1

MicroD

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**An Object-Oriented Delta Approach to
Telecommunication Service Specification**

by

Linda Simon

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario
January 10, 1990

© 1990, Linda Simon

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-59033-5

The undersigned hereby recommended to the Faculty of Graduate
Studies and Research acceptance of the thesis

*An Object-Oriented Delta Approach to
Telecommunication Service Specification*

submitted by Linda Simon, B.Eng.,
in partial fulfilment of the requirements
for the degree of Master of Engineering.

Dr. R.J. Buhr
Thesis Supervisor

Dr. C.M. Woodside
Thesis Supervisor

Dr. S.A. Mahmoud,
Chairman,
Department of Systems and Computer Engineering

Carleton University
January 15, 1990

ABSTRACT

An "Object-Oriented Delta" specification approach is presented as a means to address the problems in specifying a class of telecommunication services known as supplementary services. Supplementary services are those which modify a basic telecommunication service (e.g. telephony) by adding or changing its characteristics or behaviour.

In this approach, supplementary services are specified in terms of the differential behaviour they introduce to the basic telecommunication service. Alternatives for expressing this "delta" behaviour within an object-oriented framework are presented which exploit the notions of inheritance and orthogonality, respectively.

The concepts of the "object-oriented delta" approach are explored through its application to a number of individual and combined supplementary services. The example supplementary service specifications are then translated into an executable specification using the MLog programming language.

It is concluded that the object-oriented delta specification approach provides a compact and structured way for defining individual and combined supplementary services.

ACKNOWLEDGEMENTS

I wish to express my gratitude to Professor Woodside and Professor Buhr, who acted as joint thesis supervisors, for their valuable advice, encouragement and responsiveness. Many thanks to Gerald Karam for his MLog advice and fruitful discussions.

Thanks to those involved in the service specification process within the standards forums for their perspectives. Also thanks to Mac for assisting in the presentation aspects of this work.

Finally, I would like to thank my family for their patience, understanding and support. Special appreciation is due to my husband, Fred, to whom this thesis is dedicated.

TABLE OF CONTENTS

CHAPTER 1 Introduction.....	1
1.1 Background and Motivation.....	1
1.2 Thesis Objectives.....	3
1.3 Thesis Contributions.....	4
1.4 Thesis Organization.....	5
CHAPTER 2 Overview of Telecommunication Services and Their Specification Approaches.....	7
2.1 Aspects of Telecommunication Service Provision.....	7
2.2 Overview of the Current Telephone Network.....	8
2.3 Reference Configuration.....	10
2.3.1 Subscriber Access.....	11
2.3.2 Network Signalling.....	12
2.4 Interactions between Telecommunication Systems.....	12
2.5 Specification Techniques.....	16
2.6 CCITT Service Specification Method.....	19
2.7 Study Method.....	20
CHAPTER 3 Object Model for the Basic Telecommunication Service..	22
3.1 Definition of the Basic Telecommunication Service.....	22
3.2 Overview of Object Model.....	23
3.2.1 Concepts.....	23
3.2.2 Comparison with SDL.....	26
3.3 Basic Service Objects and Their Communication Relationships.....	27
3.3.1 Network Service Provider Object.....	29
3.3.2 Subscriber Access Control Object.....	32
3.3.3 Association Control Object.....	35
3.3.4 Other Objects.....	36
3.4 Mapping Onto Telecommunication Systems.....	37
3.5 Interactions between Objects.....	41
3.5.1 Channel Definition.....	41
3.5.2 Subscriber Access Channel.....	42
3.5.3 Subscriber Association Control Channel.....	43
3.5.4 Subscriber Resource Channel.....	44
3.5.5 Association Resource Channel.....	45
3.5.6 Other Channels Between Network Service Provider Sub-Objects.....	46
3.5.7 Channels Between Subscriber Access Control Sub-Objects.....	47
3.5.8 Channels Between Association Control Sub-Objects.....	48
3.5.9 Subscriber Association Control Message Routes.....	48
3.5.10 Association Control Message Routes.....	51
3.6 Temporal behaviour of Objects.....	53
3.6.1 Semantics of Object Communication.....	53
3.6.2 State Diagram Notation.....	54
3.6.2.1 Input and Output Expressions.....	55

3.6.2.2	Condition Expressions.....	57
3.6.2.3	Action Expressions.....	58
3.6.2.4	An Example of Expression Syntax.....	59
3.6.2.5	Comparison with other Notations.....	60
3.6.3	Subscriber Access Control.....	61
3.6.3.1	Transaction Control.....	62
3.6.3.2	Outgoing Association.....	63
3.6.3.2.1	Resource Allocation and Establishment of AC Identity.....	65
3.6.3.2.2	Outgoing Association Establishment.....	68
3.6.3.2.3	Association Disestablishment.....	69
3.6.3.3	Incoming Association.....	71
3.6.4	Association Control.....	73
3.6.4.1	Association Control Creator.....	73
3.6.4.2	Bearer Association Control.....	74
3.6.4.2.1	Establishment of Responder's Identity.....	74
3.6.4.2.2	Establishment of Association.....	76
3.6.4.2.3	Release of Association.....	78
3.7	Summary.....	79
CHAPTER 4	Overview of Delta Service Specification Approach.....	80
4.1	Overview of the Service Specification Problem.....	81
4.2	Service Delta Components as Specification Building Blocks.....	84
4.3	Representation of Service Delta Components within Object Model.....	86
4.3.1	Supplementary Service Delta Object Classes.....	86
4.3.2	A State Diagram Notation for Expressing Delta Behaviour.....	90
4.3.2.1	Overview of Notation.....	90
4.3.2.2	Comparison with OSDL.....	94
4.4	Expressing Combined Supplementary Services.....	95
4.4.1	Example 1 Combining Services Formed of Peer Objects.....	96
4.4.2	Example 2 Combining Services Formed of Peer and Subclass.....	96
4.4.3	Example 3 Combining Services Formed of Subclass Objects.....	99
4.4.4	Conflicts in Combining Services.....	102
4.5	Summary of the Delta Service Specification Approach.....	105
CHAPTER 5	Application of Delta Approach to Supplementary Service Specification.....	107
5.1	Hold Service.....	108
5.1.1	Interaction with the Basic Telecommunication Service.....	108
5.1.1.1	Outgoing Association Object Interactions.....	108
5.1.1.2	Incoming Association Object Interactions.....	111
5.1.1.3	Association Control Object Interaction.....	111
5.1.2	Expression of Hold Delta Behaviour for Incoming and Outgoing Association Objects.....	112
5.1.2.1	Peer Object Specification.....	112

5.1.2.2 Subclass Object Specification.....	118
5.1.3 Expression of Hold Delta Behaviour for Association Control.....	120
5.1.4 Hold Service Summary	121
5.2 Call Waiting Service	123
5.2.1 Interactions with the Basic Telecommunication Service	123
5.2.1.1 No Channel Available on Incoming Association.....	123
5.2.1.2 Call Waiting Notification.....	124
5.2.1.3 Receipt of Connect Message.....	124
5.2.1.4 Decrement of Number of Waiting Calls	125
5.2.2 Expression of Call Waiting Delta Behaviour.....	125
5.2.3 Summary of the Call Waiting Service.....	127
5.3 Call Forwarding Services.....	127
5.3.1 Interactions With the Basic Telecommunication Service	128
5.3.1.1 Service Activation/Deactivation	129
5.3.1.2 Service Invocation.....	130
5.3.1.2.1 CFU Interactions.....	130
5.3.1.2.2 CFB Interactions	132
5.3.1.2.3 CFNR Interactions.....	133
5.3.2 Expression of Call Forwarding Delta Behaviour.....	135
5.3.2.1 Service Activation/Deactivation	135
5.3.2.2 Transaction Control Object.....	135
5.3.2.3 Incoming Association Object.....	137
5.3.2.4 Outgoing Association Object.....	137
5.3.2.5 Association Control Object.....	141
5.3.3 Summary of Call Forwarding Services.....	141
5.4. CCBS Service.....	144
5.4.1 Interactions with the Basic Telecommunication Service.....	144
5.4.2 Expression of Delta Behaviour for CCBS Monitoring/Scanning Requirements	147
5.4.3 Expression of CCBS Delta Behaviour for Outgoing Association.....	150
5.4.3.1 Peer Object Specification.....	150
5.4.3.2 CCBS Subclass Object Specification.....	153
5.4.4 CCBS Service Summary	154
5.5 Combination of Services.....	154
5.5.1 Combining Service Subclass Deltas Derived from Same SuperClass.....	156
5.5.1.1 CFU and CFB.....	156
5.5.1.2 Call Waiting and CFNR.....	157
5.5.1.3 Call Waiting and CFB.....	158
5.5.1.4 CBS and CFU.....	159
5.5.1.5 Summary of Service Combination.....	160
5.5.2 Combining Service Deltas Derived from Different SuperClass.....	160
5.5.2.1 Hold and Other Services	161
5.5.2.2 CCBS and CFU	161
5.6 Summary of Service Analysis.....	162

CHAPTER 6 Telecommunication Service Specification and Validation using MLog.....	166
6.1 Introduction To MLog.....	167
6.1.1 MLog OOPS Roots.....	167
6.1.2 MLog Logic Programming Roots.....	169
6.1.3 Concurrency and Backtracking in MLog.....	171
6.1.4 Motivation for MLog.....	174
6.2 Overview of the Approach to Mapping Service Specifications To MLog.....	174
6.2.1 Communication between Objects.....	175
6.2.1.1 Potential for Deadlock.....	175
6.2.1.2 Impact on State Machines.....	179
6.2.2 Expression of State Machines in Logic.....	180
6.2.2.1 Decomposition of State Machine Rules.....	180
6.2.2.2 Modification of Rules by Subclasses.....	183
6.2.3 Runtime Specification.....	184
6.2.4 Expression of Combined Deltas.....	185
6.3 Scope of MLog Telecommunication Service Prototype.....	186
6.4 MLog Object Classes.....	188
6.4.1 Basic Service Object Classes.....	188
6.4.1.1 Object.....	188
6.4.1.2 Main.....	189
6.4.1.3 Transaction Control.....	189
6.4.1.4 Subscriber Association.....	191
6.4.1.5 Association Control.....	193
6.4.1.6 Service Provision.....	193
6.4.1.7 Resource Control.....	194
6.4.1.8 Directory Database.....	195
6.4.1.9 Timer.....	195
6.4.1.10 Transporter and Message Queue.....	196
6.4.1.11 Datafill.....	196
6.4.1.12 Subscriber.....	196
6.4.2 Supplementary Service Delta Object Classes.....	197
6.4.2.1 Supplementary Service Database.....	197
6.4.2.2 TC Hold.....	198
6.4.2.3 Hold Service.....	198
6.4.2.4 Service Activation Control Delta.....	199
6.4.2.5 TC Call Forwarding Deltas.....	200
6.4.2.6 AC Call Forwarding Deltas.....	201
6.4.2.7 IA Call Forwarding Deltas.....	201
6.4.2.8 Call Waiting Class.....	202
6.4.2.9 Call Waiting/CFNR Class.....	202
6.4.3 Summary.....	203
6.5 MLog Prototype Run Time Behaviour.....	203
6.6 Representation of Object Behaviour.....	207
6.6.1 Transaction Control.....	208
6.6.2 TC Service Deltas.....	209

6.6.3 Outgoing Association, Incoming Association and Association.....	210
6.6.4 IA and AC Service Deltas.....	212
6.7 Validation of the Telecommunication Services.....	213
6.8 Design Issues.....	214
6.8.1 Number of Object Instances.....	214
6.8.2 Interaction Architectures.....	214
6.8.3 Determination of Appropriate Service Object	215
6.8.4 Reconfiguring Behaviour.....	215
6.8.5 Combined Services.....	216
6.9 Summary of MLog Prototype	217
CHAPTER 7 Summary and Conclusions.....	219
7.1 Summary	219
7.2 Conclusions and Suggestions for Further Work	221
REFERENCES.....	224
APPENDIX A List of Acronyms and Abbreviations.....	A.1
APPENDIX B Supplementary Service Prose Description.....	B.1
APPENDIX C MLog Telecommunication Service Prototype Source Code	C.1
APPENDIX D Sample Traces From MLog Telecommunication Services Prototype.....	D.1

LIST OF FIGURES

Figure 2.1: Reference Configuration for the Current Telephone Network.....	9
Figure 2.2: Subscriber and Network Interfaces Assumed in Study.....	11
Figure 2.3: Service Scenarios.....	14
Figure 2.3 (cont.): Service Scenarios	15
Figure 3.1: An Example Showing Objects and Their Interactions.....	25
Figure 3.2: An Example Showing Object Decomposition.....	26
Figure 3.3: Top Level Basic Service Object View.....	28
Figure 3.4: Decomposition of the Network Service Provider Object.....	30
Figure 3.5: Decomposition of the Subscriber Access Control Object.....	33
Figure 3.6: Decomposition of the Association Control Object.....	35
Figure 3.7: Mapping of Object Model to Telecommunication Systems.....	38
Figure 3.8: Association Types.....	39
Figure 3.8(cont.): Association Types.....	40
Figure 3.9: Transaction Control State Diagram.....	62
Figure 3.10: Outgoing Association State Diagram.....	64
Figure 3.11: Zoom-in of the Outgoing Resource Allocation and Establish AC Identity States.....	67
Figure 3.12: Zoom-in of the Outgoing Association Establishment State.....	68
Figure 3.13: Zoom-in of Association Disestablishment State.....	70
Figure 3.14: Incoming Association State Diagram.....	72
Figure 3.15: Association Control Creator State Diagram.....	73
Figure 3.16: Bearer Association Control State Diagram.....	75
Figure 3.17: Zoom-in of the Establish Switch Path State.....	75
Figure 4.1: Combinatorial Service Specification Problem.....	82
Figure 4.2: Delta Approach to Service Specification Problem.....	85
Figure 4.3: Approaches to Express Delta Behaviour Within Object-oriented Framework.....	89
Figure 4.4: State Diagram Notation for Expressing Delta Behaviour in Subclass Approach.....	92
Figure 4.5: Example 1: Combining Service Deltas with Peer Objects.....	97

Figure 4.6: Example 2: Combining Service Deltas with Peer and Subclass Objects.....	98
Figure 4.7: Example 3: Combining Service Deltas with Subclass Objects.....	100
Figure 4.8: Combining Subclass Objects with and without Conflicts.....	101
Figure 4.9: Combining Peer Service Deltas with Conflicts.....	103
Figure 4.10: Combining Subclass Service Deltas with Conflicts.....	104
Figure 5.1: Hold Service Interaction.....	109
Figure 5.2: A Peer Object Approach to Hold Service.....	112
Figure 5.3: Transaction Control Hold Delta (subclass approach).....	114
Figure 5.4: Association Control Hold Delta (subclass approach).....	122
Figure 5.5: Call Waiting Delta (subclass approach).....	126
Figure 5.6: CF Activation Control Delta (peer object approach).....	129
Figure 5.7: TC CFU and TC CFB Deltas (subclass approach).....	136
Figure 5.8: IA CFB Delta (subclass approach).....	138
Figure 5.9: IA CFNR Delta (subclass approach).....	139
Figure 5.10: OA CFU/B and OA CFNR Deltas (subclass approach).....	140
Figure 5.11: AC CFU and AC CFB Deltas (subclass approach).....	142
Figure 5.12: AC CFNR Delta (subclass approach).....	143
Figure 5.13: CCBS Interaction.....	145
Figure 5.14: CCBS Monitoring/Scanning Delta (peer object approach).....	148
Figure 5.15: OA CCBS Delta (subclass approach).....	152
Figure 5.16: Summary of Supplementary Service Delta Object Classes	155
Figure 6.1: Potential Rendezvous Deadlock.....	177
Figure 6.2: Role of the Buffer and Transporter Objects.....	178
Figure 6.3: Supplementary Service Delta Object Classes Included in MLog Prototype	204

CHAPTER 1

Introduction

1.1 Background and Motivation

The specification and design of large event-driven systems is a challenging endeavour due to the complex, often time-dependent, relationships between inputs or events and outputs. The specification problem is rooted in the difficulty of developing intuitive and comprehensive descriptions which are also formal and unambiguous so as to facilitate computerized validation and simulation [HARE 87].

Telecommunication systems are an example of complex event-driven systems. A number of different techniques have been used to specify telecommunication systems. Most notably is the Specification and Description Language (SDL) which was specifically developed by the International Telephone and Telegraph Consultative Committee (CCITT) for the specification of telephone switching systems .

The development of comprehensible and formal specifications of telecommunication systems will become an increasing critical activity as these systems evolve in complexity to support a wide range of services under the Integrated Services Digital Network (ISDN) concept. An ISDN is an all-digital network providing a multitude of services (voice, data, video) via a small set of standardized multi-purpose user-network interfaces.

ISDN supplementary services are one class of services which present a challenging specification problem. Supplementary services are those which modify a basic telecommunication service (e.g. telephony, circuit-switched data services) by adding to or changing its characteristics or behaviour. These services cannot be offered on a stand-

alone basis but must be provided in conjunction with a basic telecommunication service. These services will be offered in various combinations to subscribers.

The difficulties in specifying these services is largely attributed to their dependency on, or coupling to, a basic service. For example, the behaviour of the call waiting supplementary service cannot be described separately from the basic service but rather is described as a modification to the basic service. With the basic service, a call is rejected (i.e. a busy condition is returned to the calling subscriber) if there is no available channel to offer the incoming call to a called subscriber. With the call waiting service, the same condition (no channel available) results in a call waiting indication being provided on incoming calls to the called user and an alerting condition returned to the calling subscriber.

The specification approach actively being pursued by CCITT is to describe each ISDN supplementary service individually using a combination of natural language and SDL diagrams. This has resulted in a number of standalone dynamic descriptions (i.e. one for basic service and one for each supplementary service). There is overlap or commonality between the dynamic descriptions as the services generally derive their behaviour from the basic service.

Although the CCITT approach may be appropriate for specifying individual services, it does not address the problem of how to formally specify combinations of these services. The only guidance CCITT has provided on how services can be combined is to identify the interactions between supplementary services using natural language text.

One approach to the combination problem is to develop a specification which describes the composite behaviour of the set of services that are to be supported. This is equivalent to creating one large state machine with its associated flat structure. The behaviour of individual services are difficult to distinguish within this combined description. Maintainability also becomes a major issue. As each new service is introduced, the complex combined specification must be analyzed in order to identify the additional behaviour and/or changes to the existing behaviour which must be accommodated.

Therefore, there is a requirement for a technique which will allow these services to be specified in a structured and modular fashion recognizing potential interactions between services and the dependency of each service on the basic service. This has been the prime motivation for the work presented in this thesis.

1.2 Thesis Objectives

The prime objective of this thesis is to describe an "object-oriented delta" approach and analyze its applicability to the telecommunication service specification problem.

In pursuit of this objective, the sub-objectives of the study presented in this thesis are:

1. To specify the basic telecommunication service in terms of a number of communicating objects, each object exhibiting a particular behaviour.
2. To characterize the impact of introducing supplementary services into the basic telecommunication service model in terms of differential behaviour. This differential or "delta" behaviour expresses the additions and/or modifications to the basic service behaviour.
3. To explore alternatives for the expression of this delta behaviour. The concept of inheritance within the object-oriented paradigm is investigated, by means of examples, as an approach to accommodating this delta behaviour. With this approach, a single object forms the supplementary service via inheritance of the basic service characteristics. The ability to encapsulate this delta behaviour in a separate object which communicates with the basic service object to form the supplementary service is also explored.
4. To investigate, by means of examples, whether the delta behaviour associated with the individual supplementary services can be combined in a non-conflicting manner to form the combined service. That is, the combined service behaviour can be expressed as a simple function (e.g. union) of the individual deltas.

5. To specify and validate the basic telecommunication service and a number of supplementary services using an object-oriented language, MLog, which was specifically developed for the rapid prototyping of systems [KARA 89a].
6. To analyze the ability of an object-oriented language, specifically MLog, to express the concepts of the object-oriented delta specification approach.

1.3 Thesis Contributions

The specification approach presented in this thesis provides a compact and structured way for defining services which are closely coupled to the basic service. Each new service does not have to be specified in its entirety, resulting in a number of specifications with overlapping behaviour. Only the additions and/or modifications to the basic service introduced by the new service are specified.

By specifying this delta behaviour (i.e. additions and/or modifications to the basic service), the analyst can focus on a less complex service combination problem. That is, the overall behaviour of one supplementary service does not have to be analyzed against the overall behaviour of another supplementary service to determine the resulting combined behaviour. Only the differences between the two services need to be analyzed as the common behaviour of the two services has been factored out.

The specification approach also addresses to some extent the problem of how to combine services without requiring all possible combinations of services to be specified. It is shown that some services can be combined as a simple union of their individual "delta specification components". Other combinations require the specification of additional "combined delta specification" components. However, these additional combined specification components represent a more compact and formal expression of combined service specifications.

Other contributions are in the areas of:

1. Providing visual techniques to express the "delta behaviour" or partial state machine descriptions.
2. Representing state machines as logic programming rules in such a way to be amenable to partial inheritance of behaviour.
3. Providing some general rules for mapping the visual techniques for describing behaviour to logic programming rules.

It is considered that one of the main contributions of this thesis is the many interesting and open issues it has raised. The further development and formalization of this approach may form the basis of future research.

1.4 Thesis Organization

Chapter 2 provides an overview of telecommunication services and their specification approaches.

Chapter 3 develops a high-level object model for the basic telecommunication service. Within this model, the interactions between the objects are identified and the temporal behaviour of the objects is expressed using finite state machines. This model provides the basis for analyzing the impact of introducing new services.

Chapter 4 provides an overview of the "object-oriented delta specification" approach. It introduces the concept of "delta specification components" as the basic building blocks for forming individual and combined supplementary service specifications. It identifies alternatives for expressing the "delta specification components" within the scope of an object-oriented framework. The way in which the "object-oriented delta specification" approach addresses the service combination problem is also discussed by means of a number of examples.

Chapter 5 applies the "object-oriented delta specification" approach developed in Chapter 4 to the specification of a number of supplementary services and their combinations.

In Chapter 6, the concepts of the object-oriented, delta specification approach are explored using the MLog programming language. The basic telecommunication service object model of Chapter 3 and the supplementary service delta specifications of Chapter 5 are translated into an executable specification: the MLog Telecommunication Service Prototype. The issues in mapping the service specifications to the MLog prototype are discussed.

Chapter 7 provides the summary and conclusions of the thesis.

CHAPTER 2

Overview of Telecommunication Services and Their Specification Approaches

This chapter provides an overview of telecommunication services and the various techniques which have been used to specify the telecommunication systems which support them.

Sections 2.1 to 2.4 introduce some basic telecommunication service concepts and terminology. The various telephone systems as well as the signalling and transmission components which constitute the current and planned telephone network are identified. The interactions or communications between various telephone systems to fulfil a subscriber's request for a telecommunication service are outlined

Section 2.5 provides an overview of the various specification techniques used for telecommunication applications. Section 2.6 identifies how the SDL technique is being used within a three-step method within CCITT for the specification of telecommunication services.

Section 2.7 outlines the approach taken by this study to telecommunication service specification.

2.1 Aspects of Telecommunication Service Provision

Network service providers in offering telecommunication services to their subscribers are concerned with the following:

1. Providing or allocating network resources, on demand, in response to subscriber requests for telecommunication services.

- 2 **Operating, administrating and maintaining telecommunication networks and services.** This includes billing the subscriber for the service; provisioning or engineering resources in response to customer service orders and long-term traffic forecasts; maintaining the integrity of the resources to ensure an overall quality of service for the subscriber.

The first aspect is the primary focus of this study. However, interactions between the dynamic behaviour of the service and the administration, operational and maintenance aspects are considered in this study. For example, different entities (e.g. processes) may simultaneously attempt to seize a resource: one for dynamically supporting a service request and another for maintenance purposes.

The next section identifies the telephone systems and the network transmission and signalling capabilities which are involved in the dynamic provision of POTS (Plain Old Telephone Service) to subscribers. Section 2.3 then presents the reference configuration (i.e. interconnection of telecommunication systems and subscriber equipment) assumed as the basis of this study.

2.2 Overview of the Current Telephone Network

Figure 2.1 illustrates the various telecommunication systems which are involved in the dynamic provision of POTS (Plain Old Telephone Service) [BELL 82]. Subscribers are connected to a local switching office or exchange via an analog *subscriber loop* or *subscriber access* supporting the transmission of voice and voiceband data on a 4 kHz channel. User to network signalling (i.e. on-hook and off-hook signals, dialed digits) is carried inband (i.e. over the same channel as the user information). This arrangement allows the provision of telephone (voice) service and limited data services, such as telex and facsimile between modem-equipped terminals at each end.

Transit exchanges are telecommunication systems which only support interfaces to other telecommunication systems (i.e. they do not support subscriber interfaces). One or more transit exchanges may be involved in the provision of an end-to-end communication path

Figure 2.1: Reference Configuration for the Current Telephone Network

between two subscribers. A request from a subscriber to establish a communication path to another subscriber served by the same local exchange (e.g. in Figure 2.1 between subscribers s_{11} and s_{1n}) does not require the involvement of transit exchanges.

The interface between telephone exchanges includes both a signalling component to convey supervisory and address information, and a transmission component to convey user information (voice and data). A mixture of digital and analog transmission facilities are currently supported within most telephone networks. Current signalling systems are predominantly inband. That is, signalling information is carried over the same channel as the user information.

2.3 Reference Configuration

The current analog subscriber access and inband network signalling systems limit the range of services that can be provided. The telephone network is evolving towards an Integrated Services Digital Network (ISDN) [TANE 88, MILL 88] by the gradual incorporation of additional capabilities (e.g. digital subscriber access, out-of-band signalling systems). The ISDN is an all-digital network which supports a wide range of services and is accessible to the user through a common interface.

In the ISDN, additional components will be deployed within the network (e.g. signalling points, distributed database systems) to support the range of service requirements.

This study assumes a reference configuration in which subscribers are connected to a local exchange via an ISDN digital interface and common channel signalling is used between exchanges (see Figure 2.2).

Figure 2.2: Subscriber and Network Interfaces Assumed in Study

2.3.1 Subscriber Access

For the purpose of this study, a digital ISDN subscriber interface is assumed which supports more than one bearer channel, and a powerful and flexible signalling system. The bearer channels (called B-channels) are used to convey user information (e.g. voice and data). The signalling procedures or protocols for controlling the bearer channels are conveyed over a separate signalling channel (called the D-channel). The ISDN digital subscriber access protocol [CCITT 88b], which is a message-based protocol, has been used as the basis for describing the interactions between the subscriber and the network service provider.

The subscriber access may support a single terminal equipment configuration, as well as an arrangement supporting multiple terminal equipment on a passive bus as shown in Figure

2.2. Further, the subscriber access can support multiple simultaneous transactions. A transaction, in this context, refers to a set of interactions pertaining to a single objective. Transactions may be classified as follows:

- 1. Bearer associated transactions:** which are the set of interactions corresponding to the establishment and release of a *bearer connection*.
- 2 Non-bearer associated transactions:** which do not involve the establishment of bearer connection. An example of a non-bearer associated transaction is a request to activate the call forwarding supplementary service.

With current subscriber access arrangements, only one transaction is active at a time. That is, a subscriber currently engaged in a telephone conversation with another user, cannot originate another call or activate a service over the same subscriber access.

2.3.2 Network Signalling

For the purpose of this study, a network interface supporting a powerful and flexible signalling system is assumed. The ISDN signalling system number 7, an out-of-band message-based signalling system, has been used as the basis for describing the interactions between a pair of exchanges [APPE 86, CCITT 88c]. A single signalling interface between two exchanges can control a number of transmission channels. A network transmission channel is referred to in the subsequent discussion as a trunk or circuit.

2.4 Interactions between Telecommunication Systems

The previous section identified the telecommunication systems involved in the provision of telecommunication services and the nature of the interfaces between:

- telecommunication systems (i.e. network interfaces).

- subscriber terminal equipment and telecommunication systems (i.e. subscriber or user-network interfaces).

This section identifies the interactions across these interfaces for various service scenarios. A description of these interactions is intended to describe the telecommunication system in terms of its external observable behaviour, i.e. in terms of the interactions between the system and its environment.

Figure 2.3 identifies three ISDN basic telecommunication service scenarios as follows:

Scenario a: describes the various messages exchanged across the user-network interface for a successful service establishment between two subscribers served by the same local exchange. In this scenario, it is assumed that Subscriber A initiates the service disestablishment.

Scenario b: describes the various messages exchanged across the user-network interface for an unsuccessful service establishment (e.g. Subscriber B is busy).

Scenario c: describes the various messages exchanged across the user-network interface and the network-network interface for a successful service establishment between two subscribers served by different local exchanges. In this scenario, it is assumed that Subscriber B initiates the service disestablishment.

In these three scenarios, a subscriber initiates a request for a service by transmitting a "setup" message across the user-network interface. The "setup" message contains the identity of the subscriber with which the subscriber wishes to communicate (referred to as the destination user in the following discussion). The local exchange will return an indication to the user that it is able to process the request (i.e. "call proceeding" message). Resources to support the requested service (e.g. a bearer channel for transferring the user information) are allocated by the local exchange.

(a) Successful Service Establishment Between Subscribers Served by the same Local Exchange

(b) Unsuccessful Service Establishment

Figure 2.3: Service Scenarios

(c) Successful Service Establishment Between Subscribers Served by Different Local Exchanges

Figure 2.3 (cont.): Service Scenarios

If the destination user is served by the same local exchange (scenario a, b), the local exchange will send a "setup" message across the destination user-interface. However, if the destination user is served by another local exchange, an "initial address" message, containing the service request and destination address, is forwarded over the network-interface towards the destination local exchange. A network transmission channel or trunk will be allocated by the local exchange and its identity included in the "initial address" message.

The destination subscriber may respond to a "setup" message by transmitting an "alerting" message across the user-network interface (scenario a and c). "Alerting" indicates that the user is being alerted to the incoming service request by, for example, a terminal-provided ringing tone. An indication that user alerting has been initiated at the destination address will be returned to the originating subscriber.

The destination subscriber indicates acceptance of the requested service by returning a "connect" message across the user-network interface. An indication that a connection has been established through the network to the destination subscriber is returned to the originating user.

Once the service has been successfully established, either subscriber may initiate the disestablishment of the service by sending a disconnect message across the user-network interface. This message causes the release of resources by the "release/release complete" message exchange across the user-network interface and the "release/release complete" message exchange within the network (scenario c).

The destination subscriber may reject an incoming service request (e.g. user is busy) by returning a release complete message in response to a setup message.

2.5 Specification Techniques

This section identifies a number of techniques which have been used in the specification of telecommunication systems. The comments that follow are brief, as this thesis is not

intended to provide a comparison of specification techniques. However, each technique was considered as to its applicability to address the service specification problem under consideration.

The notion of state machines, which react to inputs changing state and generating outputs, has been advocated in [WHIT 81] as well suited for telephony applications. However, as telecommunication systems evolve in complexity to support a range of services, over and above POTS, the use of simple unstructured state machine descriptions becomes questionable. It is generally agreed that a complex system cannot be beneficially described in terms of simple state machine descriptions due to the exponentially growing multitude of states, all which must be arranged in a flat and unstructured way [HARE 87].

Finite State Machines serve as the underlying model of the *Specification and Description Language* (SDL) [CCITT 88a]. SDL was developed in the late 1970s by CCITT (International Telegraph and Telephone Consultative Committee) for the external behaviour and internal design of telephone switching systems. SDL supports both a Graphical Representation form (SDL/GR) and a textual Phrase Representation form (SDL/PR) which are equivalent from semantic points of view. The SDL/PR form has not been as widely used as the SDL/GR form. As identified in Section 2.6, the SDL/GR technique has been adopted within CCITT as part of an overall ISDN service specification method.

SDL provides for the decomposition of the behaviour of a system into a collection of *processes*. A process is an extended finite state machine, that works autonomously and concurrently with other processes [BELI 89]. A process communicates with other processes by sending and receiving messages, asynchronously, over channels. Process descriptions are represented graphically using a flow chart-like diagram notation with special symbols defined for process *start*, *state*, *input*, *output*, *decision*, *action*, etc.

There is no provision for hierarchy or depth in the process descriptions which often results in relatively simple "processes" having quite complex-looking SDL descriptions [HARE 87].

The Requirements Language Processor (RLP) [DAVI 79, DAVI 82] is part of an overall Requirements Processing System (RPS) developed by GTE Laboratories in Waltham, Massachusetts. In the RLP, the external behaviour of a telecommunication system is defined in terms of *stimulus-response sequences*. A stimulus-response sequence is essentially a trace of a two-way dialogue between the system under specification and its environment [DAVI 88]. Each sequence corresponds to a customer-oriented external system service such as a local telephone call, the call forwarding service. A system is then viewed in terms of the union of these stimulus-response sequences.

RLP focuses on the customer's perspective of the system. Each service, described as a stimulus-response sequence, is specified in isolation.¹ This particular technique does not provide much guidance on how telecommunication systems can be structured and designed to support the required services.

Zave's approach in [ZAVE 85] is to use sequence diagrams from the Jackson Structured Programming (JSP) and Jackson System Development (JSD) methods augmented with the concept of "aliases". A specification of a system (e.g. a telephone system) is a set of views, each view having a sequence diagram. A "filter" is provided which received inputs and passes each one on to all views to which it is relevant. Each view may be considered as a concurrent entity, co-ordinated to some extent by the "filter".

More recently, it was shown in [FACI 89] that the OSI (Open Systems Interconnection) formal description technique LOTOS (Language of Temporal Ordering Specifications) can be appropriately used for specifying the observable behaviour of telephone systems. In [FACI 89], a LOTOS specification was developed for POTS. It was advocated that more complex services could be specified in LOTOS using the basic mechanism described for POTS.

¹The RLP, however, does detect potential conflicts between individual service descriptions.

2.6 CCITT Service Specification Method

CCITT has adopted a three step approach to the specification of ISDN services. The first step of this method is the development of a prose description of the service as perceived by the end users or subscribers. This prose description is accompanied by a dynamic description of the service, again from the end user's perspective, which considers the network as a single entity. This dynamic description is expressed in graphical SDL.

During this first stage, the interactions between this service and other services are specified in terms of natural language text.

The second stage consists of the development of a functional model for the service. This involves:

1. The identification of the "functions" to support the service.
2. The mapping of these functions to different functional entities.
3. The expression of the interactions between functional entities in terms of information flow diagrams.

The dynamic behaviour of the functional entities is specified using graphical SDL.

The final stage is the specification of the ISDN protocols (user-network and network-network) to support the services. The ISDN protocols are developed from the information flow diagrams output from stage 2 of the method. The protocols are specified using a combination of prose and graphical SDL descriptions.

The CCITT method is primarily concerned with identifying the role of the various telecommunication systems in the provision of the service and the protocol requirements to support communications between these systems.

2.7 Study Method

Within each of the specification techniques outlined in Section 2.5 (with the exception of unstructured finite state machines) there is provision for decomposing the complex behaviour of the system. With SDL, the behaviour of the system is constituted by the combined behaviour of a number of processes in the system. With Zave's approach, the behaviour of the system is specified in terms of a set of views. The "stimulus-response" sequence for a service is the basic unit of behavioural specification in RLP.

Within this study, an "object-oriented" approach has been adopted. Within this approach the various objects (i.e. abstractions of the physical or logical entities) which interact to provide the required services, are identified. An object has state which may be affected through its interactions with other objects.

An object-oriented approach was selected based on the following considerations:

1. Fundamental to this method is the concept of refinement. Refinement provides for the recursive decomposition of objects into aggregates of other objects. This allows the system to be viewed at various levels of abstraction. Also telecommunication systems can be considered as aggregations of objects. For example, a telephone system consists of subscriber interfaces, network interfaces, switching network, etc.
2. Fundamental to object-oriented programming systems [WEGN 87] is the concept of inheritance. Inheritance allows an object class (called a subclass) to be defined which inherits characteristics or behaviour of another class (called parent or superclass). The subclass can extend or modify some of the characteristics of the parent class. Therefore, a new class can be defined by specifying the differences between it (i.e. the subclass) and an existing class. Inheritance can be exploited in the specification of supplementary services as these services extend or modify the characteristics of the basic telecommunication service.

It should be emphasized that an "object" in this study differs from the "everything is an object" approach (including "primitive" objects such as integers) supported in OOPS systems such as Smalltalk [GOLD 85]. In this study, an object represents a large-grain communicating sequential process similar to the concept of active object in MLog [KARA 89a]. Furthermore, inheritance is used sparingly in the service specifications developed in this study. That is, inheritance is used in its broadest sense to explore the feasibility of specifying supplementary services in terms of incremental or differential behaviour to the basic telecommunication service. The use of inheritance is not directed at developing highly reusable telecommunication service specification components.

In the following chapter, a high-level object model for the basic telecommunication service is developed. Within this model, the behaviour of the objects is specified in terms of finite state machines with extensions provided by the *statechart* formalism [HARE 87]. Specifically, the statechart concepts of clustering and refinement are used to address the "state explosion" problems associated with conventional finite state machines.

Having developed a high-level object model for the basic telecommunication service, Chapter 4 looks at the problems of specifying individual and combined supplementary services. Approaches to accommodate supplementary services within the scope of an object-oriented approach are introduced. In one approach, the possibility of "building" the supplementary service specification from the basic service object model by introducing new objects is considered. In another approach, the concept of inheritance is used in the specification of supplementary services. These approaches are explored by means of a number of supplementary service examples in Chapter 5.

In Chapter 6, the basic telecommunication service and a number of supplementary services are specified and validated using an object-oriented language, MLog. The development of this "telecommunication service" prototype allowed the following to be examined:

- The run-time behaviour of the specification.
- The ability of current object-oriented programming language to express the service specification approaches identified in Chapter 4.

CHAPTER 3

Object Model for the Basic Telecommunication Service

This chapter develops a basic telecommunication service model based on the object-oriented paradigm. Specifically, the concept of "objects" is used for describing the architectural structure of the basic telecommunication service specification.

Section 3.1 provides a prose definition of the basic telecommunication service. Section 3.2 introduces the concepts and notations used in development of the object model.

The objects which interact to support the basic telecommunication service are then identified in Section 3.3. A potential mapping of objects to telecommunication systems is discussed in Section 3.4. The interactions between these objects are described in Section 3.5. Section 3.6 specifies the temporal behaviour of objects. A summary of the chapter is provided in Section 3.7.

3.1 Definition of the Basic Telecommunication Service

The "basic telecommunication service" is defined in this study as one which provides for the establishment, supervision and release of circuit-switched "bearer" connections, as well as for the transfer of information over the connection.

Upon receipt of a request for the basic telecommunication service from a subscriber (subsequently referred to as the originating user), the network service provider establishes an end-to-end circuit switched connection to the destination or responding user. Once the connection is established, user information (e.g voice, data) can be exchanged between the connected users. Either the originating or responding user can request the

disestablishment of the service. Upon receipt of such a request, the network service provider releases the circuit-switched path between the two parties.

An association is a temporary relationship which is established between two entities to support a telecommunication service. With the basic telecommunication service, an association and a "circuit-switched" bearer connection (i.e. capable of transporting the user information) are simultaneously established and released. An *outgoing association* is established between an originating user and the network service provider. An *incoming association* is established between the network service provider and the responding user. That is, outgoing and incoming refer to the directions as perceived by the subscriber.

In the description of the current Plain Old Telephone Service (POTS), a "telephone call", or simply "call" is used to denote the temporary relationship between two subscribers. The originating user is referred to as the "calling party" and the responding user as the "called party". Furthermore, the originating user places an "outgoing call" to another user and the responding user receives an "incoming call". Because "association" is a more general term than "call" it has been adopted in this study.

3.2 Overview of Object Model

3.2.1 Concepts

The main concepts of the basic telecommunication service model to be presented here are named "objects", "channels", "decomposition" and "extended finite state machines". The concepts of "objects", "channels", and "decomposition" are used for describing the architectural structure of the specification. An "extended finite state machine" is used to describe the temporal behaviour of a given "object" in the specification.

In the model presented in this chapter, an *object* represents an autonomous and concurrent entity which co-operates with other objects via message passing. An object has state which is affected by its interactions with other objects. Finally, an object is actually an instance of

a *class*. A class is a "template", a static declaration or *description* of the behaviour of instances of that class.

As identified in Chapter 2, the concept of object in this study differs from an object in Smalltalk. That is, an object in this study is similar to the concept of an active object in MLog or a task in Ada [BUHR 84].

Objects are interconnected by *channels*. A channel specifies an abstract method of describing interactions between objects which is independent of whether these interactions take place locally (between two objects within a system) or remotely.

Object classes are represented graphically by boxes marked with the name of the object. A channel is represented graphically by a line connecting two object classes. The line has an arrowhead indicating the direction of the message flow and is marked with the name of the channel. A channel which is bidirectional is represented by a line with two arrowheads.

Figure 3.1 provides an example of the use of the notation for describing objects and their communication relationships. In this Figure, an instance of the object class "O1" sends messages to an instance of the object class "O2" over the channel "C1" and communicates, in both directions, with an instance of the object class "O3" over the channel "C2". Note that the notation does not identify the number of instances in the model and the corresponding "instance relationships". That is, Figure 3.1 does not indicate whether several instances of object "O1" communicate with only one instance of object "O3", or whether only one instance of object "O1" communicates with only one instance of object "O3".

In the model, it is assumed that objects can be decomposed into other objects to provide different levels of abstraction. This is shown graphically by including smaller boxes ("sub-objects") within a larger box (object).

"Sub-objects" within an "object" may also interact. The concept of channels is also used to denote the communication relationship between sub-objects.

Figure 3.1: An Example Showing Objects and Their Interactions

The messages conveyed over the channel external to the larger box (object) may be applicable to different "sub-objects" within the box. To express the relationship between messages conveyed over the "external" channel and the relevant sub-object, the concept of "message route" has been introduced. The message routes connect the sub-objects of the larger box to the external channel of the larger box.

Figure 3.2 provides an example of the notations used to express object decomposition. In this Figure, Object O1 from Figure 3.1 has been decomposed into two sub-objects: O11 and O12. These sub-object communicate with one another over the channel C11. Messages received over channel C2 are "sent" over the message route "R2" or "R3". Messages sent by sub-object O11 over the route "R2" are conveyed over the channel "C2" to the object O3.

The notations provided in this section are used in the following section for identifying the objects and the "channels" which constitute the basic telecommunication service model. Section 3.5 provides the channel descriptions: the messages which are conveyed over each channel, the direction of the message flow and the identification of the endpoints (i.e. objects) of the channel. Section 3.6 provides the behavioural description of each object in terms of extended finite state machines.

Figure 3.2: An Example Showing Object Decomposition

3.2.2 Comparison with SDL

The objects in this study are modelled in SDL by SDL blocks and processes. SDL blocks, graphically represented by rectangles, define the "static" structure; whereas SDL processes, graphically represented by rectangles with clipped corners, specify the temporal behaviour. SDL blocks can be partitioned into (sub)blocks. Repeated block partitioning results in a block tree structure. Leaf-blocks of the block tree structure are not partitioned and contain only SDL processes.

In this study, the static structure and temporal behaviour are not explicitly represented, as in SDL, by two different concepts. A single concept, object, provides the means for decomposing the specification as well as the basis for describing the temporal behaviour. This allows the service to be specified at various levels of abstraction.

For example, in the following section, the "network service provider" object constitutes the top level of detail; that is, the specification of the service as perceived by the subscriber.

Thus the behaviour of the service, as viewed by the subscriber, can be modelled as a single extended finite state machine. Whereas, this description provides the subscriber with a general understanding of the operation of the service, it does not specify how the service is supported in a distributed environment involving a number of different telecommunication systems. Therefore, the "network service provider" object is further decomposed into a number of objects which are then mapped onto telecommunication systems as described in Section 3.4.

The SDL concept of channel and signal routes differs from the concept of channels and message routes identified in this study as follows:

1. An SDL channel interconnects SDL blocks and an SDL block to its environment. SDL channels *do not* interconnect SDL processes; SDL processes are connected by signal routes (see 2 below). A channel, as defined in this study, connects objects *or* sub-objects.
2. An SDL signal route interconnects SDL processes and a process to the boundary of the block. A message route, as defined in this study, connects the sub-object of the larger object to the external channel of the larger object. Therefore, a message route is equivalent to the concept of a SDL signal route when it (i.e. the SDL signal route) applies to the interconnection of a SDL process to the boundary of the block.

The SDL notation for signal routes (line with arrows at the ends) has been used for representing channels and message routes in this study.

3.3 Basic Service Objects and Their Communication Relationships

The object model for the basic telecommunication service was developed using an approach of successive decomposition. The highest level view identifies subscriber objects as users of the telecommunication services provided by the network service provider object (see Figure 3.3). The subscriber's view of the service is defined by the interactions or messages which are conveyed between the subscriber and the network service provider

Figure 3.3: Top Level Basic Service Object View

over the "subscriber access" channel. The list of messages that can be transported by the "subscriber access" channel is provided in Section 3.5.

Note that the messages conveyed over the "subscriber access" channel are for the purpose of *controlling the service*; that is, for establishing and releasing a circuit-switched connection to another subscriber and *not* for the actual transport of user information over this physical connection.

As indicated in Section 3.2.1, the notation for describing the communication relationships between objects does not specify the "instance relationships". With regards to Figure 3.3, there would be several instances of the object "subscriber" communicating with a "network service provider" object instance. To support a particular basic service request, an instance of a subscriber class (referred to as the originating subscriber) would communicate with an instance of a "network service provider" class which in turn would communicate with an instance of a subscriber class referred to as the responding subscriber.

This highest level view corresponds to the first step of the CCITT service specification method for ISDN services (see Section 2.6). That is, the service is first described from the perspective of the end user or subscriber and considers the network *as a single entity*.

In the following section, the **network service provider** object is decomposed into a number of sub-objects which interact to provide the service as perceived by the **subscriber** object. The specification of the **subscriber** object is beyond the scope of this study.

3.3.1 Network Service Provider Object

The "Network Service Provider" object can be decomposed into a number of objects as shown in Figure 3.4. The "Subscriber Access Control", "Association Control" and "Resource Control" objects have been shaded in this Figure to signify that these objects are the primary focus of this study.

Note that Figure 3.4 is intended to show the communication relationships among objects and not how instances of these objects may be mapped onto various telecommunication systems (see Section 3.4 for a discussion of a potential mapping of objects to telecommunication systems). Two network interface control objects are shown in Figure 3.4 to indicate that an instance of this class may communicate with another instance of the same class. This is not true for the other objects identified in Figure 3.4. That is, an instance of a "subscriber access control" object does not communicate with another "subscriber access control" object.

The messages conveyed over the channels identified in Figure 3.4 are for the purpose of controlling the service.

The following provides a description of each of the "Network Service Provider" sub-objects:

Subscriber Access Control: controls the overall service requests across the subscriber access. This object essentially models the control functions associated with a subscriber access. There is one object instance defined for each subscriber access supported.

Figure 3.4: Decomposition of the Network Service Provider Object

Association Control: establishes, maintains and releases associations between two objects. It is responsible for the routing and relaying functions, as well as co-ordinating the switching functions.

