,Remove(x,s))= false;
104       x <> y => IsIn(x,Remove(y,s))= IsIn(x,s);
105       NotIn(x,s)         = not(IsIn(x,s));
106       Is_Empty(Empty)    = true;
107       Is_Empty(Insert(x,s))= false;
108      ofsort Phone
109       Head(Insert(x,s))= x;
110  endtype (* PhoneSet *)
111
(* Define a type PhonePair and some useful operations on it.
   An operation of type PhonePair requires two phone numbers.
*)
112  type PhonePair is Phone_Num, Boolean
113    sorts PhonePair
```

```
 114    opns
 115     ForPair : Phone,Phone -> PhonePair
 116     _==_,_<>_: PhonePair,PhonePair -> Bool
 117    eqns
 118     forall f1,f2,f3,f4:Phone
 119     ofsort Bool
 120      ForPair(f1,f2) == ForPair(f3,f4)= (f1 == f3)
                            and (f2 == f4);
 121      ForPair(f1,f2) <> ForPair(f3,f4)= (f1 <> f3)
                            or (f2 <> f4);
 122  endtype (* PhonePair *)
 123
(* Define a second set type ForwardPairSet and some operations
   on it. The elements of this set are of type PhonePair.
*)
 124  type ForwardPairSet is PhonePair
 125    sorts ForwardPairSet
 126    opns
 127     NoPair: -> ForwardPairSet
 128     Insert: PhonePair,ForwardPairSet->
                 ForwardPairSet
 129     Remove: PhonePair,ForwardPairSet->
                 ForwardPairSet
 130     IsIn  : PhonePair,ForwardPairSet-> Bool
 131     NotIn : PhonePair,ForwardPairSet-> Bool
 132     ForwardTo: Phone,ForwardPairSet    -> Phone
 133    eqns
 134     forall x,y:PhonePair,s:ForwardPairSet,
                p1,p2,p3:Phone
 135     ofsort ForwardPairSet
 136      Insert(x,Insert(x,s)) = Insert(x,s);
 137      Remove(x,Insert(x,s)) = s;
 138      Remove(x,Insert(x,NoPair))= NoPair ;
 139      x <> y => Remove(x,Insert(y,s))=
                    Insert(y,Remove(x,s));
 140     ofsort Bool
```

```
 141      IsIn(x,NoPair)      = false;
 142      IsIn(x,Insert(y,s))= (x == y) or
                               (IsIn(x,s));
 143      IsIn(x,Remove(x,s))= false;
 144      x <> y => IsIn(x,Remove(y,s)) = IsIn(x,s);
 145      NotIn(x,s)          = not(IsIn(x,s));
 146     ofsort Phone
 147      ForwardTo(p1,Insert(ForPair(p1,p2),s))= p2;
 148      p1 <> p2 => ForwardTo(p1,Insert(ForPair
                     (p2,p3),s))= ForwardTo(p1,s);
 149  endtype (* ForwardPairSet *)
 150
(* Define the keys as of type Key with their comparisons. *)
 151  type Key  is Boolean
 152    sorts Key
 153    opns
 154     OffHook,DialKs,SendVo,HangUp,ReleaseK,
 155     HoldK,RiAgK,TransK,ConfK,ForK :-> Key
 156     _==_,_<>_ : Key,Key -> Bool
 157    eqns
 158     forall x,y:Key
 159      ofsort Bool
 160    HoldK == HoldK = true ;HoldK <> HoldK = false;
 161    HoldK == TransK= false;HoldK <> TransK= true ;
 162    HoldK == ConfK = false;HoldK <> ConfK = true ;
 163    TransK== HoldK = false;TransK<> HoldK = true ;
 164    TransK== TransK= true ;TransK<> TransK= false;
 165    TransK== ConfK = false;TransK<> ConfK = true ;
 166    ConfK == HoldK = false;ConfK <> HoldK = true ;
 167    ConfK == TransK= false;ConfK <> TransK= true ;
 168    ConfK == ConfK = true ;ConfK <> ConfK = false;
 169  endtype (* Key *)
 170
(* Define the tones as of type Tone. *)
 171  type Tone is
 172    sorts  Tone
```

```
 173    opns
 174     Busy_T,Dito_T,Oofs_T,
 175     Rano_T,Ring_T :-> Tone
 176  endtype (* Tone *)
 177
(* Define normal signals as of type Signal with their
   comparisons.
*)
 178  type Signal is Boolean
 179    sorts  Signal
 180    opns
 181     COAT,CONF,CONFN,CONN,CORE,
 182     HOLD,HOLDN,ONRI,RECO,RIAG,
 183     TRAN,TRANN,VOICE,DSIN :-> Signal
 184     _==_,_<>_ : Signal,Signal -> Bool
 185    eqns
 186     forall x,y:Signal
 187     ofsort Bool
 188    HOLDN == HOLDN = true ;HOLDN <> HOLDN = false;
 189    HOLDN == TRANN = false;HOLDN <> TRANN = true ;
 190    HOLDN == CONFN = false;HOLDN <> CONFN = true ;
 191    TRANN == HOLDN = false;TRANN <> HOLDN = true ;
 192    TRANN == TRANN = true ;TRANN <> TRANN = false;
 193    TRANN == CONFN = false;TRANN <> CONFN = true ;
 194    CONFN == HOLDN = false;CONFN <> HOLDN = true ;
 195    CONFN == TRANN = false;CONFN <> TRANN = true ;
 196    CONFN == CONFN = true ;CONFN <> CONFN = false;
 197  endtype (* Signal *)
 198
(* Define control signals as of type Control_Signal. *)
 199  type Control_Signal is
 200    sorts  Control_Signal
 201    opns
 202     BUSY,DISC,DITO,FORW,OOFS,
 203     RANO,RING :-> Control_Signal
 204  endtype (* Control_Signal *)
```

```
 205
 206            (* Behavioural Expressions *)
 207  behaviour
 208
 209   hide line in
 210
(* Synchronization of the connections with the system network on
   gate line. Initially, all the sets are empty.
*)
 211    Multi_Connections [user,line]
 212    |[line]|
 213    System_Network [user,update,line](Empty,Empty,
                                          Empty,NoPair)
 214
(* The process Multi_Connections consists of multiple connections
   being processed in parallel but without synchronizing with each
   other.
*)
 215    where
 216     process Multi_Connections [user,line]:noexit:=
 217       One_Connection [user,line]
 218       |||
 219       i ;
 220       Multi_Connections [user,line]
 221
(* A single connection consists of a number of subscribers and
   a controller "Connection_Handler" of their scenarios.
   The latter controls internal events only.
*)
 222     where
 223       process One_Connection [user,line]:noexit:=
 224         Subscribers [user,line]
 225         |[line]|
 226         Connection_Handler [line]
 227
(* The process Subscribers consists of two basic subscribers that
   interleave with each other. Additional subscribers may be added
   to the connection from within the corresponding processes.
```

```
*)
 228        where
 229          process Subscribers [user,line]:noexit:=
 230            Origination_Side [user,line]
 231            |||
 232            Destination_Side [user,line]
 233
(* The origination side whose phone number is given by N starts
   a connection by going OffHook.
*)
 234          where
 235            process Origination_Side [user,line]
                        :noexit:=
 236              user ?N:Phone !OffHook ;
 237              (line !N !DITO ;
 238               user !N !Dito_T ;
 239               ((user !N !DialKs ?CN:Phone ;
 240                 line !CN !CORE ;
 241                 exit
 242                 [>
(* The originator aborts the call at the origination phase. *)
 243                   Hang_Up [user,line](N)
 244               )
 245               >>
 246
(* The communication path is being established. *)
 247               line !N !CONN ;
 248               User_Talk [user,line](N)
 249               []
(* The destination line is out of service. *)
 250               line ?RN:Phone !OOFS ;
 251               user !N !Oofs_T ;
 252               Hang_Up [user,line](N)
 253               []
(* The destination line is currently busy. *)
 254               line ?RN:Phone !BUSY ?Set:PhoneSet ;
```

```
 255                user !N !Busy_T ;
(* The originator goes OnHook after it gets a busy tone. *)
 256                (Hang_Up [user,line](N)
 257                 []
(* The originator invokes a ring again feature after it gets
   a busy tone.
*)
 258                 [N <> RN] ->
 259                  user !N !RiAgK ;
 260                  line !N !RIAG ;
 261                  Ring_Again_Feature [user,line]
                                        (N,RN)
 262              ))
 263              []
(* The originator line is currently busy. An extension of its
   telephone is being used.
*)
 264              line !N !BUSY !Empty ;
 265              user !N !Busy_T ;
 266              user !N !HangUp;
 267              stop
 268              []
(* The originator line is out of service. *)
 269              line !N !OOFS;
 270              user !N !Oofs_T ;
 271              user !N !HangUp;
 272              stop
 273             )
 274
(* Formal definition of processing the ring again feature. *)
 275           where
 276           process Ring_Again_Feature [user,line]
                       (N,RN:Phone):noexit:=
 277             user !N !HangUp ;
(* The originator cancels the ring again feature before it gets
   a ring again notification "RANO".
*)
```

```
 278             (user !N !RiAgK;
 279              line !N !DISC ;
 280              stop
 281              []
(* A ring again notification is being applied on the originator
   telephone.
*)
 282              line !N !RANO !RN ;
 283              user !N !Rano_T ;
(* The originator goes OffHook to call the same destination. *)
 284              (user !N !OffHook ;
 285               line !N !RIAG ;
(* The communication path is being established. *)
 286               (line !N !CONN ;
 287                User_Talk [user,line](N)
 288                []
(* The destination line is busy after it has been released. *)
 289                line !RN !BUSY ?Set:PhoneSet ;
 290                user !N !Busy_T ;
 291                Ring_Again_Feature[user,line](N,RN)
 292               )
 293               []
(* The originator cancels the ring again feature after it gets
   a ring again notification "RANO".
*)
 294               user !N !RiAgK;
 295               line !N !DISC ;
 296               stop
 297             ))
 298           endproc (* Ring_Again_Feature *)
 299           endproc (* Origination_Side *)
 300
(* The destination side is involved in a connection when a call
   terminates to its line. It starts with a connection attempt
   "COAT" sent to its line.
*)
 301           process Destination_Side [user,line]
```

