SERVICE CREATION AND DEPLOYMENT ON AN INTELLIGENT NETWORK

A thesis submitted as a requirement for the degree of
Master of Engineering in Electronic Engineering.

Michael Collins B. Eng.

School of Electronic Engineering

Dublin City University

August 17th 1999

Supervisor: Dr. T. Curran

ACKNOWLEDGEMENTS

I would like to thank Dr. Tommy Curran for his supervision and support throughout the project. I would also like to thank Olga Ormond, Derek Toibin, Jelena Vasic, Fiona Lodge, Brendan Jennings and Robert Brennan for their assistance.

DECLARATION

I hereby declare that, except where otherwise indicated, this document is entirely my own work and has not been submitted in whole or in part to any other university.

Signed: Michael Collins Date: 17th August 1999

ABSTRACT

Active competition in the telecommunications industry has caused a dramatic shift in focus for public network operators. Service designers need to be able to easily and rapidly create services according to the customer's requirements. This is achievable by using Intelligent Networks (INs). Two primary goals of service development under the Intelligent Network paradigm are rapid service creation using new software technologies and the minimisation of service development costs through switch vendor independence. This thesis examines the development of an IN architecture and the deployment of two call control services on it using the ITU-T Service Independent Building Block (SIB) methodology. The services are deployed on a narrow-band Excel switching platform.

Various aspects of the IN Conceptual Model (INCM) are examined with a particular emphasis on the middle two planes: the Global Functional Plane (GFP) and the Distributed Functional Plane (DFP). Representations of these planes are designed using the ITU-T Specification and Description Language (SDL) [SDL89] and implemented using Telelogic's SDL Development Tool (SDT). SDL provides capabilities to allow logical structuring of the INCM into its constituent entities, the modelling of communication between these entities and the processing within them. The Intelligent Network paradigm was developed with a view to extendibility. Two call control services, *Ringback* and *Group Call Pickup*, are implemented using the SIB methodology. Further services may be created by rearranging the order of execution of the existing SIBs or, if necessary, by adding new SIBs to the architecture.

Given the demand for multimedia applications to run on top of emerging broadband networks it is becoming increasingly more important for network operators to study the enhancement and evolution of their IN service platforms in order to cope with new customer requirements. TINA is the leading architecture for multimedia service control and delivery, which defines an emerging open service platform. Migration from IN to TINA is explored in this thesis by considering two individual paths of migration. The first path involves the replacement of the IN service control and management elements (SCF, SMF, SDF) with appropriate TINA Computational Objects while the switching elements (SSF, CCF) remain IN compliant. As there is no one-to-one mapping of IN functional entities to TINA computational objects, an Adaptation Unit is required to facilitate interaction between the legacy IN entities and the

TINA Computational Objects. The advantage of this step is that it is possible to keep the investment of deployed IN SSPs while taking advantage of TINA service modelling. The second step involves the introduction of TINA into the switch. In this approach the switch and its switching capabilities may be viewed as a TINA object in itself.

Interworking between IN and TINA (as a step towards full migration to TINA) yields a much richer service platform. This service platform facilitates the creation of services that incorporate both IN and TINA features. A TINA service may use pieces of IN functionality and IN services may also use TINA functionality. It is possible to invoke these hybrid services from either a PSTN or a TINA terminal.

Three hybrid IN/TINA services were designed to demonstrate the increased resources available to the service designer using such a platform. A user connected to the IN switch invokes the Freephone Service. This service uses a TINA database to convert the 1-800 number to an extension number. The call is then connected using IN switching functionality. The *Audio Video Conference*, uses TINA computational objects to set up a video stream between participating users while the audio connection is handled by the IN switching functionality. This service is invoked from a TINA terminal. A user connected to the IN switch invokes the Ringback Service but it runs in the TINA domain using both TINA and IN functionality. Therefore, as well as presenting an application of the IN technology, this thesis proposes possible steps towards migration to the TINA architecture.

LIST OF FIGURES

LIST OF TABLES

TABLE OF CONTENTS

Chapter 1 INTRODUCTION

1.1 Overview

The introduction of software controlled digital exchange systems in the 1960s created the basis for public telecommunications services with increasingly complex call control features. By the eighties the growing demand for new services could no longer be satisfied by following the traditional way of directly modifying the control software in the exchange systems. The need for time and cost effective, rapid service deployment and for wide-area network vendor compatibility was satisfied with the introduction of the Intelligent Network (IN) as a new architectural concept [Q.1201]. The IN concept is based on the separation between call control logic and service logic with a well-defined interface between the two.

Intelligent Networks facilitate the development, with considerable ease and rapidity, of services by customers or service designers according to a variety of service requirements. Because of the separation between the call control and the service logic, services can be produced independently of equipment manufacturers.

This thesis describes an Intelligent Network architecture, developed using Specification and Description Language (SDL) [SDL89], which supports the creation of services using ITU-T's Service Independent Building Block (SIB) methodology. The architecture is based on the IN Conceptual Model, which contains four planes: the Service Plane [Q.1202], Global Functional Plane (GFP) [Q.1203], Distributed Functional Plane (DFP) [Q.1204] and the Physical Plane [Q.1205]. Each of these planes is considered but particular attention is paid to the middle two, the DFP and GFP, as the SDL system describes these two planes. The system runs on a UNIX based network environment and communicates with an Excel narrow-band switching platform [Excel] to which phone extensions are attached.

Two typical IN call control services were selected for implementation on the system, using the ITU-T SIB methodology. These services, *Ringback* and *Group Call Pickup*, are constructed from blocks of service functionality, i.e. the SIBs, which reside in the GFP. Users connected to the Excel switching platform deploy the services.

TINA (Telecommunications Information Networking Architecture) is a distributed architecture that caters for broadband multimedia type services. With the increasing demand for multimedia applications on top of broadband networks, interworking between IN and

TINA is an important area for exploration. Two paths for this migration are presented in this thesis and their applicability in relation to the system developed. Interworking makes the IN Service Logic available to objects in a distributed heterogeneous TINA network and vice versa, thus yielding hybrid IN/TINA services.

1.2 Objectives of Thesis

This thesis will document the design and development of an Intelligent Network, in SDL, based on the ITU-T standards. In order to give the reader sufficient understanding to appreciate the decisions made in the design of the system, the Intelligent Network concept itself will be fully explored and the ITU-T standards will be clarified. The various tools and concepts used in the project, which include CORBA, SDL, SDT and IDL, are explained as are the techniques used to bring them together to achieve the project's objectives.

To demonstrate the operation of the IN system two call-control services will designed. These services will be explained as will the SIBs from which they are composed. The execution of these services on the IN will be observed and the simulations run using SDT will be documented, using Message Sequencing Charts (MSCs).

The functionality and capability of the Excel Narrowband switching platform, on which the services are deployed, will be explained, as will the methods of controlling and interacting with the switch.

Also explored are two paths for achieving migration from the Intelligent Network to TINA [INtoTINA]. Migration to TINA makes possible the use of broadband service features in conjunction with the IN architecture. Paths to achieve interworking, as a step towards full migration, are explored as are the hybrid IN/TINA services running on this hybrid platform.

Future research and directions, based on an anticipation of new technologies becoming available, are also considered in the thesis.

1.3 Structure of Thesis

Chapter 2 presents all the information required to fully explain the concept of the Intelligent Network. An overview of the Q.1200 series of ITU-T IN Recommendations [Q.1200] is given covering the areas essential to the design of the IN system. The two call control

services, selected to be implemented on the IN system, are discussed in detail and broken down into their constituent SIBs, which are also discussed in detail subsequently. A state of the art description of the IN and TINA technologies is given in this chapter. An overview of TINA is presented and two existing paths of migration from IN to TINA are also outlined.

A discussion of the tools needed to develop the IN is provided in **Chapter 3**, including the Specification and Description Language (SDL), used to describe the system, and the SDL Development Tool (SDT) used to develop it. The concept of CORBA, which facilitates the communication between distributed software objects, is discussed as is the Interface Definition Language (IDL) used to define the interfaces between these objects. A discussion on how all of these concepts are brought together to realise the IN system is also given.

The SDL description of the middle two planes of the INCM is presented in **Chapter 4**. Each of the entities implemented in SDL is discussed as is the signalling that occurs between them. The Excel switch on which the services are deployed is also discussed as is the method of communication between it and the IN system. Possible interworking scenarios between the IN system and a TINA network are subsequently presented. A description of three hybrid IN/TINA services running on the resultant interworking platform is provided.

Chapter 5 outlines the criteria, set out before the development of the system, in order to evaluate it. A discussion of the success or failure to meet each criterion is also presented.

Chapter 6 concludes the thesis. A summary of the work done in the project is given and possible directions for future work are discussed.

The **Bibliography** is followed by the glossary of terms and acronyms in **Appendix A**. The vendor specific API message set for communication with the Excel switch is given in **Appendix B**. **Appendix C** provides some Abstract Data Type functions and the Environment Functions used by the IN system to communicate with the switch. In **Appendix D** the SDL signals in the IN system are listed and explained. Message Sequencing Charts showing the execution of the pure IN and hybrid IN/TINA services are provided in **Appendix E**.

Chapter 2 THE INTELLIGENT NETWORK

2.1 Introduction

In traditional telephony systems the services offered by network operators consisted principally of basic call connectivity. However, service technology has recently become a great deal more advanced and increasingly complex services are currently available. The structure of the traditional Public Switched Telephone Network (PSTN) is not very suitable for the creation and deployment of such services. Services created on the PSTN are part of the switching system and this vendor dependence ties the system to the network. The main problem with this scenario is that the services are localised. In other words if the software for a service is loaded at a network node only users directly attached to that node may use that service. If a service is to be altered or upgraded the code must be changed at every node offering the service. This makes the provisioning and maintenance of a service both very difficult and very slow. It is also very wasteful of resources as the same service software is replicated at a wide range of locations.

As the role of services in networks increased in importance the necessity arose for a network architecture which combated the above problems and allowed network operators and service providers to design, implement and maintain services with as little effort as possible. The Intelligent Network (IN) was developed in order to meet these demands by reducing the length of the service design and development phase. This is achieved by providing a standard development environment with a set of reusable function blocks and tools to facilitate the rapid design of a service. IN's much shorter deployment and provisioning phases are achieved through centralising all service execution software so that the code for a new service needs to be installed at only one location from which it is available to all customers on the network. This centralisation simplifies the task of upgrading and maintaining services. Independence from switch hardware and software vendors is also provided.

2.2 IN Development

2.2.1 Basic and Intelligent Call Processing

Before examining the details of the Intelligent Network it is important to review the way in which a standard non-IN call is handled:

The caller dials a telephone number and the switching system uses its call processing software to decide how to handle the call. It either rejects the call request or attempts to set up a circuit connection to the called customer.

In an Intelligent Network the call is handled as follows:

The caller dials a telephone number and the switching system uses its call processing software to decide whether to handle the call itself or to request help from external service logic. If help is required, the switching system sends a message to the external service logic, which decides how the call is to be handled. One or more call control instructions are sent to the switching system to tell it how to handle the call. The switching system then either rejects the call request or attempts to set up a circuit connection to the called customer based on these external service logic instructions.

This ability to access external service logic by a switching system is the basic principle behind Intelligent Network call processing.

2.2.2 Intelligent Network Objectives

The objective of IN is to allow the inclusion of additional capabilities to facilitate the provisioning of services, independent of the service/network implementation in a multi-vendor environment [Q.1201]. The ITU-T objectives for this new architectural concept can be summarised as follows:

- IN should be applicable to all telecommunication networks - PSTNs, ISDNs, mobile networks, etc.
- IN should enable service providers to define their own services, independent of service specific developments by equipment suppliers.
- IN should enable network operators to allocate functionality and resources within their networks and efficiently manage their networks, independent of network-specific developments by equipment suppliers.
- IN should be introduced starting from the existing networks and the current ITU-T recommendations.
- IN should evolve to reflect implementation experiences, new technological opportunities and market evolution.

Driven by the need for standardisation and evolution, ITU-T embarked on a program of work that addresses global international IN standards and a framework for the standardised evolution of IN. A phased standardisation process towards the target IN architecture was adopted, defining a set of Capability Sets for each phase. Each Capability Set (CS) is intended to address requirements for one or more of the following: service creation, service management, service interaction, network management, service processing, and network interworking. This phased standardisation approach takes into account the fact that the specification and deployment of networks that meet all the objectives of the IN target architecture will take many years. In order to allow smooth evolution towards the target, the IN set of Recommendations shall allow backward compatibility of each evolutionary phase and open-endedness towards long term views. The goal of the ITU-T is to define a new architectural concept that meets the needs of telecommunication service providers.

2.2.3 Intelligent Network Service Processing Model

The main three elements of this model are:

- The Basic Call Processes (BCPs).
- The Detection Points (DPs) which allow BCPs to interact with IN service logic.
- IN service logic that can be programmed to implement new supplementary services.

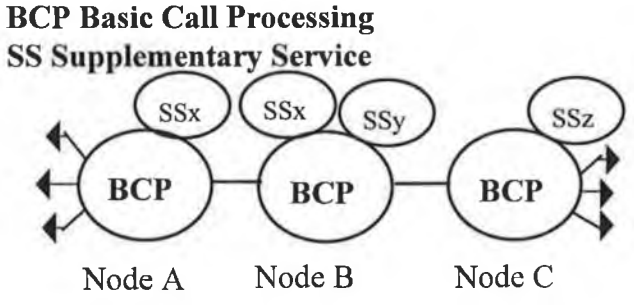


Figure 2.1 Typical non-IN Service Processing Model

The Basic Call Process should be available all over the network and is designed to support, with optimal performance, services that do not require special features. In order to achieve flexibility in service processing the basic call process needs to be modularised into service-independent sub-processes such that these can be executed autonomously (without interference from the outside during execution).

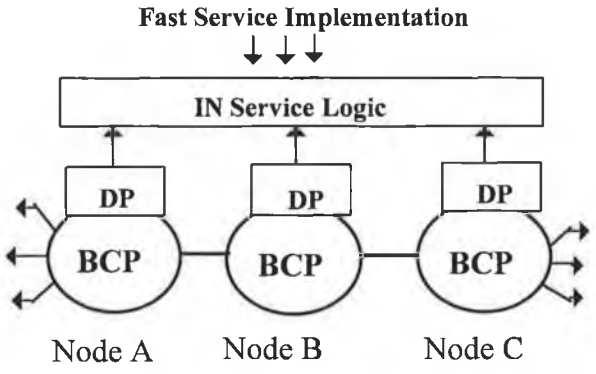


Figure 2.2 IN Service Processing Model

The Detection Points (DPs) are to be added to the basic call process forming the links between the individual basic call sub-processes and the service logic. During a call the basic call process should continuously check for the occurrence of conditions on which an interaction session with IN service logic should be started.

IN service logic uses a programmable software environment that needs to be developed to allow fast implementation of new supplementary services. New supplementary services can be created by means of linking blocks of functionality together in the form of service logic. The IN service logic is able to interact with the basic call process. Thus, by changing logic at the service control point and modifying network data, a new service that uses existing network capabilities can readily be implemented.

2.3 Capability Sets

The International Telecommunications Union (ITU) have a phased approach to the development of the IN standards. Each version of the standards, known as a Capability Set (CS), is a superset of its predecessor. At the time of writing, the current version of the standards is ITU Capability Set 1 (CS-1) [Q.1211] which was released in 1993. The phased approach of developing Capability Sets is illustrated in Figure 2.3.

It is necessary to explain the functionality of the IN, as defined in CS-1, as the operation of the IN system developed in this project essentially represents this. CS-1 capabilities are intended to support services and service features that fall into the category of single ended, single point of control services, referred to as Type A, while all other services are placed in a category called Type B. A single-ended service feature applies to only one party in a call and

is orthogonal (independent) to both the service and topology levels to any other parties that may be participating in the call. Orthogonality allows another instance of the same or a different single-ended service feature to apply to another party in the same call as long as the service feature instances do not have feature interaction problems with each other. Single point of control describes a control relationship where the same aspects of a call are influenced by only one Service Control Function at any point in time [Q.1211].

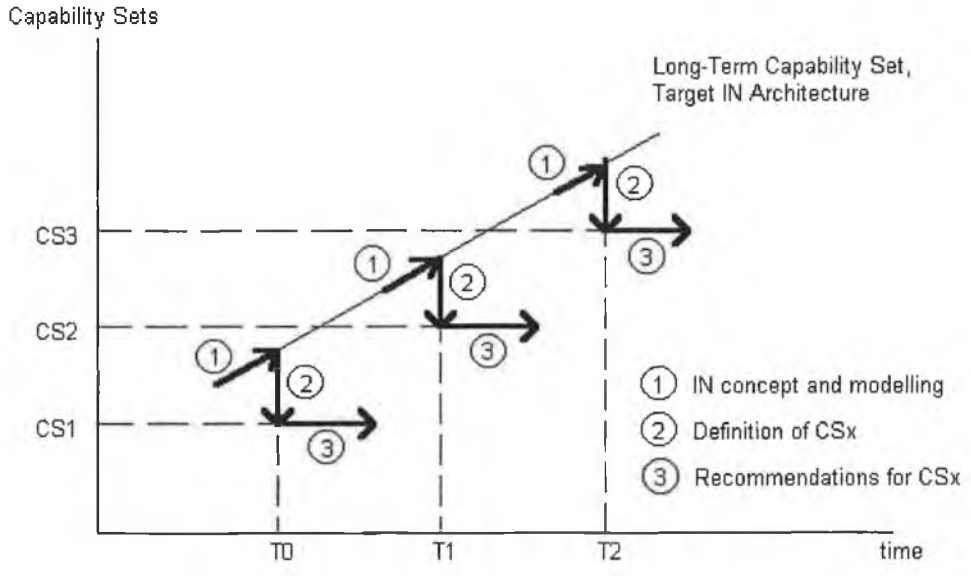


Figure 2.3 Sequencing of Capability Sets versus Time

There is a limited set of services targeted by IN-CS1. These are listed in Table 2.1. Some of these services (indicated by a *) might be only partially supported in CS-1 because they require additional Type B capabilities. Parts of these services are considered in CS-1 as long as these parts belong to Type A and do not impose capabilities additional to those required for other services in the list.

ABD Abbreviated Dialling	FPH Freephone
ACC Account Card Calling	MCI Malicious Call Identification
AAB Automatic Alternate Billing	MAS Mass Calling
CD Call Distribution	OCS Originating Call Screening
CF Call Forwarding	PRM Premium Rate
CRD Call Re-routing Distribution	SEC Security Screening
CCBS Call Completion to Busy	SCF Selective Call Forwarding on

Subscriber (*)		Busy/No Answer	
CON	Conference Calling (*)	SPL	Split Charging
CCC	Credit Card Calling	VOT	Televoting
DCR	Destination Call Routing	TCS	Terminating Call Screening
FMD	Follow Me Diversion	UAN	Universal Access Number
VPN	Virtual Private Network	UDR	User-Defined Routing
UPT	Universal Personal Telecommunications		

Table 2.1 IN Services targeted by CS-1

Intensive research has gone on since the release of CS-1 in the definition and development of Capability Set 2 (CS-2), an expansion of existing CS-1 concepts. CS-2 will provide definitions and classifications to overcome many of the limitations of CS-1, while addressing all the services and SIBs of CS-1 plus additions.

2.3.1 ETSI Standards

The European Telecommunications Standards Institute (ETSI) is a non-profit making organisation whose mission is to determine and produce the telecommunications standards that will be used for decades to come. It is an open forum, representing administrations, network operators, manufacturers, service providers, and users.

It is the ETSI members that fix the standards work programme in function of market needs. Accordingly, ETSI produces voluntary standards - some of these may go on to be adopted by the EC as the technical base for Directives or Regulations - but the fact that the voluntary standards are requested by those who subsequently implement them, means that the standards remain practical rather than abstract.

ETSI promotes the world-wide standardisation process whenever possible. Its work programme is based on, and co-ordinated with, the activities of international standardisation bodies, mainly the ITU-T and the ITU-R.

When an ITU-T Capability Set is released it is analysed by ETSI and improved upon to make it more practically applicable. In drafting new versions of the Capability Sets the ITU-T body take these refinements and improvements into consideration.

2.4 The IN Conceptual Model

The IN Conceptual Model (INCM) should not be considered in itself an architecture. It is a framework for the design and description of the IN architecture. Various models and concepts will be used in the standardisation of IN. The INCM is intended to represent an integrated, formal framework within which these concepts are identified, characterised and related. To achieve this, the INCM consists of four planes where each plane represents a different abstract view of the capabilities provided by an IN-structured network. These views address service aspects, global functionality, distributed functionality and physical aspects of IN [Q.1201] [Q.1203].

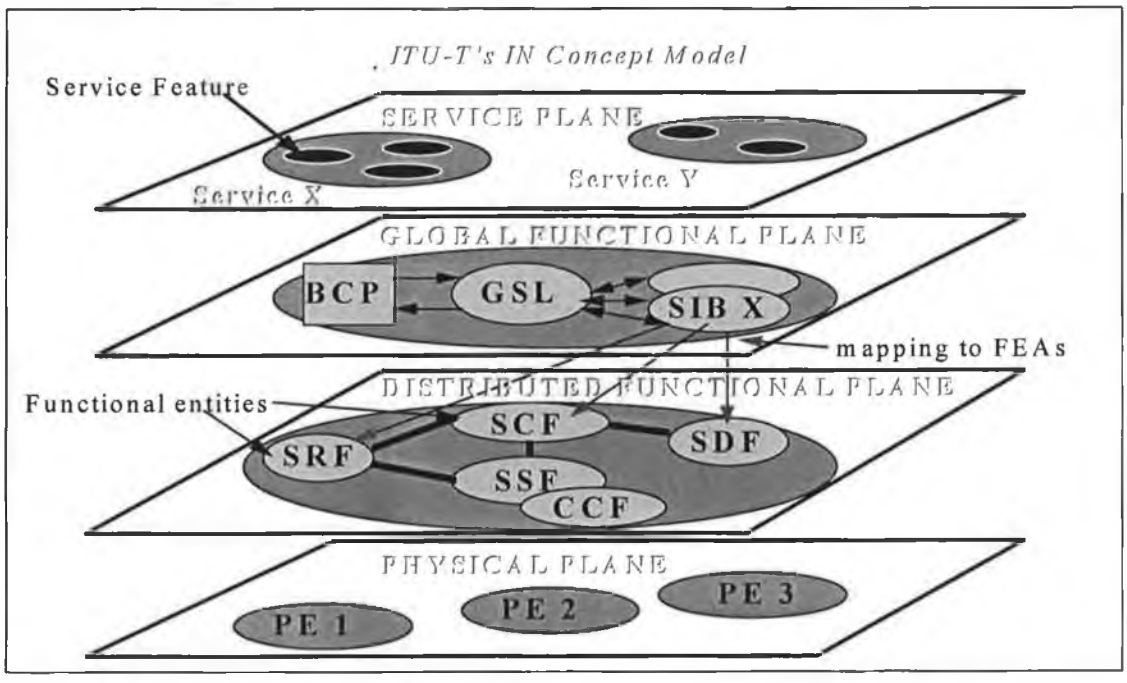


Figure 2.4 Intelligent Network Conceptual Model

BCP Basic Call Process	PE Physical Entity
GSL Global Service Logic	SIB Service Independent Building Block
CCF Call Control Function	SRF Specialised Resource Function
SSF Service Switching Function	SDF Service Data Function

Table 2.2 INCM Acronyms

2.4.1 Service Plane

The Service Plane [Q.1202] illustrates that IN-supported services can be described to the end user or subscriber by means of a set of generic blocks called Service Features. A service is a stand-alone commercial offering, characterised by one or more Service Features and can be optionally enhanced by other Service Features. The service plane represents an exclusively service-oriented view. This view contains no information whatsoever regarding the implementation of the services in the network. All that is perceived is the network's service-related behaviour as seen, for example, by a service user.

2.4.2 Global Functional Plane

The Global Functional Plane (GFP) [Q.1203] models network functionality from a global or network-wide point of view. As such, the IN structured network is said to be viewed as a single entity in the GFP. In this plane, Services and Services Features are redefined in terms of the broad network functions required to support them. These functions are neither Service nor Service Feature specific and are referred to as Service Independent Building Blocks (SIBs).

Contained in the Global Functional Plane are:

- **SIBs** which are standard reusable network-wide capabilities used to realise Services and Service Features
- **Basic Call Process** (BCP) is a SIB which identifies the normal call process from which IN services are launched, including Points of Initiation (POI) and Points of Return (POR) which provide the interface to the Global Service Logic (GSL).
- **Global Service Logic** (GSL) which describes how SIBs are chained together to describe Service Features. The GSL also describes interactions between the BCP and the SIB chains.

By definition, SIBs, including the BCP, are service independent and cannot contain knowledge of subsequent SIBs. Therefore, GSL is the only element in the GFP which is specifically service dependent.

In the GFP basic non-IN calls are processed within the BCP. When an IN supported service is to be invoked its GSL is launched at a Point of Initiation (POI) by a triggering mechanism

from the BCP. In order to chain SIBs together, knowledge of the connection pattern, decision options, and data required by SIBs must be available. The GSL describes SIB chaining, potential branching, and where branches rejoin. At the end of the chain of SIBs, the GSL also describes the returning point to the BCP by indicating the specific Point of Return (POR). For a given service at least one POI is required, however, depending upon the logic required to support the service, multiple PORs may be defined. The use of SIBs is discussed in greater detail in Section 2.6.

2.4.3 Distributed Functional Plane

The Distributed Functional Plane (DFP) [Q.1204] describes the functional architecture of an IN-structured network in terms of units of network functionality, referred to as *Functional Entities* (FEs) and the information that flows between functional entities, referred to as *relationships*. The DFP architecture is vendor/implementation independent, i.e. the functional entities and relationships are described independently of how the functionality is physically implemented in the network. The definition of a Functional Entity (FE) is as follows:

A *Functional Entity* is a unique group of functions in a single location and a subset of the total set of functions required to provide a service. One or more functional entities can be located in the same physical entity [Q.1204].

FEs contain subdivisions, known as *Functional Entity Actions* (FEAs). SIBs on the GFP are realised on the DFP by a sequence of Functional Entity Actions and resulting information flows. The DFP is illustrated in Figure 2.5.

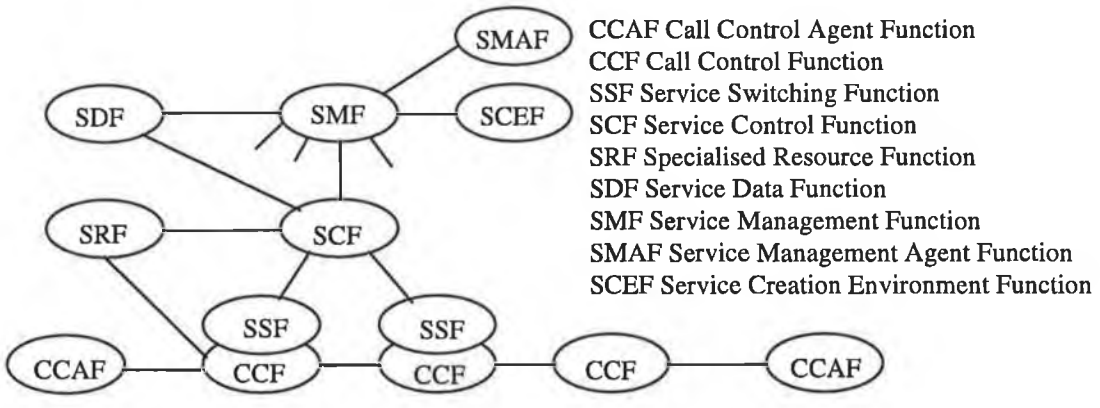


Figure 2.5 Functional Entities and Relations in the DFP

There follows a brief description of the functional entities in the DFP:

The **Call Control Agent Function** (CCAF) provides access for users. It is the interface between user and network call control functions. It receives indications relating to the call or service from the CCF and relays them to the user as required.

The **Call Control Function** (CCF) provides call/connection processing and control. It also provides the capability to associate and relate CCAF functional entities that are involved in a particular call and/or connection instance. It also provides trigger mechanisms to access IN functionality (e.g. passes events to the SSF).

The **Service Switching Function** (SSF) interacts between the CCF and a SCF. It modifies call/connection processing functions (in the CCF) as required to process requests for IN provided service usage under the control of the SCF.

The **Service Control Function** (SCF) commands call control functions in the processing of IN provided and/or custom service requests. The SCF may interact with other functional entities to access additional logic or to obtain information (service or user data) required to process a call/service instance. It contains the logic and processing capability required to handle IN provided service attempts. It interfaces and interacts with other SCFs if necessary.

The **Service Data Function** (SDF) contains customer and network data for real time access by the SCF in the execution of an IN provided service. It interfaces and interacts with other SDFs if necessary.

The **Specialised Resource Function** (SRF) provides the specialised resources required for the execution of IN provided services (e.g. digit receivers, announcements, conference bridges, etc.). It may contain logic and processing capability to receive/send and convert information received from users. It may also contain functionality similar to the CCF to manage bearer connections to the specialised resources.

The **Service Creation Environment** allows services provided in IN to be defined, developed, tested and input to the SMF. The output of this function would include service logic, service management logic, service data template and service trigger information.

The **Service Management Agent Function** (SMAF) provides an interface between service managers and the SMF. It allows service managers to manage their services (through access to the SMF).

The **Service Management Function** (SMF) allows deployment and provision of IN provided services and allows the support of ongoing operation. Particularly, for a given service it allows the co-ordination of different SCF and SDF instances, e.g. billing and statistic information are received from the SCFs and made available to authorised service managers through the SMAF; modifications in service data are distributed in SDFs, and it keeps track of the reference service data values. The SMF manages, updates and/or administers service related information in SRF, SSF and CCF.

2.4.4 Physical Plane

The Physical Plane [Q.1205] of the IN Conceptual Model identifies the different Physical Entities (PEs) and the interfaces between them.

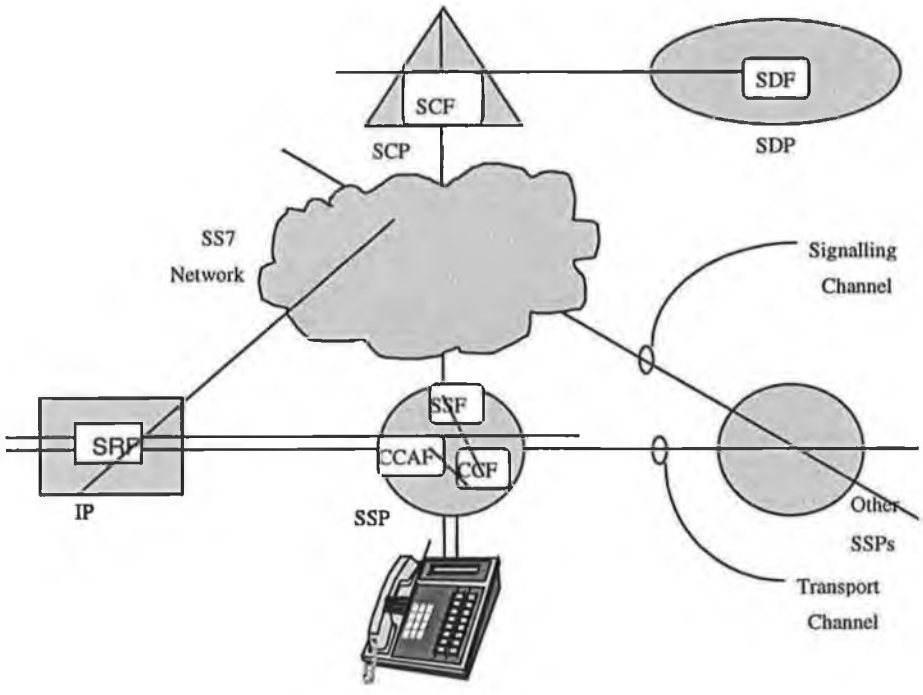


Figure 2.6 Physical Architecture of the Intelligent Network

The IN consists of the following PEs:

- The **Service Switching Point** (SSP) - user access to service functionality is provided through the SSP, which handles call processing, detects service requests and provides connectivity to the SCP and other SSPs in the network. The SSP contains three discrete functions - the CCAF, the CCF and the SSF;

- The **Service Control Point** (SCP) - the Service Control Function (SCF) resides here along with the Global Service Logic (GSL);
- The **Service Data Point** (SDP) - this houses the SDF and is connected directly to the SCP. It contains all network data relevant to the execution of services;
- The **Intelligent Peripheral** (IP) - the SSP maintains a number of channels between itself and the IP, which contains the Service Resource Function (SRF). Interactions occur between the SRF and users when the SSP opens a channel between them. The IP receives instructions relating to announcements and digit collection directly from the SCF and, when necessary, returns any acquired information.

2.5 Particulars of IN CS-1 Service Control

2.5.1 Basic Call Manager

The Basic Call Manager (BCM) is now described. The particular subjects of the BCM discussed include the Basic Call State Model (BCSM) and connection events which can lead to the invocation of IN service logic instances or be reported to already active IN service logic instances.

The BCSM is a high-level finite state machine description of the CCF activities required to establish and maintain communication paths for users. Only the aspects that are reflected upward to the IN service Switching Manager (SM) and Feature Interaction Manager (FIM)/Call Manager (CM) are visible to IN service logic instances and will be the subject of standardisation.

The BCSM identifies points in a basic call (PIC: Point In Call) and connection processing when IN service logic instances are permitted to interact with basic call and connection control capabilities. PICs identify CCF activities to complete one or more call/connection activities of interest to IN service logic instances. Detection Points (DP) indicate points in basic call and connection processing at which transfer of control can occur. Transitions indicate the normal flow of basic call/connection processing from one Point in Call (PIC) to another. Events cause transitions into and out of PICs.

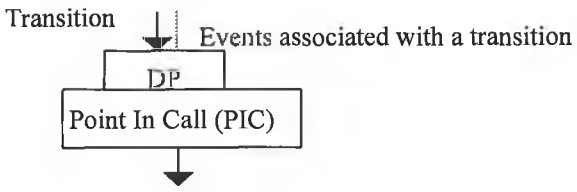


Figure 2.7 Key components of the BCSM

The BCSM reflects the functional separation between the originating and terminating portions of calls. Each is managed by a functionally separate BCM in the SSF/CCF. For IN-CS1 the following BCSM is defined;

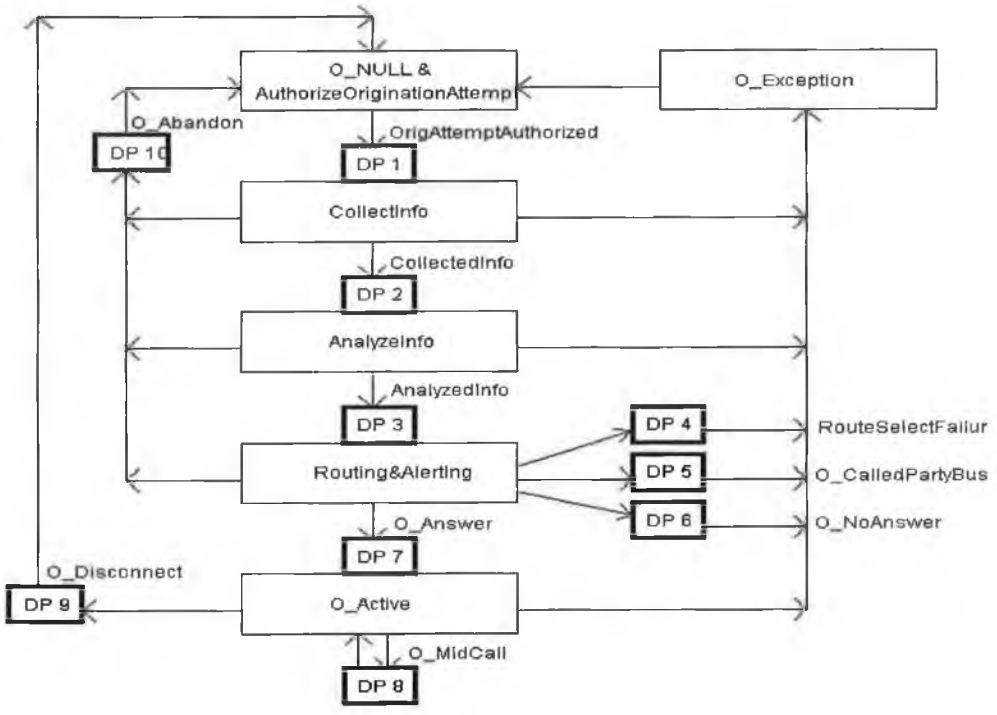


Figure 2.8 Originating side BCSM for CS-1

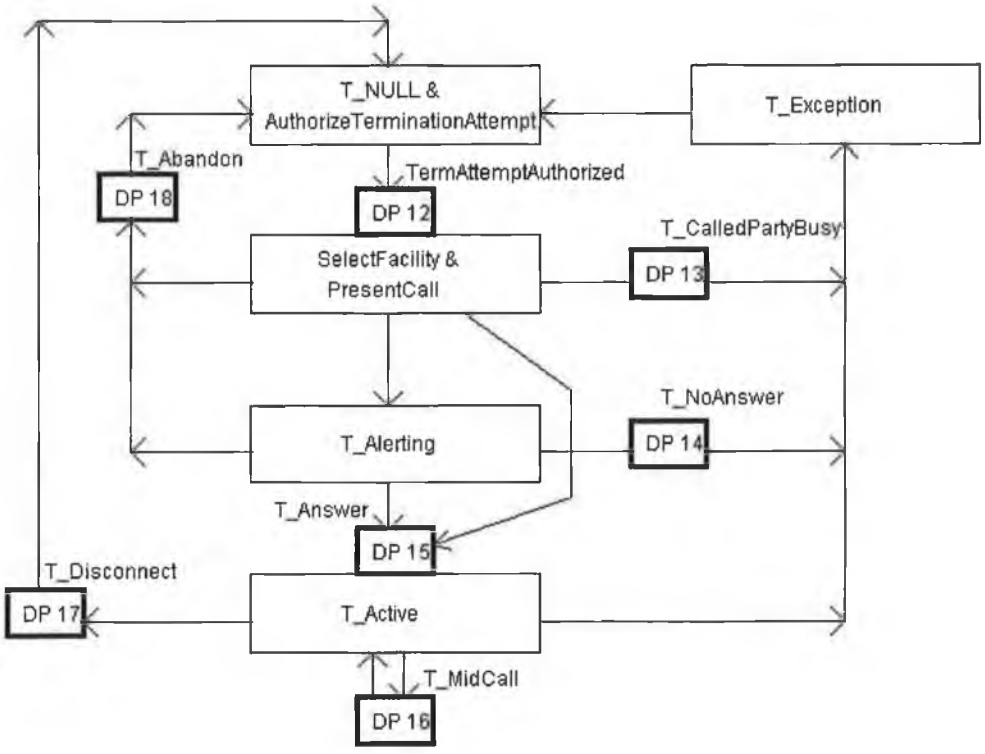


Figure 2.9 Terminating side BCSM for CS-1

1. Initiate a T_BCSM when the authority to place a call attempt has been verified in the Routing&Alerting PIC of the O_BCSM and the originating Basic Call Manager has sent the call attempt to the terminating BCM for further processing.
2. An indication is sent from the T_BCSM to the O_BCSM that the Called Party is busy (causes Routing&Alerting PIC to DP5 transition in O_BCSM).
3. An indication is sent from the T_BCSM to the O_BCSM that the Called Party is being alerted (causes ring indication to be sent to the Calling Party in Routing&Alerting PIC of the O_BCSM).
4. An indication is sent from the T_BCSM to the O_BCSM that the Called Party has not answered within a specified time period (causes Routing&Alerting PIC to DP6 transition in O_BCSM).
5. An indication is sent from the T_BCSM to the O_BCSM that the Called Party has accepted and answered the call attempt (causes Routing&Alerting PIC to DP7 transition in O_BCSM).