Network Interface Control: controls the overall service requests across the network interface. It is responsible for allocating and deallocating a network circuit or trunk to an association. It specifies the network signalling messages and procedures required for the establishment and control of circuit-switched connections.

Management: is responsible for system management functions, including billing, maintenance, provisioning services and resources, etc.

Resource Control: is responsible for co-ordinating access to the various telecommunication resources. Telecommunication resources are the "physical" and "logical" entities used to support telecommunication services. For example, a bearer channel supported on the subscriber access is an example of a telecommunication resource. A "switched path" between two bearer channels within an exchange is also an example of a telecommunication resource. A number of resources may be dynamically allocated/deallocated to an association during its lifetime. Furthermore, a management object may access a resource for the maintenance purposes.

The decomposition of the "Network Service Provider" object into the sub-objects described above is based on the ISDN Q.931 protocol model [CCITT 88b] which identifies:

- a "subscriber protocol" block (responsible for communication with the subscribers),
- a "call control" block (responsible for interworking between the subscriber protocol block and the network protocol block),

- the "network protocol" block (responsible for communications between exchanges), and
- a "resource management" block (provides access to subscriber B-channels).

A management object, not identified in [CCITT 88b] and which is not directly involved in the "dynamic provision" of the service has been included in this study for completeness. As identified in Section 2.1, the operation, administration and maintenance of telecommunication networks and services is an important aspect that network service providers must consider in providing services to their subscribers.

A similar partitioning can also be recognized (with the exception of management) in the LOTOS specification of [FACI 89].

3.3.2 Subscriber Access Control Object

The "subscriber access control" object can be further decomposed into the following sub-objects (see Figure 3.5). The interaction with the management object is not specified as it is not directly relevant to this study.

Transaction Control: controls and co-ordinates the various types of transactions across the subscriber access. In the basic telecommunication service description, only one type of transaction is supported, namely: bearer associated transactions. As indicated in Section 2.3.1, a bearer-associated transaction encompasses the set of interactions between a subscriber and the network service provider for the purpose of supporting a bearer service (e.g. a circuit-switched connection between two subscribers).

The "transaction control" object controls the number of bearer associated transactions that are simultaneously active across the subscriber interface. It dynamically creates an "outgoing association" object when it receives a basic service request from a subscriber (see below). It dynamically creates an "incoming association" object when it wants to send a basic service request to a subscriber (see

Figure 3.5: Decomposition of the Subscriber Access Control Object

below). The "outgoing association" and "incoming association" objects are then responsible for handling all subsequent interactions with the subscriber associated with the basic service request attempt. Recall from Section 3.1 that "outgoing" and "incoming" are defined from the subscriber's perspective.

It should be noted that there is no requirement for sub-objects to be dynamically created. The decomposition of the "subscriber access control" object into a persistent transaction control sub-object (i.e. created at system initialization time and persists until explicitly destroyed), and dynamically created outgoing and incoming association sub-object instances was a specification decision considered to contribute significantly to the understanding of the service.

Outgoing Association: is responsible for interacting with the subscriber for the purpose of establishing, controlling and terminating an outgoing association. The "outgoing association" object is responsible for the allocation/deallocation of a bearer channel to the outgoing association. It is also responsible for collecting the complete destination address in the case where the complete information is not provided in the initial service request. The "outgoing association" object instance terminates when the outgoing association is terminated.

Incoming Association: is responsible for interacting with the subscriber for the purpose of establishing, controlling and terminating an incoming association. It is responsible for the allocation/deallocation of a bearer channel to the incoming association. It is also responsible for providing the procedures to handle multiple users responding to a service request, including the completion of the association to a selected user. The "incoming association" object instance terminates when the incoming association is terminated.

For the basic telecommunication service, the behaviour of the "incoming" and "outgoing" association objects differs primarily during the establishment phase of the association. For example, the "outgoing association" supports procedures to collect the complete address when the subscriber does not provide all information in the service establishment request. However, on an "incoming association" the complete address information is provided in the service request.

Furthermore, the subscriber access may support multiple users on a passive bus arrangement (see Section 2.3.1). Therefore, the "incoming association" object must provide procedures to handle multiple replies to a service request. For an outgoing association, there is only one user originating the service request.

3.3.3 Association Control Object

The association control object can be decomposed into the following sub-objects (see Figure 3.6):

Figure 3.6: Decomposition of the Association Control Object

AC Creator: dynamically creates a bearer association control object instance when a setup Indication is received from an invoking object (e.g. "outgoing association" object). The bearer association control object instance handles all subsequent messages associated with a particular association attempt.

Bearer Association Control: establishes, maintains and releases an association between two objects (e.g. between an outgoing association and incoming association object) for the purpose of providing a bearer service. It is responsible for the routing and relaying functions, as well as co-ordinating the connection between two bearer channels. The "bearer association control" object instance terminates when the association is successfully released.

3.3.4 Other Objects

Each of the remaining network service provider sub-objects (i.e. network interface control, resource control, management) could in principal be further refined into sub-objects. For example, the management object could potentially be refined into provisioning, maintenance, accounting, etc. sub-objects.

A similar structure to that of the subscriber access control object could be explored for the network interface control object. That is, an overall co-ordinating object (similar to the transaction control object) could be defined for the management of transactions across the network interface. Objects analogous to the subscriber access control's "outgoing association" and "incoming association" sub-objects could be specified for managing service requests destined to/received from a peer network interface control object, respectively.

CCITT is currently investigating the structure of what has been denoted as the "network interface control" object in this study. The architecture which is evolving within CCITT is the definition of "two sub-objects" or SDL blocks: one block providing a set of functions for association handling; whereas the other supports a set of functions for the control of bearer connections. The separation of bearer connection control from association control is seen to provide a more flexible arrangement between an association and its corresponding bearer connections. This flexibility is perceived as essential to support future multi-media services involving multiple bearer connections. It will also support the independent development of bearer connection control procedures for new broadband bearer services.

The focus on the "subscriber access control" and the "association control" objects is sufficient for meeting the objectives of this study: to develop a modular approach to the specification of a group of closely interrelated services (i.e. supplementary services). As will be shown in Chapter 5, these services largely affect the subscriber access control object.

The specification of the "network interface control" object, taking into consideration new complex multi-media and broadband services, may constitute an interesting investigation in its own right.

3.4 Mapping Onto Telecommunication Systems

Figure 3.7 identifies a potential mapping of the network service provider sub-objects to the telecommunication systems identified in the reference configuration of Figure 2.1. Note that the arrow heads on the channels have been dropped solely with the intent of making the diagrams more compact.

The local exchange contains the following object classes:

- subscriber access control
- network interface control
- association control
- management
- resource control

The transit exchange contains the following objects:

- network interface control
- association control
- management
- resource control

Figure 3.7: Mapping of Object Model to Telecommunication Systems

The transit exchange does not contain the subscriber access control object as it does not terminate any subscriber interfaces.

The local exchange supports two types of associations:

- an association involving two "subscriber access object" instances. This corresponds to the scenario wherein the two subscribers are served by the same local exchange (see Figure 3.8 a).
- an association involving a "subscriber access object" instance and a "network interface control" object instance. This corresponds to the scenario wherein the two

subscribers are served by different local exchanges. In this scenario, an end-to-end association between two subscriber access objects can logically be considered as consisting of a number of intermediate associations (see Figure 3.8 b).

Only one type of association is supported by the transit exchange, namely: an association involving "network interface" objects. This type of association is in support of an end-to-end association between two subscriber access object instances (see Figure 3.8 c).

(a) Association between two Subscriber Access Control Objects

Figure 3.8: Association Types

It should be noted that the interactions between the "network interface control" objects corresponds to "remote interactions" (i.e. interactions between two telecommunication systems). Remote interactions are the subject of standardization by various committees such as the CCITT. Local interactions (i.e. interactions between objects within the same telecommunication system) are not subject to standardization. Depending on the physical architecture of the telecommunication system and the mapping of objects onto this physical architecture, local interactions may involve communications between processes on the same or different processors.

Figure 3.8 (cont.): Association Types

3.5. Interactions between Objects

This section specifies the interactions between objects in terms of the messages which are conveyed over the channels.

3.5.1 Channel Definition

A "channel" definition specifies a set of interactions, called messages, which may be sent between two objects that are connected by this channel. The channel definition has the following form:

```

channel channelName (object1,object2)
    by object1
        message1
        message2
        message3
    by object2
        message1
        message4

```

The "by" construct identifies the list of messages (e.g. message1, message2, message3) which may be sent by one object to the other object. The channel definition notation adopted in this study is based on the Estelle formal description technique developed by the International Standards Organization (ISO) [ISO 87].

Each message can contain a list of parameters. However, the parameters are omitted in the channel definitions of the following sections as they do not affect the present study.

A message route describes the subset of messages sent or received over a channel which are applicable to a sub-object. A message route definition has a similar form to that of a channel definition:

```

messageRoute messageRouteName (object1,subObject2)
  by object1
    message1
    message2
  by subObject2
    message1

```

Assuming that "subObject2" is a sub-object within the object "object2", then the above definition indicates that the messages, "message1 and message 2" are sent by object1 to "subObject2".

3.5.2 Subscriber Access Channel

The *subscriber access* channel describes the interactions between the "subscriber" and the "network service provider" objects (see Figure 3.3) and is defined as follows:

```

channel subscriberAccess (subscriber,network service provider)
  by subscriber
    setup          ----- conveys service establishment request
    alerting        ----- conveys indication that user is being
                        ----- alerted (e.g tone is being applied)
    connect         ----- conveys service establishment
                        ----- confirmation
    disconnect      ----- conveys service disestablishment
                        ----- request
    release         ----- conveys release resources request
    release complete ----- conveys release resources confirmation
  by network service provider
    setup
    association proceeding ----- conveys service establishment
                        ----- proceeding
    progress        ----- provides information concerning the

```

----- progress of the service establishment

alerting
connect
disconnect
release
release complete

Each of the above interactions can be mapped into a message defined within the ISDN user-network protocol. For example, the "setup", "disconnect" and "release" operations map into corresponding messages within the ISDN protocol.

The "setup" interaction is invoked in order to request a particular service. It causes the simultaneous establishment of an "association" and a "bearer connection". The "setup" message contains a unique transaction identifier to distinguish it from other transactions. This "transaction identifier" is called a "call reference" in the ISDN user-network protocol.

3.5.3 Subscriber Association Control Channel

The *subscriber association control* channel describes the interactions between the "subscriber access control" object and the "association control" object (see Figure 3.4) and is defined as follows:

channel subscriberAssociationControl

(subscriber access control, association control)

by subscriber access control

setupIndication	----- conveys association ----- establishment indication
objectId	----- conveys identifier of object to ----- which further communication ----- should be directed
setupConfirm	----- conveys association ----- establishment confirmation

report	----- report information concerning
	----- the progress of the service
	----- establishment/association
rejectIndication	----- reject association establishment
releaseIndication	----- release association indication
releaseConfirm	----- release association confirmation
by association control	
objectId	----- conveys identifier of object to
	----- which further communication
	----- should be directed
setupRequest	----- conveys association
	----- establishment request
setupResponse	----- conveys association
	----- establishment response
report	
releaseRequest	----- release association request
releaseResponse	----- release association response

The *network association control* channel would support a similar set of interactions between the "association control" object and the "network interface control" object.

3.5.4 Subscriber Resource Channel

The *subscriber resource* channel describes the interactions between the "subscriber access control" object and the "resource control" object (see Figure 3.4) and is defined as follows:

channel subscriber resource (subscriber access control, resource control)	
by subscriber access control	
seizeBchannel	----- conveys request that a
	----- bearer channel be
	----- allocated to an

	----- association
releaseChannel	
by resource control	
noChannel	---- indicates that no channel
	---- is available for the
	---- association
channelAvailable	---- indicates that a channel
	----- is available and provides
	----- the identity of channel
	---- which has been allocated
	---- to the association
reportChannelFailure	---- indicates that the channel
	---- has failed and can no
	---- longer support the service

The *network resource* channel would support a similar set of interactions between the "network interface control" object and the "resource control" object, the only difference being that a network trunk or circuit is seized/released as opposed to a subscriber bearer channel.

3.5.5 Association Resource Channel

The *association resource* channel describes the interactions between the "association control" object and the "resource control" object (see Figure 3.4) and is defined as follows:

channel association resource (association control, resource control)	
by association control	
connectChannels	----- conveys request that a
	----- switched path be
	----- established between two
	----- channels

releaseConnection	----- conveys request to
	----- release switched path
	----- between two channels
by resource control	
channelsConnected	----- indicates switched path
	----- established
connectionFailed	----- indicates switched path
	----- failed

3.5.6 Other Channels Between Network Service Provider Sub-Objects

Each of the remaining channels identified in Figure 3.4 (i.e. network interface, subscriber object management, association object management, network object management, resource management) could in principal be defined in a similar manner. For example, the "network interface" channel would specify the interactions between two "network interface control" objects residing in different telecommunication systems. These interactions are governed by the messages defined by the CCITT for the Signalling System No. 7 ISDN user part protocol.

The interactions between the management object and the "subscriber access control" (SAC), "association control" (AC) and "network interface control" (NIC) objects are similar in that the object to be managed (i.e. SAC, AC, NIC) reports events (e.g. failure conditions, successful service establishment) to the "management" object. These events may be used by the management object to support the billing function and to effect the maintenance functions.

The interactions between the "management" object and the "resource control" object (i.e. defined by the "resource management " channel) include the following:

- The resource control object reports error and failure conditions to the management object.

- The management object instructs the resource control object to define/remove a specified resource.
- The management object requests that a resource be allocated for maintenance purposes.

The further specification of these channels is not of direct relevance to this study.

3.5.7 Channels Between Subscriber Access Control Sub-Objects

In Figure 3.5, two channels are defined between the "subscriber access control" (SAC) sub-objects, namely SAC C1 and SAC C2. The SAC C1 channel describes the interactions between the "transaction control" object and the "incoming association" object and is defined as follows:

channel SAC C1 (transaction control, incoming association)

by transaction control	
setupRequest	----- conveys incoming ----- association request
by incoming association	
transaction complete	----- indicates that the ----- transaction is complete ----- (i.e. the incoming ----- association is ----- terminated)

The SAC C2 channel describes the interactions between the "transaction control" object and the "outgoing association" object and is defined as follows:

channel SAC C2 (transaction control, outgoing association)

by transaction control	
setup	----- conveys outgoing ----- association request

by outgoing association
 transaction complete ----- indicates that the
 ----- transaction is complete
 ----- (i.e. the outgoing
 ----- association is
 ----- terminated)

3.5.8 Channels Between Association Control Sub-Objects

In Figure 3.6, a channel is defined between the "association control" (AC) sub-objects, namely the AC C1 channel. The AC C1 channel describes the interactions between the "association control creator" object and the "bearer association control" object and is defined as follows:

channel AC C1 (association control creator, bearer association control)
 by association control creator
 setupIndication ----- conveys association
 ----- establishment indication

3.5.9 Subscriber Association Control Message Routes

Figure 3.5 identifies a number of message routes to specify the interactions between the subscriber access control (SAC) sub-objects (i.e. "transaction control", "incoming association", "outgoing association" objects) and the "association control", "resource control" and "subscriber" objects.

The interactions between the association control object and the SAC sub-objects is defined by the three message routes: SAC R1, SAC R2, SAC R3. Each of these message routes defines a subset of the messages from the "subscriber association control" channel (see Section 3.5.3) which are relevant to the particular SAC sub-object.

SAC R1 identifies the interactions between the association control object and the transaction control object and is defined as follows:

messageRoute SAC R1 (transaction control, association control)

by transaction control

objectId

rejectIndication

by association control

setupRequest

SAC R2 identifies the interactions between the association control object and the incoming association object and is defined as follows:

messageRoute SAC R2 (incoming association, association control)

by incoming association

setupConfirm

report

releaseIndication

releaseConfirm

by association control

releaseRequest

releaseResponse

SAC R3 identifies the interactions between the association control object and the outgoing association object and is defined as follows:

messageRoute SAC R3 (outgoing association, association control)

by outgoing association

setupIndication

releaseIndication

releaseConfirm

by association control

- objectId
- setupResponse
- report
- releaseRequest
- releaseResponse

The interactions between the "subscriber" object and the SAC sub-objects is defined by the three message routes: SAC R6, SAC R7, SAC R8. Each of these message routes defines a subset of the messages from the "subscriber access control" channel (see Section 3.5.2) which are relevant to the particular SAC sub-object.

SAC R6 identifies the interactions between the subscriber object and the transaction control object and is defined as follows:

messageRoute SAC R6 (subscriber,transaction control)
by subscriber
 setup
by transaction control
 release complete

SAC R7 identifies the interactions between the subscriber object and the incoming association object and is defined as follows:

messageRoute SAC R7 (subscriber,incoming association)
by subscriber
 alerting
 connect
 disconnect
 release
 release complete
by incoming association
 setup

disconnect
 release
 release complete

SAC R8 identifies the interactions between the subscriber object and the outgoing association object and is defined as follows:

messageRoute SAC R8 (subscriber,outgoing association)
by subscriber
 disconnect
 release
 release complete
by outgoing association
 association proceeding
 progress
 alerting
 connect
 disconnect
 release
 release complete

The interactions between the "resource control" object and the SAC sub-objects is defined by the two message routes: SAC R4 and SAC R5. These message routes are equivalent to the "subscriber resource" channel (see Section 3.5.4) in that they support the same interactions defined by the "subscriber resource" channel. Therefore, the definition of these message routes are not repeated in this section.

3.5.10 Association Control Message Routes

Figure 3.6 identifies a number of message routes to specify the interactions between the association control sub-objects (i.e. "association control creator" and "bearer association control" objects), and the "subscriber access control" and "resource control" objects.

The message route "AC R1" identifies the interactions between the "subscriber access control" object and the "association control creator" sub-object and encompasses a subset of the messages specified in the "subscriber association control" channel definition (see Section 3.5.3) The AC R1 message route is defined as follows:

```
messageRoute AC R1 (subscriber access control, association control
creator)
    by subscriber access control
        setupIndication
    by association control creator
        objectId
```

The AC R1 message route definition indicates that the "subscriber access control" object sends a "setupIndication" message to the "association control creator" object. The association control creator then returns an "objectId" message indicating the object (i.e. the bearer association control object) to which further communications should be directed.

The message route "AC R2" identifies the interactions between the "subscriber access control" object and the "bearer association control" sub-object. The AC R2 message route encompasses all messages specified in the "subscriber association control" channel definition (see Section 3.5.3) with the exception of the messages that are exchanged between the "subscriber access control" and the "association control creator" objects as identified in the "AC R1" message route definition above.

The AC R2 message route is defined as follows:

```
messageRoute AC R2 (subscriber access control, bearer association control)
    by subscriber access control
        objectId
        setupConfirm
        report
```

rejectIndication
releaseIndication
releaseConfirm
by association control
setupRequest
setupResponse
report
releaseRequest
releaseResponse

The AC R3 message route identifies the messages that are exchanged between the "resource control" object and the "bearer association control" sub-object. The definition of the AC R3 message route is equivalent to the "association route" channel (see Section 3.5.5) and hence, is not repeated in this section.

3.6 Temporal behaviour of Objects

This section specifies the temporal behaviour of the "association control" object and the "subscriber access control" sub-objects using finite-state machine descriptions. These "behavioural" descriptions form the basis for analyzing the impact of introducing a number of supplementary services on the basic telecommunication service model in Chapter 5.

The underlying object communication semantics is identified in Section 3.6.1. The notation adopted for expressing the state transitions (i.e. events, actions) is provided in Section 3.6.2. The notation is then used in Section 3.6.3 and 3.6.4 for describing the behaviour of the "subscriber access control" sub-objects and the association control sub-objects, respectively.

3.6.1 Semantics of Object Communication

In Section 3.5, the interactions between objects was expressed as a number of messages which could be invoked or sent by one or both of the objects over an abstract channel. In

this description, no assumptions were made regarding the underlying communication semantics.

In the development of the extended finite state machine descriptions for the objects, it is assumed that co-operation between objects is performed asynchronously. Thus, when an object sends a message to another object, the sender object continues to work autonomously. The target object maintains a FIFO (first-in first-out) queue of incoming messages.

3.6.2 State Diagram Notation

The behaviour of the "subscriber access control" and "association control" objects are expressed as state-transition diagrams (or state diagrams for short). State diagrams consist of states and "state transitions". State transition are generally of the form: "when event e occurs in state s_1 , if condition c is valid then perform action a and enter state s_2 ".

Conventional state diagrams are "flat" and unstructured. The statechart formalism [HARE 87] provides an extended state diagram representation for capturing depth and hierarchy. The "statechart" concepts of state clustering and refinement have been exploited in developing the state diagrams in the following sections.

A notation for expressing the state transitions has been developed with the following general form:

input[condition]/action;output

The syntax of each of the above elements is provided in the following subsections. Note that not all elements may be present in a given state transition. However, an input must be present in every transition and an element, if present, must be specified in the above order.

3.6.2.1 Input and Output Expressions

The expression for "input" is defined as follows:

Source?message(parameters)

This indicates that a message has been received from a specified source. Parameters may be optionally specified. Compound input statements are also possible:

Source?message1 or Source?message2

This indicates that message1 or message2 has been received from a specified source.

An "input" may also represent an internal object event, which is also assumed to be queued in the object's incoming FIFO queue. The only internal event defined is as follows:

?timeout(timerIdentifier)

This input indicates that a timer named "timerIdentifier" has expired. The source is implicit (i.e. is the object itself) in the timeout case and hence is omitted. In all other cases, the source must be explicitly specified.

Note that there are different ways in which to model timing constraints. The approach taken in this chapter, which is consistent with the SDL approach, is to assume that the timer mechanism is an inherent property or feature of the object. As will be discussed in Chapter 6, the MLog prototype provides a separate timer object with which other objects communicate to request timing services.

The expression for "output" is defined as follows:

Destination!message(parameters)

This indicates that a message has been sent to a specified destination. Parameters may be optionally specified. Sequential output statements are also possible:

Destination1!message1;Destination2!message2

This indicates that message1 is sent to a specified destination1 and message2 is sent to a specified destination2.

Note that for every "output" there is a corresponding "input" in the target object's state machine description.

There are a number of options for specifying the "source" and "destination" of a message. For example, in SDL, the destination can be specified:

- *explicitly* by specifying the process identifier in the output. The concept in SDL is that the process identifier is unique and is assigned during the creation of the process.
- *implicitly* by omitting the process identifier in the output. This approach can be used if the destination is uniquely defined by the specification structure. That is, there is only one process in the specification that can receive a particular message.
- *by naming the signal route or channel* on which the message should be sent.

In order to enhance the overall readability of the state diagrams, the approach taken in this study is to specify the "source" and "destination" in terms of the object class. That is, in examining a particular state transition diagram which specifies for a given transition that message *x* should be sent to object class *y*, the reader can immediately go to the state transition diagram associated with object class *y* to determine how the receipt of message *x* is handled.

In some cases, the specification of only the object class in the input or output expression causes some ambiguities in the state transition diagrams. For example, a transaction control object may send a "setup" message to multiple outgoing association object

instances. That is, the communication is one sender to many receivers. If the object class only is specified, then the description is ambiguous as to which object instance the setup message should be sent. To address this problem, the identifier of the object instance is optionally included in parenthesis following the object class in the input and output expressions as follows:

ObjectClass(objectIdentifier)?message and ObjectClass(objectIdentifier)!message

The inclusion of the objectIdentifier in the state transition diagrams has not been rigorously applied in the following sections; it was included only when viewed to assist in the understanding of the state transition diagrams.

The following abbreviations have been used in the state diagrams for the identification of "ObjectClass":

- **TC** for Transaction Control
- **OA** for Outgoing Association
- **IA** for Incoming Association
- **Resource** for Resource Control
- **ACcreator** for Association Control Creator
- **AC** for Bearer Association Control

3.6.2.2 Condition Expressions

Condition expressions may consist of a simple name representing the semantics of a condition. For example, a true condition "releaseReceived" indicates that a release message has been received by the object.

Compound conditions can be expressed as follows:

not C --- is true when condition C is not true
C1 and C2 --- is true when C1 and C2 are true

C1 or C2 --- is true when either C1 or C2 is true

3.6.2.3 Action Expressions

Action expression are defined as one of the following:

make_true(C) --- assign the value TRUE to the condition
--- C

make_false(C) --- assign the value FALSE to the condition
--- C

create(ObjectClass,OID) --- create an object instance of the class
--- ObjectClass. OID is the object identifier
--- assigned to the object instance.

delete(ObjectClass,OID) --- delete an object instance of the class
--- ObjectClass which has the object identifier
--- OID

V :- EXP --- assign the value of the expression EXP
--- to the object instance variable V. The
--- expression EXP may represent an
--- arithmetic expression (e.g. $V + 1$).

startTimer(timerIdentifier,timeUnits) --- start a timer with
--- name "timerIdentifier"
--- which will expire in
--- "timeUnits" from now

stopTimer(timerIdentifier) --- stop an active timer with
--- the name timerIdentifier

3.6.2.4 An Example of Expression Syntax

As an example of the state transition expression syntax, consider the following state transition found in the behaviour of the transaction control objects (see Figure 3.9):

```

AC(acId)?setupRequest[not maxAssoc]/
create(IncomingAssociation,OID);
numAssoc :- numAssoc + 1;
AC(acId)!objectId(OID);
IA(OID)!setupRequest(acId)

```

This transition states that if a "setupRequest" message is received from an "association control" object (with the object identifier = acId) and the maximum number of associations has not been reached, then the following actions are performed and the following outputs sent:

1. An incoming association object instance is created to handle the association request (i.e. create(IncomingAssociation,OID)).
2. The number of associations is incremented (i.e. numAssoc :- numAssoc + 1).
3. An "objectId" message is returned to the association control object instance which sent the "setupRequest" message (i.e. AC(acId)!objectId(OID)). This message provides, as a parameter, the object identifier of the "newly created" incoming association object instance to which further communication should be directed.
4. A "setupRequest" message is sent to the "newly created" incoming association object instance.

3.6.2.5 Comparison with other Notations

A notation for expressing the state transitions has been developed which has adopted *some* of the expression syntax of STATEMATE [ILOG 87]. STATEMATE is a graphical computerized tool developed by i-Logix, Inc. for the specification, analysis, design and documentation of large and complex reactive systems. Statecharts is one of the languages supported by STATEMATE.

The format for expressing event, condition (enclosed in square brackets), and action (preceded by the "/" character, multiple actions separated by ";") labels in STATEMATE has been adopted in this study. The compound condition expressions found in Section 3.6.2.2 of this study are supported by STATEMATE. Furthermore, the "make_true(C)", "make_false(C)" and "V:= EXP" action expressions identified in Section 3.6.2.3 are supported by STATEMATE. The remaining action expressions identified in Section 3.6.2.3 (e.g. "create(ObjectClass,OID)") were developed specifically for this study. It should be noted that the symbol "!" is used in abbreviated STATEMATE action expressions. For example, the abbreviated "make_true(C)" action expression is represented in STATEMATE as "tr!(C)". The symbol "!" has been assigned a different meaning within the scope of this study as described in Section 3.6.2.1 and below.

The major departure from the STATEMATE expression syntax for state transitions is in the description of communication between "components" or objects. Statecharts employ a *broadcast* mechanism. One component generates an event and all other components sense it, acting in response if so specified. In this study, a message is sent to a specified target object. The channel description specifies the messages which may be communicated between two objects.

The symbols "?" and "!" are used in this study for expressing "inputs" and "outputs", respectively. This convention finds its roots in the CSP (communicating sequential processes) language proposed by Hoare [HOAR 78] for the specification of inter-processor communication. However, the identification of the sender/recipient of the

message using the "ObjectClass" together with an optional identifier of the object instance was developed specifically for this study.

3.6.3 Subscriber Access Control

This section describes the temporal behaviour of the SAC sub-objects (i.e. transaction control, outgoing association and incoming association) in terms of extended finite state machines. The state machine descriptions of the outgoing association and incoming association sub-objects are based on the ISDN Q.931 SDL descriptions [CCITT 88b]. The departure from the ISDN Q.931 SDL descriptions of [CCITT 88b] include the following:

1. It is assumed that the complete destination address is provided in the subscriber service request and hence, the messages and states identified in [CCITT 88b] associated with "overlap sending" (destination address is provided in several messages) has not been explicitly represented.
2. In [CCITT 88b], the interaction with the equivalent "resource control" object is assumed to be via procedure call. Hence, a number of states included in this study associated with the interaction with an autonomous resource control object are not found in [CCITT 88b].
3. Unexpected messages (messages not expected in a given state) and failure conditions (e.g. failure of the signalling channel) identified in [CCITT 88b] have not been represented in the state diagrams of this study.

The state machine description for the transaction control object was developed specifically for this study.

3.6.3.1 Transaction Control

The behaviour of the transaction control object can be expressed in terms of a single state with several state transitions (see Figure 3.9).

AC = Association Control
 IA = Incoming Association
 OA = Outgoing Association
 numAssoc = number of associations
 maxAssoc = maximum allowed number of associations
 OID = object Identifier
 acId = Association Control object identifier

Figure 3.9: Transaction Control State Diagram

Within the "wait for request" state, the transaction control object can receive the following "messages":

- A "setup" message from the "subscriber" object. There is a one-to-one correspondence between a given subscriber object instance and a transaction control object instance.
- A "setupRequest" message from an "association control" object. Many association control object instances may send this message to a given transaction control object instance.
- A "transactionComplete" message from either the "incoming association" or "outgoing association" objects. Several "incoming association" and "outgoing association" object instances may communicate with a given transaction control object instance.

In the case where a "setup" or "setupRequest" is received, the number of current associations is checked against the maximum permissible for the subscriber access. If the maximum number of associations has been reached, then the service request is rejected with the appropriate indication returned to the invoking object (i.e. subscriber or association control object). If the maximum number of associations has not been reached, then the appropriate object is created; the "number of associations" updated; and the appropriate service request sent to the created object instance.

When a "transactionComplete" message is received from an "outgoing" or "incoming" association object, the "number of associations" is updated.

3.6.3.2 Outgoing Association

The behaviour of the "outgoing association" object is described in the state diagram of Figure 3.10.

The concept of clustering provided by the statechart formalism has been exploited in this Figure. For example, a number of internal states are clustered into the outgoing

establishment state. This clustering recognizes that the receipt of "release-related" messages in any one of these internal states causes a transition to an "association disestablishment" state.

The "zoom-out" concept of statecharts is used in the representation of the "outgoing resource allocation", "establish AC Identity" and "association disestablishment" states. That is, the internal states and transitions of these states are not shown in Figure 3.10.

The association disestablishment behaviour is common to both the "outgoing association" and "incoming association" objects. Hence, the "zoom-out" capability provides a compact way of expressing the "association disestablishment" behaviour in both the "outgoing association" and "incoming association" state diagrams. A "zoom-in" of the "association disestablishment" state is provided in Figure 3.13.

The following describes the behaviour of the outgoing association object. In the description it is assumed that the subscriber provides the complete destination address in the "setup" message.

3.6.3.2.1 Resource Allocation and Establishment of AC Identity

When the "outgoing association" object is created, it is initialized to the "idle" state. Upon receipt of a "setup" message from the "transaction control" object in this state, a "seizeBchannel" message is sent to the "resource control" object in order to allocate a bearer channel to the outgoing association. The "outgoing resource allocation" state is then entered.

If no bearer channel is available, then the transaction is terminated and an indication returned to the subscriber. If a channel is available, then a setupIndication is sent to the "association control creator" object. The "establish AC identity" state is then entered. Since an association control object instance is dynamically created by the AC creator when a setupIndication is received, the AC creator must return an object identifier to the "outgoing association" object. This object identifier uniquely identifies the association

control object instance responsible for establishing, maintaining and releasing the association.

When the object identifier message is received, an indication is returned to the subscriber that the establishment of the requested service is proceeding. The outgoing association establishment state is then entered.

Since asynchronous communication is assumed, it is possible for the "outgoing association" object to receive a release request from the subscriber when it is waiting for a response from either the resource object (i.e. it is in "outgoing resource allocation" state) or the association control creator object (i.e. it is in the "establish AC identity" state). Figure 3.11 provides a "zoom-in" of the "outgoing resource allocation" and "establish AC identity" states. These states encompass a number of internal states and transitions for the handling of a release request from the subscriber in the form of a disconnect, release or release complete message.

For example, consider the zoom-in of the "outgoing resource allocation" state. If a disconnect message is received from the subscriber in the internal "wait for resource" state, then the "disconnect received" state is entered. When a response is received from the resource object in this state (i.e. noChannel or channelAvailable message), the following is performed:

- A release message is returned to the subscriber.
- A releaseChannel message is sent to the resource control object if a channel was successfully allocated (i.e. the channelAvailable message was received).
- The release pending state is entered.

When a release complete message is received from the subscriber in the "release pending" state, the transaction is terminated.

Figure 3.11: Zoom-in of the Outgoing Resource Allocation and Establish AC Identity States

3.6.3.2.2 Outgoing Association Establishment

Within the "outgoing association establishment" state, (see Figure 3.12) one or more "report" indications may be received from the association control object. Receipt of the "report" message containing an "alerting" parameter in the "association proceeding" state causes the transition to the "association delivered" state. Subsequent "report" indications do not cause a state change.

Figure 3.12: Zoom-in of the Outgoing Association Establishment State

Receipt of a "setupResponse" message from the association control object, results in a "connect" message being transmitted to the subscriber object. The "associated" state is then entered.

3.6.3.2.3 Association Disestablishment

The association release may be initiated by either the originating or responding subscriber. The originating subscriber initiates the release by sending a "disconnect" message to the "outgoing association" object. Similarly, the responding subscriber initiates the release by sending a "disconnect" message to the "incoming association" object. The receipt of a "releaseComplete" or "release" message from the subscriber object is also treated as an association release request.

A release request from the responding subscriber is observed by the outgoing association object as the receipt of a "releaseRequest" message from the association control object.

The handling of the disconnect, release, releaseComplete and releaseRequest messages are described in Figure 3.13. Two general observations can be made regarding the internal states and transitions of the association disestablishment state:

1. Interactions between the subscriber and the "outgoing or incoming association" object must be co-ordinated with interactions between these objects (i.e. "outgoing or incoming association" object) and the association control object. For example, the receipt of a disconnect message from the subscriber results in a releaseIndication message being sent to the association control object. When an acknowledgement is received from the association control object (i.e. a releaseResponse message is received), a release message is returned to the subscriber.
2. The state transitions take into consideration that two objects may attempt to "simultaneously" release the association. For example, an outgoing association object may send a "releaseIndication" message to the association control object at the "same time" the association control object sends a "releaseRequest" message to

Figure 3.13: Zoom-In of Association Disestablishment State

the outgoing association object. This is shown by the receipt of the "releaseRequest" message in the "disconnectIndication" state. In this case, the "releaseRequest" message is treated as an acknowledgement to the "releaseIndication" message.

It should be noted that the Statechart feature of *junctions* has been exploited in Figure 3.13 to economize on the number of lengthy arrows present in the diagram and in the number of identical portions of transition labels. That is, the transitions leading into the "release request" state all share the common output (subscriber!release) and action (startTimer(releaseRequest,4)) and hence, this identical portion of the transitions is shown after the junction.

3.6.3.3 Incoming Association

The behaviour of the "incoming association" object is shown in the state diagram of Figure 3.14. As for the outgoing association object, the statechart concepts of clustering have been exploited. In the following description, it is assumed that only one subscriber responds to the setup message.

When the incoming association object is created, its state is initialized to idle. Upon receipt of a "setupRequest" message from the transaction control object in this state, a "seizeBchannel" message is sent to the resource control object in order to allocate a bearer channel to the incoming association. The "setupRequest" message received from the transaction control object contains the identity of the association control object instance to which further communication should be directed.

If no channel is available, a "releaseIndication" message is returned to the association control object and an internal association disestablishment state is entered to wait an acknowledgement.

Figure 3.14: Incoming Association State Diagram

If a channel can be allocated to the incoming association, a "setup" message is transmitted to the subscriber object and the "incoming association establishment" state is entered. The subscriber can respond to the setup message by returning a connect or report message causing the appropriate state to be entered. Release of the association is similar to that described for the "outgoing association".

3.6.4 Association Control

The state machine descriptions for the association control sub-objects were developed specifically for this study.

3.6.4.1 Association Control Creator

The temporal behaviour of the association control creator object is described in terms of a single state, "wait for request", and a single transition (see Figure 3.15).

Figure 3.15: Association Control Creator State Diagram

The receipt of a "setupIndication" from an outgoing association control object causes the following:

- A "bearer association control" object instance to be created.
- The identity of the created instance to be returned to the invoking object.
- A "setupIndication" message to be transmitted to created instance.

3.6.4.2 Bearer Association Control

The temporal behaviour of the bearer association control object is described in the state diagram of Figure 3.16. There is a one-to-one mapping between an bearer association control object instance and each end of the association: the invoking object instance (which is an instance of the "outgoing association" class) and the responding object instance (which is an instance of the "incoming association" class).

3.6.4.2.1 Establishment of Responder's Identity

When the bearer association control object is created, its state is initialized to idle. Upon receipt of a "setupIndication" from the association control creator object in this state, the bearer association control object performs a routing function. Included in the setupIndication message is the identity of the initiator or invoker of the association (e.g. the outgoing association object instance).

If the routing fails (e.g. no defined route, congestion), then a "releaseRequest" message is sent to the invoking object (e.g. outgoing association object instance). If the routing is successful, then a "setupRequest" message is sent to the transaction control object identified from the routing function and the "wait for Responder Id" state is entered. Recall from Figure 3.9 that the transaction control object will dynamically create an object instance to handle the incoming association when a "setupRequest" message is received from an association control object. Hence, the transaction control object must return the

Figure 3.16: Bearer Association Control State Diagram

"responder identity" (i.e. the identity of the "incoming association" object instance) to the bearer association control object.

When the identity of the responding object (i.e. incoming association object instance) is received, the "wait for setup confirm" state is entered.

The invoking outgoing association object instance may request the release of the association before the identity of the responding object is received. This corresponds to the receipt of a `releaseIndication` in the "wait for responder id" state. In this case the bearer association control object will acknowledge the invoker's release request and will enter the release pending" state. If a `rejectIndication` is then received by the transaction control object, then the association attempt is terminated. However, if the transaction control object returns the identity of the responder, a `releaseRequest/releaseConfirm` message exchange must be performed with the responder before the association attempt is terminated.

3.6.4.2.2 Establishment of Association

An association report message received in the "wait for setup confirm" state is relayed between the "associated" objects.

The receipt of a "setupConfirm" message causes a "connectChannels" message to be sent to the resource control object in order to establish a switched path between the bearer channels allocated to the association. The "establish switched path" state is then entered".

When a "channelsConnected" message is returned by the resource control object, a `setupResponse` message is returned to the invoking outgoing association object instance and the "association established" state is entered.

Figure 3.17 provides a zoom-in of the "establish switched path" state. It shows the various internal states and transitions for handling a release request from either end of the association when a response to the "connectChannels" message is pending.

Figure 3.17: Zoom-in of the Establish Switched Path State

3.6.4.2.3 Release of Association

Receipt of a `releaseIndication` message from one end of the association causes a `releaseResponse` to be returned to that end; a `releaseRequest` message to be sent to the other end of the association; and a `releaseConnection` message to be sent to the resource control object in order to release the switched path.

In order to provide a more compact state transition diagram, only one transition is shown corresponding to the receipt of a `releaseIndication` or `releaseConfirm` message from "one end" of the association. As identified at the bottom of Figure 3.16, this single transition should be replaced with two transitions: one corresponding to the receipt of a `releaseIndication` or `releaseConfirm` message from the outgoing association object instance and the other corresponding to receipt of a `releaseIndication` or `releaseConfirm` message from the incoming association object instance. That is, "one end" has been used to generically refer to either the invoking or responding object.

Further, as shown at the bottom of Figure 3.16, when the handling of a message from "one end" results in the sending of a message to the "other end", then the effect of binding the "one end" to the outgoing association object causes the "other end" to be bound to the incoming association object and vice versa (i.e. when "one end" becomes the incoming association object, then the "other end" becomes the outgoing association object).

Before the "wait for release confirm" state is entered, the identity of the end from which the `releaseConfirm` message is pending is stored in the variable *waitingEnd*. The receipt of a `releaseConfirm` message in the "wait for release confirm" state causes the transition to the terminated state if the "one end" sending this message is the end from which the `releaseConfirm` message is pending (i.e. "one end" equals *waitingEnd*).

3.7 Summary

In this chapter, an "object-oriented" approach was used to specify the basic telecommunication service. Within this "object-oriented" perspective, the specification was structured or organized in terms of "objects". Objects could be decomposed into sub-objects to provide various levels of abstraction. Critical to this object-oriented perspective was the notion of concurrency and state. Each "object" could be modelled in terms of an extended finite state machine which works autonomously and concurrently with other objects.

The concepts and notations of channels provided by the SDL and Estelle formal specification techniques were used to express the communication relationships between objects. The concepts of hierarchy provided by the statechart formalism was exploited in expressing the temporal behaviour of the "objects". Thus the concepts and notations of a variety of formal specification techniques were used in the development of the basic telecommunication service specification.

Having developed the basic telecommunication service specification, the next chapter examines the problems of specifying supplementary services and introduces another concept of object-orientation, namely inheritance, as a means to addressing these problems.

CHAPTER 4

Overview of the Delta Service Specification Approach

This chapter outlines an approach which will allow new services to be introduced into the basic telecommunication model in a structured and modular fashion.

The chapter commences with an overview of the service specification problem. Section 4.2 then outlines the basis of a "delta service specification" approach which addresses the concerns raised in Section 4.1. In the development of this approach, the concept of "delta specification components" as the basic building blocks for forming specifications of individual and combined supplementary services is introduced.

Section 4.3 explores the delta specification approach as it applies to the specification of individual supplementary services. It identifies how the delta specification components can be represented within an object-oriented framework as "supplementary service delta" objects. Two types of "supplementary service delta" objects are distinguished and their relationship to the basic service objects are defined. A state diagram notation for expressing the behaviour of the "supplementary service delta" objects is then developed.

Section 4.4 presents examples of how combined supplementary services can be represented with the delta specification approach and raises the issue of possible conflicts in combining services. Section 4.5 summarizes the essential features of the delta specification approach.

Chapter 5 applies this approach to the specification of a number of supplementary services and their combinations.

4.1 Overview of the Service Specification Problem

Figure 4.1 considers the development of telecommunication service specifications in terms of three steps. The starting point is the development of a specification for the basic telecommunication service. This step was the subject of the previous chapter, and encompassed the specification of a number of objects, their communication relationships, and their temporal behaviour expressed as extended finite state machines.

The second step shown in Figure 4.1 is the specification of supplementary services on an individual basis. As shown on the "specification shelf" of Figure 4.1, there is a specification for each supplementary service. Each supplementary service specification includes elements of the basic service specification as supplementary services derive their behaviour from the basic service.

Step 2 recognizes that one network service provider may select to offer "supplementary service 1" to its subscribers; whereas another network service provider may support only "supplementary service 3". Furthermore, the network service provider that supports a number of different services will want the flexibility to allow subscriber A to subscribe to "supplementary service 1" and subscriber B to subscribe to "supplementary service 3".

Finally step 3 is the specification of combinations of supplementary services. The combined supplementary service specifications include elements of the basic service specification and the individual supplementary service specifications.

Step 3 acknowledges that different network service providers will offer different services in various combinations to their subscribers.

Figure 4.1: Combinatorial Service Specification Problem

As shown in Figure 4.1, this approach may lead to a combinatorial explosion of service specifications. That is, upon completion of step 3, there are the following number of separate specifications or "boxes" on the conceptual specification shelf:

- One basic service specification,
- N supplementary service specifications; and
- $\sum_{r=2}^n \frac{n!}{(n-r)!r!}$ combined supplementary service specifications

The latter assumes that all n supplementary services can be combined without any restrictions. That is, a subscriber with a given set of services is not prevented from subscribing to any other service.

Another issue concerns the overlap between specifications. Since the supplementary services derive their behaviour from the basic service, there is duplication between individual service specifications (e.g. between the basic service specification and the supplementary service 1 specification, between the supplementary service 1 specification and the supplementary service 1 + 2 specification).

Duplication between service specifications may potentially lead to inconsistencies between specifications. Furthermore, there is the question of maintainability. If there is a change or correction to the basic service specification, this change must be reflected in all the individual and combined supplementary service specifications.

Therefore, the service specification problem is perceived as two separate but related sub-problems:

1. The first concerns the need for an approach which will allow supplementary services and their combinations to be expressed in a more *compact* manner. In other words, there is an objective to have *smaller* "boxes" on the conceptual specification shelf of Figure 4.1.

2. The second relates to the need for an approach which will allow combined services to be specified in a *manageable* way (i.e. without resulting in a combinatorial explosion of service specifications). In other words, there is an objective to have *fewer* "boxes" on the conceptual specification shelf of Figure 4.1.

The following section outlines the basis of an approach which addresses the above issues.

4.2 Service Delta Components as Specification Building Blocks

A particular characteristic of supplementary services is that they modify or supplement a basic telecommunication service. This characteristic can be exploited in the development of a specification approach. Specifically, the behaviour of a supplementary service can be specified in terms of the additions and/or modifications it introduces to the basic telecommunication service. These additions and/or modifications are referred to in this study as the "delta" behaviour.

Figure 4.2 summarizes the delta approach to the service specification problem. Supplementary services can be specified in terms of a number of smaller "components". These specification components are identified at the top of the Figure and constitute a "basic service specification" component and a number of supplementary service delta components. An individual supplementary service can be specified in terms of the "basic service specification" component and its corresponding supplementary service delta. A combined supplementary service specification is composed of the basic service specification component and the corresponding supplementary service deltas.

The premise of this specification approach is that these "deltas" can be combined in a non-conflicting manner. That is, the combined service specification can be expressed as a simple function (e.g. union) of the individual deltas. For example, if the combined service introduced a new behavioural element which was not "easily" expressed as a combination of the individual deltas, then a new "combined" delta would need to be added to our list of "specification components".

2

Figure 4.2: Delta Approach to Service Specification Problem

The question then arises as to whether the delta approach may lead to a combinatorial explosion of deltas. Assuming this is the case then the combinatorial problem has not been completely resolved. However, what has been achieved is a more compact expression of the combined service specifications.

Section 4.3 identifies how the delta approach to individual service specification can be accommodated within the object-oriented model. Sections 4.4 investigates the delta approach as it relates to the combinatorial specification problem.

4.3 Representation of Service Delta Components within Object Model

The previous section introduced the concept of defining a number of "specification components" which could be combined in different ways to form specifications of individual supplementary services and combined supplementary services. However, the nature of the "specification components" were not explicitly discussed other than to identify "delta components" as those which express additions and/or modifications to the basic service specification component.

Section 4.3.1 identifies two approaches to accommodate the "delta specification components" within the object-oriented model. Specifically, two types of "supplementary service delta" object classes are identified and their relationship to the basic service objects discussed.

Section 4.3.2 then describes how the state diagram notation used to represent the behaviour of basic service objects in Chapter 3, can be adapted to specify the behaviour of the "supplementary service delta" objects.