```
                        :noexit:=
 302             (line ?CN:Phone !COAT ;
 303              exit(CN)
 304              [>
 305                stop
 306             )
 307             >>
 308             accept CN:Phone in
 309
(* Successful termination of a call to the destination "CN". *)
 310             Destination_Phone_Ring [user,line](CN)
 311             []
(* A forward feature is active on the destination line "CN". *)
 312             Call_Forwarded [user,line](CN)
 313
(* Formal definition of processing the forward feature. *)
 314            where
 315            process Call_Forwarded [user,line]
                        (CN:Phone):noexit:=
 316             line !CN !FORW ;
 317             line ?FN:Phone !FORW ;
 318             (line !FN !CORE ;
 319              line !FN !COAT ;
 320              exit(FN)
 321              [>
 322                stop
 323             )
 324             >>
 325             accept FN:Phone in
 326
(* Successful termination of the call to the next destination
   line "FN".
*)
 327             Destination_Phone_Ring [user,line](FN)
 328             []
(* A forward feature is active on a telephone at the next
   destination line "FN".
```

```
*)
 329             Call_Forwarded [user,line](FN)
 330            endproc (* Call_Forwarded *)
 331
 332            process Destination_Phone_Ring [user,
                        line](CN:Phone):noexit:=
 333             line !CN !RING ;
(* The destination telephone starts ringing. *)
 334             user !CN !Ring_T ;
(* The telephone keeps ringing until the destination side answers
   "OffHook" or the call is aborted. In the latter case, its line
   must be released "DISC".
*)
 335             (Ringing [line](CN)
 336              [>
 337               (user !CN !OffHook ;
 338                line !CN !CONN ;
 339                User_Talk [user,line](CN)
 340                []
 341                line !CN !DISC ;
 342                stop
 343             ) )
 344            endproc (* Destination_Phone_Ring *)
 345            endproc (* Destination_Side *)
 346
(* This process describes the conversation between subscribers in
   one connection. The latter is disabled whenever a feature is
   invoked "Feature", or a subscriber leaves it for good
   "User_Termination". The process User_Talk is associated with
   every subscriber. If a subscriber disables a conversation, the
   latter is still active with the other subscribers.
*)
 347            process User_Talk [user,line](N:Phone)
                        :noexit:=
 348             Conversation [user,line](N)
 349             [>
 350                Feature[user,line](N,HoldK,HOLD,
                                       HOLDN)
```

```
 351                []
 352                Feature [user,line](N,TransK,TRAN,
                                        TRANN)
 353                []
 354                Feature [user,line](N,ConfK,CONF,
                                        CONFN)
 355                []
 356                User_Termination [user,line](N)
 357
 358            endproc (* User_Talk *)
 359
(* This process is associated with every subscriber participating
   in a conference call conversation. Any one can either invoke a
   conference feature "Feature" or leave for good the conversation
   "Hang_Up".
*)
 360            process More_Conference [user,line]
                        (N:Phone):noexit:=
 361             Conversation [user,line](N)
 362             [>
 363               Feature [user,line](N,ConfK,CONF,
                                       CONFN)
 364               []
 365               Hang_Up [user,line](N)
 366            endproc (* More_Conference *)
 367
 368          process Feature[user,line](N:Phone,K:
                       Key,Sign,Note:Signal):noexit:=
(* A subscriber behaves as the originator of a feature. *)
 369            user !N !K ;
 370            (Invoke_Feature [user,line](N,K,
                                             Sign)
 371             |||
 372             Additional_Subscriber[user,line](K)
 373            )
 374            []
(* A subscriber behaves as the party waiting for the invoker of a
```

```
   feature to be re-connected.
*)
 375            Party_On_Hold [user,line](N,Note)
 376           endproc (* Feature *)
 377
 378           process Invoke_Feature [user,line]
                       (N:Phone,K:Key,Sign:Signal)
                       :noexit:=
 379            line !N !Sign ;
 380            (user !N !DialKs ?TN:Phone ;
 381             line !TN !CORE ;
 382             exit
 383             [>
(* The invoker of a feature aborts the new call before its
   termination to the new line (Additional Subscriber's Side).
*)
 384               User_Recon [user,line](N,K)
 385            )
 386            >>
(* Termination of the new call to a busy line "TN". *)
 387            line ?TN:Phone !BUSY ?Set:PhoneSet ;
 388            user !N !Busy_T ;
 389            User_Recon [user,line](N,K)
 390            []
(* Termination of the new call to an out of service line. *)
 391            line ?TN:Phone !OOFS ;
 392            user !N !Oofs_T ;
 393            User_Recon [user,line](N,K)
 394            []
(* Successful termination of the new call to its destination. *)
 395            line !N !CONN ;
 396            (Conversation [user,line](N)
 397             [>
(* While in conversation with the new party, the invoker of the
   feature gets a disconnection indication "DSIN".
*)
 398               line !N !DSIN ;
```

```
(* If the feature was a transfer, this side is re-connected to
   the original call.
*)
 399               ([K == TransK] ->
 400                 user !N !TransK ;
 401                 line !N !RECO ;
 402                 User_Talk [user,line](N)
 403                []
(* If the feature was not a transfer, check process User_Recon. *)
 404                [K <> TransK] ->
 405                  User_Recon [user,line](N,K)
 406               )
 407               []
 408               [K == TransK] ->
(* While in conversation with the new party, the invoker of the
   transfer feature establishes a three-way conversation.
*)
 409                (user !N !TransK ;
 410                 line !N !RECO ;
 411                 (Conversation [user,line](N)
 412                  [>
 413                    Hang_Up [user,line](N)
 414                    []
 415                    line !N !DSIN ;
 416                    User_Talk [user,line](N)
 417                 )
 418                 []
(* The invoker of the transfer feature transfers the original
   call and then leaves it for good.
*)
 419                 user !N !ReleaseK ;
 420                 Hang_Up [user,line](N)
 421                )
 422               []
(* While in conversation with the new party, the invoker of the
   feature (other than the transfer feature) terminates it.
*)
```

```
 423               [K <> TransK] ->
 424                User_Recon [user,line](N,K)
 425            )
 426          endproc (* Invoke_Feature *)
 427
(* This process describes the behaviour of the subscriber involved
   in a new call.
*)
 428          process Additional_Subscriber[user,line]
                        (K:Key):noexit:=
 429            (line ?TN:Phone !COAT ;
 430             exit(TN)
 431             [>
 432               stop
 433            )
 434            >>
 435            accept TN:Phone in
 436
(* Successful termination of the new call to its line "TN". *)
 437            New_Phone_Ring [user,line](K,TN)
 438            []
(* A forward feature is currently active on this telephone. *)
 439            New_Subscriber_Forward [user,line]
                                      (K,TN)
 440
 441            where
 442            process New_Subscriber_Forward [user,
                       line](K:Key,TN:Phone):noexit:=
 443             line !TN !FORW ;
 444             (line ?FN:Phone !FORW ;
 445              line !FN !CORE ;
 446              line !FN !COAT ;
 447              exit(FN)
 448              [>
 449                stop
 450             )
```

```
 451             >>
 452             accept FN:Phone in
 453
(* Successful termination of the new call to next line "FN". *)
 454             New_Phone_Ring [user,line](K,FN)
 455             []
(* A forward feature is currently active on the next destination's
   telephone "FN".
*)
 456             New_Subscriber_Forward [user,line]
                                        (K,FN)
 457           endproc (* New_Subscriber_Forward *)
 458
(* Processing the successful termination of a new call. *)
 459           process New_Phone_Ring [user,line]
                        (K:Key,TN:Phone):noexit:=
 460            line !TN !RING ;
 461            user !TN !Ring_T ;
(* The telephone of the new party is ringing. *)
 462             (Ringing [line](TN)
 463              [>
(* The new party answers the new call. *)
 464                user !TN !OffHook ;
 465                line !TN !CONN ;
 466                (Conversation [user,line](TN)
 467                 [>
(* The new party leaves for good the call. *)
 468                   Hang_Up [user,line](TN)
 469                   []
(* The new party is added to a conference conversation (if it is
   a conference feature).
*)
 470                   [K == ConfK] ->
 471                    line !TN !RECO ;
 472                    More_Conference [user,line](TN)
 473                   []
 474                   [K <> ConfK] ->
```

```
(* The feature was not a conference, and the new party gets a
   disconnection indication.
*)
 475                    line !TN !DSIN ;
 476                    ([K == TransK] ->
(* If it is a transfer feature, the new party is automatically
   connected in a normal call with the party on hold.
*)
 477                     User_Talk [user,line](TN)
 478                    []
 479                    [K == HoldK] ->
(* If it was a hold feature, the new party must hang up its
   telephone.
*)
 480                     Hang_Up [user,line](TN)
 481                )    )
 482                []
(* The new call is aborted while it is ringing. Therefore,
   disconnection of this line is required.
*)
 483                line !TN !DISC ;
 484                stop
 485             )
 486            endproc (* New_Phone_Ring *)
 487          endproc (* Additional_Subscriber *)
 488
(* This process describes the behaviour of the party that gets a
   feature notification and it is put on hold.
*)
 489          process Party_On_Hold [user,line]
                        (N:Phone,Notify:Signal):noexit:=
(* This party gets a feature notification. *)
 490            line !N !Notify ;
 491            (line !N !RECO ;
 492             ([Notify == TRANN] ->
(* This side is re-connected to a three-way conversation. *)
 493               (Conversation [user,line](N)
 494                [>
```

```
(* While in a three-way conversation, this side may get a
   disconnection indication and be re-connected with the remaining
   subscriber in a normal call. It may also leave the conversation
   at any time.
*)
 495                  line !N !DSIN ;
 496                  User_Talk [user,line](N)
 497                  []
 498                  Hang_Up [user,line](N)
 499               )
 500              []
 501              [Notify == CONFN] ->
(* This side is re-connected to a conference call conversation. *)
 502               More_Conference [user,line](N)
 503              []
 504              [Notify == HOLDN] ->
(* This side is re-connected again to the original call. *)
 505               User_Talk [user,line](N)
 506             )
 507             []
(* It then gets a disconnection indication from the invoker of the
   feature. This side is re-connected with the new party.
*)
 508             line !N !DSIN ;
 509             line !N !RECO ;
 510             User_Talk [user,line](N)
 511             []
(* While on hold, this party leaves for good the call. *)
 512             Hang_Up [user,line](N)
 513           )
 514          endproc (* Party_On_Hold *)
 515
(* This process handles some cases of a re-connection. *)
 516          Process User_Recon [user,line]
                         (N:Phone,K:Key):noexit:=
 517           user !N !K ;
 518           line !N !RECO ;
```