6. An indication is sent from the O_BCSM to the T_BCSM that the Calling Party has disconnected (causes T_Active PIC to DP17 transition in T_BCSM).
7. An indication is sent from the T_BCSM to the O_BCSM that the Called Party has disconnected (causes O_Active PIC to DP9 transition in O_BCSM).
8. An indication is sent from the O_BCSM to the T_BCSM that the Calling Party has abandoned (causes SelectFacility & PresentCall or T_Alerting PIC to DP18 transition in T_BCSM)

Note: Indications 6 and 7 are mutually exclusive [Q.1214].

2.5.2 Detection Points

Certain basic call and connection events may be visible to IN service logic instances. Detection Points (DPs) are the points in call processing at which these events are detected. A DP can be armed in order to notify an IN service logic instance that the DP was encountered, and potentially to allow the IN service logic instance to influence subsequent call processing. If a DP is not armed, the CCF/SSF continues call processing without SCF involvement. DPs are characterised by the following four attributes:

1. **Arming Mechanism:** the mechanism by which the DP is armed. A DP may be statically or dynamically armed. A DP is statically armed through SMF service feature provisioning. A statically armed DP remains armed until explicitly disarmed by SMF. A DP is dynamically armed by the SCF within the context of a call-associated IN service control relationship. A dynamically armed DP remains armed until detected, or until the end of the relationship.
2. **Criteria:** in addition to the condition that a DP be armed, conditions that must be met in order to notify the SCF that the DP was encountered. DP Criteria can be assigned to a DP from the viewpoint of range of effectiveness (individual line/trunk based criteria, group based criteria, office based criteria).
3. **Relationship:** given that an armed DP was encountered and DP criteria are met, the SSF may provide an information flow via a relationship:
   - If this relationship is between the SSF/CCF and the SCF for the purpose of call/service logic processing, it is considered to be an IN service control relationship. This relationship may be of two types: a control relationship if the SCF is able to

influence call processing via the relationship; or a monitor relationship, if the SCF is not able to influence call processing. With respect to an IN service control relationship, the information flow provided by the SSF to the SCF on encountering a DP may initiate a control relationship, may be within the context of an existing control relationship, or may be within the context of an existing monitor relationship.

- If this relationship is between SSF/CCF and the SCF or SMF for management purposes, it is considered to be a service management control relationship.

4. **Call Processing Suspension:** given that an armed DP was encountered and DP criteria are met for an IN service control relationship, the SSF may suspend call processing to allow the SCF to influence subsequent call processing. When call processing is suspended, the SSF sends an information flow to the SCF requesting instruction, and waits for a response. When call processing is not suspended, the SSF sends an information flow notifying the SCF that DP was encountered, and does not expect a response. The call processing suspension attribute is set by the same mechanism that arms the DP.

Based on these attributes, four types of DPs are identified for CS1:

- Trigger Detection Point - Request (TDP- R)
- Trigger Detection Point - Notification (TDP- N)
- Event Detection Point - Request (EDP- R)
- Event Detection Point - Notification (EDP- N)

Since a DP may be armed as a TDP and/or EDP for the same call, the BCM should apply a set of rules during DP criteria processing to ensure single point of control:

- DP-N criteria are processed before DP-R criteria. This ensures that notifications are provided to SCFs and are not bypassed as a result of call handling instructions that may occur as a result of a DP-R.
- TDP-R criteria may not be processed if there is any existing control relationship for this portion of the call. This ensures not to violate the single point of control restriction for CS-1.

- The same DP may be armed multiple times as a TDP-R, with the DP criteria placed in precedence order through administrative procedures. Subsequent TDP-R criteria can only be processed if there is no existing control relationship caused by previous DP processing.
- TDP-N criteria may be processed whether or not there is an existing control relationship for the same portion of the call since TDP-N does not open a control relationship.
- A control relationship persists as long as there is one or more than one EDP-R armed for this portion of the call. A control relationship terminates, if there are no more EDPs armed or the call clears. During a control relationship, EDPs may be dynamically disarmed by the SCF, or are disarmed by the SSF as they are encountered and reported to the SCF, or when the call clears.
- A control relationship changes to a monitor relationship if there are no more EDP- Rs armed and one, or more than one EDP-Ns armed. A monitor relationship terminates if there are no more EDP-Ns armed or the call clears. During a monitor relationship, EDP-Ns are disarmed by the SSF and reported to the SCF as they are encountered or when the call clears [Q.1214].

2.6 The use of SIBs in the Global Functional Plane

The network is viewed from the GFP as a single structure. It describes units of service functionality, referred to as service independent building blocks (SIBs), which are independent of how the functionality is distributed in the network. The Global Service Logic (GSL) can describe a service as a chain of SIBs to be executed. An instance of a string of SIBs describing a service is a Service Logic Program Instance (SLPI). Each SIB is mapped onto one or more Functional Entity Actions contained within the Functional Entities in the Distributed Functional Plane (DFP).

Normal, or non-IN, supported services are processed within the BCP. When an IN supported service is to be invoked its GSL is launched at the point of initiation (POI) by a triggering mechanism from the BCP. The GSL also describes the returning point to the BCP by indicating a specific point of return (POR).

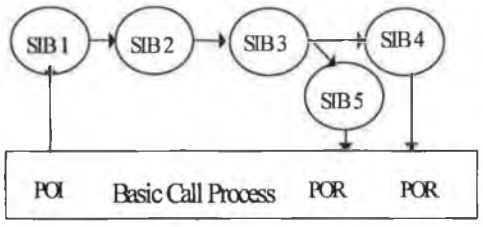


Figure 2.10 Using SIBs in the GFP

2.6.1 Definition of a SIB

A SIB is a standard reusable network-wide capability residing in the global functional plane which is used to create service features. SIBs are of a global nature and their detailed realisation is not considered at this level but can be found in the DFP and the physical plane. SIBs are reusable and can be chained together in various combinations to realise services in the service plane. SIBs are defined to be independent of the specific service and technology for which, or on which, they will be realised [Q.1203].

2.6.2 Characteristics of a SIB

- SIBs are defined completely independently from considerations of any specific distributed functional and physical plane architectures [Q.1203].
- Interactions among FEs in the DFP are invisible to SIBs in the GFP.
- Individual SIBs must be defined using a standard methodology to allow multi-vendor IN products to identically support them and to allow service designers to have a common understanding of the SIB.
- SIBs are monolithic building blocks that the service designer uses to develop new services. All service features are described by a single SIB or a chain of SIBs.
- SIBs are realised in the DFP by Function Entity Actions (FEAs) which may reside in one or more Functional Entities (FEs).
- A SIB has one logical starting point and one or more logical end points. Data required by each SIB is defined by SIB support data parameters and call instance parameters.
- SIBs are reusable. They are used without modifications for other services.

2.6.3 Data Parameters For SIBs

SIBs are service independent and have no knowledge about previous or subsequent SIBs used to describe the service. In order to describe a service using these generic SIBs service dependent elements are needed. Service dependence can be described by using data parameters, which enable a SIB to be tailored to perform the required functionality. Data parameters are specified independently for each SIB and are made available to the global service logic. Two types of parameters are required for each SIB; static parameters named Service Support Data (SSD) and dynamic parameters known as Call Instance Data (CID).

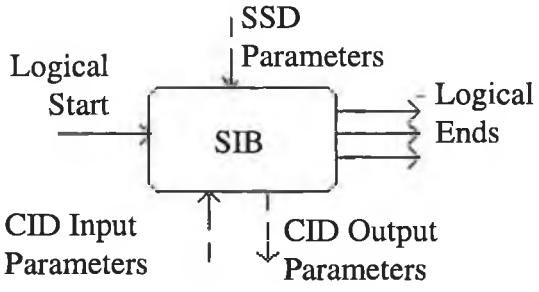


Figure 2.11 Graphic Representation of SIB

Call Instance Data (CID)

CID defines dynamic parameters whose value will change with each call instance. They are used to specify subscriber specific details such as calling or called line information. This data can be made available from the BCP SIB, generated by a SIB, or entered by the subscriber. Associated with each CID value is a logical name, referred to as the CID Field Pointer (CIDFP). If a SIB requires a CID to perform its function there will be an associated CIDFP assigned through the SSD.

Service Support Data (SSD)

SSD defines the data parameters required by a SIB which are specific to the service feature description. The GSL specifies the SSD values for a SIB. The SSD consists of:

- **Fixed parameters**: Data parameters whose values are fixed for all call instances. For instance, the "File Indicator" SSD for the Translate SIB needs to be specified uniquely for each occurrence of that SIB in a given service. The "File Indicator" value is then said to be fixed, as the value is determined by the service description, not by the call instance.

- **Field pointers**: Identify which CID is required by the SIB. "CIDFP-xxxx" signifies them, where "xxxx" names the data required. For instance, "CIDFP-Info" for the Translate SIB will specify which CID element is to be translated. If more than one CID is required by a SIB to perform its function, then the SSD data parameters will contain multiple fields.

2.6.4 Basic Call Process (BCP)

The BCP is responsible for providing basic call connectivity between parties in the network. It can be viewed as a specialised SIB which provides basic call capabilities, including:

- connection of calls
- disconnection of calls
- retaining CID for further processing

IN supported services are represented through the use of chains of SIBs connected to the BCP SIB (see Figure 2.10). The interface points between the BCP SIB and the chains of SIBs are described as points of initiation and points of return:

A *Point Of Initiation* (POI) is the BCP functional launching point for the SIB chains.

A *Point Of Return* (POR) identifies the point in the BCP where the SIB chains terminate. The need for specific POI/POR functionality is that the same chain of SIBs may represent a different service if launched from a different point in the BCP.

2.6.5 The Service Logic

The Global Service Logic (GSL) contains the logic that is used to run IN services. An instance of a services logic is referred to as a Service Logic Programming Instance (SLPI). A SLPI defines the order in which SIBs are chained together to accomplish services. The SIB chain begins with a POI and ends with a POR to the BCP. The SLPI also provides the CID and SSD parameters for the SIBs.

2.6.6 CS-1 SIBs

SIBs identified in Capability Set 1 include the following:

SIB	Description
Algorithm	Applies a mathematical algorithm to data to produce a data result
Charge	Determines special charging treatment for the call, where special refers to any charging in addition to that normally performed by the basic call process.
Compare	Performs a comparison of an identifier against a specified reference value. Three results are possible : identifier is GREATER than the value, identifier is LESS than the value, identifier is EQUAL to the value
Distribution	Distribute calls to different logical ends of the SIB based on user specified parameters
Limit	Limit the number of calls related to IN provided service features
Log Call Information	Log detailed information for each call to a file. The collected information may be used by management services and not by call related services
Queue	Provide sequencing of IN calls to be completed to a called party
Screen	Perform a comparison of an identifier against a list to determine whether the identifier has been found in the list
Service Data Management	Enables end user specific data to be replaced, retrieved, incremented, or decremented
Status Notification	Provide the capability of inquiring about the status and/or status changes of network resources
Translate	Determine output information from input information.
User Interaction	Allows information to be exchanged between the network and a called party, where a call party can be either a calling or called party
Verify	Provide confirmation that information received is syntactically consistent with the expected form of such information

Table 2.3 CS-1 SIBs

2.7 The Services

Two call control IN services; *Ringback* and *Group Call Pickup,* were chosen to be implemented on the IN Architecture using the ITU-T SIB methodology. The services are now described in terms of how they work and the SIBs necessary to implement them.

2.7.1 Ringback Service

This service is invoked by a caller (Party A) who, having phoned a busy extension (Party B), decides to request a phone call from that extension when they become available. When the busy Party B becomes available Party A is rung. When Party A goes off hook Party B is rung. When Party B answers a call is set up between them. The special Ringback service digit-sting used to invoke the service is '69'. The SIB chain designed to depict this service is shown in Figure 2.12.

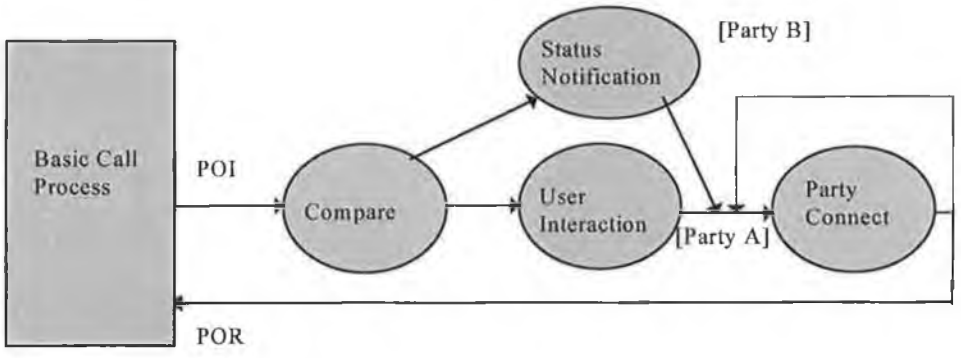


Figure 2.12 Ringback Service SIB Chain

The Basic Call Process (BCP) launches the SIB chain with a Point of Initiation (POI). The Compare SIB is used to compare the Party B (called party) number to the Party A (service initiator) number to ensure that they are not the same i.e. that Party A has not initiated an infinite looping Ringback service on itself. Once it has been established that this is not the case a User Interaction SIB is called to inform Party A that their request for the Ringback service has been accepted and is being processed. Simultaneously the Status Notification SIB begins to continuously monitor Party B's line for an on-hook status. On detecting Party B going on-hook the Party Connection SIB polls (rings) Party A. When Party A picks up and goes into the off-hook state the Party Connect SIB is called again but this time to poll Party B. The two parties are connected as soon as Party B goes off-hook. A Point of Return (POR) indicates to the BCP that the service has ended. IN call processing has terminated at this point and the POR indicates that the basic call processing is to be handed over to the call control function in the switch.

2.7.2 Group Call Pick Up Service

A user must be a member of a pre-defined group to use this service. A user at any extension within a particular group can automatically answer calls ringing on other group member's extensions by selecting the Group Pick Up option. The special Group Call Pickup digit-sting used to invoke the service is '70'. Figure 2.13 shows the SIB chain for Group Call Pickup.

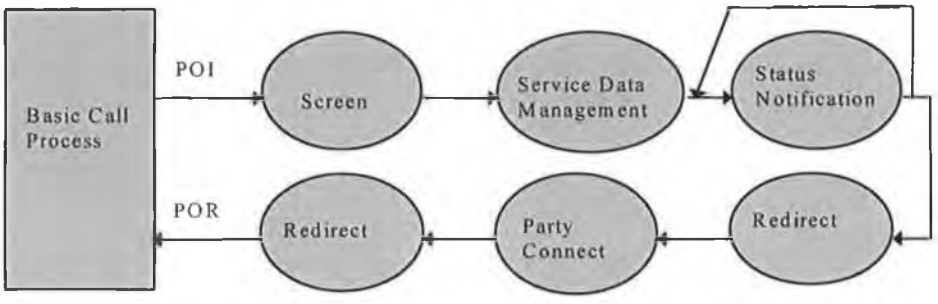


Figure 2.13 Group Call Pickup Service SIB Chain

The BCP detects the request for the IN group pick up service, and a POI instigates the SIB chain for the service. The service initiator (calling party) number is screened, using the Screen SIB, to find the group identity for this party. The Service Data Management SIB is then used to retrieve the list of party members for this group id. Each party line id in the party member list is then systematically monitored by the Status Notification SIB for their current status, until eventually a party with the current status 'BeingPolled' (Ringing) is found. Using the number of the party that is being polled the Redirect SIB queries the switch to find the original calling party. The Redirect SIB then terminates the Originating BCSM set up for the service initiator, and creates a replacement Terminating BCSM. The Party Connect SIB is given the original caller id and the service initiator id to connect the two. SIB Redirect cleans up, by killing the Terminating BCSM created for the initial called party, thus the first phone stops ringing. Call processing is then handed over to the Call Control Function by sending the POR to the BCP, which in turn instructs the switch to take over the processing.

2.8 The SIBs used to implement the Services

Seven SIBs were required to implement the two chosen services. Five of these SIBs (*Compare, User Interaction, Status Notification, Screen and Service Data Management*) are defined in Capability Set 1 (CS-1). Two new SIBs (*Party Connect* and *Redirect*) had to be

designed in order to make the implementation of these services possible. This fact serves to demonstrate the incompleteness of CS-1 as a defining standard.

SIBs are located in the GFP and map onto functionality located in the DFP. A SIB in the GFP will call one or more FEA contained within FEs in the DFP. The naming convention for each of the FEAs is as follows;

FEA_XYYZ, where [Q.1214]:

- X represents the Functional Entity:

  2 represents the CCF/SSF

  3 represents the SRF

  4 represents the SDF

  9 represents the SCF

- YY represents the section number of the SIB:

  Algorithm SIB is 01

  Compare SIB is 03

  Screen SIB is 08

  Service Data Management SIB is 09

  User Interaction SIB is 12

- Z distinguishes the particular FEAs which have a common XYY.

Therefore, for example, the Compare SIB located in the SCF will use FEA_9031.

2.8.1 Compare SIB

The Compare SIB performs a comparison of an identifier against a specified reference value, returning one of the three logical results:

- identifier is GREATER than the reference value,
- identifier is LESS than the reference value,
- identifier is EQUAL to the reference value.

Other logical results can be formulated by combining two of the logical outputs together (e.g.<=, >=, or <>). Potential service applications for this SIB include time dependent routing, and checking the relationship of current network time to a customer specified time. The compare function takes place in a FEA in the SCF.

There are two main comparison types identified in the CS-1 standards [Q.1213];

Identifier value - compare the identifier against the reference value. Time - compare network time to the reference time. Network time is specified as:

- time of day;
- day of week; or,
- day of year;

The use of the 'time' comparison type is not required for the current implementation of the compare SIB, and therefore the only comparison type available in this initial prototype is the 'value' comparison type.

Compare Input Data

The two types of input data for the compare SIB; Service Support Data (SSD) and Call Instance Data (CID) are described.

The Service Support Data (**SSD_Compare**) consists of the:

- comparison type, if it is a value or a time that is being compared,
- identifier type, whether the identifier value, and therefore reference value, is of type integer or a char string,
- integer value, this field is filled in with the identifier value if it's of type integer,
- char string value, this field is filled with the identifier value if it's type char string.

It should be noted that the compare identifier and reference value must be of the same type, i.e. both integer values or both char string values.

The Call Instance Data (**CID_Compare**) consists of:

- the integer value, this is the field which contains the reference value if it is of type integer,
- the char string value, this field contains the reference value if it is of type char string.

Compare Output Data

The output (or result) of the compare SIB is defined by the type **Compare_result**. There are four possible results; *greater_than, less_than, equal_value or error_found.* The error is described in the sort **SIB_error_cause** which accompanies the *Compare_result*, the possible errors for this SIB are *invalid_identifier* or *invalid_reference_value*.

2.8.2 Screen SIB

Screen performs a comparison of an identifier against a list to determine whether or not the identifier exists in the list. The two possible solutions are:

- there is a match for the identifier in the list, or
- there is no match for the identifier in the list.

The SIB could be used, for example, to verify a user ID or PIN, or for the selective call forward on busy/don't answer service. The screening function takes place in an FEA in the SDF functional entity.

Screen Input Data

The two types of input data for the Screen SIB, SSD and CID, are described.

Service Support Data (**SSD_Screen**)

- Screen list indicator, or database Id, indicates the list to search for the identifier match,
- key field name, the field in the database in which the identifier should be located,
- key field type, the type of the key value,
- key field integer, this field is filled with the key value if that value is an integer,
- key field char string, the value of the key if the key is of type char string.

Call Instance Data (**CID_Screen**)

The information in this type is partly filled in by the GSL prior to SIB initiation the remainder of the information is the result of the screening function carrier out by the FEA in the SDF. This type is then sent back as part of the SIB's result.

The contents of the CID_Screen are as follows:

- requested information field name, this presents the field in the database which must be checked against the reference or key value,
- requested information field type, is it of type integer or char string,
- requested information field integer, if the information is of type integer,
- requested information field's char string, if the information is of type char string.

Screen Output Data

The resultant output is a match, no match or a logical error. The type **Screen_result** contains this logical result and also the requested information. The sort **SIB_error_cause** accompanies the result information. The error cause for screen can be *invalid_identifier*, or *invalid_screen_list*.

2.8.3 Service Data Management SIB

The Service Data Management SIB enables user specific data in the database to be replaced, retrieved, incremented, or decremented. An example application would be to retrieve or replace a customer's call forwarding number. The service data management operations takes place in an FEA in the SDF functional entity.

SDM Input Data

The two types of input data for the SDM SIB, SSD and CID, are described.

Service Support Data (**SSD_SDM**) contains:

- database ID, this indicates the subscriber data file to be operated on,
- operation, whether this is a replace, retrieve, increment, or decrement operation,

- offset value, specifies the amount by which the data element is to be incremented or decremented.
- new integer value, the replacement value for the data element if it is of type integer,
- new char string value, the replacement char string if the data element to be replaced is of type char string.

Call Instance Data (**CID_SDM**) contains

- key field name, the name of the field used as a key to indicated the data,
- key field type, the type of the information contained in the key field i.e. int or char,
- key field integer, if the key element is of type integer this field contains the key value,
- key field char string, if key field type is of type char string then this field contains the key value for indicating the data,
- retrieved information,
  - the return value name, the field in which the return value exists,
  - the return value type, whether it is an array of integers, or of char strings,
  - return integer array, the array of integers being returned,
  - return char string array, the array of char strings being returned.

SDM Output Data

The information returned from this SIB depends on the type of operation that has been performed. The success of any of the operations is reported by a simple boolean value, **logicalError**. If the result is positive, i.e. an error has occurred, the reason for this error is given by the accompanying **SIB_error_cause** parameter. The possible error causes for this SIB are *invali_ file_indicator, invalid_operation, invalid_reference_value, invalid_information_value, invalid_offset_value.*

The retrieve action has its own special output report, consisting of the retrieved information, described above in the CID information.

2.8.4 Status Notification SIB

The Status Notification SIB allows the SCF the capability of keeping track of the status of network calls or resources, and the option to store the status in the SDF. There are three possible types of monitor the SIB can place on a resource or a call:

- A continuous monitor monitors the resource continuously for a set length of time,
- A next change monitor traces the resource until the next change in status and then notifies the SCF, and
- A current status check checks the present status of the resource and reports it immediately to the SCF.

There is a time limit set for both the continuous and the next change monitors. A resource can be in one of seven states:

- idle,
- waiting for a dial-tone,
- waiting for a dial-string,
- polling,
- being polled,
- connected,
- off hook following alert.

The monitoring function is performed by the CCF, which is instructed by an FEA in the SCF via the SSF functional entity.

Status Notification Input Data

There is one type of input data **SSD_StatusN**. Internal to the SIB the **StatusReportArg** data type relays the required information between the SCF and the CCF via the SSF.

Service Support Data (**SSD_StatusN**) contains three fields:

- monitor type, continuous, next or current monitor,
- call party, the resource id on which the monitor is to be placed,
- monitor duration, the duration that a continuous or next monitor can last.

The Status Report Argument contains data sent between the SCF and the switch with the request to monitor a resource. This argument contains the members:

- monitor type,
- monitor duration,
- current line id, the resource id to be monitored,
- resource status, the status required by the SCF, or the status detected by the CCF
- logical error indication, a boolean value with true indicating that errors are present,
- error code, this is the SIB_error_code found if there was an error.

Status Notification Output Data

There are two items of data returned by the Status Notification SIB: **StatusN_result** and the **SIB_error_cause**. StatusN_result contains:

- the resource status,
- the resource id,
- the logical result, whether there was an error or not.

The possible errors for this SIB are: *invalid_type, invalid_resource, invalid_timer, inconsistent_timer_set, invalid_resource_status, status_timer_expired.*

2.8.5 User Interaction SIB

User Interaction is a SIB that provides interaction functions with the user, such as the playing of an announcement, or the collecting of information from a party. The announcements can be customised or generic audio message, DTMF tones, or network progression tones. The collected information can be DTMF tones, audio, IA5 String Text etc. This SIB provides the specified announcement to the user, and depending on the repetition type, the message is repeated a set number of times, or for a set time length. A functional entity action in the SCF

firstly directs the SSF to set-up a connection to the SRF for the traffic of the user interaction switch instructions. The SCF then instructs the SRF on what operation to perform and an information flow takes place between the FEA in the SRF and the SSF. On completion of the interaction phase the SCF is alerted with the result. It then instructs the SSF to close down the connection to the SRF FEA. The collection of information interaction is not required at this stage of the project.

User Interaction Input Data

SSD and CID are both sent to the User Interaction SIB.

Service Support Data (**SSD_UI**) consists of:

- connected SRF Id, returned by the SSF after the SRF connection has been set up,
- information to send, this is the information that is passed on to the SSF in the form of **PlayAnnouncementArg** and includes:
  - announcement type, the announcement to send,
  - request for repetitions, whether the announcement should be repeated,
  - the type of repetition, time repeat or count repeat,
  - the repetition interval, time between repetitions,
  - maximum number of repetitions, if the repetition type is of type count,
  - maximum repetition duration, if the repetition type is of the type time.

Call Instance Data (**CID_UI**) contains:

- collect information indicator, boolean value set to true if there is information to be collected,
- call party Id, the resource/ destination Id with which to interact

User Interaction Output Data

The result is contained in the type **UI_result** and contains a logical error result, whether there is an error or not. It is accompanied by the **SIB_error_cause** value, which can be set to any

of the following: *call_abandon, collection_time_out, incorrect_digit_string, announcement_resource_unavailable, data_collection_resource_unavailable, invalid_announcement, invalid_call_party, inconsistent_timer_setting, call_status_incompat_with_ann, call_ status_incompat_with_info_collect.*

2.8.6 Party Connect SIB

The first newly designed SIB (not provided by CS-1) is the Party Connect SIB whose function is to instruct the switch to poll (ring) and to connect different parties. There are three different operations that this created SIB can provide:

- party connect, connecting two parties,
- poll party, ringing the defined party,
- poll and connect party, poll the identified party and then when the party answers connect to the second party.

The functionality is located in the FEA at the SCF, which sends the appropriate instructions to the switch via the SSF.

Party Connect Input Data

There is no service support data defined for the Party Connect SIB, only CID.

Call Instance Data (CID_PartyC) contains the following information:

- connection type, whether it's a poll, poll and connect or connect,
- caller id, the service initiator Id,
- party A id, the identity of the first party to be polled for a poll connection, to be connected to in a poll and connect operation, and to be one of the connect parties for the connect action,
- party B id, the identity of the second party involved in the Party Connect action, B is the polled party in the poll and connect operation, and second connect party for the connect operation,

- first party polled indication, this is a simple Boolean value to indicate that the first party has already been polled or is ready for the connect state, for the poll and connect operation.

Party Connect Output Data

The result, **PartyC_result**, of the SIB is output along with the **SIB_error_cause**. It contains a logical value, indicating the presence of an error, or not. Possible error causes for the Party Connect SIB are: *invalid_party_id, invalid_resource_status.*

2.8.7 Redirect SIB

The second newly designed SIB (not provided by CS-1) is the redirect SIB, used in the Group Pick Up service for redirecting the call from the original called party to the group member party who picks up. The actions for this SIB are again performed in the SCF, which directs the switch on the particular steps.

There are three different operations making up the functionality of the SIB. These can be processed as detailed or may be performed individually. The operations are;

- original calling party, determines the original caller,
- terminate party Id, terminates any unneeded basic call state model (BCSM), either terminating or originating for the particular party id given,
- create party Id, creates the required BSCM for the indicated party.

Redirect Input Data

The input data for this SIB is contained in one type, CID.

Call Instance Data (**CID_Redirect**) consists of:

- full redirect indicator, flag showing whether the full redirect procedure is to be performed or if only a single redirect operation is required,
- the redirect phase, what redirect operation is to be performed, if the single operation option is chosen above,
- create party Id, the identity of the party for which a new BCSM is to be created,

- terminate party Id, the identity of the party whose BCSM must be terminated.

Redirect Output Data

The return values from the redirect SIB are contained in the two types, **Redirect_result** and **SIB_error_cause**. The result type contains

- the redirect phase, if it was only one operation that was performed,
- originating success, a logical result showing the original calling party id was found,
- creation success, a logical value indicating successful creation of a BCSM,
- termination success, a logical value indicating successful termination of a BCSM.

There is only one error associated with this SIB: *Invalid_Party_Id.*

2.9 INAP and the addition of new SIBs

The Intelligent Network Application Protocol (INAP) [Q.1218] specifies the information flows to be exchanged between the different entities of the IN functional model in terms of protocol data units (PDUs) described by ASN.1 (Abstract Syntax Notation #1). The PDUs themselves represent Remote Operations in the scope of the Transaction Capability Application Part (TCAP). For CS-1 the INAP messages between the different functional entities are already defined.

Given that two new SIBs which are not specified by CS-1have been created (*Party Connect* and *Redirect*) it is clear that new INAP messages will also need to be created. The new INAP messages between the FEs because of the Party Connect SIB are as follows:

From the Party Connect SIB to the SSF (in signal list PC_2_SSF of Figure 4.4);

- Poll_req_ind
- Poll_Connect_req_ind
- Connect_req_ind

From the SSF to the Party Connect SIB (in signal list SSF_2_PC in Figure 4.4);

- Poll_resp_conf

- Poll_Connect_resp_conf
- Connect_resp_conf

The information flows between the FEs due to the creation of the Redirect SIB are:

From the Redirect SIB to the SSF (in signal list Rdir_2_SSF in Figure 4.4);

- OrigCallerId_req_ind
- TerminateBCSM_req_ind
- CreateBCSM_req_ind

From the SSF to the Redirect SIB (in signal list SSF_2_Rdir in Figure 4.4);

- OrigCallerId_resp_conf
- TerminateBCSM_resp_conf
- CreateBCSM_resp_conf

In defining these new SIBs and information flows it was necessary to go beyond the scope of INAP. This again serves to demonstrate the incompleteness of the current standards.

2.10 State of the Art

This Section explains the current state of the IN and TINA technologies.

2.10.1 IN Today

The Intelligent Network (IN) concept has evolved gradually from the 1960s when Common Channel Signalling (CCS) was first introduced in the United States telecommunications network. Subsequent developments led to the introduction of direct service dialling, Freephone services, Virtual Private Networks (VPN), etc. The term Intelligent Network emerged in 1983/84 as part of the work done in the USA by Bellcore and others to define the first non-proprietary network design capable of supporting Freephone, credit card verification and VPNs. In 1988 the Telecommunication Standardisation Sector of the International Telecommunications Union (ITU-T) and the European Telecommunications Standards Institute (ETSI) started work on IN standards. The key goal of this standardisation has been to enable the development of network systems that could support rapid deployment of new

services without having to redefine/enhance the network infrastructure with new call processing features and signalling interfaces.

Today Intelligent Networks are in full commercial service in many European countries, the Asia-Pacific region and North America. Many other countries are also planning their introduction. The most widely deployed services in IN are Freephone, Virtual Private Networks, cashless calling and premium rate services. IN is seen as a key element for competitive advantages in the telecommunications market. For telecommunication operators, service customisation, provisioning and management are the immediately assured applications of IN. The major driving forces for IN are to speed up time to market, to provide more effective use of capital investment and to provide the ability to rapidly create new services.

The work carried out by ITU and ETSI is leading to increased world-wide standardisation of IN. The current version of the ITU standards is Capability Set 2 (CS-2) which was recently released. The services and SIBs in this project, designed before the release of CS-2, is based on CS-1. As CS-2 is a superset of CS-1 there should not be any compatibility problems when services based on CS-2 (and later Capability Sets) are implemented.

2.10.2 Existing IN Models

Using the SDL/SDT suite, models of the IN architecture have been implemented to study various aspects of the Intelligent Network Conceptual Model and the suitability of SDL as a development language.

A model implemented by [Csu94] was developed to explore the IN Recommendations [Q.1200] to [Q.1218] and to be used as a working illustration of the Intelligent Network paradigm. This project also pointed out various ambiguities in the Recommendations.

A model of the INCM developed by [Morris] incorporated a Service Creation Environment (SCE). A Universal Personal Telecommunications service was developed on the SCE and its performance and that of the INCM model were evaluated. The work done on this project proved that SDL is extremely suitable for the development of models of the INCM as it is very adaptable and conducive to extendibility, which is a key aspect of the IN architecture.

Both of these models are restricted to the situation where a single user can interact with the model and only one service can execute at any one time.

While these models were very helpful in clarifying ambiguities in the Q Series of IN standards [Q.1200] it was felt that the development of a real IN system, which could handle actual user calls, would be extremely beneficial. The system developed in this project is not a model of the INCM but an actual implementation of a working and extendible IN system. Users attached to a switching platform can set-up standard calls or invoke IN services in real time. It was felt that implementing a simultaneous multi-user, multi-service system would give a more complete understanding of how a platform provided by a commercial IN service provider would actually behave and what real implementation and design issues would be encountered.

2.10.3 TINA Overview

Telecommunications Information Networking Architecture (TINA) is an effort carried out by the TINA Consortium (TINA-C) to overcome current network limitations in the provisioning of services, fully exploiting the integration of Telecommunications and Information Technology. TINA-C is an international collaboration aimed at defining and validating an open architecture for telecommunication systems for the broadband, multimedia and information era. TINA addresses the needs of traditional voice based services, future interactive multimedia services, information services, and operations and management type services, and provides the flexibility to operate services over a wide variety of technologies.

In a TINA network the process space within which all the objects communicate is extended using a Distributed Processing Environment (DPE). CORBA is a leading example of a DPE and has been selected as the TINA-C DPE of choice. If the objects are to be distributed an underlying platform must be defined to mediate the communication and presentation aspects between objects. Classes of interfaces can be defined using CORBA's Interface Definition Language (IDL) and then automatically translated into language specific constructs.

The concepts of a *Role* and *Stakeholder* determine the Business Model of the TINA architecture. A Role is a position in the telecommunication market while a Stakeholder is a business entity such as a company. In Figure 2.14 the business model covering the service-provisioning phase is shown. The identified Roles in this case are the following:

- **Customer**: User of services provided by the TINA system.
- **Service Retailer**: Provides Customers with access to Services (it may use other providers to support the provision of these services).

- **3rd Party Service Provider (3SP)/Content Provider (CP)**: Aims to support Retailers and other third party providers with services but does not have direct obligation to the Customer.
- **Broker**: Provides stakeholders with information that enables them to find other stakeholders and services in the TINA system.
- **Connectivity Providers**: Providing transport connectivity on the Access Network and/or on the Backbone Network.

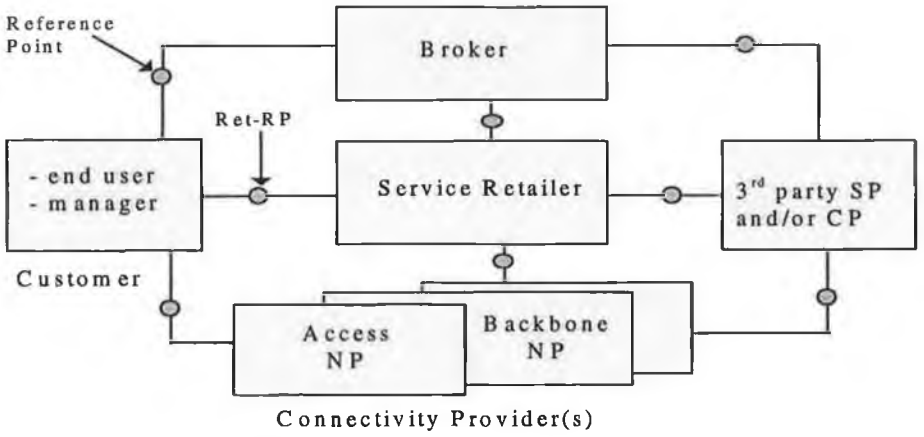


Figure 2.14 TINA Business Model

Reference Points between different Roles are identified in order to support interoperability. The Reference Points comprise a set of interfaces describing the interactions taking place between the Roles. For example, the Retailer Reference Point (Ret-RP) describes the relation between Customer and Service Retailer.

A Broker mediates with a user to find an appropriate Service Provider (SP). The chosen Service Provider is then the Retailer for that service. The Retailer mediates access to the services offered by 3rd Party Service Providers by providing added value functionalities to the Customer. In addition it also supports the user in service management related activities, such as subscription, service customisation, management of customer profiles and data, and billing procedures. The Retailer acts as a broker for 3SPs and Connectivity Providers (CPs) by filtering user requests and assuring the Quality of Service (QoS).

The TINA Architecture is decomposed into four main parts:

- **Computing Architecture:** defines a set of concepts and principles for the design and building of distributed software and the software support environment.

- **Service Architecture:** defines a set of concepts and principles for the design, specification, implementation and management of telecommunication services.
- **Network Architecture:** defines a set of concepts and principles for the design, specification, implementation, and management of transport networks.
- **Management Architecture:** defines a set of concepts and principles for the design, specification, and implementation of software systems that are used to manage services, resources, software, and underlying technology.

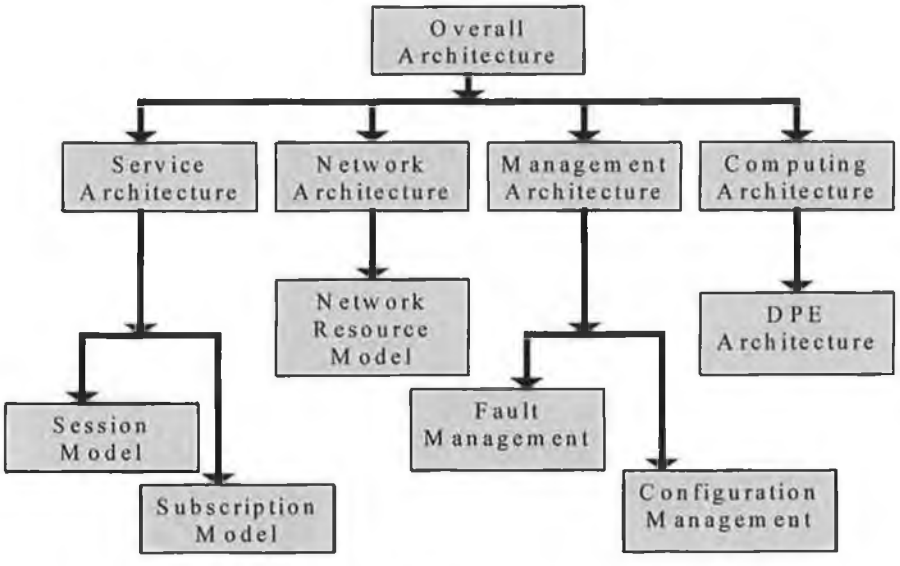


Figure 2.15 Decomposition of the TINA Architecture

2.10.3.1 The Computing Architecture

The Computing Architecture describes a telecommunication system from five viewpoints:

- **Enterprise**: Focuses on the purpose, scope and policies for the system.
- **Information**: Focuses on the semantics of information and information processing activities in the system. It shows how a system is specified in terms of information entities (objects, classes) and the relationships between them.
- **Computational**: Focuses on the decomposition of the system into a set of interacting objects that are the candidates for distribution.
- **Technology**: Focuses on the choice of technology to support the system.

The Computing Architecture also defines the Distributed Processing Environment (DPE) lying over the Native Computing and Communication Environment (NCCE) of computing nodes. The DPE provides the support system allowing objects to locate and interact with each other.

2.10.3.2 The Service Architecture

There are three main sets of concepts and principles in the service architecture:

- Session concepts, which address service activities and temporal relationships,
- Access concepts, which address user and terminal associations with networks and services, and,
- Management concepts, which address service management issues.