4.3.1 Supplementary Service Delta Object Classes

The "basic service" specification component, developed in Chapter 3, is based on an object-oriented model. The overall view of the service is described as a single object, the

behaviour of which is constituted by the combined behaviour of a number of sub-objects². That is, the basic service model was developed using an approach of successive decomposition.

The question then arises as to whether an approach of successive "composition" can be used to "build" the supplementary service specification from the basic service object and a "supplementary service delta" object. This approach is possible if the "delta behaviour" introduced by the supplementary service can be encapsulated in a separate object which works autonomously and concurrently with the basic service object. This further implies that the "delta behaviour" can be expressed as a "separate" state machine from the basic service state machine. This approach is referred to in the following discussion as: *"peer object specification of delta behaviour"*.

However, typically the nature of supplementary services is such that they modify the behaviour of the basic service object. In this case, the "delta" behaviour cannot be expressed as a "separate" state machine but rather reflects an alteration of the basic service state machine. This alteration may be manifested in changes to the basic service state transitions, introduction of new state transitions, introduction of new states, etc.

The concept of inheritance, which is fundamental to object-oriented programming languages [WEGN 87], can be used to express this "alteration" of behaviour. Inheritance allows new objects to be defined as "specializations" or "variations" of other objects. The new object type or class is a *subclass* of the existing class and the existing class is the *superclass* of the new class. The approach of defining a new "supplementary service delta" object which specifies the variations on the basic service object is referred to as: *"subclass object specification of delta behaviour"*.

²In general, supplementary services affect the behaviour of several objects of the basic service model developed in Chapter 3. For the purpose of providing a high level discussion in this section, the basic service impact is discussed from the perspective of the effect on the basic service as a whole (i.e. the impact on a single "basic service" object).

The concept of using inheritance to specify variations of object behaviour is identified in [MOLL 87]. This paper presents OSDL, an object-oriented extension of SDL, which supports extensions for expressing generalization/specialization (class/subclass) of process types. SDL processes are considered the primary objects in OSDL.

The subclass object specification approach uses the *OSDL concepts* of specialization to specify additional states and additional/modified transitions. However, there are differences between the OSDL notation for expressing generalization/specialization and the notation adopted in this study for expressing the delta behaviour. These differences are discussed in Section 4.3.2.

Figure 4.3 summarizes the two approaches discussed above for the specification of delta behaviour introduced by supplementary services within an object-oriented framework. In this Figure the following conventions have been adopted:

- Boxes or squares are used to represent object classes as in Chapter 3. Thick lines are used to represent the class relationships or hierarchy.
- Parallelograms are used to represent active objects which are related to object classes via the "instance of" relationship (shown as arrows drawn from the active object to the object class).
- A supplementary service is described in terms of active objects and their communication relationships. Lines drawn between active objects identify a particular communication relationship.

In both approaches the "delta behaviour" is encapsulated in a new object class: "a supplementary service delta" object class. As shown in the Figure, the class relationship

**Figure 4.3: Approaches to Express Delta Behaviour
Within Object-oriented Framework**

between the "basic service" object and "supplementary service delta" object is as follows :

1. *Peer Object Specification:* The "basic service" object class and the "supplementary service delta" class represent "peer" classes in that there is no class relationship between them.
2. *Subclass Object Specification:* In the class hierarchy, the "supplementary service delta" class is a subclass of the "basic service" object.

The supplementary service is formed from instances of these object classes as follows:

1. *Peer Object Specification:* In this approach, an active instance of the basic service object communicates with an active instance of the "peer" supplementary service delta object to form the supplementary service.
2. *Subclass Object Specification:* In this approach, an active instance of the "subclass" supplementary service delta object forms the supplementary service via inheritance of the basic service characteristics

4.3.2 A State Diagram Notation for Expressing Delta Behaviour

4.3.2.1 Overview of Notation

In Chapter 3, the behaviour of each object was modelled as a finite state machine (FSM) and represented using a state diagram based notation. The question then arises as to whether this notation can also be used in the specification of the delta behaviour introduced by the supplementary services.

Consider first the "peer object" approach to the specification of delta behaviour identified in Section 4.3.1. The premise of this approach is that the delta behaviour can be encapsulated in a separate or "peer" object which works autonomously and concurrently with the basic service objects. Therefore, the behaviour of the "peer object" can be

modelled as a separate finite state machine and expressed using the notational conventions presented in Chapter 3. That is, the state diagram notation introduced in Chapter 3 is sufficient for describing the "delta behaviour" in the "peer object" specification approach.

Now consider the "subclass object" approach to the specification of delta behaviour. In this approach, the delta behaviour (encapsulated in a subclass supplementary service delta object) cannot be modelled as a separate finite state machine, but rather reflects an alteration of the basic service state machine.

This "alteration of behaviour" could be represented by a "partial" or "delta" state machine. This partial or delta state machine would identify the new states and transitions which should be added to the basic service state machine, as well as transitions which should override the basic service transitions.

To illustrate this concept consider the example shown in Figure 4.4. A hypothetical basic service object class is modelled in terms of two states (state1 and state2) and three transitions (t1, t2 and t3). A "delta supplementary service object" class is then identified as subclass of the basic service object. This delta object is expressed in terms of a new state (state3) and a number of transitions. The transitions represent both new and modified transitions as described below.

New states and additional/modified transitions are represented in the delta supplementary service object using the same state and transition notation for the basic service object, the only difference being that the symbols are drawn with thicker lines. There is a requirement to have the new states drawn with thicker lines to distinguish them from "basic service" states. It is not essential for the transitions to be drawn with thicker lines as their presence in the subclass service delta object is sufficient for identifying additional/modified behaviour.

The relationship between the transitions shown in the supplementary service delta object

Figure 4.4: State Diagram Notation for Expressing Delta Behaviour in Subclass Approach

and the transitions defined in the basic service object description are as follows:

1. Transition t4 shown in the supplementary service delta object represents a new transition (i.e. it represents additional behaviour which is introduced by the supplementary service).
2. Transition t2' shown in the supplementary service delta overrides transition t2 in the basic service object (i.e. it represents modified behaviour which is introduced by the supplementary service). In general, a transition is overridden in the basic service object if for a given *current state, input and set of conditions*; a different result is specified (i.e. different next state and/or action and/or output).
3. Transition t1" in the supplementary service delta object description provides *incremental* behaviour to transition t1 defined in the basic service object in the sense that it extends how the receipt of input1 is handled in state1. The supplementary service specifies that upon receipt of input1 in state1, if condition1" is true, then perform action1". If condition1" is not valid, then the receipt of input1 in state1 is handled as defined for the basic service (i.e. action1 is performed).

The supplementary service delta class state transition description presented in Figure 4.4 can be compared with a subclass definition in object oriented programming systems (OOPS). In OOPS, a subclass may provide new method and instance variable declarations. As well, the subclass may override methods of a superclass by providing a new method for an old method's selector (i.e. the symbolic name of a method). As illustrated in the example of Figure 4.4, the supplementary service delta class provides new state and transition "declarations". As well, the service delta class may override transitions of a superclass by providing a new transition for an old transition's "selector" (i.e. *input message and conditions*).

The behaviour or overall state machine description of the supplementary service is shown at the bottom of Figure 4.4. Recall that in the *subclass object specification* approach an active

instance of the supplementary service delta object forms the supplementary service via inheritance of the basic service characteristics. Thus, the overall description is obtained by applying the additions/modifications expressed in the supplementary service delta object to the state machine description of the basic service object.

4.3.2.2 Comparison with OSDL

OSDL is an object-oriented extension to SDL which supports the concepts of generalization/specialization. In [MOLL 87], three ways of "specializing" a process definition are identified as follows:

1. By simple inheritance: a specialization of a process definition can be defined which inherits all the states, signals (i.e. input messages) and transitions from a general process definition, in addition to adding new states, signals and transitions.
2. By "virtual procedures": which allow a *part* of a transition in a general process definition to be modified; specifically the actions performed as part of the transition may be specialized.
3. By "virtual transitions": which allow the complete transition to be specialized (i.e. modified in the specialized process definition).

In case 1 above (i.e. simple inheritance) the specialized process definition introduces new states, signals and transitions using the *same* symbols as in SDL. The concept of introducing new states and new transitions in the subclass delta object definition is also supported in the "subclass object" specification approach of this study. However, the new states and transitions are distinguished in the subclass object definition by drawing the states and transitions symbols with thicker lines.

There is no equivalent concept of "virtual procedures" in the "subclass object" specification approach of this study. That is, in the "subclass object" specification approach the "transition" is the smallest unit of specialization.

The OSDL concept of "virtual transitions" is supported in the "subclass object" specification approach of this study in that a transition in the subclass definition may override a transition in the superclass definition. However, in OSDL transitions are identified as "virtual" in the *superclass* process definition. That is, the superclass process definition explicitly specifies which transitions can be modified by a subclass definition.

In OSDL, a transition is identified as virtual by using a "virtual input" symbol in the superclass definition. The same symbol is used for a virtual input as for the SDL input, the only difference being that the lines of the symbol are not fully drawn. The various actions, output, etc., following a virtual input specify the default behaviour. This behaviour (i.e. the actions, output, etc. following the virtual input) may be modified in a subclass definition.

In the "subclass object" specification approach of this study, no assumptions or information concerning the possible specialization of an object's behaviour is indicated in the superclass definition.

4.4 Expressing Combined Supplementary Services

In the previous section, the delta behaviour introduced by individual supplementary services was expressed within the object model in terms of a "supplementary service delta" object. This object was identified as either a "peer object" class to the "basic service" object class or a "subclass" of the "basic service" object class.

This section provides examples of how combined supplementary services can be represented within the object model using the concepts of the "peer" and "subclass" supplementary service delta object concepts developed in the previous section. The issue of potential conflicts in combining services is briefly introduced in Section 4.4.4.

4.4.1 Example 1: Combining Services Formed of Peer Objects

In the first example (see Figure 4.5), supplementary service 1, formed from the "peer" supplementary service 1 delta object, is combined with supplementary service 2, which is formed from the "peer" supplementary service 2 delta object. In the combined class hierarchy, the basic service object class, the supplementary service 1 delta class and the supplementary service 2 delta class represent peer object classes.

The combined supplementary service is formed from active instances of the basic service class, supplementary service 1 delta class and supplementary service 2 delta class. The basic service instance communicates with both the supplementary service 1 delta instance and the supplementary service 2 delta instance.

4.4.2 Example 2: Combining Services Formed of Peer and Subclass Objects

In the second example (see Figure 4.6), supplementary service 1, formed from the peer supplementary service 1 delta object, is combined with supplementary service 2, which is formed from the subclass supplementary service 2 delta object. In the combined class hierarchy, the basic service object class and the supplementary service 1 delta class represent peer object classes. The supplementary service 2 delta object class is a subclass of the basic service class.

The combined supplementary service is formed from active instances of the "peer" supplementary service 2 delta class and "subclass" supplementary service 1 delta class. The supplementary service 1 instance communicates with the supplementary service 2 delta instance.

Figure 4.5: Example 1: Combining Service Deltas with Peer Objects

Figure 4.6: Example 2: Combining Service Deltas with Peer and Subclass Objects

4.4.3 Example 3: Combining Services Formed of Subclass Objects

In the third example (see Figure 4.7), supplementary service 1, formed from the subclass supplementary service 1 delta object, is combined with supplementary service 2, which is formed from the subclass supplementary service 2 delta object. In the combined class hierarchy, both the supplementary service 1 delta class and the supplementary service 2 delta class are subclasses of the basic service class.

The combined supplementary service is formed from an active instance of the supplementary service 1 delta class and the supplementary service 2 delta class. This scenario is conceptually equivalent to the concept of "multiple inheritance" in object-oriented programming languages. That is, the combined service behaviour is a union of the behaviour of the supplementary service 1 delta behaviour and supplementary service 2 delta behaviour.

However, object-oriented languages which support multiple inheritance such as Eiffel [MEYE 88] require the definition of a new class which inherits from multiple superclasses. Following this approach, Figure 4.7 could have included a "supplementary service 1/2 delta" class in the class hierarchy which would be a subclass of both the "supplementary service 1" delta class and the "supplementary service 2" delta class. The combined supplementary service would then be formed from an active instance of this "supplementary service 1/2" delta class.

It should be noted that in the example of Figure 4.7, the combined "supplementary service 1/2" class does not specify any additional and/or modified behaviour to the individual supplementary service delta classes. Therefore, if one viewed inside of the box (i.e. the supplementary service 1/2 class specification) he/she would find it to be empty (i.e. no states or transitions specified).

It is considered that a *combined* supplementary service delta class should only be explicitly shown (as illustrated as an additional "box" in Figure 4.8 a) when there is a requirement to

Figure 4.7: Example 3: Combining Service Deltas with Subclass Objects

Figure 4.8: Combining Subclass Objects with and without Conflicts

specify additional and/or modified behaviour to the individual supplementary service delta classes. The need to introduce such a combined class is considered in Section 4.4.4.

It should be noted that in the case where the supplementary services can be combined in a non-conflicting manner (i.e. combined service does not specify any additional and/or modified behaviour to the individual supplementary service delta classes), Figure 4.8b can be used as an alternative representation to Figure 4.7. That is, the individual supplementary service deltas are arranged in a single inheritance tree where the order of inheritance may or may not be important.

4.4.4 Conflicts in Combining Services

In the discussions of Sections 4.4.1 to 4.4.3, it was assumed that there was no interaction or conflict between supplementary services when combined. This allowed services to be combined in a modular and linear fashion.

However, the question then arises as to what impact does interactions between supplementary services have on the resulting combined specification. Two examples are considered below.

Consider example 1 (Section 4.4.1) and assume that a particular event within the supplementary service delta 1 description causes another event to occur within the supplementary service 2 delta. For example, the activation of supplementary service 1 causes supplementary service 2, if active, to be terminated or cancelled. In this case, the behaviour of the supplementary service 2 delta is modified when the supplementary service 1 delta is present. As a consequence, the resulting combined service specification may include additional supplementary service 1/2 delta classes which specify the modified behaviour when both services are present (see Figure 4.9).

Now consider example 3 (Section 4.4.3) and the behaviour introduced by each of the supplementary service deltas is expressed as shown in Figure 4.10.

Figure 4.9: Combining Peer Service Deltas with Conflicts

Figure 4.10: Combining Subclass Service Deltas with Conflicts

As illustrated in Figure 4.10, both individual supplementary service deltas specify a transition for the handling of the inputX in the stateX. The supplementary service delta 1 specifies that upon receipt of inputX in stateX, action1 should be performed; output1 should be sent and stateY entered. The supplementary service delta 2 specifies that upon receipt of inputX in stateX, action 2 should be performed; output2 should be send and stateY entered.

Now assume that when supplementary service 1 and 2 are combined, the combined behaviour is specified such that upon receipt of inputX in stateX, action1 and action2 should be performed; output2 should be sent and stateY entered.

In this case, the combined behaviour is not a complete union of the individual supplementary service deltas (i.e. output 2 is sent in the combined service, as opposed to both output 1 and 2 being sent). As a consequence, a new combined supplementary service 1/2 object class may need to be defined, as shown in Figure 4.10, which would be a subclass of the individual service delta classes. This new combined service delta subclass would inherit the behaviour of the individual delta service classes and provided modified behaviour for the "common" transition.

4.5 Summary of the Delta Service Specification Approach

In this chapter, the problems of specifying individual and combined supplementary services were introduced and an approach based on the concept of "delta service specification components" was developed. Two ways for accommodating the "delta specification components" within an object-oriented framework were identified. In the *peer object specification approach*, an autonomous and concurrent "supplementary service delta" object is specified which communicates with the basic service object to form the supplementary service. In the *subclass object specification approach*, a "supplementary service delta" object is specified which forms the supplementary service via inheritance of the basic service behaviour.

Visual formalisms were presented for expressing the class, instance and communication relations in the *peer object* and *sub-object* specification approaches. A state diagram notation was also developed for expressing the delta behaviour in the *sub-object* specification approach.

Finally, this chapter presented some examples of how combined services could be represented within the object-oriented delta specification approach and the question of possible conflicts in combining services.

In the following chapter the concepts of the object-oriented delta specification approach will be examined by applying it to the specification of a number of supplementary services and their combinations. The expressiveness of the visual formalisms presented in this Chapter will be explored by means of these service examples. Potential interactions or conflicts between supplementary services will be investigated and the requirements to specify combined service delta components will be analyzed.

CHAPTER 5

Application of Delta Approach to Supplementary Service Specification

This chapter applies the delta service specification approach outlined in Chapter 4 to the specification of the following supplementary services and their combinations:

- **Hold Service**
- **Call Waiting Service**
- **Call Forwarding Services**
- **Call Completion to Busy Subscriber (CCBS) Service**

The discussion of the individual supplementary services (Sections 5.1 to 5.4 inclusive) is organized as follows:

- A brief prose definition of the service is provided. A more detailed description is included in Appendix B, as well as references to the documents from which the prose descriptions were derived.
- The interactions or delta behaviour introduced by the supplementary service is discussed at a high level.
- The representation of the delta behaviour in terms of the "peer object" and "subclass object" specification approaches identified in Chapter 4 are explored. In the "subclass object" approach, the state diagram notation developed in Section 4.3.2 is used to express the alterations on the basic service state machines.

The ability to combine multiple services by combining the individual service deltas is analyzed in Section 5.5.

Section 5.6 summarizes the results of the service analysis.

5.1 Hold Service

The "Hold" service allows a subscriber to interrupt communications on an existing incoming or outgoing association and then subsequently, if desired, to re-establish communication. It is assumed that the bearer channel assigned to the association is released for subsequent re-use by another association when the "hold" service is granted. There is also the provision to reserve the bearer channel so that the channel will always be available on the subscriber's access for the user to re-establish communications. This latter scenario is not considered in the following analysis.

As a subscription option, a notification may be sent to the remote subscriber to notify that: 1) the association is being held and 2) the held association is being retrieved.

5.1.1 Interaction with the Basic Telecommunication Service

The hold service affects the behaviour of a number of objects in the basic service model, specifically: the outgoing association, incoming association and the association control objects. The impact on each of these objects is discussed in the following subsections.

5.1.1.1 Outgoing Association Object Interactions

The hold service does not directly affect the states of the outgoing association object. Rather, the hold service introduces a number of additional states which are decoupled from the outgoing association states. This decoupling is shown in Figure 5.1 by showing two components of which the orthogonal product describes the overall state of the supplementary service. One component describes the behaviour of the outgoing association object, whereas the other component reflects the additional behaviour introduced by the hold service (i.e. the hold supplementary service delta).

Figure 5.1: Hold Service Interaction

The following outlines the interactions between the "hold" component and the outgoing association object:

1. A number of transitions in the hold component are dependent on the state of the outgoing association object. For example, a "hold request" from a subscriber will be rejected if the outgoing association object is not in the "outgoing association establishment" state. Furthermore, a retrieve request from the subscriber in the "held" state is rejected if the outgoing association object is not in the "association disestablishment" state (i.e. the association is in the process of being released).
2. A transition in the hold component is dependent on transitions in the outgoing association component. Specifically, the transitions defined in the outgoing association object leading from the "association disestablishment" to the "terminated" state cause the hold component to return to the idle state. That is, when the outgoing association transaction terminates, the hold service essentially "terminates". The transition of the hold component to the idle state is shown in Figure 5.1 by the receipt of a generic "association terminated" event. The nature of this event is discussed in Section 5.1.2.
3. Another interaction concerns the requirement of both components to know whether a bearer channel is allocated to the association and if so, the identity of the channel. In the absence of the "hold" component, a bearer channel is allocated and deallocated by the outgoing association object. By introducing the "hold" service, another component, (i.e the hold component) can also allocate/deallocate a bearer channel to the association.

The implication of this interaction is that when the outgoing association object releases the association, there is no longer a requirement to send a "releaseBchannel" message to the resource control object in all circumstances. That is, when the hold component is in the "held" state, the channel has already been released.

5.1.1.2 Incoming Association Object Interactions

The impact of the hold service on the behaviour of the incoming association object is similar to the impact on the outgoing association object. The only variation is the conditions under which an incoming association can be held. Figure 5.1 could be used to reflect the impact of the hold service on the incoming association object, with the following changes:

1. The component on the right (i.e. the behaviour of the outgoing association object) is replaced by the incoming association state diagram.
2. In the hold component, a "hold request" will be rejected if the incoming association object is not in the "associated" sub-state of the "incomingAssociationEstablishment" state. That is, the condition on the transition from the "idle" to "held" state is changed.

5.1.1.3 Association Control Object Interaction

The impact of the hold service on the behaviour of the association control object is as follows:

1. The remote user is optionally informed when the association is held and the held association is retrieved. The association control object must therefore relay this notification between the two associated objects.
2. When the "hold request" is granted to the requesting subscriber, the switched path is released. In the basic service description, the association control object is responsible for co-ordinating the establishment and release of the switched path between the bearer channels allocated to the association. The switched path is released in conjunction with the release of the association. With the introduction of the hold service, the switched path may be released, while maintaining the association.

5.1.2 Expression of Hold Delta Behaviour for Incoming and Outgoing Association Objects

The following subsections explore two approaches for the specification of the additional behaviour introduced by the hold service on the incoming and outgoing association objects.

The first approach investigated is the encapsulation of the hold delta in a peer object class. The second approach examined is the specification of the hold delta as a subclass of the basic service objects.

5.1.2.1 Peer Object Specification

The decoupling of the hold states from the basic service states may naturally lead to the specification of a new peer hold service delta class which encapsulates the "hold state machine" description shown in Figure 5.1. This object would represent an autonomous and concurrent object which would communicate with either the incoming or outgoing association object (depending on whether the hold service was invoked on an outgoing or incoming association). Figure 5.2 shows the peer object specification approach to the hold service. An instance of the peer hold delta class communicates with an instance of an outgoing association class. The nature of the interactions between the peer hold delta object and the outgoing association object are identified as part of the general discussion of issues below.

Figure 5.2: A Peer Object Approach to Hold Service

Issue 1: Termination of the Hold Object

The hold service is only valid in conjunction with an existing outgoing or incoming association. That is, once the outgoing or incoming association is released, the hold service is "terminated". This interaction was expressed in Figure 5.1 by the receipt of a generic "association terminated" message in the held state of the hold component.

The transaction control object could potentially co-ordinate this interaction between the "hold delta" object and the "outgoing (incoming) association" object. An instance of the "hold delta" class could be dynamically created by the transaction control object when a hold request is received from the subscriber. That is, it would be handled in a similar fashion as a setup request (i.e. an outgoing association object is dynamically created).

The hold delta object may terminate on its own (i.e. the held association is successfully retrieved). In this case, a "holdTransactionComplete" message is sent to the transaction control object. When the outgoing or incoming association object terminates (i.e. a "transactionComplete" message is received by the transaction control object), the transaction control object would remove the "hold delta" object instance.

The advantage of this approach is that the behaviour of the outgoing and incoming association objects is not affected by the introduction of the hold service. However, in preserving the behaviour of the outgoing and incoming association objects the transaction control behaviour has been modified.

The way in which the transaction control behaviour has been modified is identified in Figure 5.3. In this figure, a "transaction control (TC) hold delta" object class is specified as a subclass of the transaction control object class. The "TC hold delta" class specifies a number of new transitions using the state diagram notation developed in Section 4.3.2. Thus by introducing a "peer delta" object as shown in Figure 5.1, there is a need to modify the TC behaviour as in the *subclass* approach.

numAssoc = number of associations
maxAssoc = maximum allowed number of associations

Figure 5.3: Transaction Control Hold Delta (subclass approach)

New transitions are specified in order to:

1. Handle the receipt of a hold message from a subscriber. The hold message contains an identifier (called a **REFERENCE**) of the incoming or outgoing association to which the hold request applies. If the subscriber has subscribed to the hold service, then an instance of the **holdDelta** class is created; the variable "**holdServiceActiveList**" is updated to include the reference and the hold delta object instance identifier; and the hold message is sent to the "newly" created hold delta object instance.
2. Handle the receipt of a **holdTransactionComplete** message from a **holdDelta** object instance. This message causes the variable "**holdServiceActiveList**" to be updated in order to remove the particular reference and **holdDelta** object instance identifier.
3. Describe the additional behaviour for the handling of a "**transactionComplete**" message from an outgoing or incoming association object instance. When the **transactionComplete** message is received and the hold service is active on the specified association (identified by the **REFERENCE** value), then the number of associations will be decremented (as in the basic service description) and the appropriate hold delta object instance will be deleted.

Issue 2: Requirement for Basic Service State Information

A second issue concerns the requirement for the hold delta object to access the state of the basic service object (i.e. outgoing or incoming association object). As shown in Figure 5.1, the hold delta object requires this information in deciding whether a hold/retrieve request from a subscriber can be accepted.

This requirement could be modelled by defining a new "hold channel" between the hold delta object and the outgoing (incoming) association object over which the hold delta object could send a message soliciting the state of the basic service object.

The question then arises as to what effect does this new "message" have on the behaviour of the outgoing (incoming) association object. Essentially, it implies that a new transition is added to every state of the outgoing (incoming) association object state machine. This new transition specifies that upon receipt of a *queryState* message, the current state of the object should be returned to the invoking object and the object shall remain in the same state.

However, the capability of receiving a "queryState" message in every state may be an inherent requirement of the basic service definition. As noted in Chapter 3, all objects in the model and their interactions were not explicitly expressed. It is likely that in the basic service model a management object may request the state of any object in the system. In this case, the ability to receive and respond to a *queryState* message is not a new requirement introduced by the hold service.

Issue 3: Requirement for Sharing of Channel Information

A third issue that must be addressed concerns the requirement for the hold delta object and the basic service object to share responsibility for the allocation/deallocation of the bearer channel to the association.

There are many ways in which this interaction could be modelled. For example, the outgoing (incoming) association object could encapsulate information concerning the identity of the bearer channel allocated to the association. This information would be used by both the outgoing (incoming) association and the hold delta object in co-ordinating the allocation/deallocation of the bearer channel to the association as follows:

1. The hold delta object would update this information when it releases the B-channel on a successful hold service invocation. It would also update this information when it successfully reallocates a channel on a retrieve request from a subscriber. In order to update this information, the outgoing (incoming) association object would need to support a "new message": *setChannelId*.

2. When the outgoing (incoming) association object releases an association, it would use this information in deciding whether to send a "releaseBchannel" message to the resource control object. Recall from Section 5.1.1.1 that in the absence of the hold service, the state of the outgoing (incoming) object uniquely determines whether a B-channel is allocated to the association. When the hold service is introduced, the allocation of the bearer channel also becomes dependent on the state of the hold delta object. This dependency is handled by introducing additional "B-channel allocated state" information in the outgoing (incoming) association object which the hold delta object can modify as described above.

As discussed for the *queryState* message, the *setChannelId* message implies a modification to the behaviour of the outgoing (incoming) association object. That is, the outgoing (incoming) association object must define "transitions" to respond to these new messages. Furthermore, the behaviour of the outgoing (incoming) association object is modified on the release of an association, in that a releaseBchannel message will only be sent to the resource control object if a B-channel is currently allocated.

Issue 4: Delay of Sending Hold Notification Message

A fourth issue concerns the transmission of the "hold" notification by the hold delta object. The service definition assumed in the development of Figure 5.1, was that an optional "hold/retrieve notification" would be sent by the peer "hold delta" object to the remote subscriber. This is represented in Figure 5.1 by the sending of the report message to the association control object when the association is successfully held and retrieved.

There is currently a proposal within CCITT, to delay sending the "hold" notification to the remote user until a user has been selected at remote user-network interface. Recall from Chapter 2 that multiple users may respond to an incoming association on an ISDN passive bus subscriber access arrangement. No benefit is perceived in notifying all responding users of a held association. Only the selected user should be notified.

The implication of this proposal is that it more closely couples the outgoing association object and the peer "hold delta" object. That is, a state transition in the outgoing association object (i.e. remote user selected) must now somehow trigger the sending of a hold notification to the remote user.

The outgoing association object could potentially send a "remote user selected" message to the peer hold delta object when the remote user is selected. Upon receipt of this message, the peer "hold delta" message would send, if appropriate, a hold notification to the remote user. Within this approach, the outgoing association would need to know whether the hold service is active on the particular association, as well as the identity of the peer hold delta object instance. Whereas the existence of a peer "hold delta" object always implies the existence of either an outgoing or incoming association object, the latter is not always true (i.e. the existence of an outgoing association object does not imply the existence of a peer hold delta object).

Alternatively, the outgoing association object could encapsulate information concerning whether a "hold notification" should be sent to the remote user. The peer hold delta object could update this information as appropriate. When the outgoing association object receives an indication that a remote user has been selected, then it would use this information in deciding whether a hold notification should be sent to the remote user. Clearly, the behaviour of the outgoing association object has been modified to support this alternative.

In either case, the requirement to delay the sending of the hold notification affects the orthogonality or independence of the outgoing association object and the peer hold delta object.

5.1.2.2 Subclass Object Specification

It is also possible to express the delta behaviour introduced by the hold service by defining a new hold delta object class as a subclass of the incoming and outgoing association object

classes. This new subclass would encapsulate the additional behaviour associated with the "hold component" of Figure 5.1.

In this approach, it is assumed that the transaction control object would create an object instance of the hold delta class upon receipt of a setup request from a subscriber. This ensures that the additional behaviour required to support the hold service is available if this service is subsequently invoked on the association. Note that if the subscriber has not subscribed to the hold service, then an instance of the outgoing association object class will be created as in the basic service.

Since a *single object instance* incorporates the behaviour of both the outgoing (incoming) association and the hold service delta, when this object terminates (i.e. on release of the association), the hold service is explicitly terminated. This can be compared with the "peer object specification" approach (Section 5.1.2.1, issue 1) where the behaviour of the transaction control was modified in order to co-ordinate the termination of the hold service with the termination of the basic service.

Since the hold service is applicable to both outgoing and incoming associations, there is a need to specify the hold delta as a subclass of both the outgoing and incoming associations. The hold delta subclasses are generally equivalent (and hence, have overlapping behaviour) the only exceptions being:

- The conditions under which an incoming versus an outgoing association can be held.
- The potential requirement to delay sending the hold notification on an outgoing association.

It should be recalled from Chapter 4 that the "subclass object" approach to the specification of delta behaviour was introduced to handle the situation wherein the delta behaviour *could not* be expressed as a "separate" state machine. Rather, the delta behaviour reflected an alteration to the basic service state machine. For the hold service, the "subclass object"

approach has also been applied in the case where the delta behaviour *can* be expressed as a "separate" state machine. However, the resulting active object instance created from the "hold delta" subclass has slightly different semantics from an active object formed from the type of "supplementary service delta" subclass discussed in Chapter 4.

The resulting active object instance created from the "hold delta" subclass essentially incorporates "two state machines". That is, it has two states: the basic service state and the "ancillary" hold service state. In Chapter 4, the resulting active object instance created from the supplementary service delta subclass encapsulated one state machine: the modified basic service state machine.

A potential advantages of using the "subclass" approach in the case where the delta behaviour *can* be expressed as a "separate" state machine, as seen for the hold service, is that information required by both state machines can be more readily shared. For example, the "channel information" and "basic service state" required by the "hold state machine" is "made available" via inheritance.

5.1.3 Expression of Hold Delta Behaviour for Association Control Object

As identified in Section 5.1.1.3, the hold service introduces the additional requirement for the association control object to relay a "held/retrieved" notification between associated objects.

This notification could be conveyed by the general "report" message which is relayed by the bearer association control object. However, in the basic service description, the report message is only received in the "wait for setup confirm" state (see Figure 3.14 - bearer association control state diagram). The hold service requires that the report message can also be received in the "association established" state.

Furthermore, as identified in Section 5.1.1.3, the hold service introduces the additional requirement to release and connect the switched path when the association is held and retrieved, respectively.

To express this additional behaviour, a new "AC hold delta" object class can be specified that is a subclass of the bearer association control object class. This subclass specifies the additions/modifications to the AC object class (shown in Figure 3.14) using the notation described in Section 4.3.2.

Figure 5.4 identifies that the "association control hold delta" object introduces a new state (i.e. association established, switch path released) and a number of new transitions as follows:

- The receipt of a "report" message in the "association established" state.
- Transitions leading from the "association established" to the new state (association established, switch path released) and from the new state back into the basic service association control states.

5.1.4 Hold Service Summary

In this section, it was identified that the hold service introduces additional behaviour which can largely be described as a set of states which are decoupled from the basic service behaviour. This decoupling allowed the "peer object" specification approach of Chapter 4 to be exploited. It was shown that a "subclass object" specification approach could also be used, however, it has slightly different semantics than the "subclass object" specification approach discussed in Chapter 4.

In the "peer object" specification approach, a new peer "hold delta" object was identified which worked autonomously and concurrently with the basic service objects. However, in introducing this new object into the model, the behaviour of some of the basic service objects were modified. For example, the behaviour of the transaction control object was

Figure 5.4: Association Control Hold Delta (subclass approach)

modified in order to co-ordinate interactions between the peer hold delta object and the outgoing association object.

Therefore, the hold service could not be expressed, as described in Chapter 4, by simply introducing a new peer object into the model. The introduction of the peer object affected the behaviour of other objects within the model.

5.2 Call Waiting Service

The "call waiting" service allows a subscriber to be notified of an incoming association with an indication that no bearer channel is available. The user then has the choice of accepting, rejecting or ignoring the waiting association.

5.2.1 Interactions with the Basic Telecommunication Service

The call waiting service affects the behaviour of the incoming association object in a number of ways. The delta behaviour introduced by the call waiting service is described in the following sections.

5.2.1.1 No Channel Available on Incoming Association

Referring to Figure 3.14 (Incoming Association State Diagram), in the absence of the call waiting service, receipt of a "noChannel" message in the "incoming resource allocation" state causes the transaction to be terminated (i.e. completed). With the "call waiting" service, the receipt of the "noChannel" indication in this same state causes a setup message to be sent to the subscriber and the "number of waiting associations" to be incremented if the following conditions are satisfied:

1. The user has subscribed to the call waiting service.
2. The maximum number of "waiting associations" for the interface (a subscription parameter for the service) has not been exceeded.

The setup message sent to the subscriber includes an indication that "no channel" is available.

5.2.1.2 Call Waiting Notification

When the subscriber responds to the setup message with an alerting message, the incoming association object will check whether the subscriber has subscribed to the optional "remote user receives notification that the association is waiting". If the subscriber has not subscribed to this option then a report message with an alerting parameter will be sent to the association control object as in the basic service description (see Figure 3.14).

However, if the subscriber has subscribed to this option, then a report message with both an alerting *and* a call waiting indication parameter will be sent to the association control object.

5.2.1.3 Receipt of Connect Message

Another modification to the basic service description occurs in the handling of the "connect" message from the subscriber. In the basic service description, receipt of this message causes the associated state to be entered. With call waiting, receipt of the "connect" message causes a request to be sent to the resource control object to allocate a bearer channel to the association. Here it is assumed that the subscriber will make a channel available by releasing or placing on hold another association, before the connect message is sent for the waiting association.

If a bearer channel is not available for the waiting association, then the incoming association will initiate the release of the association. In the basic service description, the association is released only upon receipt of an explicit request by either the subscriber or the association control object.

5.2.1.4 Decrement of Number of Waiting Calls

Finally, the call waiting service requires that the number of waiting associations be decremented when the waiting association is either successfully connected or the waiting association is released.

5.2.2 Expression of Call Waiting Delta Behaviour

The delta behaviour introduced by the call waiting service represents an alteration to the behaviour of the incoming association object. The "peer object" specification approach is not applicable since the delta behaviour cannot be modelled as a separate state machine. Rather, the following alteration of behaviour is manifested:

- extensions and modifications to the basic service transitions,
- introduction of new states and associated transitions.

Figure 5.5 expresses the delta behaviour of the "call waiting" service using the "subclass specification" approach. The "IA call waiting delta" object, specifies the additions/modifications to the incoming association object state machine (see Figure 3.14).

The call waiting delta introduces two new states: "call waiting, resource allocation" and "call waiting, release initiated". The "call waiting, resource allocation" state is entered from the basic service states when the subscriber responds to a waiting association with a "connect" message.

The "call waiting, release initiated" state is entered from the "call waiting, resource allocation" state if a B-channel cannot be allocated to the waiting association. A number of new transitions are defined which lead from the "call waiting release initiated" state back into the sub-states of the basic service "association disestablishment" state.

Figure 5.5: Call Waiting Delta (sub-class Approach)

The call waiting delta also extends the transitions of the incoming association object as follows:

1. The receipt of a "noChannel" message in the "incoming resource allocation" state causes the "incoming association establishment" state to be entered if the "maximum number of associations" has not been reached *and* a "releaseIndication" has not been received from the association control object. If these two conditions are not true, then the receipt of a "noChannel" message in the "incoming resource allocation" state is handled as defined for the basic service (see Figure 3.14).
2. The receipt of an "alerting" message in the "wait for user response" state will cause a report message with both an alerting *and* a call waiting indication parameter to be returned to the association control object if the subscriber has subscribed to the notification option. If this condition is not true, then the handling of the alerting message is as defined in the basic service description.
3. The transitions leading from the association disestablishment state to the terminated state are extended to include the action, "decrement number of waiting associations" if the association was still "waiting" on termination.

5.2.3 Summary of the Call Waiting Service

The impact of the call waiting service on the basic telecommunication service model is limited to the incoming association object. The nature of the impact is such that only the "subclass specification" approach can be used to specify the delta behaviour.

5.3 Call Forwarding Services

The "Call Forwarding" services allow a subscriber to have the service provider send incoming associations to another number under specified conditions. There are three variations:

1. **Call Forwarding Unconditional (CFU):** all incoming associations will be forwarded to another number.
2. **Call Forwarding Busy (CFB):** all incoming associations encountering a "busy" condition will be forwarded to another number.
3. **Call Forwarding No Reply (CFNR):** all incoming associations encountering a "no reply" condition will be forwarded to another number.

5.3.1 Interactions With the Basic Telecommunication Service

Two types of "transactions" associated with the call forwarding service can be distinguished, namely:

1. Transactions concerned with the activation/deactivation of the service on a subscriber access basis.
2. Transactions concerned with the invocation of the service on a per-association basis. Note that the service can be invoked (i.e. an association forwarded) only after the service has been activated.

The following subsections describe how these two types of transactions interact with the basic telecommunication service.

In the ensuing discussion, the "served" user identifies the user who has requested that incoming associations to his number be forwarded. This user may also be referred to as the "forwarding" user. The "originating user" is the user who originated the association. The "forwarded-to" user is the user to which the association will be forwarded.

5.3.1.1 Service Activation/Deactivation

The delta behaviour introduced by the activation/deactivation of the call forwarding service can be described as a set of states which are decoupled from the basic telecommunication service states.

The additional behaviour introduced by the call forwarding activation/deactivation requirement is expressed in the state diagram of Figure 5.6

Figure 5.6: CF Activation Control Delta (peer object approach)

The "cfActivateRequest" and "cfDeactivateRequest" sent by a subscriber, may include a password which the service provider will verify before activating/deactivating the service. Furthermore, the service provider may optionally validate the "forwarded-to-number" included in the "cfActivateRequest" to determine that is it a valid number (e.g. it is not a special number such as the police). The verification of the password and the validation of

the forwarded-to-number may require that access to information in an external database be provided. The requirement to interact with another object for the purpose of obtaining information is reflected by introducing the "cfActivateRequest" and "cfDeactivateRequest" states.

5.3.1.2 Service Invocation

Invocation of the Call Forwarding service interacts with the transaction control, incoming association, outgoing association and the association control object. The impact on these objects is different for the three call forwarding service variations.

5.3.1.2.1 CFU Interactions

The "call forwarding unconditional" service (CFU) affects the behaviour of the transaction control, outgoing association and association control objects as follows:.

1) Transaction Control Impact:

With the CFU service, the transaction control object must check whether the CFU service is activated when a setupRequest is received from the association control object. If the service is activated, then a "forwarding indication" should be returned to the invoking association control object with the "forwarded-to" number, the reason for forwarding (e.g. CFU) and an indication whether the originating user should be notified that the association has been forwarded (this indication is set based on the served subscriber's subscription options).

If the served user has subscribed to the "served user receives notification that the association has been forwarded", the transaction control object must provide an forwarding indication to the served user.

2) Association Control Impact:

With the CFU service, the association control object must be able to receive and respond to a new message, "forwarding indication", in the "wait for responder id" state. Upon receipt of this request, the association control object will attempt to reroute the association to the specified forwarding number. If the routing is successful, then a `setupRequest` is sent to the forwarded-to object (the transaction control object of the forwarded-to user) and a report message is returned to the outgoing association object. The report message indicates the following:

- That the association has been forwarded; and
- Whether the originating user should be informed that the association has been forwarded.

If the rerouting fails, then the association is released.

3) Outgoing Association Impact:

With the CFU service, the outgoing association object must be able to receive a report message, containing the following information, in the "association processing" sub-state of the "outgoing association establishment" state:

- an indication that the association has been forwarding
- an indication whether this information (association forwarding) should be presented to the originating user or not.

Upon receipt of the report message, a forwarding notification may be returned by the outgoing association object to the originating user depending on the condition of the forwarding presentation indication.

5.3.1.2.2 CFB Interactions

"Busy" is defined as either:

1. Network-defined busy: The transaction control object determines that the maximum number of associations have been reached.
2. Subscriber-defined busy: The subscriber object responds to an incoming setup message with a release complete message with reason of "busy" specified.

Call Forwarding on "network-determined busy" affects the behaviour of the transaction control, association control and outgoing association objects. Call Forwarding on "subscriber-determined busy" affects the behaviour of the incoming association, association control and outgoing association objects.

1) Transaction Control Impact:

When a setupRequest message is received from the association control object, and the transaction control object determines that the "maximum number of associations" has been reached, then a "forwarding" request should be returned to the invoking association control object. The information sent in the "forwarding request" is the same as described for the CFU service.

As for the CFU service, if the served user has subscribed to the "served user receives notification that the association has been forwarded", the transaction control object must provide this notification to the served user.

2) Incoming Association Impact:

In the basic service description, the incoming association object will send a `releaseIndication` message to the association control object when the subscriber requests to release the association. When the CFB service is activated, the receipt of a `release complete` message from the subscriber with a reason of "busy", causes a "forwarding" indication to be returned to the association control object.

3) Association Control Impact:

The impact of CFB on the association control object is the similar to the CFU service with the exception that the association control object must be able to accept the "forwarding indication" message in both the "wait for responder id" (network determined busy) and "wait for setup confirm" (subscriber determined busy) states.

4) Outgoing Association Impact:

The impact of the CFB service on the outgoing association object is the same as for the CFU service.

5.3.1.2.3 CFNR Interactions

The CFNR service affects the behaviour of the incoming association, association control and outgoing association objects as follows:

1) Incoming Association Impact:

Referring to Figure 3.14 (Incoming Association State Diagram), in the absence of the "CFNR service", receipt of an "alerting" message in the "wait for user response" state causes the "user responded" state to be entered. When the CFNR service is active, the receipt of this message causes an additional action to be performed, specifically, the starting of a "CFNR timer".

As a result of this action, a new event, "expiry of the CFNR timer" is possible in the "user responded" state. Upon receipt of this new event, a forwarding indication is sent to the association control object. The forwarding indication contains the same information as identified for the CFU service.

2) Association Control Impact:

Appendix B identifies two options for the handling a CFNR timeout. The first is the release of the incoming association to the served subscriber and the establishment of association towards the forwarded-to subscriber.

The second option is to maintain the incoming association towards the served user while establishing an association towards the forwarded-to user. Basically, whoever responds first (i.e. served user or the forwarded-to user) is connected to the originating user. The other party is then released.

The first option introduces the minimum impact on the association control object. The association control object must be able to accept a forwarding indication in the "wait for setup confirm" state. Upon receipt of this indication it will send a release indication to the incoming association object and a setupRequest to the forwarded-to object.

The second option introduces the requirement for the association control object to coordinate interactions between three objects: the outgoing association object associated with the original user, the incoming association object associated with served user and the incoming association object associated with the forwarded-to user.

3) Outgoing Association Control Impact:

The interaction with the outgoing association control object is similar to the CFU and CFB services, the only difference being that the "report (forwarding)" message is received in the

"association delivered" sub-state as opposed to the "association processing" sub-state of the "outgoing association establishment" state.

5.3.2 Expression of Call Forwarding Delta Behaviour

5.3.2.1 Service Activation/Deactivation

The additional behaviour introduced by the call forwarding activation/deactivation requirement can be encapsulated in a new peer object class, referred to as the "CF (call forwarding) activation control delta" object class. The behaviour of this object can be represented by the state machine description of Figure 5.6.

5.3.2.2 Transaction Control Object

The impact of the CFU and CFB services on the transaction control object cannot be modelled as a separate state machine but represent variations on the transaction control object. Hence, only the "subclass object specification" approach can be considered.

A "TC CFU delta" and a "TC CFB delta" object class are defined in Figure 5.7 as subclasses of the transaction control object. Both these subclasses extend the handling of a `setupRequest` from the association control object.

The TC CFU delta specifies that if a `setupRequest` message is received and the CFU service is activated, then a forwarding indication should be returned to the invoking AC object.

The TC CFB delta specifies that if a `setupRequest` is received and the following two conditions are valid, then a forwarding indication should be returned to the invoking object:

1. The maximum number of associations have been reached; and
2. CFB is activated.

Figure 5.7:TC CFU and TC CFB Deltas (subclass approach)

5.3.2.3 Incoming Association Object

The impact of the CFB and CFNR services on the incoming association object cannot be modelled as a separate state machine but represent variations on the incoming association object. Hence, only the "subclass object specification" approach can be considered.

A new "IA CFB delta" object class is defined in Figure 5.8 as a subclass of the incoming association object. The "IA CFB delta" specifies the extended handling of the receipt of a releaseComplete message in the "wait for user response" state. If the subscriber returns a reason of "busy" in the release complete message and the CFB service is activated, then a "forwarding indication" will be sent to the association control object. The incoming association is then released as specified for the basic service.

A new "IA CFNR delta" object class is defined in Figure 5.9 as a subclass of the incoming association object. The "IA CFNR delta" object extends the handling of an alerting message in the "wait for user response" state and the handling of a connect message in the "user responded" state. That is, if an alerting message is received in the "wait for user responded" state and the CFNR service is activated, then a new action will be performed: the starting of the CFNR timer. Similarly, if a connect message is received in the "user responded" state and the CFNR service is activated, then the CFNR timer is stopped. A new transition is also specified in the "user responder" state to handle the expiry of the CFNR timer.

5.3.2.4 Outgoing Association Object

The impact of the CFU, CFB and CFNR services on the outgoing association object cannot be modelled as a separate state machine but represent variations on the outgoing association object. Hence, only the "subclass object specification" approach can be considered.

A new "OA CFU/B delta" and a new "OA CFNR delta" object class are defined in Figure 5.10 as subclasses of the outgoing association (OA) object. The additional behaviour

Figure 5.8: IA CFB Delta (subclass approach)

Figure 5.9: IA CFNR Delta (subclass approach)

Figure 5.10: OA CFU/B and OA CFNR Deltas (subclass approach)

introduced by the CFU and CFB services are the same and hence, have been encapsulated in the same supplementary service delta subclass object (i.e. OA CFU/B delta).