```
 519            ([K <> ConfK] ->
 520              User_Talk [user,line](N)
 521             []
 522             [K == ConfK] ->
 523              More_Conference [user,line](N)
 524            )
 525          endproc (* User_Recon *)
 526
 527          process Conversation [user,line]
                        (N:Phone):noexit:=
 528           user !N !SendVo ;
 529           line !N !VOICE ;
 530           Conversation [user,line](N)
 531          endproc (* Conversation *)
 532
(* A subscriber may be a terminator of a call or the receiver of
   a disconnection indication from the other side.
*)
 533          process User_Termination [user,line]
                        (N:Phone):noexit:=
 534           Hang_Up [user,line](N)
 535           []
 536           line  !N  !DSIN ;
 537           Hang_Up [user,line](N)
 538          endproc (* User_Termination *)
 539
(* This process handles termination of a call by a subscriber. *)
 540          process Hang_Up [user,line](N:Phone)
                                :noexit:=
 541           user !N !HangUp ;
 542           line !N !DISC ;
 543           stop
 544          endproc (* Hang_Up *)
 545         endproc(* Subscribers *)
 546
(* The process Connection_Handler describes the behavior of the
   controller of scenarios within a single connection. Only hidden
```

```
   events "gate line" are of concern.
*)
 547          process Connection_Handler[line]:noexit:=
(* It forces a dial tone "DITO", a busy signal "BUSY", or an
   out of service "OOFS" signal to be the first internal event in
   a connection. These signals are applied on the telephone of
   the originator.
*)
 548           line ?N:Phone !DITO ;
 549           Control_Termination [line](N,Empty)
 550           []
 551           line ?N:Phone !BUSY !Empty ;
 552           stop
 553           []
 554           line ?N:Phone !OOFS ;
 555           stop
 556
(* Controls the termination of a call to its destination. *)
 557           where
 558           process Control_Termination [line]
                          (N:Phone,Set:PhoneSet):noexit:=
 559            (line ?TN:Phone !CORE ;
 560             line !TN !COAT ;
 561             exit(TN)
 562             [>
 563               line !N !DISC ;
 564               stop
 565            )
 566            >>
 567            accept TN:Phone in
 568
(* Check if the current destination "TN" didn't forward the same
   call (this is to detect a cycle in the forward processing).
*)
 569            [NotIn(TN,Set)] ->
 570             line !TN !RING ;
 571             (Ringing [line](TN)
```

```
 572              [>
(* The current call is being established. *)
 573               (line !TN !CONN ;
 574                line !N !CONN ;
 575                Control_Conversation [line](N,TN)
 576                []
(* The current call is aborted. Disconnection of both subscribers
   is required.
*)
 577                line !N !DISC ;
 578                line !TN !DISC ;
 579                stop
 580             ) )
 581            []
(* If a forward feature is currently active on this telephone,
   the call is forwarded to the pre-selected line "FN".
*)
 582            [NotIn(TN,Set)] ->
 583             line !TN !FORW ;
 584             line ?FN:Phone !FORW ;
(* Control termination of this call to the next destination. *)
 585             Control_Termination [line]
                                    (N,Insert(TN,Set))
 586            []
 587            Unsuccessful_Termination [line]
                                        (N,TN,Set)
 588          endproc (* Control_Termination *)
 589
 590          process Unsuccessful_Termination [line]
                    (N,TN:Phone,Set:PhoneSet):noexit:=
(* The current destination is busy. Either a cycle in the forward
   processing is detected or the line is currently being used.
*)
 591            line !TN !BUSY !Set ;
 592            (line !N !DISC ;
 593             stop
 594             []
```

```
(* The originator invokes a ring again feature on a busy tone. *)
 595             line !N !RIAG;
 596             Control_RingAgain [line](N,TN,Set)
 597            )
 598            []
(* The current destination line is out of service. *)
 599            line !TN !OOFS ;
 600            line !N !DISC ;
 601            stop
 602
 603          where
 604          process Control_RingAgain [line]
                      (N,TN:Phone,Set:PhoneSet):noexit:=
(* The originator cancels the ring again feature before it gets
   a ring again notification.
*)
 605            (line !N !DISC ;
 606             stop
 607             []
(* The originator gets a ring again notification. *)
 608             [NotIn(TN,Set)] ->
 609             line !N !RANO !TN ;
 610             (line !N !RIAG ;
 611              ([NotIn(TN,Set)] ->
(* Successful termination of the current call. *)
 612               line !TN !RING ;
 613               (Ringing [line](TN)
 614                [>
(* The current call is being established. *)
 615                (line !TN !CONN ;
 616                 line !N !CONN ;
 617                 Control_Conversation [line](N,TN)
 618                 []
(* The originator aborts the current call while it is ringing
   at the destination side.
*)
 619                 line !N !DISC ;
```

```
 620                  line !TN !DISC ;
 621                  stop
 622               ) )
 623               []
(* The current destination is busy after it has been released.
   The originator may then wait for further notification or cancel
   the feature.
*)
 624               line !TN !BUSY !Set ;
 625               Control_RingAgain [line](N,TN,Set)
 626              )
 627              []
(* The originator cancels the ring again feature after it gets
   a notification (when the line of other side becomes free).
*)
 628              line !N !DISC ;
 629              stop
 630           ))
 631          endproc (* Control_RingAgain *)
 632          endproc (* Unsuccessful_Termination *)
 633
(* This process controls the conversation between two subscribers
   "N" and "RN".
*)
 634          process Control_Conversation [line]
                         (N,RN:Phone):noexit:=
 635           Talking [line](N)
 636           |||
 637           Talking [line](RN)
 638           [>
(* While in conversation, any subscriber may invoke a feature. *)
 639             Control_Feature [line](N,RN)
 640             []
 641             Control_Feature [line](RN,N)
 642
(* This process controls the behaviour of a conversation after a
   feature has been invoked.
*)
```

```
 643          where
 644          process Control_Feature [line]
                      (N,RN:Phone):noexit:=
 645           Control_Hold [line](N,RN)
 646           []
 647           Control_Transfer [line](N,RN)
 648           []
 649           Control_Conference [line](N,RN)
 650           []
 651           Control_Disconnection [line](N,RN)
 652
 653          where
 654
(* This process controls a conversation if the invoked feature is
   a hold.
*)
 655          process Control_Hold [line](N,RN:Phone)
                                          :noexit:=
(* Controls the origination phase of the new call. *)
 656           Control_Feature_Origination [line]
                                (N,RN,HOLD,HOLDN)
 657           >>
 658           accept TN:Phone  in
 659
(* Controls the termination phase of the new call. *)
 660           Control_Hold_Ring [line](N,RN,TN,Empty)
 661           []
(* A forward feature is active on the destination side's line. *)
 662           Control_Hold_Forward [line]
                                (N,RN,TN,Empty)
 663
(* Controls the behaviour of the new call if it terminates to a
   line where a forward feature is in effect.
*)
 664          where
 665          process Control_Hold_Forward [line]
                 (N,RN,TN:Phone,Set:PhoneSet):noexit:=
```

```
(* Controls the origination phase of a forward feature. *)
 666            Control_Forward_Origination [line]
                                    (N,RN,TN,Set)
 667            >>
 668            accept FN:Phone,NewSet:PhoneSet in
 669
(* Controls the termination phase of the new call to the next
   destination.
*)
 670            Control_Hold_Ring[line](N,RN,FN,NewSet)
 671            []
(* A forward feature is active on the next destination line. *)
 672            Control_Hold_Forward [line]
                                (N,RN,FN,NewSet)
 673          endproc (* Control_Hold_Forward *)
 674
(* This process controls the behaviour of a conversation when the
   new call terminates to its destination line. This is the case
   of a hold feature.
*)
 675          process Control_Hold_Ring [line]
                 (N,RN,TN:Phone,Set:PhoneSet):noexit:=
 676            [NotIn(TN,Set)] ->
(* Successful termination of the new call to line "TN". *)
 677            line !TN !RING ;
 678            (Ringing [line](TN)
 679             [>
(* A communication path for the new call is being established. *)
 680              line !TN !CONN ;
 681              line !N !CONN ;
 682              ((Talking [line](N)
 683                |||
 684               Talking [line](TN)
 685              )
 686               [>
(* The originator of the feature terminates the new call. *)
 687                (line !N !RECO ;
```

```
 688                line !TN !DSIN ;
 689                line !TN !DISC ;
 690                (line !RN !RECO ;
 691                 Control_Conversation[line](N,RN)
 692                 []
 693                 Control_Disconnection[line](RN,N)
 694                )
 695                []
(* The new party terminates the new call. *)
 696                line !TN !DISC ;
 697                line !N !DSIN ;
 698                Reconnection [line](N,RN)
 699             ) )
 700             []
(* The originator of the feature cancels it. Disconnection of the
   new party's line "TN" is required.
*)
 701             line !N !RECO ;
 702             line !TN !DISC ;
 703             (line !RN !RECO ;
 704              Control_Conversation [line](N,RN)
 705              []
 706              Control_Disconnection [line](RN,N)
 707           ) )
 708           []
(* Unsuccessful termination of the new call. *)
 709            Unsuccessful_Feature_Termination [line]
                                        (N,RN,TN,Set)
 710          endproc (* Control_Hold_Ring *)
 711          endproc (* Control_Hold *)
 712
(* This process describes the behaviour of a conversation after
   a transfer / three-way calling feature is invoked.
*)
 713          process Control_Transfer [line]
                                 (N,RN:Phone):noexit:=
(* Controls the origination phase of a feature. *)
```