Session Concepts

Although services may differ in nature they all share the fundamental property of providing a context for relating activities - termed a Session. Four types of sessions have been identified:

- A **Service Session** is the single activation of a service. It relates the users of the service together so that they can interact with each other and share entities. A service session contains the service logic and is computationally represented by a *Service Session Manager* (SSM). An SSM offers two types of operational interfaces. The first is a *Generic Session Control Interface*. This provides the operations that allow users to join and leave service sessions. For certain services it may also offer operations to suspend and resume involvement in a service. The second type of interface will provide *Service Specific Operations*, and will be dictated by the capabilities offered by the service logic.
- A **User Session** maintains state about a user's activities and the resources allocated for their involvement in a service session. Examples of states held in a user session include the user's accumulated charge, suspension and resumption history, and service specific state. When a user joins a service session a user session is created. It is deleted when the user leaves. The service session maintains links to the user sessions and thus provides a group oriented view.

- A **Communication Session** is a service-oriented abstraction of connections in the transport network. A communication session maintains state about the connections of a particular service session, such as the communication paths, End Points and Quality of Service (QoS) characteristics. A communication session is only required when streams between users (or resources) are required. Computationally a *Communication Session Manager* (CSM) provides the features of a communication session.
- An **Access Session** maintains state about a user's attachment to a system and their involvement in services. A user can attach to a system in order to launch new or join existing service sessions. A user may be involved in many services at the same time and an access session maintains state about this involvement.

Access Concepts

Users need flexible access to services in terms of the locations from which they access the service and the types of terminals they use. User access is therefore distinguished from terminal access. An agent concept is used in defining the access model. The following types of agent have been identified:

- A **Provider Agent** (PA) provides the access mechanism to the Retailer domain and connects a user to their User Agent.
- A **User Agent** (UA) receives requests from users to establish service sessions, or to join existing service sessions, and creates or negotiates with existing service sessions as appropriate. Creation is actually carried out by the User Agent sending a request to a *Service Factory* (SF). A factory is a DPE service that can dynamically create objects. The creation of a service session is subject to subscription and authentication checks. A User Agent also receives and processes requests to join a service session from service sessions themselves. This is a form of incoming call processing where another user has created a service session and invites the user to join. UAs know the subscribed services that a user may create. This list can be presented to the user when the user logs onto his UA.
- A **Terminal Agent** (TA) is responsible for representing a terminal and obtaining the precise location of the terminal. Two examples are, which network access point a portable computer is attached to, and which cell a mobile phone is currently in.

In order to access a service, users must associate their User Agents with Terminal Agents. This may form part of a logging on process to establish an access session. A user may be simultaneously associated with many terminals. For example, in a video conference a user may be using both a workstation and a telephone. Similarly a terminal may be simultaneously associated with many users, for example, when in a meeting all users associate their user agents with the telephone in the meeting room. On receiving incoming session requests, a UA has to determine which TA should be contacted. If the user is currently accessing the system then an announcement could be issued to one of the terminals being used, and the user can instruct the UA which terminal to use. Otherwise the UA will have to determine which terminal and pass a request to the TA who can alert the user. This determination can be done through user registration, preferences, and defaults.

2.10.3.3 The Network Architecture

The purpose of the network architecture is to provide a set of generic concepts that describe transport networks in a technology independent way, and to provide mechanisms for the establishment, modification, and release of network connections. The network architecture defines a set of abstractions that the resource layer can work with. At one end it provides a high level view of network connections to services. At the other end it provides a generic descriptions of elements, which can be specialised to particular technologies and products.

2.10.3.4 The Management Architecture

The TINA management architecture provides the concepts and principles to build management systems, which can manage TINA entities. The Management Architecture takes as input the TMN Recommendation M.3010 [M.3010] that defines the layers in which the management functionality may be considered to be partitioned for operational purposes. As with the service architecture, the computing architecture is used to define object types and interfaces that should be used to manage TINA services, resources and infrastructures.

2.10.4 Existing Paths for Migration from IN to TINA

Typically, the types of services implemented on an Intelligent Network are narrow-band call control services while those for a TINA network are broadband multimedia type services. It is becoming increasingly more important for network operators to study the enhancement and evolution of their IN service platforms in order to cope with new customer requirements. Interworking between IN and TINA is a step towards the full migration to TINA. It makes possible the accessing of IN service logic or data by TINA objects and, conversely, the accessing and invocation of TINA services or functionality by IN entities. Hybrid IN/TINA services may be developed which contain aspects of both technologies, therefore yielding a richer service platform.

Two major evolutionary steps have been identified [INtoTINA]: The *Bridge to Legacy Path* involves the replacement of the SCF with appropriate TINA objects while the SSF remains IN compliant. The *Open Switch Path* adapts the switch control interface so that the switch appears as a TINA object. It is important to note that there is not a one-to-one mapping of IN functional entities to TINA computational objects. Adaptation Units (AUs) are therefore required to provide the interworking between the TINA Computational Objects and the legacy IN [HerzMage].

2.10.4.1 Open Switch Path

This path of migration from IN to TINA might be preferred by switch vendors as it involves wrapping their existing switches so that they appear as software objects in a TINA network. The solution is based on a wide distribution of intelligent functions over a network of servers and terminals and on a manager-agent approach for controlling the switching functions. The network intelligence moves from a few centralised SCPs to a "web" of network servers that support a seamless interaction with CPE/CPN (Customer Premises Equipment/Customer Premises Network) and Service Provider's equipment. Network servers are designed to support specific services or functions. CORBA based platforms that enable applications to expose and use rich IDL (Interface Definition Language) interfaces for controlling and managing services support the interaction between software components. This use of public interfaces allows the definition of open and scalable service-specific signalling. The control and programmability of switching functions is possible by means of rich and published interfaces exposed by so-called "Open Switches".

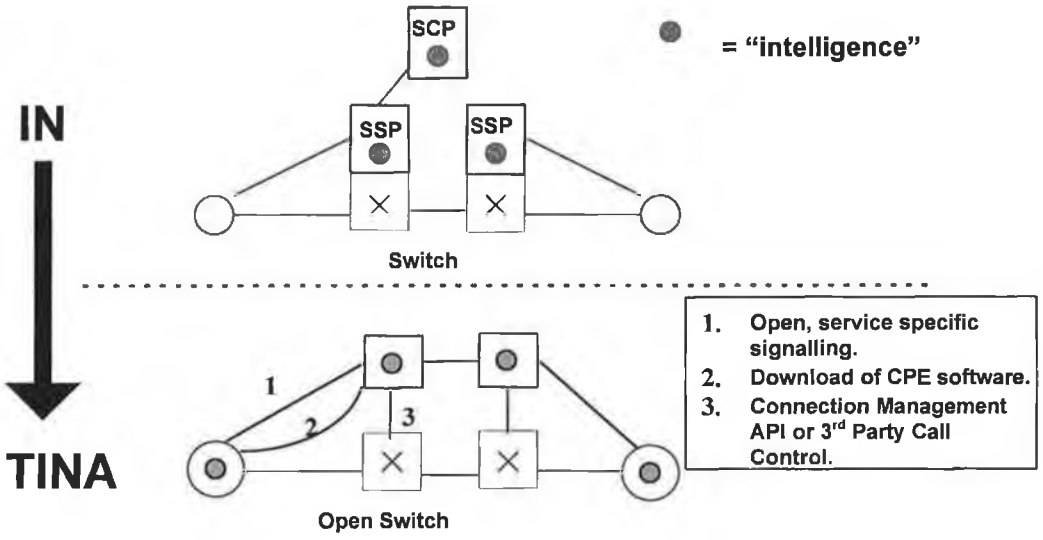


Figure 2.16 Open Switch Path

Figure 2.16 represents a possible applicability scenario for the Open Switch path. In the first stage the CPE accesses the network intelligence that offers retailing functions by means of the specific protocol; the Retailer Reference Point (Ret-RP). When the choice of a service or provider is made, the software needed to use the requested service is downloaded. The CPE is then loaded with a full-fledged set of functions for interacting with the service. Upon user request, network servers of the intelligent layer can orchestrate the network resources on behalf of the users and provider satisfying their communication needs. This solution is characterised by:

- A high degree of distribution of the intelligence. Intelligent functions are equally scattered between CPEs, network servers, and Service Provider's equipment;
- Some intelligent functions are moved from switches to the intelligent layer;

2.10.4.2 Bridge to Legacy Path

This solution is aligned with the current implementation of IN. It impacts on SCP and Service Management Systems (SMS) without spoiling the current legacy of network systems. The bulk of intelligence is deployed in a network of servers that provide advanced functions. The control over switching functions is exerted by means of current or future versions of INAP protocol [Q.1218]. The network intelligence moves from the centralised SCPs to a "web" of network servers designed to provide specific functions. Interactions with CPEs may be mediated by network equipment. A user may access a service either from the SSF side of the

gateway or from the TINA side via an access session using the Ret-RP protocol. The interaction between software components within the intelligent layer is supported by CORBA platforms.

Figure 2.17 shows that intelligent functions may be triggered in the same way as for an IN structured network. SSPs forward INAP messages to the distributed SCP via a gateway that maps signalling primitives to CORBA IDL interfaces. Services can be provided according to TINA technology, using Adaptation Units (AUs) to interact with the non-TINA world. A user may also launch a service from their Terminal/Browser via an access session using the Ret-RP protocol.

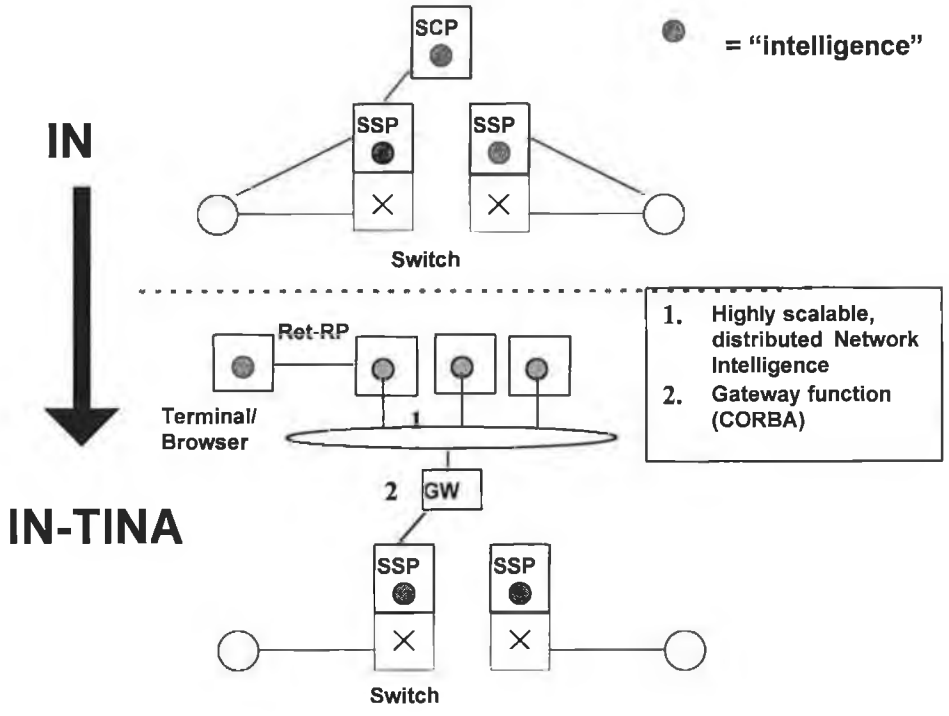


Figure 2.17 Bridge to Legacy Path

This solution is characterised by:

- Consistency with the legacy IN systems. The application of TINA technology in this way is very compatible with the existing IN approach. New services developed on the open platform can coexist with already deployed services supported by existing SCPs,
- Alignment to the evolution of IN. The flexibility of this architectural solution is strongly tied to the evolution and the capabilities standardised for future releases of IN Capabilities Sets. The definition of mid-call triggers, non-call related functions, user-to-service signalling, and the integration with B-ISDN could greatly improve the ability to provide more advanced services,

- High degree of distribution of the intelligence within the network infrastructure. Intelligent functions are distributed in a network of specialised servers, services can be specialised on a customer basis,
- Management and control functions can be easily integrated due to the use of a common processing platform and by the use of TINA solutions.

2.11 Conclusions on IN

ITU-T and ETSI are responding to the telecommunications industry's need for IN standardisation and evolution with a program of work that addresses global IN standards and a framework for the standardised evolution of IN. An immediate problem for vendors is that the CS-1 standards have a very restricted scope: full IN standardisation is an ideal, not a current reality. CS-1 itself has shortcomings, the standards are not sufficiently unambiguous to guarantee incompatibility between different vendor products. CS-1, though not being specified in 100% detail, is however conceptually complete and is the first of many planned sets of capabilities.

To date IN development has been aimed at the construction of a set of interfaces and protocols that clearly separate the switching from the service aspects of telecommunication networks. IN is beginning to address the potentially distributed nature of the service application software itself and its interoperability with distributed networks such as TINA. This development is discussed further in Section 4.9. The Telecommunications Information Networking Architecture Consortium (TINA-C) has been formed by the major network operators to discuss information networking problems with telecommunications and computing vendors. TINA-C has developed the TINA architecture, which builds on current advances in broadband communications and distributed computing technologies, specifying a software-based architecture for future information networks.

Chapter 3 TOOLS FOR DEVELOPMENT

3.1 Introduction

This chapter discusses the software tools necessary for the development of the IN architecture described in the previous chapter. These tools include the Specification Description Language (SDL) used to describe the Intelligent Network, the SDL Design Tool (SDT) used to implement the design and the CORBA Architecture which facilitates the execution of distributed applications.

3.2 Specification Description Language (SDL)

Developed primarily to meet the software design needs of the telecommunications industry, Specification Description Language (SDL) [SDL89] is a graphical specification language that is both formal and object oriented. Its strength is its ability to describe the structure, behaviour and data of real-time interactive and distributed systems. It provides a means by which the structure of a system may be laid out in a hierarchical manner allowing different levels of abstraction, from overview to detail. SDL has been designed to provide the specific capabilities needed to produce high quality telecommunications software and to minimise the development cost. The specifications and descriptions expressed in SDL are intended to be formal in the sense that they may be analysed and interpreted unambiguously. SDL Specifications may be represented in the form of a Graphical Representation (GR) and/or an equivalent Phrase (or textual) Representation (PR).

A system described in SDL consists of one or more blocks connected by channels, as illustrated in Figure 3.1. Every block or channel may be further decomposed into blocks and channels at any level.

At the lowest level a block must contain one or more processes, as shown in Figure 3.2. The actual behaviour of the system is described by the combined behaviour of a set of processes, each of which is modelled as an extended finite state machine that works concurrently with other processes. Some of the entities typically contained in a process are shown in Figure 3.3.

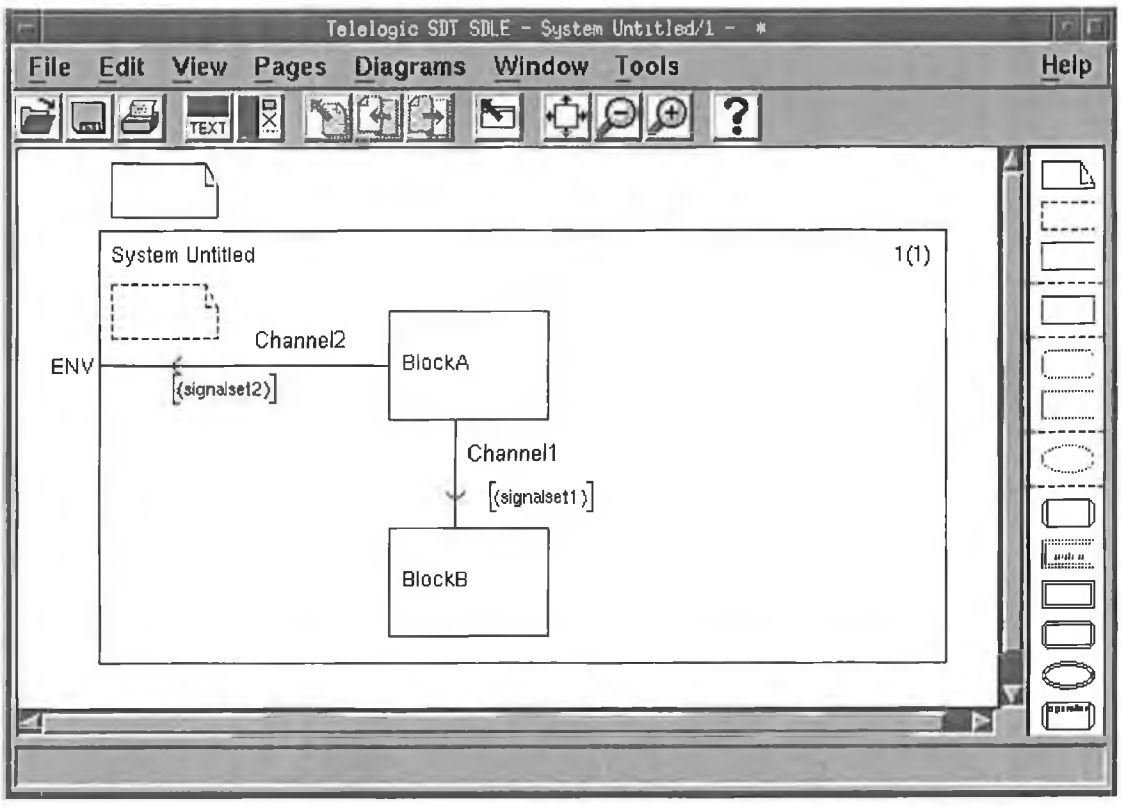


Figure 3.1 SDL System at System Level

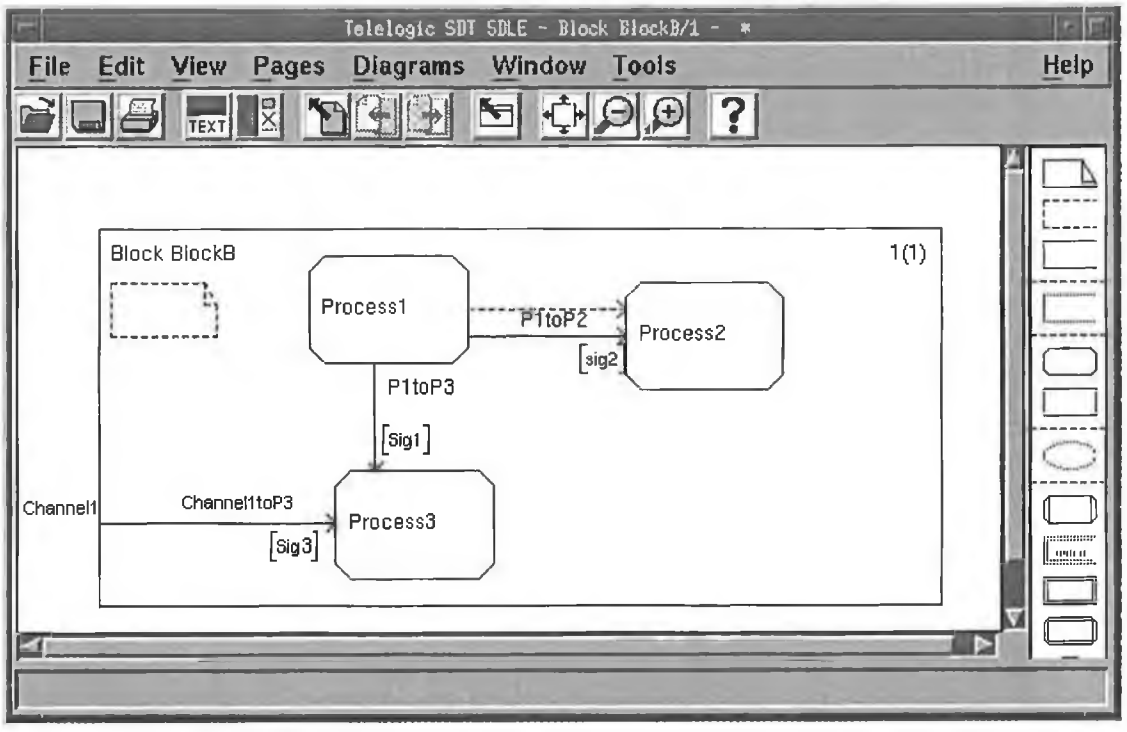


Figure 3.2 SDL System at Block Level

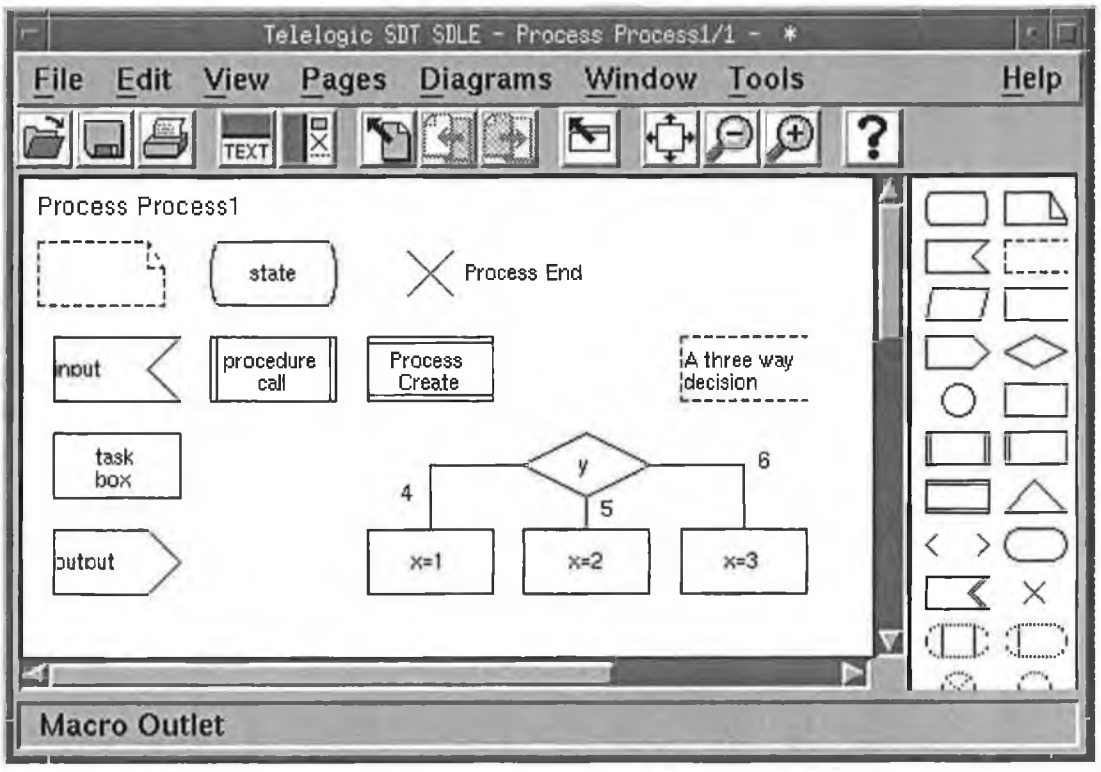


Figure 3.3 SDL System at Process Level

Co-operation between processes is performed asynchronously by discrete messages, called signals. The channels between processes are the carriers of these signals. A *signalset* is a set of signals that flow on a given channel.

The data type concept in SDL is based on Abstract Data Types (ADT). In other words a data type is defined in terms of its abstract properties rather than in terms of some concrete implementation.

In SDL each process has a local memory for storing its variables. A variable can be of a predefined or user-defined type. A process can manipulate its variables by assignment task actions. The values of variables can, for example, be used to control a transition by a decision action. Signals can carry values as parameters. When sending a signal, the actual values are specified as part of the output symbol. In receiving processes, the values are stored in local variables specified by the input action.

The Environment denotes anything outside the SDL system. The SDL system may communicate with objects located outside its boundaries but a limitation is that it is not possible to distinguish between different objects. Communication with the environment is achieved using the Environment Functions, which can send and receive messages to and from

objects outside the boundary of the SDL system. The Environment Functions are discussed in detail in Chapter 4.

The state machine symbols in a process are connected by arcs, which convey the direction of control flow as the process executes symbol by symbol. States in SDL processes are waiting places for events and the process is blocked for that event and transitions out of the state only when that event becomes true. Decision symbols are non-blocking which means the condition or expression is evaluated and whichever branch has the value that branch is followed. Another process may be created from within a process using the *create process* entity. Also, procedures may be called using the *procedure call* entity.

3.3 SDL Design Tool (SDT)

SDT is the tool developed by Telelogic to ensure correct implementation of the SDL standard. SDT comprises all SDL rules and syntax [SDL89] and also facilitates the analysis of system operation along with target code generation for various real-time operating systems [SDT93]. Via tools for simulation, graphical debugging of systems is also possible. Validation of design systems helps ensure conformity to requirement specifications, along with detection of temporal anomalies such as race conditions, buffer overflows, and deadlocks. Figure 3.1, Figure 3.2 and Figure 3.3 are screen shots of the SDT Tool. The tools incorporated into the SDT package are outlined in Table 3.1:

TOOL	PURPOSE
Editor	Allows specification to be implemented graphically.
Message Sequence Chart (MSC) Editor	Allows creation and editing of MSCs which can be verified against an SDL system using the SDT Validator.
Analyser	Performs syntax, semantic, and dynamic analysis of SDL descriptions.
C Code Generator	Converts SDL code to C.
Simulator	Allows simulation and testing of the system operation.
Performance Simulator	30-40 times faster than the ordinary simulator and can accept input from a file.
File Viewer	Keeps track of files generated by the Editor.

Table 3.1 Major Tools of the SDT Suite

SDT provides the ability to step through a specification and observe its behaviour, i.e. simulate it. This is an essential requirement in developing complex systems. This simulation activity can be achieved by using graphical user interfaces that can interact with the underlying SDL specification to both drive and observe its resultant behaviour, or relevant parts of the resultant behaviour. The Simulator tool allows for the manual examination of the specification to be achieved by executing transition by transition. The Simulator also allows for the internal status of the specification to be checked, e.g. to check input queues of certain processes, or see the values of certain variables or timers. For complex systems the tool allows for the hiding of certain events or for considering only those transitions associated with particular processes in the system, or the environment of the system.

Message Sequencing Charts (MSC) can be generated to record the simulation paths of interest. These can also be used for regressive testing of specifications, e.g. when a specification has its functionality extended, it is necessary to ensure that its previous behaviours are still present. MSCs can be seen in Appendix E.

3.4 CORBA

This section aims to give an introduction to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA). It provides an understanding of the basic mechanics of the architecture and its components.

3.4.1 The Object Management Group (OMG)

The Object Management Group is a non-profit consortium created in 1989 with the purpose of promoting the theory and practice of object technology in distributed computing systems. It aims to reduce the complexity, lower the costs, and hasten the introduction of new software applications. Originally formed by 13 companies, OMG membership has grown to over 500 software vendors, developers and users. OMG realises its goals through the creation of standards that allow interoperability and portability of distributed object-oriented applications. They do not produce software or implementation guidelines; only specifications that are put together using ideas of OMG members who respond to Requests For Information (RFI) and Requests For Proposals (RFP). The strength of this approach comes from the fact

that most of the major software companies interested in distributed object oriented development are among OMG's members.

3.4.2 The Object Management Architecture (OMA)

OMA is a high-level vision of a complete distributed environment. It consists of four components that can be divided roughly into two parts:

- System oriented components, consisting of;
  - Object Request Brokers (ORBs) and
  - Object Services, and;
- Application oriented components, consisting of;
  - Application Objects and
  - Common Facilities.

Of these parts, the Object Request Broker (ORB) is the one which constitutes the foundation of OMA and manages all communication between its components. It allows objects to interact in a heterogeneous, distributed environment, independent of the platforms on which these objects reside and the techniques used to implement them. In performing its task it relies on Object Services which are responsible for general object management such as creating objects, access control, keeping track of relocated objects, etc. Common Facilities and Application Objects are the components closest to the end user, and in their functions they invoke services of the system components.

3.4.3 The Common Object Request Broker (CORBA)

CORBA specifies a system that provides interoperability between objects in a heterogeneous, distributed environment and in a way transparent to the programmer. Its design is based on the OMG Object Model.

3.4.3.1 The OMG Object Model

The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation independent way. In this model clients request services from objects through a well-defined interface. This interface is specified in OMG IDL (Interface Definition Language). A client accesses an object by issuing a request to the object. The request is an event and carries information including an operation, the object reference of the service provider, and actual parameters, if any. The object reference is an object name that defines an object reliably.

3.4.3.2 The Basic Mechanics of issuing a request

The central component of CORBA is the Object Request Broker (ORB). It encompasses the entire communication infrastructure necessary to identify and locate objects, handle connection management and delivers data. In general, the ORB is not required to be a single component; it is simply defined by its interfaces. The ORB Core is the most crucial part of the Object Request Broker; it is responsible for the communication of requests.

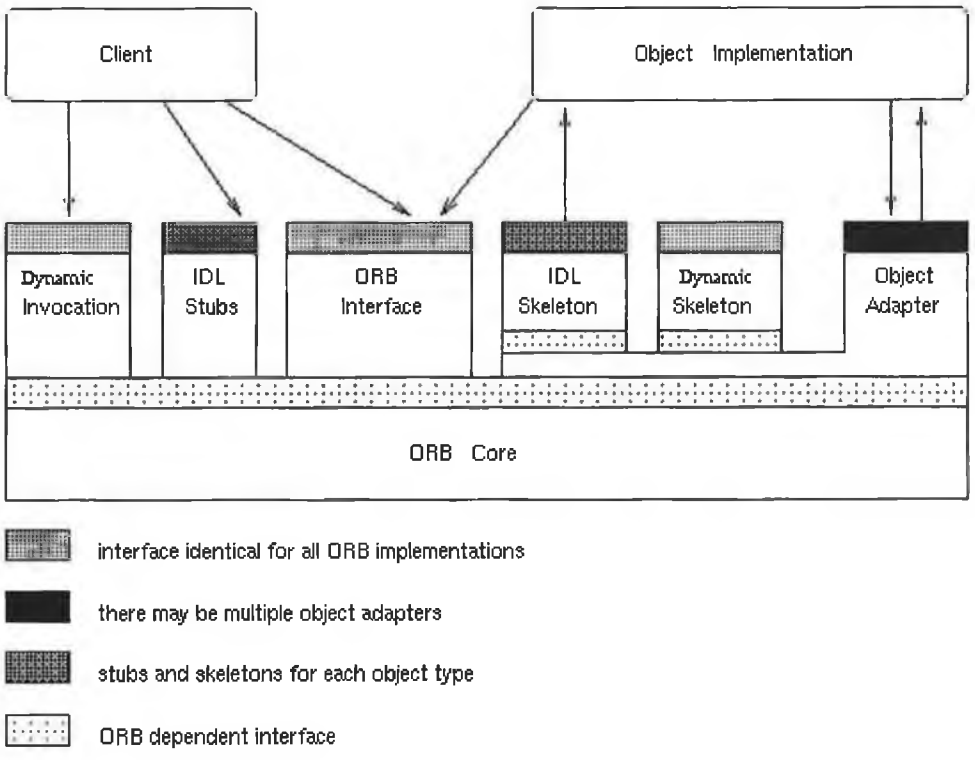


Figure 3.4 ORB Architecture

The basic functionality provided by the ORB consists of passing the requests from clients to the object implementations on which they are invoked. In order to make a request the client can communicate with the ORB Core through the IDL stub or through the Dynamic Invocation Interface (DII). The stub represents the mapping between the language of implementation of the client and the ORB core. Thus the client can be written in any language as long as the implementation of the ORB supports this mapping. The ORB Core then transfers the request to the object implementation which receives the request as an up-call through either an IDL skeleton, or a dynamic skeleton. Figure 3.4 shows the main components of the ORB architecture and their interconnections.

3.4.3.3 Overview of Architectural Components

The Object Adapter (OA) effects the communication between the object implementation and the ORB core. The OA handles services such as generation and interpretation of object references, method invocation, security of interactions, object and implementation activation and deactivation, mapping references corresponding to object implementations and registration of implementations. It is expected that there will be many different special-purpose object adapters to fulfil the needs of specific systems i.e. databases.

OMG specifies four policies in which the OA may handle object implementation activation:

- Shared Server Policy: multiple objects may be implemented in the same program.
- Unshared Server Policy.
- Server-per-Method Policy: a new server is started each time a request is received.
- Persistent Server Policy.

Only in the Persistent Server Policy is the object's implementation supposed to be constantly active (if it is not a system exception results). If a request is invoked under any other policy the object will be activated by the OA in the policy specific way. In order to be able to do that, the OA needs to have access to information about the object's location and operating environment. The database containing this information is called the Implementation Repository and is a standard component of the CORBA architecture. The information is obtained from there by the OA at object activation. The Implementation Repository may also contain other information pertaining to the implementation of servers, such as debugging, version and administrative information.

The interfaces to objects can be specified in two ways: either in OMG IDL, or they can be added to the Interface Repository, another component of the architecture. The Dynamic Invocation Interface (DII) allows the client to specify requests to objects whose definition and interface are unknown at the client's compile time. In order to use DII the client has to compose a request (in a way common to all ORBs) including the object reference, the operation and a list of parameters. These specifications - of objects and services they provide - are retrieved from the Interface Repository, a database which provides persistent storage of object interface definitions. The Interface Repository also contains information about types of parameters, certain debugging information, etc.

A server side analogue to DII is the Dynamic Skeleton Interface (DSI); with the use of this interface the operation is no longer accessed through an operation-specific skeleton, generated from an IDL interface specification, instead it is reached through an interface that provides access to the operation name and parameters (as in DII above the information can be retrieved from the Interface Repository). Thus DSI is a way to deliver requests from the ORB to an object implementation that does not have compile-time knowledge of the object it is implementing. Although it seems at the first glance that this situation doesn't happen very often, in reality DSI is an answer to interactive software development tools based on interpreters and debuggers. It can also be used to provide inter-ORB interoperability which will be discussed in the next section.

3.4.3.4 Interoperability

There are many different ORB products currently available; this diversity is very wholesome since it allows the vendors to gear their products towards the specific needs of their operational environment. It also creates the need for different ORBs to inter-operate. Furthermore, there are distributed and/or client/server systems which are not CORBA-compliant and there is a growing need for providing interoperability between those systems and CORBA. In order to answer those needs OMG has formulated the ORB interoperability architecture.

Implementational differences are not the only barriers that may separate objects. Other reasons might include strict enforcement of security, or providing a protected testing environment for a product under development. In order to provide a fully inter-operable environment all those differences have to be taken into account. This is why CORBA

introduces the higher-level concept of a domain, which roughly denotes a set of objects which for some reason, be it implementational or administrative, are separated from all other objects. Thus, objects from different domains need some bridging mechanism (mapping between domains) in order to interact. This bridging mechanism should be flexible enough to accommodate both the scenarios where very little or no translation is needed (as in crossing different administrative domains within the same ORB). The interoperability approaches can be most generally divided into *immediate* and *mediated* bridging. With mediated bridging interacting elements of one domain are transformed at the boundary of each domain between the internal form specific to this domain and some other form mutually agreed on by the domains. This common form could be either standard (specified by OMG, for example IIOP), or a private agreement between the two parties. With immediate bridging elements of interaction are transformed directly between the internal form of one domain and the other. The second solution has potential to be much faster, but is the less general one; it should be therefore possible to use both. If the mediation is internal to one execution environment (for example TCP/IP) it is known as a "full bridge", otherwise if the execution environment of one ORB is different from the common protocol we say that each ORB is a "half bridge".

Bridges can be implemented either internally to an ORB (say just crossing administrative boundaries), or in the layers above it. If they are implemented within an ORB they are called *in-line* bridges, otherwise they are called *request-level* bridges. The in-line bridges can be implemented through either requiring that the ORB provide certain additional services or through introducing additional stub and skeleton code. Interacting through the request-level bridges goes roughly like this: the client ORB "pretends" that the bridge and the server ORB are parts of the object implementation and issues a request to this object through the DSI (DSI does not need to know the specification of its object at compile time). The DSI, in co-operation with the bridge, translates the request to a form which will be understood by the server ORB and invokes it through DII of the server ORB; the results (if any) are passed back via a similar route. In order to perform its function the bridge has to know something about the object; thus it either needs to have access to the Interface Repository, or be only an interface specific bridge with the applicable interface specifications "hardwired" into it.

In order to make bridges possible it is necessary to specify some kind of standard transfer syntax. This function is fulfilled by General Inter-ORB Protocol (GIOP) defined by the OMG; it has been specifically defined to meet the needs of ORB-to-ORB interaction and is designed to work over any transport protocol that meets a minimal set of assumptions. Of

course, versions of GIOP implemented using different transports will not necessarily be directly compatible; however their interaction will be made much more efficient.

Apart from defining the general transfer syntax, OMG also specified how it is going to be implemented using the TCP/IP transport and thus defined the Internet Inter-ORB Protocol (IIOP). In order to illustrate the relationship between GIOP and IIOP, OMG points out that it is the same as between IDL and its concrete mapping, for example its C++ mapping. IIOP is designed to provide "out of the box" interoperability with other compatible ORBs (TCP/IP being the most popular vendor-independent transport layer). Further, IIOP can also be used as an intermediate layer between half-bridges and in addition to its interoperability functions, vendors can use it for internal ORB messaging (although this is not required, and is only a side-effect of its definition). The specification also makes provision for a set of Environment-Specific Inter-ORB Protocols (ESIOP). These protocols should be used for "out of the box" interoperability wherever implementations using their transport are popular.

3.5 Interface Definition Language

Interface Definition Language (IDL) is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. In distributed object technology, it is important that new objects be able to be sent to any platform environment and discover how to run in that environment. An Object Request Broker (ORB) is an example of a program that would use an interface definition language to "broker" communication between one object program and another one.

An interface definition language works by requiring that a program's interfaces be described in a stub or slight extension of the program that is compiled into it. A broker program uses the stubs in each program to allow them to communicate.

IDL defines the types of objects by specifying their interfaces. It is required that all OMG services be specified using a declarative language emphasising the separation of interface and implementation. It is programming language neutral and network neutral, and thus it is used as a means of describing data types. It provides a two-tier encapsulation system: data-types (basic and user-defined), and objects, which allows for sophisticated modelling of a distributed domain. The syntax of OMG IDL is derived from C++, removing the constructs

of an implementation language and adding a number of new keywords required to specify distributed systems.

To sum up, OMG IDL is used to statically define the interfaces to objects, to allow invocation of operations on objects with differing underlying implementations. From the IDL definitions, it is possible to map CORBA objects into particular programming languages or object systems.

3.6 Using SDL to develop CORBA object implementations

With the advent of CORBA to manage large, heterogeneous object systems, the need to be able to specify, verify, and test such systems is becoming increasingly important. It is not sufficient to be able to specify object behaviour using for example C++, because the behaviour of such systems is not easily verifiable. Instead, by using a formal description language such as SDL to define the object behaviour all the commonly used techniques for verification and validation, provided by SDT [SDT93], become available.

3.6.1 Using CORBA and SDL

There are basically two ways to use SDL with CORBA, depending on whether SDL is viewed as a specification or design language:

- The *SDL-oriented approach*, where a CORBA platform is used as execution system for SDL processes.
- The *CORBA-oriented approach*, where SDL is supported as the implementation language for the definition of behaviour and treated in the same way as the already supported implementation language C++.

The SDL-oriented approach

The SDL-oriented approach is primarily used when SDL is viewed as a specification language. Once the specification of the entire distributed system has been performed in SDL, it should be partitioned into arbitrarily small subsystems that are then (optionally) implemented on different machines and communicate with each other using CORBA. The

smallest unit of partitioning should be at the process level. In this case, IDL descriptions are automatically generated from the SDL system; it might, however, be necessary to limit the available SDL concepts when using this approach.