5.3.2.5 Association Control Object

The impact of the CFU and CFB services on the association control object cannot be modelled as a separate state machine but represent variations on the association control object. Hence, only the "subclass object specification" approach can be considered for these services.

A new "AC CFU delta" object class and a new "AC CFB delta" object class are defined in Figure 5.11. The CFB service introduces the same variations on the association control object as the CFU service, plus it adds some additional behaviour. This has been exploited in the class relationship by defining the "AC CFU delta" as a subclass of the bearer association control object and the "AC CFB delta" as a subclass of the CFU delta object class.

Figure 5.12 specifies the additional behaviour introduced by the CFNR service as a "AC CFNR delta" object class, which is a subclass of the bearer association control object. In this figure, it is assumed that the association control object releases the incoming association to the served user before forwarding the association to the forwarded-to user.

5.3.3 Summary of Call Forwarding Services

The Call Forwarding services introduce a new peer "call forwarding activation control delta" object class to support the additional behaviour required for the activation/deactivation of the service. These services also introduce a number of new delta objects which are subclasses of the basic service object classes.

Figure 5.11: AC CFU and CFB Deltas (subclass approach)

Figure 5.12: AC CFNR Delta (subclass approach)

5.4 CCBS Service

The "Completion of Calls to Busy Subscriber (CCBS)" service allows a subscriber encountering a busy condition on a basic telecommunication service request, to be notified when the destination becomes not busy. The network service provider will then reinitiate the association to the specified destination if the subscriber so desires.

5.4.1 Interactions with the Basic Telecommunication Service

The CCBS service does not directly impact the states of the outgoing association object. Rather, the CCBS service introduces a number of additional states which are decoupled from the outgoing association object states. To represent this decoupling, two components are defined in Figure 5.13, of which the orthogonal product describes the behaviour of the supplementary service. One component describes the behaviour of the (basic service) outgoing association object, whereas, the other component reflects the additional behaviour introduced by the CCBS service.

The "interaction" between the two components occurs in the handling of an "releaseRequest" message from the association control object. The outgoing association object handles this message as in the basic service description. However, if the releaseRequest message contains a release reason of "busy" and the subscriber has subscribed to the CCBS service, then the "CCBS delta" component will start a CCBS retention timer and enter the "wait for user to invoke CCBS" state. The way in which the "releaseRequest" message is provided to both components is discussed in Section 5.4.3.

There is no interaction between the components after the CCBS delta component enters the "wait for user to invoke CCBS" service. That is, the outgoing association may be released by the user (outgoing association object enters the terminated state) while the CCBS delta component remains in the "wait for user to invoke CCBS". Similarly, the CCBS delta component may enter the "monitor for not busy" state upon receipt of a "CCBS invoke request" from the subscriber, while the outgoing association object remains in the "association disestablishment" state.

Figure 5.13: CCBS Interaction

In the "monitor for not busy" state, the destination user is being monitored/scanned for not busy. When the destination is no longer busy, a "CCBS recall indication" will be sent to the subscriber and the CCBS component "wait for user recall" state will be entered. When the subscriber accepts the CCBS recall, the CCBS component will send a "setup" message to the transaction control object containing the original association establishment information (e.g. destination user number). The CCBS component is then terminated. From this point on, the basic service procedures are followed. That is, the transaction control object will create a new outgoing association object to handle the new association attempt. If the destination is again busy, the user may invoke a new CCBS request at this time .

As shown in Figure 5.13, the CCBS component may enter the "terminated" state from other states. For example, the subscriber may explicitly request that the CCBS service request be cancelled when the CCBS component is in the "monitor for not busy" or "wait for user recall" states. Furthermore, the "terminated" state is entered if the subscriber does not respond to the CCBS recall within the "recall" period of time.

There are two separate issues to consider in specifying the additional behaviour introduced by the CCBS service, namely:

1. The additional requirement introduced by the CCBS service to monitor/scan the destination subscriber for the "not busy" condition.
2. The coupling or dependency between an original outgoing association and the "recalled" outgoing association. This coupling is illustrated in Figure 5.13 by the entering of a CCBS component state machine when the original outgoing association received a "release on busy" request. The termination of the CCBS component coincided with the startup of a new outgoing association object to handle the "recalled" association.

5.4.2 Expression of Delta Behaviour for CCBS Monitoring/Scanning Requirements

There are several ways in which the monitoring/scanning capability required to support the CCBS service could be modelled. One approach is to specify a new object class which would be responsible for the monitoring/scanning functions. One object *instance* would be responsible for monitoring/scanning an *individual subscriber* for a "not busy" condition.

Figure 5.14 identifies the behaviour of the monitoring/scanning object assuming that the "not busy" condition is determined by polling various objects for their status.

As shown in Figure 5.14, when the monitoring/scanning object receives a "monitorRequest" in the "idle" state, the following is performed:

- The maximum monitoring timer is started. This timer specifies the maximum length of time the subscriber will be scanned/monitored for a not busy condition.
- The polling timer is started.
- An acknowledgement is returned to the object which initiated the monitor request .
- The "wait for polling timeout" state is entered.

In the "wait for polling timeout" state, the receipt of a "polling timeout" causes various objects to be polled for their status, such as:

- The transaction control object to determine whether the number of associations is currently at the maximum number defined for the subscriber access.
- The resource control object to determine if there is an available bearer channel for the association.

Figure 5.14: CCBS Monitoring/Scanning Delta (peer object approach)

If, based on this polling, the monitor/scanning object determines that the subscriber is "not busy", then an "idle guard timer" is started and the "wait for guard timeout" state is entered. This timer specifies the maximum length of time the monitoring/scanning object will delay, after the subscriber has become not busy, before returning a "not busy" indication to the invoking object. This timer allows the subscriber, who has become not busy, to immediately initiate an outgoing association.

If the subscriber is still "not busy" upon expiry of the "idle guard timer" then a "not busy" indication is returned to the object which invoked the "monitoring request" (see below for a discussion on the identity of the object which invoked the "monitoring request").

The monitoring/scanning may be stopped and the idle state entered if:

- A "cancel" request is received from the object which initiated the "monitoring request".
- The "maximum monitoring" timer expires.

Figure 5.14 raises a number of issues as follows:

1. Which object initiates the "monitor request" and "cancel request"?
2. What impact does the monitoring/scanning object have on the basic service objects?
3. Is it possible for the monitoring/scanning object to receive more than one monitor request for a given subscriber?

Consider the first question. The monitor request is initiated as a result of the originating subscriber's request to invoke the CCBS service. Since the invocation of the CCBS service involves the outgoing association object, the discussion of this issue is deferred to the following subsection.

The next issue concerns the impact of the monitoring/scanning object on the basic service objects. The manner in which the monitoring capability has been modelled is for the monitoring/scanning object to solicit the status of various basic service objects. This implies that the basic service objects must be able to accept, and respond to a "queryStatus" message. Whether this represents a new requirement introduced by the CCBS service or is an inherent requirement of the basic service definition could be more fully addressed with a further developed basic service model. As discussed in Section 5.1.2.1, it is likely that in a more fully developed basic service model, (e.g. one that takes into consideration management interactions), a management object may request the state or status of any object in the system.

Finally, consider the third issue. In the current CCBS service definition, it is possible for multiple CCBS requests to be queued against a particular subscriber. Appendix B specifies how these multiple CCBS requests are managed. Clearly, the behaviour of a monitoring/scanning object which supports multiple CCBS requests is more complex.

5.4.3 Expression of CCBS Delta Behaviour for Outgoing Association Object

The following subsections explore the "peer object" and "subclass" approaches to the specification of the additional behaviour introduced by the CCBS service on the outgoing association object.

5.4.3.1 Peer Object Specification

The decoupling of the "CCBS states" from the basic service states, shown in Figure 5.13, suggest that a new peer "CCBS delta" object can be specified which encapsulates the "CCBS delta state machine" and communicates with the basic service objects. This peer CCBS delta object would provide the "linkage" between the original outgoing association and an outgoing association which is established as the result of a CCBS recall.

Within this approach, the question arises as to how the transition from the "idle" to the "wait for user to invoke CCBS" state is realized. In Figure 5.13, this transition was caused by the receipt of a `releaseRequest` message from the association control object with a reason of busy. However, within the basic service description, the "releaseRequest" message is sent by the association control object to the "outgoing association" object. How is this message also sent to the peer "CCBS delta" object?

One alternative is to modify the behaviour of the transaction control so that it creates both an outgoing association object and a peer "CCBS delta" object upon receipt of a "setup" message from the subscriber. The association control object would also need to be modified in order to send the `releaseRequest` message to both the outgoing association object and to the peer "CCBS delta" object.

Alternatively, the behaviour of the outgoing association object could be modified such that upon receipt of a "releaseRequest" message with a reason of "busy", it performs the following:

- It dynamically creates a peer "CCBS delta" object instance; and
- It sends a "releaseRequest" message or an equivalent message (e.g. `startCCBS`) to the newly created "CCBS delta" object instance.

This latter alternative is preferred as the CCBS delta object is created only when it is required (i.e. on a busy condition).

The additional behaviour required by the outgoing association object in this latter alternative is shown in Figure 5.15. In this Figure a new "OA CCBS delta" class is defined as a subclass of the outgoing association class.

Before leaving this section, an issue outstanding from the previous section concerns the identity of the object which sends the "monitor request" to the monitoring/scanning object. The peer CCBS delta object would be responsible for sending this request upon receipt of "CCBS invoke request" from the subscriber in the "wait for user to invoke CCBS" state.

Figure 5.15: OA CCBS Delta(subclass approach)

After sending this request, the "CCBS delta object" would enter the "monitor for not busy" state and await a reply from the "monitoring/scanning" object.

5.4.3.2 CCBS Subclass Object Specification

It is also possible to express the delta behaviour introduced by the CCBS service by defining a CCBS delta object class as a subclass of the outgoing association object class. The CCBS delta subclass would encapsulate the behaviour of the CCBS delta component identified in Figure 5.13.

In this approach, it is assumed that the transaction control object would create an object instance of the CCBS delta class upon receipt of a setup request from the subscriber. This ensures that the additional behaviour required to support the CCBS service is available if this service is subsequently invoked on the association. Note that if the subscriber has not subscribed to the CCBS service, then an instance of the outgoing association object class will be created as in the basic service.

As discussed in Section 5.1.2.2 for the hold service, since the delta behaviour introduced by the CCBS *can* be expressed as a "separate" state machine, the resulting active object instance created from the "CCBS delta" subclass has slightly different semantics from an active object formed from the type of "supplementary service delta" subclass discussed in Chapter 4. That is, the resulting active object instance created from the "CCBS delta" subclass essentially incorporates "two state machines": the basic service state machine and the "ancillary" CCBS service state machine.

It is considered that the "outgoing association" and "CCBS service delta" are decoupled and therefore, there is merit in representing them as separate objects as in the "peer object" specification approach. Furthermore, the semantics of the outgoing association object have been modified in the "subclass specification" approach. Recall in the basic service definition, that an outgoing association object instance was created to handle an outgoing association and was terminated when the outgoing association was released. By extending the behaviour of the "outgoing association" in the "subclass" specification approach, the

new object instance (which incorporates both the outgoing association and the CCBS service delta state machines) terminates only when both the outgoing association is successfully released and the CCBS service is terminated.

5.4.4 CCBS Service Summary

The monitoring/scanning requirement of the CCBS service represented a new capability and could be specified within a new "CCBS monitoring/scanning" peer object. It was identified that CCBS service affects the behaviour of the outgoing association object. The impact on the outgoing association control object could be described as a set of states which are decoupled from the outgoing association states. This decoupling allowed both the "peer object" specification and "subclass" specification approaches to be exploited.

However, it was identified that the introduction of the "peer CCBS delta object" caused the behaviour of some of the basic service objects to be modified.

5.5 Combination of Services

Figure 5.16 summarizes the various "supplementary service delta" object classes introduced by the supplementary services analyzed in Section 5.1 to 5.4, inclusive and their relationships to the basic service object classes. In this Figure, only the "peer object" specification approaches have included as reflected by the present of the "peer" hold delta and "peer" CCBS delta object classes.

In combining the individual services, only the differences between the services, expressed in delta object classes, need to be analyzed as the common behaviour has been factored out.

This section examines the ability to specify combined services by the union of their individual service deltas. A combined service can be expressed as a simple "union" of the individual service deltas *if* a new combined service delta does not have to be specified.

Figure 5.16: Summary of Supplementary Service Delta Object Classes

Section 5.5.1 analyzes services that affect the same object, as reflected in the present of their corresponding subclass delta objects. Section 5.5.2 investigates the combination of services which are specified in terms of "service deltas" which are derived from different superclasses.

5.5.1 Combining Service Subclass Deltas Derived from Same SuperClass

This section focuses on combining services formed of subclass objects and examines whether the combined service can be expressed as a simple union of the individual service subclass deltas as shown in Figure 4.7.

5.5.1.1 CFU and CFB

Both the CFU and CFB services affect the behaviour of the transaction control object as reflected by the present of the "TC CFU delta" subclass and "TC CFB delta" subclass.

As identified in Figure 5.7, both the "TC CFU delta" and the "TC CFB delta" specify transitions for the handling of the "setupRequest" message from the association control (AC) object in the "wait for request" state.

The TC CFU delta specifies that upon receipt of the "setupRequest" message, a forwardingIndication should be sent to the invoking association control object if the CFU service is activated. The CFB delta specifies that upon receipt of the "setupRequest" message, a forwardingIndication should be sent to the invoking association control object if the maximum number of associations have been reached and the CFB service is activated.

The combined CFU and CFB service is specified such that the CFU service takes precedence over the CFB service. That is, if the CFU service is activated, the association will be forwarded regardless of the CFB activation state and the access condition. If the CFU service is not activated, then the association will be forwarded if the maximum associations have been reached and the CFB service is activated.

The combined CFU and CFB service can be specified as a union of the individual service deltas with a precedence relationship identified for the handling of the `setUpRequest` message. That is, the CFU service delta takes precedence over the CFB service delta in the combined specification.

5.5.1.2 Call Waiting and CFNR

The call waiting service and the CFNR service both affect the incoming association object as reflected by the presence of their corresponding service delta subclasses.

In the specification of the call waiting service delta identified in Figure 5.5 is compared with the specification of the IA CFNR service delta shown in Figure 5.9 it can be observed that both service deltas specify transitions for:

- The handling of the "alerting" message in the "wait for user response" state.
- The handling of the "connect" message in the user responded" state.

Consider the handling of the "alerting" message in the "wait for user response" state. The call waiting service delta specifies that a report message containing a call waiting indication should be sent to the association control object if the served user has subscribed to "call waiting notification" option. The CFNR delta specifies that a report message (without the call waiting indication) should be sent to the association control object and the CFNR timer should be started if the CFNR service is activated.

The combined call waiting and CFNR service is defined such that upon receipt of the alerting message the CFNR timer is started if the CFNR service is activated and a report message containing a call waiting indication is sent to the association control object if the user has subscribed to the "call waiting notification" option.

In this case, the combined service represents a "partial union" of the individual service deltas. That is, the output sent in combined service description depends only on the call

waiting delta (i.e. it overrides the output sent in the CFNR delta). Therefore, the combined service can not be specified as a simple union of the individual deltas but is a function of the individual deltas.

Now consider the handling of the "connect" message in the "wait for user response" state. The call waiting service delta specifies that a "seizeBchannel" message should be sent to the resource control object and the "call waiting, resource allocation" state should be entered. The CFNR delta specifies that the "cfnr timer" should be stopped and a setup confirm message should be sent to the association control object if the CFNR service is activated.

The combined call waiting and CFNR service is defined such that upon receipt of the connect message in the user responded state: the CFNR timer is stopped if the CFNR service is activated; a "seizeBchannel" message is sent to the resource control object; and the "call waiting, resource allocation" state is entered.

In this case, the combined service represents a "partial union" of the individual service deltas. That is, the output sent and the state entered in combined service description depends only on the call waiting delta (i.e. it overrides the output sent in the CFNR delta). Therefore, the combined service can not be specified as a simple union of the individual deltas but is a function of the individual deltas.

A new "combined service delta" can be specified which inherits the behaviour of the individual service deltas and provides the modified behaviour for the handling of the "common transitions" (e.g. receipt of the alerting and connect messages).

5.5.1.3 Call Waiting and CFB

The call waiting and CFU services both affect the behaviour of the incoming association object as identified by the presence of the "IA call waiting" delta and the "IA CFB" delta.

If the "call waiting" delta illustrated in Figure 5.5 is compared with the "IA CFB" delta of Figure 5.8, it can be observed that there is no "overlap" between transitions specified in the two deltas. Hence, there appears to be no apparent conflict in combining the service deltas to form the combined service.

CCITT has specified that there is no interactions between the two services. Therefore, the combined service can be specified as a simple union of the individual service deltas.

5.5.1.4 CCBS and CFU

The CCBS and CFU services both affect the behaviour of the outgoing association object as identified by the presence of the "OA CCBS" delta and the "OA CFU" delta.

If the "OA CCBS" delta illustrated in Figure 5.15 is compared with the "OA CFU" delta of Figure 5.10, it can be observed that there is no "overlap" between transitions specified in the two deltas. Hence, there appears to be no apparent conflict in combining the service deltas to form the combined service.

However, CCITT has specified the following interactions between the CFU and CCBS service: "If an association to user B is forwarded to user C by the CFU service and C is busy, then a CCBS request is made by user A (the originator of the association) shall be rejected because user A has not encountered a busy destination B".

This definition implies that the combined service cannot be specified in terms of a simple union of the "OA CFU delta" and "OA CCBS" delta. The combined service introduces new behaviour which must be specified in a new combined "OA CFU and CCBS delta". This combined delta would specify that upon receipt of a "releaseRequest" message from the outgoing association with a reason of "busy", *if a forwarding unconditional indication was previously received*, then a "CCBS delta" object should be created and a message should be sent to the CCBS delta object to indicate that any attempts to invoke CCBS should be rejected.

The exploration of other alternatives to express this interaction between the CFU and CCBS services without introducing a new combined service delta is an area for further study.

5.5.1.5 Summary of Service Combination

Four examples were illustrated in combining services formed of subclass objects. It was identified that for some service combinations, the combined behaviour could be expressed, as in Section 4.4.3, as a simple union of the individual service deltas or as a simple union of deltas with a specified precedence relationship.

In other cases, a new combined service delta had to be defined which inherited the behaviour of the individual service deltas and specified additional/modified behaviour specific to the combined service. There was a need to define a new combined service delta in the following two situations:

1. The individual service deltas specify "overlapping" transitions (i.e. transitions which relate to the same current state and input) and these transitions cannot be combined by either a simple union or precedence relationship.
2. The individual service deltas do not specify "overlapping" transitions, however, a transition in one service delta indirectly affects a transition in the other service delta. That is, the combined service introduces new behaviour which is not present in either delta.

5.5.2 Combining Service Deltas Derived from Different SuperClasses

This section focuses on service combinations involving "peer" supplementary service objects and examines whether the combined service can be expressed as a simple union of the individual service deltas (as shown in Figure 4.5 and 4.6).

5.5.2.1 Hold and Other Services

The hold service can be combined with any other service by the simple union of the individual service deltas. The "hold delta" peer object is such that it does not interact with any other service delta. The hold service also introduces a "TC hold delta" subclass object and a "AC hold delta" subclass object. Therefore, the only potential conflict is with the CF services which also define subclass deltas for the TC and AC objects. However, the nature of the additional behaviour introduced by the hold service is such that it is independent of the CF delta behaviour.

5.5.2.2 CCBS and CFU

The CCBS and CFU services both introduce peer objects to encapsulate additional functionality specific to the support of these services. That is, the peer "CCBS monitoring/scanning" delta object supports the monitoring/scanning requirement specific to the CCBS service and the "CF activation control" delta peer object supports the activation/deactivation requirements of the call forwarding services.

The question then arises as to whether the combined CCBS/CFU service can be expressed in terms of the presence of an active instance of each of the peer service delta objects (as shown in Figure 4.5). This requires that there are no interactions between the "CCBS monitoring/scanning" delta object and the "CF activation control" object. That is, the behaviour of the "CF activation control" object is the same regardless of whether there is an active instance of the "CCBS monitoring/scanning" delta object present or not.

An earlier proposal within CCITT specified that when the served user activated the CFU service, the monitoring/scanning of the served user (resulting from an earlier CCBS request from user A) would be cancelled. The implication of this proposal is that the behaviour of "CF activation control" object is modified when the "CCBS monitoring/scanning" delta object instance is present. Therefore, to specify the combined service, combined service deltas are required as shown in Figure 4.9.

Another proposal is for the monitoring/scanning to continue at the served user after the CFU service is activated. When the served user becomes not busy the resulting "recalled" association will be forwarded. Clearly this proposal allows the combined service to be specified as the union of the individual service deltas.

5.6 Summary of Service Analysis

In this chapter, the delta service specification approach outlined in Chapter 4 was applied to the specification of four supplementary services and their combinations. The following summarizes the results of this analysis:

- 1) The "partial state diagram" notation developed in Section 4.3.2 was considered sufficiently powerful for expressing the ways in which the supplementary services analyzed affected or modified the basic service state machine in the "sub-object" specification approach. It was considered difficult to express the class inheritance, object instance and communication relationships in one compact diagram. Therefore, the supplementary service delta objects were generally represented only in terms of their *class relationships* to the basic service objects.
- 2) It was shown that the "peer object" specification approach could be applied in the following two cases:

Case a: The impact of the service on a basic service object could be modelled as a set of states which are decoupled from the basic service object states. That is, the additional behaviour introduced by the service could be expressed as an "orthogonal component" to the basic service state machine.

Case b: The service introduced new functionality not supported in the basic service model. For example, the CCBS service introduced the requirement to support a new "monitoring/scanning" function. The call forwarding services identify the requirement to support a service activation/deactivation capability. This

new functionality/capability can be encapsulated in a new peer object which works autonomously and concurrently with the basic service objects.

- 3) When the "peer object" specification approach could be applied in case a above, it was identified that:
 - By introducing the "peer object" in the model, the behaviour of some of the basic service objects were affected. That is, the delta behavior could not be expressed, as in Chapter 4, by simply introducing a new peer object into the model. The need for the peer object to interact with the basic service objects resulted in the modification of behaviour of other objects within the model. Therefore, a *hybrid* "peer object" and "subclass object" specification approach was required.
 - A *pure* "subclass" specification approach could also be used to specify the additional behaviour introduced by the service.
- 4) In combining services which affect the same object, as reflected by the presence of their "service delta" subclasses, it was identified that:
 - When there is no overlap between the transitions specified in each service delta, the combined service can, generally, be specified as a simple union of the individual service deltas.
 - When the service deltas specify transitions which relate to the same "current state" and "input", the combined service can be specified as either:
 - a) A simple union of the individual service deltas with a "precedence" relationship governing which service delta subclass should be "visited" or looked at first.

- b) A new "combined service delta" class which inherits the behaviour of the individual service deltas and provides additional/modified behaviour for the "common" transition (e.g. as shown in Figure 4.8 a).
- 5) In certain cases, there is no apparent conflict in combining service deltas (i.e. the services affect different objects or the services affect the same object but there is no overlap between service delta transitions), however, the combined service specification introduces new behaviour which is not present in either delta. In these cases, a new combined service delta must be specified.

The analysis of this chapter has raised a number of open issues which may form the basis of further research. These issues are summarized below:

- 1) Are the visual formalisms developed in Chapter 4 sufficiently flexible, unambiguous and powerful to support the concepts of the delta specification approach? For example, can the "partial state diagram" notation express all possible ways in which the supplementary service may affect or modify the basic service state machine? Are there more concise or compact alternatives for expressing the class, instance and communication relations in the *peer object* and *sub-object* approaches to the specification of delta behaviour?
- 3) Is a "pure" *peer object specification approach* a valid proposition? That is, by introducing "peer" objects in the model which communicate with basic service objects has the behaviour of the basic service objects to support this communication not been affected? To what extent has the behaviour of the basic service objects been modified? Is there always a requirement to model this "modified behaviour" in terms of a new subclass delta object? Or can it be a matter of making the basic service objects more general and versatile such that a new delta basic service subclass does not have to be introduced in conjunction with the "peer" object?
- 3) Can the conditions for conflict between services be more formally stated? For example, it was identified that when two service deltas have "overlapping"

transitions and these transitions cannot be combined in a simple union relationship, a new combined service delta had to be defined which contained the resulting transition. However, the resulting transition is derived from (i.e. is a function of) the individual service deltas; it does not specify any additional behaviour which is not present in either delta. If the combined service can be expressed as a function of the individual services deltas, then is this a "conflict", requiring the specification of a combined delta? The way in which the individual service deltas were expressed within this chapter, any function other than a "union" required the specification of a new combined service delta.

CHAPTER 6

Telecommunication Service Specification and Validation using MLog

This chapter presents an "executable" specification of the basic telecommunication service and a number of supplementary services. This specification was developed using the MLog (Multitasking Object-Oriented Prolog) programming language and is subsequently referred to as the MLog "telecommunication service" prototype.

Section 6.1 provides a brief introduction to MLog. For a more comprehensive discussion, the reader is referred to [KARA 89a, KARA 89b].

Section 6.2 provides an overview of the approach taken to map the basic telecommunication service object model developed in Chapter 3 and the supplementary service delta objects identified in Chapter 5 to a MLog prototype.

Section 6.3 identifies the scope of the MLog telecommunication service prototype. Section 6.4 identifies the MLog Object Classes which constitute the basic and supplementary service specifications. Section 6.5 describes the runtime behaviour of the prototype. Section 6.6 discusses the mapping of the state machine descriptions developed in Chapter 3 and Chapter 5 into logic programming rules.

The approach taken to validate the telecommunication service specifications is discussed in Section 6.7. Section 6.8 analyzes the MLog prototype as it relates to potential design issues. A summary of the chapter is provided in Section 6.9.

6.1 Introduction To MLog

MLog is a powerful programming language which combines (1) the packaging and inheritance features of object-oriented programming systems (OOPS), (2) the top-down resolution mechanism of Prolog, and (3) concurrency based on communicating sequential processes and the rendezvous mechanism for interprocess communication [KARA 89a].

6.1.1 MLog OOPS Roots

MLog's support of the OOPS concepts of object, class, meta-class, inheritance, methods, message-passing, and instance variables is based on Smalltalk [GOLD 85]. However, unlike Smalltalk, the following two types of objects are supported:

1. *passive* objects: which are sequential and not capable of autonomous behaviour. Messages sent to passive objects are executed as procedure calls.
2. *active* objects: which represent "large-grain" communicating sequential processes. Messages are sent to active objects using the rendezvous. A message from an active object *to itself* is executed as a procedure call. All objects specified in the MLog "telecommunication service" prototype represent active objects.

An object in MLog is an instance of a class. A class is a description of one or more similar objects. It defines the procedures (methods) and data format (instance variables) which are used by all instances of that class.

In MLog, specialization of classes is achieved by subclassing. A class is a *subclass* (specialization) of another class, its *superclass*. The top level class in MLog is called **object** and by definition has no superclass. A subclass inherits all of the methods and instance variables of its superclass. A subclass may add instance variables and methods that are appropriate to more specialized objects. In addition, a subclass may override the methods of a superclass. To assemble the complete description of a class, it is necessary to

look at its superclass, its superclass' superclass, and so forth, until the **object** class is reached.

In MLog, classes are defined using *class descriptions*. A class description is composed of at most the following four parts:

- the *class declaration* which indicates the start of a class description, the name of the class to which the description applies and the name of the superclass of the class.
- *instance variable declarations*: which specify the instance variable names associated with the class and their optional initial (compile-time) value. An object is created as an instance of the class with a private copy of the declared instance variables.
- *class methods*: which specify methods that are associated with the class and not with instances of the class. In MLog, like Smalltalk, classes are themselves objects. As such, classes must be an instance of a class. To accomplish this, every class is the sole instance of a *meta class*, and every meta class is an instance of class **metaclass**. For further details on the class/meta-class relationships, the reader is directed to [KARA 89b].
- *instance methods*: which specify the methods associated with instances of the class.

Sending a message to a class will cause it to execute one of its class methods, or an inherited class method. Sending a message to an object instance will cause it to execute one of its instance methods, or an inherited instance method.

A message is composed of an object identifier (the target), the method name and the method arguments. The syntax is: *object-method(args,...)*. The object identifier may be: 1) a symbolic name which is internally translated into an object pointer or 2) an instantiated variable containing a symbolic name or an object pointer. If the object identifier is not provided, then it is assumed to be **self**, the object calling the method.

Methods that are only ever to be called by **self** are referred to as *local methods*. The convention used for identifying local methods (this convention is not enforced by MLog) is to begin the method with a dollar symbol. For example, in the MLog telecommunication service prototype, the **transaction control** class defines a local method **\$setup/1**. This method is only ever to be called from a method defined within the **transaction control** class.

6.1.2 MLog Logic Programming Roots

Computation in MLog is based on the execution paradigm of conventional Prolog, where the object methods are actually Prolog procedures, and message-passing equates to solving goals [KARA 89a]. The syntax of a method in MLog is identical to that used in Edinburgh Prolog: the clause head (a message body) followed by **:-** (the neck) followed by a sequence of messages separated by commas, terminating with a period.

The clause head is the same as a Prolog head goal. That is, it identifies which messages can be recognized by this clause. As indicated in Section 6.1.1, each message has the form **object-method(arg, ..)** or simply **method(arg,...)**. If the object identifier is not provided then **self** is assumed.

The arguments of a method can be one of the following Prolog terms: a predicate variable, a symbol, integer, or structure (which includes the Prolog list).

In MLog, instance variables are represented explicitly and not by asserting and retracting facts [KARA 89a]. MLog supports a number of built-in methods to retrieve and update instance variables. As well, it provides a number of built-in methods which provide the functionality of Prolog built-in predicates.

As an example of an MLog method, consider the following method defined in the **transaction control** class, **setup/1**:

```

1      $setup([REFERENCE,CALLEDNUMBER]) :-
2          theValueOf(subscriberNum,SN),
3          provision-getSubscriberServices(SN,LISTOFSERVICES),
4          $determineOutgoingInstance(LISTOFSERVICES,OBJECTNAME),
5          myself(OID),
6          OBJECTNAME-new(PID,active,[tcId=OID,subscriberNum=SN,
                           reference=REFERENCE),
7          $incrementAssoc,
8          $addToRefList(REFERENCE,PID),
9          $storeMessage([PID,setup,CALLEDNUMBER]),
10         !.
```

In line 1, the clause head has the selector **\$setup** and one argument which is a structure. This structure corresponds to a Prolog list consisting of two variables which are instantiated when the **\$setup** method is called. In line 2, the built-in MLog method, **theValueOf/2**, is used to retrieve the value of an instance variable called **subscriberNum** into the variable **SN**. In line 3, a message is sent to an object with the symbolic name of "provision". In lines 4, 7, 8 and 9, a message is sent to self (no object identifier is included) in order to execute a local method defined by the transaction control class. In lines 2 and 5, a message is also sent to self in order to execute one of the MLog built-in methods (i.e. a method defined in the **object** class).

In line 6, a variable, **OBJECTNAME**, is used as a placeholder for the object identifier of the message; it is expected to be instantiated with an object identifier before the message is sent. In fact, in line 4, the variable **OBJECTNAME** is instantiated as a result of calling the local method **\$determineOutgoingInstance/2**.

Finally, the cut method (**!/0**) in line 10 has the same semantics as a cut in Prolog. That is, it is used to prevent exploring an alternative path for **setup/1** if the method which called **setup**

fails. An overview of backtracking and execution using active targets in MLog is provided in the next section.

6.1.3 Concurrency and Backtracking in MLog

MLog uses the *rendezvous* [USDO 83] as the basis for communication between active objects. A sender object is blocked until the message is accepted and processed by a target object. A FIFO (first-in, first-out) queue of messages is used to buffer incoming messages from multiple senders.

A target object accepts a message by sending a **waitToAccept/1** message to itself. This special method has as its argument a term representing the message to be accepted: (1) its method name and arguments or (2) a single uninstantiated variable. The **waitToAccept/1** method will take the first message from the target object's message queue and apply the two steps of an MLog rendezvous [KARA 89a]:

1. *eligibility*: To pass this step, an incoming message must unify with the pattern defined in the **waitToAccept/1** argument. Once this unification has succeeded the method specified in the message is invoked by the target object.
2. *evaluation*: Provided the eligibility step is successful, the evaluation step will begin. In the evaluation step, the method head is unified with the result of the eligibility step and the corresponding method body is evaluated. If the method body evaluation is successful, then the rendezvous is completed, and the sender and target objects resume "concurrent" execution. That is, the sender continues to execute the next message with the resulting unifications. The target's message **waitToAccept/1** succeeds with the resulting unifications and the target continues to the next message. If a failure in a method body occurs, then normal Prolog backtracking will take place and alternate method clauses may be explored. If no method clause can succeed then the evaluation step fails.

If either the eligibility or evaluation steps fails, then the message will remain in the target's message queue and the next message in the queue will be tried. If no message is successful, then the target is blocked until a new message arrives. When a new message arrives, the first message in the queue will be retried so that a message that failed on one invocation of **waitToAccept/1** will be re-examined on the next invocation. Note that during the time that the target is (re-)trying the message or is suspended, the sender is also blocked.

In the MLog "telecommunication service" prototype, objects are set up so as to ensure that the two steps of the rendezvous succeed, and hence the **waitToAccept/1** method always succeeds.

In MLog, once a rendezvous has been successfully completed, there is no further relationship between the sender and target objects. If the sender backtracks over the message which caused the rendezvous, then the sender's variables bound during the rendezvous are undone. However, there is no effect on the target's variables which were bound during the rendezvous. Similarly, if the target backtracks over the **waitToAccept/1** message, then the target's variables are unbound but the sender's variables are not.

One last point to highlight is the concept of "in-rendezvous" and "post-rendezvous" message processing and the role of the **begin/0** method. When any active object is created, it automatically sends itself a **begin/0** message. The execution of the **begin/0** method, which results from sending the **begin/0** method, is used by an active object to represent its "main-line" program [KARA 89a].

The default **begin/0** method provided by MLog (i.e. provided in class **object**) is as follows (note that the percentage sign **%** when used outside of a quoted symbol denotes the start of a comment in MLog):

begin :-

repeat,

% this method always succeeds on backtracking

```

waitToAccept(X),    % see above discussion
fail.              % causes failure of the message

```

Thus the **begin/0** method specifies that any message pattern will be tried, all messages that pass the evaluation step will be processed (any message will pass the eligibility step because **X** is uninstantiated), backtracking will then occur and the process will be repeated.

During the evaluation step, the sender is blocked until the rendezvous is completed. In order to minimize the delay of the sender, any processing that does not explicitly require the blocking of the sender should be executed *after* the rendezvous is completed [KARA 89b]. This stage is referred to as post-rendezvous processing.

Post-rendezvous processing can be supported by redefining the **begin/0** method as follows:

```

begin :-
    repeat,
    waitToAccept(X),
    postRendezvous(X),
    fail.

method1(VAR) :-
    .....

postRendezvous(method1(VAR)) :-
    .....

```

An object which defines the above **begin/0** method will cycle endlessly between accepting and processing messages. A message **method1/1** accepted by this object will be passed, including its final bindings, to the **postRendezvous/1** method for any post-rendezvous processing.

Section 6.6 identifies the way in which the active objects defined in the MLog "telecommunication service" prototype use the concepts of in-rendezvous and post-rendezvous processing and the begin/O method.

6.1.4 Motivation for MLog

MLog was considered appropriate for specifying telecommunication services for the following reasons:

- Its OOPS features (i.e. support of objects, classes, inheritance and message-passing) are fundamental to the expression of the concepts of the object-oriented, delta specification approach.
- Its support for concurrency is consistent with the view that the basic service and peer "supplementary service delta" objects represent concurrent and autonomous "entities".
- Its logic programming features provide for very readable and concise state machine descriptions.

6.2 Overview of the Approach to Mapping Service Specifications To MLog

The mapping of the basic telecommunication service object model of Chapter 3 and the supplementary service delta specifications of Chapter 5 into a MLog prototype involved:

1. Translating the basic service and supplementary service delta objects into MLog Object Classes.
2. Translating the state machine descriptions into logic programming rules.

However, there were a number of issues to consider in the development of the MLog prototype. An overview of these issues and the approach taken to address them is provided in Sections 6.2.1 to 6.2.4.

6.2.1 Communication between Objects

The object model developed in Chapter 3 assumed an asynchronous mode of communication (i.e. non-blocking send) with unbounded FIFO queues. The implication of supporting the MLog rendezvous mechanism on the object model is twofold:

1. It introduces the possibility of deadlock;
2. It affects the state machine descriptions.

6.2.1.1 Potential for Deadlock

Within the basic telecommunication service model, messages can flow in both directions between the following objects:

- transaction control object and an outgoing association object
- transaction control object and an incoming association object
- transaction control object and an association control object
- association control object and an outgoing association object
- association control object and an incoming association object

Within a "rendezvous" environment direct communication between these objects may result in deadlock.

For example, consider an association between an outgoing association object and an incoming association object. This involves communication, in both directions, between:

- an outgoing association object and an association control object
- an incoming association object and an association control object

Recall from Chapter 3, that either end of the association (i.e. the incoming or outgoing association object) may initiate the disestablishment of the association by sending a "releaseIndication" message to the association control object. The association control object will then return a "releaseResponse" message to the end which initiated the release and will send a "releaseRequest" message to the other end.

Now consider, within the rendezvous mechanism of communication, the case where both ends "simultaneously" initiate the disestablishment of the association. Assume that the association control object first accepts the "releaseIndication" message from the outgoing association object.

As shown in Figure 6.1, the association control object will complete the rendezvous, unblocking the outgoing association object. As a result of processing the "releaseIndication" message, the association control object will send a "releaseResponse" message to the outgoing association object. When the rendezvous is completed with the outgoing association object, a "releaseRequest" message is sent to the incoming association object. The association control object is then blocked waiting for the incoming association object to accept the message. However, the incoming association object is blocked waiting for the association control object to accept the releaseIndication message.

In order to prevent such deadlock situations from arising, "transporter" and "message queue" objects were specified within the MLog prototype. Figure 6.2 illustrates the role of these objects in the provision of deadlock-free communications between the transaction control, outgoing association, incoming association and association control objects.

Associated with every transaction control object instance is a transporter object instance. The transporter object instance provides message pickup and delivery services for its transaction control object instance. The transporter object may deliver a message to an outgoing association object instance, an incoming association object instance or an association control object instance.

Figure 6.1: Potential Rendezvous Deadlock

Figure 6.2: Role of the Buffer and Transporter Objects

A single transporter object instance and a single message queue object instance are responsible for providing message pickup and delivery services for all association control object instances. As identified in Section 6.5, the "association control" transporter and message queue object instances are created upon initialization of the MLog prototype.

Association control object instances deposit their messages within the message queue object instance. The "association control" transporter object then picks up messages from the message queue object and delivers them to a transaction control object instance, an outgoing association object instance or an incoming association instance.

6.2.1.2 Impact on State Machines

Consider the outgoing association and incoming association state machine descriptions in Figure 3.10 and 3.14, respectively. An "outgoing resource allocation" and an "incoming resource allocation" state was specified in these state diagrams in order to wait for a response to a "seizeBchannel" message from the resource control object. In these states, it was assumed that messages from other objects could be received. For example, within the "outgoing resource allocation" state, a "disconnect", "release" or "releaseComplete" message could be received from the subscriber (see Figure 3.11).

However, in the MLog prototype, communication with the resource control object is defined such that it will return the channel identification as a parameter in the rendezvous reply. Therefore, the state machine descriptions are simplified in the MLog prototype as the "outgoing (incoming) resource allocation" state is not required by the outgoing (incoming) association object. In general, any state which was introduced in the state diagrams identified in Chapters 3 and 5 to wait for a response from the "resource control" object is not included in the MLog state machine descriptions.

The outgoing association state machine is further simplified in the MLog prototype in that the "establish AC identity" state is not required. That is, the "ACcreator" returns the association control identifier in reply to a "setupIndication" message rendezvous with the outgoing association object.

6.2.2 Expression of State Machines in Logic

There are a number of examples of state machines which have been expressed in terms of logic programming rules. In [URAL 85], a logic specification of the OSI Transport protocol entity (TPE) is provided which encompasses a collection of clauses defining the state transitions of the TPE.

In this example, a *single* Prolog procedure is defined which encompasses a collection of clauses defining the state transitions.

This approach, (i.e. specifying a single MLog method which encompasses all state transitions) could not be adopted in the MLog prototype due to the requirement to be able to modify "parts" of the state machine in the subclass specification approach. That is, supplementary services do not modify or override the complete behaviour of the basic service but introduce variations in the form of new states and modified/new transitions. In other words, there is a requirement to modify only some of the clauses or "state transitions" in the method.

The approach taken in this study for decomposing the state machine rules into "subsets" of modifiable rules is discussed in Section 6.2.2.1. Section 6.2.2.2 identifies how these "sets" of rules are modified by the subclass supplementary service delta objects.

6.2.2.1 Decomposition of State Machine Rules

Given the requirement for a finer "granularity" for specifying rules of behaviour, a different approach for specifying state machines in logic was taken. In this approach, a method (i.e. Prolog procedure) is defined whose sole purpose is to call other local methods based on the value of the current state of the object and/or input message. The method which calls the other local methods is generically referred to as the "objectStateMachine" method.

Depending on the degree of granularity required, these "other" local methods can represent

either:

1. The *set of transitions* which are associated with a *particular state*.
2. A particular transition in the object's state machine description.

In the first case, the method consists of a number of clauses or rules with the following form:

```
stateName(INPUT,NEXTSTATE,OUTPUT) :-
    conditions,
    actions.
```

The method selector, `stateName`, corresponds to the state name in the object state transition diagrams of Chapter 3 and 5. The `INPUT` identifies a particular message and associated parameters which may be received in this state. The `NEXTSTATE` specifies the new state of the object after the execution of the transition. The `OUTPUT` specifies the messages which are sent by the object if the transition is successful. The conditions is the set of predicates which must hold for the transition to take place. The actions are the actions associated with the transitions such as the updating of any instance variables.

As an example, consider the state transition diagram of the association control object shown in Figure 3.16 and in particular, the "wait for responder id" state. There are three transitions associated with this state corresponding to the receipt of the following three messages: "objectId", "releaseIndication" and "rejectIndication".

Within the MLog "telecommunication service" prototype, the association control object defines a method which specifies the transitions associated with this state as follows:

```
waitForResponderId([objectId,RESPONDERID],waitForSetupConfirm, []) :-
    makeTheValueOf(responderId,RESPONDERID), !.
```

Micro-D

```

waitForResponderId([releaseIndication,[OID,REASONI_]],releasePending,
  [[OID,releaseResponse,_]]) :-
  theValueOf(invokerId,OID), !.

```

```

waitForResponderId([rejectIndication,[OID,REASONI_]],waitForReleaseConfirm,
  [[INVOKER,releaseRequest,_]]) :-
  theValueOf(invoker,INVOKER), !.

```

In the above method, invoker and responderId are instance variables which store the identity of the outgoing association object instance and the incoming association object instance, respectively.

Thus the state diagrams could be easily mapped into logic rules by defining a method for each state with method clauses representing the state transitions associated with that state.

However, in some cases the same transition is specified for a given input for a number of states. This is represented in the statechart formalism by showing a transition leaving a super-state (this corresponds to the same transition leaving *all* substates). For example, in Figure 3.14 (incoming association state transition diagram), a number of transitions are specified leaving the superstate "incoming association establishment". These transitions have been expressed within the MLog "telecommunication service" prototype by defining a method corresponding to each transition as follows:

```

transitionName(PARAMETERS,NEWSTATE,OUTPUT) :
  conditions,
  actions.

```

The method selector, transitionName, is a name assigned to a transition from a specific state. This name may correspond to the name of the input message. The PARAMETERS identifies any parameters included in the input message. The NEXTSTATE specifies the new state of the object after the execution of the transition. The OUTPUT specifies the messages which are sent by the object if the transition is successful. The conditions is the

set of predicates which must hold for the transition to take place. The actions are the actions associated with the transitions such as the updating of any instance variables.

For example, consider the transition specified Figure 3.14 leading from the incoming association establishment superstate to an association disestablishment sub-state upon receipt of a "release" message from the subscriber. This transition is translated into the following method in the MLog prototype:

```
disestablishOnRelease(REASON,releaseIndication,
    [[ACID,releaseIndication,[OID,REASON]]]) :-
    myself(OID),
    theValueOf(acId,ACID).
```

6.2.2.2 Modification of Rules by Subclasses

Given the approach for decomposing the state machine rules into "groups" of transitions, how do the various subclasses extend or modify these rules to support the delta behaviour introduced by the various supplementary services?

Recall from Chapter 5, that the subclass supplementary service deltas may extend or modify the behaviour of the superclasses as follows:

1. They may introduce new states.
2. They may introduce new transitions, override transitions or extend transitions.

New states can be introduced by extending the "objectStateMachine" method.(see Section 6.2.2.1) in the subclass as follows:

```
objectStateMachine(newState,INPUT,NEXTSTATE,OUTPUT) :-
    newState(INPUT,NEXTSTATE,OUTPUT), !.
```

```

objectStateMachine(CURRENTSTATE,INPUT,NEXTSTATE,OUTPUT) :-
    super-objectStateMachine(CURRENTSTATE,
                              INPUT,NEXTSTATE,OUTPUT), !.

```

The subclass specifies a new method "newState/3" for handling all messages received within this "new state". The handling of messages in all other states is as specified by the superclass method `objectStateMachine`.

New transitions can be specified by the `newState/3` method clauses.

Extended transitions are handled also by the notion of the super construct. Transitions can be overridden by providing a new method in the subclass for the super's message body. Examples of how the subclasses modify the rules of the superclasses are provided in Section 6.6.

6.2.3 Runtime Specification

The basic telecommunication service and the supplementary service delta specifications presented in Chapter 3 and 5, respectively, represent "static" specifications. That is, they specify the service in terms of object classes and subclasses, and the behaviour encapsulated by each object class. In addition to addressing the "static specification", the MLog prototype provides a dynamic or runtime specification of the service. That is, it describes how and when object instances are created from these classes to provide the desired behaviour.

Consider for example the transaction control object. In the basic service description, a transaction control object was defined which was responsible for controlling and coordinating the various types of transactions across the subscriber access. A "transaction control" object instance was assumed to exist for each subscriber access.

In Chapter 5, a number of "supplementary service delta" objects were defined as subclasses of the transaction control object. For example, a "TC hold delta" object class

was defined. A "TC hold delta" object instance (instead of a transaction control object instance) was assumed to exist for those subscribers that have subscribed to the hold service.

The conditions under which a basic service object instance (e.g. transaction control) versus a supplementary service delta object instance (e.g. TC hold delta) is created must be specified by the MLog prototype.

The approach taken in the MLog prototype is to create a "transaction control" object instance when a new subscriber is defined. When the subscriber subscribes to a supplementary service, the currently active transaction control object instance is deleted and replaced with a newly created supplementary service delta object instance (e.g. a TC hold delta object).