```
 714            Control_Feature_Origination [line]
                                  (N,RN,TRAN,TRANN)
 715            >>
 716            accept TN:Phone  in
 717
(* Controls the termination phase of the new call to the next
   destination.
*)
 718            Control_Transfer_Ring [line]
                                   (N,RN,TN,Empty)
 719            []
(* A forward feature is active on the destination side's line. *)
 720            Control_Transfer_Forward [line]
                                   (N,RN,TN,Empty)
 721
(* Controls the behaviour of the new call if it terminates to a
   line where a forward feature is in effect.
*)
 722           where
 723           process Control_Transfer_Forward [line]
                  (N,RN,TN:Phone,Set:PhoneSet):noexit:=
(* Controls the origination phase of a forward feature. *)
 724            Control_Forward_Origination [line]
                                  (N,RN,TN,Set)
 725            >>
 726            accept FN:Phone,NewSet:PhoneSet in
 727
(* Controls the termination phase of the new call to the next
   destination.
*)
 728            Control_Transfer_Ring [line]
                                   (N,RN,FN,NewSet)
 729            []
(* A forward feature is active on the next destination line. *)
 730            Control_Transfer_Forward [line]
                                   (N,RN,FN,NewSet)
 731           endproc (* Control_Transfer_Forward *)
```

```
 732
(* This process controls the behaviour of a conversation when the
   new call terminates to its destination line. This is the case
   of a transfer / three-way calling feature.
*)
 733          process Control_Transfer_Ring [line]
                  (N,RN,TN:Phone,Set:PhoneSet):noexit:=
 734            [NotIn(TN,Set)] ->
(* Successful termination of the new call to line "TN". *)
 735            line !TN !RING ;
 736            (Ringing [line](TN)
 737             [>
(* A communication path for the new call is being established. *)
 738              line !TN !CONN ;
 739              line !N !CONN ;
 740              (Talking [line](N)
 741               |||
 742               Talking [line](TN)
 743               [>
(* The originator establishes a three-way conversation. *)
 744                line !N !RECO ;
 745                (line !RN !RECO ;
 746                 (Talking [line](N)
 747                  |||
 748                  Talking [line](RN)
 749                  |||
 750                  Talking [line](TN)
 751                  [>
(* One of the three subscribers leaves for good the three-way
   conversation.
*)
 752                   (Special_Control [line](RN,TN,N)
 753                    []
 754                    Special_Control [line](N,TN,RN)
 755                    []
 756                    Special_Control [line](N,RN,TN)
 757                ) )
```

```
 758                  []
(* The party on hold leaves for good the original call. *)
 759                  line !RN !DISC ;
 760                  Control_Conversation [line](N,TN)
 761                 )
 762                 []
(* The new party leaves for good the new call. *)
 763                 line !TN !DISC ;
 764                 line !N !DSIN ;
 765                 Reconnection [line](N,RN)
 766                 []
(* The originator of a transfer feature transfers the current
   call then leaves it for good.
*)
 767                 line !N !DISC ;
 768                 line !TN !DSIN ;
 769                 (line !RN !DSIN ;
 770                  line !RN !RECO ;
 771                  Control_Conversation [line](RN,TN)
 772                  []
 773                  Control_Disconnection[line](RN,TN)
 774               ) )
 775               []
(* The originator of the feature cancels it. Disconnection of the
   new party's line "TN" is required.
*)
 776               line !N !RECO ;
 777               line !TN !DISC ;
 778               (line !RN !RECO ;
 779                Control_Conversation [line](N,RN)
 780                []
 781                Control_Disconnection [line](RN,N)
 782              ))
 783              []
(* Unsuccessful termination of the new call. *)
 784              Unsuccessful_Feature_Termination[line]
                                          (N,RN,TN,Set)
```

```
 785            endproc (* Control_Transfer_Ring *)
 786            endproc (* Control_Transfer *)
 787
(* This process controls the case when a subscriber in a
   three-way conversation leaves it for good.
*)
 788            process Special_Control [line]
                                (N1,N2,DN:Phone):noexit:=
 789             line !DN !DISC ;
 790             line !N1 !DSIN ;
 791             line !N2 !DSIN ;
 792             Control_Conversation [line](N1,N2)
 793            endproc (* Special_Control *)
 794
(* This process describes the behaviour of a conversation after
   a conference call feature is invoked.
*)
 795            process Control_Conference [line]
                                  (N,RN:Phone):noexit:=
(* Controls the origination phase of a feature. *)
 796             Control_Feature_Origination [line]
                                   (N,RN,CONF,CONFN)
 797             >>
 798             accept TN:Phone in
 799
(* Controls the termination phase of the new call to the next
   destination.
*)
 800             Control_Conference_Ring [line]
                                       (N,RN,TN,Empty)
 801             []
(* A forward feature is active on the destination side's line. *)
 802             Control_Conference_Forward [line]
                                       (N,RN,TN,Empty)
 803
(* Controls the behaviour of the new call if it terminates to a
   line where a forward feature is in effect.
```

```
*)
 804            where
 805            process Control_Conference_Forward
                        [line](N,RN,TN:Phone,
                        Set:PhoneSet):noexit:=
(* Controls the origination phase of a forward feature. *)
 806             Control_Forward_Origination [line]
                                    (N,RN,TN,Set)
 807             >>
 808             accept FN:Phone,NewSet:PhoneSet in
 809
(* Controls the termination phase of the new call to the next
   destination.
*)
 810             Control_Conference_Ring [line]
                                        (N,RN,FN,NewSet)
 811             []
(* A forward feature is active on the next destination line. *)
 812             Control_Conference_Forward [line]
                                        (N,RN,FN,NewSet)
 813            endproc (* Control_Conference_Forward *)
 814
(* This process controls the behaviour of a conversation when the
   new call terminates to its destination line. This is the case
   of a conference call feature.
*)
 815            process Control_Conference_Ring [line]
                 (N,RN,TN:Phone,Set:PhoneSet):noexit :=
 816             [NotIn(TN,Set)] ->
(* Successful termination of the new call to line "TN". *)
 817             line !TN !RING ;
 818             (Ringing [line](TN)
 819              [>
(* A communication path for the new call is being established. *)
 820                line !TN !CONN ;
 821                line !N !CONN ;
 822                (Talking [line](N)
```

```
 823                 |||
 824                 Talking [line](TN)
 825                 [>
(* The originator establishes a conference call conversation. *)
 826                  line !N !RECO ;
 827                  (line !RN !RECO ;
 828                   line !TN !RECO ;
 829                   (On_Conference [line](N)
 830                    |||
 831                    On_Conference [line](RN)
 832                    |||
 833                    On_Conference [line](TN)
 834                   )
 835                   []
(* The party on hold leaves for good the original call. *)
 836                   line !RN !DISC ;
 837                   Control_Conversation[line](N,TN)
 838                  )
 839                  []
(* The new party leaves for good the new call. *)
 840                  line !TN !DISC ;
 841                  line !N !DSIN ;
 842                  Reconnection [line](N,RN)
 843                )
 844                []
(* The originator of the feature cancels it. Disconnection of the
   new party's line "TN" is required.
*)
 845                line !N !RECO ;
 846                line !TN !DISC ;
 847                (line !RN !RECO ;
 848                 Control_Conversation [line](N,RN)
 849                 []
 850                 Control_Disconnection [line](RN,N)
 851             )  )
 852             []
```

```
(* Unsuccessful termination of the new call. *)
 853             Unsuccessful_Feature_Termination[line]
                                        (N,RN,TN,Set)
 854            endproc (* Control_Conference_Ring *)
 855            endproc (* Control_Conference *)
 856
(* This process controls the origination phase of processing a
   feature among hold, transfer/three-way calling, and conference
   call features.
*)
 857            process Control_Feature_Origination
                        [line](N,RN:Phone,Sign,
                        Note:Signal):exit(Phone):=
 858             line !N !Sign ;
 859             line !RN !Note ;
 860             (line ?TN:Phone !CORE ;
 861              line !TN !COAT ;
 862              exit(TN)
 863               [>
(* The invoker of a feature cancels it before the termination
   of the new call to its destination.
*)
 864                Reconnection [line](N,RN)
 865             )
 866            endproc (*Control_Feature_Origination*)
 867
(* This process controls the processing of forwarding a call
   from a line to a pre-selected one.
*)
 868            process Control_Forward_Origination
                       [line](N,RN,TN:Phone,Set:
                       PhoneSet):exit(Phone,PhoneSet):=
 869             [NotIn(TN,Set)] ->
 870              line !TN !FORW ;
 871              (line ?FN:Phone !FORW ;
 872               line !FN !CORE ;
 873               line !FN !COAT ;
```

```
 874                exit (FN,Insert(TN,Set))
 875                [>
(* While a new call is being forwarded, its originator aborts it.
   This is done by canceling the invoked feature.
*)
 876                   Reconnection [line](N,RN)
 877              )
 878            endproc (*Control_Forward_Origination*)
 879
(* This process controls the situation when a subscriber is being
   re-connected to the original call.
*)
 880            process Reconnection [line](N,RN:Phone)
                                          :noexit:=
 881              line !N !RECO ;
 882              (line !RN !RECO ;
 883               Control_Conversation [line](N,RN)
 884               []
 885               Control_Disconnection [line](RN,N)
 886              )
 887            endproc (* Reconnection *)
 888
(* This process controls an unsuccessful termination of a new
   call.
*)
 889           process Unsuccessful_Feature_Termination
                     [line](N,RN,TN:Phone,Set:
                     PhoneSet):noexit:=
(* The destination line is currently being used. *)
 890              line !TN !BUSY !Set ;
 891              Reconnection [line](N,RN)
 892              []
(* The destination line is currently out of service. *)
 893              line !TN !OOFS ;
 894              Reconnection [line](N,RN)
 895            endproc
 896            endproc (* Control_Feature *)
```