The CORBA-oriented approach

The CORBA-oriented approach, on the other hand, is primarily used when SDL is viewed as an implementation language, and a given IDL description should be designed using SDL. That IDL description is used to generate a stub (skeleton) in SDL, to which behaviour is then added. The most important parts of the development process when using the CORBA-oriented approach are shown in Figure 3.5.

The activities, which can be clearly distinguished when using this approach, are:

1. Convert an IDL description to an SDL stub.
2. Define the behaviour of the SDL system using the generated SDL stub as the starting point.
3. Generate the C/C++ code that is used to implement the SDL application.
4. Generate the C++ code that is needed by the ORB.

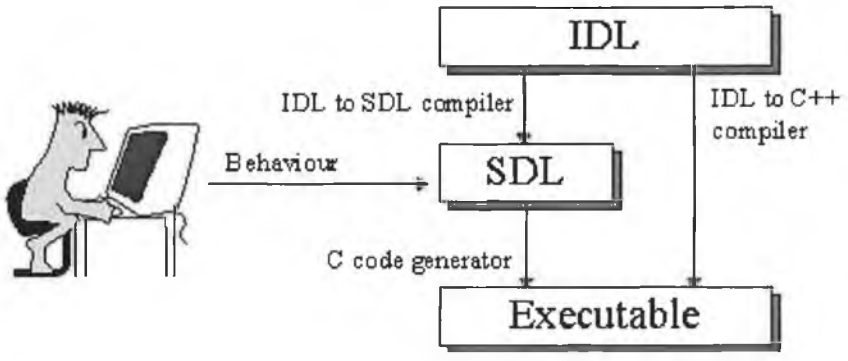


Figure 3.5 Activities of the CORBA-Oriented Approach

The activities which concern the developer are the first two; the user must be aware that the SDL system is a CORBA server (i.e. an object implementation that can also act as a client by requesting services from other object implementations). By using the mapping rules, described in Section 3.6.2 below, it is possible create a tool that automatically converts an IDL description to an SDL stub. The latter two activities described above are also performed automatically.

3.6.2 Mapping IDL to SDL

In order to create the implementation language stub it is necessary to provide mapping rules from IDL to SDL. The mapping rules may be summarised as follows:

- A module provides a namescope and a mechanism to group interfaces. As such it is mapped to a block type, where nested modules become nested block types.
- An interface is mapped to a process type. As the interface contains the attributes and operations that are available on an object, the corresponding process type contains the appropriate SDL definitions of these. Object references are mapped to PId values.
- An operation describes a service that is offered by an object, and it can be defined either as synchronous, where the client is blocked while the request is being handled by a server, or as asynchronous (using the keyword *one-way*), where the client making the request simply continues executing. In the former case, the operation is mapped to a remote procedure, while in the latter it is mapped to a signal. Raises and context expressions are still subject to be mapped, as are exceptions.
- An attribute is mapped to a declaration of a variable together with two remote procedures that are used to get and set the value of the variable. If the attribute is defined as read-only, the set operation is omitted.
- Interface inheritance must be flattened in SDL, where both operations and attributes defined in a base type have to be duplicated in a derived type, i.e., SDL inheritance cannot be used.
- Constants are mapped to synonyms.

The mapping of basic types is shown in Table 3.2.

IDL Type	SDL Type	syntype of
long	CORBA_long	Integer
short	CORBA_short	Integer
unsigned long	CORBA_unsigned_long	Integer
unsigned short	CORBA_unsigned_short	Integer
double	CORBA_double	Real
float	CORBA_float	Real
char	CORBA_char	Character

boolean	CORBA_boolean	Boolean
octet	CORBA_octet	Octet

Table 3.2 Mapping Basic Types

Note that predefined types have the prefix 'CORBA_' in SDL. The type *any* currently have no suitable mapping in SDL. Constructed types are mapped as follows:

- An enum is mapped to a newtype with the appropriate literals.
- A struct is mapped to a newtype with the corresponding struct.
- A union is mapped to a newtype with a corresponding struct, where the first member of the struct is a tag matching the union's switch type.

The template types are mapped as follows:

- A sequence is mapped to the generator string.
- A string is mapped to the type CORBA_string, which is a syntype of Charstring.

The complex declarators are handled as follows:

- An array is either mapped to the generator array, together with an additional type to define its index range, or as the generator CORBA_array, which is defined to have a limited index range corresponding to that of the original IDL array.

SDL System Structure

An SDL system based on the above mapping rules has a particular architecture; one package is used to define interface concepts, such as types, signals and remote procedures, whereas another is used to contain the structural information that can be derived from the IDL description. Other SDL clients that want to access services from this object implementation can reuse the interface package. Consider the SDL system shown in Figure 3.6.

In this example, an object implementation in SDL is created from the IDL file named *objects.idl*. The stub that is created will contain one package named *objects_interface*, and one package named *objects_definition*; this latter package also contains a system type named *objects_system*. The object implementation should make use of services that are defined in the IDL file *other.idl*. In order for the system implementing objects to access these services, the IDL description of other is also converted to SDL, but only the interface package (*other_interface*) needs to be used.

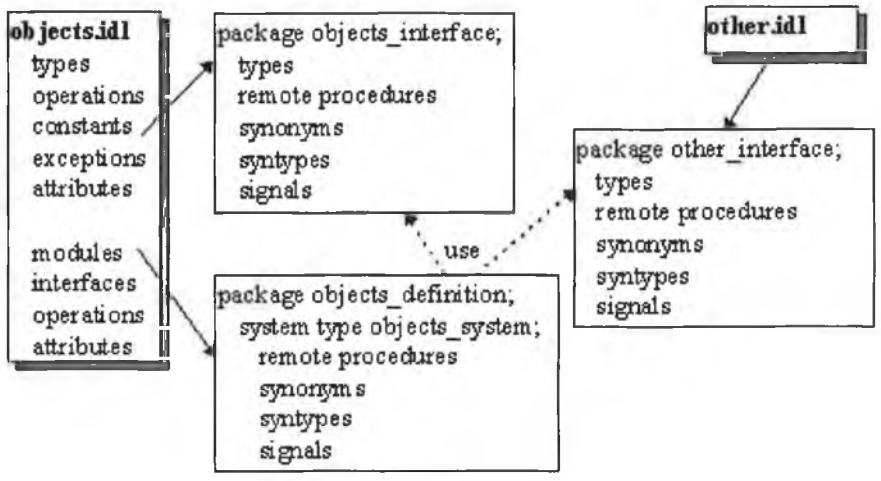


Figure 3.6 Schematic View of the SDL System Structure

3.6.3 Mapping Object Models to IDL

The *Paste As* functionality of SDT allows a developer to copy an object in one model and paste it as IDL while creating an implementation link (implink) between the two. Implementation links are used to maintain a relationship between objects in different models. The transformation rules that are considered here for that are very simple and straightforward. The strength of this approach is that for each object that is pasted, an implink is created between the copied class and the IDL entity, thereby facilitating trace mechanisms for example.

Classes: An object model class can be mapped to either a module or an interface. In the former case, the resulting module is always empty, even if the class contained both attributes and operations. In the latter case, however, all attributes and operations are mirrored using IDL syntax as far as possible. This latter mapping is shown in Figure 3.7.

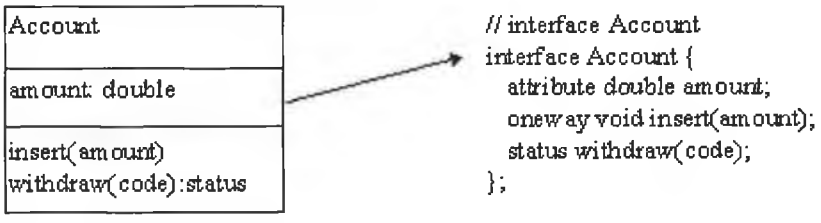


Figure 3.7 Mapping an Object Model Class to an IDL Interface

The comment included in the mapping is used to contain the optional implink that is created between the class and the interface. It is not possible to express all information that is needed in the IDL description simply by mapping a class like this. Usually some manual additions must be made, such as type definitions, constants, and exceptions, but also simple matters like defining for each operation parameter whether it is an *in*, *inout* or *out* parameter.

Furthermore, it might be necessary to define an attribute as *readonly*, or whether an operation is *oneway* or not. All operations are considered asynchronous by default, unless it can be determined that it should be synchronous. This is done by either defining a return type for the operation, or by inserting {*sync*} or {*async*} after it.

Inheritance: The mapping of object model class inheritance is only applicable when an object model class is mapped to an interface, and in that case the class inheritance is mapped as interface inheritance, as is shown in Figure 3.8.

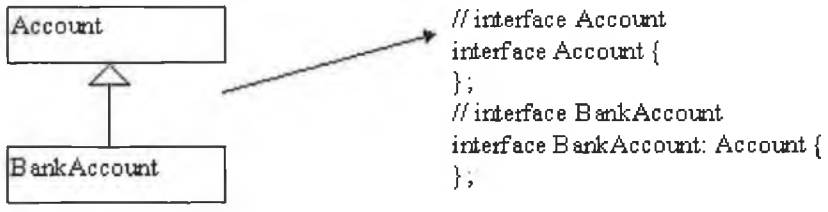


Figure 3.8 Mapping Object Model Class Inheritance to IDL Inheritance

Aggregation: When mapping aggregations, all object model classes that are not leaves are mapped as modules. However, the leaves themselves may be either modules or interfaces, and it is the responsibility of the user to decide which. Aggregations are thus mapped to nested modules where the innermost layer may be either interfaces or modules. Figure 3.9 shows an example of aggregation mapping.

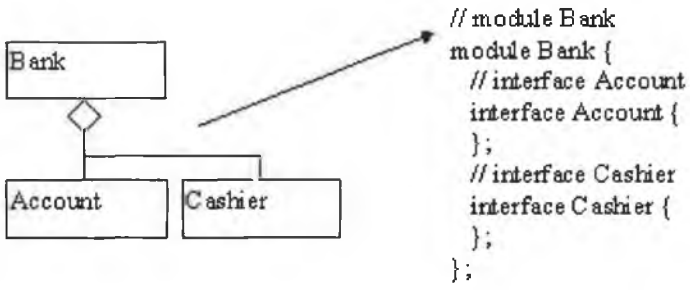


Figure 3.9 Mapping Object Model Class Aggregation to IDL

3.6.4 Implementing an SDL Application

In this section the implementation of the CORBA-oriented approach is discussed in more detail. The SDL application created is supposed to work as a client/server in a heterogeneous, distributed environment as only one of many such parts, i.e. the system has been decomposed into lesser parts that may or may not be implemented using different programming languages. CORBA is the glue between these parts.

3.6.4.1 The System Architecture

An SDL application can be used as both a server and a client. This is illustrated architecturally in Figure 3.10.

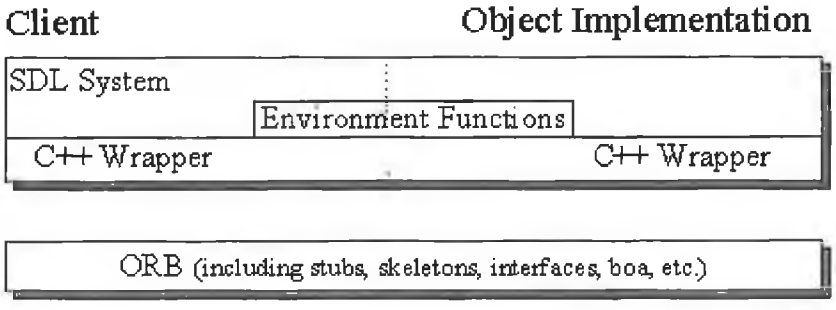


Figure 3.10 System Architecture of an SDL Application

An ordinary SDL application without the CORBA support only consists of the SDL system itself along with its environment functions that are used to connect the system to the outside world. In the CORBA-oriented approach, a wrapper is placed around the SDL system to imitate the behaviour of objects in a way that makes sense to the ORB. The C++ wrapper and its functionality is further discussed in Section 3.6.4.2.

Based on the development process depicted in Figure 3.5, the development of the object implementation side starts with an IDL description. The complete process using Orbix (the CORBA package from IONA) is described as follows :

1. The IDL description is used as the basis for an SDL stub, which is generated by an SDT specific IDL compiler. At the same time, a C++ wrapper for the object implementation side is also generated using the same IDL compiler.
2. The ORB also uses the IDL description to generate ORB specific C++ code, but has an ORB specific IDL compiler for this purpose.
3. A developer then provides the behaviour of the SDL system stub.
4. If any services from other servers are required (on the client side of the SDL application), their IDL descriptions should be converted to SDL using the SDT specific IDL compiler. Code is also generated for the C++ wrapper. As for the ORB, it uses its own IDL compiler to generate the necessary C++ code.
5. C code representing the designed SDL system is generated automatically using the SDT C Code Generator.

These steps are shown graphically in Figure 3.11.

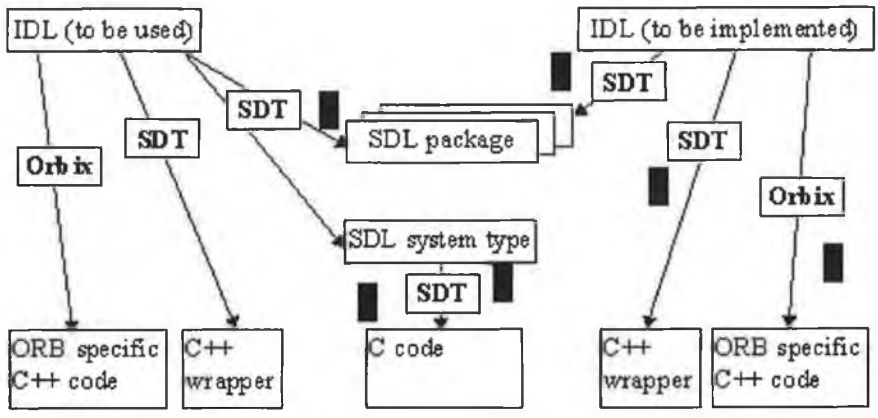


Figure 3.11 Generating Code for the SDL Application

The environment functions are predefined and are always the same regardless of the SDL (CORBA) application; all system specific information is placed in the wrapper. When the C and C++ code has been generated as above it is compiled and linked with a set of pre-compiled libraries (including an SDL kernel providing runtime support of the SDL system and ORB specific libraries) to form an executable SDL application.

3.6.4.2 Wrapping the SDL system

To get a functioning SDL application, the behaviour of the SDL system wrapper is especially important. The wrapper can be said to consist of two parts: the environment functions and the class definitions that are generated from the IDL descriptions.

Scheduling

Since the SDL system is capable of acting as a client, it has its own scheduling mechanism. Requests made to other servers are sent through an environment function to the C++ wrapper, where the request is processed and then passed on through the ORB to the intended server. At regular intervals, the scheduler calls another environment function to check whether there are events pending from the ORB. Should such an event be present, it is processed, and then sent to the appropriate process instance. The entire SDL application is scheduled within a single UNIX process, and requests that are made internally in the SDL system are not managed by CORBA. Contrast this with the SDL-oriented approach, where all requests would be sent through CORBA, possibly from multiple UNIX processes.

Environment Functions

The environment functions are responsible for managing the communication between the ORB and the SDL system. Two of these environment functions are responsible for passing requests to and from the SDL system. Another environment function is used to initialise the SDL system and its environment. In this case, it is particularly used to initialise the communication with the ORB. The environment functions use buffers to store all requests before they are treated, both for incoming and outgoing requests.

Objects versus Process Instances

When the ORB communicates with the object implementation, it only sees the C++ wrapper and the objects that are defined within it; the SDL system itself is completely hidden.

The SDL system, on the other hand, controls the objects that are present in the C++ wrapper by mirroring each process instance as an object. Whenever a new process instance is created in SDL a new object representing that process instance is created in the wrapper. When a process instance is terminated, the object is also removed.

An object that receives a requests passes it along to its corresponding process instance (it first has to convert the request into a suitable format, i.e. a signal or remote procedure). In SDL, all object references are represented using PId values. A C++ client, however, would still access the process instance of an SDL object implementation using its ordinary object references.

Multithreading

In order to handle requests correctly, it is necessary to make use of a multithreaded ORB. On the object implementation side, each new request from the ORB must be treated in a thread of its own. The reason for this is that a synchronous operation is mapped to a remote procedure in SDL, which in turn is implemented as the sending of two asynchronous signals (call and reply). When the request is received, a call signal is sent into the SDL system. While the operation is waiting for a reply, the SDL system must be able to execute other requests, which is not possible unless each request is executed in its own thread, while the SDL system is scheduled in a main thread.

On the client side, the situation is very similar. While waiting for a reply to a request made to the ORB, the SDL system must be able to continue to execute, which means that all such server requests must also be executed in separate threads.

The SDL system thus executes in one thread, and each request that is received by or sent from the SDL system results in a new, detached thread which disappears once the request has been handled.

The ORB is responsible for creating the appropriate threads for requests to the SDL system, while the C++ wrapper is responsible for creating threads for requests aimed at other servers. The C++ wrapper must also provide a sufficiently multithread-safe environment.

The C++ implementation classes

As part of the ORB specific code, a set of IDL C++ classes are generated which represent the IDL interfaces. It is then up to the developer to provide C++ implementation classes that implement these IDL C++ classes. The ORB provides mechanisms for connecting the IDL C++ classes and the C++ implementation classes to each other. Each IDL C++ class must have at least one corresponding C++ implementation class.

In the C++ wrapper, the C++ implementation classes are the most important part, as they provide the 'behaviour' of the SDL system. A request that is made to an object is passed to the appropriate process instance after having been processed. This processing is specific depending on whether the operation is asynchronous or synchronous. Due to the increased complexity when dealing with synchronous operations it is considerably easier to manage asynchronous operations.

For an asynchronous operation, the following steps have to be performed by an object:

1. Allocate memory for the signal.
2. Convert the parameters of the operation to SDL signal parameters.
3. Send the signal to the process instance corresponding to the current object.
4. Return, i.e. exit the request thread.

For a synchronous operation, on the other hand, some additional steps are necessary:

1. Allocate memory for the remote procedure call.
2. Convert the parameters of the operation to SDL remote procedure call parameters.
3. Send the remote procedure call to the process instance corresponding to the current object.
4. Wait for a reply, i.e. block the request thread until the appropriate remote procedure reply is received. It is necessary to pass information about the current thread's identity in the call/reply signals to ensure that the appropriate thread receives the correct reply (since the order in which replies are received is not guaranteed to be the same as the order in which they were sent).
5. Convert the SDL remote procedure reply parameters to C++ parameters.
6. Release the memory held by the reply.
7. Return, i.e. exit the request thread, and pass the obtained data back to the client.

Locating a server object

One particular problem that must be addressed is how a server object is located. In SDL a request can be made either to a specific PId value or without specifying a particular receiver. In the first case, the object reference of the receiver is already known, and there is no

problem. In the second case, however, an appropriate receiver must first be located. There are two approaches to this problem:

A *bind* concept can be introduced in SDL which would be responsible for finding a server implementing the required request. However, this solution does not remove the problem with requests that are made without specifying a receiver.

When the C++ wrapper receives a request with no apparent receiver, an ORB specific bind command, to find an appropriate server object, can be performed. Once the request has been performed this object reference can be stored for subsequent requests of the same kind to reuse, or released, thereby requiring a new bind command for each new request.

3.7 Conclusions

This chapter presented an overview of the tools and technologies required to complete the project. These included the programming language SDL, in which the IN system was designed and the tool used to develop it, SDT. As the project is also concerned with the extendibility and evolution of IN towards TINA it was also necessary to explain the concepts of CORBA and IDL, which facilitate the operation of distributed applications.

It has been shown that it is useful to use SDL in conjunction with CORBA. By combining the two, SDL becomes an important language to consider when dealing with distributed systems. SDL can be supported as an implementation language for the definition of behaviour of a distributed system. This is useful given the many advantages of SDL and the package SDT. The behaviour of these CORBA objects can be tested using SDT Simulations. Also Message Sequencing Charts (MSCs) can be produced. With SDL systems the behaviour and architecture of what they describe can be both very easily understood and modified. By using a CORBA platform as the execution system for interacting SDL processes the advantages of distribution can be gained. Different components of a large heterogeneous system can be located on different platforms and can interact by sending messages to each other via the ORB. They need know nothing of the other objects implementation other than their interfaces.

Chapter 4 IMPLEMENTATION OF THE IN CONCEPTUAL MODEL

4.1 Introduction

This chapter describes the development in SDL, using the SDT package, of an Intelligent Network architecture based on the ITU-T Standards [Q.1200]. The system interacts with an Excel switching platform [Excel] via the CCAF processes. Thus users attached to the switch may interact with the architecture in the establishment of basic calls or the running of IN services, two of which are provided with the IN. Users communicate with the DFP layer of the INCM via the Excel switching platform, as shown in Figure 4.1 below. To establish a basic call they go offhook and dial the appropriate digits. The CCF (Call Control Function) connects the user to their terminating party. If, after going offhook, the user wishes to invoke an IN service they dial the appropriate digits and control is passed up to the SCF (Service Control Function) and the higher layers which deal with IN service control.

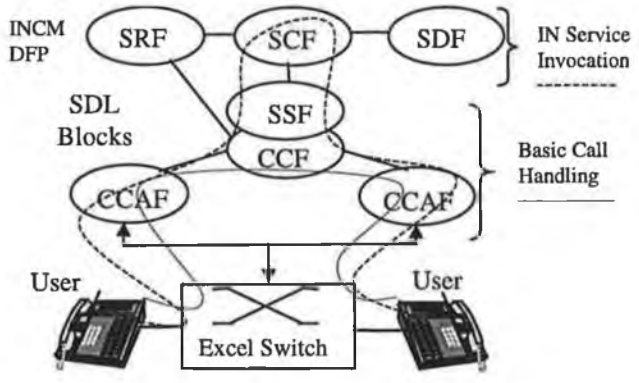


Figure 4.1 IN System Architecture

The SDL representation of the IN is based on the middle two planes of the Intelligent Network Conceptual Model (INCM), namely, the Global Functional Plane (GFP) and the Distributed Functional Plane (DFP). The mapping of the INCM entities to SDL constructs is outlined in the next section of this Chapter. The architecture of the IN system is then discussed with each SDL block being considered individually. The organisational and communicational aspects of each block are also discussed.

Models of the INCM have been developed before but having the IN interact with a switching platform, to which users are connected, allows us to see the actual operation of a real time IN system. The Excel switch will be discussed in detail in this chapter as will the method of communication between it and the IN system.

4.2 Overview of the System

This section discusses the objectives in designing the IN system and how the INCM entities are mapped to SDL.

4.2.1 Objectives

The aim of the project was to build an executable Intelligent Network based on ITU-T's IN Conceptual Model. Two call control services, *Ringback* and *Group Call Pick Up*, were to be realised on the system using the ITU-T SIB methodology. The IN system must be able to communicate with an Excel CSN Programmable Switching Platform on which the services are deployed. The IN system must be capable of handling simultaneous multiple users and multiple services and also be capable of incorporating further services easily, with the addition of new SIBs or the rearrangement of the order of execution of the existing ones.

4.2.2 Mapping of INCM entities to SDL

The SDL system developed represents the middle two planes of the INCM, namely the Global Functional Plane (GFP) and the Distributed Functional Plane (DFP). By definition an SDL system and its environment are conceived of as a structure of blocks connected by channels. It follows that both the GFP and DFP should be represented by a block structure. These two blocks are connected to each other and to the environment via channels.

GFP Entities

In the GFP the entities to be represented in SDL include the SIBs, the BCP and the GSL. Seven different SIB types are used in various combinations to realise the two services; *Ringback* and *Group Call Pickup*. Each of these SIBs may be invoked any number of times

and each time with different data parameters. Thus a mechanism is needed to be able to dynamically create an instance of a SIB type and assign it data parameters. The process concept was therefore considered to be the most suitable way to represent each of the SIB types. A manager process is therefore necessary to dynamically create SIB process instances. Since the GSL is used to control the order in which SIBs are invoked it should be represented by this manager process concept also. This GSL process has the ability to create a process instance of any of the SIB types and dynamically configure the parameters. A process represents the BCP since it is also a SIB. The BCP and GSL processes communicate via a signal route as do the GSL and each of the SIB processes. SIBs in the GFP are representations of functionality realised by FEAs in the DFP so it must be possible for a SIB to communicate with the relevant FEAs in the DFP. Each of the SIBs are connected via a signal route to the channel connecting the GFP and DFP. When a SIB instance is created it sends a signal to an FEA in the DFP. This causes a number of interactions between the FEAs in the DFP in order to realise the functionality of the SIB and the SIB process instance in the GFP receives a response from the DFP via the channel.

DFP Entities

Block structures in SDL can be nested in order to achieve more abstraction or clarity. The DFP block is subdivided into blocks, which contain processes in order to improve the readability of the system. In the DFP there are two levels of granularity, FEs and FEAs. The FEs are represented by SDL blocks while the FEAs are represented by SDL processes within. The FE blocks communicate with each other via channels. Two of the FE blocks, the SCF and CCFSSF, communicate with the GFP via the channel between the GFP and the DFP. When a chain of FEAs is executed to realise a SIB the SCF block must communicate with the GFP as the first and last FEAs in the chain always reside in the SCF. The first FEA receives the initial request from the SIB process in the GFP and the last FEA executed sends a response to the SIB process in the GFP. The BCP process in the GFP sends instructions to the SSF process in the block CCFSSF. In the CCFSSF block a single process each represents the originating Basic Call State Model (BCSM) and the terminating BCSM.

4.3 The SDL System

The operation of the IN system is described by the Message Sequence Charts (MSC) included in Appendix E.

4.3.1 Organisation

As shown in Figure 4.2, the INCM has two blocks representing the GFP and the DFP.

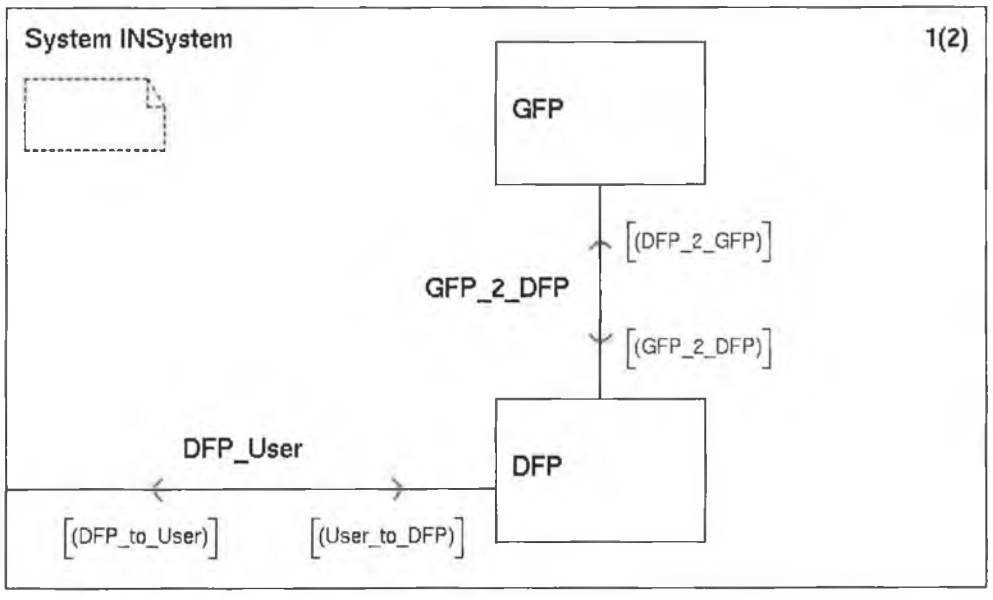


Figure 4.2 Overview of the IN Conceptual Model in SDL

4.3.2 Communication

The two blocks communicate via the GFP_DFP channel. Signal lists are used to reduce the amount of information in the diagram. The Excel switch resides in the environment, outside the SDL system. The interactions with the Excel switch are detailed in Section 4.8. A limitation with SDL is that it is not possible to separate different entities in the environment i.e. when a signal is sent it is only possible to define the environment to be the recipient and not a specific entity in the environment.

4.4 The DFP Block

The Distributed Functional Plane lies between the Global Functional Plane and the Physical Plane in the INCM.

4.4.1 Organisation

The Distributed Functional Plane is represented in SDL by the DFP block. The DFP block is partitioned into sub-blocks for each of the FEs which in turn contain the FEA processes. The naming convention for FEA processes is given in Section 2.8. The DFP block can be called a subsystem specification as it allows the designer to gain an overview by grouping a number of block specifications into higher level specifications.

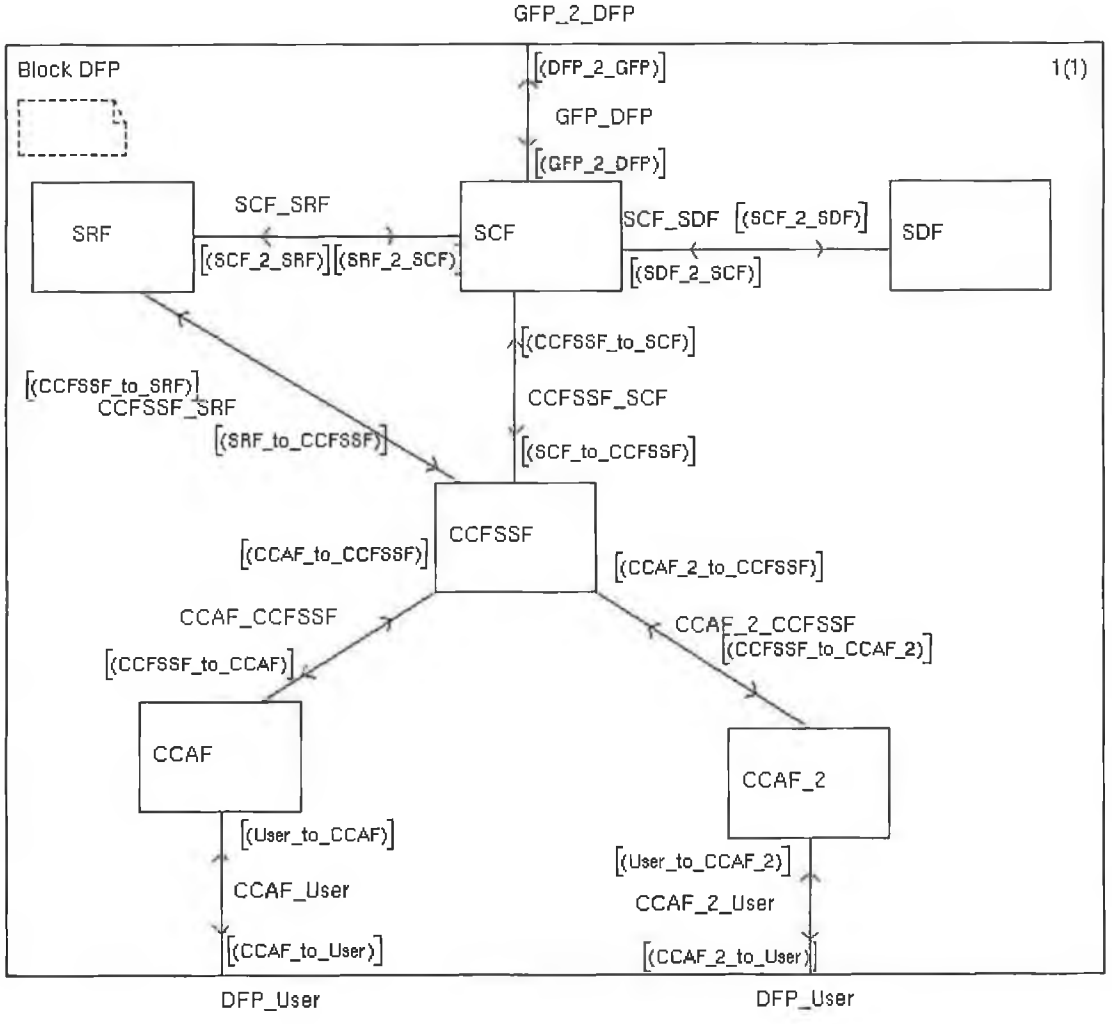


Figure 4.3 DFP Represented in SDL

The FEs and the communication paths between them correspond directly with the model of the DFP presented in Chapter 2. From Figure 4.3 it can be seen that an originating (CCAF) and a terminating (CCAF_2) Call Control Agent Function have been specified. This allows for the simulation of non-IN calls, i.e. the connection of a calling party attached to CCAF and a called party attached to CCAF_2. The CCFSSF block specifies the functionality for both the CCF and SSF.

4.4.2 Communication

Channels are used to allow communication between FEs and to the surrounding system. Signal lists are used to improve the readability of the diagram. The DFP communicates with the GFP via the channel GFP_DFP.

When a Detection Point is encountered in the Basic Call State Model (BCSM) the CCF sends the signal *InitialDP* to the GFP. Parameters carried in the signal indicate the type of the detection point. By sending signals to the SSF the Global Service Logic (GSL) in the GFP can influence how the call is completed. As can be seen from Figure 4.2, only the SCF is allowed to communicate with the SDF, via the channel SCF_SDF.

The SRF and the SCF communicate via the channel SRF_SCF. The SCF sends signals such as *Play_req_ind* to the SRF indicating that an announcement should be played to a user. The SRF responds to the SCF with the message *Completion_resp_conf* if the announcement is successful and no input is expected from the user.

The functionality of a SIB in GFP is realised by sending a query (e.g. *Screen_Query*, with parameters of the query being contained in the parameters of the signal) from the SIB to the relevant FEA in the DFP. On receiving the query the FEA will generate and send, if necessary, a query to other FEAs in order to process the query. The response to the query (e.g. *Screen_Reponse*) will be sent to the SIB in the GFP. Signals sent within the system are shown in Appendix D.

4.4.3 The SCF Block

The SCF block represents the Service Control Function. The SCF commands call control functions in the processing of IN provided and/or custom requests.

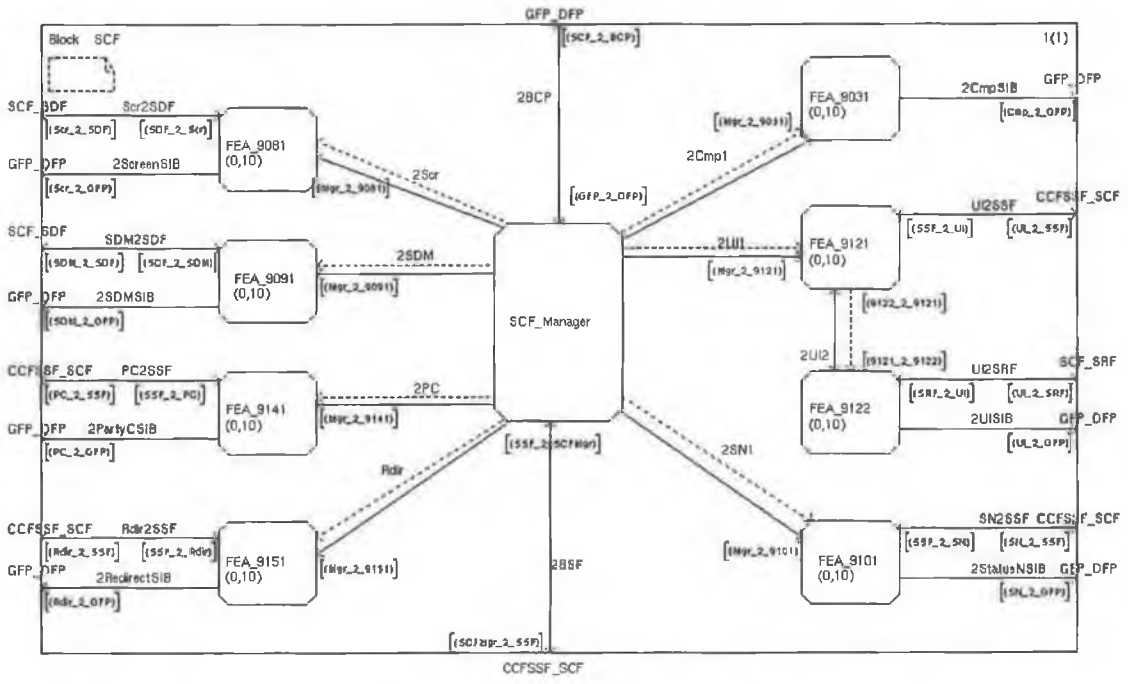


Figure 4.4 SCF represented in SDL

Organisation

The SCF block, shown in Figure 4.4, contains seven FEA processes, which are created by the SCF_Manager when needed. When a SIB runs a sequence of FEAs the first and last FEA to be run is always in the SCF.

Communication

The FEAs are connected to the edge of the block via signal routes. Some FEAs act as interfaces to the GFP, i.e. they receive and send signals to and from SIB processes in the GFP. These FEA processes e.g. process FEA_9081, can communicate with the GFP via the channel GFP_DFP. The rest of the FEAs communicate with other FEAs in the SDF and SRF blocks.

4.4.4 The SDF Block

The Service Data Function is represented in SDL by the SDF block. The SDF contains customer and network data for real time access by the SCF in the execution of IN services.

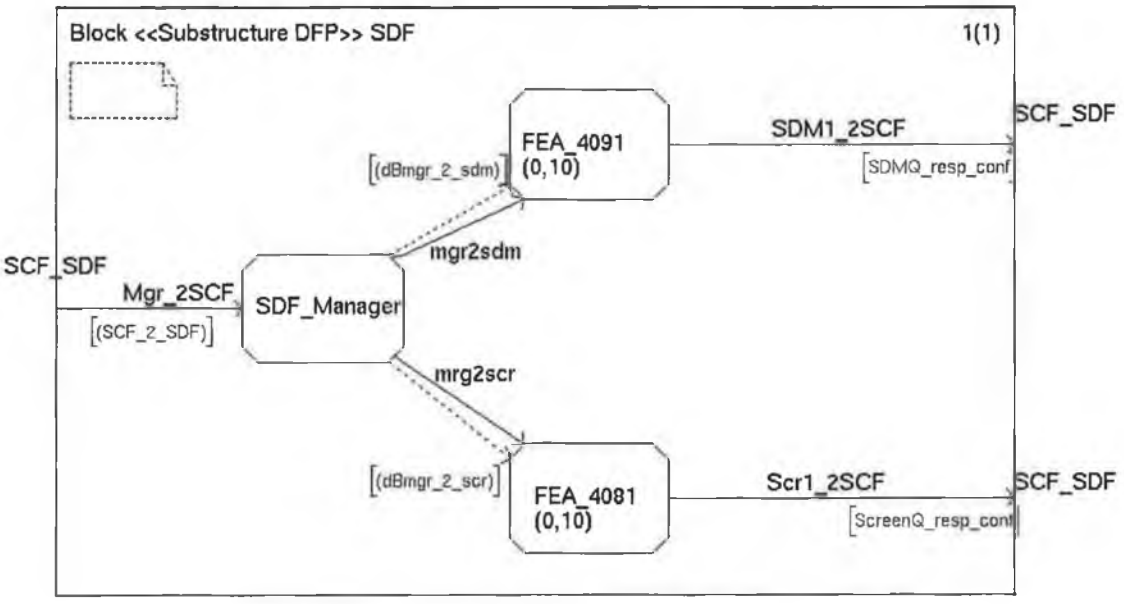


Figure 4.5 SDF Represented in SDL

Organisation

The organisation of the SDF is shown in Figure 4.5. Two SIBs in the GFP make use of FEAs in the SDF: the Screen SIB and the Service Data Management (SDM) SIB. The screen SIB makes use of the process FEA_4081 that performs the screening of the database needed for the screen SIB. The SDM SIB makes use of the process FEA_4091 that performs the retrieval operations.

Communication

Each of these processes communicates with FEAs in the SCF. An SDF request follows the following basic format:

- An FEA process in the SCF generates a database query and sends it to the relevant process in the SDF.
- The FEA process in the SDF receives the signal, performs the database operation and returns the result to another FEA in the SCF (not the one that generated the query).
- The FEA process in the SCF then returns the result to the SIB in the GFP.