Sections 6.4 and 6.5 identify the approach taken in the MLog prototype for creating the appropriate object instances to support the required basic and supplementary services.

6.2.4 Expression of Combined Deltas

In Chapter 5, the concept of combining services by the combining their individual service deltas was explored. It was shown that in some cases the combined service could be specified in terms of a simple union of the individual service deltas. In other cases, a new "combined service delta" was required which inherited the behaviour of the individual service deltas and provided additional/modified behaviour specific to the combined description.

This raises the issue of how the combined service deltas can be expressed within the MLog prototype. In some OOPS (e.g. Eiffel [MEYE 88]), the concept of multiple inheritance is supported which allows behaviour from multiple superclasses to be inherited. The mechanisms for supporting multiple inheritance vary from one OOPS to another particularly in their stand towards resolving ambiguities in variable and method name conflicts.

The version of MLog used in the development of the "telecommunication service" prototype did not support multiple inheritance. The approach taken to supporting combined services within the single-inheritance scheme was to define new object classes which duplicated the information from other classes.

There are other alternatives which could have been explored. For example, MLog provides the ability to modify (add/remove) methods of a class. This capability could have been exploited to create new classes (i.e. a combined class) from existing classes (individual classes). Clearly, this approach is more complex.

6.3 Scope of MLog Telecommunication Service Prototype

As discussed in the previous section, the MLog telecommunication service prototype goes beyond the state machine specifications of Chapters 3 and 5 to address the following:

1. The mechanisms for providing deadlock-free communication between objects in a rendezvous environment.
2. How and when object instances are created from object classes to provide the desired behaviour.

Further, the MLog telecommunication service prototype implements a number of capabilities not defined in the state machine specifications of Chapters 3 and 5 as follows:

1. The prototype implements a simple "routing function" (see Section 6.4.1.8) in order to establish an association between an "outgoing association" and "incoming association" object. In the Bearer Association Control state diagram (see Figure 3.16), "routing failed" and "routing passed" conditions are specified with the latter condition providing the identity of the destination Transaction Control object. However, the mechanism by which the "routing failed" and "routing passed" conditions are determined is not modelled in Chapter 3.

2. The prototype specifies how transaction identifies or *references* are assigned to subscriber service requests by the Transaction Control object (see Section 6.4.1.3). In the Transaction Control state diagram (see Figure 3.9), a "reference" was shown as a parameter in the messages sent/received by the Transaction Control object. However, the mechanism by which the references are managed by the Transaction Control object is not specified in Chapter 3.
3. The handling of unrecognized or unexpected messages in the state machines.
4. The prototype specifies how the timer services are supported (see Section 6.4.1.9).

The following summarizes the scope of the MLog telecommunication service prototype as it relates to the object classes and state machine specifications of Chapters 3 and 5:

1. The MLog telecommunication service prototype is a prototype of classes required to support services between two subscribers served by the *same local exchange*. Hence, it only supports associations between an "outgoing association" object and "incoming association" object as shown in Figure 3.8a.
2. It supports only a subset of the supplementary services and their combinations specified in Chapter 5. The supplementary services supported by the MLog prototype are: hold service, call waiting, CFU, CFB and CFNR. The CCBS service is not specified. The following combination of services are supported:
 - CFU and CFB.
 - hold service and call waiting
 - call waiting and CFNR.
3. The control of the "switch path" by the Association Control object has not been specified in the MLog prototype.

4. Some of the functionality of the supplementary services described in the state machine specifications of Chapter 5 has not been implemented, such as the following:
- The sending of a hold notification to the remote subscriber
 - The sending of a call forwarding notification to the originating subscriber.

The MLog telecommunication service prototype is not intended to represent a complete and comprehensive specification of the basic telecommunication service and a number of supplementary services. Rather it provides an executable specification that can be validated which in turn provides confidence in the graphical state machine specifications developed in Chapter 3 and 5. The prototype also provides the vehicle for exploring the ability of an object-oriented language, specifically MLog, to express the concepts of the object-oriented delta specification approach. Limiting the scope of MLog telecommunication service prototype as described above has not in any way impeded the above objectives to be realized.

6.4 MLog Object Classes

This section defines the MLog Object Classes which comprise the basic and supplementary service specifications.

6.4.1 Basic Service Object Classes

This section defines the MLog Object Classes which comprise the basic telecommunication service specification. The objects identified in the basic service telecommunication model of Chapter 3 are mapped to the corresponding MLog object classes.

6.4.1.1 Object

As identified in Section 6.1.1, the MLog top-level class is called **object** (i.e. it has the special superclass of **nil**). All MLog basic service objects inherit from class **object**.

The **object** class has been extended in the prototype to include a number of methods which are common to all basic service objects. Methods are defined to manipulate lists, the primary data structures used in the MLog prototype and to support inter-object communication.

6.4.1.2 Main

A MLog convention that has evolved is for every application to have a class **main** which is used to construct the key active and passive objects of the prototype [KARA 89b]. This approach has been followed in the development of the "telecommunications service" prototype.

The **main** object specifies the basic service objects which should be created upon initialization of the MLog telecommunication service prototype. The role of the main object in supporting the run-time specification is described in Section 6.5.

6.4.1.3 Transaction Control

The **transaction control** class maps to the transaction control object identified in the basic service model of Chapter 3. An instance of this class exists for every basic service subscriber. Section 6.5 describes how instances of the transaction control class are created.

In addition to specifying the behaviour described in Figure 3.9, the transaction control class defines instance methods for:

- 1) **Allocating/Deallocating "references" to incoming associations.**

Recall that a subscriber access may support multiple simultaneous transactions. Therefore, every message sent to/received from a subscriber must contain a transaction identifier or *reference*. The setup message sent from the subscriber to the transaction

control object contains a *reference* which *has been allocated by the subscriber* and uniquely identifies a particular outgoing association request. All subsequent messages which refer to this outgoing association must contain this reference.

On incoming associations, the transaction control object is responsible for allocating a reference to the association. This reference will be included in the setup message sent to the subscriber. Since an outgoing (incoming) association object instance is created to handle a particular outgoing (incoming) association, there is a one-to-one mapping between a reference and an object instance. This mapping is maintained by the transaction control object in the instance variable: **referenceList**.

2) Routing Messages from the Subscriber Object

In Chapter 3, it was assumed that the subscriber object would send all messages relating to a particular association (with the exception of a setup message) directly to the appropriate incoming or outgoing association object.

In the MLog prototype all messages from the subscriber are sent to the transaction control object. With the exception of a setup message, all messages will be directly routed (i.e. the TC object does not process the message) to the appropriate incoming or outgoing instance. This routing is based on the reference included in the message and the information stored in the transaction control's instance variable **referenceList**.

The transaction control class specifies local methods **\$setup/1**, **\$setupRequest/1**, **\$transactionComplete/1** which correspond to the transitions defined in Figure 3.9 for the handling of a setup, setupRequest and transactionComplete message, respectively. These methods in turn call other local methods for performing the actions associated with each transition. For example, an **\$incrementAssoc/0** method is provided to increment the current number of associations (specified by the instance variable: **numberAssoc**).

Two methods which consider special attention are the **\$determineOutgoingInstance/2** and **\$determineIncomingInstance/2** local methods. These methods determine the

appropriate object instance to create to handle the outgoing/incoming association request. This determination is based on the services to which the subscriber has subscribed.

Recall from Chapter 5 that in order to accommodate the additional behaviour introduced by some supplementary services, "supplementary service delta" objects were defined as subclasses of the incoming and outgoing association objects. For example, a "call waiting" delta object was defined as a subclass of the "incoming association" object. If a subscriber has subscribed to the call waiting service, then the **\$determineIncomingInstance/2** method will identify that an instance of the "call waiting delta" class should be created to handle the incoming association request. This is to ensure that the additional behaviour required by the call waiting service is available if this service is subsequently invoked.

6.4.1.4 Subscriber Association

The **subscriber association** class specifies the behaviour which is common to both the **outgoing association** and **incoming association** classes (described below). The **subscriber association** class represents an "abstract" class since an instance of this class is never created.

The **subscriber association** class defines the following instance variables which are common to the outgoing and incoming association objects:

- **basicServiceState**: specifies the current state of the outgoing(incoming) object. It is initialized to "idle".
- **tcId**: specifies the object identifier of the transaction control object which created the outgoing(incoming) object instance. This variable is assigned when the object instance is created.
- **subscriberNum**: specifies the subscriber number to which the outgoing (incoming) association pertains. This variable is assigned when the object instance is created.

- **channelId:** specifies the channel identifier allocated to the outgoing(incoming) association. It is initialized to "noChannel".
- **reference:** specifies the reference associated with this particular outgoing(incoming) association. This variable is assigned when the object instance is created.
- **acId:** specifies the object identifier of the association control object to which the outgoing(incoming) object communicates. For the incoming association object this variable is assigned when the object instance is created. For the outgoing association object, this variable is assigned by the outgoing association instance as the result sending an "setupIndication" message to the association control class (see Section 6.4.1.5)

The subscriber association class also specifies the association disestablishment behaviour. As identified in Chapter 3, the association disestablishment state (see Figure 3.14), as well as a number of transitions leading into this state, are common to both the outgoing and incoming association objects.

The **queryBasicServiceState/1**, **seizeChannel/1** and **releaseChannel/0** methods are specified by the subscriber association object in order to support interactions with a peer hold delta object (see Section 5.1.2.1). The **seizeChannel/1** and **releaseChannel/0** methods can either be:

- Called by the outgoing (incoming) association object ; or,
- They may be invoked by the peer hold delta object sending a **seizeChannel/1** or **releaseChannel/0** message to the outgoing (incoming) association object.

The **outgoing association** and **incoming association** classes are subclasses of the subscriber association class. These objects specify the behaviour which is specific to the establishment of an outgoing (incoming) association respectively.

6.4.1.5 Association Control

The **association control** class can be mapped into both the "bearer association control" and "AC creator" objects identified in the basic service object model of Chapter 3.

The **association control** class specifies a class method: **setupIndication/4**, which supports the actions and output specified in the "setupIndication" transition of the "AC creator" state diagram of Figure 3.15. That is, when a **setupIndication** message is sent to the association control class it will:

- create an appropriate object instance to handle the association request,
- return the identity of the created instance to the sending object (which is an outgoing association object instance)
- send a "setupIndication" message to the created instance.

The **association control** specifies a number of instance methods which support the behaviour of the "bearer association control" object described in Figure 3.16. However, the control of the "switched path" has not been represented in the MLog prototype.

6.4.1.6 Service Provision

This object class provides instance methods to define new subscribers (**addSubscriber/3**) and to allow subscribers to be provided with different service subscriptions (**subscribeToService/3**). This object can be considered as encompassing some of the responsibilities of the "management" object specified in the basic service model of Chapter 3.

Only one instance of this class is present and is created upon initialization of the MLog prototype (see Section 6.5).

The list of subscribers that are currently defined are stored in the instance variable: **subscriberList**. The supplementary services to which the subscribers have subscribed, as well as the subscription parameters are stored in the **subscriptions** instance variable. The instance methods: **getSubscriberServices/2** and **getServiceParameters/3** allow other objects to access the subscription information.

6.4.1.7 Resource Control

This class maps to the "resource control" object defined in the basic service model. However, it only co-ordinates access to the subscriber B-channels since the control of "switched path" resource has not been included in the MLog prototype.

Only one instance of this class is present and is created upon initialization of the MLog prototype (see Section 6.5).

The **resource control** class defines an instance method to define a new subscriber access: **addAccess/1**. This method causes a new element to be added to the instance variable: **accessList**. This element consists of the subscriber access identifier followed by the states of the two B-channels (b1 and b2). When the element is added to the **accessList**, the state of b1 and b2 are initialized to "idle".

The "subscriber access identifier" is actually the subscriber number since it is assumed in the MLog prototype that only one subscriber number can be assigned to a subscriber access. Hence, there is a one-to-one correspondence between a subscriber access and a subscriber number.

The **seizeBchannel/3** and **releaseBchannel/2** methods cause a B-channel on a specified subscriber access to be allocated and deallocated to a particular object, respectively.

6.4.1.8 Directory Database

This object stores the mapping between a specified "key" and an object identifier for various object classes. It provides methods for other objects to define the mappings and to retrieve the object identifier for a given class and key.

Only one instance of this class is present and is created upon initialization of the MLog prototype (see Section 6.5).

This object provides support for the "routing" function required by the association control object. An instance variable: **transactionControl** is defined which stores the mapping between a subscriber number and the transaction control object instance which exists for that subscriber. When an association control instance receives a **setupIndication** message, it performs the "routing" function by obtaining from the directory object the identifier of the transaction control object associated with the "called" subscriber number.

There is no counterpart for this object in the basic service telecommunication model of Chapter 3. The model did not address how the routing function was accomplished.

6.4.1.9 Timer

This object supports the timer services required by other objects. It provides two class methods: **startTimer/3** and **stopTimer/1** which allow other objects to request that a timer be started and stopped, respectively. The **startTimer/3** class method causes an active instance of **timer** to be created and a **startTimer** message to be sent to that instance. The **stopTimer/1** class method causes the specified active **timer** instance to be deleted.

The **timer** object supports an instance method, **startTimer/2**, which causes the timer instance to delay for a specified period of time and upon completion of this timeout to send a **timerExpired/1** message to the specified object.

There is no counterpart for this object in the basic service telecommunication model of Chapter 3. As discussed in Chapter 3, it was assumed that the timer mechanism was an inherent feature or capability of the basic service objects.

6.4.1.10 Transporter and Message Queue

As identified in Section 6.2.1.1, these classes support message delivery services for other objects. The **message queue** class specifies two instance methods: **storeMessage/1** and **getMessage/1** which allow other objects to deposit and retrieve messages, respectively. These "messages" are stored in the instance variable: **myQueue**).

The **transporter** class provides message pickup and delivery services on behalf of a "sender" object (specified by the instance variable: **sender**).

6.4.1.11 Datafill

This object specifies the initial configuration for the system in terms of the subscribers which are defined and their service subscriptions. This object class supports a single class method, **loadSubscriberData/0**, which is called by the **main** object on the initialization of the MLog prototype. This method in turn calls methods provided by the **service provision** object to define new subscribers and their supplementary service subscriptions.

6.4.1.12 Subscriber

The **subscriber Receive** class supports a single instance method: **logMessage/5** which allows other objects to record the messages that they have "sent" to a subscriber object. Since a subscriber object is not explicitly modelled in the MLog prototype, this really means the message that "would have been sent" to a subscriber object if one was specified.

The **logMessage/5** method causes the MLog main console to output: 1) the messagetype, 2) the identity of the target subscriber, 3) the sending object class and instance; and 4) the parameters included in the message.

The **subscriber Send** class defines a number of methods which support different service scenarios. That is, it specifies one or more messages for invoking different services. These methods can be used in the testing and validation of the MLog prototype.

6.4.2 Supplementary Service Delta Object Classes

This section defines the MLog classes which comprise the hold, call forwarding and call waiting service specifications. The "supplementary service delta" objects identified in Chapter 5 are mapped to the corresponding MLog classes.

6.4.2.1 Supplementary Service Database

The **supplementary service database** class contains supplementary service parameters to which a number of objects require access. It is a subclass of the **object** class.

The supplementary service database class specifies the following instance variables:

- **numWaitingAssoc**: the number of associations that are currently waiting at a particular subscriber access. This has an initial value of 0.
- **cfunumber**: the number to which all associations incoming to a particular subscriber access should be forwarded. This has an initial value of "undefined".
- **cfbnumber**: the number to which associations incoming to a particular subscriber access encountering a busy condition should be forwarded. This has an initial value of "undefined".

- **cfnrnumber**: the number to which associations incoming to a particular subscriber access encountering a "no reply" condition should be forwarded. This has an initial value of "undefined".

The supplementary service database class specifies two instance methods for retrieving (i.e. **querySupplServiceParameter/2**) and updating (i.e. **updateSupplServiceParameter/2**) the above instance variables.

An instance of this class exists for every subscriber which was subscribed to a least one supplementary service. An instance is created by the **provision** object the first time a subscriber subscribes to a service (see Section 6.5).

The mapping between the subscriber number and the supplementary service database object identifier is stored in the **directory** object.

6.4.2.2 TC Hold

The **TC Hold** class maps into the "TC (Transaction Control) hold" delta object identified in Chapter 5. That is, it specifies the additional behaviour required by the transaction control object to support the hold service.

The **TC Hold** class is a subclass of the Transaction Control class. The **TC Hold** class provides additional methods to support the behaviour specified in the state diagram of Figure 5.3.

An instance of this object is created when a subscriber subscribes to the hold service. It replaces a currently active Transaction Control object instance (see Section 6.5).

6.4.2.3 Hold Service

The **Hold Service** class maps into the peer "hold" delta object identified in Chapter 5.

The **Hold Service** class is a subclass of the **object** class. The **Hold Service** class provides a number of methods to support the "hold state machine" description identified in Figure 5.1.

An instance of this class is created by a **TC Hold** object when a subscriber invokes the hold service on an incoming or outgoing association.

6.4.2.4 Service Activation Control Delta

The **Service Activation Control** class specifies the behaviour which is common to all services requiring the ability to activate/deactivate services. In Chapter 5, only one supplementary service, namely the call forwarding service, required the ability to "activate/deactivate" the service. However, other supplementary services could be specified also requiring this capability. Hence, the common "activation/deactivation" behaviour is encapsulated in the **Service Activation Control Class**.

The **Service Activation Control** is an abstract class since an instance of this class is never created.

The **CF Service Activation Control** class is a subclass of the **Service Activation Control** class. It specifies behaviour which is specific to activating/deactivating the call forwarding services. This class can be mapped into the "CF Activation Control" Delta object identified in Chapter 5.

An instance of the **CF Service Activation Control** class is created by a **TC "Call Forwarding Delta"** object (see 6.3.2.5) when a request is received from the subscriber to activate/deactivate the call forwarding service.

6.4.2.5 TC Call Forwarding Deltas

The **TC Call Forwarding** class specifies the behaviour which is common to the call forwarding services (CFU, CFB, CFNR).

The **TC Call Forwarding** class is a subclass of the **Transaction Control** class. It specifies methods for the handling of a "call forwarding activation/deactivation" request from the subscriber. It specifies the **forwardAssociation /2** method which is used in support of the CFU and CFB services to return a forwarding indication to the association control object.

The **TC CFU** and **TC CFB** classes are subclasses of the **TC Call Forwarding** class. The **TC CFU** and **TC CFB** classes map into the "TC CFU" delta object and "TC CFB" delta object, respectively, identified in Chapter 5. That is, the **TC CFU** (**TC CFB**) class specifies the additional behaviour required by the transaction control object when the CFU (CFB) service is invoked.

An instance of the **TC Call Forwarding** class is created when a subscriber subscribes to the call forwarding no reply service and has not subscribed to any other call forwarding services.

An instance of the **TC CFU** (**TC CFB**) class is created when a subscriber subscribes to the call forwarding unconditional (call forwarding busy) service. It replaces a currently active **Transaction Control** or **TC Call Forwarding** object instance (see Section 6.5).

The **TC CFU CFB** class is a subclass of the **TC CFU** class. This class specifies the additional behaviour introduced by the combined CFU and CFB services. This class inherits the methods of the **TC CFU** class and duplicates the methods found in the **TC CFB** class.

An instance of the **TC CFU CFB** class is created when a subscriber subscribes to the **CFU (CFB)** service and already subscribed to the **CFB (CFU)** service. It replaces a currently active **TC CFU** or **TC CFB** instance.

6.4.2.6 AC Call Forwarding Deltas

The **AC CFU** class is a subclass of the **association control** class. This class maps into the "AC CFU" delta object specified in Chapter 5. That is, it specifies the additional behaviour required by the association control object to support the call forwarding unconditional service.

The **AC CFB** class is a subclass of the **AC CFU** class. This class maps into the "AC CFB" delta object specified in Chapter 5. As indicated in Chapter 5, this class is specified as a subclass of **AC CFU** as it introduces the same variations on the association control object as the **CFU** service, plus it adds some additional behaviour.

The **AC CFNR** class is a subclass of the **association control** class. This class maps into the "AC CFNR" delta object specified in Chapter 5. That is, it specifies the additional behaviour required by the association control object to support the **CFNR** service.

An instance of the **AC CFU**, **AC CFB** or **AC CFNR** class is created by the **association control** class upon receipt of **setupIndication** message (See Section 6.4.1.5.) The object instance which is created is determined based on the services to which the "called subscriber" has subscribed. The identity of the called subscriber is included in the **setupIndication** message.

6.4.2.7 IA Call Forwarding Deltas

The **CFB** class is a subclass of the incoming association class. It can be mapped into the "IA CFB" delta object identified in Chapter 5. It specifies the delta behaviour introduced by the call forwarding busy service on the incoming association object.

An instance of this object is created by a transaction control object on an incoming association request if the subscriber has subscribed to the call forwarding busy service.

The CFNR class is a subclass of the incoming association class. It can be mapped into the "IA CFNR" delta object identified in Chapter 5. It specifies the delta behaviour introduced by the call forwarding no reply service on the incoming association object.

An instance of this object is created by a transaction control object on an incoming association request if the subscriber has subscribed to the call forwarding no reply service.

6.4.2.8 Call Waiting Class

The Call Waiting class is a subclass of the incoming association object. It can be mapped into the "IA call waiting" delta object identified in Chapter 5. It specifies the delta behaviour introduced by the call waiting service on the incoming association object.

An instance of this object is created by a transaction control object on an incoming association request if the subscriber has subscribed to the call waiting service.

6.4.2.9 Call Waiting/CFNR Class

The Call Waiting/CFNR class is a subclass of the call waiting class. It specifies the combined delta behaviour, identified in Chapter 5, for the combined call waiting and CFNR services.

An instance of this object is created by a transaction control object on an incoming association request if the subscriber has subscribed to both the call waiting and CFNR services.

6.4.3 Summary

Figure 6.3 summarizes the supplementary service delta objects from Chapter 5 which have been included in the MLog prototype. These objects are shaded in the diagram. As indicated in Section 6.3, the CCBS service is not supported and therefore the CCBS-related object classes are not included in the prototype (i.e. CCBS Delta, CCBS monitoring/scanning delta and OA CCBS Delta). The "AC Hold Delta" is not included in the prototype as the control of the switch path is not modelling nor is the conveying of the hold notification. The "OA CFNR Delta" and the "OA CFU/B Delta" object classes are not included in the prototype as the receipt of the call forwarding notification by an outgoing association object has not been modelled.

6.5 MLog Prototype Run Time Behaviour

As identified in Section 6.4.1.2, the **main** class is responsible for creating a number of active object instances on initialization of the MLog prototype. Within the MLog support environment, the class description file containing the class descriptions are loaded by issuing the special MLog message **load/1** from the MLog console. An active instance of **main** is created by issuing the **run/0** command from the MLog console (see Appendix D.1).

When an active instance of the **main** object is created, it will:

- 1) create an active instance of **serviceProvision** using the symbolic name **provision**.
- 2) create an active instance of **resourceControl** using the symbolic name **resource**.
- 3) create an active instance of **directoryDatabase** using the symbolic name **directory**.
- 4) create an active instance of **messageQueue** using the symbolic name **associationQueue**.
- 5) create an active instance of **transporter** using no symbolic name and with a create-time assignment of the instance variable **sender** (**[sender=associationQueue]**).

- 6) create an active instance of **subscriberReceive** using the symbolic name **subscriber**.
- 7) create an active instance of **subscriberSend** using the symbolic name **test**.
- 8) send a **loadSubscriberData** message to the **datafill** class to provide the initial configuration for the system.
- 9) terminate.

The **datafill** class upon receipt of the **loadSubscriberData** message will:

- 1) Send a series of **addSubscriber** messages to the provision object instance specifying in each message the symbolic name of the subscriber, the subscriber's number and the maximum number of associations that the subscriber can simultaneously have across the subscriber access. The convention adopted for the symbolic name of the subscriber is: **s1, s2, ..., sn**.
- 2) Send a series of **subscribeToService** messages to the provision object instance specifying in each message the identity of the subscriber (i.e. subscriber's number), the identity of the supplementary service to which the subscriber wishes to subscribe and the subscription parameters for the service.

When the **provision** object receives an **addSubscriber** message from the **datafill** object class it will:

- 1) Create an active instance of **transaction control** using the symbolic name of the subscriber supplied as an argument in the **addSubscriber** message body. The instance variables **subscriberNum** and **maxAssoc** are assigned the values supplied as arguments in the **addSubscriber** message body.
- 2) Create an active instance of **transporter** using no symbolic name and with a create-time assignment of the instance variable **sender** (the value assigned to the **sender** variable is the symbolic name of the transaction control object instance).

This transporter object instance provides message delivery services for the newly created transaction control object instance.

- 3) Send a message to the **directory** object in order to define the mapping between the subscriber number and transaction control instance.
- 4) send a message to the **resource** object in order to define a new subscriber access.

When the **provision** object receives a **subscribeToService** message from the **datafill** object class it will:

- 1) Create a new "supplementary service database" object instance if this is the first service to which the subscriber has subscribed. The mapping between the subscriber number and the supplementary service database object identifier is stored in the **directory** object.
- 2) If the service affects the behaviour of the transaction control object, it will:
 - Delete the current **transaction control** and **transporter** object instances associated with the specified subscriber.
 - Create a new instance of the appropriate "TC supplementary service delta" object class which supports the additional behaviour introduced by the supplementary service. A new **transporter** instance is also created.

In summary, upon termination of the **main** object instance (i.e. upon completion of the MLog prototype initialization), there will be:

- A single active **provision**, **resource**, **directory**, **associationQueue**, **associationTransporter** (i.e. an active instance of transporter with a sender variable = **associationQueue**), **subscriber** and **test** object.

- Several active instances of the **transactionControl** and "TC supplementary service delta", **transporter** objects. For each subscriber defined there is one active instance of either a transaction control or TC supplementary service delta object class; and an active instance of transporter.

After the MLog prototype has been initialized, the MLog prototype can: (1) respond to subscriber's requests for basic and supplementary services and (2) respond to requests to add new subscribers and define new service subscriptions for existing subscribers.

New subscribers can be added and new service subscriptions can be defined by sending an **addSubscriber** and a **subscribeToService** message, respectively, from the MLog console to the **provision** object.

Basic and supplementary services can be invoked by either:

- Sending an appropriate message from the MLog console to the active **transactionControl** (TC) or "TC supplementary service delta" object instance responsible for a particular subscriber.
- Sending an appropriate message from the MLog console to the test object to invoke one of the predefined service scenarios. The test object will, in turn, send a message(s) to the appropriate TC or TC supplementary service delta object.

The MLog prototype will then dynamically create the appropriate object instances to support the requested service. For example, in response to a basic service request, the MLog prototype will create an outgoing association instance, association control instance and an incoming association instance.

6.6 Representation of Object Behaviour

This section describes how the temporal behaviour of the transaction control, incoming association, outgoing association and association control objects are represented in MLog.

The expression of the delta behaviour (encapsulated by the various "supplementary service delta" objects) within the MLog prototype is also discussed.

6.6.1 Transaction Control

The transaction control class re-defines the begin/0 method as follows:

```
begin :-
    repeat,
    waitToAccept(Message),
    postRendezvous(Message),
    fail.
```

When an active instance of the transaction control class is created, it sends itself a begin/0 message. This causes the object to cycle endlessly, accepting and processing messages.

When a "sendMessage/2" message is sent to a transaction control object it causes the sendMessage/2 method to be invoked. This method is defined as an extension to the object class as follows:

```
sendMessage(MessageType,Parameters) :- succeed.
```

Thus the rendezvous is used simply to provide the "application message" (i.e. "MessageType" and "Parameters" provided as arguments in the sendMessage method) to the transaction control object. All processing of the "application message" is performed "postRendezvous".

PostRendezvous processing of the "application message" consists of decoding the message using the \$decodeMessage/2 method and calling the appropriate local method to handle the specific message such as:

```
$decodeMessage(setup,PARAMETERS) :- $setup(PARAMETERS), !.
```

As identified in Section 6.4.1.3. the **\$setup/1** method defines the handling of the "setup" message in the state diagram of Figure 3.9. There is a method defined for every "application message" that can be received by the transaction control object as specified in the state diagram of Figure 3.9.

6.6.2 TC Service Deltas

Now consider how the delta behaviour introduced by the CFU, CFB and Hold services on the Transaction Control object are expressed in the MLog prototype.

In Figure 5.7, a new transition was defined in the TC CFU Delta which expressed the extended handling of a **setupRequest** message from an association control object.

This transition can be represented in the MLog TC CFU class by extending the **\$setupRequest/1** method defined in the transaction control object as follows:

```
$setupRequest(PARAMETERS) :-
    $forwardAssociation, !.
```

```
$setupRequest(PARAMETERS) :- super-$setupRequest(PARAMETERS).
```

This specifies that sending a **setupRequest** message to a TC CFU object will cause it to invoke a local **\$forwardAssociation** method. If this method fails (i.e. call forwarding unconditional is not activated) then the **\$setupRequest** method of the "super" (the transaction control object) will be invoked.

This approach has also been used in the TC Hold class for extending the **\$transactionComplete/1** and **\$decodeMessage/2** methods of the transaction control class. The **\$decodeMessage/2** method has been extended to allow processing of messages which are specific to the hold service (i.e. **holdTransactionComplete**, **hold**, **retrieve**).

Thus a message sent to the TC Hold object, which is a hold service-specific message, will cause a new method defined within the TC Hold class to be invoked. If the message is not hold-specific, then the methods defined in the "super" class (transaction control) will be invoked.

Therefore, the new transitions specified in the various TC supplementary service deltas of Chapter 5 can be easily supported in the MLog prototype by specifying *within the subclass* additional methods and extended behaviour for the superclass methods (using the "send to super" construct) .

6.6.3 Outgoing Association, Incoming Association and Association Control

The outgoing association, incoming association and association control objects are all similar in that instances of their class are dynamically created and terminate when the association is released.

These classes re-define the **begin/0** method as follows:

```
begin :-
    repeat,
    waitToAccept(Message),
    postRendezvous(Message),
    theValueOf(objectState,terminated).
```

When an active instance of one of these classes is created, it sends itself a **begin/0** message. This causes a repeat loop similar to the transaction control object to be entered with the exception that the looping will be terminated (and the object instance), when the object enters the "terminated" state.

The postRendezvous method defined in the outgoing association, incoming association and association control classes has the following form:

```
postRendezvous(sendMessage(MessageType,Parameters)) :-
    theValueOf(objectState,CURRENTSTATE),
    objectStateMachine(CURRENTSTATE,[MessageType,Parameters]
        ,NEXTSTATE,OUTPUT),
    makeTheValueOf(objectState,NEXTSTATE),
    outputMessage(OUTPUT).
```

The postRendezvous method calls a local "objectStateMachine" method with the current state of the object (stored as an instance variable) and the received input (i.e. received MessageType and Parameters). The object state will then be set to the NEXTSTATE instantiated by the objectStateMachine method. One or more messages may be send according to the OUTPUT instantiated by the objectStateMachine method.

The "objectStateMachine" method in turn calls other methods depending on the currentstate and/or input.

For example, the state machine method defined as part of the incoming association object will call the idle/3 method for the following values of currentState:

```
stateMachine(idle,[MessageType,Parameters],NEXTSTATE,OUTPUT) :-
    idle([MessageType,Parameters,NEXTSTATE,OUTPUT]).
```

The idle method specifies the rules for handling the receipt of a setupRequest message in the idle state. There are two rules defined:

```
idle([setupRequest,CallingNumber],waitForUserResponse,[]) :-
    seizeChannel(CHANNELID),
    CHANNELID \== noChannel,
    $performEstablishIncomingAssocActions, !.
```

```

idle([setupRequest,CallingNumber],waitForUserReleaseResponse,
    [[ACID,releaseIndication,[OID,noChannelAvailable]]) :-
    myself(OID),
    theValueOf(acId,ACID).

```

The first rule states that if a channel can be allocated to the incoming association, then the "waitForUserResponse" state will be entered and the actions specified by the **\$performEstablishIncomingAssocActions/0** method. If the first rule fails (i.e. no channel available), then the second rule states that a releaseIndication should be sent to the association control object and the "waitForReleaseResponse" state is entered.

6.6.4 IA and AC Service Deltas

Now consider how the delta behaviour introduced by the various supplementary service on the incoming association and association control objects are expressed in the MLog prototype.

In the case where the service introduces a new state, the "objectStateMachine" method must be extended in the subclass. For example, the call **waiting** class extends the incoming association stateMachine method as follows:

```

stateMachine(cwReleaseInitiated,INPUT,NEXTSTATE,OUTPUT)
    :- cwReleaseInitiated(INPUT,NEXTSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,INPUT,NEXTSTATE,OUTPUT)
    :- super-stateMachine(CURRENTSTATE,INPUT,
        NEXTSTATE,OUTPUT)

```

In the case where the service introduces extended handling of an input in a specified state, then the appropriate method must be extended in the subclass using the "send to super"

construct. For example, the CFB class extends the handling of a releaseComplete message in the waitForUserResponse state.

In the case where the service overrides behaviour in the super, a new method is provided for the super's message body. For example, the call waiting class provides a new method (set of rules) for the incoming association idle/3 method.

6.7 Validation of the Telecommunication Services

The telecommunication service specifications were analyzed and debugged by "running" or executing the MLog prototype interactively. From the MLog console, an "external event" was generated (i.e. a message was send to an MLog "telecommunication service" object) and the results of this "event" was observed by 1) using the MLog trace facilities (i.e. the debug/0 method) and 2) viewing the output printed on the MLog console. For example, the outgoing association, incoming association and association control objects upon processing of an application message will print on the MLog console, the following:

- The application message type received.
- The current state of the object when the message was received.
- The next state of the object as the result of processing the message.

After the MLog prototype was transformed into the new status as a result of processing the external event, another event was generated and the response to this next event observed. This allowed a step-by-step validation of the transitions of the state machines described in Chapter 3 and 5.

Appendix C provides the MLog Telecommunication Service Prototype source code. Appendix D provides some example traces which were generated during the execution of the MLog Telecommunication Service Prototype.

6.8 Design Issues

This section provides a number of comments and observations regarding the MLog prototype as the basis for an ultimate design and implementation specification.

6.8.1 Number of Object Instances

The MLog prototype has adopted an instance-per-subscriber request approach. An outgoing (incoming) association is dedicated to handling a particular subscriber's outgoing (incoming) association request. An association control object instance is dedicated to handling a particular association between two objects. A "hold" instance is associated with a subscriber's hold request on a particular incoming (outgoing) association. A "call forwarding activation control" instance is dynamically created on every subscriber's call forwarding activation/deactivation request.

This approach contributed to the overall understanding of the specification due its simpler logic and data structures. However, this simplicity must be weighted against the overhead in supporting so many different object instances (i.e. tasks).

6.8.2 Interaction Architectures

There are a number of approaches to structuring two-way interactions (messages flowing in both directions) between two objects. As identified in Section 6.2.1.1, a transporter and a message queue object instance were used in the MLog prototype to provide message delivery services for the association control objects. A single transporter object was initially considered for providing this service. However, this approach was discarded since it introduced additional complexity required for:

- The transporter object to access a list of currently active association control object instances.
- The transporter object to be prevented from accessing a terminated association control object instance.

A discussion of different ways to structure two-way interactions and the advantages/disadvantage of each approach is provided in [Buhr 83].

6.8.3 Determination of Appropriate Service Object

In the MLog prototype, the transaction control object dynamically creates the appropriate object instance based on the nature of the service request and the services to which the subscriber has subscribed. The determination of the appropriate object instance is made on *every* incoming and outgoing association request. This introduces an overhead on every basic service request in order to:

- retrieve the list of services from the provision object. If the provision object was mapped on to a remote database node in the network, then this may represent a considerable overhead.
- process the list of services to determine which object should be created.

Alternatively, since a subscriber's subscriptions change relatively infrequently, the determination of the appropriate object instance can be performed only when a subscriber changes his/her subscriptions (e.g. when the subscriber subscribes to a new service or withdraws from a service). The appropriate class type could be stored by the transaction control object and then updated by the provision object when a change to the subscriber's service subscriptions is made.

6.8.4 Reconfiguring Behaviour

A "transaction control" object instance is created when a new subscriber is defined. Once defined, a subscriber can subscribe to, and withdraw from, different supplementary services. These actions (subscription and withdrawal) may result in different behaviour required by the transaction control object.

The approach taken in the MLog prototype is to delete the currently active transaction control object instance and replace it with a newly created supplementary service delta

object instance. It is recognized that this approach will cause any currently active transactions to be terminated.

Two other approaches for reconfiguring this behaviour were considered:

- Replace existing and/or add new methods to the transaction control object instance to reflect the new behaviour. In MLog, it is possible to modify the methods of a class but not individual object instances.
- Similar to the concept above, is to define instance variables which represent modifiable behaviour. When a new service is subscribed to, these instance variables can be updated to reflect the new behaviour. This approach requires "some hooks" to be built into the basic service description.

The approach adopted for reconfiguring the behaviour will depend on the requirements specified for maintaining current transactions.

6.8.5 Combined Services

Clearly without the support of some type of multiple inheritance in MLog, the specification of combined services becomes difficult. In the approach taken in the MLog prototype, information must be duplicated in several classes.

If multiple inheritance is supported by defining new classes from existing classes, then within the prototype there would need to specify all possible class combinations. For example, assuming supplementary delta service class 1, 2 and 3 can be combined without conflict (as a simple union of the individual classes), then there would be a need to specify four class declarations:

- suppl1/2: class of (suppl,suppl2).
- suppl1/3: class of (suppl,suppl3).
- suppl2/3: class of (suppl2,suppl3).

- **suppl1/2/3: class of (suppl,suppl2,suppl3).**

In determining the appropriate object instance to create, there would be a requirement for an algorithm which takes a subscriber's set of services and maps them into the appropriate class, as follows:

[suppl1,suppl2] -> suppl1/2

[suppl1,suppl3] -> suppl1/3

•

•.

As discussed in Section 6.8.3, the determination of the appropriate class could be performed on every request or when a change is made to a subscriber's subscriptions.

The issue arises as to whether the need to define every possible combined class declaration together with an algorithm to determine which class is appropriate for a given subscriber offsets the advantages of the delta specification approach. Ideally, a mechanism is needed which does not require a "static" description of all possible classes. Rather an object can be created with the behaviour specified by a list of non-conflicting service delta classes.

6.9 Summary of MLog Prototype

In summary, the basic telecommunication service object model of Chapter 3 and the supplementary service delta specifications of Chapter 5 were easily translated into an MLog prototype. This raises the issue as to whether the "graphical" class and state machine descriptions can be, in the future, "automatically" translated into an MLog prototype. The advantage of "automating" this translation is that it allows a visual representation of the specification to be mapped into an executable specification. The advantages of an executable specification is that it can be validated for consistency and completeness.

It was identified that the "peer" object approach could be more easily accommodated within the MLog prototype than the "subclass" object approach. In supporting the "subclass" object approach, a number of issues had to be addressed such as:

1. The requirement to reconfigure behaviour from, for example, an instance of a superclass (e.g. the basic service class) to an instance of a subclass (e.g. a supplementary service delta subclass). In the "peer" object approach, "reconfiguring" behaviour (e.g. adding in new behaviour) was achieved by creating an instance of the "peer" supplementary service delta class.
2. The requirement to determine the appropriate object instance (e.g. basic service object, supplementary service1 delta object, supplementary service2 delta object) to create to support a given service request.
3. The requirement to support combined subclass service deltas in a manageable way. A means to adequately address this requirement within MLog current capabilities was not resolved in this thesis. However, it was identified that a mechanism is needed which does not require a static declaration of all possible service combinations in terms of combined class declarations.

This chapter concludes with the following general question: "How does the object-oriented delta specification approach fit into a larger methodology involving the entire lifecycle?" For example, can the same paradigm, "object-orientation", be used in all phase of development (i.e. specification, design, and implementation) of telecommunication systems?

CHAPTER 7

Summary and Conclusions

7.1 Summary

In Chapter 1, the objectives of this thesis were stated and the achievements outlined. This chapter was followed by an introductory chapter which provided an overview of telecommunication services and their specification techniques.

In Chapter 3 an object-oriented model of the basic telecommunication service was developed. This model consisted of a number of concurrent and autonomous entities which interacted by sending messages over channels. The dynamic behaviour of the objects was then expressed using state diagrams with extensions provided by the statechart formalism.

Having developed a basic telecommunication service specification, the next Chapter looked at the problems of 1) specifying individual supplementary services 2) specifying combinations of supplementary services. The concept of defining "delta specification components" as the basic building blocks for forming specifications of individual and combined supplementary services was then introduced. These "delta specification components" express the "delta" or differential behaviour introduced by the supplementary services in terms of the additions and/or modifications to the basic service specification.

Further, Chapter 4 identified two approaches for accommodating the "delta specification components" within an object-oriented framework. In the *peer object specification approach*, an autonomous and concurrent "supplementary service delta" object is specified which communicates with the basic service object to form the supplementary service. In the *subclass object specification approach*, a "supplementary service delta" object is specified which forms the supplementary service via inheritance of the basic service

behaviour. A state diagram notation for expressing behaviour in the *subclass object specification* approach was also developed in this Chapter. Finally, Chapter 4 presented examples of how combined services could be represented within the object-oriented framework and raised the question of possible conflicts in combining services.

Chapter 5 applied the "delta specification approach" developed in Chapter 4 to the specification of a number of supplementary services and their combinations. It was shown that the "peer object" specification approach could be applied in the following two cases: 1) The supplementary service affected the behaviour of a basic service object. This impact could be expressed as an "orthogonal component" to the basic service object state machine; and, 2) The supplementary service introduced new functionality not supported in the basic service model. This functionality could be encapsulated in a new peer object which works autonomously and concurrently with the basic service objects.

It was also identified in Chapter 5 that introducing a peer object resulted in the modification of other objects within the model. Therefore, a *hybrid* "peer object" and "subclass object" specification approach was required. It was also possible to use a pure "subclass" object specification approach in place of this hybrid approach.

On the subject of combining services, Chapter 5 identified combinations of services which required the specification of additional "combined delta specification components".

In Chapter 6, the concepts of the object-oriented, delta specification approach were explored using the MLog programming language. The basic telecommunication service object model of Chapter 3 and the supplementary service delta specifications of Chapter 5 were translated into an executable specification: the MLog "telecommunication service" prototype. This mapping was shown to be relatively straightforward given MLog's support of OOPS features such as objects, classes and inheritance coupled with its support of concurrency and logic programming.

A number of issues were identified in Chapter 6 in expressing supplementary services formed of "subclass supplementary service" objects. The difficulties arose due to the

requirement to "reconfigure behaviour" from one class to another class (e.g. a subclass) as a subscriber's service subscriptions changed. Furthermore, the version of MLog used in the development of the prototype did not support the ability to inherit behaviour from multiple classes. This limited the options which could be exploited for expressing combined services.

7.2 Conclusions and Suggestions for Further Work

The object-oriented delta specification approach presented in this thesis provides a compact and structured way for defining services which are closely coupled to the basic service. Each new service does not have to be specified in its entirety, resulting in a number of specifications with overlapping behaviour. Only the additions and/or modifications to the basic service introduced by the new service are specified. The merits in having more compact specifications are considered as follows:

1. By not duplicating behaviour, it reduces the potential for inconsistencies between service specifications. Furthermore, a change or correction to the basic service specification is automatically reflected in the supplementary service specification.
2. It provides for a less complex service analysis problem. That is, the overall behaviour of one supplementary service does not have to be analyzed against the overall behaviour of another supplementary service to determine the resulting combined behaviour. Only the differences between the two services need to be analyzed as the common behaviour of the two services has been factored out.

The object-oriented delta specification approach presented in this thesis addresses to some extent the service combination problem. It concluded that whereas some services can be combined as a simple union of their individual "delta specification components", other combinations required the specification of additional "combined delta specification" components. However, these additional "combined delta specification" represent a more compact expression of combined service specifications. No conclusions were drawn regarding the larger question of whether individual delta service components can be

combined in such a way that does not required the specification of combined service components. This is one of the questions which is identified below as a possible area of further study.

This thesis has raised a number of interesting and open issues which may form the basis of further research. These issues are summarized below:

1. Are the visual formalisms developed in Chapter 4 sufficiently flexible, unambiguous and powerful to support the concepts of the delta specification approach? For example, can the "partial state diagram" notation express all possible ways in which the supplementary service may affect or modify the basic service state machine? Are there more concise alternatives for expressing the class, instance and communication relations in the *peer object* and *sub-object* approaches to the specification of delta behaviour?
2. Related to question 1 above, can the class and state machine descriptions be translated "automatically" into MLog? It was shown that the basic service model and the individual supplementary service delta specifications could be easily translated into an MLog prototype. The basic service and supplementary service delta objects were translated into MLog object classes. The state machine descriptions were translated into logic programming rules.

The advantage of "automating" this translation is that it allows a visual representation of the specification to be mapped into an executable specification. The advantages of an executable specification is that it can be validated for consistency and completeness.

3. Is a "pure" *peer object specification approach* a valid proposition? That is, by introducing "peer" objects in the model which communicate with basic service objects have we not changed the behaviour of the basic service objects to support this communication? To what extent have we modified the behaviour of the basic service objects? Is there always a requirement to model this "modified behaviour"

in terms of a new subclass delta object? Or can it be a matter of making the basic service objects more general and versatile such that a new delta basic service subclass does not have to be introduced in conjunction with the "peer" object?

4. On the question of combining services, can the conditions for conflict be more formally stated? For example, it was identified that when two service deltas have "overlapping" transitions and these transitions can not be combined in a simple union relationship, a new combined service delta had to be defined which contained the resulting transition. However, the resulting transition is derived from (i.e. is a function of) the individual service deltas; it does not specify any additional behaviour which is not present in either delta. If we can express the combined service as a function of the individual services deltas, then is this a "conflict", requiring the specification of a combined delta? The way in which the individual service deltas were expressed within this thesis, any function other than a "union" required the specification of a new combined service delta.
5. What capabilities must be supported by an object-oriented language to support the combination of service deltas in a manageable way? For example, a mechanism is required which does not require a static declaration of all possible service combinations in terms of combined class declarations.
6. Finally, how does the "object-oriented, delta" specification approach fit into a larger methodology involving the entire lifecycle? For example, can the same paradigm, "object-orientation", be used in all phases of development (i.e. specification, design and implementation) of telecommunication systems?