```
 897            endproc (* Control_Conversation *)
 898
(* This process controls the behaviour of every subscriber
   participating in a conference call conversation.
*)
 899            process On_Conference [line](N:Phone)
                                              :noexit:=
 900             Talking [line](N)
 901             [>
(* One of the subscribers leaves for good the conference call
   conversation.
*)
 902               line !N !DISC ;
 903               stop
 904               []
(* One of the subscribers invokes a conference call feature to add
   a new subscriber to the conference conversation.
*)
 905               Add_Conf [line](N)
 906            endproc (* On_Conference *)
 907
(* This process controls the situation where a subscriber
   participating in a conference call conversation invokes a
   conference call feature to consult privately with a new
   subscriber or to add to the conversation.
*)
 908            process Add_Conf [line](N:Phone)
                                        :noexit:=
 909             line !N !CONF ;
 910             (line ?TN:Phone !CORE ;
 911              line !TN !COAT ;
 912              exit(TN)
 913              [>
(* The invoker of the conference call feature cancels it before
   the termination of the new call to its destination.
*)
 914               line !N !RECO ;
 915               On_Conference [line](N)
```

```
 916             )
 917             >>
 918             accept TN:Phone in
 919
(* Termination of the new call to its destination. *)
 920             Control_MoreConference_Ring [line]
                                             (N,TN,Empty)
 921             []
(* The new call is to be forwarded to a second destination. *)
 922             Control_MoreConference_Forward [line]
                                             (N,TN,Empty)
 923
(* This process controls the forwarding process of new call. *)
 924           where
 925           process Control_MoreConference_Forward
                       [line](N,TN:Phone,Set:
                       PhoneSet):noexit:=
 926            [NotIn(TN,Set)] ->
 927            line !TN !FORW ;
 928            (line ?FN:Phone !FORW ;
 929             line !FN !CORE ;
 930             line !FN !COAT ;
 931             exit (FN,Insert(TN,Set))
 932              [>
(* The invoker of the conference call feature cancels it before
   the termination of the new call to its next destination.
*)
 933                line !N !RECO ;
 934                On_Conference [line](N)
 935            )
 936            >>
 937            accept FN:Phone,NewSet:PhoneSet in
 938
(* Termination of the new call to its next destination. *)
 939            Control_MoreConference_Ring [line]
                                             (N,FN,NewSet)
```

```
 940             []
(* The new call is to be forwarded to another destination. *)
 941             Control_MoreConference_Forward [line]
                                              (N,FN,NewSet)
 942           endproc
 943
(* This process controls the termination of the new call to its
 destination.
*)
 944           process Control_MoreConference_Ring
                       [line](N,TN:Phone,Set:
                       PhoneSet):noexit:=
 945             [NotIn(TN,Set)] ->
(* Successful termination of the new call to line "TN". *)
 946             line !TN !RING ;
 947             (Ringing [line](TN)
 948              [>
(* A communication path for the new call is being established. *)
 949                line !TN !CONN ;
 950                line !N !CONN ;
 951                (Talking [line](N)
 952                 |||
 953                 Talking [line](TN)
 954                 [>
(* The originator of the conference feature allows the new party
   to participate in the conference conversation.
*)
 955                  line !N !RECO ;
 956                  line !TN !RECO ;
 957                  (On_Conference [line](N)
 958                   |||
 959                   On_Conference [line](TN)
 960                  )
 961                  []
(* The new party leaves for good the new call. *)
 962                  line !TN !DISC ;
 963                  line !N !DSIN ;
```

```
 964                    line !N !RECO ;
 965                    On_Conference [line](N)
 966                )
 967                []
(* The originator of the feature cancels it. Disconnection of the
   new party's line "TN" is required.
*)
 968                line !N !RECO ;
 969                line !TN !DISC ;
 970                On_Conference [line](N)
 971             )
 972             []
(* The end destination "TN" is currently busy. *)
 973             line !TN !BUSY !Set ;
 974             line !N !RECO ;
 975             On_Conference [line](N)
 976             []
(* The end destination "TN" is currently out of service. *)
 977             line !TN !OOFS ;
 978             line !N !RECO ;
 979             On_Conference [line](N)
 980           endproc (*Control_MoreConference_Ring*)
 981           endproc (* Add_Conf *)
 982
(* This process controls the conversation at every side. *)
 983           process Talking [line](N:Phone)
                                :noexit:=
 984             line !N !VOICE ;
 985             Talking [line](N)
 986           endproc (* Talking *)
 987
(* This process controls the disconnection of a subscriber. *)
 988           process Control_Disconnection [line]
                                (N,RN:Phone):noexit:=
 989             line !N  !DISC ;
 990             line !RN !DSIN ;
```

```
 991            line !RN !DISC ;
 992            stop
 993          endproc (* Control_Disconnection *)
 994        endproc (* Connection_Handler *)
 995
(* This process controls the ringing signal on a telephone. *)
 996        process Ringing [line](N:Phone):noexit:=
 997         line !N !ONRI ;
 998         Ringing [line](N)
 999        endproc (* Ringing *)
1000      endproc (* One_Connection *)
1001    endproc (* Multi_Connections*)
1002
(* The process System_Network interacts with both the
   telephones (gate line) and the environment (gates user and
   update). Only the choice operator is used in the definition of
   this process. A predicate is associated with each action to
   control the synchronization of this process with the telephones
   on signals. With the environment, a predicate controls the
   updating of the system and also, it controls the use of a
   forward feature on a telephone.
*)
1003    process System_Network [user,update,line]
             (ServiceSet,BusySet,ForwardSet:PhoneSet,
              ForwardPairs:ForwardPairSet):noexit:=
(* Interaction with the environment to update the system. *)
1004      update ?N:Phone ;
1005      ([NotIn(N,ServiceSet)] ->
(* The line "N" is to be added to the telephone system. *)
1006        System_Network [user,update,line]
                  (Insert(N,ServiceSet),BusySet,
                   ForwardSet,ForwardPairs)
1007       []
1008       [IsIn(N,ServiceSet) and NotIn(N,BusySet)]->
(* The line "N" is to be removed from the telephone system. *)
1009        System_Network [user,update,line]
                  (Remove(N,ServiceSet),BusySet,
```

```
                    ForwardSet,ForwardPairs)
1010        []
1011        [IsIn(N,BusySet)] ->
(* The line "N" is currently busy, and therefore it cannot be
   removed from the telephone system.
*)
1012         System_Network [user,update,line]
                    (ServiceSet,BusySet,ForwardSet,
                     ForwardPairs)
1013       )
1014       []
(* A subscriber invokes a forward feature on its telephone. *)
1015       Forward_Feature [user,update,line]
                   (ServiceSet,BusySet,ForwardSet,
                    ServiceSet,ForwardPairs)
1016       []
(* The signal being exchanged is not constrained to any condition,
    and therefore no predicate is needed for this action.
*)
1017       line ?N:Phone ?sign:Signal ;
1018       System_Network [user,update,line]
                   (ServiceSet,BusySet,ForwardSet,
                    ForwardPairs)
1019       []
(* Synchronization on DITO signal. The line "N" must be free
 and in service.
*)
1020       line ?N:Phone !DITO
1021       [NotIn(N,BusySet) and IsIn(N,ServiceSet)] ;
1022       System_Network[user,update,line]
                   (ServiceSet,Insert(N,BusySet),
                    ForwardSet,ForwardPairs)
1023       []
(* Synchronization on RING signal. The line "N" must be free,
   in service, and no forward feature is currently active on an
   associated telephone.
*)
1024       line ?N:Phone !RING
```

```
1025      [NotIn(N,BusySet) and IsIn(N,ServiceSet) and
                                NotIn(N,ForwardSet)] ;
1026      System_Network[user,update,line]
                (ServiceSet,Insert(N,BusySet),
                 ForwardSet,ForwardPairs)
1027      []
(* Synchronization on BUSY signal. The line "N" must be busy
   or a forward feature is currently active on an associated
   telephone.
*)
1028      line ?N:Phone !BUSY ?Set:PhoneSet
1029      [ IsIn(N,BusySet) or IsIn(N,Set) ] ;
1030      System_Network[user,update,line](ServiceSet,
                BusySet,ForwardSet,ForwardPairs)
1031      []
(* Synchronization on OOFS signal. The line "N" must be out
   of service.
*)
1032      line ?N:Phone !OOFS
1033      [ NotIn(N,ServiceSet) ] ;
1034      System_Network[user,update,line](ServiceSet,
                BusySet,ForwardSet,ForwardPairs)
1035      []
(* Synchronization on DISC signal. The line "N" must be busy.
*)
1036      line ?N:Phone !DISC
1037      [ IsIn(N,BusySet) ] ;
1038      System_Network[user,update,line]
                (ServiceSet,Remove(N,BusySet),
                 ForwardSet,ForwardPairs)
1039      []
(* Synchronization on RANO signal. The line "N" must be free.
*)
1040      line ?N:Phone !RANO ?TN:Phone
1041      [ NotIn(TN,BusySet) ] ;
1042      System_Network[user,update,line](ServiceSet,
                BusySet,ForwardSet,ForwardPairs)
```

```
1043       []
(* Synchronization on FORW signal. The line "N" must be free,
   in service, and a forward feature is currently active on an
   associated telephone. The operation ForwardTo returns the
   line to whom this call is to be forwarded.
*)
1044       line ?N:Phone !FORW
1045       [IsIn(N,ForwardSet) and IsIn(N,ServiceSet)
                               and NotIn(N,BusySet) ] ;
1046       line !ForwardTo(N,ForwardPairs) !FORW ;
1047       System_Network[user,update,line](ServiceSet,
                BusySet,ForwardSet,ForwardPairs)
1048
(* This process loops on the set ServiceSet1, to allow all the
   subscribers using the telephones on the lines currently
   connected to the telephone system to invoke a forward feature.
*)
1049     where
1050     process Forward_Feature [user,update,line]
                 (ServiceSet1,BusySet,ServiceSet,
                 ForwardSet:PhoneSet,ForwardPairs:
                 ForwardPairSet):noexit:=
1051       [not(Is_Empty(ServiceSet1))] ->
(* Instantiates the process Forward_Feature2 with the first line
   in the set ServiceSet1.
*)
1052       (Forward_Feature2[user,update,line]
                (Head(ServiceSet1),BusySet,ServiceSet,
                 ForwardSet,ForwardPairs)
1053        []
(* Loops on the remaining lines in the set ServiceSet1. *)
1054        Forward_Feature [user,update,line]
                (Tail(ServiceSet1),BusySet,ServiceSet,
                 ForwardSet,ForwardPairs)
1055       )
1056
(* This process checks the status of the line "N", then it
   either allows the subscriber on an associated telephone to
```