4.4.5 The SRF Block

The Specialised Resource Function is represented in SDL by the SRF block. The SRF provides the specialised resources required for the execution of IN provided services i.e. digit receivers and playing announcements.

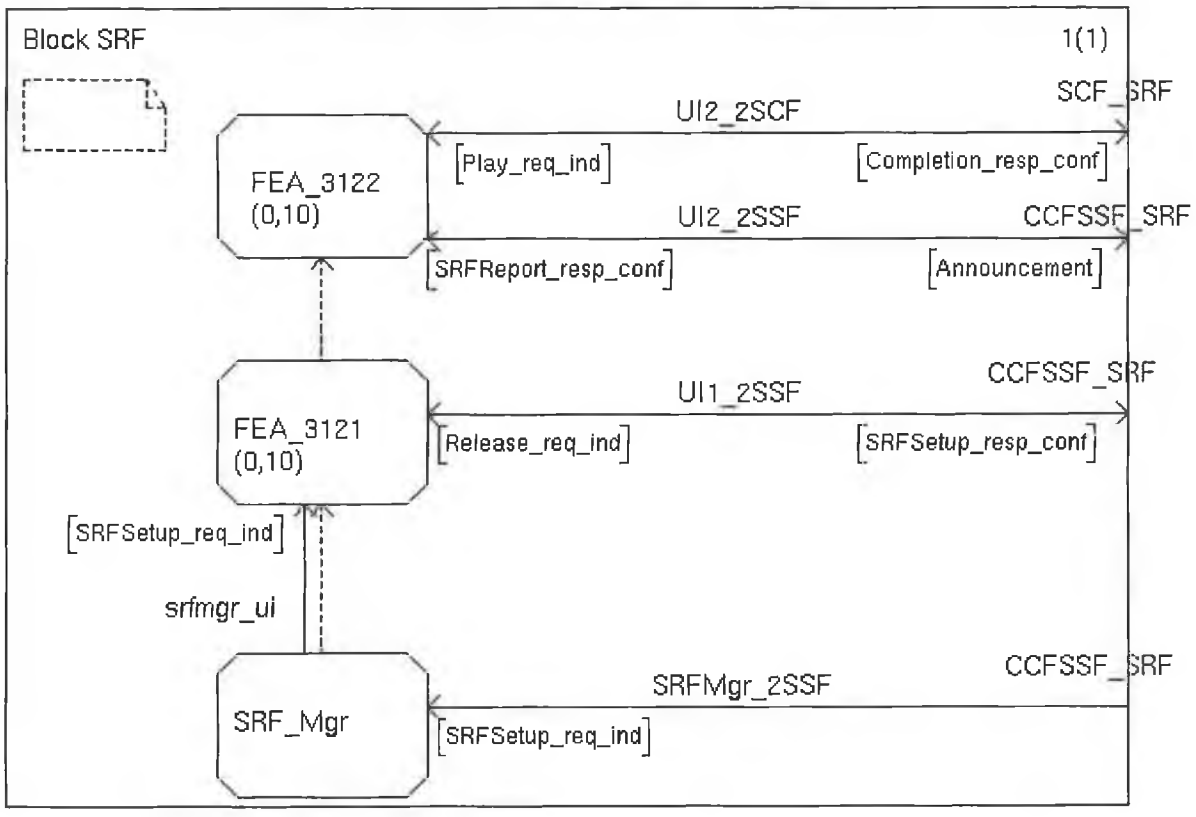


Figure 4.6 SRF represented in SDL

Organisation

The SRF block is illustrated in Figure 4.6. There are three processes in the block:

- Process FEA_3121 accepts an SRF set-up request from the SSF, finds an available resource and returns the resource identifier to the SSF.
- Process FEA_3122 plays announcements to the user and collects user digits
- Process SRF_Mgr manages the creation and deletion of processes in the block.

Communication

Process FEA_3121 communicates with the SSF via the signal route UI1_2SSF. This signal route carries SRF set-up messages. Announcements to be played to the user are sent from the SCF to the process FEA_3122 via the signal route UI2_2SCF. These announcements are then relayed to the SSF via the signal route UI2_2SSF.

4.4.6 The CCF/SSF Block

The block CCFSSF represents the CCF and SSF combined. The CCF is the Call Control Function in the network that provides call connection, processing and control. The SSF is the Service Switching Function which, associated with the CCF, provides the set of functions required for interaction between the CCF and SCF.

Organisation

The organisation of the block CCFSSF is shown in Figure 4.7. The Orig_BCSM process represents the Originating Basic Call State Model (BCSM). This process contains the functionality to realise the call model presented in Chapter 2. The user interacts with the process Orig_BCSM via the block CCAF. The terminating BCSM (represented by the process Term_BCSM) handles the call completion aspects of a call. The process SSF contains the functionality of the switch.

Communication

The process Orig_BCSM communicates with the CCAF via the signal route CCAF_Orig_BCSM. During the call set-up phase all information from the user is sent to the Orig_BCSM process. If the SSF decides that upon reception of a detection point (DP), it needs to communicate with the intelligent service layers above then it sends a request to the service logic. This is done by sending a DP to the Basic Call Process in the GFP where it is interpreted as a POI for the launch of a SIB chain. The service logic may instruct the SSF to set-up a connection between the user and the SRF. In this case all information from the user is routed transparently through the SSF to the SRF.

The SSF communicates with the BCP process in the GFP via the signal route SSF_SCF. The SSF is connected directly to the SCF, CCAF, CCAF2 and SRF processes.

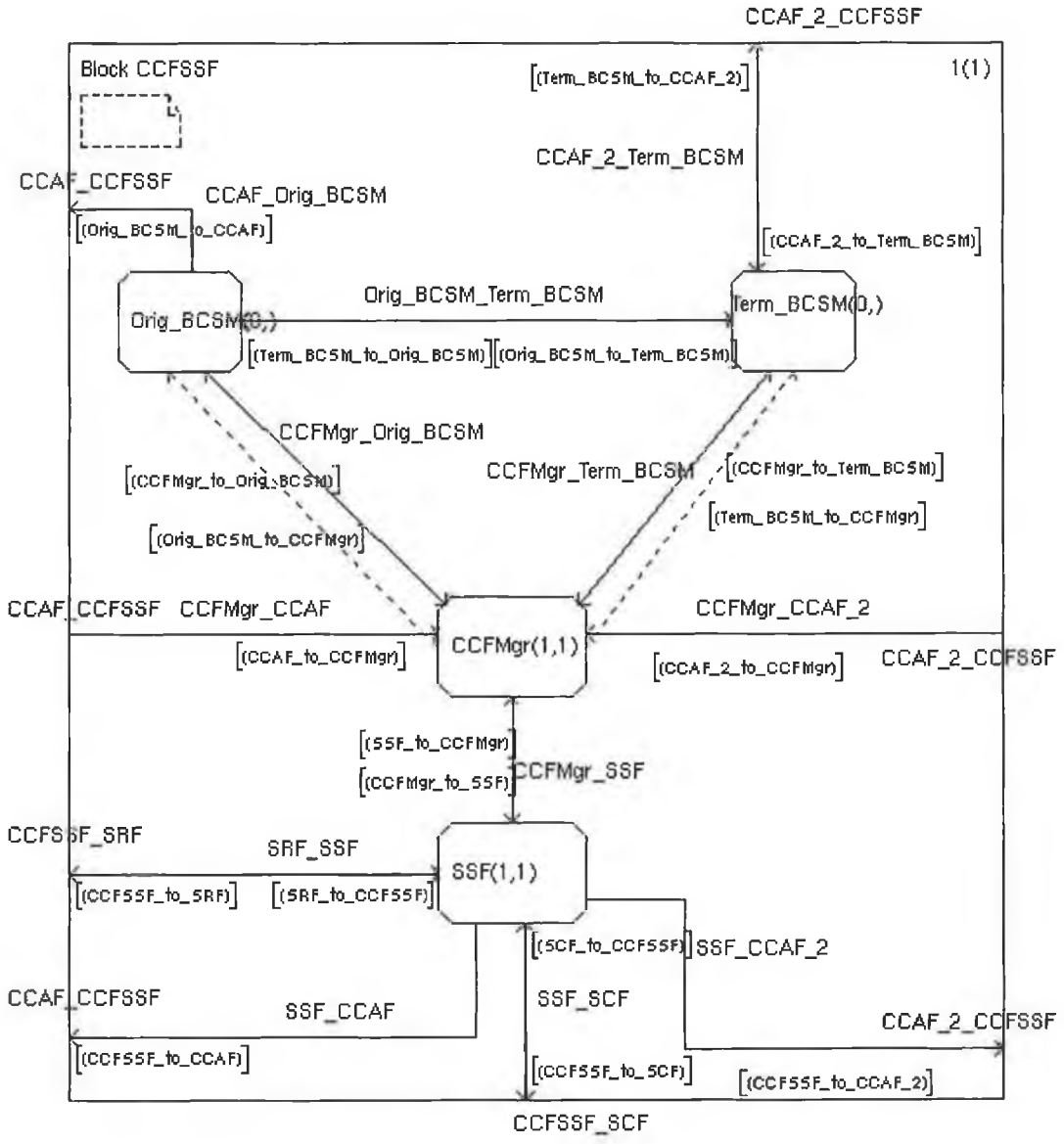


Figure 4.7 CCF and SSF represented in SDL

4.4.7 The CCAF Block

The Call Control Agent Functions (CCAFs) provide access to users via the Excel Switch. The originating Call Control Agent Function and a terminating CCAF are each represented by a single block; CCAF and CCAF2. The CCAF process interacts with the *calling* party of a call while the CCAF_2 process interacts with the *called* party.

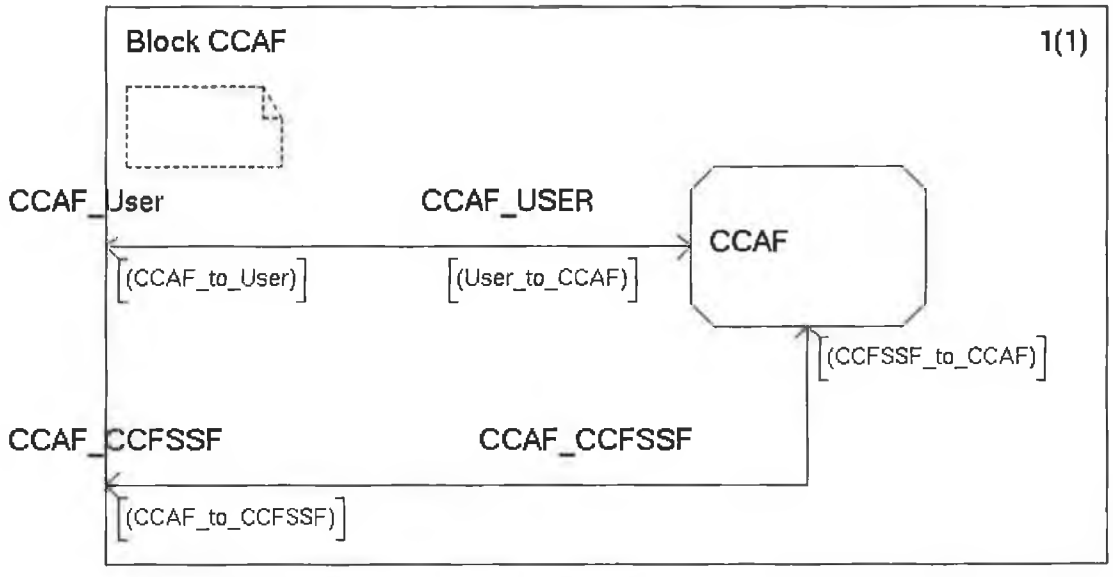


Figure 4.8 CCAF represented in SDL

Organisation

The block CCAF is illustrated in Figure 4.8. As the block CCAF2 is very similar to block CCAF so it is not discussed any further. The block contains only one process.

Communication

The process CCAF communicates with the user via the signal route CCAF_USER. All interaction between the user and the Intelligent Network passes through the Excel switch and the process CCAF. The interaction between the Process CCAF and the Excel Switch will be discussed in more detail in Section 4.8. The process CCAF sends any user input to the block CCFSSF via the signal route CCAF_SSF. Similarly any announcements to be played to the user are received from the block CCFSSF and passed on to the user

4.5 The GFP Block

4.5.1 Organisation

The organisation of the GFP block is shown in Figure 4.9.

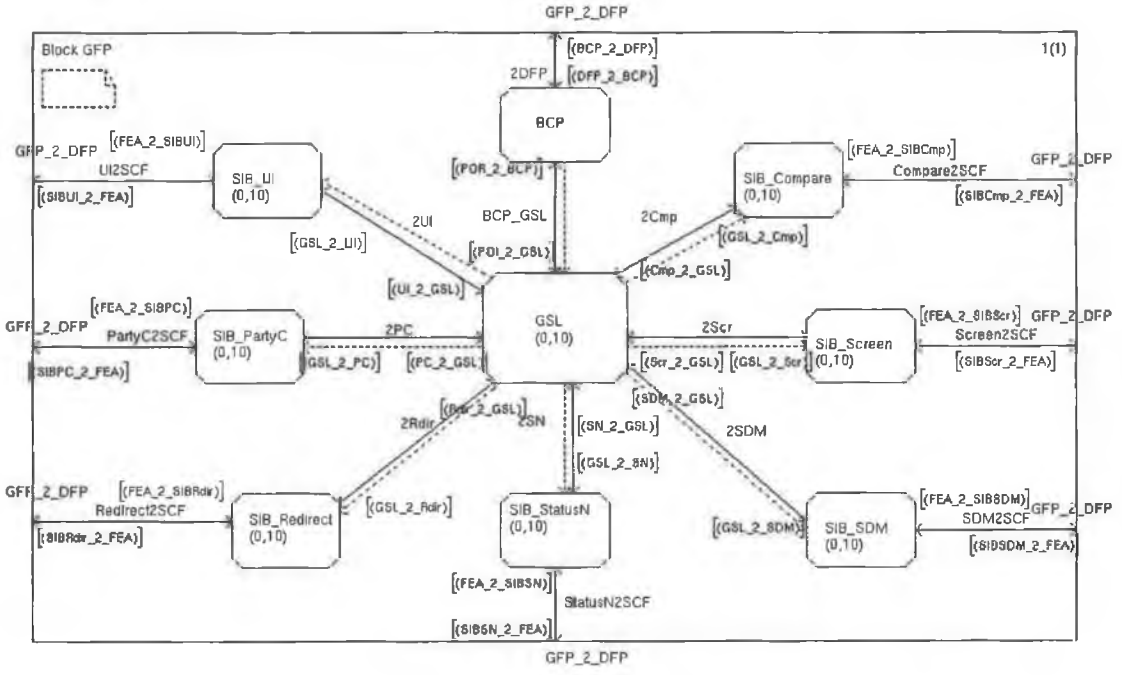


Figure 4.9 GFP represented in SDL

A single process each represents the GSL, the BCP and each SIB type. The process GSL contains the service logic for both services, describing the SIB chains and providing all the variables. The process BCP acts as an interface between the SSF and GSL. Detection points are sent from the SSF to the BCP. These detection points are interpreted as POIs and then the service is launched from the GSL. Each of the SIBs and the GSL process symbols bear the text *(0,10)*. This is SDL notation to define how many instances of a process initially exist and what the maximum number of instances allowed is. The first value inside the parenthesis is the initial number and the second is the maximum number of instances allowed to exist at any one time. In this case it is clear that at system start-up there are no instances of any processes in the GFP other than the BCP and that there can be a maximum of ten instances of each of the other processes. Therefore since only ten instances of the GSL are allowed a maximum of ten services can run on the system at any one time.

4.5.2 Communication

To invoke the next SIB in the SIB chain, the GSL must configure the parameters for the SIB and send these parameters in a signal to the relevant SIB process. Upon reception of the SIB parameters the SIB process will immediately call on an FEA in the DFP to carry out the functionality of the SIB with the given parameters. From Figure 4.9, it can be seen that the GSL has a separate signal route to each of the SIBs and these SIBs each have a signal route

connected to the channel GFP_DFP. An FEA in the DFP will return a response, via the channel GFP_DFP, to the SIB process that originated the request. This response will then be sent from the SIB process back to the GSL which will make a decision based on this response as to which SIB to invoke next. The SIBs used and their data parameters have been discussed in Chapter 2.

4.6 SDL At Process Level

It would be impossible to explain how each SDL process in the SDL system operates yet rather than completely ignoring the system operation at this level a certain section of one process will be considered. We will look at a small part of the CCAF process – one page of code out of five. The process page is shown below in Figure 4.10.

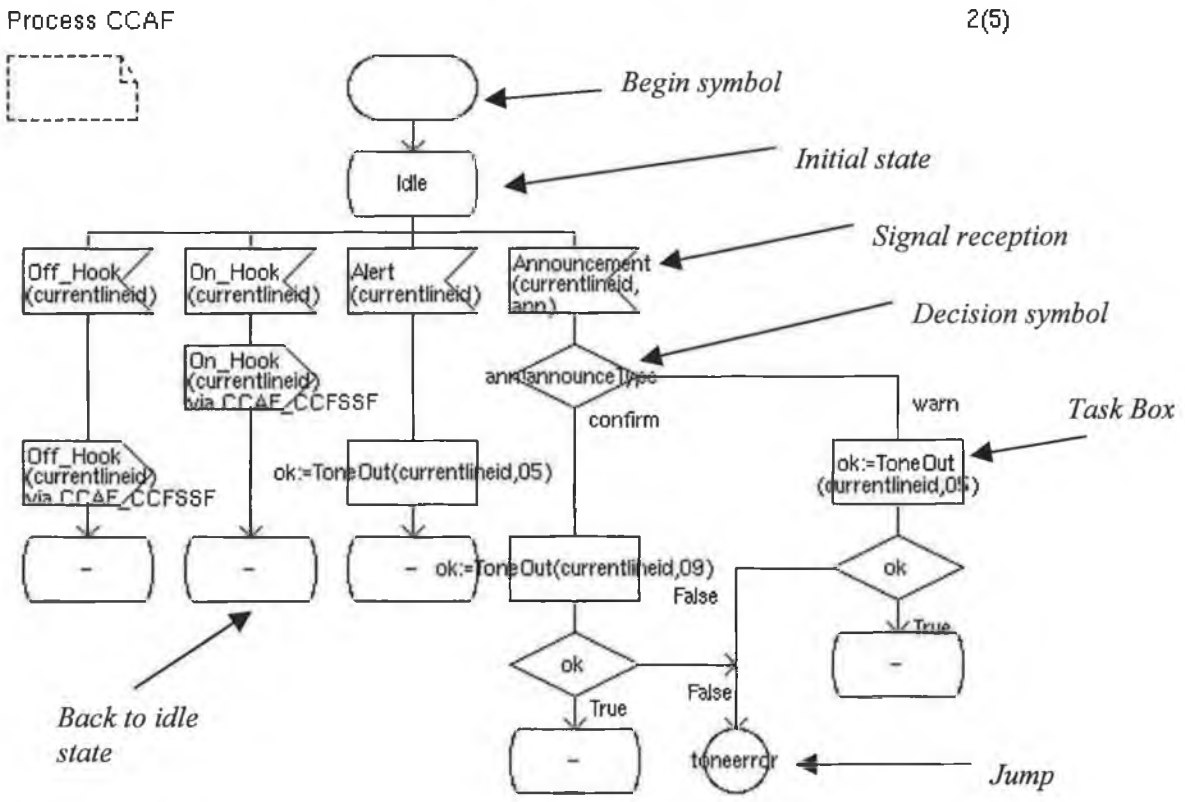


Figure 4.10 CCAF Section at Process Level

The process begins, according to SDL protocol, with a Begin symbol as shown above. It immediately moves to the Idle state. From this state the process can receive the signals *Off_Hook*, *On_Hook*, *Alert* or *Announcement*. Signals arriving in the CCAF are either from the switch or the CCF process. Which of these signals that the CCAF receives determines which path to follow in the process. If it receives the *Off_Hook* signal from the switch (bearing the extension number of the user who has gone offhook – *currentlineid*) then the process sends this signal on to the CCF via the CCAF_CCFSSF channel. The CCAF then returns to the idle state – this is represented diagrammatically by the state symbol with a dash in it instead of a state name. If the CCAF receives an *On_Hook* signal from the switch it passes it on to the CCF in the same fashion.

If the CCAF receives an Alert signal from the CCFSSF then it knows that it must send an Alert tone to the appropriate user. This is done in the task box shown above. In the task box the ToneOut ADT is passed the *ToneId* and user line identifier, as parameters, and run. If this ADT is successful (it returns *TRUE* to value *ok*) the process returns to the Idle state. Otherwise it jumps to the part of the CCAF process that deals with errors.

If the CCAF process receives an Announcement signal from the CCFSSF then it checks what type of announcement is to be sent in a decision box. If the announcement is of the type *confirm* then it calls the *ToneOut* ADT again and passes it the appropriate parameters. If it is to be a *warn* announcement then the ADT is called with a different *ToneId*. Again, if an error occurs the process jumps to the section that deals with errors. Otherwise the process returns to the Idle state where it is ready to receive more SDL signals.

This is a very small part of the CCAF's functionality to consider but it gives an idea of how the SDL system operates at process level.

4.7 The Excel CSN Switch

The call control services developed on the Intelligent Network Architecture are deployed on a telephony switching system. The IN architecture communicates with and controls this remote switching system. The switching system used is the Excel CSN (Communications Service Node) switching platform shown in Figure 4.11.

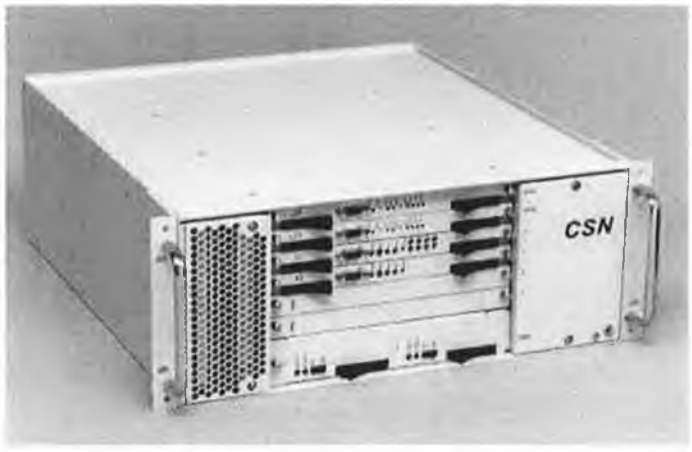


Figure 4.11 Excel CSN Programmable Switch

This section discusses the switch itself and the techniques used to establish communication between it and the IN. The CSN Programmable Switch is a narrow-band 1,024 port non-blocking programmable switch from the Excel product range. The switch itself is relatively unintelligent; all but the most basic software being resident on a host computer.

The switch comes with a Developers Tool Kit (DTK); a set of C code functions and programs which can be manipulated to develop applications to be run on the switch. The Developers Tool Kit, which is resident on the host computer, communicates with the switch by means of an Application Programming Interface (API) messaging set, which is provided in Appendix B. The Developers Tool Kit also provides four sample programs which show how these C functions may be used in an application.

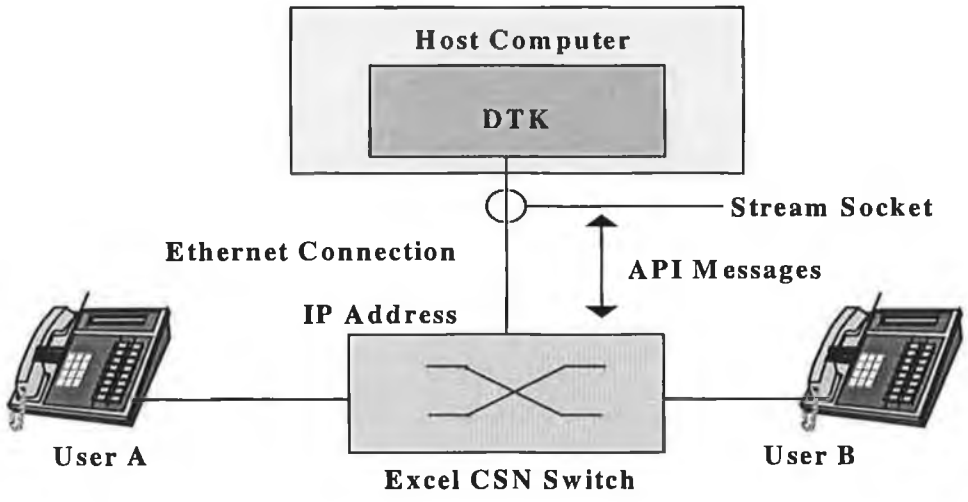


Figure 4.12 Excel Switch and Host Computer

The CSN hardware components include a matrix CPU, network interface cards, Digital Signal Processing (DSP) service card, ISDN and DASS 2 packet engine cards. It adapts to all standard network and line interfaces including T1, E1, J1, ISDN PRI and DASS 2.

4.7.1 Switch Features

Compact, powerful, flexible 1,024 port switch

- T1, E1, J1 and analogue network interfaces
- SS7 and ISDN signalling packet engines
- Multi-function DSP

High performance, scaleable switching

- Distributed, multiprocessor architecture
- Fully compatible with other switching systems from the Excel product range such as the LNX platform and EXS environment
- Exclusive selective space switching on service cards conserves time-slots for network
- Node in distributed EXS architecture via EXNET controller
- Bus interface (RBI) for network-transparent access to external service resources

Open, programmable architecture

- High level application programming interface (API)
- Exclusive Programmable Protocol Language (PPL)

The core of the CSN consists of a 1,024 port matrix CPU in a small-footprint, seven-slot chassis. The CSN is hardware and software compatible with the LNX product also from the Excel range. It uses identical network line interfaces, service resources, common channel signalling packet engines, and host interfaces.

Open Programmability: Communications between the CSN and the host use a standard API message format, conducted over standard protocols. Thus, the CSN does not require a specific type of computer, operating system or programming language.

Configurability: It is possible to connect directly to external voice resources without using network interfaces. When configured with Excel's Resource Bus Interface (RBI), the CSN connects to voice resource products through the use of industry-standard interfaces such as PEB, or MVIP, providing access between these resources and all ports. The RBI occupies only one I/O slot, instead of the front and back slot pair occupied for each network interface. Thus, service providers can make wider use of the universal slots in the CSN.

Scalability: The CSN and the Excel LNX switch share major components so it is possible to upgrade the CSN to an LNX to support higher port capacities. With the installation of an EXNET Controller, the CSN becomes a node in an Expandable Switching System (EXS), supporting expansion to 30,720 non-blocking ports over dual EXNET rings. This scalability provides seamless integration of all EXS switching resources with CSN resources.

Performance and Flexibility: The modular design of the CSN, featuring a mid-plane chassis and extensive use of daughter boards, enables new technologies to be introduced to the system. The CSN's distributed processing architecture utilises microprocessors on each element in the switch environment, resulting in improved performance per footprint. Excel's selective space switching architecture places a time-slot interchange on all major logical elements to optimise system resources. The result is an efficient use of network interfaces and cost savings for the service provider. Service resources and common channel signalling packet engines are available to all network line interfaces, eliminating the need to dedicate line interface channels to these resources and providing system-wide access to all internal resources. CSN system software takes advantage of the distributed architecture by utilising a layered call processing software architecture which maps messages to an internal ISDN-based format. This enables applications to be developed independently of network protocols, simplifying development efforts and expediting time to market.

Reliability: The CSN delivers the robust, system-wide fault tolerance commonly associated only with large central office (CO) switching environments. The CSN offers full 1+1 redundancy on the local system bus and the power supply. Within the chassis parameters, optional 1+1 redundancy can be achieved for the matrix CPU, host connection and for ISDN, SS7 and Sub-rate controllers. With multiple DSP cards, redundant pooling of individual DSP

resources is also available. N+1 redundancy is available for all digital T1, E1 network interfaces. CSN hardware redundancy works in tandem with automatic fault isolation and switches over to affect a seamless backup without interruption in service. In addition, the ability to physically remove, replace or add board components while the CSN remains in operation enhances and speeds maintenance and even expansion because it eliminates the need to power down the system. This is achieved by staggered connector design which applies power to the installing component prior to the connection of logic.

4.7.2 The Host/Switch Communications Link

A link must be established between the host computer and the switch so that they can communicate. This can be either an RS-232 serial link or Ethernet. In this scenario Ethernet link is used. The host software provides Transport Control Protocol/Internet Protocol (TCP/IP) messaging between the host and the switch. The communicating paradigm between the switch and the host is the standard client/server model where the switch performs the role of the server and the host performs the role of client. All Ethernet communication occurs through the stream transport service TCP. The switch has an IP address that is stored in EPROM on the EX/CPU card. The IN, developed in SDL, runs on a UNIX platform on the host computer and communicates with the switch by opening up a stream socket through which all API messages pass. It is the responsibility of the host to establish and maintain this connection since the host performs the role of the client. The host establishes this connection by performing a *connect()* call that specifies a port number, along with the IP address configured when the EX/CPU card was powered up.

4.7.3 The Application Programming Interface (API)

The Excel Application Programming Interface (API) is a set of messages used for communication between the host and the switch. Using the API messages the host can configure the switch and perform call processing functions. The switch uses the API messages to communicate call processing information, alarms, polls and status information to the host. There are 256 API messages which can be seen in Appendix B. The messages can be categorised into three groups:

- Configuration messages

- Call Processing Messages
- Alarms and Maintenance Messages.

The receiver of an API message must return a response. The response contains a status byte, which includes the result of the message sent. The host should analyse all status bytes to host-initiated messages to determine if they succeeded or failed.

Messaging between the host and switch is not symmetrical. Host messages are different from switch messages. The message flows between the host and switch are asynchronous. The host can send multiple messages to the switch, while the switch simultaneously sends multiple messages to the host. The host uses call processing to set up the intended connection. The communications driver software matches up acknowledgement messages (ACKs) returning from the switch for status validations.

Unique message sequence numbers align returning message ACKs with the original message. The originator of the message selects the sequence number. Typically, the host software driver provides this function for host initiated messages. This provides a unique sequence number counter for each message type. The host allows up to 255 outstanding messages for each message type.

4.7.4 The Developers Tool Kit (DTK)

The Developers Tool Kit software provides the host application developer with a set of C callable functions that perform basic functions required by host applications for Excel switching products. These functions include the following;

- Message Framing
- Checksum Calculation
- Handling Of Special Characters (0xFD)
- Inbound and Outbound Message Queues
- Message Acknowledgement Utilities
- Excel System Software Download
- Log file
- Text translation

The Tool Kit consists of these basic function modules, header files, and four sample programs that demonstrate how to use the functions. The four sample programs included with the DTK do the following;

- Download switch system software from a file residing on the host to the switch.
- Configure the switch from the host, using a configuration file.
- Continuously poll the switch for Switch originated messages.
- Log all operations in a text file.

The operations of the basic functions in the DTK are outlined below.

Receive Message Functions: Messages received by the host from the switch are handled by the functions *xlcom_rcv_framer()* and *xlcom_rcv_msg()*. The application, on reading bytes from the switch, stores them in a buffer. These bytes are then sent to the *xlcom_rcv_framer()* function which handles the lower level communications protocol. It ensures that all messages begin with the frame byte 0xFE, takes care of special character encoding and validates message checksums. The message is then queued until removed by the *xlcom_rcv_msg()* function. This function yields a return code to indicate success or failure of its operation. It also yields a pointer to a data structure containing the message itself.

Send Message Functions: The logic for sending messages to the switch from the host is provided by the functions *xlcom_snd_framer()* and *xlcom_snd_msg()*. In order to send a message the application calls *xlcom_snd_msg()*. This assigns the next available message sequence number to the message and saves it for later acknowledgement match up. This function then invokes *xlcom_snd_framer()* to frame the message, take care of checksum and handle special characters before sending it.

Sequence Number Assignment Logic: When sent, each message is assigned a sequence number ranging from 0 to 255. It is therefore possible to have 256 host initiated messages awaiting switch acknowledgement messages at any time. Every time *xlcom_snd_msg()* is called the sequence number to be used is incremented. If this number is already in use then a failure code is returned. Failure codes also occur when messages have been in the

acknowledgement waiting queue for too long or if too many messages have been sent in a short amount of time.

Acknowledgement Message Matching Logic: When the function *xlcom_rcv_msg()* receives a message from the switch it determines if it was switch initiated or host initiated. If it was host initiated then it is an acknowledgement from the switch. The original message is then located using its sequence number and matched to the acknowledgement message. The *xlcom_rcv_msg()* function also provides an expiration mechanism for messages awaiting acknowledgement. This avoids messages being "stuck" in the acknowledgement queue and the potential of not having available message sequence numbers for new outgoing messages.

Table 4.1 provides a list of some of the functions provided by the DTK:

Function	Summary
xlcom_init()	To initialise the Excel Developers Tool Kit
xlcom_rcv_framer()	To frame/queue Excel API messages sent from the Excel switch to the Host.
xlcom_rcv_msg()	To return a switch initiated message or an acknowledgement message received over the Excel API.
xlcom_snd_framer()	To translate Host send message data into Excel API "framed" message data.
xlcom_snd_msg()	To send a Host initiated message to the switch over the Excel API.
xlcom_snd_ack()	To send a Host acknowledgement to a switch initiated message over the Excel API.
xlcom_check_msg_ack_timers()	To check for switch acknowledgement message wait timer expiration.

Table 4.1 Excel Developers Tool Kit functions

4.8 Intelligent Network Interaction with the Switch

The IN interacts with a user via the Excel switch. The Call Control Agent Functions (CCAF and CCAF_2) of the IN are SDL processes which communicate with users via the switch. A

CCAF exists for every user on the system. In a call connection scenario the CCAF process deals with the *calling* party while the CCAF_2 process deals with the *called* party.

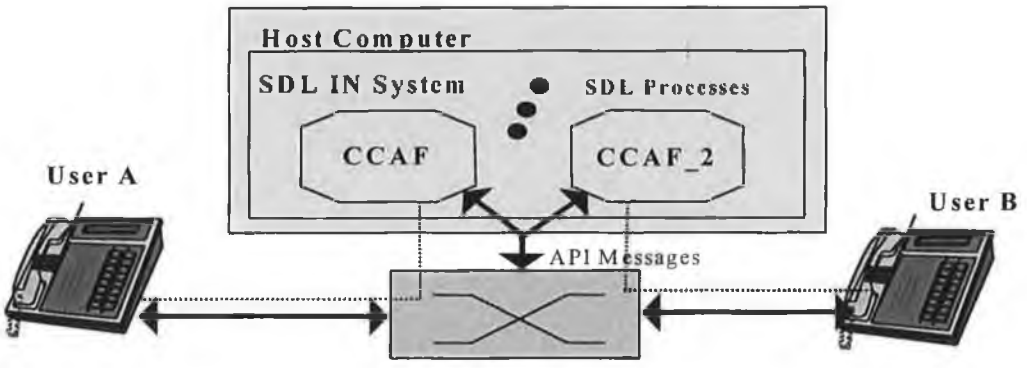


Figure 4.13 SDL to User interaction via the Excel Switch

The CCAF processes communicate with the switch using the API message set as shown in Figure 4.13. The SDL system uses a combination of two separate techniques to communicate with the switch using the API messages. These techniques are:

- Abstract Data Types (ADTs)
- Environment Functions

Abstract Data Types can be used to execute C code (i.e. the DTK functions) to interact with the switch. An ADT can generate an appropriate API message, send it and match it with the acknowledgement message when received. However by using ADTs it is not possible for the IN system to know when a switch initiated message has occurred i.e. an offhook, onhook or dialled digits. In order to capture such switch initiated messages the IN system must be able to continuously poll the switch. The SDL system achieves this by using the Environment Functions (in *ExcelEnv.c*). Therefore ADTs are suitable for sending host initiated messages while the Environment Functions are more suitable for capturing switch initiated messages.

4.8.1 Abstract Data Types

An Abstract Data Type has no specified data structure. Instead it specifies a set of values, a set of operations allowed on the data type and a set of equations that the operations must fulfil. This approach makes it possible to map SDL data types to data types used in other high-level languages such as C or C++. ADTs can also be used to execute code that is located in a program outside the SDL system so long as that external program is linked to the SDL

system at compile time. ADTs are used in the Intelligent Network SDL system to execute the C code functionality of the DTK in order to interact with and control the switch. A call to execute C code using an ADT can be made from within a task box in a SDL process. To explain this let us consider the use of ADTs to send a tone to a user. When the CCAF process needs to send a tone (Ringing, Busy, Dial, or Error etc.) to a particular user an ADT, *ToneOut,* is used. Figure 4.14 shows the command in the task box.

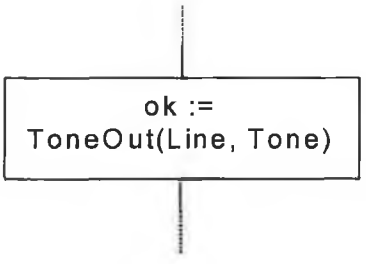


Figure 4.14 Calling ToneOut ADT from within an SDL Process

Calling *ToneOut* is rather like calling a procedure in other programming languages such as C. The ADT is passed two parameters, *Line* and *Tone*, and it supplies a return value, *ok.*

- *ok* is the return value. The ADT returns *True* if it executes correctly and *False* if an error condition is encountered.
- *ToneOut* is the name of the ADT procedure.
- *Line* is an integer value associated with a particular user. Each user attached to the switch is identifiable by a unique *Line* identifier.
- *Tone* is an integer value used to identify the type of tone that should be sent.

The *Tone* identifiers are as follows:

Tone	**Tone Identifier**
Dial Tone	1
Ringing Tone	2
Busy Tone	3
Error Tone	4
Alert Tone	5

Table 4.2 Table Tone Identifiers

The C code that executes this ADT's functionality resides outside the SDL system in a C program called *User.c*. This program contains the C code functionality of all the ADTs used in the system. At compile time the SDL system is converted to C code. This C program is then linked with *User.c*.

The ToneOut ADT is shown in Appendix C, complete with comments to explain its operation. It can be seen that the *ToneOut* ADT performs the following;

- Generates an API message *Connect Tone Pattern* and gives it the appropriate parameters to identify the user and the Tone Type.
- Frames and sends the API message to the switch across the stream socket.
- Waits for an acknowledgement message from the switch.
- Matches the acknowledgement message with the one sent.
- If the message was received and acknowledged return a Boolean value of True.

Another ADT, *EndTone*, is used to cancel tones. This is also called from a task box in the SDL application in the same manner as the ToneOut ADT.

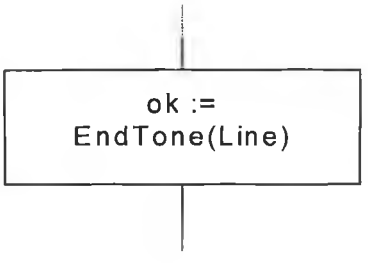


Figure 4.15 Calling EndTone ADT from within an SDL Process

This ADT stops *all* tones to a user so it is not necessary to pass it a Tone identifier as a parameter. *EndTone* generates a *Disconnect Tone Pattern* API message, frames it, sends it to the switch and waits for and matches the acknowledgement in the same way as the *ToneOut* ADT does. Again the functionality of this ADT is in the program *User.c*. Like the *ToneOut* ADT the *ok* return value is True if no errors were encountered. The *EndTone* ADT is also shown and commented upon in Appendix C.

Figure 4.16 shows how User.c, the environment functions and the IN are linked at compile time.