REFERENCES

- [APPE 86] Appenzeller, H.R., "Signalling System No. 7 ISDN User Part", *IEEE Journal on Selected Areas in Communication.*, vol. SAC-4, pp. 366-371, May 1986.
- [BELI 89] Belina, F. and Hogrefe, D., "The CCITT-Specification and Description Language SDL", *Computer Networks and ISDN Systems*, Vol. 16, pp. 311-341, (1989).
- [BELL 82] Bellamy, J., *Digital Telephony*, New York, NY: John Wiley & Sons, 1982.
- [BELL 89] BellSouth Services, "ISDN Call Forwarding", Contribution T1S1.1 89-417, Submitted to ANSI T1S1.1 Service Descriptions Group, September 18-22, 1989.
- [BUHR 84] Buhr, R.J.A., *System Design with Ada*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [CCITT 88a] CCITT Recommendation Z.100 : Specification and Description Language SDL, Blue Book, Volume VI.20-VI.24.
- [CCITT 88b] CCITT Recommendation Q.931: Digital Subscriber Signalling No. 1 (DSS 1), Network Layer , COM XI-R 174 to COM XI-R 177, June 1988 (To be published in the Blue Book).
- [CCITT 88c] CCITT Recommendations Q.761, Q.762, Q.763 and Q.764: Signalling System No.7 ISUP, COM XI-R 164 to 165, June 1988 (To be published in the Blue Book).
- [CCITT 88d] CCITT Recommendation Q.82: Functions and Information Flows for Services in the ISDN, Call Offering Supplementary Services, COM XI-R 149, June 1988 (To be published in the Blue Book).
- [CCITT 88e] CCITT Recommendation Q.83: Functions and Information Flows for Services in the ISDN, Call Completion Supplementary Services, COM XI-R 150, June 1988 (To be published in the Blue Book).

- [DAVI 79] Davis, A.M., et al., "RLP: An Automated Tool for the Processing of Requirements", *Proceedings of IEEE COMPSAC '79* (Chicago, Ill., November 5-8), pp. 289-299, IEEE Press, Wash., D.C., 1979.
- [DAVI 82] Davis, A. M., "The Design of a Family of Applications-Oriented Requirements Languages", pp.21-28, *IEEE Computing*, Vol. 15, No. 5, May 1988.
- [DAVI 88] Davis, A. M., "A Comparison of Techniques for the Specification of External System Behavior", pp.1098-1115 *Communications of the ACM*, Vol. 31, No. 9, September 1988.
- [FACI 89] Faci, M., Logrippo, L. and Stepien, B., *Formal Specifications of Telephone Systems in LOTOS*, TR-89-07, Protocols Research Group, Dept. of Computer Science, University of Ottawa, Ottawa, Ontario, February 1989.
- [GOLD 85] Goldberg, A. and Robson, D., *Smalltalk-80: The Language and Its Implementation*. Reading, MASS: Addison-Wesley, 1985.
- [HARE 87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, Vol.8, (1987).
- [HOAR 78] Hoare, C.A.R., "Communicating Sequential Processes", *Communication of the ACM*, Vol. 21, No.8, pp.666-677, Aug. 1978.
- [ILOG 87] *STATEMATE: The Languages of Statemate*, Technical Report, i-Logix Inc., Burlington, Mass. 1987..
- [ISO 87] ISO DIS9074 (1987), Estelle: A formal description technique based on an extended state transition model.
- [KARA 89a] Karam, G.M., *MLOG: A Language for Prototyping Concurrent Systems*, TR SCE-88-3, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, March 1989.
- [KARA 89b] Karam, G.M., *The MLog User Guide*, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, May 1989.
- [MEYE 88] Meyer, B., "Eiffel: Resuability and Reliability", *Software Reuse: Emerging Technology*, Computer Society Press, Wash., D.C., 1988.
- [MILL 88] Miller, M.J. and Ahamed, S.V., *Digital Transmission Systems and Networks*, Rockville, Maryland: Computer Science Press, Inc., 1988.

- [MOLL 87] Moller-Pedersen, B., Belsnes, D. and Dahle, H. P., "Rationale and Tutorial on OSDL: An Object-Oriented Extension of SDL", *Computer Networks and ISDN Systems*, Vol. 13, No. 2, pp. 97-117, (1987).
- [SGXVIII 89] CCITT Study Group XVIII, "Completion of Calls to Busy Subscriber," Joint Experts Meeting of Study Groups I and XVIII, Geneva, 9-13 January 1989, Temporary Document 10-E, Revision 1.
- [TANE 88] Tanenbaum, A.S., *Computer Networks*, Second Edition, Englewood Cliffs, NJ: Prentice Hall, 1988.
- [URAL 85] Ural, H., *Construction of Logic Specifications for Communications Protocols and Services*, TR-85-17, Dept. of CSI, University of Ottawa, Ottawa, Ontario, 1985.
- [USDO 83] U.S. Department of Defence, *Reference Manual for the Ada Programming Language* MIL-STD-1815a. U.S. Department of Defence, 1983.
- [WEGN 87] Wegner, P., "Dimensions of Object-Based Language Design", *Proceeding of the OOPSLA 87* (October), ACM Press, New York, N.Y., 1987.
- [WHIT 81] Whitis, V., and Chiang, W., "A State Machine Development Method for Call Processing Software", *Proceeding of the IEEE Electro 81* (Apr.) IEEE Press, Wash., D.C., 1981.
- [ZAVE 85] Zave, P., "A Distributed Alternative to Finite-State-Machine Specifications", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pp. 10-36, (January 1985).

APPENDIX A

List of Acronyms and Abbreviations

AC	Association Control
CCBS	Call Completion to Busy Subscriber
CCITT	International Telegraph and Telephone Consultative Committee
CF	Call Forwarding
CFB	Call Forwarding Busy
CFNR	Call Forwarding No Reply
CFU	Call Forwarding Unconditional
CSP	Communicating Sequential Processes
FSM	Finite State Machine
IA	Incoming Association
ISDN	Integrated Services Digital Network
ISO	International Standards Organization
JSD	Jackson Structured Development
JSP	Jackson Structured Programming
LOTOS	Language of Temporal Ordering Specifications
MLog	Multitasking Object-Oriented Prolog
NDUB	Network Determined User Busy
OA	Outgoing Association
OID	Object Identifier
OOPS	Object-Oriented Programming Systems
OSDL	Object-Oriented Specification and Description Language
OSI	Open Systems Interconnection
POTS	Plain Old Telephone Service

RLP	Requirements Language Processor
SDL	Specification and Description Language
TC	Transaction Control
UDUB	User Determined User Busy

APPENDIX B

Supplementary Service Prose Descriptions

This Appendix provides a prose description for each of the supplementary services analyzed in this thesis. Aspects of the service operation which were not explicitly considered in the development of the delta supplementary service specifications are highlighted.

The prose descriptions are derived from the ISDN supplementary service descriptions currently being progressed within the international (i.e. CCITT) and national (i.e ANSI - American National Standards Institute) standards committees.

B.1 Hold Service

The hold service [CCITT 88d] allows a subscriber to interrupt communications on an existing association, place the association on hold, and make the B-channel (bearer channel) available for another association. A B-channel may or may not be reserved after the communication is interrupted to allow the origination or possible termination of other association(s). The Hold service includes a retrieve function, which reestablishes communication on a B-channel between the served user and the held party.

B.1.1 Provision/Withdrawal

The hold service is provided after prearrangement with the service provider. The service can be offered with the subscription options described in Table B.1.

Table B.1: Hold Service Subscription Options

<u>Subscription Options</u>	<u>Value</u>
Held user receives notification that association has been held and the held association has been retrieved.	- NO - YES
Reservation of B-channel when association placed on hold.	- NO - YES

B.1.2 Activation/Deactivation

Activation of the Hold service should be by subscription and deactivation, by withdrawal.

B.1.3 Normal Operation

B.1.3.1 Hold Service Invocation

A subscriber invokes the hold service by sending a hold message to the network service provider. The hold message contains an identifier of the incoming or outgoing association to which the hold request applies.

Upon receipt of this message by the network service provider: the association will be placed on hold; the B-channel allocated to the association will be released; and a hold acknowledge message will be returned to the subscriber, providing the following conditions are valid:

1. The subscriber has subscribed to the hold supplementary service; and,

2. For an outgoing association the association is in the outgoing association establishment state. For an incoming association the association is in the associated state.

If the subscriber has subscribed to the hold notification option, an indication that the association has been placed on hold is sent to the remote user.

B.1.3.2 Retrieve Function Operation

The subscriber that invoked the hold service can request the retrieval of the held association by sending a retrieve message to the network service provider.

The subscriber may initiate the retrieve message only for a held association, and before the disestablishment of the association has begun.

The retrieve message contains an identifier of the association to which the retrieve request applies. The retrieve message may contain an optional indication that:

1. any channel is acceptable for retrieval, or
2. a specific channel is preferred for retrieval, or
3. a specified channel is exclusively required for retrieval.

If this optional indicator is not provided, then any B-channel is assumed. (**** NOTE: the case where the retrieve message contains a B-channel indicator was not addressed within the scope of this thesis ****).

If the requested channel is available, then the retrieved association is connected to this channel and the network service provider returns a retrieve acknowledge message to the subscriber to confirm that the retrieve function has been successfully performed.

If the subscriber has subscribed to the hold notification option, an indication that the held association has been retrieved is sent to the remote user.

B.1.3.3 Reservation

(** NOTE: The reservation capability was not explored within the scope of this study *****)**

If the user has subscribed to the reservation option, either the B-channel held (previously used by the existing association) or a B-channel (any available B-channel) is reserved for used by the held association(s) of the user.

B.1.4 Exceptional Procedures

B.1.4.1 Hold Service Invocation

The hold message sent by the user to hold an association may result in a failure as a result of conditions such as: (1) the user has not subscribed to the service, or (2) for some other reason, the service provider cannot hold the association, including a situation wherein the association, requested to be held, is in the process of being released by the network service provider. Under such conditions a hold reject message will be sent back to the subscriber.

If the network service provider receives a hold message for a particular association and determines that the user who sent the request has not subscribed to the service, then a hold reject message will be returned to the user with a cause of "requested service not subscribed".

If the network service provider receives a hold request for a particular incoming association and determines that the association is not in the associated state, then a hold reject message will be returned to the user with a cause of "request not compatible with association state".

If the network service provider receives a hold request for a particular outgoing association and determines that the association is not in the outgoing association establishment state, then a hold reject message will be returned to the user with a cause of "request not compatible with association state".

B.1.4.2 Retrieve Function Operation

The retrieve message, sent by the served user to retrieve a previously held association, may result in failure due to conditions such as: (1) no B-channel is available for the association, or (2) the held association is in the process of being released. Under such conditions, the network service provider will reject the retrieve request by sending back a retrieve reject message to the user.

If the network service provider receives a retrieve message for a particular association and determines that a B-channel cannot be selected for the association, the network will reject the retrieve request with a retrieve reject message containing a cause of either: (1) no channel available or (2) requested B-channel not available.

If the network service provider receives a retrieve message for a particular association and determines that the association is in the process of being released, then a hold reject message will be returned to the user with a cause of "request not compatible with association state".

B.2 Call Waiting Service

The call waiting service [CCITT 88d] permits a served user to be notified of an incoming association (as in basic service) with an indication that no B-channel is available. The user then has the choice of accepting, rejecting or ignoring the waiting association.

B.2.1 Call Waiting Terminology

The following terminology is used in the description of the service.

Subscriber B	This is the subscriber who is provided by the network service provider with the call waiting service on a particular subscriber interface.
User B	This is the one user who reacts to the call waiting at B.
User C	This is the user who has originated an association to B which causes the call waiting service to be invoked.
User A	This represents a user who is engaged in an association with user B.

B.2.2 Provision/Withdrawal

The call waiting service is provided after prearrangement with the service provider. The service can be offered with the subscription options identified in Table B.2.

Table B.2: Call Waiting Service Subscription Options

<u>Subscription Options</u>	<u>Value</u>
Remote user receives notification that their association is waiting	- NO - YES
Maximum number of associations which can be waiting	- 1 - x where $1 \leq x \leq n-m$

The parameter m represents the maximum number of bearer channels defined for the subscriber access. The parameter n represents the maximum number of associations that

can be handled (e.g. active, held, alerting, waiting) for a given subscriber access and is specified as a basic service subscription parameter.

B.2.3 Activation/Deactivation

(**** NOTE: It was assumed in this study that activation of the service was by subscription only ****).

Activation of the call waiting service may be by subscription and deactivation, by withdrawal. Optionally, a network service provider may allow subscribers to activate and deactivate the call waiting service with an appropriate request.

B.2.4 Normal Operation

B.2.4.1 Call Waiting Service Invocation

When an incoming association from user C arrives at the access of Subscriber B, the call waiting service will be invoked when all of the following conditions are true:

1. the maximum number of B-channels for the interface are in use (either busy or reserved);
2. the number of associations has not reached the maximum value defined for the interface; and
3. the number of waiting associations has not reached the maximum value defined for the interface.

When all of the above conditions are true, the association will be offered to Subscriber B with an indication that no B-channel is available.

B.2.4.2 Notification

If the network service provider receives a response at the B access, (within the normal basic service period) that user B is being informed about the incoming association, then an alerting message will be returned to user C. If subscriber B has subscribed to the call waiting notification option, an indication that the association is waiting is sent in the alerting message returned to user C.

B.2.4.3 User B Acceptance of Incoming Association

User B should make a channel available before accepting the waiting association. This may be achieved by releasing the association with user A or placing the association with user A on hold (provided subscriber B has subscribed to the hold service). It is also possible that another user that is not contending for the incoming association, and has a channel allocated to another association, can make a channel available by releasing the association. In any case, the available channel is not reserved by the network service provider for the waiting association.

User B accepts a waiting association by sending a connect message to the network service provider. The network service provider then attempts to allocate a B-channel to the waiting association. If a channel can be allocated to the waiting association, then a connect acknowledge is returned to user B and the "number of waiting associations" is decremented.

B.2.4.4 User B Rejects Waiting Association

After having responded to an incoming association, user B may explicitly reject the waiting association using the basic service establishment procedures. The number of waiting associations should be decremented when the waiting association is released.

B.2.4.5 User C Releases Waiting Association

The waiting association may be released by user C using the basic service disestablishment procedures. The number of waiting associations should be decremented when the waiting association is released.

B.2.5. Exceptional Procedures

If when user B accepts the incoming association and there is no available channel, the network service provider should initiate the release of the association.

B.3 Call Forwarding Unconditional (CFU) Service

The Call Forwarding Unconditional (CFU) [CCITT 88e, BELL 89] service permits a served user to have the network service provider send incoming associations for the served user's number to another number. The served user's originating service is unaffected. If this service is activated, incoming associations are forwarded no matter what the condition of the access.

B.3.1 CFU Terminology

The following terminology is used in the description of the service.

Served Subscriber	This is the subscriber who is requesting that incoming associations to his number be forwarded. This subscriber may also be referred to as the forwarding subscriber.
Forwarded-to-User	This is the user to whom the association will be forwarded.

Originating User

This is the user that initiated an outgoing association that has been forwarded.

B.3.2 Provision/Withdrawal

CFU is provided after prearrangement with the service provider. The service can be offered with the subscription options identified in Table B.3.

Table B.3: Call Forwarding Unconditional Service Subscription Options

<u>Subscription Options</u>	<u>Value</u>
Served User Receives Notification	- NO
that the association has been forwarded.	- YES
Originating User Receives	- NO
Notification that his association has been forwarded.	- YES

B.3.3 Activation/Deactivation

The served user activates the CFU service by sending a cfActivateRequest message to the network service provider. The cfActivateRequest message includes:

1. The service which should be activated (i.e. CFU,CFB,CFNR).
2. The forwarded-to number.
3. Optionally, a password.

The network service provider upon receipt of this message shall return notification of

acceptance or rejection of the request. Possible causes for rejection include:

- service not subscribed
- forwarded-to number is invalid
- forwarded-to number is a special service code (e.g. police).
- forwarded-to number is served user's number
- invalid password

When the served user activates the CFU service, any previous activation for that service will be overridden.

A subscriber may request that the CFU service be deactivated by sending a `cfDeactivateRequest` message to the network service provider. The `cfDeactivateRequest` message identifies the service to be deactivated (e.g. CFU, CFB, CFNR) and may optionally include a password. The service provider will return notification of acceptance or rejection of the deactivate request. The deactivate request may be rejected if a valid password was not provided.

B.3.4 Normal Operation

B.3.4.1 CFU Invocation

When CFU is active, all incoming associations will be forwarded without being offered to the served user. The served user may, as a subscription option, receive notification of the call forwarding (but will not be able to respond to the incoming association). This notification is provided as soon as the forwarding attempt is started.

As a subscription option, the server user can request that the originating user receive a notification that the association has been forwarded.

The forwarded-to user will receive an indication that the association has been forwarded.

B.3.5 Exceptional Procedures

If the CFU forwarded association cannot be completed to the forwarded-to user, then the association should be released with an appropriate reason returned to the originating user.

An association may be forwarded several times. The network service provider shall limit the number of forwardings (e.g. to a maximum of five) for a given association in order to prevent infinite looping.

(**** NOTE: checking for maximum forwardings was not expressed in the CF delta service specifications ****).

B.4 Call Forwarding Busy Service

Call Forwarding Busy (CFB) [CCITT 88e, BELL 89] permits a served user to have the network service provider send incoming associations for the served user's number, which meet "busy", to another number. The served user's originating service is unaffected.

B.4.1 CFB Terminology

Same as for CFU (See Section B.3.1).

B.4.2 Provision/Withdrawal

CFB is provided after prearrangement with the service provider. The service can be offered with the subscription options identified in Table B.4.

Table B.4: Call Forwarding Busy Service Subscription Options

<u>Subscription Options</u>	<u>Value</u>
Served User Receives Notification that the association has been forwarded.	- NO - YES
Originating User Receives Notification that his association has been forwarded.	- NO - YES

B.4.3 Activation/Deactivation

Same procedures as for the CFU service.

B.4.4 Normal Operation

B.4.4.1 CFB Invocation

CFB can be invoked as a result of either a Network Determined User Busy (NDUB) or a User Determined User Busy (UDUB) condition. These two conditions are described below.

If CFB is active and the served user is NDUB (i.e. the maximum number of associations have been reached) then an incoming association to the served user will be forwarded without being offered to the served user. The served user may, as a subscription option, receive notification of the call forwarding (but will not be able to respond to the incoming association). This notification is provided as soon as the forwarding attempt is started.

If CFB is active and the served user is UDUB (i.e. the subscriber responds to an incoming association request with a release indication with a reason of busy) then the incoming association will be forwarded.

As a subscription option, the server user can request that the originating user receive a notification that the association has been forwarded.

The forwarded-to user will receive an indication that the association has been forwarded.

B.4.5 Exceptional Procedures

If the CFB forwarded association cannot be completed to the forwarded-to user, then the association should be released with an appropriate reason returned to the originating user.

An association may be forwarded several times. The network service provider shall limit the number of forwardings (e.g. to a maximum of five) for a given association in order to prevent infinite looping.

B.5 Call Forwarding No Reply Service

Call Forwarding No Reply (CFNR) [CCITT 88e, BELL 89] permits a served user to have the network service provider send incoming associations for the served user's number which meet no reply, to another number. The served user's originating service is unaffected.

CFNR occurs when the incoming association is offered and an indication of a compatible user is received, but no answer (connect) is received before expiry of the CFNR timer.

B.5.1 CFNR Terminology

Same as for CFU (See Section B.3.1).

B.5.2 Provision/Withdrawal

CFNR is provided after prearrangement with the service provider. The service can be offered with the subscription options identified in Table B.5.

Table B.5: Call Forwarding No Reply Service Subscription Options

<u>Subscription Options</u>	<u>Value</u>
Originating User Receives Notification that his association has been forwarded.	- NO - YES
No reply condition timer	- 1..60 seconds

B.5.3 Activation/Deactivation

Same procedures as for the CFU service.

B.5.4 Normal Operation

B.5.4.1 CFNR Invocation

When CFNR is active, incoming associations will be offered to the served user. When an indication is received that the served user is being notified of the incoming association, the CFNR timer will be started. If the served user does not return a connect message within the CFNR time interval, the association will be forwarded. Once the association has been forwarded, the network service provide will release the association to the served user. As a network option [BELL 89], the network service provider will continue to maintain an association to the served user (i.e. will continue to alert the served user), while the association is being forwarded to the forwarded-to user. (**** NOTE: the option which maintains an association to the served user was not modelled in the delta supplementary service specifications ****).

As a subscription option, the server user can request that the originating user receive a notification that the association has been forwarded.

The forwarded-to user will receive an indication that the association has been forwarded.

B.5.4.2 Forwarded-to User Responds with Alerting

If the forwarded-to user responds to the forwarded association with either an alerting or connect indication, then the network service provider will cease to alert the served user and the association towards the served user will be released.

B.5.4.3 Forwarded-to User Responds with Release

If the forwarded-to user responds to the forwarded association with a release indication, then the association towards the forwarded-to user will be released and the network service provider will continue to wait for a response from the served user.

B.5.4.4 Served User Responds with Connect

If the served user responds with a connect message before a response from the forwarded-to user is received, then the association towards the forwarded-to user will be released.

B.5.4.5 Originating User Initiates the Release

If the originating user initiates the release of the association before a response from the forwarded-to user is received, then the association towards the forwarded-to user will be released, as well as the association to the served user.

B.5.5 Exceptional Procedures

An association may be forwarded several times. The network service provider shall limit the number of forwardings (e.g. to a maximum of five) for a given association in order to prevent infinite looping.

B.6 Call Completion to Busy Subscriber (CCBS) Service

Call Completion to Busy Subscriber (CCBS) [SGXVIII 89] allows an originating user A, encountering a busy destination B, to be notified when the busy destination becomes not busy and to have the network service provider reinitiate the association to the specified destination B if user A desires.

B.6.1 CCBS Terminology

The following terminology is used in the description of the service.

Served User A	This is the subscriber who has originated the association and has requested the CCBS service.
Destination User B	This is the busy user to whom the association will be reinitiated when the user is no longer busy.
CCBS Recall	This a network-to-user indication informing the user that (a) destination B is no longer busy and (b) the network service provider will assist the user to reestablish the association if acceptance of the CCBS recall is indicated by the user prior to any service timeouts.

B.6.2 Provision/Withdrawal

CCBS is provided after prearrangement with the service provider.

B.6.3 Activation/Deactivation

Activation of the CCBS service may be by subscription and deactivation, by withdrawal.

B.6.4 Normal Procedures

B.6.4.1 CCBS Service Invocation

When a user A encounters a busy destination B, user A can request the CCBS service. If user A is allowed to use this service and has received a busy indication, the network service provider will retain information in the original association request. This information will be retained for a defined period of time during which user A may invoke CCBS on destination user B. If user A requests CCBS on destination user B, within this period, the service provider will acknowledge the user's request.

After having placed a "CCBS invoke request", user A shall be able to originate and respond to other associations.

When the CCBS request is accepted, the CCBS service duration timer is started. This timer specifies the maximum length of time the destination user will be scanned/monitored for the not busy condition.

B.6.4.2 CCBS Service Cancellation

User A may send a cancellation request to the network service provider requesting that: 1) all CCBS requests should be cancelled, 2) the last CCBS request should be cancelled and 3) a specific CCBS request should be cancelled.

B.6.4.3 Idle Guard Timer

When Destination B becomes not busy, the network service provider will start a Destination B Idle Guard Timer. This timer specifies the amount of time the network will delay, after Destination B has become not busy, before notifying user A. This timer is

intended to allow Destination B time to immediately initiate an outgoing association before any notification is given to User A that Destination B has become not busy. If Destination B is not busy after the Destination B Idle Guard Timer expires, then the recall procedures shall be applied as specified in Section B.6.4.5.

B.6.4.4 Optional Monitoring of User A

As a network service provider option, if the service provider determines that user A is busy when user B becomes not busy, the service provider will not start the CCBS recall timer but will monitor user A for not busy. When user A becomes not busy and destination B is not busy, then the network service provider will invoke the CCBS recall. (**** NOTE: the optional monitoring of User A was not addressed within the scope of this thesis ****).

B.6.4.5 CCBS Recall

The network service provider will provide a CCBS recall indication to User A informing him that Destination B has become not busy. A CCBS recall timer is started by the network service provider to await a response from user A. If User A accepts the CCBS recall, the network service provider will complete the association establishment to destination user B using information retained from the previous association attempt.

If user A responds with rejection of the CCBS recall, the requested CCBS service will be cancelled.

If user A does not respond to the CCBS recall before the CCBS recall timer expires, then the CCBS service will be cancelled.

B.6.4.6 Management of Multiple CCBS Request

(**** NOTE: the management of multiple CCBS Request was not explicitly addressed within the scope of this thesis.***)

Up to "N" (value of N is a network provider option) CCBS requests can be queued against Destination B. Multiple CCBS requests towards the same destination shall be queued and processed on a first-in, first-out (FIFO) basis. The User A whose CCBS request arrived first shall be informed first when Destination B becomes idle.

When Destination B becomes not busy, B's CCBS request queue shall be serviced upon expiration of the Destination B Idle Guard Timer. If Destination B is idle, but User A is subsequently found to be busy, the network shall mark the corresponding CCBS request as inactive. User A shall retain its position in B's request queue. The inactive CCBS requests shall be reactivated upon User A again achieving an idle status.

The network shall proceed to inform the other active User As' in the queue of Destination B's idle status, in the order in which they were placed in the queue. When an idle User A is found, no other User As in the queue shall be informed of Destination B's idle status for a period of time (Destination B Request Queue Service Timer). After the Destination B Request Queue Service Timer expires, and if Destination B is still idle, Destination B's request queue shall again be serviced.

The servicing shall continue on a FIFO basis for all active entries. Each time the Destination B Request Queue Service Timer expires, the first request in the queue shall be checked for active status, then the second, and so on until an active request is found.

A CCBS request shall be removed from the queue if:

- User A is offered a CCBS Recall and the CCBS Recall Timer expires.
- User A's exchange requests cancellation of the entry.
- The CCBS Service Duration Timer expires.

If User A attempts a subsequent CCBS request to the same Destination B, while their previous request is still present in the queue, the CCBS request shall retain its place in the queue and the CCBS Service Duration Timer shall be restarted.

B.6.5 Exceptional Procedures

The CCBS supplementary service is automatically cancelled and user A is notified if:

- destination B is still busy after the CCBS service duration timer expires.
- user A does not accept the CCBS recall before the recall timer expires.

APPENDIX C

MLog Telecommunication Service Prototype Source Code

```

%*
%*
%* MLOG PROGRAM
%*
%* This MLog program represents an executable specification
%* of the basic telecommunication service and a number of
%* supplementary services (i.e. call hold, call waiting,
%* call forwarding unconditional, call forwarding busy and
%* call forwarding no reply) for subscribers served by the same
%* local exchange.
%*
%* DATE: November 14, 1989
%* AUTHOR: Linda Simon
%*

```

```

%*
%*
%* CLASS : MAIN
%*
%* This object specifies the basic service objects
%* which should be created upon initialization of the
%* MLog Telecommunication Service Prototype. It also
%* sends a message to the datafill class to provide
%* the initial configuration for the MLog prototype.
%*

```

class: main of object.

begin :-

```

serviceProvision-new(provision,active,[]),
resourceControl-new(resource,active,[]),
directoryDatabase-new(directory,active,[]),
messageQueue-new(associationQueue,active,[]),
transporter-new(X,active,[sender=associationQueue]),
subscriberReceive-new(subscriber,active,[]),
subscriberSend-new(test,active,[]),
mainConsole-writeln("All initialized"),
dataFill-loadSubscriberData,
mainConsole-writeln("data filled").

```

```

%*
%*
%* C L A S S : O B J E C T
%*
%* This is the top-level class. This object class has
%* been extended in the MLog Telecommunication Service
%* prototype to include methods which are common to all
%* basic service and supplementary service objects.
%* This includes methods to manipulate lists, the
%* primary data structures used in the MLog prototype,
%* and methods to support inter-object communication.
%*

```

class: object of nil.

```

% check whether X is an element of the list L.
membt .(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

```

```

% subst(X,L,A,M): substitute the first occurrence of element X
% in the List L with the element A to form the new list M.
subst(X,[X|L],A,[A|L]) :- !.
subst(X,[Y|L],A,[Y|M]) :- subst(X,L,A,M).

```

```

% efface(X,Y,L): remove the first occurrence of element S from list Y, giving
% the effaced list L.
efface(X,[X|L],L).
efface(X,[Y|L],[Y|L2]) :- efface(X,L,L2).

```

```

% intersection(L1,L2,L3) : L3 is the intersection between the lists L1, L2
intersection([],L,[]).
intersection([X|L1],L2,[X|L3]) :- member(X,L2), intersection(L1,L2,L3).
intersection([X|L1],L2,L3) :- not(member(X,L2)), intersection(L1,L2,L3).

```

```

% addToList(ELEMENT,LIST,LIST2): add the ELEMENT to the beginning of the LIST
% to form the new list LIST2
addToList(ELEMENT,LIST,[ELEMENT|LIST]).

```

```

sendMessage(MESSAGE TYPE,PARAMETERS) :- succeed.

```

```

outputMessages([]).
outputMessages([X|L]) :- sendOutput(X), outputMessages(L).

```

```

sendOutput([OID,MESSAGE TYPE,PARAMETERS]) :-
    OID-sendMessage(MESSAGE TYPE,PARAMETERS).

```



```

%*
%* C L A S S: TRANSACTION CONTROL
%*
%* This class controls and coordinates the various types of
%* transactions across the subscriber access. There is an
%* object instance associated with each subscriber access
%* that is defined. This object controls the number of
%* simultaneously active associations (ivar: numberAssoc)
%* across the subscriber access. The maximum number of
%* associations are defined by the ivar: maxAssoc. It
%* dynamically creates the appropriate object based on the
%* nature of the service request and the services to which
%* the subscriber has subscribed. It routes messages from
%* the subscriber to the appropriate object based on the
%* mapping from reference (included in all messages) to
%* object identifier stored in the ivar: referenceList.
%*

```

class: transactionControl of object.

```

ivars: [subscriberNum,maxAssoc,numberAssoc=0,
        referenceList=[],nextReferenceNum=100,
        outgoingMessageQueue=[]].

```

```

begin :-
    repeat,
        waitToAccept(Message),
        postRendezvous(Message),
    fail.

```

```

getMessage(DATA) :-
    theValueOf(outgoingMessageQueue,QUEUE),
    listLength(QUEUE,S), S > 0,
    $dequeueItem(DATA,QUEUE,QUEUE2),
    makeTheValueOf(outgoingMessageQueue,QUEUE2).

```

```

postRendezvous(sendMessage(MESSAGETYPE,PARAMETERS)) :-
    $decodeMessage(MESSAGETYPE,PARAMETERS).

```

```

postRendezvous(getMessage(DATA)) :- succeed.

```

```

$decodeMessage(setup,PARAMETERS) :-
    $setup(PARAMETERS), !.

```

```

$decodeMessage(setupRequest,PARAMETERS) :-
    $setupRequest(PARAMETERS), !.

```

\$iecodeMessage(transactionComplete,PARAMETERS) :-
 \$transactionComplete(PARAMETERS), !.

\$decodeMessage(MESSAGE,PARAMETERS) :-
\$routeMessage(MESSAGE,PARAMETERS).

\$setup([REFERENCE,CALLEDNUMBER]) :-
 theValueOf(referenceList,REFLIST),
 theValueOf(subscriberNum,SN),
 member([REFERENCE|_],REFLIST),
 myself(OID),
 subscriber-logMessage(SN,releaseComplete,transactionControl,OID,
 " CAUSE = setup message received for reference in Use"),
 !.

\$setup([REFERENCE,CALLEDNUMBER]) :-
 \$maxAssocReached,
 theValueOf(subscriberNum,SN),
 myself(OID),
 subscriber-logMessage(SN,releaseComplete,transactionControl,OID,
 "CAUSE = Maximum associations reached"),
 !.

\$setup([REFERENCE,CALLEDNUMBER]) :-
 theValueOf(subscriberNum,SN),
 provision-getSubscriberServices(SN,LISTOFSERVICES),
 \$determineOutgoingInstance(LISTOFSERVICES,OBJECTNAME),
 myself(OID),
 OBJECTNAME-new(PID,active,[tcId=OID,subscriberNum=SN,
 reference=REFERENCE]),
 \$incrementAssoc,
 \$addToRefList(REFERENCE,PID),
 \$storeMessage([PID,setup,CALLEDNUMBER]), **!.**

\$setupRequest([ACid,CALLINGNUMBER]) :-
 % maximum number of associations have been reached - this is a
 % network determined busy condition
 \$maxAssocReached,
 \$handleMaxAssocReached(ACid), **!.**

\$setupRequest([ACid,CALLINGNUMBER]) :-
 theValueOf(subscriberNum,SN),
 provision-getSubscriberServices(SN,LISTOFSERVICES),
 \$getReference(REFERENCE),
 \$determineIncomingInstance(LISTOFSERVICES,OBJECTNAME),
 myself(OID),
 OBJECTNAME-new(PID,active,[tcId=OID,subscriberNum=SN,

```

    reference=REFERENCE,acId=ACid)),
    $incrementAssoc,
    $addToRefList(REFERENCE,PID),
    $storeMessage([ACid,objectID,PID]),
    $storeMessage([PID,setupRequest,CALLINGNUMBER]).

$transactionComplete([bearerAssociated,REFERENCE]) :-
% A bearer associated transaction has completed, decrement the number of
% associations. Remove the reference from the reference list.
    $decrementAssoc,
    $removeFromRefList(REFERENCE), !.

$transactionComplete([nonBearerAssociated,REFERENCE]) :-
% A non-bearer associated transaction has completed, remove the Reference from
% the reference list.
    $removeFromRefList(REFERENCE).

$routeMessage(MESSAGE,[REFERENCE,OTHERPARAMETERS]) :-
    theValueOf(referenceList,REFLIST),
    member([REFERENCE,OID_],REFLIST),
    $storeMessage([OID,MESSAGE,OTHERPARAMETERS]), !.

$routeMessage(MESSAGE,PARAMETERS) :-
    myself(OID),
    theValueOf(subscriberNum,SN),
    subscriber-logMessage(SN,releaseComplete,transactionControl,OID,
        " CAUSE = reference not in Use").

$maxAssocReached :-
    theValueOf(maxAssoc,MAXASSOC),
    theValueOf(numberAssoc,MAXASSOC).

$handleMaxAssocReached(ACid) :-
% With the basic service when the maximum number of associations have been
% reached the association is released.
    myself(OID),
    $storeMessage([ACid,rejectIndication,[OID,maxAssocReached]]), !.

$incrementAssoc :-
    theValueOf(numberAssoc,N),
    N2 is N + 1,
    myself(OID),
    mainConsole-(write("number of Associations associated with TC object,"),
        write(OID),write(", has increased to: "),
        write(N2),nl(2)),
    makeTheValueOf(numberAssoc,N2).

```

```

$decrementAssoc :-
    theValueOf(numberAssoc,N),
    N2 is N - 1,
    myself(OID),
    mainConsole-(write("Transaction Complete received by TC object "),
        write(OID),nl,write(" Number of associations are: "),
        write(N2),nl(2)),
    makeTheValueOf(numberAssoc,N2).

$addToRefList(REFERENCE,OID) :-
    theValueOf(referenceList,REFLIST),
    makeTheValueOf(referenceList,[[REFERENCE,OID]|REFLIST]).

$removeFromRefList(REFERENCE) :-
    % remove the REFERENCE from the reference list.
    theValueOf(referenceList,REFLIST),
    efface([REFERENCE|_],REFLIST,REFLIST1),
    makeTheValueOf(referenceList,REFLIST1).

$getReference(SOLUTION) :-
    theValueOf(nextReferenceNum,START),
    $findAvailableReference(START,SOLUTION),
    $nextRef(SOLUTION,NEXTREF),
    makeTheValueOf(nextReferenceNum,NEXTREF).

$findAvailableReference(SOLUTION,SOLUTION) :-
    theValueOf(referenceList,REFLIST),
    not(member(SOLUTION,REFLIST)).

$findAvailableReference(START,SOLUTION) :-
    theValueOf(referenceList,REFLIST),
    member(START,REFLIST),
    $nextRef(START,NEWSTART),
    $findAvailableReference(NEWSTART,SOLUTION).

$nextRef(REF,REF2) :- REF2 is REF - 1, REF2 @>= 0, !.
$nextRef(REF,100).

$determineOutgoingInstance(LISTOFSERVICES,outgoingAssociation) :- !.

$determineIncomingInstance([],incomingAssociation) :- !.
$determineIncomingInstance(LISTOFSERVICES,incomingAssociation) :-
    intersection(LISTOFSERVICES,[cfb,cfnr,callWaiting],[]), !.
$determineIncomingInstance(LISTOFSERVICES,OBJECTNAME) :-
    % only one service subscribed
    intersection(LISTOFSERVICES,[cfb,callWaiting,cfnr],[OBJECTNAME]), !.

```

```

$determineIncomingInstance(LISTOF SERVICES,callWaitingCfnr) :-
% two services subscribed
intersection(LISTOF SERVICES,[cfb,callWaiting,cfnr],X),
member(cfnr,X),
member(callWaiting,X), !.

```

```

$storeMessage(DATA) :-
theValueOf(outgoingMessageQueue,QUEUE),
$enqueueItem(DATA,QUEUE,QUEUE2),
makeTheValueOf(outgoingMessageQueue,QUEUE2).

```

```

$dequeueItem(X,[XIL],L).

```

```

$enqueueItem(X,[],[X]).

```

```

$enqueueItem(X,[YIL],[YIL2]) :- $enqueueItem(X,L,L2).

```

```

%*
%* CLASS: SUBSCRIBER ASSOCIATION
%*
%* This object specifies the behaviour which is common to
%* both the outgoing association and incoming association
%* classes. It represents an abstract class since an
%* instance of this class is never created. It specifies
%* the "association disestablishment" behaviour.
%*

```

```

class: subscriberAssociation of object.

```

```

ivars: [basicServiceState=idle,tcId,subscriberNum,channelId=noChannel,
reference,acId,activeTimer=[]].

```

```

begin :-
repeat,
waitToAccept(Message),
postRendezvous(Message),
theValueOf(basicServiceState,terminated),
!.

```

```

timerExpired(TIMERREFERENCE) :- succeed.

```

```

queryBasicServiceState(CURRENTSTATE) :-
% returns the current basic service state. This method is used to support
% interactions with the hold service
theValueOf(basicServiceState,CURRENTSTATE).

```

seizeChannel(CHANNELID) :-

% this method can be called by the outgoing or incoming association object.

% It can also be invoked by a "hold" object sending a "seizeChannel"

% message to the outgoing or incoming association object.

myself(OID),

theValueOf(subscriberNum,SN),

resource-seizeBchannel(SN,OID,CHANNELID),

makeTheValueOf(channelId,CHANNELID).

releaseChannel :-

% this method can be called by the outgoing or incoming association object.

% It can also be invoked by a "hold" object sending a "releaseChannel"

% message to the outgoing or incoming association object.

theValueOf(channelId,CHANNELID),

CHANNELID \== noChannel,

theValueOf(subscriberNum,SN),

resource-releaseBchannel(SN,CHANNELID), !.

releaseChannel :- !.

postRendezvous(queryBasicServiceState(CURRENTSTATE)) :- succeed.

postRendezvous(releaseChannel) :- succeed.

postRendezvous(seizeChannel(CHANNELID)) :- succeed.

postRendezvous(timerExpired(TIMERREFERENCE)) :-

postRendezvous(sendMessage(timerExpired,TIMERREFERENCE)).

postRendezvous(sendMessage(MESSAGEYPE,PARAMETERS)) :-

theValueOf(basicServiceState,CURRENTSTATE),

**stateMachine(CURRENTSTATE,[MESSAGEYPE,PARAMETERS],NEWSTATE,
OUTPUT),**

myself(OID),

mainConsole-(write(MESSAGEYPE),write(" received by object "), write(OID),

write(" in state:"), write(CURRENTSTATE), nl,

write(" New state is:"),

write(NEWSTATE),nl),

makeTheValueOf(basicServiceState,NEWSTATE),

outputMessages(OUTPUT), !.

**stateMachine(waitForReleaseResponse,[MESSAGEYPE,PARAMETERS],
NEWSTATE,OUTPUT) :-**

**waitForReleaseResponse([MESSAGEYPE,PARAMETERS],NEWSTATE,OUTPUT),
!.**

```

stateMachine(releaseIndication,[MESSAGE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    releaseIndication([MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

stateMachine(disconnectIndication,[MESSAGE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    disconnectIndication([MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

stateMachine(releaseRequest,[MESSAGE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    releaseRequest([MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

stateMachine(disconnectRequest,[MESSAGE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    disconnectRequest([MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[releaseRequest,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    disestablishOnReleaseRequest(CURRENTSTATE,PARAMETERS,
        NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[disconnect,PARAMETERS],NEWSTATE,OUTPUT) :-
    disestablishOnDisconnect(CURRENTSTATE,PARAMETERS,
        NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[release,PARAMETERS],NEWSTATE,OUTPUT) :-
    disestablishOnRelease(CURRENTSTATE,PARAMETERS,
        NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[releaseComplete,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    disestablishOnReleaseComplete(CURRENTSTATE,PARAMETERS,
        NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[MESSAGE,_,CURRENTSTATE,[]]) :-
    $sendMessageToSubscriber(subscriberAssociation,messageReject,
        "unexpected or unrecognized message ").

waitForReleaseResponse([releaseResponse,REASON],
    terminated,[OUTPUT]) :-
    $associationReleased(OUTPUT).

releaseIndication([MESSAGE,_,terminated,[OUTPUT]) :-
    member(MESSAGE,[releaseRequest,releaseResponse]),
    $sendMessageToSubscriber(subscriberAssociation,releaseComplete,[]),
    releaseChannel,
    $associationReleased(OUTPUT).

```

```

disconnectIndication([MESSAGE,TYPE,_,],releaseRequest,[]) :-
    member(MESSAGE,TYPE,[releaseRequest,releaseResponse]),
    myself(OID),
    $sendMessageToSubscriber(subscriberAssociation,release,[]).

releaseRequest([MESSAGE,TYPE,REASON],terminated,[OUTPUT]) :-
    member(MESSAGE,TYPE,[release,releaseComplete]),
    releaseChannel,
    $associationReleased(OUTPUT).

disconnectRequest([disconnect,_,],releaseRequest,[]) :-
    $sendMessageToSubscriber(subscriberAssociation,release,[]), !.

disconnectRequest([MESSAGE,TYPE,_,],terminated,[OUTPUT]) :-
    member(MESSAGE,TYPE,[release,releaseComplete]),
    releaseChannel,
    $associationReleased(OUTPUT).

disestablishOnReleaseRequest(CURRENTSTATE,REASON,disconnectRequest,
    [[ACID,releaseConfirm,OID]]) :-
    myself(OID),
    $sendMessageToSubscriber(subscriberAssociation,disconnect,[]),
    theValueOf(acId,ACID).

disestablishOnDisconnect(CURRENTSTATE,REASON,disconnectIndication,
    [[ACID,releaseIndication,[OID,REASON]]]) :-
    myself(OID),
    theValueOf(acId,ACID).

disestablishOnRelease(CURRENTSTATE,REASON,releaseIndication,
    [[ACID,releaseIndication,[OID,REASON]]]) :-
    theValueOf(acId,ACID),
    myself(OID).

disestablishOnReleaseComplete(CURRENTSTATE,REASON,
    waitForReleaseResponse,[[ACID,releaseIndication,[OID,REASON]]]) :-
    theValueOf(acId,ACID),
    myself(OID),
    releaseChannel.

$associationReleased([X,transactionComplete,[bearerAssociated,REFERENCE]]) :-
    theValueOf(tcId,X),
    theValueOf(reference,REFERENCE).

$sendMessageToSubscriber(CLASS,MESSAGE,TYPE,PARAMETERS) :-
    myself(OID),
    theValueOf(subscriberNum,SN),
    subscriber-logMessage(SN,MESSAGE,TYPE,CLASS,OID,PARAMETERS).

```



```

%*
%* CLASS: OUTGOING ASSOCIATION
%*
%* This object specifies the behaviour which is specific
%* to an outgoing association. It is a sub-class of the
%* subscriberAssociation class.
%*

```

class: outgoingAssociation of subscriberAssociation.

```

stateMachine(idle,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT) :-
    idle([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT), !.

```

```

stateMachine(associationProceeding,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    associationProceeding([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT),
    !.

```

```

stateMachine(associationDelivered,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    associationDelivered([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT), !.

```

```

stateMachine(CURRENTSTATE,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,
    OUTPUT) :-
    super-stateMachine(CURRENTSTATE,[MESSAGE,TYPE,PARAMETERS],
        NEWSTATE,OUTPUT), !.

```

```

idle([setup,CALLEDNUMBER],associationProceeding,[]) :-
    myself(OID),
    theValueOf(subscriberNum,SN),
    seizeChannel(CHANNELID),
    CHANNELID \== noChannel,
    associationControl-setupIndication(SN,CALLEDNUMBER,OID,ACID),
    makeTheValueOf(acId,ACID),
    !.

```

```

idle([setup,CALLEDNUMBER],terminated,[OUTPUT]) :-
    $sendMessageToSubscriber(outgoingAssociation,releaseComplete,
        "REASON = no channel available"),
    $associationReleased(OUTPUT).

```

```

associationProceeding([report,progress],associationProceeding,[]) :-
    $sendMessageToSubscriber(outgoingAssociation,progress,[]).
associationProceeding([report,alerting],associationDelivered,[]) :-
    $sendMessageToSubscriber(outgoingAssociation,alerting,[]).

```

```
associationProceeding([report,[alerting,NOTIFICATION]],associationDelivered,[]) :-  
    $sendMessageToSubscriber(outgoingAssociation,alerting,NOTIFICATION).
```

```
associationProceeding([setupResponse,_],associated,[]) :-  
    $sendMessageToSubscriber(outgoingAssociation,connect,[]).
```

```
associationDelivered([report,progress],associationDelivered,[]) :-  
    $sendMessageToSubscriber(outgoingAssociation,progress,[]).  
associationDelivered([setupResponse,_],associated,[]) :-  
    $sendMessageToSubscriber(outgoingAssociation,connect,[]).
```

```
%*  
%*  CLASS:INCOMING ASSOCIATION  
%*  
%*  This object specifies the behaviour which is specific  
%*  to an incoming association. It is a sub-class of the  
%*  subscriberAssociation class.  
%*
```

class: incomingAssociation of subscriberAssociation.

```
stateMachine(idle,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT) :-  
    idle([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT), !.
```

```
stateMachine(waitForUserResponse,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,  
    OUTPUT) :-  
    waitForUserResponse([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT),  
    !.
```

```
stateMachine(userResponded,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,  
    OUTPUT) :-  
    userResponded([MESSAGE,TYPE,PARAMETERS],NEWSTATE,OUTPUT), !.
```

```
stateMachine(CURRENTSTATE,[MESSAGE,TYPE,PARAMETERS],NEWSTATE,  
    OUTPUT) :-  
    super-stateMachine(CURRENTSTATE,[MESSAGE,TYPE,PARAMETERS],  
        NEWSTATE,OUTPUT), !.
```

```
idle([setupRequest,CALLINGNUMBER],waitForUserResponse,[]) :-  
    seizeChannel(CHANNELID),  
    CHANNELID \== noChannel,  
    $performEstablishIncomingAssocActions,  
    !.
```

```

idle([setupRequest,CALLINGNUMBER],waitForReleaseResponse,
  [[ACID,releaseIndication,[OID,noChannelAvailable]]]) :-
  myself(OID),
  mainConsole-writeln("no channel available"),
  theValueOf(acId,ACID), !.