```
   invoke or to cancel a forward feature.
*)
1057     where
1058     process Forward_Feature2 [user,update,line]
                  (N:Phone,BusySet,ServiceSet,
                  ForwardSet:PhoneSet,ForwardPairs:
                  ForwardPairSet):noexit:=
1059       [NotIn(N,ForwardSet) and NotIn(N,BusySet)]->
(* The line "N" is not busy and also no forward feature is
   currently active on this telephone, and therefore a subscriber
   may invoke such feature on the same telephone.
*)
1060        user !N !ForK ?FN:Phone;
1061        System_Network [user,update,line]
                  (ServiceSet,BusySet,Insert(N,
                   ForwardSet),Insert(ForPair(N,FN),
                   ForwardPairs))
1062       []
1063       [IsIn(N,ForwardSet) and NotIn(N,BusySet)] ->
(* A forward feature is currently active on this telephone and
   also the line is not busy, and therefore a subscriber may
   cancel such feature on the same telephone.
*)
1064        user !N !ForK;
1065        System_Network [user,update,line]
            (ServiceSet,BusySet,Remove(N,ForwardSet),
            Remove(ForPair(N,ForwardTo(N,
            ForwardPairs)),ForwardPairs))
1066     endproc (* Forward_Feature2 *)
1067     endproc (* Forward_Feature *)
1068    endproc (* System_Network *)
1069  endspec (* Sample_Telephone *)
```

Appendix C Test Processes

1. The following process is a formal representation of a test sequence (of observable actions only) that validates two functionalities of the specification. First, if U1 calls U2 and U2 is free, a call may be established between U1 and U2. Then, while U1 and U2 are in conversation, another subscriber uses an extension of the telephone attached to the same line being used by U2. Therefore, that subscriber will get a busy tone *Busy_T*.

Note here that, since this sequence tests for two calls being processed in parallel, the internal action '**i**' is replaced by the gate '*line*', see Section 4.6. This is to allow the generation of a second call upon synchronization on gate '*line*'.

```
process Test_Sequence2 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  update  !U2 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  line ;
  user  !U2 !OffHook ;
  user  !U2 !Busy_T ;
  user  !U2 !HangUp ;
  exit
endproc (* Test_Sequence2 *)
```

2. This process describes the situation where a subscriber U1 dials a number whose line is not connected to the *Telephone System*. The subscriber U1 will then get an out of service tone *Oofs_T*.

```
process Test_Sequence3 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U1 !Oofs_T ;
  user  !U1 !HangUp ;
  exit
endproc (* Test_Sequence3 *)
```

3. The sequence presented by this process describes two calls being handled in parallel. In one call, subscriber U1 is calling U2. Meanwhile, subscriber U3 picks up the handset and tries to call U1. The subscriber U3 gets a busy tone *Busy_T*. It then invokes a *Ring Again* feature "*RiAgK*" and hangs up its telephone.

While the *Ring Again* feature is active on telephone U3, U1 goes *OnHook* "*HangUp*". Subscriber U3 then, gets a ring again notification "*Rano_T*". Once U3 goes *OffHook* to proceed in the call, telephone U1 starts ringing *Ring_T*. Subscriber U1 then goes *OffHook* and both subscribers, U1 and U3, enter in a conversation.

```
process Test_Sequence4 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(7,4,1,9,7,2,9),
      U3:Phone = Num(5,6,4,9,4,3,9) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  line ;
  user  !U3 !OffHook ;
  user  !U3 !Dito_T ;
  user  !U3 !DialKs !U1 ;
  user  !U3 !Busy_T ;
  user  !U3 !RiAgK ;
  user  !U3 !HangUp ;
  user  !U1 !HangUp ;
  user  !U3 !Rano_T ;
  user  !U3 !OffHook ;
  user  !U1 !Ring_T ;
  user  !U1 !OffHook ;
  user  !U3 !SendVo ;
  user  !U1 !SendVo ;
  exit
endproc (* Test_Sequence4 *)
```

4. This sequence of actions describes the (observable) behaviour of the situation where a subscriber on U1 calls another subscriber on telephone U2, while the latter has forwarded *ForK* further incoming calls to its telephone, to another pre-selected one "U3". However, because U3 is not connected to the *Telephone System*, U1 will surely get an out of service tone *Oofs_T*.

```
process Test_Sequence5 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  update  !U2 ;
  user  !U2 !ForK !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U1 !Oofs_T ;
  user  !U1 !HangUp ;
  exit
endproc (* Test_Sequence5 *)
```

5. This process describes a more complicated situation of a call processing. In this situation, two calls are being handled in parallel. Initially, four lines are connected to the *Telephone System* and a *Forward* feature *ForK* is active on telephone U2. Further incoming calls to U2 will be forwarded to U3 (as stated in the process below).

The subscriber on U1 calls U2. This call is automatically forwarded to the line U3 whose telephone starts ringing *Ring_T*. Then the communication path is established between subscribers on U1 and U3. Meanwhile, another call is being processed where a subscriber on U4 is calling U2. Because a *Forward* feature is still active on U2 and U3 is busy in the first call, subscriber will get a busy tone *Busy_T*.

```
process Test_Sequence6 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U2 !ForK !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
  line ;
  user  !U4 !OffHook ;
  user  !U4 !Dito_T ;
  user  !U4 !DialKs !U2 ;
  user  !U4 !Busy_T ;
  user  !U4 !HangUp ;
  exit
endproc (* Test_Sequence6 *)
```

6. This sequence handles the situation where there is a cycle in the forwarding process of a call. In this situation, U2 forwards "*ForK*" incoming calls to its telephone to the telephone on line U3. The latter, at the same time, forwards these calls to U4, while the telephone on U4 forwards those calls to the telephone on line U2 that has forwarded the same calls. If the system cannot detect such a cycle, a call may never return a signal.

In this example, a subscriber on line U1 calls the line U2. As discussed above, this call is forwarded from line U2 to line U3 then from U3 to line U4. It is also forwarded from line U4 to line U2. Now the system detects that this same call was forwarded by this line U2. It then, returns a busy tone *Busy_T* to the caller on line U1.

```
process Test_Sequence7 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U2 !ForK !U3 ;
  user  !U3 !ForK !U4 ;
  user  !U4 !ForK !U2 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U1 !Busy_T ;
  user  !U1 !HangUp ;
  exit
endproc (* Test_Sequence7 *)
```

7. This is the case where a call has been successfully forwarded. At the initial state, a subscriber invokes a *Forward* feature "*ForK*" on a telephone of line U2 to forward incoming calls to this telephone to line U3. Meanwhile, a *Forward* feature is active on a telephone of line U3. Its aim is to forward incoming calls to a telephone on line U4.

A subscriber on U1 goes *OffHook* to call the number U2. This call is forwarded from U2 to U3, then from U3 to U4 as its *End Destination*. The telephone on U4 is free and therefore it starts ringing *Ring_T*. The subscriber on U4 goes *OffHook* and the conversation may begin.

```
process Test_Sequence8 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U2 !ForK !U3 ;
  user  !U3 !ForK !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  exit
endproc (* Test_Sequence8 *)
```

8. In this sequence, a subscriber on U3 goes *OffHook* to make its line busy. At the same time, a subscriber on U1 goes *OffHook* and dials the number U2. The conversation between U1 and U2 may begin.

While in conversation, the subscriber on U1 invokes a *Hold* feature and calls U3. But, because U3 is busy, subscriber U1 will get a busy tone *Busy_T*. It must then, press the hold key *HoldK* again to return to the original call.

```
process Test_Sequence9 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  user  !U3 !OffHook ;
  user  !U3 !Dito_T ;
  line ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !HoldK ;
  user  !U1 !DialKs !U3 ;
  user  !U1 !Busy_T ;
  user  !U1 !HoldK ;
  exit
endproc (* Test_Sequence9 *)
```

9. In this sequence, two features are concerned. First, a subscriber on a telephone of line U3 forwards incoming calls to its telephone to line U4. Then, a subscriber on U1 calls U2 that is currently free. After the conversation between the subscribers on U1 and U2 begins, the subscriber on U1 invokes a *Hold* feature "*HoldK*" and calls U3.

The call is forwarded to U4. Since U4 has been added to the *Telephone System* and it is currently free, the associated telephone starts ringing *Ring_T* and the conversation between the subscribers on U1 and U4 may begin.

```
process Test_Sequence10 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(7,4,1,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U3 !ForK !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !HoldK ;
  user  !U1 !DialKs !U3 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !HoldK ;
  user  !U4 !HangUp ;
  exit
endproc (* Test_Sequence10 *)
```

10. In this process, a subscriber on line U1 calls a subscriber on line U2. Then, when the call terminates to U2, the latter invokes a *Transfer* feature "*TransK*" to transfer this call to a telephone on line U3, then establish a *Three-Way Calling* between subscribers using the lines U1,U2, and U3. This is achieved by the second action *TransK*. Line U3 is free, and therefore its telephone starts ringing *Ring_T*.

```
process Test_Sequence11 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U2 !TransK ;
  user  !U2 !DialKs !U3 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U2 !TransK ;
  exit
endproc (* Test_Sequence11 *)
```

11. The scenario in this process is similar to the one discussed above in process *Test_Sequence11* except that, the subscriber on U2 transfers *TransK* the calls to U3 then leaves for good "*ReleaseK*" the conversation. The feature is then called a *Transfer* feature for short.

```
process Test_Sequence12 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U2 !TransK ;
  user  !U2 !DialKs !U3 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U2 !ReleaseK ;
  user  !U2 !HangUp ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
  exit
endproc (* Test_Sequence12 *)
```

12. The current situation is similar to the one presented in process *Test_Sequence12* except that a *Forward* feature is active "*ForK*" on a telephone of line U2. Therefore, the call is forwarded to U3. The latter invokes a *Transfer* feature and transfers "*TransK*" the call to U4 then leaves for good "*ReleaseK*" the call. The conversation is now between U1 and U4.