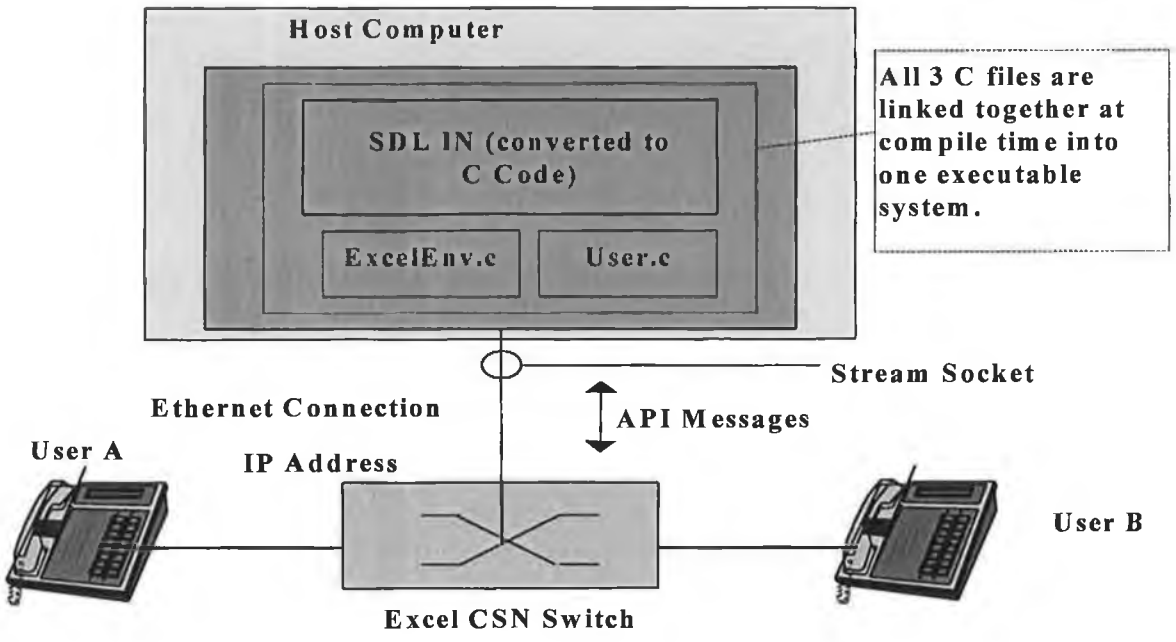


Figure 4.16 IN System on Host communicating with the Switch

4.8.2 The Environment Functions

The environment functions manage the SDL system's interactions with the external environment (the Excel switch) and are located *ExcelEnv.c*, which is linked to the SDL system at compile time. This C program handles all messages received by the SDL system from the switch. The Environment Functions are:

- **xInitEnv** which is called during the initialisation of the application
- **xCloseEnv**, which is called during the termination of the application.
- **xOutEnv** which should treat signals sent to the environment.
- **xInEnv** which treats signals sent to the SDL system from the environment.

As mentioned previously ADTs, rather than the *xOutEnv*, function are used to send host initiated messages to the Excel switch. Because ADTs have no way of knowing when switch initiated messages occur the *xInEnv* function is used to receive messages into the SDL system. *xInEnv* continuously polls the environment for the occurrence of such switch initiated messages. This is achieved by using a recurring loop in *xInEnv* function.

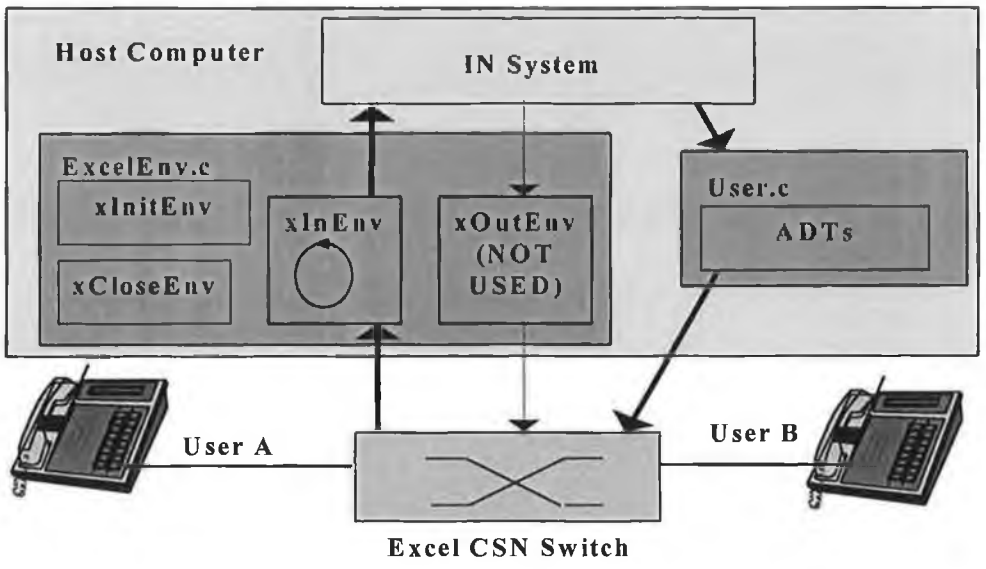


Figure 4.17 Interacting with the switch using ADTs and ExcelEnv.c

The environment functions for the interacting with the Excel switch, as shown in Figure 4.17, are now described:

Functions xInitEnv and xCloseEnv

These functions handle the initialisation and termination of the environment. The function *xInitEnv* will be called during the start up of the program. Its first action is to create the socket stream between the switch and the host. It also initialises the switch. The *xCloseEnv* function shuts down the socket between the host and the switch and frees up all unused memory. Both functions are shown in Appendix C.

Function xOutEnv

Each time a signal is sent from the SDL system to the environment of the system, the function *xOutEnv* is called. This function is not used in conjunction with the IN SDL system because ADTs were deemed to be more suitable. However its use will be explained for the sake of completeness. The *xOutEnv* function would have as a parameter the signal being sent. The signal contains the signal type, the sending and receiving process instance and the parameters of the signal.

```
void xOutEnv ( xSignalNode *S );
```

The parameter of *xOutEnv* is an address to *xSignalNode*, that is an address to a pointer to a struct representing the signal. The reason for this is that the signal that is given as parameter to *xOutEnv* should be returned to the pool of available memory before return is made from the *xOutEnv* function. This is made by calling the function *xReleaseSignal*, which takes an address to an *xSignalNode* as parameter, returns the signal to the pool of available memory, and assigns 0 to the *xSignalNode* parameter. So there should be a call:

```
xReleaseSignal(S);
```

before returning from *xOutEnv*. The *xReleaseSignal* function is defined as follows:

```
void xReleaseSignal ( xSignalNode *S );
```

In the function *xOutEnv* you may use the information in the signal that is passed as parameters to the function. First it is usually suitable to determine the signal type. This is best performed by *if* statements containing expressions of the following form, assuming the signal has the name *Sig1* in SDL:

```
(*S)->NameNode == Sig1
```

Suitable expressions to reach the Receiver, the Sender, and the signal parameters are:

```
(*S)->Receiver
(*S)->Sender
((yPDP_Sig1)(*S)) -> Param1
((yPDP_Sig1)(*S)) -> Param2
```

Sender will always refer to the sending process instance, while Receiver is either a reference to a process in the environment or the value xEnv. xEnv is a PId value that refers to an environment process instance, which is used to represent the general concept of environment, without specifying an explicit process instance in the environment. The structure of the xOutEnv function is shown in Appendix C.

Function xInEnv

In order to make it possible to receive signals from the environment and send them to the SDL system the function *xInEnv* is repeatedly called during the execution of the system. During such a call the environment is scanned to see if anything has occurred (i.e. a switch initiated message) which should trigger a signal to be sent to the CCAF process within the SDL system.

```
void xInEnv (SDL_Time Time_for_next_event);
```

To implement the sending of a signal into the SDL system two functions are available: *xGetSignal*, which is used to obtain a data area suitable to represent the signal, and *SDL_Output*, which sends the signal to the specified receiver according to the semantic rules of SDL. The parameter *Time_for_next_event* contains the time for the next event scheduled in the SDL system. The parameter will either be:

- 0, which means that the function can be executed immediately,
- Greater than 0, indicating that the next event is a timer output scheduled at the specified time, or
- A very large number indicating that there is no scheduled action in the system, that is, the system is waiting for external stimuli.

The environment should be scanned, current outputs performed, and returned as fast as possible if *Time* has past *Time_for_next_event*. For communication with the switch the *Time_for_next_event* parameter is zero since it is desirable to capture a switch initiated message as soon as it occurs.

The function *xGetSignal*, which is one of the service functions suitable to use when a signal should be sent, returns a pointer to a data area that represents a signal instance of the type specified by the first parameter.

```
xSignalNode xGetSignal (xSignalIdNode SType, SDL_PId Receiver, SDL_PId Sender );
```

The components Receiver and Sender in the signal instance will also be given the values of the corresponding parameters.

***SType*:** This parameter should be a reference to the symbol table node that represents the current signal type. Using the system interface header file, such a symbol table node may be

referenced using the signal name directly. The message received will be either *Onhook*, *Offhook* or *DigitsDialled*, which are all of a special type defined in the system interface header file. The system interface header file is an automatically generated header file that is included with the system at compile time.

***Receiver*:** This parameter should either be a PId value for a process instance within the SDL system, or the value *xNotDefPId*. The value *xNotDefPId* is used to indicate that the signal should be sent as an output without a TO clause, while if a PId value is given it is treated as an output with a TO clause. Note that PId values for process instances in an SDL system cannot be calculated but have to be captured from the information (sender or parameter) carried by signals coming from the system. This is the normal procedure in SDL to establish direct communication. The PId value in this case will be that of the CCAF process.

***Sender*:** Sender should either be a PId value representing a process instance in the environment of the current SDL system or the value *xEnv*. *xEnv* is a PId value that refers to an environment process instance, which is used to represent the general concept of the SDL environment, without specifying an explicit process instance in the environment. In this scenario *xEnv* is used to refer to the Excel switch.

The function *SDL_Output* takes a reference to a signal instance and sends the signal into the SDL system according to the rules of SDL.

```
void SDL_Output ( xSignalNode S,  xIdNode ViaList[] );
```

S: This parameter is a reference to a signal instance i.e. *Offhook*, *Onhook* or *DigitsDialled*, with all components filled in.

ViaList: This parameter is used to specify whether or not a VIA clause part of the output statement. The value *(xIdNode *)0* is used if no VIA clause is present.

To introduce a via list in the output requires a variable, which should be an array of *xIdNode*, that contains references to the symbol table nodes representing the current channels (or signal routes) in the via list. In more detail, we need a variable:

```
ViaList xIdNode[N];
```

where N should be replaced by the length of the longest via list we want to represent plus one. The components in the variable should then be given appropriate values, such that component 0 is a reference to the first channel (its symbol table node) in the via list, component 1 is a reference to the second channel, and so on. The last component with a reference to a channel must be followed by a component containing a null pointer (the value *(xIdNode)0*). Components after the null pointer will not be referenced. It is shown below how to create a via list of the two channels, CCAF_User and CCAF_2_User.

```
ViaList xIdNode[4];
/* longest via has length 3 */
...
/* this via has length 2 */
ViaList[0] = (xIdNode)xIN_CCAF_User;
ViaList[1] = (xIdNode)xIN_CCAF_2_User;
ViaList[2] = (xIdNode)0;
```

The variable *ViaList* may then be used as a ViaList parameter in a subsequent call to SDL_Output. To send a signal *Offhook* from *xEnv* into the SDL system the code will be (for the sake of simplicity lets consider the case without parameters):

```
SDL_Output(xGetSignal(Offhook, CCAFPId, xEnv), ViaList[0]);
```

The code will be similar for an *Onhook* signal. Both signals will contain the integer parameter *LineID* which specifies which extension caused the event. The handling of parameters is dealt with as follows:

If *DigitsDialled*, with two integer parameters (i.e. '69' invoking Ringback service) should be sent from xEnv to the process instance referenced by the variable CCAFPId, the code will be:

```
xSignalNode OutputSignal; /* local variable */
OutputSignal = xGetSignal(DigitsDialled, CCAFPId, xEnv);
((yPDP_DigitsDialled)OutputSignal)->Digits = 69;
SDL_Output( OutputSignal, ViaList[0]);
```

It is more difficult to give a structure for the *xInEnv* function, than for the *xOutEnv* function discussed previously.

```
void xInEnv (SDL_Time Time_for_next_event)
{
 xSignalNode S;
if ( Sig1 should be sent to the system ) {
   SDL_Output (xGetSignal(Sig1, xNotDefPId,  xEnv), (xIdNode *)0);
}
if ( Sig2 should be sent to the system ) {
   S = xGetSignal(Sig1, xNotDefPId, xEnv);
   ((xPDP_Sig2)S)->Param1 = 3;
    ((xPDP_Sig2)S)->Param2 = SDL_True;
   SDL_Output (S, (xIdNode *)0);
}
 /* and so on */
}
```

An *xInEnv* function will in principle consist of a number of if statements where the environment is investigated. The environment is checked continuously until a switch initiated message such as *Offhook*, *Onhook* or *DigitsDialled* is detected. When such a message is detected then that signal is to be sent to the SDL system by executing the appropriate code to send the signal. The structure given above demonstrates the design of the *xInEnv* function. The *xInEnv* function is presented in Appendix C.

4.9 Migrating Towards TINA

Two possible paths for the migration from IN to TINA have been identified: *the Open Switch* path and the *Bridge to Legacy* path [INtoTINA]. These paths have been discussed in Section 2.10.4. Realising these paths was beyond the scope of this project but there follows a study of how they could be applied to the IN architecture developed here.

4.9.1 Application of the Open Switch Path

Migration from IN to TINA could be achieved by providing an IDL interface to the Excel switch so that it could be observed as a TINA object. Across this IDL interface the switch could send to and receive messages from objects in a TINA network. A TINA service could be invoked when a user interacts with the switch (i.e. goes off-hook, on-hook or dials digits). TINA objects could control the switch by sending instructions to its IDL interface and a TINA session may call on the switch to perform certain functions such as connecting users, playing announcements, sending tones etc. In this scenario the IN has disappeared and full migration to TINA has been achieved. Users attached to the switch may invoke and interact with TINA services immediately.

Interworking between the IN system developed in this project and such a TINA styled version of the Excel switch would be a step towards this full migration. This could be achieved by allowing the Call Control Agent Functions (CCAFs) of the IN to control the switch via its IDL interface in a CORBA environment. Work has already been carried out by the COMET group [COMET] in providing standard CORBA based interfaces to broadband switches produced by different vendors. This standardisation, referred to as Open Signalling [OpenSig], removes the necessity to know the vendor dependent API message sets when interacting with a switch. The work done by the COMET group in this area would be useful in designing an interface to the Excel switch. However, the COMET group focused exclusively on broadband switches whereas this project is concerned with developing an interface for the narrow-band Excel switch.

The CCAFs of the IN must also be given IDL wrapping so that they can communicate with CORBA messages across a TINA domain. An object, *CCAFInterface*, would be created to receive CORBA signals on behalf of the IN. These CORBA signals would then be converted to SDL and sent into the CCAFs via the Environment Functions. Similarly, by creating an object, *ExcelInterface*, which contained the Excel Developers Tool Kit (DTK) functionality and has an IDL interface, CORBA signals could be received on behalf of the switch, converted to the Excel API and passed to the Excel switch. All these CORBA signals would pass between objects via an ORB. This is illustrated below.

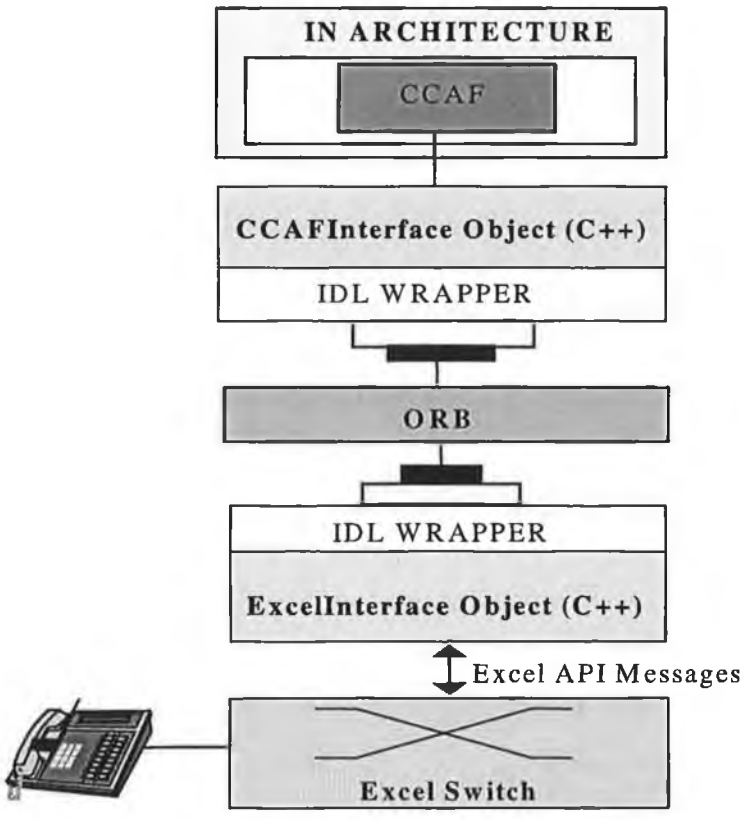


Figure 4.18 Communication between the CCAF and Excel Switch

Since *ExcelInterface* contains the DTK functionality to communicate with the switch it can receive CORBA messages to control the switch. Therefore a user wishing to control the switch can do so without having to know the vendor specific Excel API message set (Given in Appendix B). There are two directions of communication to be considered: CCAF initiated messages and switch initiated messages.

CCAF Initiated Messages

The basic call control functionality of the CCAFs includes the setting up and tearing down of calls, the detection of Onhooks, Offhooks and digits dialled, and the sending of announcements and tones. The CCAF call control functionality is achieved by invoking the following IDL methods on the interface of *ExcelInterface*:

Calls from CCAF to Switch	Purpose
Connectus(LineID, LineID)	Connect two specified parties in a voice call.
Disconnectus(LineID, LineID)	Tear down a call between two specified parties.
ToneOut(LineID, ToneID)	Send a specified tone to a specified party.

EndTone(LineID)	Cancel a tone to a specified party.
Announcement(LineID, AnnID)	Send a specified announcement to a party.

Table 4.3 Calls from the CCAF to the Excel Switch

These messages carry parameters as defined below:

LineID is an integer value which maps onto the channel number that the Excel switch uses to denote each extension attached to it.

AnnID is an integer value identifying the announcement played to users.

ToneID is an integer referring to the specific tone to be sent to an extension. See Table 4.4.

ToneID	**Tone**
1	Dial Tone
2	Ringing Tone
3	Busy Tone
4	Error Tone

Table 4.4 Tone Types

When *ExcelInterface* receives a CORBA signal the appropriate Excel API message is generated. This message is then framed and sent to the switch using the DTK functionality. *ExcelInterface* then waits for acknowledgement of the message from the switch. *ExcelInterface* also contains translation tables so that *LineIDs* can be converted to channel identifiers understood by the switch.

To examine further the communication between the CCAF and the switch, consider the following example, while referring to Figure 4.18.

Example: Connecting a Call

The SDL system wishes to connect two parties in a basic call. The following SDL signal is sent from the CCAF to the *CCAFInterface* object via the Environment Functions:

Connectus(Party1, Party2)

The parameters *Party1* and *Party2* represent the users to be connected in the call. *CCAFInterface* sends this message and its parameters as a CORBA signal, via the ORB, to the IDL interface of *ExcelInterface*. This CORBA signal invokes the method *Connectus(Party1, Party2)* in *ExcelInterface*. This method generates the API message *Connect* (See Appendix B). The *Party1* and *Party2* identifiers received in the CORBA message are translated to channel identifiers understood by the switch and passed as parameters of the API message. The API message is framed and sent to the switch. A copy of the message is stored in a buffer until an acknowledgement is received back from the switch. The acknowledgement and the stored message are then matched and the message is removed from the buffer. If no acknowledgement is received an error condition, *ConnectFailed*, is raised. On receipt of the *Connect* message the switch establishes the connection between the two parties. The IDL Definition for *ExcelInterface* is as follows:

```
typedef short LineID;
typedef short ToneID;
typedef short AnnID;
exception ConnectFailed;
exception DisconnectFailed;
exception ToneOutFailed;
exception EndToneFailed;
exception AnnFailed;
interface ExcelInterface {
    void Connectus (in LineID origparty, in LineID termparty)
        raises (ConnectFailed);
    void Disconnectus (in LineID origparty, in LineID termparty)
        raises (DisconnectFailed);
    void ToneOut (in LineID origparty, in ToneID tone)
        raises (ToneOutFailed);
    void EndTone (in LineID party) raises (True, False)
    raises (EndToneFailed);
    void Announcement (in LineID party, in AnnID announcement)
        raises (AnnFailed);
};
```

Switch Initiated Messages

Switch initiated messages occur when a user interacts with the switch by going Onhook, Offhook or by dialling digits. These messages are shown in Table 4.5 below.

Calls from the Switch to IN	Purpose
Offhook(LineID)	When a user goes Offhook.
Onhook(LineID)	When a user goes Onhook.
DialledDigits(Digits)	When a user dials digits.

Table 4.5 Calls from the Excel Switch to the CCAF

The parameter *Digits* is an integer representing dialled digits.

As the occurrence of switch initiated messages can not be predicted *ExcelInterface* must continuously poll the switch. When an event does occur it is converted to the appropriate CORBA message of Table 4.5. Channel numbers representing extensions will be converted to *LineID* parameters of the CORBA message. The message is sent to *CCAFInterface* via the ORB. From *CCAFInterface* the message is passed to the appropriate CCAF process in the IN system via the Environment Functions. *CCAFInterface* determines which CCAF the message is intended for and sends it into the SDL system via the Environment Functions. This is illustrated by the example described below.

Example: A user going Offhook

A user connected to the Excel switch goes Offhook. The switch sends the API message *Call Processing Event* (see Appendix B) to *ExcelInterface* which is continuously polling the switch. The *Call Processing Event* message contains the channel identifier of the user and an indicator that the event was an Offhook as parameters of the API message. *ExcelInterface* converts the channel identifier to a *LineID* and sends the CORBA message *Offhook* to *CCAFInterface* with this *LineID* as a parameter. *CCAFInterface* sends this message to the CCAF process in the SDL system via the environment functions. The IDL description of *CCAFInterface* is shown below.

```
typedef short LineID;
typedef short Digits;
```

```
typedef boolean status;
exception CommsFailure;
interface CCAFInterface {
    status Onhook (in LineID party)
    raises (CommsFailure);
    status Onhook (in LineID party)
    raises (CommsFailure);
    status DialledDigits (in LineID party, in Digits digits)
    raises (CommsFailure);
};
```

4.9.2 Application of the Bridge to Legacy Path

The Bridge to Legacy Path of migration to TINA was discussed in Section 2.10.4.2. A solution that would be very suitable to the system developed in this project would be to place the gateway within the SCP rather than between the SSP and the SCP, as proposed in the Bridge to Legacy path. The gateway would separate the SIBs and the Service Logic from the SCF, allowing certain entities of the SCF to double up as TINA objects. This technique is more readily achievable on the IN system developed in this project than full bridge to legacy implementation proposed in Section 2.10.4.2, while still allowing for an exploration of IN TINA interworking.

Wrapping the IN SIBs and the Service Logic of the SCP in IDL would make them viewable as TINA objects. Thus they may be accessed by either the IN SCF or by other distributed TINA objects. Thus hybrid IN/TINA services may be developed. Services (IN, TINA or hybrid IN/TINA) may be launched by sending a detection point from the SSF to the SCF or indeed via a service retailer in the TINA environment. This scenario provides a service designer with a richer service platform, as it is possible to use aspects of both technologies IN and TINA.

Objects in a TINA network may therefore interact with the IN system via an ORB. This allows TINA services to incorporate IN logic in their design, thus creating hybrid IN-TINA services. TINA services may be launched from the IN system in the same manner as IN services, i.e. by launching a SLPI from the BCP. The IN system will interact with these TINA objects across a gateway as if they were IN Service Features in the Service Plane [Q.1202]. The resulting service platform consists of two parts: a legacy IN part and a new TINA part. It is necessary for these two parts to interwork for the provisioning of services. Consequently a

corresponding Adaptation Unit (AU) must be developed for the mapping between the IN architecture and the TINA architecture [HerzMage]. For the *full* bridge to legacy path this AU would be located between the SSP and the TINA service architecture. The mapping between INAP operations [Q.1218] and interface operations of TINA Computational Objects on a DPE would also be defined. Locating the gateway between the SIBs and SLPI, and the SCF, reduces the complexity of the AU while still allowing Interworking between the legacy IN and the TINA network and therefore facilitates exploration of the benefits of Interworking within the context of this project.

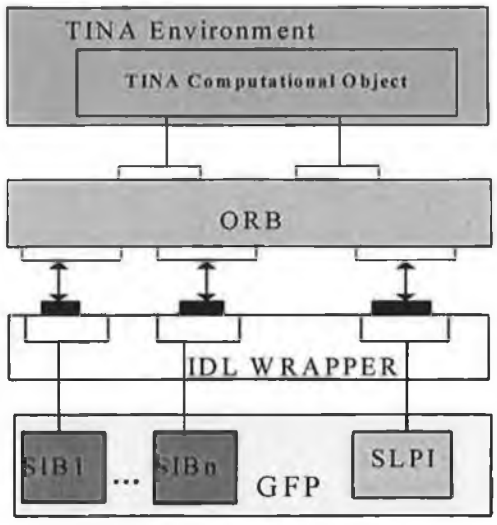


Figure 4.19 Accessing the IN Architecture via an ORB

The TINA Computational Objects (which have been discussed in the TINA Overview of Section 2.10.3) that will communicate with the IN system across the Adaptation Unit include:

User Application (UApp): The part of the service seen from the User domain.

Provider Agent (PA): This provides the access mechanism to the Retailer domain and connects a user to their User Agent.

User Agent (UA): This represents a user in the Retailer domain. It allows for the startup and the joining of service sessions by a user. There is one of these for every service.

Service Factory (SF): This is an object that knows how to start up a particular type of service. There is one of these for each type of service available. On a service request from a user, the User Agent contacts the Service Factory and passes on the request.

Service Session Manager (SSM): This deals with the general management of a service. The Service Factory creates one of these per service session.

Service Support Object (SSO): This is an object which provides service logic to support a service.

Figure 4.20 shows a hybrid service consisting of both IN and TINA service logic. Such a service or indeed a pure TINA service may be invoked from the IN system. The SSM invokes IN SIB functionality as well as TINA functionality, i.e. a TINA SSO.

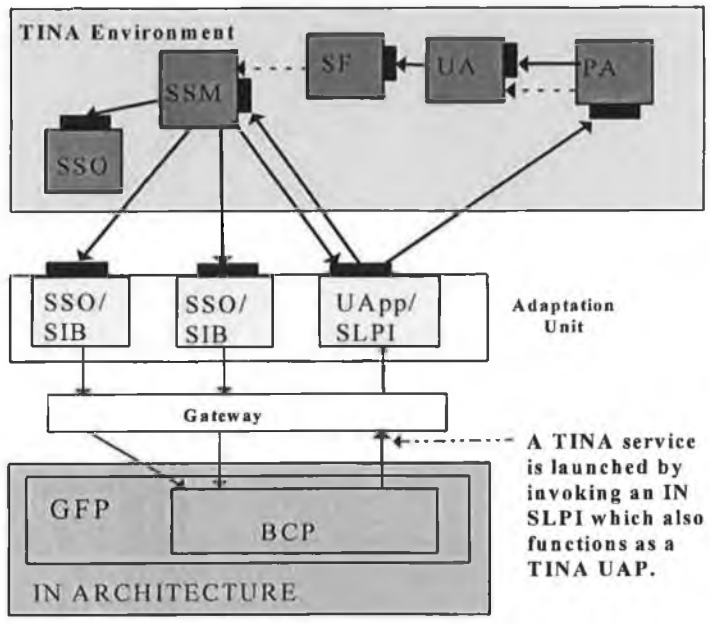


Figure 4.20 A hybrid IN/TINA Service being invoked from IN

Figure 4.21 gives another view of the Adaptation Unit interfacing between the TINA architecture and the legacy IN.

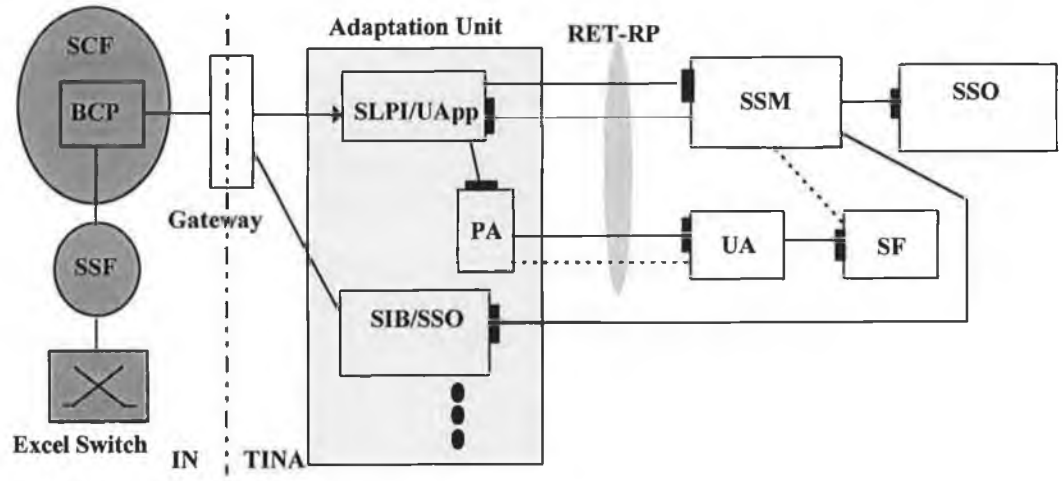


Figure 4.21 Adaptation Unit between IN and TINA

From the TINA side the AU appears as an end user system, i.e. the Retailer Reference Point (RET-RP), consisting of a User Application (UApp), any number of Service Session Objects (SSOs) and a Provider Agent (PA). From the IN side the AU appears as an SLPI and a set of SIBs. The SLPI and SIBs existed previously as SDL processes. They have been extracted from the SCF and given IDL wrappings so that they now exist as TINA objects. The extracted SIBs and SLPI must now be accessed from the IN system via a CORBA gateway. In this design the SLPI doubles up as a TINA UApp and a single TINA object SLPI/UApp contains the functionality of both. This object is seen from the IN side as the SLPI and from the TINA side as the UApp. An SLPI/UApp object will exist for each service. Similarly, the SIBs of the IN double up as TINA SSO object(s). An SSO may correspond to a single SIB or a sequence of two or more SIBs.

4.9.2.1 Hybrid IN/TINA Freephone Service

The execution of the hybrid IN/TINA Freephone Service is now detailed. The service is launched from the IN side of the AU. The steps explained below are numbered in Figure 4.22. An MSC for this service is provided in Appendix E.5.

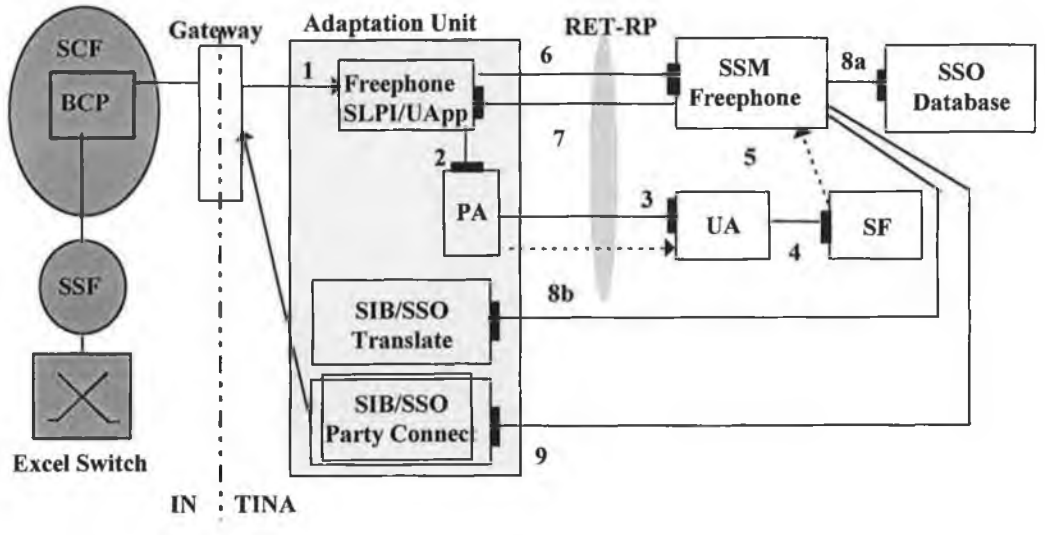


Figure 4.22 Hybrid IN/TINA Freephone Service

1: Having received the appropriate detection point from the SSF, the Basic Call Process selects and launches the Freephone SLPI/UApp across the gateway.

2: The SLPI/UApp requests the Provider Agent (PA) to create a Freephone User Agent (UA). Note: A new UA will exist for each service launched.

3: The PA creates a new UA specifically for the Freephone service.

4: The UA requests the Service Factory (SF) to create a Freephone SSM.

5: The SF does so and passes the SSM the reference for the UApp, which has been passed as a parameter of messages 2,3 and 4.

6: The Freephone SSM now queries the UApp.

7: The UApp responds to the SSM in the manner appropriate to the service being run. For the case of the Freephone service the UApp passes the SSM the Freephone number (i.e. 1800-) for translation to a destination line identifier.

8a: The SSM checks the translation table for the Freephone number using a TINA SSO i.e. a Database SSO.

8b: Alternatively, the SSM could use an IN SIB/SSO in the AU. In this example the SIB/SSO is the IN Translate SIB which converts the Freephone number into a destination line address.

9: The SSM launches the Party Connect SIB/SSO which connects the service initiator with the Freephone party by signalling to the IN SSF across the gateway.

4.9.2.2 Hybrid IN/TINA Audio-Video Conference

The execution of a hybrid IN/TINA audio-video conference service across the AU is now described. In this example two parties are connected in the conference. It is, however, possible to connect any number of parties. Each party in the conference has a User Application (UApp) on which they view the video connection and a telephone extension on which they hear the audio connection. The voice connections between the telephone extensions (ExtnA and ExtnB) are made by the narrow-band IN system while the video stream connections between User Applications are handled by the broadband TINA side. User A launches the service from UAppA. The conference uses the IN Party Connect SIB to set up the audio connection. The Connection Manager (CM), is used to connect the video stream between UAppA and UAppB. For the sake of clarity, the entities in the AU other than the Party Connect SIB/SSO have been omitted from Figure 4.23.

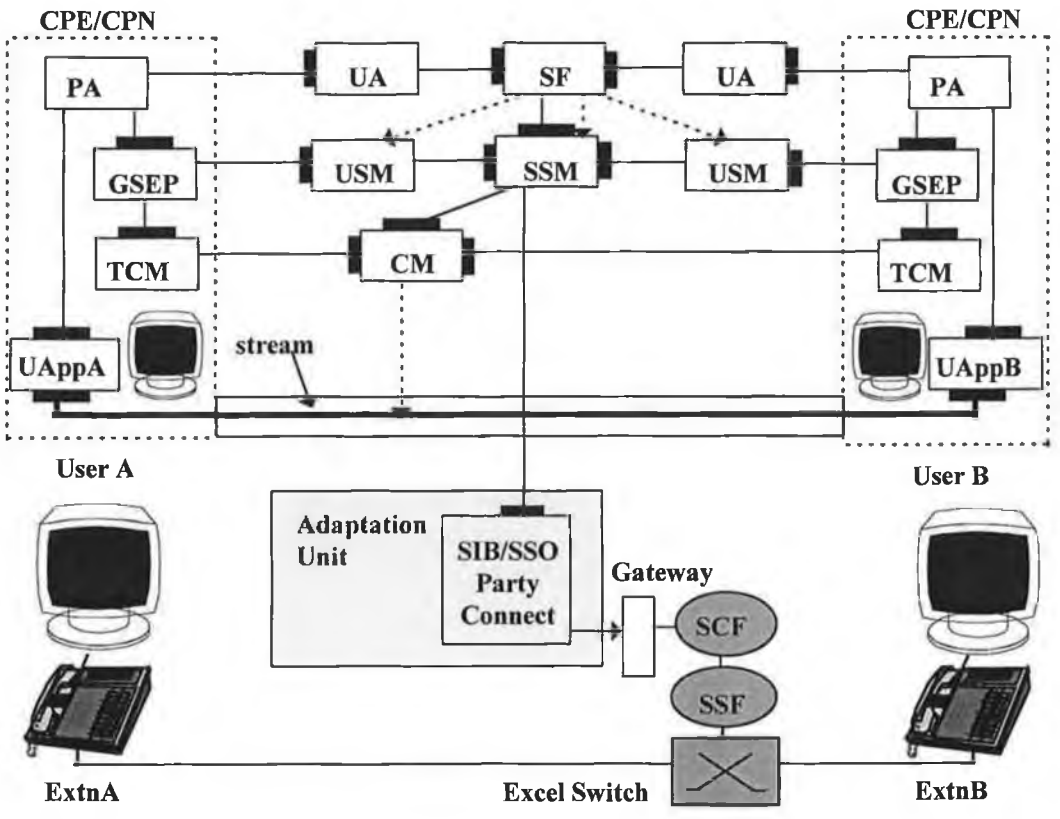


Figure 4.23 Hybrid IN/TINA Audio-Video Conference Service

The USM supports a user-centred view of the service in terms of personalised interfaces. The SSM has a global view of the session. It co-ordinates the association of parties within the session, supports the negotiation of communication resources and QoS parameters, and orchestrates the end-to-end communication. The Connection Manager (CM) sets up the video connection across the stream interface and co-operates with terminals' applications in order to close the communication on stream interfaces. The negotiation is based on an abstract view of resources and communication links independent from the underlying transport network. The Generic Session End-Point (GSEP) is a service independent computational object. It models the minimal set of capabilities as an end-point of an access session by interacting with the User Agent to perform service session control and invitation deliverance. When the video stream has been set-up the SSM invokes the Party Connect SIB/SSO to connect the phone extensions by calling both and setting up a basic call. The steps to create this service are illustrated in the MSC in Appendix E.6 and described below:

1. User A requests the Conference service from the PA. This is done from UAppA on their terminal. The extension number (ExtnA) of their nearest phone is passed as a parameter of the service request.

2. The PA passes the service request on to the SF via the UA.
3. The SF creates the Conference SSM and a USM for User A.
4. User A invites User B to join the conference by sending an invite message from UAppA to the USM.
5. The USM passes the invitation request on to the SSM.
6. The SSM passes the request to the UA of User B.
7. This invitation is then passed on to UAppB.
8. User B decides to join the service and sends a join request to the SSM, carrying the extension number of the nearest phone (ExtnB) as a parameter.
9. The SSM requests a stream from the GSEP of both parties.
10. The GSEP of both parties negotiates with their TCM for the stream information appropriate to the user's terminal.
11. This stream information for each terminal is sent to the SSM via the GSEPs.
12. The SSM asks the CM to set up the stream connections.
13. The CM creates the stream connection and informs the TCMs that they are ready.
14. The SSM establishes the audio connection by calling the IN SIB Party Connect and passing it the extension numbers to be connected as parameters. The SIB interacts with the IN system across the gateway to achieve this audio connection.

When the service ends resources are released and the session closes down. This requires the deletion of resources from the Session Graph and the corresponding release of network resources. When the graph is empty the session is completed. The audio connection ends when the users go on-hook and the call is torn down as normal.

4.9.2.3 Hybrid IN/TINA Ringback Service

The execution of the hybrid IN/TINA Ringback service across the AU is now detailed. The service is launched from the IN side of the AU. (It may be useful to reread Section 2.7.1 which describes the operation of the Ringback Service.)