```

```

waitForUserResponse([alerting,_],userResponded,[[ACID,report,alerting]]) :-
  theValueOf(acId,ACID), !.

```

```

waitForUserResponse([connect,_],associated,[[ACID,setupConfirm,_]]) :-
  theValueOf(acId,ACID), !.

```

```

userResponded([connect,_],associated,[[ACID,setupConfirm,_]]) :-
  theValueOf(acId,ACID).

```

```

$performEstablishIncomingAssocActions :-
  myself(OID),
  theValueOf(reference,REFERENCE),
  % timer-startTimer(60,OID,TIMERREFERENCE),
  $sendMessageToSubscriber(incomingAssociation,setup,REFERENCE).

```

```

%*
%*  A S S O C I A T I O N  C O N T R O L
%*
%*  This object is responsible for establishing, maintaining
%*  and releasing an association between an outgoing
%*  association object (identified by the ivar: invokerId)
%*  and an incoming association object (identified by the
%*  ivar: responderId). This association is in support of
%*  providing a basic telecommunication service between
%*  an originating subscriber (identified by the ivar:
%*  callingNumber) and a responding subscriber (identified by
%*  the ivar: calledNumber).
%*

```

class: associationControl of object.

ivars: [state=idle,callingNumber,calledNumber,invokerId,responderId].

methods: class.

```

setupIndication(CALLINGNUMBER,CALLEDNUMBER,INVOKER,ACID) :-
  provision-getForwardingServices(CALLEDNUMBER,X),
  determineAssociationControlInstance(X,CLASS),
  CLASS-new(ACID,active,[invokerId=INVOKER]),
  ACID-sendMessage(setupIndication,[CALLINGNUMBER,CALLEDNUMBER]).

```

determineAssociationControlInstance(X,acCFB) :-
 member(cfu,X),
 member(cfb,X), !.

determineAssociationControlInstance([cfb],acCFB) :- !.

determineAssociationControlInstance([cfu],acCFU) :- !.

determineAssociationControlInstance([cfnr],acCFNR) :- !.

determineAssociationControlInstance(X,associationControl).

member(X,[X|L]).

member(X,[Y|L]) :- member(X,L).

methods: instance.

begin
 repeat,
 waitToAccept(Message),
 postRendezvous(Message),
 theValueOf(state,terminated),
 !.

postRendezvous(sendMessage(MESSAGE,TYPE,PARAMETERS)) :-
 theValueOf(state,CURRENTSTATE),

acStateMachine(CURRENTSTATE,[MESSAGE,TYPE,PARAMETERS],NEXTSTATE,
 OUTPUT),
 mainConsole-(nl,write(MESSAGE),
 write(" received by association control object in state "),
 write(CURRENTSTATE),nl,
 write(" New state is: "),
 writeln(NEXTSTATE),nl(2)),
 makeTheValueOf(state,NEXTSTATE),
 outputMessages(OUTPUT), !.

acStateMachine(idle,[MESSAGE,TYPE,PARAMETERS],NEXTSTATE,OUTPUT) :-
 idle([MESSAGE,TYPE,PARAMETERS],NEXTSTATE,OUTPUT), !.

acStateMachine(waitForResponderId,[MESSAGE,TYPE,PARAMETERS],NEXTSTATE,
 OUTPUT) :-
 waitForResponderId([MESSAGE,TYPE,PARAMETERS],NEXTSTATE,OUTPUT), !.

```

acStateMachine(releasePending,[MESSAGE,PARAMETERS],NEXTSTATE,
OUTPUT) :-
    releasePending([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT), !.

acStateMachine(waitForSetupConfirm,[MESSAGE,PARAMETERS],
NEXTSTATE, OUTPUT) :-
    waitForSetupConfirm([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT),
    !.

acStateMachine(waitForReleaseConfirm,[MESSAGE,PARAMETERS],
NEXTSTATE,OUTPUT) :-
    waitForReleaseConfirm([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT), !.

acStateMachine(CURRENTSTATE,[releaseIndication,PARAMETERS],NEXTSTATE,
OUTPUT) :-
    releaseAssociation(PARAMETERS,NEXTSTATE,OUTPUT), !.

acStateMachine(CURRENTSTATE,[queryState,_],CURRENTSTATE,[]) :-
    mainConsole-(write("Current State of Association Control Object is "),
        write(CURRENTSTATE),nl), !.

acStateMachine(CURRENTSTATE,[MESSAGE,PARAMETERS],
CURRENTSTATE,[]) :-
    myself(OID),
    mainConsole-(write("Unexpected or unrecognized message received by "),
        write("Association Control Object Instance "),
        write(OID),
        write(" Message type is: "),
        write(MESSAGE),nl), !.

idle([setupIndication,[CL,CD]],waitForResponderId,
[[X,setupRequest,[ACID,CL]]]) :-
    directory-getObjectId(transactionControl,CD,X),
    X \== nilObject,
    makeTheValueOf(callingNumber,CL),
    makeTheValueOf(calledNumber,CD),
    myself(ACID), !.

idle([setupIndication,[CL,CD]],waitForReleaseConfirm,
[[INVOKER,releaseRequest,routingFailure]]) :-
    theValueOf(invokerId,INVOKER).

waitForResponderId([objectID,RESPONDERID],waitForSetupConfirm,[]) :-
    makeTheValueOf(responderId,RESPONDERID).

```

```

waitForResponderId([releaseIndication,[OID,REASON|_]],releasePending,
  [[OID,releaseResponse,_]]) :-
  theValueOf(invokerId,OID), !.

waitForResponderId([rejectIndication,[OID,REASON|_]],waitForReleaseConfirm,
  [[INVOKER,releaseRequest,_]]) :-
  theValueOf(invokerId,INVOKER).

releasePending([objectID,RESPONDERID],waitForReleaseConfirm,
  [[RESPONDERID,releaseRequest,_]]).

releasePending([releaseIndication,[OID,REASON|_]],terminated,
  [[OID,releaseResponse,_]]).

waitForSetupConfirm([setupConfirm,_],associationEstablished,
  [[INVOKER,setupResponse,_]]) :-
  theValueOf(invokerId,INVOKER).

waitForSetupConfirm([report,PARAMETER],waitForSetupConfirm,
  [[INVOKER,report,PARAMETER]]) :-
  theValueOf(invokerId,INVOKER).

waitForReleaseConfirm([releaseConfirm,OID],terminated,[]).
waitForReleaseConfirm([releaseIndication,_],terminated,[]).

releaseAssociation([OID,REASON|_],waitForReleaseConfirm,
  [[OID,releaseResponse,_],[OTHERENDOID,releaseRequest,REASON]]) :-
  $getOtherEndOfAssociation(OID,OTHERENDOID).

$getOtherEndOfAssociation(OID,OTHERENDOID) :-
  theValueOf(invokerId,OID),
  theValueOf(responderId,OTHERENDOID), !.

$getOtherEndOfAssociation(OID,OTHERENDOID) :-
  theValueOf(invokerId,OTHERENDOID).

sendOutput(X) :-
  associationQueue-storeMessage(X).

```

```

%*
%*  C L A S S :  S E R V I C E  P R O V I S I O N
%*
%*  This object provides methods to define new subscribers,
%*  and to allow existing subscribers to subscriber to a number
%*  of different supplementary services.
%*  The supplementary services to which the subscriber
%*  has subscribed, as well as the subscription parameters are
%*  stored by this object in the subscriptions ivar.
%*  A number of methods are provided to allow other objects to
%*  access the subscription information.
%*

```

class: serviceProvision of object.

```

ivars: {subscriberList=[],subscriptions=[],
        validServiceList=[cfu,cfb,cfnr,ccbs,callWaiting,callHold]}.

```

addSubscriber(OID,SN,MAXASSOC) :-

```

% This is a new subscriber. Update subscriberList, create transaction control
% object and transaction control transporter object. Add the translation from
% subscriber number to object identifier in the directory database. Define a
% new access in the access database.
    theValueOf(subscriberList,SL),
    not(member([SN|_],SL)),
    transactionControl-new(OID,active,[subscriberNum=SN,maxAssoc=MAXASSOC]),
    transporter-new(X,active,[sender=OID]),
    addToList([SN,MAXASSOC,X,transactionControl],SL,SL2),
    makeTheValueOf(subscriberList,SL2),
    mainConsole-(write("New subscriber added with number = "),
                  write(SN),nl),
    directory-addObject(transactionControl,SN,OID),
    resource-addAccess(SN), !.

```

addSubscriber(OID,SN,MAXASSOC) :-

```

    mainConsole-writeln("subscriber is already defined").

```

subscribeToService(SN,NEWSERVICE,PARAMETERS) :-

```

% check that subscriber is defined.
    $subscriberNotDefined(SN), !.

```

subscribeToService(SN,NEWSERVICE,PARAMETERS) :-

```

% check that the requested service is a valid service
    theValueOf(validServiceList,VALIDSERVICES),
    not(member(NEWSERVICE,VALIDSERVICES)),
    mainConsole-(write("Not a valid service, Valid services are:"),

```

```

write(VALIDSERVICES), nl), !.

subscribeToService(SN,callHold,_) :-
% request to subscribe to call hold service.
  $addService(SN,[callHold]),
  $modifyTCbehaviour(SN,callHold), !.

subscribeToService(SN,callWaiting,[MaxWaitingAssoc,NotificationToOriginator]) :-
% request to subscribe to call waiting service, check for valid service
% parameters. First check notification parameter. This parameter
% indicates whether the originating party should receive a notification that
% the service request is being offered using the call waiting service operation.
% In this case the parameter value is invalid, service cannot be subscribed.
  $invalidNotificationParameterValue(NotificationToOriginator), !.

subscribeToService(SN,callWaiting,[MaxWaitingAssoc,NotificationToOriginator]) :-
% check for valid service parameter: maximum waiting associations
  MaxWaitingAssoc @>= 1,
  MaxWaitingAssoc @=< 5,
  $addService(SN,[callWaiting,[MaxWaitingAssoc,NotificationToOriginator]]), !.

subscribeToService(SN,callWaiting,[MaxWaitingAssoc,NotificationToOriginator]) :-
  mainConsole-(
    write("Invalid parameter value for maximum waiting associations"),
    nl,write("Valid values are: 1 <= n => 5"),nl), !.

subscribeToService(SN,cfu,[Password,NotificationToOriginator,
  NotificationToServedUser]) :-
% request to subscribe to the call forwarding unconditional service. Check
% for valid service parameters. First check NotificationToOriginator parameter.
% This parameter indicates whether the originating party should be notified
% that the association has been forwarded. In this case the parameter value
% supplied is invalid so the service subscription request is rejected.
  $invalidNotificationParameterValue(NotificationToOriginator), !.

subscribeToService(SN,cfu,[Password,NotificationToOriginator,
  NotificationToServedUser]) :-
% next check that the NotificationToServedUser parameter is valid. This parameter
% indicates whether the served user (the subscriber who has requested that
% incoming associations to his number be forwarded) should be notified
% that a particular association has been forwarded.
  $invalidNotificationParameterValue(NotificationToServedUser), !.

subscribeToService(SN,cfu,[Password,NotificationToOriginator,
  NotificationToServedUser]) :-
% all parameters valid, add service. Modify the behaviour of the transaction
% control instance to reflect the new behaviour when the cfu service is added.
% This new behaviour is described in the callForwardingUnconditional object.

```



```

$addService(SN,[cfu,[Password,NotificationToOriginator,
    NotificationToServedUser]]),
$modifyTCbehaviour(SN,cfu), !.

subscribeToService(SN,cfb,[Password,NotificationToOriginator]) :-
% request to subscribe to the call forwarding on busy service. Check that the
% NotificationToOriginator parameter is valid. This parameter has the same
% semantics as for the cfu service.
$invalidNotificationParameterValue(NotificationToOriginator), !.

subscribeToService(SN,cfb,[Password,NotificationToOriginator]) :-
% all parameters valid, add service. Modify the behaviour of the transaction
% control instance to reflect the new behaviour when the cfb service is added.
% This new behaviour is described in the callForwardingBusy object.
$addService(SN,[cfb,[Password,NotificationToOriginator]]),
$modifyTCbehaviour(SN,cfb), !.

subscribeToService(SN,cfnr,[Password,NotificationToOriginator,TimerValue]) :-
% request to subscribe to the call forwarding on no reply service. Check that
% the NotificationToOriginator parameter is valid. This parameter has the same
% semantics as for the cfu service.
$invalidNotificationParameterValue(NotificationToOriginator), !.

subscribeToService(SN,cfnr,[Password,NotificationToOriginator,TimerValue]) :-
% check the timerValue parameter. This parameter indicates the time in seconds
% that the network will wait for a connect from the user before it forwards the
% association.
TimerValue isA integer,
TimerValue @> 0,
$addService(SN,[cfnr,[Password,NotificationToOriginator,TimerValue]]),
$modifyTCbehaviour(SN,cfnr), !.

subscribeToService(SN,cfnr,[Password,NotificationToOriginator,TimerValue]) :-
% invalid timer value parameter.
mainConsole-(write("Invalid parameter value for cfnr timer"),nl,
    write("Valid values are: > 0"),nl), !.

getSubscriberServices(SN,[]) :-
% check subscriber is defined
$subscriberNotDefined(SN), !.

getSubscriberServices(SN,LISTOFSERVICES) :-
% subscriber has services currently defined.
theValueOf(subscriptions,SUBSCRIPTIONLIST),
member([SN|SERVICEINFO],SUBSCRIPTIONLIST),
$extractServiceList(SERVICEINFO,LISTOFSERVICES), !.

```

```

getSubscriberServices(SN,[]) :- succeed, !.

getForwardingServices(SN,FORWARDINGLIST) :-
    getSubscriberServices(SN,LISTOFSERVICES), !,
    intersection(LISTOFSERVICES,[cfu,cfb,cfnr],FORWARDINGLIST).

getServiceParameters(SN,SERVICE,PARAMETERS) :-
    % check subscriber is defined
    $subscriberNotDefined(SN), !.

getServiceParameters(SN,SERVICE,PARAMETERS) :-
    % subscriber has services currently defined.
    theValueOf(subscriptions,SUBSCRIPTIONLIST),
    member([SNISERVICEINFO],SUBSCRIPTIONLIST),
    $extractServiceParameters(SERVICE,SERVICEINFO,PARAMETERS), !.

getServiceParameters(SN,SERVICE,PARAMETERS) :-
    mainConsole-writeln("subscriber has not subscribed to service").

$subscriberNotDefined(SN) :-
    theValueOf(subscriberList,SL),
    not(member([SN|_],SL)),
    mainConsole-writeln("not a valid subscriber").

$invalidNotificationParameterValue(NotificationParameter) :-
    not(member(NotificationParameter,[yes,no])),
    mainConsole-(write("Invalid parameter value for Notification"),nl,
        write("Valid values are: yes or no"),nl).

$addService(SN,NEWSERVICEINFO) :-
    % Subscriber has other services already defined.
    theValueOf(subscriptions,SUBSCRIPTIONLIST),
    member([SN|CURRENTSERVICEINFO],SUBSCRIPTIONLIST),
    $updateServiceList(SN,NEWSERVICEINFO,CURRENTSERVICEINFO,
        SUBSCRIPTIONLIST),
    !.

$addService(SN,[NEWSERVICE|NEWSERVICEINFO]) :-
    % This is the first service subscriber has subscribed to. Add subscriber with
    % service information to the subscriptions list. Create a new supplementary
    % service database object instance which is required to store parameters
    % modified during the operation of the service.
    theValueOf(subscriptions,SUBSCRIPTIONLIST),
    addToList([SN,[NEWSERVICE|NEWSERVICEINFO]],SUBSCRIPTIONLIST,SL2),
    makeTheValueOf(subscriptions,SL2),
    supplServiceDatabase-new(OID,active,[]),
    directory-addObject(supplServiceDatabase,SN,OID),
    mainConsole-(write("Subscriber with number = "),write(SN),

```

```

        write(" has subscribed to service = "),
        write(NEWSERVICE),nl), !.

$updateServiceList(SN,[NEWSERVICE|_],CURRENTSERVICEINFO,
    SUBSCRIPTIONLIST) :-
    %check that service is not already subscribed to.
    member([NEWSERVICE|_],CURRENTSERVICEINFO),
    mainConsole-writeIn("Service is already subscribed to"), !.

$updateServiceList(SN,[NEWSERVICE|NEWSERVICEINFO],
    CURRENTSERVICEINFO,SUBSCRIPTIONLIST) :-
    subst([SN|CURRENTSERVICEINFO],SUBSCRIPTIONLIST,
        [SN,[NEWSERVICE|NEWSERVICEINFO]|CURRENTSERVICEINFO],
        UPDATEDLIST),
    makeTheValueOf(subscriptions,UPDATEDLIST),
    mainConsole-(write("Subscriber with number = "),write(SN),
        write(" has subscribed to service = "),
        write(NEWSERVICE),nl),
    !.

$extractServiceList([],[]).
$extractServiceList([_|SERVICE|_||L1],[SERVICE|L2]) :-
    $extractServiceList(L1,L2).

$extractServiceParameters(SERVICE,SERVICEINFO,PARAMETERS) :-
    member([SERVICE,PARAMETERS],SERVICEINFO), !.

$extractServiceParameters(SERVICE,SERVICEINFO,[]) :-
    not(member([SERVICE|PARAMETERS],SERVICEINFO)),
    mainConsole-writeIn("service is not subscribed to"), !.

$extractServiceParameters(SERVICE,SERVICEINFO,[]) :-
    % service is subscribed to, but there are no parameters associated with this
    % service.
    succeed.

$modifyTCbehaviour(SN,NEWSERVICE) :-
    directory-getObjectld(transactionControl,SN,OID),
    theValueOf(subscriberList,SL),
    member([SN,MAXASSOC,TRANSPORTER,TCCLASS],SL),
    $determineNewTCclass(TCCLASS,NEWSERVICE,NEWCLASS),
    mainConsole-(write("NEW Transaction Class for subscriber number "),
        write(SN),write(" is: "),
        write(NEWCLASS),nl),
    transporter-removeInstance(TRANSPORTER),
    TCCLASS-removeInstance(OID),
    NEWCLASS-new(OID,active,[subscriberNum=SN,
        maxAssoc=MAXASSOC]),

```

```

transporter-new(X,active,[sender=OID]),
subst([SN,MAXASSOC,TRANSPORTER,TCCLASS],SL,[SN,MAXASSOC,X,
NEWCLASS],UPDATEDLIST),
makeTheValueOf(subscriberList,UPDATEDLIST),
!.

```

```

$determineNewTCclass(transactionControl,cfnr,tcCallForwarding) :- !.

```

```

$determineNewTCclass(transactionControl,NEWSERVICE,NEWCLASS) :-
concatenate(tc,NEWSERVICE,NEWCLASS), !.

```

```

$determineNewTCclass(tcCallForwarding,cfu,tccfu) :- !.
$determineNewTCclass(tcCallForwarding,cfb,tccfb) :- !.

```

```

$determineNewTCclass(tccfu,cfb,tccfucfb) :- !.
$determineNewTCclass(tccfb,cfu,tccfucfb) :- !.

```

```

%*
%* C L A S S:  RESOURCE CONTROL
%*
%* This class co-ordinates access to the subscribers'
%* bearer channels. Two bearer channels, b1 and b2, are
%* defined for each subscriber access.
%*

```

```

class: resourceControl of object.

```

```

ivars: {accessList=[]}.

```

```

addAccess(NUMBER) :-
theValueOf(accessList,AL),
addToList([NUMBER,idle,idle],AL,AL2),
makeTheValueOf(accessList,AL2).

```

```

seizeBchannel(NUMBER,OBJECTID,CHANNELID) :-
theValueOf(accessList,AL),
member([NUMBER|CHANNELS],AL),
$channelAvailable(CHANNELS,CHANNELID),
$updateAccessList(NUMBER,CHANNELS,CHANNELID,OBJECTID,AL), !.

```

```

seizeBchannel(NUMBER,OBJECTID,noChannel) :- !.

```

```

releaseBchannel(NUMBER,CHANNELID) :-
theValueOf(accessList,AL),
member([NUMBER|CHANNELS],AL),

```

```

$updateAccessList(NUMBER,CHANNELS,CHANNELID,idle,AL).

$channelAvailable([idle,_],b1) :- !.
$channelAvailable([_,idle],b2).

$updateAccessList(NUMBER,CHANNELS,CHANNELID,VALUE,AL) :-
    $updateChannels(CHANNELID,VALUE,CHANNELS,NEWCHANNELS),
    subst([NUMBER|CHANNELS],AL,[NUMBER|NEWCHANNELS],AL2),
    makeTheValueOf(accessList,AL2).

queryAllAccesses(AL) :-
    theValueOf(accessList,AL).

querySubscriberAccess(NUMBER,ACCESS) :-
    theValueOf(accessList,AL),
    member([NUMBER|ACCESS],AL), !.

querySubscriberAccess(NUMBER,[]).
$updateChannels(b1,VALUE,[_,B2],[VALUE,B2]) :- !.
$updateChannels(b2,VALUE,[B1,_],[B1,VALUE]).

%*
%*      C L A S S :  D I R E C T O R Y  D A T A B A S E
%*
%*  This object stores the mapping between a specified key
%*  and the object identifier for various object classes.
%*  It provides methods for other objects to define the
%*  mappings and to retrieve the object identifier for a
%*  given object class and key.
%*

class: directoryDatabase of object.

ivars: [transactionControl = [],supplServiceDatabase=[]].

addObject(CLASS,KEY,OID) :-
% add the mapping between the KEY and the object identifier for a object
% instance of the specified class
    theValueOf(CLASS,OBJECTMAPPING),
    not(member([KEY|_],OBJECTMAPPING)),
    addToList([KEY,OID],OBJECTMAPPING,OBJECTMAPPING2),
    makeTheValueOf(CLASS,OBJECTMAPPING2), !.

addObject(CLASS,KEY,OID) :-
    mainConsole-(write("ERROR: Mapping between key = "),write(KEY),
        write(" and object identifier for object class= ")),

```

```

        write(CLASS),nl,
        write(" Cannot be added to the directory database"),nl).

getObjectId(CLASS,KEY,OID) :-
% get the object identifier for an object instance of specified class
  theValueOf(CLASS,OBJECTMAPPING),
  member([KEY,OID],OBJECTMAPPING), !.

getObjectId(CLASS,KEY,nilObject).

%*
%* C L A S S:  T I M E R
%*
%* This class provides the timer services for other objects.
%*

class: timer of object.

methods: class.

startTimer(SECONDS,INVOKER,REFERENCE) :-
  new(REFERENCE,active,[]),
  REFERENCE-startTimer(SECONDS,INVOKER),
  mainConsole-writeln("timer instance started").

stopTimer(REFERENCE) :-
  removeInstance(REFERENCE),
  mainConsole-writeln("timer instance deleted").

methods: instance.

begin :-
  waitToAccept(Message),
  postRendezvous(Message), !.

startTimer(SECONDS,INVOKER) :- succeed.

postRendezvous(startTimer(SECONDS,INVOKER)) :-
  delay(SECONDS),
  mainConsole-(write("delayed for: "),write(SECONDS),write(" seconds"),nl),
  myself(OID),
  INVOKER-timerExpired(OID).

```

```

%*
%*  C L A S S:  TRANSPORTER
%*
%*  This class provides message pickup and delivery services
%*  on behalf of a sender object (identified by the
%*  var: sender)
%*

```

class: transporter of object.

ivars: [sender].

```

begin :-
  repeat,
    theValueOf(sender,SENDER),
    SENDER-getMessage([TARGET,MESSAGE,PARAMETERS]),
    TARGET-sendMessage(MESSAGE,PARAMETERS),
  fail.

```

```

%*
%*  C L A S S:  MESSAGEQUEUE
%*
%*  This class provides a queue for other objects to deposit
%*  messages in and retrieve messages from
%*

```

class: messageQueue of object.

ivars: [myQueue=Q/Q].

```

storeMessage(DATA) :-
  theValueOf(myQueue,QUEUE),
  $enqueueItem(DATA,QUEUE,QUEUE2),
  makeTheValueOf(myQueue,QUEUE2).

```

```

getMessage(DATA) :-
  size(S), S>0,
  theValueOf(myQueue,QUEUE),
  $dequeueItem(DATA,QUEUE,QUEUE2),
  makeTheValueOf(myQueue,QUEUE2).

```

```

size(S) :-
  theValueOf(myQueue,Q/[ ]),
  listLength(Q,S).

```

\$enqueueItem(X,Q/[X|T],Q/T).

\$dequeueItem(X,[X|T]/T2,T/T2).

```
%*
%* C L A S S: DATAFILL
%*
%* This class specifies the initial configuration for the
%* telecommunication service prototype in terms of the
%* subscribers which are defined and their service
%* subscriptions
%*
```

class: dataFill of object.

methods: class.

loadSubscriberData :-

```
provision-addSubscriber(s1,7379356,4),
provision-addSubscriber(s2,8310888,3),
provision-addSubscriber(s3,911,1),
provision-addSubscriber(s4,411,3),
provision-subscribeToService(8310888,cfnr,[fred,no,2]),
provision-subscribeToService(8310888,callWaiting,[1,yes]),
provision-subscribeToService(411,callHold,_),
provision-subscribeToService(7379356,cfu,[linda,no,no]),
provision-subscribeToService(411,callWaiting,[1,no]),
provision-subscribeToService(7379356,cfb,[linda,no]).
```

```
%*
%* C L A S S: SUBSCRIBERSEND
%*
%* This class defines a number of methods which support
%* different service scenarios.
%*
```

class: subscriberSend of object.

basicsetup(FROM,TO) :- FROM-sendMessage(Setup,[0,TO]).

activateCFNR(X) :- s2-sendMessage(facility,[33,cfnr,activate,fred,X]).

deactivateCFNR :- s2-sendMessage(facility,[33,cfnr,deactivate,fred]).

activateCFU(X) :- s1-sendMessage(facility,[44,cfu,activate,linda,X]).


```

deactivateCFU :- s1-sendMessage(facility,[44,cfu,deactivate,linda]).
deactivateCFU :- s1-sendMessage(facility,[44,cfu,deactivate,linda]).
activateCFB(X) :- s1-sendMessage(facility,[44,cfb,activate,linda,X]).
deactivateCFB :- s1-sendMessage(facility,[44,cfb,deactivate,linda]).

```

```

%*
%*  C L A S S:  SUBSCRIBERRECEIVE
%*
%*  All messages generated by the objects which are destine
%*  to a subscriber will be sent to an instance of this class
%*  for logging on the mainConsole.
%*

```

class: subscriberReceive of object.

```

logMessage(SN,MESSAGETYPE,CLASS,OID,[]) :-
    mainConsole-(nl,write(MESSAGETYPE),
        write(" received by subscriber number:"),
        write(SN),write(" from Object: CLASS = "),
        write(CLASS),write(" IDENTIFIER = "),write(OID),nl(2)), !.

```

```

logMessage(SN,MESSAGETYPE,CLASS,OID,PARAMETERS):-
    mainConsole-(nl,write(MESSAGETYPE),
        write(" received by subscriber number:"),
        write(SN),write(" from Object: CLASS = "),
        write(CLASS),write(" IDENTIFIER = "),write(OID),
        nl,write(" Parameters included in message are: "),
        write(PARAMETERS),nl(2)), !.

```

```

%*
%*  C L A S S:  SUPP(lementary) SERVICE DATABASE
%*
%*  This object contains supplementary service parameters to
%*  which a number of objects require access. For example,
%*  ivar: num(ber) (of) waiting ass(ociations) is retrieved and
%*  updated by different instances of the Call Waiting Class.
%*

```

class: supplServiceDatabase of object.

```

ivars: [numWaitingAssoc = 0,cfunumber=undefined,cfbnumber=undefined,
    cfnrnumber=undefined].

```

```

querySupplServiceParameter(PARAMETER,VALUE) :-
    theValueOf(PARAMETER,VALUE), !.

querySupplServiceParameter(PARAMETER,undefined) :-
    mainConsole-(write("ERROR: unknown supplementary service parameter: "),
        write(PARAMETER),nl).

updateSupplServiceParameter(numWaitingAssoc,increment) :-
    theValueOf(numWaitingAssoc,N),
    N2 is N + 1,
    makeTheValueOf(numWaitingAssoc,N2), !.

updateSupplServiceParameter(numWaitingAssoc,decrement) :-
    theValueOf(numWaitingAssoc,N),
    N2 is N - 1,
    N2 @>= 0,
    makeTheValueOf(numWaitingAssoc,N2), !.

updateSupplServiceParameter(numWaitingAssoc,decrement) :- !.

updateSupplServiceParameter(PARAMETER,VALUE) :-
    makeTheValueOf(PARAMETER,VALUE), !.

updateSupplServiceParameter(PARAMETER,VALUE) :-
    mainConsole-(write("ERROR: unknown supplementary service parameter: "),
        write(PARAMETER),nl).

%*
%*   C L A S S:  T C   C A L L   H O L D
%*
%*   This object specifies the additional or delta behaviour
%*   introduced by the hold service.  It is a sub-class of
%*   the basic service class: Transaction Control (TC).
%*

class: tccallHold of transactionControl.

$decodeMessage(supplementaryServiceTransactionComplete,PARAMETERS) :-
    $supplementaryServiceTransactionComplete(PARAMETERS), !.

$decodeMessage(hold,REFERENCE) :-
    $holdService(REFERENCE), !.

$decodeMessage(retrieve,REFERENCE) :-
    $routeMessageToHoldObject(retrieve,REFERENCE), !.

```

4

OF/DE

4

MICRO

```

$decodeMessage(MESSAGETYPE,PARAMETERS) :-
%% This is not a hold service specific message.
    super-$decodeMessage(MESSAGETYPE,PARAMETERS), !.

$transactionComplete([bearer-associated,REFERENCE]) :-
% A bearer associated transaction has completed, decrement the number of
% associations. Remove any supplementary service objects, remove the Reference
% from the reference list.
    $decrementAssoc,
    $removeSupplementaryServiceObjects(REFERENCE),
    $removeFromRefList(REFERENCE), !.

$transactionComplete(PARAMETERS) :-
    super-$transactionComplete(PARAMETERS), !.

$supplementaryServiceTransactionComplete([CLASS,REFERENCE]) :-
% supplementary service object has terminated, remove from reference list.
    theValueOf(referenceList,REFLIST),
    member([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST),
    efface([CLASS,OID],OTHERSERVICES,OTHERSERVICES2),
    subst([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST,
        [REFERENCE,BASICSERVICEID|OTHERSERVICES2],REFLIST2),
    makeTheValueOf(referenceList,REFLIST2).

$holdService(REFERENCE) :-
% user has invoked the hold service, first verify that this is a valid
% reference.
    theValueOf(referenceList,REFLIST),
    not(member([REFERENCE|_],REFLIST)),
    theValueOf(subscriberNum,SN),
    myself(OID),
    subscriber-logMessage(SN,holdReject,tcHoldDelta,OID,
        "REASON = hold service invoked with an invalid reference"), !.

$holdService(REFERENCE) :-
% check that user is allowed to invoke the service, i.e. user has subscribed
% to service
    theValueOf(subscriberNum,SN),
    provision-getSubscriberServices(SN,LISTOFSERVICES),
    not(member(callHold,LISTOFSERVICES)),
    myself(OID),
    subscriber-logMessage(SN,holdReject,tcHoldDelta,OID,
        "REASON = subscriber has not subscribed to service"), !.

$holdService(REFERENCE) :-
% check that a hold service object is not already in existence (ie. this is
% a duplicate hold request). If a hold service object exists then send the
% hold message to that object for subsequent error handling.

```

```

theValueOf(referenceList,REFLIST),
member([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST),
member([holdService,HOLDID],OTHERSERVICES),
$storeMessage([HOLDID,hold,_]), !.

```

\$holdService(REFERENCE) :-

```

% create hold object. Update the reference list to include the
% holdService object identifier, so subsequent messages can be routed to that
% object.

```

```

myself(OID),
theValueOf(subscriberNum,SN),
theValueOf(referenceList,REFLIST),
member([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST),
holdService-new(HOLDID,active,[basicServiceID=BASICSERVICEID,tcId=OID,
                           subscriberNum=SN,reference=REFERENCE]),
subst([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST,
      [REFERENCE,BASICSERVICEID,[holdService,HOLDID]|OTHERSERVICES],
      REFLIST2),
makeTheValueOf(referenceList,REFLIST2),
$storeMessage([HOLDID,hold,_]), !.

```

\$routeMessageToHoldObject(MESSAGE,REFERENCE) :-

```

theValueOf(referenceList,REFLIST),
member([REFERENCE,BASICSERVICEID|OTHERSERVICES],REFLIST),
member([holdService,HOLDID],OTHERSERVICES),
$storeMessage([HOLDID,MESSAGE,_]), !.

```

\$routeMessageToHoldObject(MESSAGE,REFERENCE) :-

```

theValueOf(subscriberNum,SN),
myself(OID),
subscriber-logMessage(SN,releaseComplete,transactionControl,OID,
    " CAUSE = reference not in Use").

```

\$removeSupplementaryServiceObjects(REFERENCE) :-

```

% basic service has terminated, terminate any supplementary service
% objects that were created in conjunction with the basic service object.

```

```

theValueOf(referenceList,REFLIST),
member([REFERENCE,_|OTHERSERVICES],REFLIST),
$removeServiceInstances(OTHERSERVICES).

```

\$removeServiceInstances([]) :- succeed.

\$removeServiceInstances([[CLASS,OID]|L1]) :-

```

CLASS-removeInstance(OID), $removeServiceInstances(L1).

```

```

%*
%*  C L A S S:  H O L D  S E R V I C E
%*
%*  This class specifies the additional behaviour introduced
%*  by the hold service. An instance of this class
%*  communicates with an incoming or outgoing association
%*  (identified by the ivar: basicServiceID) to form the hold
%*  service.
%*

```

class: holdService of object.

ivars: [holdState=idle,basicServiceID,tcId,subscriberNum,reference].

begin :-

```

repeat,
waitToAccept(Message),
postRendezvous(Message),
theValueOf(holdState,terminated),
!.

```

```

postRendezvous(sendMessage(MESSAGE TYPE,PARAMETERS)) :-
theValueOf(holdState,CURRENTHOLDSTATE),
holdStateMachine(CURRENTHOLDSTATE,[MESSAGE TYPE,PARAMETERS],
NEWHOLDSTATE,
OUTPUT),
makeTheValueOf(holdState,NEWHOLDSTATE),
outputMessages(OUTPUT).

```

```

holdStateMachine(idle,[hold,_],NEWHOLDSTATE,OUTPUT) :-
holdRequest(NEWHOLDSTATE,OUTPUT), !.

```

```

holdStateMachine(held,[retrieve,_],NEWHOLDSTATE,OUTPUT) :-
retrieveRequest(NEWHOLDSTATE,OUTPUT), !.

```

```

holdStateMachine(CURRENTHOLDSTATE,[MESSAGE TYPE,_],
CURRENTHOLDSTATE,[]) :-

```

```

% Unexpected or Unrecognized Message received, stay in current state and
% print out a warning to the user.
myself(OID),
theValueOf(subscriberNum,SN),
subscriber-logMessage(SN,messageReject,holdService,OID,
"unexpected or unrecognized message "), !.

```

```

holdRequest(held,[]) :-

```

```

% For an outgoing association, the association may be placed on hold when
% the association is in the establishment phase (i.e.associationProceeding,

```

```

% associationDelivered or associated phase). For an incoming association,
% the association may be placed on hold only in the associated phase.
  theValueOf(basicServiceID,BASICSERVICEID),
  BASICSERVICEID-queryBasicServiceState(BASICSERVICESTATE),
  member(BASICSERVICESTATE,[associationProceeding,associationDelivered,
    associated]),
  BASICSERVICEID-releaseChannel,
  myself(OID),
  theValueOf(subscriberNum,SN),
  subscriber-logMessage(SN,holdAcknowledge,holdService,OID,[]), !.

```

```

holdRequest(terminated,[[OID,supplementaryServiceTransactionComplete,
  [holdService,REFERENCE]]]) :-

```

```

% reject hold request, basic service state is not in an allowed hold state.
% Inform the Transaction Control object that the hold object is terminated.
  theValueOf(tcId,OID),
  theValueOf(reference,REFERENCE),
  myself(HOLDID),
  theValueOf(subscriberNum,SN),
  subscriber-logMessage(SN,holdReject,holdService,HOLDID,
    "CAUSE = not in a valid state"), !.

```

```

retrieveRequest(held,[]) :-

```

```

% For an outgoing association, the association may be retrieved when
% the association is in the establishment phase (i.e.associationProceeding,
% associationDelivered or associated phase). For an incoming association,
% the association may be retrieved only in the associated phase.
% If is not in the appropriate state, then the retrieve request is rejected
% and the association remains in the held state.
  theValueOf(basicServiceID,BASICSERVICEID),
  BASICSERVICEID-queryBasicServiceState(BASICSERVICESTATE),
  not(member(BASICSERVICESTATE,[associationProceeding,associationDelivered,
    associated])),
  myself(HOLDID),
  theValueOf(subscriberNum,SN),
  subscriber-logMessage(SN,retrieveReject,holdService,HOLDID,
    "CAUSE= not in a valid state"),
  !.

```

```

retrieveRequest(held,[]) :-

```

```

% Association is in a valid state, but there is no available channel to
% retrieve the association. Association remains in held state.
  theValueOf(basicServiceID,BASICSERVICEID),
  BASICSERVICEID-seizeChannel(CHANNELID),
  CHANNELID == noChannel,
  myself(HOLDID),
  theValueOf(subscriberNum,SN),
  subscriber-logMessage(SN,retrieveReject,holdService,HOLDID,

```

```

        " CAUSE= no available channel"),
    !.

retrieveRequest(terminated,[[OID,supplementaryServiceTransactionComplete,
    [CLASS,REFERENCE]]]) :-
% Retrieve request is successful. Inform Transaction control object that
% hold object has terminated.
    theValueOf(tcId,OID),
    theValueOf(reference,REFERENCE),
    myself(HOLDID),
    theValueOf(subscriberNum,SN),
    subscriber-logMessage(SN,retrieveAcknowledge,holdService,HOLDID,[]).

```

```

%*
%*  C L A S S:  SERVICE ACTIVATION CONTROL
%*
%*  This class specifies the behaviour which is common to all
%*  services requiring the capability to activate/deactivate
%*  the service for a specified period of time. This is an
%*  abstract class since an instance of this class is never
%*  created.
%*

```

```

class: serviceActivationControl of object.

```

```

ivars: [activationState = idle].

```

```

begin :-
    repeat,
    waitToAccept(Message),
    postRendezvous(Message),
    theValueOf(activationState,terminated),
    !.

```

```

postRendezvous(sendMessage(MESSAGETYPE,PARAMETERS)) :-
    $decodeMessage(MESSAGETYPE,PARAMETERS).

```

```

$decodeMessage(facility,[REFERENCE,PROCEDURE|OTHERPARAMETERS]) :-
    $invokeProcedure(REFERENCE,PROCEDURE,OTHERPARAMETERS), !.

```

```

$decodeMessage(MESSAGETYPE,[REFERENCE,PROCEDURE|OTHERPARAMETER
S]) :-
    mainConsole-writeLn("unrecognized message").

```



```

$invokeProcedure(REFERENCE,activate,OTHERPARAMETERS) :-
    $activateService(REFERENCE,OTHERPARAMETERS), !.

$invokeProcedure(REFERENCE,deactivate,OTHERPARAMETERS) :-
    $deactivateService(REFERENCE,OTHERPARAMETERS), !.

$invokeProcedure(REFERENCE,interrogate,OTHERPARAMETERS) :-
    $interrogate(REFERENCE,OTHERPARAMETERS), !.

$invokeProcedure(REFERENCE,PROCEDURE,OTHERPARAMETERS) :-
    mainConsole-writeln("invalid service activation procedure").

$activateService(REFERENCE,OTHERPARAMETERS).

$deactivateService(REFERENCE,OTHERPARAMETERS).

$interrogate(REFERENCE,OTHERPARAMETERS).

```

```

%*
%*  C L A S S: CF (CALL FORWARDING) SERVICE
%*  ACTIVATION CONTROL
%*
%*  This class specifies the behaviour to activate/deactivate
%*  the call forwarding services. This is a sub-class of
%*  the service activation control class.
%*

```

```

class: cfServiceActivationControl of serviceActivationControl.

```

```

ivars: [cfType,subscriberNum].

```

```

$activateService(REFERENCE,[PASSWORD,ForwardingAddress]) :-
% if a valid password is provided then update the forwarding address
% in the supplementary service database. A currently active forwarding
% address will be overwritten.
    $validPassword(PASSWORD),
    $updateCfNumber(ForwardingAddress),
    theValueOf(subscriberNum,SN),
    myself(OID),
    subscriber-logMessage(SN,cfActivateAcknowledge,cfServiceActivationControl,
        OID,ForwardingAddress),
    $activationControlTransactionComplete(REFERENCE), !.

```

```

$activateService(REFERENCE,[PASSWORD,ForwardingAddress]) :-
% invalid password.
    theValueOf(subscriberNum,SN),
    myself(OID),

```

```

subscriber-logMessage(SN,cfActivateReject,cfServiceActivationControl,
    OID," invalid password"),
$activationControlTransactionComplete(REFERENCE).

```

```

$deactivateService(REFERENCE,[PASSWORD]) :-
% if a valid password is provided then update the forwarding address
% in the supplementary service database to not defined.
    $validPassword(PASSWORD),
    $updateCfNumber(undefined,
        theValueOf(subscriberNum,SN),
        myself(OID),
        subscriber-logMessage(SN,cfDeactivateAcknowledge,cfServiceActivationControl,
            OID,[]),
        $activationControlTransactionComplete(REFERENCE), !).

```

```

$deactivateService(REFERENCE,[PASSWORD]) :-
    theValueOf(subscriberNum,SN),
    myself(OID),
    subscriber-logMessage(SN,cfDeactivateAcknowledge,cfServiceActivationControl,
        OID," invalid password"),
    $activationControlTransactionComplete(REFERENCE).

```

```

$validPassword(PASSWORD) :-
    theValueOf(subscriberNum,SN),
    theValueOf(cftype,SERVICE),
    provision-getServiceParameters(SN,SERVICE,[SubscribedPASSWORDI_]),
    SubscribedPASSWORD == PASSWORD.

```

```

$updateCfNumber(CfNumber) :-
    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,OID),
    theValueOf(cftype,CFTYPE),
    concatenate(CFTYPE,number,CFPARAMETER),
    OID-updateSupplServiceParameter(CFPARAMETER,CfNumber).

```

```

$activationControlTransactionComplete(REFERENCE) :-
    theValueOf(subscriberNum,SN),
    makeTheValueOf(activationState,terminated),
    directory-getObjectId(transactionControl,SN,TC),
    outputMessages([[[TC,transactionComplete,
        [nonBearerAssociated,REFERENCE]]]), !.

```

% *

% * C L A S S: TC CALL FORWARDING

% *

**% * This object specifies the "common" additional behaviour
 % * introduced by the call forwarding services (cfu, cfb
 % * cfnr). This object is a sub-class of the basic service
 % * object: transaction control (TC). It is a superclass of**

```

%* the objects tccfu (tc call forwarding unconditional),
%* and tccfb (call forwarding busy).
%*

```

```

class: tcCallForwarding of transactionControl.

```

```

$decodeMessage(facility,PARAMETERS) :-
%% a new messagetype is introduced by the call forwarding service.
$facilityRequest(PARAMETERS), !.

```

```

$decodeMessage(MESSAGETYPE,PARAMETERS) :-
%% if this is not a specific call forwarding message then handle as in
%% basic service.
super-$decodeMessage(MESSAGETYPE,PARAMETERS).

```

```

$facilityRequest([REFERENCE,SERVICETYPE,
REQUESTTYPE|OTHERPARAMETERS]) :-
% check that subscriber has subscribed to service.
theValueOf(subscriberNum,SN),
provision-getSubscriberServices(SN,LISTOFSERVICES),
not(member(SERVICETYPE,LISTOFSERVICES)),
myself(OID),
subscriber-logMessage(SN,facilityReject,transactionControl,OID,
"REASON = subscriber has not subscribed to service"), !.

```

```

$facilityRequest([REFERENCE,SERVICETYPE,REQUESTTYPE|OTHERPARAMETER
S]) :-
% This is call forwarding activation/deactivation request. Create a
% call forwarding service activation control object instance to handle
% request.
member(SERVICETYPE,[cfu,cfb,cfnr]),
member(REQUESTTYPE,[activate,deactivate]),
theValueOf(subscriberNum,SN),
cfServiceActivationControl-new(CFID,active,
[subscriberNum=SN,cftype=SERVICETYPE]),
$addToRefList(REFERENCE,CFID),
$storeMessage([CFID,facility,[REFERENCE,
REQUESTTYPE|OTHERPARAMETERS]]),
!.

```

```

$facilityRequest([REFERENCE,SERVICETYPE,REQUESTTYPE|OTHERPARAMETER
S]) :-
theValueOf(subscriberNum,SN),
myself(OID),
subscriber-logMessage(SN,facilityReject,transactionControl,OID,
"REASON = unrecognized facility request message").

```

```

$forwardAssociation(ACid,CFTYPE) :-
%% this method in common to the cfu and cfb. It specifies the forwarding
%% of the association when the cfu or cfb service is active.
    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,OID),
    concatenate(CFTYPE,number,CFNUMBER),
    OID-querySupplServiceParameter(CFNUMBER,ForwardingNumber),
    ForwardingNumber \== undefined,
    myself(TC),
    $storeMessage([ACid,rejectIndication,
        [TC,reroute,CFTYPE,ForwardingNumber]]),
    mainConsole-(write(CFTYPE),write(" invoked"),nl).

```

```

%*
%*  C L A S S:  T C  C F U
%*
%*  This object specifies the additional or delta behaviour
%*  introduced by the call forwarding unconditional service.
%*  It is a sub-class of the TCcallForwarding class.
%*

```

class: tcclu of tcCallForwarding.

```

$setupRequest([ACid,CALLINGNUMBER]) :-
% If call forwarding unconditional is activated then the association will
% be forwarded otherwise it will be handled as in basic service.
    $forwardAssociation(ACid,cfu), !.

```

```

$setupRequest([ACid,CALLINGNUMBER]) :-
    super-$setupRequest([ACid,CALLINGNUMBER]).

```

```

%*
%*  C L A S S:  T C  C F B
%*
%*  This object specifies the additional or delta behaviour
%*  introduced by the call forwarding busy service.
%*  It is a sub-class of the TCcallForwarding class.
%*

```

class: tcclb of tcCallForwarding.

```

$handleMaxAssocReached(ACid) :-
% If call forwarding busy is activated then the association will be
% forwarded, otherwise the association will be handled as described for basic
% service.
    $forwardAssociation(ACid,cfb), !.

```

**\$handleMaxAssocReached(ACid) :-
super-\$handleMaxAssocReached(ACid), !.**

**%*
%* CLASS: TC CFU CFB
%*
%* This object specifies the additional or delta behaviour
%* introduced by the call forwarding unconditional and
%* call forwarding busy services. It represents a union of
%* the tccfu and tccfb classes
%***

class: tccfucfb of tccfu.