```
process Test_Sequence13 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U2 !ForK !U3 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
  user  !U3 !TransK ;
  user  !U3 !DialKs !U4 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U3 !SendVo ;
  user  !U4 !SendVo ;
  user  !U3 !ReleaseK ;
  user  !U3 !HangUp ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  exit
endproc (* Test_Sequence13 *)
```

13. This situation is similar to the one presented in process *Test_Sequence13*, except that the line to whom the call is forwarded from U3, after its transfer to U3, is currently not connected to the *Telephone System* "*Oofs_T*".

```
process Test_Sequence14 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  user  !U3 !ForK !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U2 !TransK ;
  user  !U2 !DialKs !U3 ;
  user  !U2 !Oofs_T ;
  user  !U2 !TransK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  exit
endproc (* Test_Sequence14 *)
```

14. This process describes a sample situation where a *Conference Call* feature is requested by subscribers while they are in conversation.

Initially, four lines are connected to the *Telephone System*. A subscriber using the line U1 calls a subscriber whose telephone number is U2. While in conversation, U1 invokes a conference feature *ConfK* and calls a new subscriber whose telephone number is U3. At this stage, U1 is in a private conversation with U3. The subscriber U1 presses the key *ConfK* to bring U3 to the conference conversation.

Then, while the subscribers whose numbers are U1,U2, and U3, are in a conference conversation, U1 invokes a conference feature *ConfK* and calls another new subscriber whose number is U4. Currently, U1 is in a private conversation with U4. U1 then presses the key *ConfK* and brings U4 to the conference conversation.

At this stage of the call, all subscribers whose numbers are U1, U2, U3, and U4, are in a *Conference* conversation.

```
process Test_Sequence15 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !ConfK ;
```

```
  user  !U1 !DialKs !U3 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U4 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U4 !SendVo ;
  exit
endproc (* Test_Sequence15 *)
```

15. The following test process describes a more complicated situation of a call. Initially, four lines are connected to the *Telephone System*. A *Conference* conversation is established between these telephone as explained in process *Test_Sequence15* above.

While in conversation, the subscriber on telephone U1 invokes a conference feature *ConfK* and calls a new subscriber of telephone number U5. Because the line U5 is not connected to the *Telephone System*, the subscriber using the line U1 gets an out of service tone *Oofs_T*.

The subscriber invoking this feature, then presses the conference key *ConfK* and returns to the conference conversation.

```
process Test_Sequence16 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2),
      U5:Phone = Num(5,6,0,8,6,9,0) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U3 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
```

```
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U4 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U5 ;
  user  !U1 !Oofs_T ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U4 !SendVo ;
  exit
endproc (* Test_Sequence16 *)
```

16. In the following situation, two features are of involved in call, the *Forward* and the *Conference Call* features.

Initially, five lines (U1, U2, U3, U4, and U5) are connected to the *Telephone System*. Also, two telephones using the lines, whose numbers are U3 and U5, invoke a forward feature. U3 wants "*ForK*" that any further incoming call to its line to be forwarded to a line number U4. Any call that comes to a telephone on the line U5 will also be forwarded "*ForK*" but to the line U2.

A subscriber using U1 starts the call by establishing a conversation with line U2. Then, the former invokes the conference feature *ConfK* and calls line U3. Since a *Forward* feature is active on U3 and U4 is free, this new call will successfully terminate to U4. Then, U1 brings U4 to the original call. Currently all telephones using the lines U1, U2, and U4 are in a conference conversation.

Again, a subscriber on U1 invokes the conference feature *ConfK* and calls U5. Since a forward feature "*ForK*" is active on U5, this new is forwarded to U2 as stated above. But, U2 is busy with the original call, and therefore the subscriber using U1 gets a busy tone *Busy_T* and returns *ConfK* to the conference conversation with the subscribers using the lines U2 and U4.

```
process Test_Sequence17 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2),
      U5:Phone = Num(5,6,0,8,6,9,0) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
  update  !U5 ;
  user  !U3 !ForK !U4 ;
  user  !U5 !ForK !U2 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
```

```
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U3 ;
  user  !U4 !Ring_T ;
  user  !U4 !OffHook ;
  user  !U1 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U4 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U5 ;
  user  !U1 !Busy_T ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U4 !SendVo ;
  exit
endproc (* Test_Sequence17 *)
```

17. In this sequence, two features (*Forward* and *Conference Call* features) are involved, but it describes also the situation where there is a cycle in the forwarding process.

Initially, six lines (U1, U2, U3, U4, U5, and U6) are connected to the *Telephone System* and a subscriber on line U4 invokes a forward feature *ForK* to forward incoming calls to its telephone to the line U5. The latter also invokes a forward feature *ForK* to forward incoming calls to its telephone to the line U6. In its turn, a subscriber on U6 invokes the same feature *ForK* to forward incoming calls to its telephone to the line U4.

A subscriber on U1 starts by establishing a call with U2. Then, it invokes a conference feature *ConfK* and calls U3. Currently, all lines (U1, U2, and U3) are connected to each other in a conference conversation.

The subscriber on U1, invokes again the conference feature *ConfK* and calls U4. But, because a forward feature is still active on U4, this new call is forwarded to U5. Also, a forward feature is active on U5, and therefore this call is forwarded to U6 as stated above. Again, the same feature is active on U6, and therefore the call is forwarded to U4.

However, U4 is currently busy with the original call, and therefore the caller (u1) gets a busy tone *Busy_T*. Then, the subscriber on U1 presses the conference key *ConfK* to return to the original call and join the other subscribers using the lines U2, and U3 in the conference conversation.

```
process Test_Sequence18 [user,update] :exit:=
  let U1:Phone = Num(7,3,1,6,9,7,9),
      U2:Phone = Num(5,6,4,9,4,3,9),
      U3:Phone = Num(7,4,1,9,7,2,9),
      U4:Phone = Num(5,6,4,9,2,3,2),
      U5:Phone = Num(5,6,0,8,6,9,0),
      U6:Phone = Num(5,6,0,8,5,4,5) in
  update  !U1 ;
  update  !U2 ;
  update  !U3 ;
  update  !U4 ;
```

```
  update  !U5 ;
  update  !U6 ;
  user  !U4 !ForK !U5 ;
  user  !U5 !ForK !U6 ;
  user  !U6 !ForK !U4 ;
  user  !U1 !OffHook ;
  user  !U1 !Dito_T ;
  user  !U1 !DialKs !U2 ;
  user  !U2 !Ring_T ;
  user  !U2 !OffHook ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U3 ;
  user  !U3 !Ring_T ;
  user  !U3 !OffHook ;
  user  !U1 !SendVo ;
  user  !U3 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  user  !U1 !ConfK ;
  user  !U1 !DialKs !U4 ;
  user  !U1 !Busy_T ;
  user  !U1 !ConfK ;
  user  !U1 !SendVo ;
  user  !U2 !SendVo ;
  user  !U3 !SendVo ;
  exit
endproc (* Test_Sequence18 *)
```

Appendix D Symbolic Execution Trees of Processes

1. ***Process Ring_Again_Feature Symbolic Tree***

The following symbolic execution tree describes the behaviour of the process *Ring_Again_Feature* to a depth of execution of 8. The number associated with an action is the number of the branch at a given level.

```
1 user !Phone&0:Phone !HangUp:Key [277]
| 1 user !Phone&0:Phone !RiAgK:Key [278]
    ==> continue
| | 1 user !Phone&0:Phone !Rano_T:Tone [283]
| | | 1 user !Phone&0:Phone !OffHook:Key [284]
| | | | | | 1 user !Phone&0:Phone !SendVo:Key [528]
                ==> continue
| | | | | | 2 user !Phone&0:Phone !HoldK:Key [369]
                ==> continue
| | | | | | 4 user !Phone&0:Phone !TransK:Key [369]
                ==> continue
| | | | | | 6 user !Phone&0:Phone !ConfK:Key [369]
                ==> continue
| | | | | | 8 user !Phone&0:Phone !HangUp:Key [541]
                ==> continue
| | | | | | 1 user !Phone&0:Phone !Busy_T:Tone [290]
| | | | | | | 1 user !Phone&0:Phone !HangUp:Key [277]
                  ==> continue
| | | 2 user !Phone&0:Phone !RiAgK:Key [294]
        ==> continue
```

2. ***Process Destination_Side Symbolic Tree***

The first event on the *Destination Side*'s telephone is when a call terminates to it. The telephone starts ringing and the subscriber may go *OffHook* or the call is aborted and it is disconnected.

```
| | 1 user !Phone@1:Phone !Ring_T:Tone [334]
| | | | | 2 user !Phone@1:Phone !OffHook:Key [337]
              ==> continue
| | | | 2 user !Phone@1:Phone !OffHook:Key [337]
            ==> continue
| | | 2 user !Phone@1:Phone !OffHook:Key [337]
| | | | | 1 user !Phone@1:Phone !SendVo:Key [528]
              ==> continue
| | | | | 2 user !Phone@1:Phone !HoldK:Key [369]
              ==> continue
| | | | | 4 user !Phone@1:Phone !TransK:Key [369]
              ==> continue
| | | | | 6 user !Phone@1:Phone !ConfK:Key [369]
              ==> continue
| | | | | 8 user !Phone@1:Phone !HangUp:Key [541]
              ==> continue
```

3. ***Process Invoke_Feature Symbolic Tree***

To invoke a feature, a subscriber must request the associated feature key. This event is not shown in the following symbolic tree, because this action is specified in process *Feature*, before the instantiation of process *Invoke_Feature*. The event number 2 in column 2, states that a feature may be cancelled at this stage, and also does the event number 2 in column 3.