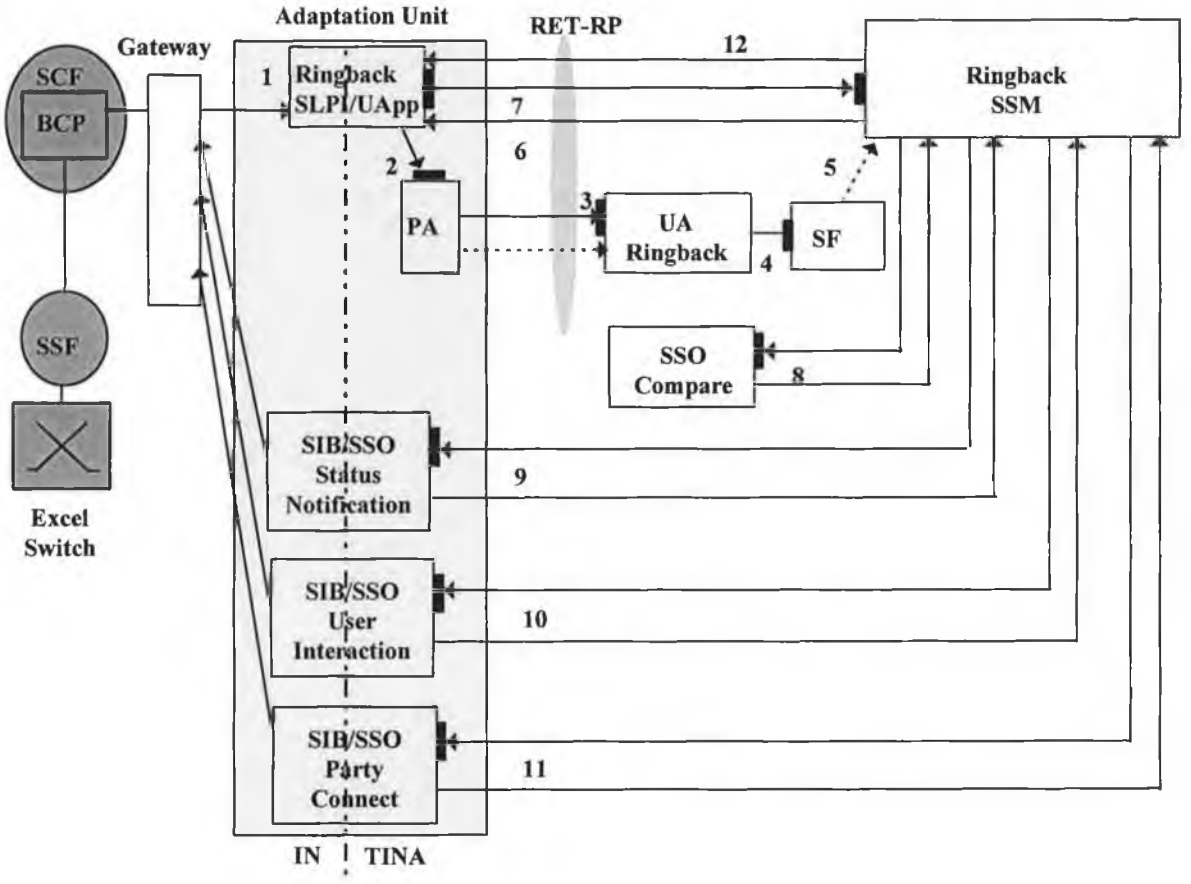


Figure 4.24 Hybrid IN/TINA Ringback Service

The steps, explained below, are numbered in Figure 4.24.

1: Having received a detection point from the SSF the Basic Call Process selects and launches the Ringback SLPI/UApp.

2: The SLPI/UApp requests the PA to create a Ringback UA.

3: The PA creates a new UA for the Ringback service.

4: The UA requests the SF to create a Ringback SSM.

5: The SF creates a Ringback SSM and passes it the reference of the SLPI/UApp, which has been passed as a parameter of messages 2,3 and 4.

6: The Ringback SSM now queries the SLPI/UApp.

7: The SLPI/UApp responds to the SSM by sending it the identifiers for the user initiating the service and the busy extension on which the Ringback is to be done.

8: The Ringback SSM launches an SSO to ensure that the extension initiating the service is not the same as the extension on which the Ringback is being done i.e. that an infinite

looping Ringback on itself is not being set up. The SSO responds to the SSM indicating that the numbers are not equal.

9: The Ringback SSM then launches the Status Notification SIB/SSO. This SIB plays an announcement to the service initiator (via the SSF) to indicate that the service has been initiated.

10: Simultaneous to the execution of step 9, the User Interaction SIB/SSO is launched. This SIB places a watch on the extension on which the Ringback is being done. As soon as this extension goes on-hook the SSM is informed.

11: Once the Ringback extension has gone on-hook the SSM launches the PartyConnect SIB/SSO which connects the two extensions.

12: When the parties have been connected the SSM informs the Ringback SLPI/UApp.

A Message Sequence Chart (MSC) showing the operation of this hybrid IN/TINA Service is provided in Appendix E.7.

It is important to note that the use of the Adaptation Unit described in the service examples above is an exploration of the feasibility of migrating to TINA from IN by interworking the service layers. The ideal solution would situate the Adaptation Unit between the SSF and the SCF. This, however, was beyond the scope of this project. An AU between the SSF and SCF which converts INAP messages [Q.1218] from the SSF to CORBA based messages for a TINA network above is currently being designed and developed by a response to a Request For Proposal by the Object Management Group telecom domain task force [OMG].

4.10 Conclusions

This Chapter described how an Intelligent Network architecture, based on the ITU-T IN Conceptual Model, was developed in SDL. The mappings of the INCM entities to SDL were explained as were the organisation and the communication channels for each entity.

The Excel Switch, on which the services were deployed, was described in detail as were the techniques used to interact with it: ADTs and the Environment Functions.

Simulations of the two selected IN services, Ringback and Group Call Pickup, were run and Message Sequence Charts showing their operation are presented in Appendix E. These MSCs show clearly how the IN services are launched from the Basic Call State Manager, and fully

describe their operation. They also demonstrate how a basic non-IN call is set up and torn down between two parties.

It was found that modifications to the Basic Call State Model processes were necessary each time a new service is introduced in the Intelligent Network. The BCSM has to know when an IN service is being invoked so that it may pass control up to the service layer via a detection point. Dialling the digit strings '69 'or '70' launches either of the services. Therefore each time a new service is added a special digit string may have to be assigned, i.e. 71, 72 and so on. This means that there is more to be done when adding new services than adding new SIBs. It will usually be necessary to modify the BCSM so that it recognises new digit string service requests. However, it is anticipated that as the system expands to contain more SIBs and services the necessary modifications will decrease.

A step towards the open switch path of migration to TINA that could be applied to the system developed in this project was presented. This path makes communication with the switch possible via a CORBA environment. Examples were presented to demonstrate how the IN would communicate with the switch via a CORBA gateway.

Instead of fully replacing the entire SCF with TINA computational objects as proposed in the Bridge to Legacy path of Migration to TINA another method of interworking suitable to the system developed in this project was proposed. This path is a step towards full migration and its benefit is that it enables the doubling up of certain entities as IN functional Entities and TINA computational objects. Thus it was shown that it is possible to invoke TINA services from an IN service platform. It was also shown that services could be built consisting of both IN and TINA functionality. This ability to combine broadband multimedia features with a narrowband IN system provides a much richer service platform.

Three hybrid IN/TINA services were designed to run on this platform: Freephone, Audio-Video Conferencing, and Ringback. These services use aspects of both the IN and TINA technologies and are therefore richer than plain IN services. The execution of these services can be observed by examining the MSCs in Appendix E.5, E.6 and E.7.

Chapter 5 EVALUATION CRITERIA FOR THE IN

Before developing the Intelligent Network architecture it was necessary to establish certain criteria with which to evaluate the system and its performance. These criteria may be divided into two categories: Functional Criteria and Non-Functional Criteria.

5.1 Functional Criteria

Multiple Users

The system must be capable of handling multiple simultaneous users. Many users will be attached to the switching system and the IN software must be able to respond appropriately to each user. The IN has no control over the timing and frequency of such user interactions but it must handle all of them regardless of current operations or events in the IN. The number of users interacting with the IN should not be limited by the IN software itself.

Multiple Services

Two IN services should be provided for execution on the IN architecture and it must be possible for both these services to run concurrently in any combination. This must also be the case for any further services added at a future date. In other words it should not matter what service is launched first and at what stage of its execution that the next service is launched and so on. The services must be able to coexist on the architecture and be completely independent of each other.

Feature Interactions are a problem that may occur in an Intelligent Network. This is a phenomenon where a service fails to perform as expected. The failure is due to interference from other services on the IN. Rigorous testing of the system must be performed to ensure that service interaction does not occur.

5.2 Non-Functional Criteria

Reliability

The operation of the system must be repeatable, dependable and sufficiently robust that crashing or hanging does not occur. Test case scenarios, designed to reveal any inappropriate behaviour of the system, should be run.

Fault Tolerance

The IN system must be able to effectively handle any runtime faults that may occur. The system should be able to recover from such faults and if necessary report them in a meaningful way to the system users.

Scalability

Scalability is essential to the concept of Intelligent Networks. INs were initially conceived in order to reduce the time to market for new services and to make the design of new services easier. It must be possible therefore that the IN system can be added to at later stages of its existence. This evolving of the system relates to the addition of new services as well as new functional entities. Also, interaction between the IN and broadband technologies such as TINA must be considered.

5.3 Meeting the Evaluation Criteria for the IN

It is now necessary to verify that the criteria outlined in Sections 5.1 and 5.2 have been met.

Multiple Users

The IN architecture is capable of handling multiple simultaneous users. The system interacts with the users via the two processes CCAF and CCAF_2. Only one instance of each of these processes ever exist. For every new user that interacts with the system a BCSM (Basic Call State Model) process in the CCFSSF Block is created. There are two types of BCSM process. If the user is a calling party then an O_BCSM (Originating BCSM) is created whereas if the

user is a called party a T_BCSM (Terminating BCSM) process is created. The BCSM process for each user is a high-level finite state machine description of that user's current state - be they involved in a basic call or an IN service invocation. The IN software puts no limit on the number of BCSM processes that can be created. There is a certain amount of overhead attached to signal flow management when dealing with multiple simultaneous user action. Therefore the memory of the host computer and the number of ports on the switching system are the only factors which limit the maximum number of users that can simultaneously interact with the IN. The Excel CSN switch used in this project can handle a maximum of 1,024 ports. Tests have shown that the IN can comfortably handle as many as 30 users interacting with it in the establishment of calls without the occurrence of any error conditions.

Multiple Services

The IN system was designed to handle the execution of multiple simultaneous services. This is made possible through the dynamic creation and termination of processes. Whenever a new instance of a service is requested the necessary processes are created and then terminated as soon as they have been executed. Thus the number of existing processes grows and decreases dynamically, throughout the life-cycle of the system, depending on how busy it is. An account of what processes are currently alive and their PId addresses is continuously updated so that there is never any confusion as to what instance of what service each process is dedicated to. Monitoring the PIds of the processes ensures that all signals are sent to the correct process instance, and that they do not interfere with other signals or processes in the system. There is an amount of overhead attached to signal flow management when dealing with multiple simultaneous service action.

Two Intelligent Network services were designed for the architecture: the *Ringback* service and the *Group Call Pickup* service. An IN service is launched when the BCP creates a GSL process and informs it which service it requires to be executed. The GSL process executes the SLP (Service Logic Program) for the requested service. These SLPs launch the appropriate SIB processes in the correct order to achieve the necessary functionality to realise the service. The IN software allows for a maximum of 10 instances of the GSL process to exist. This means that no more than 10 IN services can simultaneously coexist on the architecture.

A maximum of 10 instances of each SIB can exist at any one time. However, because each SIB terminates immediately after its operation is complete and because the design of the SIB chains that implement the services are sufficiently simple a deadlock situation will not arise

where one service is unable to invoke a SIB it requires - even if 10 services are running concurrently.

Further future services may be created which use a greater number of SIBs and where there is more sharing of SIBs between the services. This could, in a worst case scenario, reduce the maximum number of services that can coexist. However, if necessary the maximum number of SIBs or GSLs that can be invoked can be easily increased.

It will always be possible to execute a maximum of 10 instances, in any combination, of the two services developed in this project. This has proved true in rigorous testing of the system, launching the services in various combinations with varying timing.

Feature Interaction is a problem which may occur when multiple services exist simultaneously. The two services implemented in this project did not have any special feature interaction requirements so no major problems were encountered. One of the recommendations made by this thesis in the final chapter, for future work, is the development of a Feature Interaction Manager.

Reliability

The reliability of the system was verified by running simulations. These generated Message Sequence Charts (MSCs) which were checked thoroughly for any unexpected behaviour. Unexpected caller hang-ups, introduced into the system during a call or the execution of a service, were an important issue. When they occur a *Mid_Call* signal is introduced. If the abandoning user is running an IN service the signal sent to the GSL and all the active processes relating to the relevant service are terminated. An error signal is returned to the CCF via the SSF containing the error cause *unexpected_hang_up* as parameter. The CCF then kills all its active processes for that service i.e. the relevant BCSM processes.

All the parameters and process actions were examined closely at the end of each scenario test to ensure that all the required tasks were being performed. Scenarios tested were:

Single service activation:

- Ringback with various phone number combinations from 10001 - 10011,
- Group Call Pickup with various phone number and group number combinations,

- Ringback repeated with a user hang-up introduced at each SIB processing state,
- Group Call Pickup repeated with a user hang-up introduced at each SIB processing state.

Multiple simultaneous service activation:

- The two services repeatedly, beginning first with one and introducing the other at various stages and visa versa,
- The two services repeatedly, as above, but with the introduction of a user hang-up at different stages,
- Ringback, two at the same time,
- Group Call Pickup, two at the same time,
- Ringback and two Group Call Pickups,
- Group Call Pickup and two Ringbacks,

MSCs for tests with more than four or five services proved very difficult to read and took considerable effort to verify. Some of the exception cases detected are:

- Group Call Pickup operation can not be performed on a phone that is being polled due to Ringback.
- A resource cannot have the same service set twice for it, i.e. it can have both a Group Call Pickup and a Ringback service set for it, but not two Ringback or two Group Call Pickup services.

Fault Tolerance

Whenever an error condition in the IN system occurs an alarm is generated and the details of the error are printed to the screen on the host computer. A user will be able to see in which FEA, or otherwise, that the error occurred and of what type it was. In each SIB or FEA process there is a flag called *logicalError*. When an error occurs this is set to true. A variable *SIBErrorCause* is used to specify exactly what type of error it was. For example in *FEA_4081*, which carries out the functionality of the Screen SIB, possible error causes might include *invalid_screen_list* or *invalid_file_id*. Errors occurring in the lower layers of the IN are similarly reported. All ADTs return a Boolean operator to indicate success or failure.

The IN system is very robust and no crashes or hangs have yet occurred, despite the rigorous testing it has been given. The design is in itself "tolerant" to certain problems. Consider the scenarios of user and system related faults.

User Related Faults: Users of the system may stop dialling digits or stop in the middle of a call without notice. The time check mechanisms ensure that such an occurrence does not lead to an interruption of service for other users. If a user stops dialling for some reason while entering a number, then the dialled digits timer detects this interruption and kills off the process that dealt with that user's call in the first place. No interruption of service occurs for anyone else and the system continues as normal. Also, users may dial invalid numbers and again the same idea applies: service is maintained for everyone else and the user gets an announcement informing them that the number they dialled was invalid.

System Related Faults: Probably the most important aspect of fault tolerance in the system is its distributed software architecture. Controlling processes and managers within the system recognise requests and distribute them to the appropriate processes. This aspect of the architecture design is helpful in handling faults. Any "system type" problem will only effect the users of that particular call instance and not the entire system. When such a situation occurs the call for the affected user is shut down and the rest of the system carries on unaffected.

It could be possible to add some sort of "Still alive" signal to the system at a future date. This could be sent periodically from each call instance process to the controlling process or a "fault tolerant manager" indicating that everything is ok. If this signal is not received within a certain time frame then the relevant process can be killed, possibly a new one could be created to replace it and the problem reported to a log file for later study. It would be useful to build some techniques into the system to recover from such runtime faults. Recovery should be transparent to the user.

Scalability

The IN Architecture in its current form and facilitates the execution of two IN services. The architecture may, however, be further enhanced at a later stage by adding further services and features to it. The SDL/SDT development suite facilitates scalability very well. It is relatively simple to add further blocks or processes to an existing SDL system. This is one of the reasons why SDL/SDT was chosen to implement the architecture.

SDL allows the dynamic creation and termination of processes in a system at run time. This technique is used several times in this system. It is possible to change the maximum number of processes by simply incrementing the rightmost figure inside the parenthesis on the face of a process instance. This would be useful if it was decided that more numbers of SIBs need to be allowed to concurrently exist.

It is possible to add a new service to the architecture by editing the Global Service Logic, which determines which SIB functionality is to be invoked and in what order. Adding a new service, which uses the existing SIBs on the architecture, would simply be a matter of opening the GSL process and entering into it a new SIB chain describing the order of SIB execution.

It is very likely that to add a new service to the system it will be necessary to add new SIBs. Adding a new SIB process to the GFP Block can achieve this. SIB processes are relatively simple. They describe the order of execution of FEA processes in the SCF, SDF and SRF Blocks of the DFP. Further FEA processes can similarly be added to the SDL system when necessary.

Currently, the functionality of the SRF is limited to the playing of announcements. The SRF exists to provide the specialised resources required for the execution of IN provided services. Thus its functionality could later be extended to include resources such as digit receivers, conference bridges et cetera.

Similarly, the SDF, in its current form, is limited to storing and retrieving call-related data. Its functionality could be extended to include roles such as calculating call costs, storing caller specific data, network data and communicating with other SDFs if necessary.

Further Functional Entities could be added to the IN at a later date. These might include a Service Creation Environment (SCE) on which services could be designed using a Graphical User Interface (GUI) and a pallet of service functionality from which components can be dragged and dropped onto a workspace. Such a SCE would automatically generate the Service Logic from this workspace and the newly designed service could be subsequently run on the architecture.

A Service Management Function or a Service Management Agent Function, as described in Section 2.4.3, could also be added to the system at a later date. The introduction of a Feature Interaction Manager (FIM) to prevent service interactions was discussed earlier. An FIM could be developed in SDL and added to the system at a later date.

Interacting with a broadband TINA domain is an important area for future consideration. If the advantages of a TINA broadband service could be combined with those of a narrowband IN then there is greater possibility for richer hybrid IN/TINA service development. This was discussed in detail in Section 4.9.

Chapter 6 CONCLUSIONS

6.1 Overall Conclusion

The purpose of this project was to design and develop an Intelligent Network (IN) prototype, based on the standardised architecture, with particular emphasis on its service creation aspects. This thesis outlined the implementation of two services on the IN prototype using the ITU-T SIB methodology. The deployment of these services on an Excel switching platform is also described. The possibility of migrating the prototype to conform to TINA was also explored. Two techniques for doing this were investigated. The first involves making the IN SIBs and Service Logic accessible from the TINA network and vice versa. This is a step towards fully replacing the SCP by TINA computational objects. The other path for migration involves the replacement of the SSF and switch by a TINA open switching object. A step towards this is making the switch controllable from a TINA domain.

Intelligent Networks

The Telecommunication industry's need for IN standardisation and evolution is being responded to by ITU-T and other organisations. The CS-1 standards have a very restricted scope and are not sufficiently unambiguous to guarantee compatibility between different vendors' products. IN development to date has been concerned with the construction of a set of interfaces and protocols that clearly separate the switching from the service aspects of telecommunication networks. Distribution of the service application software itself is now being addressed. The TINA Consortium (TINA-C) was formed by the major network operators to discuss information-networking problems with telecommunications and computer vendors. Interworking between IN and TINA is a step towards full migration from IN to TINA and allows the development of hybrid IN/TINA services with broadband capabilities.

The Use of SDL

The SDL language was chosen to implement the Global Functional Plane (GFP) and the Distributed Functional Plane (DFP) of the IN architecture. The selection of SDL was largely

motivated by the availability of the SDT tool and the graphical syntax of the language, which provides a means of developing the system in an intuitive flow chart like manner.

Using SDL has a number of advantages over traditional programming languages. The SDL Graphical Representation (GR) description of the GFP/DFP planes of the INCM gives an overview of the system that is easier to understand than a C/C++ representation. SDL also offers the possibility to make late changes quickly and easily to the implementation if the specification is changed. Another advantage of using the SDL/SDT suite over other development environments is the ease with which testing can be performed. SDT allows simulations of the system to be run, where it is possible to step through its execution at any level of detail. At the same time MSC charts can be created so that the whole operation of the system can be viewed.

However, a limitation of SDL is that it is not possible to separate different entities in the environment, outside the SDL system. The Excel switch communicates with the IN using the Excel API message set. Messages are sent to the switch from the IN by calling Abstract Data Types (ADTs) which execute C code communication functionality that comes with the switch. Switch initiated messages are captured by the environment function and then sent into the IN as SDL signals.

Integration of the current version of SDT with CORBA is poor but this is being addressed in the next version to be released.

SDL Implementation of the IN

The SDL implementation of the IN focuses on the GFP and the DFP planes, each of which are represented by an SDL Block. The ITU-T Recommendation series [Q.1200] could be improved by using a complete SDL description. Only process diagrams are provided, without any structuring information and without any signal or data definitions. This means that the SDL diagrams in the standards only serve as illustrations of the written text in the standards. When building the IN system a number of implementation decisions had to be made. It became clear as the system was being developed that many issues concerning the operation of the INCM had not been fully examined by the standards and consequently the development became a means of exploring issues as well as producing an implementation of the standards. SDL facilitated the specification of both the distribution of functionality and communication within the system very well.

The Services

The two IN call control services chosen for implementation were *Ringback* and *Group Call Pickup*. These services were implemented using seven SIBs, five of which are defined in CS-1 while the other two had to be designed afresh in this project. Both services are narrowband call-control services that might typically be used in an office environment.

The *Ringback* service is invoked by a user (dialling '69') who has rung an extension that is busy. The user, having invoked the service, then hangs up. When the called party becomes available, the service initiator is rung. On answering, the called party is then polled and the parties are connected when the called party goes off hook.

The *Group Call Pickup* service is used when an extension rings but the person who owns it is away from their extension but at another one nearby. The call can be diverted to and answered from the other extension so long as that extension is a member of the same predefined group as the ringing extension.

The concept of migration from IN to TINA was explored and three hybrid services were designed to explore the possibilities for interworking on this architecture. The *Freephone* service is initiated by a user attached to the IN but uses a TINA database to convert the Freephone 1800 number to an extension number. The IN call is then connected by the IN Party Connect SIB. The *Audio Video Conference* is launched by a user from a terminal represented by a TINA User Application. The service connects two users by a video stream in the TINA domain. The users are connected in a voice call by using the call connection capabilities of the IN. The *Ringback* service is initiated by an IN user but uses both IN SIBs and TINA computational objects.

Service Creation

Seven SIBs were needed to implement the two selected IN services. The SIB methodology was found to be quite useful in terms of service creation. Further call control services would reuse many of these existing SIBs so that a large variety of services could be achieved without requiring an enormous bank of SIBs. Both services developed in this project reused the Party Connect and the Status Notification SIBs. Further SIBs, released in CS-2, will increase again the variety of services possible.

By interworking with a TINA environment the development of hybrid IN/TINA services was possible. Interworking yields a greater variety of services which use aspects of both the broadband and narrowband technologies.

The development of a Service Creation Environment Function (SCEF) in the DFP was outside the scope of this project.

The Excel Switch

The switch on which the services are deployed is an Excel CSN narrow band switching platform. This switch, which can have up to 1,024 ports connected to it, is itself relatively unintelligent. All but its most basic intelligence resides on the host computer connected to it. The switch comes with a Developers Tool Kit (DTK) which is a package of C code functionality residing on the host computer. The DTK facilitates communication with and control of the switch via its API message set. The functionality provided by the DTK is very adaptable and can be used to achieve a wide variety of applications to run on the switch. A disadvantage is that the API is specific to the Excel vendor. New Environment functions and ADTs would need to be written to communicate with a switch from a different vendor. It would be possible to communicate with the switch using a CORBA gateway. This concept, known as Open Signalling [OpenSig], makes control of the switch possible using a standard known interface rather than the vendor specific Excel API message set.

ADTs in the SDL system use the DTK functionality to build Excel API messages, send them across a UNIX socket stream to the switch and wait to match a returning acknowledgement. The Environment functions continuously poll the socket stream to capture switch-initiated messages when they occur. The messages were then converted to SDL signals and passed into the IN system.

6.2 Future Research

Version 3.3 of SDT was used in this project. Version 3.4 will not have any changes in the current CORBA support, however, work is being done on remodelling the entire CORBA support for the following release. This version will be based on IIOP, i.e. all signals that are sent to or from the SDL system can be sent using IIOP, thereby making it possible to connect

the SDL application to any ORB system. This also means that it will be possible to generate IDL descriptions from SDL. This will help the achievement of IN/TINA interworking.

It would be very useful to develop a Service Creation Environment Function (SCEF) with a Graphical User Interface (GUI) that is part of the IN. At present the IN services are each described in the Global Service Logic (GSL) by a chain of SIBs. With a GUI oriented SCEF it would be possible to select SIBs from a library, drag them onto a workspace and connect them to implement a service. This string of SIBs would then be automatically converted into a Service Logic Program Instance (SLPI) in the GSL.

At present the IN system has a reserve of seven SIBs. It would be beneficial to implement all the CS-1 SIBs on the Architecture. All new SIBs described in further Capability Set releases, i.e. CS-2, CS-3 etc., should also be incorporated into the architecture to provide a richer reserve of service creation capabilities.

The IN system developed in this project can handle multiple users running multiple services simultaneously. The services that were implemented did not have any special feature interaction requirements and therefore feature interaction was not a prime issue in the system design. Feature Interaction is, however, a valid topic for future study on this project it is recommended that a Feature Interaction Manager be added to the architecture.

Work is being done currently by the OMG Group [OMG] to define a standard INAP to CORBA translation for signalling between the SSP and a TINA service platform. This work would be very relevant to plans to migrate the system in this project to TINA. It would also be worthwhile to explore the possibility of replacing the SSF and switch with a TINA object and therefore achieve full migration from IN to TINA using the Open Switch technique.

BIBLIOGRAPHY

[Q.1200] ITU-T Recommendation series Q.1200: Q-Series Intelligent Network Recommendation structure, Helsinki, 1993.

[Q.1201] ITU-T Recommendation Q.1201: Principles of Intelligent Network Architecture, 1993.

[Q.1202] ITU-T Recommendation Q.1202: Intelligent Network Service Plane Architecture, 1993.

[Q.1203] ITU-T Recommendation Q.1203: Intelligent Network Global Functional Plane Architecture, 1993.

[Q.1204] ITU-T Recommendation Q.1204: Intelligent Network Distributed Functional Plane Architecture, 1993.

[Q.1205] ITU-T Recommendation Q.1205: Intelligent Network Physical Plane Architecture, 1993.

[Q.1208] ITU-T Recommendation Q.1208: General Aspects of Intelligent Network Application Protocol, 1993.

[Q.1211] ITU-T Recommendation Q.1211: Introduction to Intelligent Network Capability Set 1, 1993.

[Q.1213] ITU-T Recommendation Q.1213: Global Functional Plane for Intelligent Network CS-1, 1993.

[Q.1214] ITU-T Recommendation Q.1214: Distributed Functional Plane for Intelligent Network CS-1, 1993.

[Q.1215] ITU-T Recommendation Q.1215: Physical Plane for Intelligent Network CS-1, 1993.

[Q.1218] ITU-T Recommendation Q.1218: Interface Recommendations for Intelligent Network CS-1, 1993.

[Q.1219] ITU-T Recommendation Q.1219: Intelligent Network Users Guide for Capability Set-1, 1993.

[M.3010] ITU-T Recommendation M.3010: Principles for Telecommunication Management Network, 1993.

[SDT93] SDT Version 2.3 Users Guide and Reference Manual, Telelogic Malmoe AB, 1993.

[SDL89] ITU-T Recommendation Z.100, "Specification and Description Language SDL", 1989.

[Csu94] Peter Csurgay, "Service Creation and Deployment in an Intelligent Network Laboratory", Masters Thesis at IDT, NTH, Trondheim, 1994.

[Morris] Conor Morris, "An SDL based Intelligent Network service development environment", Masters Thesis, University of Limerick, 1996.

[INtoTINA] Cesare Mossotto, "From IN to TINA - the Step Forward?" CSELT, July 1997.

[HerzMage] U. Herzog, T. Magedanz, "Intelligent Networks and TINA - Migration And Interworking Issues", Proceedings of IS&N'97, Como, May 1997.

[OMG] Request For Proposal, "Interworking between CORBA and Intelligent Network Systems", OMG document, telecom/97-12-06, 1997.

[OpenSig] A. A. Lazar, F. Marconcini, "Towards an Open API for ATM Switch Control", COMET Group, April 1996.

[COMET] http://comet.ctr.columbia.edu/opensig/

[Excel] Excel LNX User's Manual, Excel Inc., 1996.

[ABC] Al Kelley and Ira Pohl, "A Book on C", The Benjamin/Cummings Publishing Company, second edition, 1990.

[SDTsu] Telelogic SDT Customer Support: support@telelogic.se, Telelogic Malmoe AB.

Appendix A Glossary

A.1 Terminology

Architecture:	Any ordered arrangement of the parts of a system.
Basic Call:	A call between two users that does not include additional features.
Basic Call Process:	The sequence of activities used in processing a basic call attempt.
Call Instance Data:	An identifier that defines subscriber specific details (i.e. its value will change with each call instance) for Service Independent Building Blocks in the Global Functional Plane.
Deploy:	To spread out (troops, etc.) so as to form a wider front to station or place (forces, equipment, etc.) in accordance with a plan to spread out or place like military troops.
Dynamic Arming/Disarming:	Enabling/disabling of a Detection Point by a Service Control Function in the course of service control execution for a particular call/service attempt.
Entity:	A part, device, subsystem, functional unit, equipment or system that can be individually considered.
Event:	A specific input to and/or output from a given state in a finite state machine model that causes a transition from one state to another.
Event Detection Point:	A detection point that is dynamically armed.
Feature Interaction:	A situation that occurs when an action of one feature affects an action or capability of another.
Functional Entity:	A set of processes defined for the purpose of achieving a specified objective, an entity that comprises a specific set of functions at a given location.
Global Service Logic:	Logic in the Global Functional Plane that is used to realise service features.
Information Flow:	An interaction between a communicating pair of functional entities.
IN Conceptual Model:	A planning model used for defining the Intelligent Network architecture.
Library:	An assembly of objects, routines, programs, etc. that may be drawn upon for use in the performance of functions.
Manager:	A function that directs and/or controls operations of a function or an assembly of functions to allow a functional entity to perform all or part of the expected functional entity actions.
Object:	An intrinsic component of an entity that is described at an appropriate level of abstraction in terms of its attributes and functions.
Plane:	A part of the IN Conceptual Model.
Point in Call:	A state in the Basic Call State Model.
Point of Initiation:	A functional interface between basic call processing and service logic over which service control is initiated.
Point of Return:	A functional interface between basic call processing and service logic

	over which call processing control is returned to basic call processing.
Relationship:	The complete set of Information Flows, where they exist, between two Functional Entities.
Resource:	In telecommunications, any network element that van be drawn upon in providing service, e.g. a circuit, a receiver, etc.
Service Control:	Direction of the functions or processes used to provide a specific telecommunications service.
Service Creation:	An activity whereby the capability to provide a supplementary service is brought into being from specification to development and verification.
Service Data:	Customer and/or network information required for the proper functioning of a service.
Service Feature:	A reusable part of one or more service capabilities forming all or part of a service.
Service Logic:	A sequence of processes/functions used to provide a specific service.
Service Logic Program:	A software program containing service logic.
Service Logic Program Instance:	The invocation and application of a particular Service Logic Program in providing a service or a service feature for a specific call/service attempt.
Service Management:	Management of user and/or network information required for the proper operation of a service.
Service Provider:	An organisation that commercially manages services offered to service subscribers.
Service Subscriber:	An entity that contracts for services offered by service providers.
Service Support Data:	An identifier that defines data parameters of specific service feature descriptions for Service Independent Building Blocks in the Global Functional Plane.
Single-Ended Service Feature:	A feature, e.g. call/service attempt manipulation, that applies to only one of the parties that may be involved on a call/service attempt.
Single Point Of Control:	A control relationship where the same phase or aspect of a call/service attempt is influenced by one and only one Service Control Function.
Static Arming/Disarming:	Enabling/disabling of a detection point, as directed by a Service Management Function, to cause a specific action by call/service processing whenever a specific point in call/service processing is encountered.
Supplemented Call:	A basic call with added service features or capabilities.
Trigger:	A stimulus for initiating an action.
Trigger Detection Point:	A detection point in basic call processing that is statically armed.
Vendor or Implementation Independence:	The characteristic that allows products from different vendors to work together in the same environment; and/or allows physical units serving as the same functional entity(ies) produced by different vendors to be used interchangeably.

A.2 Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
AU	Adaptation Unit
BCP	Basic Call Process
BCSM	Basic Call State Model
CCAF	Call Control Agent Function
CCF	Call Control Function
CCS	Common Channel Signalling
CID	Call Instance Data
CIDFP	Call Instance Data Field Pointer
CM	Connection Manager
CO	Computational Object
CORBA	Common Object Request Broker Architecture
CPE	Customer Premises Equipment
CPN	Customer Premises Network
CS	Capability Set
CSM	Communication Session Manager
CSN	Communications Service Node
DFP	Distributed Functional Plane
DP	Detection Point
DPE	Distributed Processing Environment
DTK	Developers Tool Kit
DTMF	Dual Tone Multi-Frequency
ESIOP	Environment Specific Inter-ORB Protocol
ETSI	European Telecommunications Standards Institute
FE	Functional Entity
FEA	Functional Entity Action
FIM	Feature Interaction Manager
GFP	Global Functional Plane
GIOP	General Inter-ORB Protocol
GSEP	Generic session End Point
GSL	Global Service Logic
GUI	Graphical User Interface
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IN	Intelligent Network
INAP	Intelligent Network Application Protocol
INCM	Intelligent Network Conceptual Model
IP	Intelligent Peripheral
ISDN	Integrated Services Digital Network
ITU	International Telecommunications Union
KTN	Kernel Transport Network
MSC	Message Sequence Chart
NCCE	Native Computing and Communication Environment
ODL	Object Description Language
OMG	Object Management Group
OO	Object Oriented
OOD	Object Oriented Design
ORB	Object Request Broker
PA	Provider Agent
PE	Physical Entity
PIN	Personal Identification Number
PPL	Programmable Protocol Language
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RET-RP	Retailer Reference Point
RFI	Request For Information
RFP	Request For Proposal
SCF	Service Control Function
SCEF	Service Creation and Environment Function
SCP	Service Control Point
SDF	Service Data Function
SDL	Specification and Description Language

SDT	SDL Design Tool
SF	Service Factory
SIB	Service Independent Building Block
SLEE	Service Logic Execution Environment
SLPI	Service Logic Program Instance
SMAF	Service Management Agent Function
SMF	Service Management Function
SOMT	SDL Object Modelling Technique
SP	Service Plane
SRF	Special Resource Function
SSD	Service Support Data
SSF	Service Switching Function
SSM	Service Session Manager
SSO	Service Support Object
TA	Terminal Agent
TBCSM	Terminating Basic Call State Model
TCM	Terminal Connection Manager
TDP	Trigger Detection Point
TINA	Telecommunications Information Networking Architecture
TMN	Telecommunication Management Network
UA	User Agent
UApp	User Application
USM	User Session Manager

Appendix B Excel Switch API Message Set

Appendix C Sample ADTs and the Environment Functions

C.1 ADTs

C.1.1 The ToneOut ADT

```
/* ------------------------------------------------------------------------ */
/* ToneOut.adt:  Sends a tone to a user.                                    */
/* ------------------------------------------------------------------------ */
SDL_Boolean    ToneOut (SDL_Integer LineID, SDL_Integer ToneID) {
 RTNCODE   rtncode;
 UBYTE     response_ack;
 CONNECT_TONE_MSG connectTone;
 response_ack = FALSE;

/* BUILD THE CONNECT TONE API MESSAGE */
 connectTone.length     = (UBYTE)8;
 connectTone.type       = XLMSGtypeCONNECT_TONE;
 connectTone.seq_num    = (UBYTE)0;
 connectTone.lsid       = (UBYTE)get_lsid(LineID);
 connectTone.channel    = (UBYTE)get_channel(LineID);
 connectTone.tone       = (UBYTE)ToneID;
 connectTone.cyclesMSB  = (UBYTE)0xFF;            /* Continuos Tone */
 connectTone.cyclesLSB  = (UBYTE)0xFF;
 connectTone.event_flag = (UBYTE)0;  /* Don't inform host of Tone completion */
/* SEND THE CONNECT TONE API MESSAGE */
  if ((rtncode=xlcom_snd_msg(matrix_id, app_key, (UBYTE *)&connectTone))
      == XLDTKrtnSUCCESS)
  {
/* WAIT FOR A RESPONSE */
     printf("Connecting Tone\n");
     while ( !response_ack && (rtncode == XLDTKrtnSUCCESS) )
     {
/* COLLECT BYTES FROM THE SWITCH */
       xlusr_com_read(XLDTKmtxA, raw_buf, sizeof(raw_buf), &byte_count);
/* FRAME MESSAGES  */
       if (byte_count > 0)
       {
```

```
        rtncode = xlcom_rcv_framer(XLDTKmtxA, &raw_buf[0], byte_count);
      }
/*  PROCESS SWITCH RESPONSE MESSAGES */
      while (xlcom_rcv_msg(XLDTKmtxA, &rcv_blk) == XLDTKrtnSUCCESS)
      {
        if (rcv_blk.app_key == app_key)
        {
/* MATCH THE ACKNOWLEDGEMENT WITH THE MESSAGE SENT */
          if ( ((CONNECT_TONE_MSG *)(&rcv_blk.rcv_msg[0]))->type == connectTone.type)
          {
            response_ack = TRUE;
            switch (rcv_blk.status)
            {
              case XLDTKrtnSWACK:
                rtncode = XLDTKrtnSUCCESS;
                break;
              default:
                rtncode = rcv_blk.status;
                break;
            }
            break;
          }
        }
      }
    }
  }
 if (rtncode == XLDTKrtnSUCCESS)
 {
/* RETURN BOOLEAN VALUE TO SYSTEM TO INDICATE SUCCESS*/
  return (SDL_True);
 }
}
```