**\$handleMaxAssocReached(ACid) :-
% This is a network-determined busy condition
% If call forwarding busy is activated then the association will be
% forwarded, otherwise the association will be handled as described for basic
% service.
\$forwardAssociation(ACid,cfb), !.**

**\$handleMaxAssocReached(ACid) :-
super-\$handleMaxAssocReached(ACid), !.**

**%*
%* CLASS: AC CFU
%*
%* This object includes the additional behaviour required
%* to support the call forwarding unconditional service.
%* This object is a sub-class of the basic service object:
%* association control (AC).
%***

class: acCFU of associationControl.

**%%%%%%%%%% THE CFU SERVICE AFFECTS THE "waitForResponderID"
transition**

**waitForResponderId([rejectIndication,[OID,reroute,CFTYPE,ForwardingNumber]],
waitForResponderId,[FORWARDINGID,setupRequest,[ACID,CL]])) :-**

**% The behaviour of the basic association control object is modified if a
% rejectIndication is received in the waitForResponderId state
% with a reason of: reroute. Upon receipt of this indication, the acForwarding
% object will attempt to reroute the association to the specified
% forwarding number. If the attempt to forward the association fails then**

% the rejectIndication is handled as for the basic service (i.e. the
% association is released.

reroute(ForwardingNumber,FORWARDINGID,CL,ACID),
!.

waitForResponderId([MESSAGETYPE,PARAMETERS],NEXTSTATE,OUTPUT) :-
% handle as for the basic behaviour

super-
waitForResponderId([MESSAGETYPE,PARAMETERS],NEXTSTATE,OUTPUT).

reroute(ForwardingNumber,FORWARDINGID,CL,ACID) :-
directory-getObjectId(transactionControl,ForwardingNumber,FORWARDINGID),
FORWARDINGID \= nilObject,
myself(ACID),
theValueOf(callingNumber,CL).

/*

/* CLASS: AC CFB

/*

/* This object includes the additional behaviour required

/* to support the call forwarding busy service.

/* This object is a sub-class of the AC CFU class.

/*

class: acCFB of acCFU.

%%%%%%%%%% THE CFB SERVICE AFFECTS THE "releaseAssociation" method

releaseAssociation([OID,reroute,cfb,ForwardingNumber],waitForResponderId,
[[OID,releaseResponse,_],[FORWARDINGID,setupRequest,[ACID,CL]]]) :-
reroute(ForwardingNumber,FORWARDINGID,CL,ACID),
!.

releaseAssociation(PARAMETERS,NEXTSTATE,OUTPUT) :-
super-releaseAssociation(PARAMETERS,NEXTSTATE,OUTPUT), !.

/*

/* CLASS: AC CFNR

/*

/* This object includes the additional behaviour required

/* to support the call forwarding no reply service.

/* This object is a sub-class of the basic service object:

/* associationControl

%*

class: acCFNR of associationControl.

ivars: [forwardedId].

```
acStateMachine(responderReleaseForwardPending,[MESSAGE,PARAMETERS],
NEXTSTATE,OUTPUT) :-
    responderReleaseForwardPending([MESSAGE,PARAMETERS],NEXTSTATE,
OUTPUT),
    !.
```

```
acStateMachine(CURRENTSTATE,[MESSAGE,PARAMETERS],NEXTSTATE,
OUTPUT) :-
    super-acStateMachine(CURRENTSTATE,[MESSAGE,PARAMETERS],
NEXTSTATE,OUTPUT).
```

```
waitForSetupConfirm([forwardingIndication,[OID,cfnr,ForwardingNumber]],
responderReleaseForwardPending,
[[OID,releaseRequest,_]]) :-
%% if can reroute to the specified number, then first release forwarding
%% responder before forwarding association.
    directory-getObjectId(transactionControl,ForwardingNumber,X),
    X \== nilObject,
    makeTheValueOf(forwardedId,X), !.
```

```
waitForSetupConfirm([forwardingIndication,_],waitForSetupConfirm,[]) :-
%% can't forward to the forwarding number, stay in current state.
    !.
```

```
waitForSetupConfirm([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT) :-
    super-
waitForSetupConfirm([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT).
```

```
responderReleaseForwardPending([releaseIndication,[OID,REASON|_]],
    waitForReleaseConfirm,[[OID,releaseResponse,_]]) :-
%% invoker is releasing association, don't forward association,
%% wait for forwarding responder to send release confirm.
    theValueOf(invokerId,OID), !.
```

```
responderReleaseForwardPending([releaseConfirm,OID],
    waitForResponderId,[[FORWARDEDID,setupRequest,[ACID,CL]]]) :-
% forward association now, forwarding responder has been released.
    theValueOf(responderId,OID),
    theValueOf(forwardedId,FORWARDEDID),
    myself(ACID),
    theValueOf(callingNumber,CL), !.
```

```

%*
%*   C L A S S: CFB (Call Forwarding Busy)
%*
%*   This class specifies the delta behaviour introduced by the
%*   call forwarding busy service on the incoming association
%*   object. It is a sub-class of the incoming association
%*   class.
%*

```

class: cfb of incomingAssociation.

```

disestablishOnReleaseComplete(waitForUserResponse,subscriberBusy,
    waitForReleaseResponse,[[ACID,releaseIndication,[OID,reroute,cfb,
        ForwardingNumber]]]) :-
    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,X),
    X-querySupplServiceParameter(cfbnumber,ForwardingNumber),
    ForwardingNumber \== undefined,
    theValueOf(acId,ACID),
    myself(OID),
    releaseChannel.

```

```

disestablishOnReleaseComplete(CURRENTSTATE,PARAMETERS,NEXTSTATE,
    OUTPUT) :-
    super-disestablishOnReleaseComplete(CURRENTSTATE,PARAMETERS,
        NEXTSTATE,OUTPUT).

```

```

%*
%*   C L A S S: CFNR (Call Forwarding No Reply)
%*
%*   This class specifies the delta behaviour introduced by the
%*   call forwarding no reply service on the incoming association
%*   object. It is a sub-class of the incoming association class.
%*

```

class: cfnr of incomingAssociation.

```

waitForUserResponse([alerting,_],userResponded,
    [[ACID,report,alerting]]) :-
    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,X),
    X-querySupplServiceParameter(cfnrnumber,ForwardingNumber),
    ForwardingNumber \== undefined,

```



```

provision-getServiceParameters(SN,cfnr,[_,_,TimerValue]),
myself(OID),
timer-startTimer(TimerValue,OID,REFERENCE),
theValueOf(activeTimer,ACTIVETIMER),
addToList([cfnrTimer,REFERENCE],ACTIVETIMER,ACTIVETIMER2),
mainConsole-(write("CFNR timer started for :"),write(TimerValue),
write(" seconds"), nl),
makeTheValueOf(activeTimer,ACTIVETIMER2),
theValueOf(acId,ACID), !.

waitForUserResponse([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT) :-
super-waitForUserResponse([MESSAGE,PARAMETERS],NEXTSTATE,
OUTPUT).

userResponded([connect,_,associated,[[ACID,setupConfirm,_]]) :-
theValueOf(activeTimer,ACTIVETIMER),
member([cfnrTimer,REFERENCE],ACTIVETIMER),
timer-stopTimer(REFERENCE),
efface([cfnrTimer,REFERENCE],ACTIVETIMER,ACTIVETIMER2),
makeTheValueOf(activeTimer,ACTIVETIMER2),
theValueOf(acId,ACID),
mainConsole-writeln("CFNR timer stopped"), !.

userResponded([timerExpired,TIMERREFERENCE],userResponded,
[[ACID,forwardingIndication,[OID,cfnr,ForwardingNumber]]) :-
theValueOf(subscriberNum,SN),
directory-getObjectId(supplServiceDatabase,SN,X),
X-querySupplServiceParameter(cfnrnumber,ForwardingNumber),
ForwardingNumber \== undefined,
theValueOf(activeTimer,ACTIVETIMER),
efface([cfnrTimer,TIMERREFERENCE],ACTIVETIMER,ACTIVETIMER2),
makeTheValueOf(activeTimer,ACTIVETIMER2),
myself(OID),
theValueOf(acId,ACID), !.

userResponded([timerExpired,TIMERREFERENCE],userResponded, []) :-
theValueOf(activeTimer,ACTIVETIMER),
efface([cfnrTimer,TIMERREFERENCE],ACTIVETIMER,ACTIVETIMER2),
makeTheValueOf(activeTimer,ACTIVETIMER2), !.

userResponded([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT) :-
super-userResponded([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT).

%*
%* CLASS: C . . . WAITING
%*
%* This object specifies the additional or delta behaviour
%* introduced by the call waiting service. It is a sub-class

```

**%* of the basic service object: incomingAssociation.
%***

class: callWaiting of incomingAssociation.

ivars: [waiting = false].

stateMachine(cwReleaseInitiated,[MESSAGE,PARAMETERS],NEWSTATE,OUTPUT) :-

**% the call waiting service modifies the basic service state machine as it
% adds a new state - call waiting (cw) release initiated. This is as a result
% of an error detected by the call waiting service
% which must initiate the disestablishment of the association.**

cwReleaseInitiated([MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

stateMachine(CURRENTSTATE,[MESSAGE,PARAMETERS],NEWSTATE,OUTPUT) :-

**%% all other states handle as for basic service
super-**

stateMachine(CURRENTSTATE,[MESSAGE,PARAMETERS],NEWSTATE,OUTPUT), !.

idle([setupRequest,CALLINGNUMBER],waitForUserResponse,[]) :-

**% B-channel is available, handle identical to basic service procedures
 seizeChannel(CHANNELID),
 CHANNELID \== noChannel,
 \$performEstablishIncomingAssocActions, !.**

idle([setupRequest,CALLINGNUMBER],waitForUserResponse,[]) :-

**% B-channel is not available and maximum waiting associations
% has not been reached. Establish Incoming Association to subscriber
% with no B-channel
 not(\$maxWaitingAssoc),
 mainConsole-writeln("call waiting invoked"),
 \$incrementWaitingAssoc,
 \$performEstablishIncomingAssocActions, !.**

**idle([setupRequest,CALLEDNUMBER],waitForReleaseResponse,
 [[ACID,releaseIndication,[OID,maxCallWaitingAssoc]]]) :-**

**% Maximum waiting associations, call waiting cannot be invoked.
 myself(OID),
 mainConsole-(write("call waiting not invoked, "),
 write("maximum number of waiting associations reached"),nl),
 theValueOf(acId,ACID), !.**

waitForUserResponse([alerting,_],userResponded,[[ACID,report,alerting]]) :-

% this is not a waiting association, handle as for basic service

theValueOf(waiting,false),
theValueOf(acId,ACID), !.

waitForUserResponse([alerting,_],userResponded,[[ACID,report,alerting]]) :-
% this is a waiting association, check call waiting notification parameter.
% The Call Waiting Notification Parameter indicates that invoking subscriber
% SHOULD NOT be notified that the association is being handled according to the
% call waiting service procedures.
theValueOf(acId,ACID),
theValueOf(subscriberNum,SN),
provision-getServiceParameters(SN,callWaiting,[_,CWNOTIFICATION]),
CWNOTIFICATION == no, !.

waitForUserResponse([alerting,_],userResponded,
[[ACID,report,[alerting,callWaiting]])]):-
% Invoking subscriber should be notified of the waiting condition.
theValueOf(acId,ACID).

waitForUserResponse([connect,_],associated,
[[ACID,setupConfirm,_]]) :-
% this is not a waiting association, handle as for basic service.
theValueOf(waiting,false),
theValueOf(acId,ACID), !.

waitForUserResponse([connect,_],associated,
[[ACID,setupConfirm,_]]) :-
% this is a waiting association,
% check that a channel is now available for the association. In this case
% a channel can be allocated to the waiting association.
theValueOf(acId,ACID),
seizeChannel(CHANNELID),
CHANNELID \== noChannel,
\$decrementWaitingAssoc, !.

waitForUserResponse([connect,_],cwReleaseInitiated,
[[ACID,releaseIndication,[OID,noChannelAvailable]])]):-
% this is a waiting association, there was no channel available to
% allocate to the waiting association, send a release indication to the
% association control object (to release originating user).
% Send disconnect to the responding subscriber when a releaseResponse is
% received.
theValueOf(acId,ACID),
myself(OID), !.

userResponded([connect,_],associated,
[[ACID,setupConfirm,_]]) :-
% this is not a waiting association, handle as for basic service.
theValueOf(waiting,false),

```

theValueOf(acId,ACID), !.

userResponded([connect,_],associated,
  [[ACID,setupConfirm,_]]) :-
% check that a channel is now available for the association.
  theValueOf(acId,ACID),
  seizeChannel(CHANNELID),
  CHANNELID \== noChannel,
  $decrementWaitingAssoc, !.

userResponded([connect,_],cwReleaseInitiated,
  [[ACID,releaseIndication,[OID,noChannelAvailable]])] :-
  myself(OID),
  theValueOf(acId,ACID).

cwReleaseInitiated([MESSAGE,_,_],disconnectRequest,[]) :-
% association control object has responded with a release response, send
% disconnect to subscriber and return to the basic service
% state: "disconnectRequest"
  member(MESSAGE,[releaseResponse,releaseRequest]),
  $sendMessageToSubscriber(callWaiting,disconnect,
    "No channel available for waiting association"), !.

cwReleaseInitiated([releaseComplete,_],waitForReleaseResponse,[]) :-
% subscriber has sent a releaseComplete message while the call waiting service
% is awaiting a reply from the association control object. Return to the
% basic service state: "waitForReleaseResponse".
  releaseChannel, !.

cwReleaseInitiated([release,_],releaseIndication,[]) :- !.
% subscriber has sent a release message while the call waiting service
% is awaiting a reply from the association control object. Return to the
% basic service state: "releaseIndication"

cwReleaseInitiated([disconnect,_],disconnectIndication,[]) :- !.
% subscriber has sent a disconnect message while the call waiting service
% is awaiting a reply from the association control object. Return to the
% basic service state: "disconnectIndication"

$maxWaitingAssoc :-
% This method succeeds if the current number of waiting associations (maintained
% by the supplServiceDatabase object) is equal to the maximum number of waiting
% associations subscribed to by this subscriber (this subscription parameter is
% maintained by the provision object).
  theValueOf(subscriberNum,SN),
  provision-getServiceParameters(SN,callWaiting,[MAXWAITING,_]),
  directory-getObjectId(supplServiceDatabase,SN,OID),
  OID-querySupplServiceParameter(numWaitingAssoc,NUMWAITING),

```

NUMWAITING == MAXWAITING.

\$associationReleased([X,transactionComplete,[bearerAssociated,REFERENCE]]) :-
% This method overrides the basic service method. When the association
% is released, the number of waiting associations will be decremented if
% this is a waiting association.
 theValueOf(waiting,true),
 \$decrementWaitingAssoc,
 theValueOf(tcId,X),
 theValueOf(reference,REFERENCE), !.

\$associationReleased([X,transactionComplete,[bearerAssociated,REFERENCE]]) :-
% Not a waiting association, handle as in basic service.
 super-\$associationReleased([X,transactionComplete,
 [bearerAssociated,REFERENCE]]).

\$incrementWaitingAssoc :-
 makeTheValueOf(waiting,true),
 theValueOf(subscriberNum,SN),
 directory-getObjectId(supplServiceDatabase,SN,OID),
 OID-updateSupplServiceParameter(numWaitingAssoc,increment).

\$decrementWaitingAssoc :-
 makeTheValueOf(waiting,false),
 makeTheValueOf(waiting,false),
 theValueOf(subscriberNum,SN),
 directory-getObjectId(supplServiceDatabase,SN,OID),
 OID-updateSupplServiceParameter(numWaitingAssoc,decrement).

%*
%* C L A S S: C A L L W A I T I N G / C F N R
%*
%* This class specifies the delta behaviour introduced by the
%* combined call waiting and CFNR services. It is a sub-class
%* of the call waiting class.
%*

class: callWaitingCfnr of callWaiting.

waitForUserResponse([alerting,...],userResponded,[ACID,report,alerting]]) :-
% this is not a waiting association, handle as for basic service
% start cfnr timer if CFNR service is activated.
 theValueOf(waiting,false),
 theValueOf(acId,ACID),
 \$startCfnrTimerIfActive, !.

```

waitForUserResponse([alerting,_],userResponded,[[ACID,report,alerting]]) :-
% this is a waiting association, check call waiting notification parameter.
% The Call Waiting Notification Parameter indicates that invoking subscriber
% SHOULD NOT be notified that the association is being handled according to the
% call waiting service procedures.
% start cfmr timer if CFNR service is activated.
  theValueOf(acId,ACID),
  theValueOf(subscriberNum,SN),
  provision-getServiceParameters(SN,callWaiting,[_,CWNOTIFICATION]),
  CWNOTIFICATION == no,
  $startCfmrTimerIfActive, !.

```

```

waitForUserResponse([alerting,_],userResponded,
  [[ACID,report,[alerting,callWaiting]])]):-
% Invoking subscriber should be notified of the waiting condition.
% start cfmr timer if CFNR service is activated.
  theValueOf(acId,ACID),
  $startCfmrTimerIfActive.

```

```

waitForUserResponse([MESSAGE,PARAMETERS],NEXTSTATE,OUTPUT) :-
  super-waitForUserResponse([MESSAGE,PARAMETERS],NEXTSTATE,
    OUTPUT).

```

```

userResponded([connect,_],associated,
  [[ACID,setupConfirm,_]]) :-
% this is not a waiting association, handle as for basic service.
% stop CFNR timer if active.
  theValueOf(waiting,false),
  theValueOf(acId,ACID),
  $stopCfmrTimer, !.

```

```

userResponded([connect,_],associated,
  [[ACID,setupConfirm,_]]) :-
% check that a channel is now available for the association.
  theValueOf(acId,ACID),
  seizeChannel(CHANNELID),
  CHANNELID \== noChannel,
  $decrementWaitingAssoc,
  $stopCfmrTimer, !.

```

```

userResponded([connect,_],cwReleaseInitiated,
  [[ACID,releaseIndication,[OID,noChannelAvailable]])]):-
  myself(OID),
  theValueOf(acId,ACID),
  $stopCfmrTimer, !.

```

```

userResponded([timerExpired,TIMERREFERENCE],userResponded,

```

```

    [[(ACID,forwardingIndication,[OID,cfnr,ForwardingNumber]])] :-
    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,X),
    X-querySupplServiceParameter(cfnrnumber,ForwardingNumber),
    ForwardingNumber \== undefined,
    theValueOf(activeTimer,ACTIVETIMER),
    efface([cfnrTimer,TIMERREFERENCE],ACTIVETIMER,ACTIVETIMER2),
    makeTheValueOf(activeTimer,ACTIVETIMER2),
    myself(OID),
    theValueOf(acId,ACID), !.

```

```

userResponded([timerExpired,TIMERREFERENCE],userResponded, []) :-
    theValueOf(activeTimer,ACTIVETIMER),
    efface([cfnrTimer,TIMERREFERENCE],ACTIVETIMER,ACTIVETIMER2),
    makeTheValueOf(activeTimer,ACTIVETIMER2).

```

\$startCfnrTimerIfActive :-

%% this local method will check whether the CFNR service is
 %% activated and if so, will start the cfnr timer.

```

    theValueOf(subscriberNum,SN),
    directory-getObjectId(supplServiceDatabase,SN,X),
    X-querySupplServiceParameter(cfnrnumber,ForwardingNumber),
    ForwardingNumber \== undefined,
    provision-getServiceParameters(SN,cfnr,[_,_,TimerValue]),
    myself(OID),
    timer-startTimer(TimerValue,OID,REFERENCE),
    theValueOf(activeTimer,ACTIVETIMER),
    addToList([cfnrTimer,REFERENCE],ACTIVETIMER,ACTIVETIMER2),
    mainConsole-(write("CFNR timer started for :"),write(TimerValue),
        write(" seconds"), nl),
    makeTheValueOf(activeTimer,ACTIVETIMER2),
    !.

```

\$startCfnrTimerIfActive :- succeed.

%% CFNR service is not activated so don't start timer, do nothing.

\$stopCfnrTimer :-

%% if cfnr timer is active, then stop this timer and remove from active
 %% timer list.

```

    theValueOf(activeTimer,ACTIVETIMER),
    member([cfnrTimer,REFERENCE],ACTIVETIMER),
    timer-stopTimer(REFERENCE),
    efface([cfnrTimer,REFERENCE],ACTIVETIMER,ACTIVETIMER2),
    makeTheValueOf(activeTimer,ACTIVETIMER2),
    mainConsole-writeln("CFNR timer stopped"), !.

```

\$stopCfnrTimer :- succeed.

APPENDIX D

Sample Traces from MLog Telecommunication Service Prototype

This appendix provides some sample traces for a number of different service scenarios output from the MLog Telecommunication Service Prototype. The MLog Telecommunication Service Prototype was run using the MLog TTY interface. The TTY interface enables the user to converse with MLog from a conventional terminal or PC. The prototype was run using the MLog timeslicing option set to 15.

D.1 Invoking MLog and Initialization of MLog Prototype

MLog is invoked by the Unix shell script `mlog` [KARA 89b]. The shell script provides a number of options for selecting the interpreter behaviour. The TTY interface is selected by specifying the `-i tty` option. Timeslicing is set to 15 by specifying the `-t 15` option. If timeslicing is set to 0 or less, the "run until block" processor sharing is used. Other options available, but not required to run the MLog Telecommunication Service Prototype, are described in [KARA 89b].

When MLog starts up, a conventional "Prolog-like" read-eval-write loop commences. The user then enters MLog messages and terminates them with a period. The result of the evaluation is then printed out.

Several special "built-in" MLog messages have been provided as follows:

- load(Filename).** - This is used to load in a file called **Filename.oop**.
- run.** - This will create an active instance of class **main**.
- bye.** - This will exit MLog and return to Unix.

Figure D.1 provides a screen dump illustrating the invocation of MLog, and the loading and initialization of the MLog Telecommunication Service Prototype. Comments have been

added on the left handed side for understanding and do not constitute output from the MLog prototype. User messages input from the MLog console are shown in bold.

The "all initialized" message is printed on the MLog console after the key basic service objects of the MLog Telecommunication Service Prototype have been created (see Section 6.3 for a list of objects which are created by the **main** object). A series of messages are then printed on the MLog console indicating the subscribers that have been defined and the supplementary services to which the subscribers have subscribed. When a supplementary service is added which requires that an active instance of the transaction control object be replaced with a new instance of the appropriate "TC supplementary service delta" object class (see Section 6.3), then a message is printed on the console indicating the new class.

The following identifies the subscriber data added by the MLog Telecommunication Service Prototype during its initialization:

Subscriber Number	Object Identifier of Transaction Control Object associated with subscriber	Supplementary Services Subscribed
7379356	s1	cfu, cfb
8310888	s2	call waiting, cfmr
411	s3	call waiting, call hold
911	s4	

After the MLog Telecommunication Service Prototype has been initialized and the subscriber data added, the user can then invoke a number of different service scenarios by entering messages at the MLog tty console. Traces from different service scenarios are included in Sections D.2 to D.5, inclusive.

MLog invoked with
tty interface option
and timeslicing set
to 15.

load in file.
create active
instance of main
Basic service
objects created
Add four subscribers

Add supplementary
services.
Change TC class

All subscriber data
added

15# lmllog
mlog -i tty -t 15
Running MLog v1.2 under SUN OS 3.5 using tty interface

MLog (C) Copyright 1989 Carleton University

MLog? load(telecom).
MLog? run.

All initialized

New subscriber added with number = 7379356
New subscriber added with number = 8310888
New subscriber added with number = 411
New subscriber added with number = 911
Subscriber with number = 8310888 has subscribed to
service=cfnr
NEW Transaction Class for subscriber number 8310888
is: tcCallForwarding
Subscriber with number = 8310888 has subscribed to
service = callWaiting
Subscriber with number = 411 has subscribed to service
= callHold
NEW Transaction Class for subscriber number 411 is:
tccallHold
Subscriber with number = 7379356 has subscribed to
service = cfu
NEW Transaction Class for subscriber number 7379356
is: tccfu
Subscriber with number = 411 has subscribed to service
= callWaiting
Subscriber with number = 7379356 has subscribed to
service = cfb
NEW Transaction Class for subscriber number 7379356
is: tccfucfb
data filled

Figure D.1: Invocation of MLog and Initialization of Prototype

D.2 Scenario 1: Basic Service, Call Waiting and Hold Services

This section provides a sample trace showing the basic service invocation, as well as the call waiting and hold supplementary services. The scenario is described as follows:

1. Subscriber 7379356 requests a basic service establishment to Subscriber 411.
2. Subscriber 411 responds to the service request with an alerting message, followed by a connect message.
3. Subscriber 7379356 and Subscriber 411 are now communicating.
4. One of the B-channels on the Subscriber 411 access is allocated in order to create a no B-channel available condition.
5. Subscriber 911 then requests a basic service establishment to Subscriber 411.
6. Since there are no B-channels available on Subscriber 411 access and Subscriber 411 has subscribed to the call waiting service, the call waiting service is invoked.
7. Subscriber 411 then places the association with Subscriber 7379356 on hold in order to connect to the Subscriber 911 association.
8. Subscriber 411 then connects to the Subscriber 911.
9. Subscriber 411 and Subscriber 911 are communicating; Subscriber 7379356 is on hold.
10. One of the B-channels on the Subscriber 411 access is released in order to create a channel available condition.
11. The held association to Subscriber 7379356 is retrieved.
12. The association between Subscriber 411 and Subscriber 911 is released.
13. The association between Subscriber 411 and Subscriber 7379356 is released.

In order to simulate the role of a subscriber, the user enters messages from the MLog tty console in the following form:

TObject-sendMessage(MESSAGE TYPE,PARAMETERS).

TCObject is the Object Identifier of the Transaction Control (TC) object instance responsible for co-ordinating and managing all transactions with the particular subscriber. Therefore, the subscriber with the number 7379356 sends messages to the Transaction Control object with the Object Identifier = s1, subscriber 8310888 sends to the TC object with the Object Identifier = s2, etc. The "sendMessage/2" message is used to provide the application message (i.e. "MessageType" and "Parameters") to the Transaction Control object (see Section 6.6.1). The first "Parameter" specified is always the "transaction identifier" or the *reference* (see Section 6.4.1.3).

Figure D.2 provides a screen dump illustrating the invocation of basic service, call waiting and hold services. As discussed in Section 6.7, the outgoing association, incoming association and association control objects upon processing of an application message will print on the MLog console, the following:

- The application message type received.
- The current state of the object when the message was received.
- The next state of the object as the result of processing the message.

A number of other messages are printed on the MLog console by the MLog Telecommunication Service prototype to assist in following the execution of the various services. For example, messages to be sent to a subscriber are printed on the MLog console (see Section 6.4.1.12).

In Figure D.2, comments have been added on the left handed side for understanding and do not constitute output from the MLog prototype.

basic service request from subscriber 7379356 to subscriber 411, reference = 0	MLog? s1-sendMessage(Setup,[0,411]). number of Associations associated with TC object,#s1, has increased to: 1
outgoing association object with identifier = #116 is created and receives setup message	setup received by object #116 in state: idle New state is: associationProceeding
association control object is created and receives setupIndication message	setupIndication received by association control object in state idle New state is: waitForResponderId
TC object associated with subscriber 411 receives setupRequest message	number of Associations associated with TC object,#s3, has increased to: 1
incoming association object been created by TC s3 and identifier sent to association control object.	objectId received by association control object in state has waitForResponderId New state is: waitForSetupConfirm
incoming association object sends setup message to subscriber, reference = 100	setup received by subscriber number:411 from Object: CLASS = incomingAssociation IDENTIFIER = #118 Parameters included in message are: 100
incoming association object receives setupRequest	setupRequest received by object #118 in state: idle New state is: waitForUserResponse
end of basic service request	yes,msg = s1-sendMessage(Setup,[0,411]) Continue? (y/n) n.

Figure D.2: Basic Service, Call Waiting and Hold Service Traces

subscriber 411 responds
to basic service request
from subscriber 7379356
with alerting

report message containing
alerting indication relayed
by association control
to outgoing association
object

Alerting message is
returned to subscriber
7379356

Report message received
by incoming association
object

subscriber 411 sends
connect message

seize a B-channel
on subscriber 411 access
to create a no-channel
available condition

MLog? s3-sendMessage(alerting,[100,_]).
alerting received by object #118 in
state:waitForUserResponse
New state is:userResponded

report received by association control object in state
waitForSetupConfirm
New state is: waitForSetupConfirm

alerting received by subscriber number:7379356 from
Object: CLASS = outgoingAssociation IDENTIFIER = #116

report received by object #116 in state:
associationProceding
New state is:associationDelivered
yes,msg = s3-sendMessage(alerting,[100,_4314])
Continue? (y/n) n.

MLog? s3-sendMessage(connect,[100,_]).
connect received by object #118 in state:userResponded
New state is:associated

setupConfirm received by association control object in
state waitForSetupConfirm
New state is: associationEstablished

connect received by subscriber number:7379356 from
Object: CLASS = outgoingAssociation IDENTIFIER = #116

setupResponse received by object #116 in
state:associationDelivered
New state is:associated
yes,msg = s3-sendMessage(connect,[100,_5037])
Continue? (y/n) n.

MLog? resource-seizeBchannel(411,junk,X).
yes,msg = resource-seizeBchannel(411,junk,b2)
Continue? (y/n) n.

Figure D.2 (cont.)

basic service request
subscriber 911 to
subscriber 411
reference = 0

MLog? s4-sendMessage(setup,[0,411]).
number of Associations associated with TC object,#s4, has
increased to: 1

ougoing association object
with identifier = #119 is
created and receives
setup message.

setup received by object #119 in state:idle
New state is:associationProceeding

setupIndication
received by association control object in state idle
New state is: waitForResponderId

TC object #s3 associated
with subscriber 411
receives a second setup
request and creates a
new incoming association
object.

number of Associations associated with TC object,#s3, has
increased to: 2

objectID received by association control object in state
waitForResponderId
New state is: waitForSetupConfirm

No B-channel is available
to offer the incoming
setup request, call waiting
is invoked

call waiting invoked

A second setup is sent to
subscriber 411
The reference assigned
to the second service
request = 99

setup received by subscriber number:411 from Object:
CLASS = incomingAssociation
IDENTIFIER = #121
Parameters included in message are: 99

setupRequest received by object #121 in state:idle
New state is:waitForUserResponse
yes,msg = s4-sendMessage(setup,[0,411])
Continue? (y/n) n.

Figure D.2 (cont.)

subscriber 411 places
the association with
subscriber 7379356 on hold

MLog? s3-sendMessage(hold,100).

holdAcknowledge received by subscriber number:411 from
Object: CLASS = holdService IDENTIFIER = #122

yes,msg = s3-sendMessage(hold,100)
Continue? (y/n) n.

a channel is now free
subscriber 411 can now
connect to subscriber 911

MLog? s3-sendMessage(connect,[99,_]).

connect received by object #121 in
state:waitForUserResponse
New state is:associated

setupConfirm received by association control object in
state waitForSetupConfirm
New state is: associationEstablished

connect received by subscriber number:911 from Object:
CLASS = outgoingAssociation IDENTIFIER = #119

setupResponse received by object #119 in
state:associationProceeding
New state is:associated
yes,msg = s3-sendMessage(connect,[99,_8273])
Continue? (y/n) n.

subscriber 411 tries to
retrieve held association
to subscriber 7379356

MLog? s3-sendMessage(retrieve,100).

retrieve rejected, no channel
is available

retrieveReject received by subscriber number:411 from
Object: CLASS = holdService IDENTIFIER = #122
Parameters included in message are: CAUSE= no available
channel
yes,msg = s3-sendMessage(retrieve,100)
Continue? (y/n) n.

release B channel which
previously seized to set up
no channel condition

MLog? resource-releaseBchannel(411,b2).
yes,msg = resource-releaseBchannel(411,b2)
Continue? (y/n) n.

Figure D.2 (cont.)

held association with
subscriber 7379356 is
successfully retrieved

MLog? s3-sendMessage(retrieve,100)

retrieveAcknowledge received by subscriber number:411
from Object: CLASS = holdService IDENTIFIER = #122
yes,msg = s3-sendMessage(retrieve,100)
Continue? (y/n) n.

disconnect request from
subscriber 411 to
subscriber 911

MLog? s3-sendMessage(disconnect,[99,_]).
disconnect received by object #121 in state:associated
New state is:disconnectIndication

releaseIndication received by association control object in
state associationEstablished
New state is: waitForReleaseConfirm

release sent to subscriber
411 in response to
disconnect message

release received by subscriber number:411 from Object:
CLASS = subscriberAssociation IDENTIFIER = #121

outgoing association object
receives release response
from association control
object

releaseResponse received by object #121
in state:disconnectIndication
New state is:releaseRequest

disconnect sent to
subscriber 911

disconnect received by subscriber number:911 from
Object: CLASS = subscriberAssociation IDENTIFIER =
#119

incoming association
object receives release
request from association
control object

releaseRequest received by object #119 in state:associated
New state is:disconnectRequest

association control object
terminates - there is no
longer an association
between subscribers
411 and 911

releaseConfirm received by association control object in
state waitForReleaseConfirm
New state is: terminated

yes,msg = s3-sendMessage(disconnect,[99,_10217])
Continue? (y/n) n.

Figure D.2 (cont.)

incoming association with
reference =99 is terminated

MLog? s3-sendMessage(releaseComplete,[99,_]).
releaseComplete received by object #121 in
state:releaseRequest New state is:terminated
Transaction Complete received by TC object #s3
Number of associations are: 1

yes,msg = s3-sendMessage
(releaseComplete,[99,_11595])
Continue? (y/n) n.

outgoing association
with reference =0 is
terminated

MLog? s4-sendMessage(release,[0,_]).
release received by object #119 in
state:disconnectRequest
New state is:terminated
Transaction Complete received by TC object #s4
Number of associations are: 0

yes,msg = s4-sendMessage(release,[0,_12108])
Continue? (y/n) n.

subscriber 7379356
initiates release of
association to subscriber
411.

MLog? s1-sendMessage(release,[0,_]).
release received by object #116 in state:associated
New state is:releaseIndication

releaseIndication received by association control object in
state associationEstablished
New state is: waitForReleaseConfirm

releaseComplete received by subscriber number:7379356
from Object: CLASS = subscriberAssociation IDENTIFIER =
1 1 6

disconnect received by subscriber number:411 from
Object: CLASS = subscriberAssociation IDENTIFIER =
1 1 8

releaseResponse received by object
#116 in state:releaseIndication New state is:terminated

releaseRequest received by object #118 in state:associated
New state is:disconnectRequest

Figure D.2 (cont.)

Transaction Complete received by TC object#s1
Number of associations are: 0

releaseConfirm received by association control object in
state waitForReleaseConfirm
New state is: terminated

yes,msg = s1-sendMessage(release,[0,_12581])
Continue? (y/n) n.

all associations have now
been released and all
dynamically created objects
have been terminated.

MLog? s3-
sendMessage(releaseComplete,[100,_]).
releaseComplete received by object #118 in
state:disconnectRequest
New state is:terminated
Transaction Complete received by TC object #s3
Number of associations are: 0

yes,msg = s3-
sendMessage(releaseComplete,[100,_14125])
Continue? (y/n) n.

Figure D.2 (cont.)

D.3 Scenrio 2: Call Forwarding No Reply Service

This section provides a sample trace showing the call forwarding no reply (CFNR) service invocation. The scenario is described as follows:

1. Subscriber 831088 activates the CFNR service to 911.
2. Subscriber 7379356 requests a basic service to Subscriber 831088.
3. Subscriber 8310888 responds to the service request with an alerting message.
4. The CFNR timer expires before a connect message is received from subscriber 8310888.
5. The incoming association to Subscriber 8310888 is released.
6. The association is forwarded to Subscriber 911.
7. Subscriber 911 rejects the incoming association request.
8. The forwarded association is released.

As described in Section D.2, the role of the subscriber is simulated by the user entering messages from the MLog tty console in the following form:

TObject-sendMessage(MESSAGE,PARAMETERS).

In addition, to activate the service, a message is sent to the test object which in turn will send the appropriate message and parameters to the Transaction Control object .

Figure D.3 provides a screen dump illustrating the invocation of CFNR service. User input entered at the MLog console is highlighted in bold. Additional comments on the left hand side are provided for clarity and do not constitute output from the prototype.

activate CFNR service
for subscriber 8310888
to subscriber number =911

MLog? test-activateCFNR(911).

cfActivateAcknowledge received by subscriber
number:8310888 from Object: CLASS =
cfServiceActivationControl IDENTIFIER = #123
Parameters included in message are: 911
yes,msg = test-activateCFNR(911)
Continue? (y/n) n.

basic service request from
subscriber 7379356 to
subscriber 8310888.

MLog? s1-sendMessage(Setup,[0,8310888]).

number of Associations associated with TC object,#s1, has
increased to: 1

Setup received by object #124 in state:idle
New state is:associationProceeding

SetupIndication received by association control object in
state idle
New state is: waitForResponderId

number of Associations associated with TC object,#s2, has
increased to: 1

objectId received by association control object in state
waitForResponderId
New state is: waitForSetupConfirm

Setup received by subscriber number:8310888 from
Object: CLASS = incomingAssociation IDENTIFIER = #126
Parameters included in message are: 100

SetupRequest received by object #126 in state:idle
New state is:waitForUserResponse
yes,msg = s1-sendMessage(Setup,[0,8310888])
Continue? (y/n) n.

alerting returned by
subscriber 8310888

MLog? s2-sendMessage(alerting,[100,_]).
timer instance started
CFNR timer started for :2 seconds
alerting received by object #126 in
state:waitForUserResponse
New state is:userResponded

Figure D.3: CFNR Service Traces

	report received by association control object in state waitForSetupConfirm New state is: waitForSetupConfirm
	alerting received by subscriber number:7379356 from Object: CLASS = outgoingAssociation IDENTIFIER = #124
	report received by object #124 in state:associationProceeding New state is:associationDelivered yes,msg = s2-sendMessage(alerting,[100,_17128]) Continue? (y/n) n.
CFNR timer expired	delayed for: 2 seconds timerExpired received by object #126 in state:userResponded New state is:userResponded
	forwardingIndication received by association control object in state waitForSetupConfirm New state is: responderReleaseForwardPending
incoming association to subscriber 8310888 is released	disconnect received by subscriber number:8310888 from Object: CLASS = subscriber Association IDENTIFIER = #126
	releaseRequest received by object #126 in state:userResponded New state is:disconnectRequest
	releaseConfirm received by association control object in state responderReleaseForwardPending New state is: waitForResponderId
association forwarded to subscriber 911	number of Associations associated with TC object,#s4, has increased to: 1
	objectID received by association control object in state waitForResponderId New state is: waitForSetupConfirm
	setup received by subscriber number:911 from Object: CLASS = incomingAssociation IDENTIFIER = #128 Parameters included in message are: 100

Figure D.3 (cont.)

	setupRequest received by object #128 in state: idle New state is: waitForUserResponse
subscriber 911 rejects	MLog? s4- sendMessage(releaseComplete,[100,_]). releaseComplete received by object #128 in state: waitForUserResponse New state is: waitForReleaseResponse
incoming service request forwarded association is released.	releaseIndication received by association control object in state waitForSetupConfirm New state is: waitForReleaseConfirm
	releaseResponse received by object #128 in state: waitForReleaseResponse New state is: terminated
	disconnect received by subscriber number: 7379356 from Object: CLASS = subscriberAssociation IDENTIFIER #124
	Transaction Complete received by TC object #s4 Number of associations are: = 0
	releaseRequest received by object #124 in state: associationDelivered New state is: disconnectRequest
	releaseConfirm received by association control object in state waitForReleaseConfirm New state is: terminated
	yes,msg = s4- sendMessage(releaseComplete,[100,_19804]) Continue? (y/n) n.
outgoing association object (subscriber 7379356) is terminated	MLog? s1- sendMessage(releaseComplete,[0,_]). releaseComplete received by object #124 in state: disconnectRequest New state is: terminated Transaction Complete received by TC object #s1 Number of associations are: 0
	yes,msg = s1- sendMessage(releaseComplete,[0,_21211]) Continue? (y/n) n.

Figure D.3 (cont.)

D.4 Scenrio 3: Call Forwarding Unconditional Service

This section provides a sample trace showing the call forwarding unconditional (CFU) service invocation. The scenario is described as follows:

1. Subscriber 7379356 activates the CFU service to 911.
2. Subscriber 8310888 requests a basic service to Subscriber 7379356.
3. The CFU service is invoked and the association forwarded to subscriber 911.
4. Subscriber 911 rejects the incoming association request.
5. The forwarded association is released.

As described in Section D.2, the role of the subscriber is simulated by the user entering messages from the MLog tty console in the following form:

TCObject-sendMessage(MESSAGETYPE,PARAMETERS).

In addition, to activate the service, a message is sent to the test object which in turn will send the appropriate message and parameters to the Transaction Control object .

Figure D.4 provides a screen dump illustrating the invocation of CFU service. User input entered at the MLog console is highlighted in bold. Additional comments on the left hand side are provided for clarity and do not constitute output from the prototype.

activate CFU for
subscriber 7379356
to subscriber number
911

MLog? test-activateCFU(911).

cfActivateAcknowledge received by subscriber
number:7379356 from Object: CLASS =
cfServiceActivationControl IDENTIFIER = #129
Parameters included in message are: 911

yes,msg = test-activateCFU(911)
Continue? (y/n) n.

basic service request
from subscriber 8310888
to subscriber 7379356

MLog? s2-sendMessage(setup,[0,7379356]).
number of Associations associated with TC object,#s2, has
increased to: 2
setup received by object #130 in state:idle
New state is:associationProceeding

setupIndication
received by association control object in state idle
New state is: waitForResponderId

cfu invoked

association control object
receives reject Indication
with a reason of forward
association

rejectIndication received by association control object in
state waitForResponderId
New state is: waitForResponderId

association forwarded to
subscriber 911

number of Associations associated with TC object,#s4, has
increased to: 1

objectID received by association control object in state
waitForResponderId
New state is: waitForSetupConfirm

setup received by subscriber number:911 from Object:
CLASS = incomingAssociation IDENTIFIER = #132
Parameters included in message are: 99

setupRequest received by object #132 in state:idle
New state is:waitForUserResponse
yes,msg = s2-sendMessage(setup,[0,7379356])
Continue? (y/n) n.

Figure D.4: CFU Service Traces

MLog? s4-sendMessage(releaseComplete,[99,_]).

**releaseComplete received by object #132 in
state:waitForUserResponse
New state is:waitForReleaseResponse**

**releaseIndication received by association control object in
state waitForSetupConfirm
New state is: waitForReleaseConfirm**

**releaseResponse received by object #132 in
state:waitForReleaseResponse
New state is:terminated**

**disconnect received by subscriber number:8310888
from Object: CLASS = subscriberAssociation IDENTIFIER
= #130
Transaction Complete received by TC object #s4
Number of associations are: 0**

**releaseRequest received by object #130 in
state:associationProceeding
New state is:disconnectRequest**

**releaseConfirm received by association control object in
state waitForReleaseConfirm
New state is: terminated**

**yes,msg = s4-
sendMessage(releaseComplete,[99,_24367])
Continue? (y/n) n.**

**MLog? s2-sendMessage(releaseComplete,[0,_]).
releaseComplete received by object #130 in
state:disconnectRequest
New state is:terminated
Transaction Complete received by TC object #s2
Number of associations are: 1**

**yes,msg = s2-
sendMessage(releaseComplete,[0,_25757])
Continue? (y/n) n.**

Figure D.4 (cont.)

request to deactivate
CFU service

MLog? test-deactivateCFU.

cfDeactivateAcknowledge received by subscriber
number:7379356 from Object: CLASS
= cfServiceActivationControl IDENTIFIER = #133

yes,msg = test-deactivateCFU
Continue? (y/n) n.

Figure D.4 (cont.)

D.5 Scenrio 4: Call Forwarding Busy Service

This section provides a sample trace showing the call forwarding busy (CFU) service invocation. The scenario is described as follows:

1. Subscriber 7379356 activates the CFB service to 911.
2. Subscriber 8310888 requests a basic service to Subscriber 7379356.
3. Subscriber 7379356 rejects the incoming association request with a reason of subscriber busy.
4. The CFB service is invoked and the association forwarded to subscriber 911.
5. Subscriber 911 rejects the incoming association request.
6. The forwarded association is released.

As described in Section D.2, the role of the subscriber is simulated by the user entering messages from the MLog tty console in the following form:

TObject-sendMessage(MESSAGE TYPE,PARAMETERS).

In addition, to activate the service, a message is sent to the test object which in turn will send the appropriate message and parameters to the Transaction Control object .

Figure D.5 provides a screen dump illustrating the invocation of CFB service. User input entered at the MLog console is highlighted in bold. Additional comments on the left hand side are provided for clarity and do not constitute output from the prototype.

request to activate CFB
service for subscriber
7379356 to subscriber
911

MLog? test-activateCFB(911).

cfActivateAcknowledge received by subscriber
number:7379356 from Object: CLASS =
cfServiceActivationControl IDENTIFIER = #134
Parameters included in message are: 911

yes,msg = test-activateCFB(911)
Continue? (y/n) n.

basic service request
from subscriber 831088
to subscriber 7379356

MLog? s2-sendMessage(setup,[0,7379356]).
number of Associations associated with TC object,#s2, has
increased to: 2

setup received by object #135 in state:idle
New state is:associationProceeding
setupIndication
received by association control object in state idle
New state is: waitForResponderId

number of Associations associated with TC object,#s1, has
increased to: 1

objectID received by association control object in state
waitForResponderId
New state is: waitForSetupConfirm

setup received by subscriber number:7379356 from
Object: CLASS = incomingAssociation IDENTIFIER = #137
Parameters included in message are: 100

Figure D.5: CFB Service Traces

	setupRequest received by object #137 in state: idle New state is: waitForUserResponse yes, msg = s2-sendMessage(setup,[0,7379356]) Continue? (y/n) n.
subscriber 7379356 rejects incoming service request with a reason of subscriber busy	MLog? s1-sendMessage(releaseComplete,[100,subscriberBusy]). releaseComplete received by object #137 in state: waitForUserResponse New state is: waitForReleaseResponse
releaseIndication is received by association control object with a reason of forward association	releaseIndication received by association control object in state waitforSetupConfirm New state is: waitforResponderId
	releaseResponse received by object #137 in state: waitForReleaseResponse New state is: terminated Transaction Complete received by TC object #s1 Number of associations are: 0
association forwarded to subscriber 911	number of Associations associated with TC object, #s4, has increased to: 1
	objectID received by association control object in state waitforResponderId New state is: waitforSetupConfirm
	setup received by subscriber number: 911 from Object: CLASS = incomingAssociation IDENTIFIER = #138 Parameters included in message are: 98
	setupRequest received by object #138 in state: idle New state is: waitForUserResponse yes, msg = s1- sendMessage(releaseComplete,[100,subscriberBusy]) Continue? (y/n) n.
	MLog? bye.

Figure D.5 (cont.)

END

1 9/0 2/9 1

FIN