```
| 1 user !Phone&0:Phone !DialKs:Key ?Phone@2.1:
    Phone [380]
| | | | 1 user !Phone&0:Phone !Busy_T:Tone [388]
| | | | | 1 user !Phone&0:Phone !K&0:Key [517]
            ==> continue
| | | | 1 user !Phone&0:Phone !Oofs_T:Tone [392]
| | | | | 1 user !Phone&0:Phone !K&0:Key [517]
            ==> continue
| | | | 1 user !Phone&0:Phone !SendVo:Key [528]
| | | | | 3 [==(K&0,TransK)]user !Phone&0:Phone
            !TransK:Key [409] ==> continue
| | | | | 4 [==(K&0,TransK)]user !Phone&0:Phone
            !ReleaseK:Key [419] ==> continue
| | | | | 5 [<>(K&0,TransK)]user !Phone&0:Phone
            !K&0:Key [517] ==> continue
| | | | | 1 [==(K&0,TransK)]user !Phone&0:Phone
            !TransK:Key [400] ==> continue
| | | | | 2 [<>(K&0,TransK)]user !Phone&0:Phone
            !K&0:Key [517] ==> continue
| | | | 3 [==(K&0,TransK)]user !Phone&0:Phone
          !TransK:Key [409] ==> continue
| | | | 4 [==(K&0,TransK)]user !Phone&0:Phone
          !ReleaseK:Key [419] ==> continue
| | | | 5 [<>(K&0,TransK)]user !Phone&0:Phone
          !K&0:Key [517] ==> continue
| | | 4 user !Phone&0:Phone !K&0:Key [517]
        ==> continue
| | 2 user !Phone&0:Phone !K&0:Key [517]
      ==> continue
| 2 user !Phone&0:Phone !K&0:Key [517]
    ==> continue
```

4. ***Process Additional_Subscriber Symbolic Tree***

This symbolic tree of process *Additional_Subscriber* presented below, shows all the possible "*OffHook*" events with the new subscriber that is called from within another call. Such an event may occur only after the telephone starts ringing "*Ring_T*".

```
| | 1 user !Phone@1:Phone !Ring_T:Tone [461]
| | | | | 2 user !Phone@1:Phone !OffHook:Key [464]
              ==> continue
| | | | 2 user !Phone@1:Phone !OffHook:Key [464]
            ==> continue
| | | 2 user !Phone@1:Phone !OffHook:Key [464]
| | | | | 1 user !Phone@1:Phone !SendVo:Key [528]
              ==> continue
| | | | | 2 user !Phone@1:Phone !HangUp:Key [541]
              ==> continue
```

5. ***Process Party_On_Hold Symbolic Tree***

The following symbolic tree represents the first stage of a behaviour of the party waiting for the other side to finish with the new call. The party on hold may at any time leave the call "*HangUp*".

```
| | 1 [==(sign&0,TRANN)]user !Phone&0:Phone
      !SendVo:Key [528] ==> continue
| | 3 [==(sign&0,TRANN)]user !Phone&0:Phone
      !HangUp:Key [541] ==> continue
| | 4 [==(sign&0,CONFN)]user !Phone&0:Phone
      !SendVo:Key [528] ==> continue
| | 5 [==(sign&0,CONFN)]user !Phone&0:Phone
      !ConfK:Key [369] ==> continue
| | 7 [==(sign&0,CONFN)]user !Phone&0:Phone
      !HangUp:Key [541] ==> continue
| | 8 [==(sign&0,HOLDN)]user !Phone&0:Phone
      !SendVo:Key [528] ==> continue
| | 9 [==(sign&0,HOLDN)]user !Phone&0:Phone
      !HoldK:Key [369] ==> continue
| | 11 [==(sign&0,HOLDN)]user !Phone&0:Phone
       !TransK:Key [369] ==> continue
| | 13 [==(sign&0,HOLDN)]user !Phone&0:Phone
       !ConfK:Key [369] ==> continue
| | 15 [==(sign&0,HOLDN)]user !Phone&0:Phone
       !HangUp:Key [541] ==> continue
| 3 user !Phone&0:Phone !HangUp:Key [541]
    ==> continue
```

Bibliography

[BB87] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[Bel87] Bell Communications Research. *Call Processing. LATA Switching Systems Generic Requirements (LSSGR)*, July 1987.

[Bel88] Bell Canada, Inc. *The DMS-100 Business Set User's Manual*, 1988.

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

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *JACM*, 31(3):560–599, 1984.

[BL91] R. Boumezbeur and L. Logrippo. Specification and Validation of Telephone Systems in LOTOS. In *TRIO Retreat*. TRIO, May 1991.

[Bri88a] E. Brinksma. On the Design of Extended LOTOS. A Specification Language for Distributed Systems. Doctoral Dissertation, 1988. University of Twente, The Netherlands.

[Bri88b] E. Brinksma. A Theory for the Derivation of Tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 63–74. North-Holland, 1988.

[BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS Specifications, their Implementations and their Tests. In G. von Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing, and Verification VI*, pages 349–360. North-Holland, 1987.

[CCI88] CCITT. *Recommendation Z100. Specification and Description Language*, 1988.

[COR81] CORNAFION. *Systemes Informatiques Repartis. Concepts et Techniques*. BORDAS, Paris, 1981.

[CV90] R. C. Cam and S. T. Vuong. A Formal Specification, in LOTOS, of a Simplified Cellular Mobile Communication System. In S. T. Vuong, editor, *Formal Description Techniques II*, pages 485–499. North-Holland, 1990.

[FLS90] M. Faci, L. Logrippo, and B. Stepien. Formal Specifications of Telephone Systems in LOTOS. In E. Brinksma, G. Scollo, and C. Vissers, editors, *Protocol Specification, Testing, and Verification IX*. North-Holland, 1990.

[FLS91] M. Faci, L. Logrippo, and B. Stepien. Formal Specifications of Telephone Systems in LOTOS: The Constraint-Oriented Style Approach. *Computer Networks and ISDN Systems*, 21:53–67, 1991.

[Gal89] S. Gallouzi. Trace Analysis of LOTOS Behaviours. Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 1989.

[GHHL88] R. Guillement, M. Haj-Hussein, and L. Logrippo. Executing Large LOTOS Specifications. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 399–410. North-Holland, 1988.

[GL89a] D. Gueraichi and L. Logrippo. Derivation of Test Cases for LAP-B from a LOTOS Specification. In S. T. Vuong, editor, *Formal Description Techniques II*, pages 361–374. North-Holland, 1989.

[GL89b] R. Guillemot and L. Logrippo. Derivation of Useful Execution Trees from LOTOS by using an Interpreter. In K. J. Turner, editor, *Formal Description Techniques*, pages 311–325. North-Holland, 1989.

[GM86] N. Gehani and A.D. McGettrick, editors. *Software Specification Techniques*. International Computer Science Series. AT and T Bell Telephone Laboratories, Inc., 1986.

[Gue89] D. Gueraichi. Derivation of Test Cases for LAP-B from a Formal Specification in LOTOS. Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, 1989.

[HH88] M. Haj-Hussein. An Interactive System for LOTOS Applications (ISLA). Master's thesis, University of Ottawa, Ottawa, Ontario, Canada, November 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, UK, LTD., 1985.

[ISO88] ISO, DIS 8807. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, June 1988.

[ISO89] ISO, IS 8807. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, May 1989.

[Kar89] G. M. Karam. MLOG: A Language for Prototyping Concurrent Systems. Technical Report TR SCE-88-3, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, Canada, March 1989.

[Lan85] M. G. Lane. *Data Communications Software Design*. Computer Science series. Boyd and Fraser Publishing Company, 1985.

[LOBF88] L. Logrippo, A. Obaid, J.P. Briand, and M.C. Fehri. An Interpreter for LOTOS, a Specification Language for Distributed Systems. *Software-Practice and Experience*, 18:365–385, 1988.

[Mar72] J. Martin. *Introduction to Teleprocessing*. Prentice-Hall, Inc., 1972.

[MdMvT89] J. A. Manas, T. de Miguel, and H. van Thienen. The Implementation of a Specification Language for OSI Systems. In P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 409–421. North-Holland, 1989.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Sringer-Verlag, 1980.

[Mye79] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[QAF89] J. Quemada, A. Azcorra, and D. Frutos. TIC: A Timed Calculus for LOTOS. In S. T. Vuong, editor, *Formal Description Techniques II*, pages 195–209. North-Holland, 1989.

[QF87] J. Quemada and A. Fernandez. Introduction of Quantitative Relative Time in LOTOS. In H. Rudin and C.H. West, editors, *Protocol Specification, Testing, and Verification VII*, pages 105–121. North-Holland, 1987.

[Sim89] L. Simon. An Object-Oriented Delta Approach to Telecommunication Service Specification. Master's thesis, Carleton University, Ottawa, Ontario, Canada, December 13, 1989.

[Sin82] W. Sinnema. *Digital, Analog, and Data Communication*. Reston Publishing Company, Inc., Reston, Virginia, 1982.

[Spr91] J.D. Spragins, editor. *Telecommunications. Protocols and Design*. Addison-Wesley Publishing Company, Inc., 1991.

[Tan84] A. S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, Inc., 1984.

[Tan89] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Inc., second edition, 1989.

[Tur87] K. J. Turner. The Formal Specification Language LOTOS. A Course for Users, August 1987. University of Stirling.

[Tur88] K. J. Turner. Constraint-Oriented Style in LOTOS, 1988. University of Stirling.

[Tvr89] I. Tvrdy. Formal Modelling of Telematics Services Using LOTOS. *Microprocessing and Microprogramming*, 25(1-5):313–317, 1989.

[vEVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. North-Holland, 1989.

[vHTZ89] V. H. P. van Hulzen, P. A. J. Tilanus, and H. Zuidweg. LOTOS Extended with Clocks. In S. T. Vuong, editor, *Formal Description Techniques II*, pages 179–193. North-Holland, 1989.

[VL86] C. A. Vissers and L. Logrippo. The Importance of the Service Concept in the Design of Data Communications Protocols. In M. Diaz, editor, *Protocol Specification, Testing, and Verification V*. North-Holland, 1986.

[VSA+89] C.A. Vissers, G. Scollo, R.B. Alderden, J. Schot, and L. Ferreira Pires. *The Architecture of Interaction Systems. The Structuring of Distributed Systems*. Lecture Notes. C.A. Vissers, Enschede, The Netherlands, February 1989.

[VSvS88] C. A. Vissers, G. Scollo, and M. van Sinderen. Architecture and Specification Style in Formal Descriptions of Distributed Systems. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, pages 189–204. North-Holland, 1988.

[Zav85] P. Zave. The Distributed Alternative to Finite-State-Machine Specifications. *ACM Trans. On Prog. Lang. and Systems*, 7(1):10–36, 1985.