C.1.2 The EndTone ADT

```
/* ------------------------------------------------------------------------ */
/* EndTone.adt: Operator Function: EndTone                                  */
/* Disconnect a tone out for a specific Line Id                             */
/* ------------------------------------------------------------------------ */
SDL_Boolean    EndTone (SDL_Integer LineID)
```

```
{
  RTNCODE        rtncode;
  UBYTE          response_ack;
  UBYTE          raw_buf[XLDTKsizeXL_MSG];
  UWORD          byte_count;
  XLCOM_RCV_BLK    rcv_blk;
  DISCONNECT_TONE_MSG disconnectTone;
  response_ack = FALSE;
/* BUILD THE DISCONNECT TONE API MESSAGE */
  disconnectTone.length     = (UBYTE)8;
  disconnectTone.type       = XLMSGtypeDISCONNECT_TONE;
  disconnectTone.seq_num    = (UBYTE)0;
  disconnectTone.lsid       = (UBYTE)get_lsid(LineID);
  disconnectTone.channel    = (UBYTE)get_channel(LineID);
  if ((rtncode=xlcom_snd_msg(matrix_id, app_key, (UBYTE *)&disconnectTone))
      == XLDTKrtnSUCCESS)
  {
/*  WAIT FOR A RESPONSE */
    printf("Disconnecting Tone\n");
    while ( !response_ack && (rtncode == XLDTKrtnSUCCESS) )
    {
/*  COLLECT BYTES FROM THE SWITCH */
      xlusr_com_read(XLDTKmtxA, raw_buf, sizeof(raw_buf), &byte_count);
/*  FRAME MESSAGES  */
      if (byte_count > 0)
      {
       rtncode = xlcom_rcv_framer(XLDTKmtxA, &raw_buf[0], byte_count);
      }
 /*  PROCESS SWITCH RESPONSE MESSAGES  */
      while (xlcom_rcv_msg(XLDTKmtxA, &rcv_blk) == XLDTKrtnSUCCESS)
      {
        if (rcv_blk.app_key == app_key)
        {
/* MATCH THE ACKNOWLEDGEMENT WITH THE MESSAGE SENT */
          if ( ((DISCONNECT_TONE_MSG *)(&rcv_blk.rcv_msg[0]))->type ==  disconnectTone.type)
          {
            response_ack = TRUE;
            switch (rcv_blk.status)
            {
              case XLDTKrtnSWACK:
```

```
                rtncode = XLDTKrtnSUCCESS;
                break;
              default:
                rtncode = rcv_blk.status;
                break;
            }
            break;
          }
        }
      }
    }
  }
 if (rtncode == XLDTKrtnSUCCESS)
 {

/* RETURN BOOLEAN VALUE TO SYSTEM TO INDICATE SUCCESS*/
  return (SDL_True);
 }
}
```

C.2 The Environment Functions

C.2.1 The xInitEnv Function

```
/*-------------------------------------------------------------------
   xInitEnv
-------------------------------------------------------------------* /
 void xInitEnv( void )
{
  int  addr_size;
  char  TmpStr[132];
  struct timeval  t;
  t.tv_sec = 60;
  t.tv_usec = 0;
  exit_program = TRUE;

  sprintf(ip_addr, "136.206.36.143"); /* IP Address of Excel Switch */
  printf("IP address of switch: %s\n", ip_addr);
```

```
 sprintf(TmpStr, "\nMy Pid: %d\n", xGlobalNodeNumber());
 printf(TmpStr);

  /* INITIALISE THE EXCELSWITCH */
  if (xlcom_init() == XLDTKrtnSUCCESS) {
    xlusr_init();
    rtncode = xlusr_com_open(ip_addr, XLDTKmtxA);
    if (rtncode == XLDTKrtnSUCCESS) {
      exit_program = FALSE;
    }
    else {
     xlusr_com_usage("Excel - SDL Interface", NULL);
     SDL_Halt();
    }
  }
  fprintf(stdout,"\nEnter <p> to poll or <q> to quit: ");
}
```

C.2.2 The xCloseEnv Function

```
/*-------------------------------------------------------------------
   xCloseEnv  extern
-------------------------------------------------------------------*/
 void xCloseEnv( void )
{
 PRINTF("\nClosing this session.");
 xlusr_com_close(XLDTKmtxA);
 xlusr_cleanup();
}
```

C.2.3.The xOutEnv Function template

```
/* The OutEnv Function is not used but its structure is shown for the sake of completeness */
void xOutEnv ( xSignalNode *S )
{
 if ( (*S)->NameNode == Sig1 ) {
/* perform appropriate actions */
  xReleaseSignal(S);
  return;
```

```
 }
if ( (*S)->NameNode == Sig2 ){
 /* perform appropriate actions */
 xReleaseSignal(S);
 return;
 }
/* and so on */
```

C.2.4 The xInEnv Function

```
/*----------------------------------------------------------------
    xInEnv
----------------------------------------------------------------*/
 void xInEnv( SDL_Time  Time_for_next_event )
{
 struct timeval t;
 fd_set        readfds;
 char         *Instr;
 char          SignalName = '\0';
 xSignalNode    yOutputSignal;
 xSignalNode        Offhook;
xSignalNode     Onhook;
xSignalNode     DigitsDialled;
 int           i = 0;
 char          chr = '\0';
/* CHECK TO SEE IF THERE IS A SWITCH INITIATED MESSAGE */
 FD_ZERO(&readfds);
#ifndef XMONITOR
 FD_SET(1,&readfds);
#endif
 FD_SET(In_Socket,&readfds);
 if ( select(getdtablesize(),&readfds,0,0,&t) > 0 ) {
/* READ THE MESSAGE FROM THE SOCKET */
 if FD_ISSET(In_Socket, &readfds) {
    Instr = (char *)xAlloc(151);
    do {
     read(In_Socket, &chr, 1);
     Instr[i++] = chr;
    } while ( chr!='\0' );
    sscanf(Instr, "%c", &SignalName);
```

```
/* REMOVE THE PARAMETER FROM THE MESSAGE */
    Parameter = GetParam(Instr);
/* OFFHOOK SIGNAL */
    if ( SignalName == 'Offhook' ) {
     /* SDL-signal Offhook */
     yOutputSignal = xGetSignal(Offhook, xNotDefPId, xEnv);
     ((yPDP_Offhook)OutputSignal)->Param = Parameter;
     SDL_Output(yOutputSignal, ViaList[0]);
    }
/* ONHOOK SIGNAL */
    else if ( SignalName == 'Onhook' ) {
    /* SDL-signal Onhook */
     yOutputSignal = xGetSignal(Onhook, xNotDefPId, xEnv);
     ((yPDP_Onhook)OutputSignal)->Param = Parameter;
     SDL_Output(yOutputSignal, ViaList[0]);
    }
/* DIGITSDIALLED SIGNAL */
else if ( SignalName == 'DigitsDialled' ) {
    /* SDL-signal DigitsDialled */
     yOutputSignal = xGetSignal(DigitsDialled, xNotDefPId, xEnv);
     ((yPDP_DigitsDialled)OutputSignal)->Param = Parameter;
     SDL_Output(yOutputSignal, ViaList[0]);
    }
/* FREE UP MEMORY */
    xFree((void**)&Instr);
   }
  }
}
```

Appendix D SDL Signals

This Appendix gives a description of some of the major SDL signals that are used in the implementation of the IN architecture. SDL signals are passed between processes on interconnecting signal routes. They are carried between SDL blocks by interconnecting channels. An SDL signal may or may not carry parameters.

SDL processes are modelled as extended finite state machines that work concurrently with other processes. The reception of a signal causes a process to leave its current state and move, via certain operations, to a new state. Depending on how the process is designed the operations it performs on its way to entering each new state dictate its behaviour.

If a signal route or channel carries many signals then the SDL diagram can become very cluttered. For this reason *signal lists* are used. Instead of showing all the individual signals on a route they can be represented collectively as a signal list. This Appendix shows the signals contained in the main signal lists of the system and explains their purpose. The signal lists can be seen in the Figures of Chapter 4.

As already mentioned some signals carry parameters. For the sake of clarity these parameters are omitted here.

DFP SIGNALS

The following signal lists, between the DFP Functional Entities, are shown in Figure 4.3.

User_to_CCAF	
Off_Hook, Dial_String, On_Hook	These are the signals which come from a user connected to the Switch in the Environment. When the user goes *Off_Hook*, *On_Hook* or dials digits (*Dial_String*) the signal is sent to the CCAF in the IN.

The parameters of these signals would contain the line number of the user and the *Dial_String* signal would also carry the digits dialled.

CCAF to User	
Dialtone, Busytone, Ringingtone, Errortone, Announcement	The IN interacts with the User through the CCAF (or CCAF_2) process. Sending one of these signals to the switch plays a tone or an announcement. The user's line number is carried as a parameter of the signal.

The same signals exist between CCAF_2 and User

CCAF to CCFSSF	
Off_Hook, Dial_String, On_Hook	The CCAF passes on the signals and parameters received from the switch to the CCFSSF process. Similarly for CCAF_2.

CCFSSF to CCAF	
Dialtone, Busytone, Ringingtone, Errortone, Announcement, Connectus, Disconnectus, Alert, CancelDialtone, CancelRingingtone, CancelErrortone, CancelAlert, CancelBusytone	The CCFSSF process instructs the CCAF processes to send tones or an announcement to a user by sending the signals *Dialtone, Busytone, Ringingtone, Errortone, Announcement* to it. The signal *Connectus* instructs the CCAF to connect users in a call. *Disconnectus* instructs it to disconnect them again *Alert* instructs it to send an Alert tone to a User, The CCAF is instructed to Cancel a user's tone by the signals *CancelDialtone, CancelRingingtone, CancelErrortone, CancelAlert, CancelBusytone*. Again these signals carry the line number of the user as parameter.

These signals are repeated between CCAF_2 and CCFSSF.

The SCF can instruct the SRF to perform a specialised resource function, such as playing an announcement to a user, as part of an IN service.

SCF_2_SRF	
Play_req_ind	SCF instructs SRF to play announcement.

SRF_2_SCF	
Completion_resp_conf	Indication to SCF that the request has been completed.

SRF_to_CCFSSF	
SRFSetup_resp_conf,	This signal confirms a connection between the SRF and CCFSSF.
Announcement	The SRF requests the SSF to play an announcement to a user.

CCFSSF_to_SRF	
SRFSetup_req_ind,	Request for a connection between the SRF and SSF.
Release_req_ind,	Indication that the connection has been terminated.
SRFReport_resp_conf	Confirmation that the SRF request (Announcement) is complete.

The Screen SIB and the SDM (Service Data Management) SIBs access the FEAs in the SDF to perform data related functions.

SCF_2_SDF	
ScreenQ_req_ind,	SCF requests the SDF to perform a Screen operation.
SDMQ_req_ind	Request for SDM operation.

SDF_2_SCF	
ScreenQ_resp_conf,	Database response for screening data.
SDMQ_resp_conf	Database response for SDM operation.

Interactions between the SCF and the lower level layers are now considered. Many SIBs in the service layers need to interact with the lower layers to connect users, retrieve more information, monitor resources etc.

SCF_to_CCFSSF	
Poll_req_ind,	Party Connect SIB instructs the SSF to call a user.
Poll_Connect_req_ind,	Party Connect SIB instructs the SSF to call and connect to a user.
StatusRep_req_ind,	SCF asks CCF to monitor a resource (i.e. wait for an onhook).
InitialDP_resp_conf,	This indicates to the CCF the end of an IN service.
Connect_req_ind	Party Connect SIB instructs the SSF to connect two parties.

CCFSSF_to_SCF	
InitialDP,	CCF invokes an IN service by sending a Detection Point to SCF.
Mid_Call,	Report of a user hang-up during the execution of a service.
SRFSetup_resp_conf,	Confirm to SCF connection between SRF and SSF.
StatusRep_resp_conf,	Response to the Status Notification SIB.
Poll_resp_conf,	Response to Party Connect SIB that a party has been polled.
Connect_resp_conf	Response to PC SIB that parties have been connected.

While the SIB processes reside in the GFP, the FEA (Functional Entity Action) processes that implement their functionality reside in the Functional Entities of the DFP. A SIB invokes the FEAs by sending a Query to the SCF. The SCF will then create the appropriate FEA process(es) and initiate them. Once all the FEAs implementing a SIB have executed the SCF will send a Response back to the SIB in the GFP.

GFP_2_DFP	
InitialDP_resp_conf	Indicates end of an IN service and gives control back to CCF.
Compare_Query,	Request from the Compare SIB to launch and run the Compare FEAs.
UI_Query,	Request from the UI SIB to launch and run the UI FEAs.
StatusN_Query,	Request from the SN SIB to launch and run the SN FEAs.
PartyC_Query,	Request from the PC SIB to launch and run the PC FEAs.
Screen_Query,	Request from the Screen SIB to launch and run the Screen FEAs.
SDM_Query,	Request from the SDM SIB to launch and run the SDM FEAs.

Redirect_Query	Request from the Redirect SIB to launch and run the Redirect FEAs.

DFP_2_GFP	
InitialDP,	The SCF instructs the BCP to launch a new service.
Mid_Call,	The SCF informs the IN service of a user hang-up.
Compare_Response,	Indicate to Compare SIB that all relevant FEAs have executed.
UI_Response,	Indicate to UI SIB in the GFP that all relevant FEAs have executed.
StatusN_Response,	Indicate to SN SIB that all relevant FEAs have executed.
PartyC_Response,	Indicate to Party Connect SIB that all relevant FEAs have executed.
Screen_Response,	Indicate to Screen SIB that all relevant FEAs have executed.
SDM_Response,	Indicate to SDM SIB that all relevant FEAs have executed.
Redirect_Response	Indicate to Redirect SIB that all relevant FEAs have executed.

The following signal lists, within the SCF Block, can be seen in Figure 4.4. When a SIB runs a sequence of FEAs the first and last FEA to be run is always in the SCF. Consider the Status Notification SIB. The first and last FEA for this SIB in the SCF is FEA_9101. Similar signals exist for the other SIBs available but they are not considered here

Mgr_2_9101	
StatusN_Query	The SCF launches this FEA and sends it (as a parameter) the line number whose Status is to be monitored.

SN_2_SSF	
StatusRep_req_ind	A request sent to the SSF for a monitor or status check on a resource.

SSF_2_SN	
StatusRep_resp_conf	The SSF response to the request for a monitor or status check.

SN_2_GFP	
StatusN_Response	When the SIB has finished the Response is sent back to the GFP.

The following signal lists, within the CCFSSF Block, can be seen in Figure 4.7.

CCAF_to_CCFMgr	
Off_Hook, Dial_String, On_Hook	When the CCAF receives a signal from the switch it passes it on to the CCFMgr process in the CCFSSF block.

CCAF_2 also passes the signals which it receives on to CCFMgr.

CCFMgr_to_Orig_BCSM	
On_Hook, Off_Hook, Dial_String, Process_Shutdown, ConnectNow,	The CCFMgr passes the CCAF signals onto the Originating BCSM. When a user abandons a call kill the BCSM representing them. Connect the user represented by OBCSM to another user.

CCFMgr_to_Term_BCSM	
On_Hook, Off_Hook, Process_Shutdown, ConnectNow	CCAF_2 signals are passed on to Term_BCSM. If the user abandons the call then shut down their BCSM. Connect the user represented by TBCSM to another user.

Orig_BCSM_to_Term_BCSM	
ConnectNow, Close_Connection, Close_Connection_conf, WhoAreYou_req_ind, WhoAreYou_resp_conf	One BCSM requests a connection to another. Request to close the connection. Confirmation of the connection. Query to find out what user a BCSM represents. The response indicating the user it represents.

Term_BCSM_to_Orig_BCSM	
ConnectNow_conf, Close_Connection_conf, Close_Connection, WhoAreYou_req_ind,, WhoAreYou_resp_conf,	Confirmation of the connection between BCSMs. Confirmation of the disconnection between BCSMs. Request to close the connection. Query to find out what user a BCSM represents. The response indicating the user it represents.

GFP SIGNALS

The following signal lists, within the GFP Block, can be seen in Figure 4.9.

BCP_2_DFP	
InitialDP_resp_conf	Indicates the end of an IN service and hands over control to CCF.

DFP_2_BCP	
InitialDP,	CCF tells the BCP to launch an IN service.
Mid_Call	Reports a user hang-up to the BCP, which will inform the GSL.

POR_2_BCP	
POR,	Signal returned to BCP from GSL at the end of service processing.
Service_Failure	The service did not execute properly.

POI_2_GSL	
POI	Point of Initiation from BCP to GSL to indicate start of a service.

GSL_2_SN	
SIBStatusN_Params	The GSL launches the SN SIB and sends it appropriate parameters.

SN_2_GSL	
SIBStatusN_Result	When the SIB has finished it sends its result to the GSL.

SIBSN_2_FEA	
StatusN_Query	The SN SIB launches the chain of FEAs beginning and ending in the SCF.

FEA_2_SIBSN	
StatusN_Response	Response from the FEA in the SCF to the SN SIB after completion.

Very similar signals exist in the GFP pertaining to the other SIBs.

Appendix E

Message Sequencing Charts

This Appendix uses Message Sequencing Charts (MSCs) to demonstrate the operation of the Intelligent Network architecture and the services deployed on it. An MSC is a graphical and textual language used for the description and specification of the interactions between system components. MSCs are also used to provide an overview specification of the communication behaviour of real-time systems.

The MSCs in this appendix describing the operation of the IN services were generated from real simulations. However, they have been modified slightly for the sake of clarity: all superfluous information has been removed so that the more important interactions and operations pertaining to the overall system can be more clearly observed. Signal parameters not relevant to the overall system have been removed where it is felt they might otherwise cause confusion.

Each process in the SDL system is represented by a vertical line. Signals are represented as horizontal lines running between the processes. Time is represented as running along the vertical axis of the system, however it is not attempted by these MSCs to truly represent the timing of the system's internal interactions.

The following aspects of the IN behaviour are represented by the MSCs:

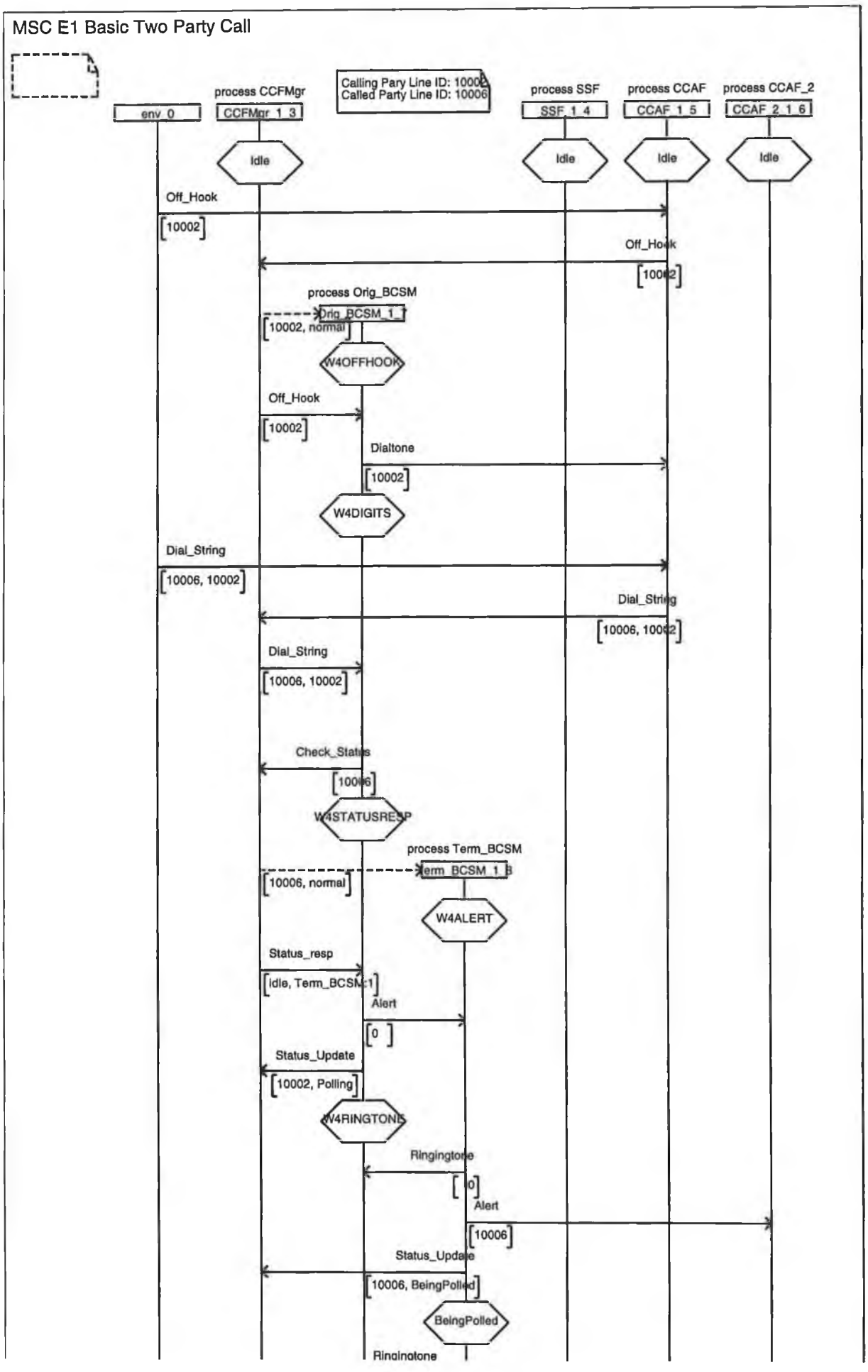
- Section E.1 shows the setting up and tearing down of a basic two party call.
- Section E.2 shows the invocation of an IN service by sending a Detection Point to the higher layers of the IN.
- Section E.3 shows the operation of the Group Call Pickup service.
- Section E.4.demonstrates the operation of the Ringback service.

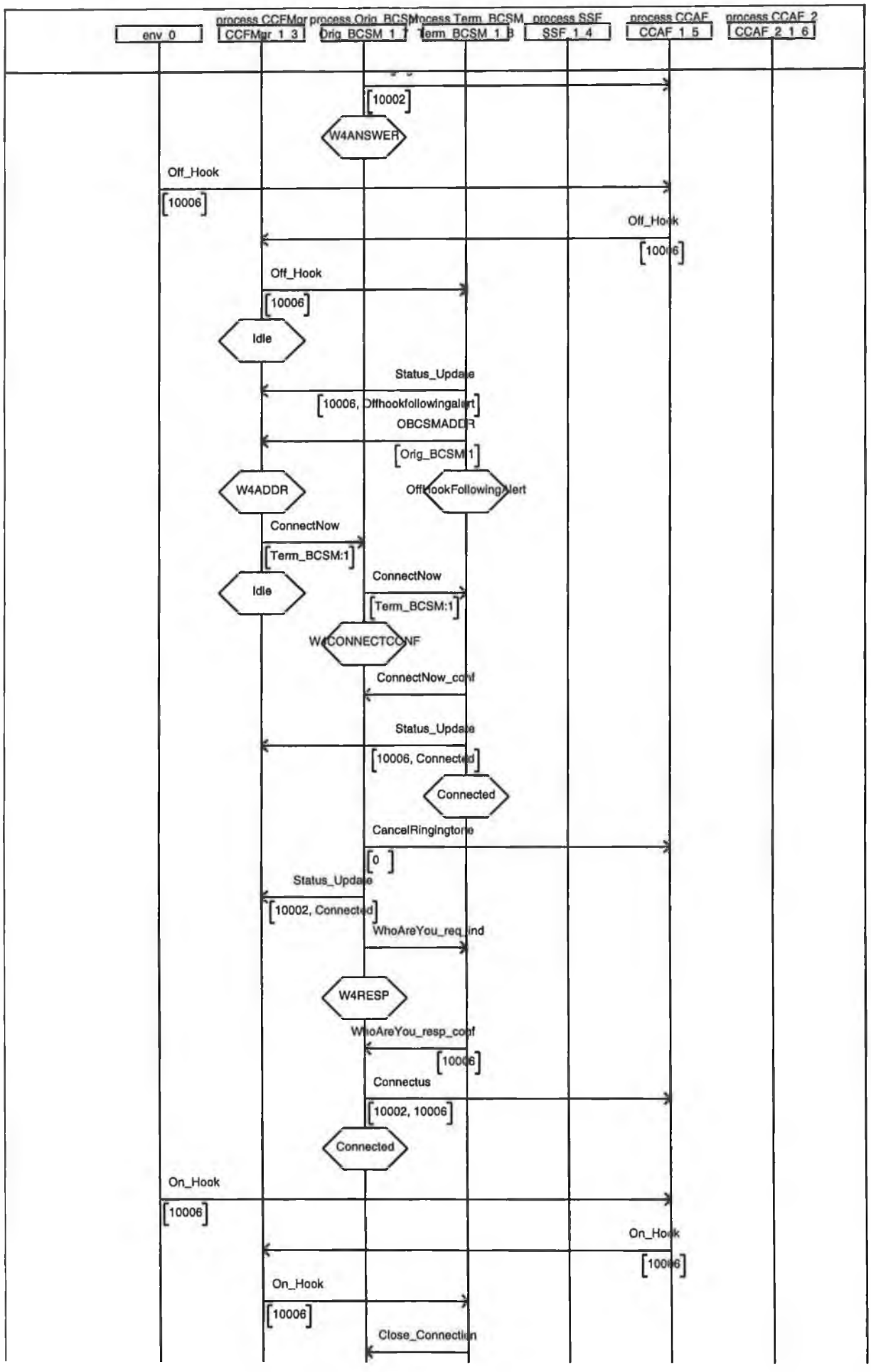
MSCs are used to explain the operation of the following hybrid IN/TINA services:

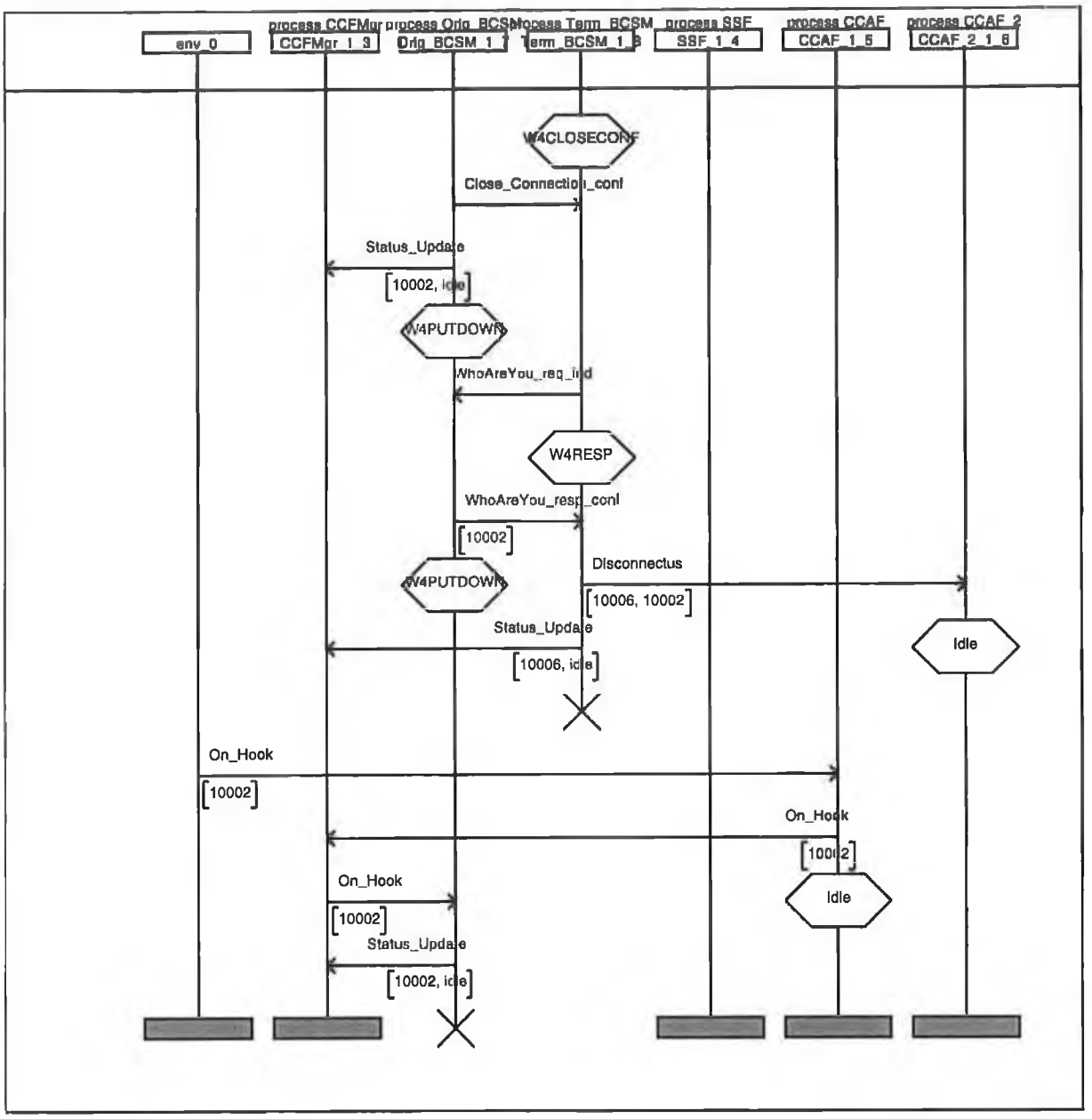
- Section E.5 details the operation of the hybrid IN/TINA Freephone Service.
- Section E.6 details the operation of a hybrid IN/TINA Audio-Video Conferencing Service.
- Section E.7 details the operation of the hybrid IN/TINA Ringback Service.

E.1 Setting up and tearing down a basic two party call

This MSC demonstrates the operation of the lower layers of the IN system in setting up a basic two party call. An IN service is not invoked therefore the higher service layers are not shown in this MSC. Both users are represented by Line Identifiers. The Line ID of the Calling party is 10002 and the Line Identifier of the Called Party is 10006. After the call has been set up, party 1006 hangs up to end the call. Party 10002 then hangs up and the call is torn down.


MSC E1 Basic Two Party Call
env_0
process CCFMgr
CCFMgr_1_3
Calling Pary Line ID: 10002
Called Party Line ID: 10006
process SSF
SSF_1_4
process CCAF
CCAF_1_5
process CCAF_2
CCAF_2_1_6
Idle
Off_Hook
[10002]
Off_Hook
[10002]
process Orig_BCSM
Orig_BCSM_1_1
[10002, normal]
W4OFFHOOK
Off_Hook
[10002]
Dialtone
[10002]
W4DIGITS
Dial_String
[10006, 10002]
Dial_String
[10006, 10002]
Dial_String
[10006, 10002]
Check_Status
[10006]
W4STATUSRESP
process Term_BCSM
Term_BCSM_1_3
[10006, normal]
W4ALERT
Status_resp
[idle, Term_BCSM:1]
Alert
[0]
Status_Update
[10002, Polling]
W4RINGTONE
Ringingtone
[0]
Alert
[10006]
Status_Update
[10006, BeingPolled]
BeingPolled
Ringingtone

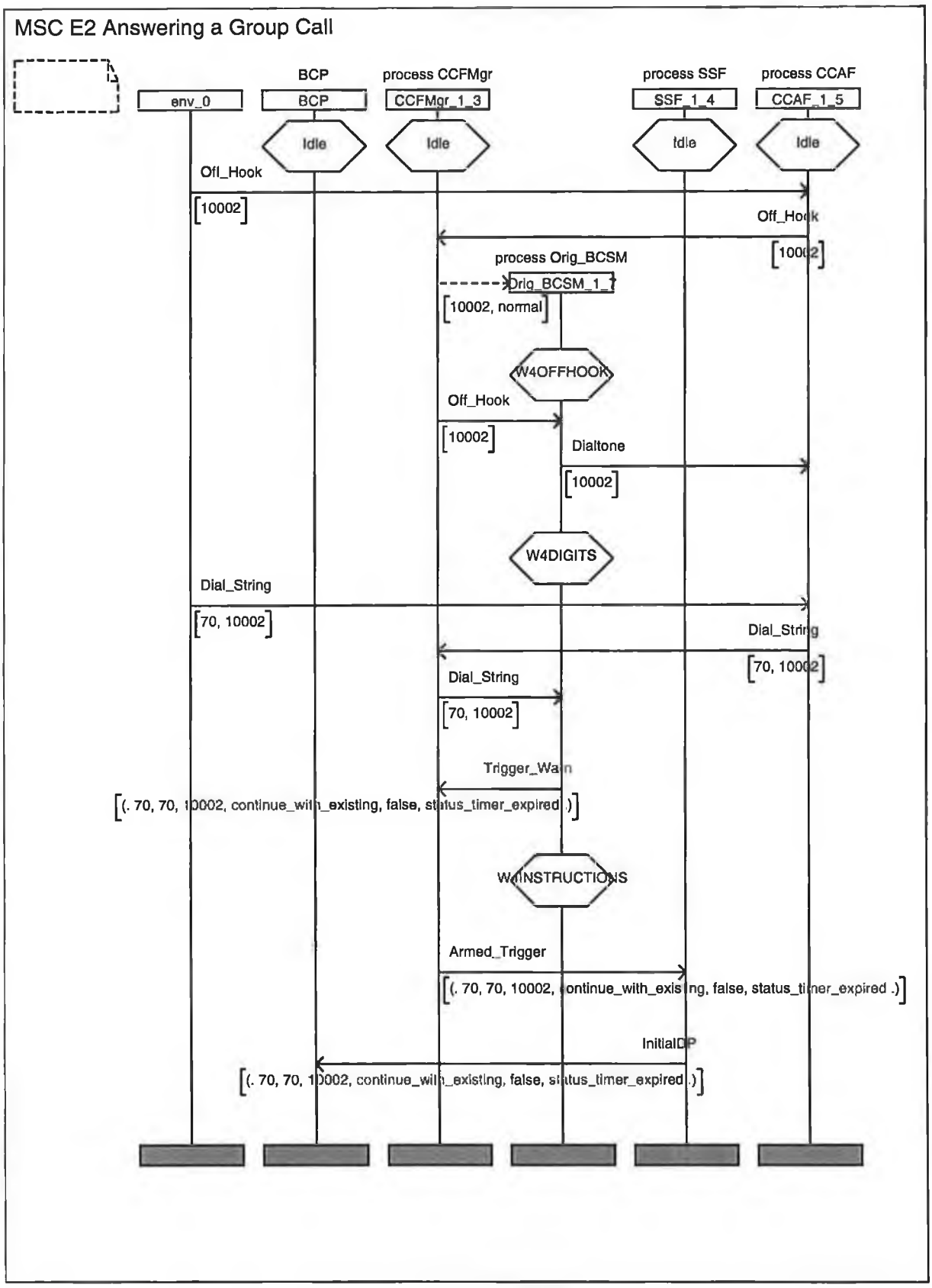

env_0
process CCFMgr
CCFMgr_1_3
process Orig_BCSM
Orig_BCSM_1_7
process Term_BCSM
Term_BCSM_1_8
process SSF
SSF_1_4
process CCAF
CCAF_1_5
process CCAF_2
CCAF_2_1_6
[10002]
W4ANSWER
Off_Hook
[10006]
Off_Hook
[10006]
Off_Hook
[10006]
Idle
Status_Update
[10006, Offhookfollowingalert]
OBCSMADDR
[Orig_BCSM:1]
W4ADDR
OffhookFollowingAlert
ConnectNow
[Term_BCSM:1]
Idle
ConnectNow
[Term_BCSM:1]
W4CONNECTCONF
ConnectNow_conf
Status_Update
[10006, Connected]
Connected
CancelRingingtone
[0]
Status_Update
[10002, Connected]
WhoAreYou_req_ind
W4RESP
WhoAreYou_resp_conf
[10006]
Connectus
[10002, 10006]
Connected
On_Hook
[10006]
On_Hook
[10006]
On_Hook
[10006]
Close_Connection


env_0
process CCFMgr
CCFMgr_1_3
process Orig_BCSM
Orig_BCSM_1_2
process Term_BCSM
Term_BCSM_1_3
process SSF
SSF_1_4
process CCAF
CCAF_1_5
process CCAF_2
CCAF_2_1_6
W4CLOSECONF
Close_Connection_conf
Status_Update
[10002, idle]
W4PUTDOWN
WhoAreYou_req_ind
W4RESP
WhoAreYou_resp_conf
[10002]
W4PUTDOWN
Disconnectus
[10006, 10002]
Status_Update
[10006, idle]
Idle
On_Hook
[10002]
On_Hook
[10002]
Idle
On_Hook
[10002]
Status_Update
[10002, idle]

E.2 Invoking an IN Service

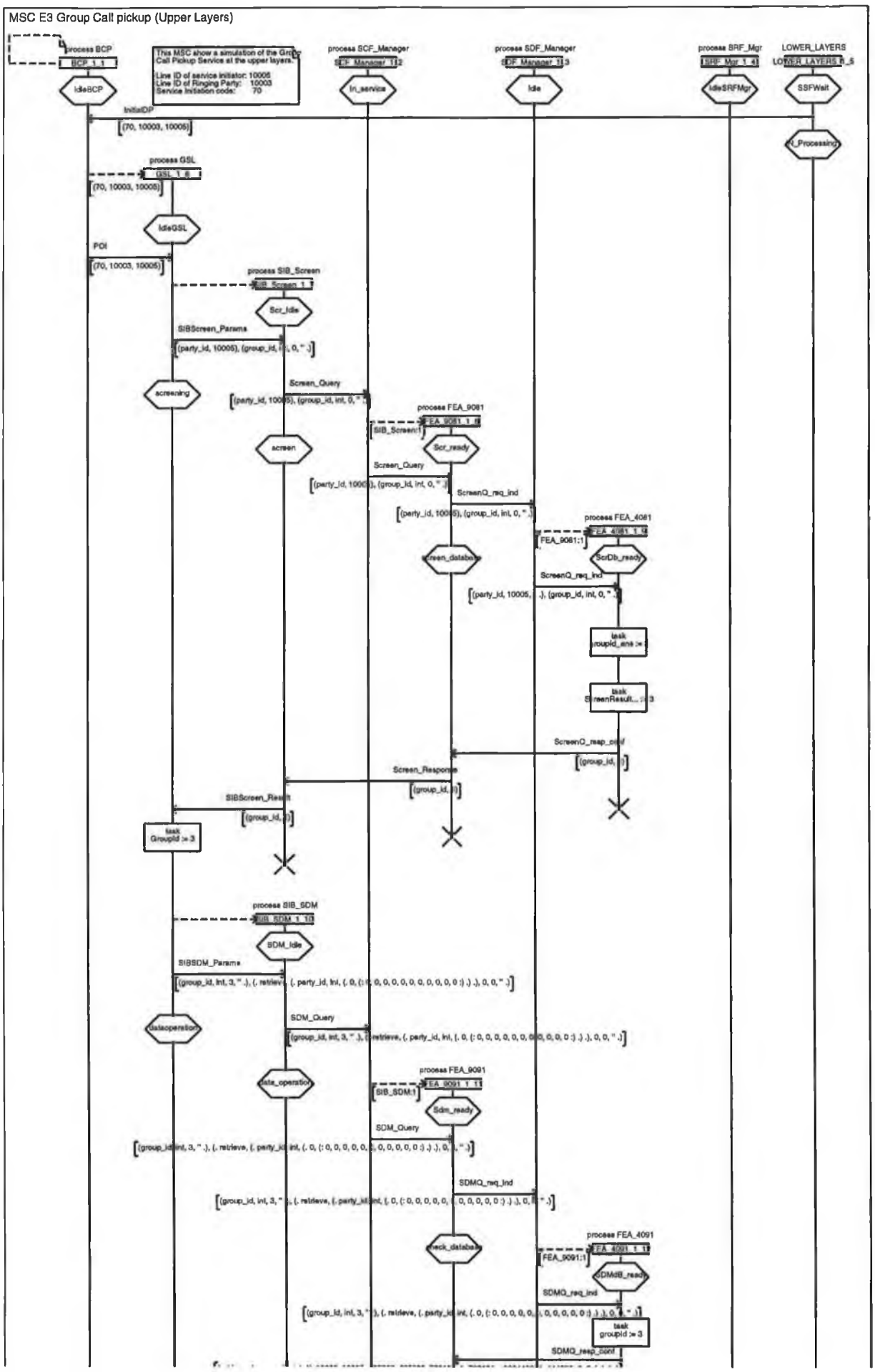
This MSC shows how an IN service is launched from the Basic Call State Manager. A user goes off hook and dials the digits '70' in order to launch the Group Call Pickup service. This scenario might arise when a user is away from their desk but hears their phone ringing. They are at an extension now which is in the same predefined group as their phone and they wish to answer their phone from this other extension.

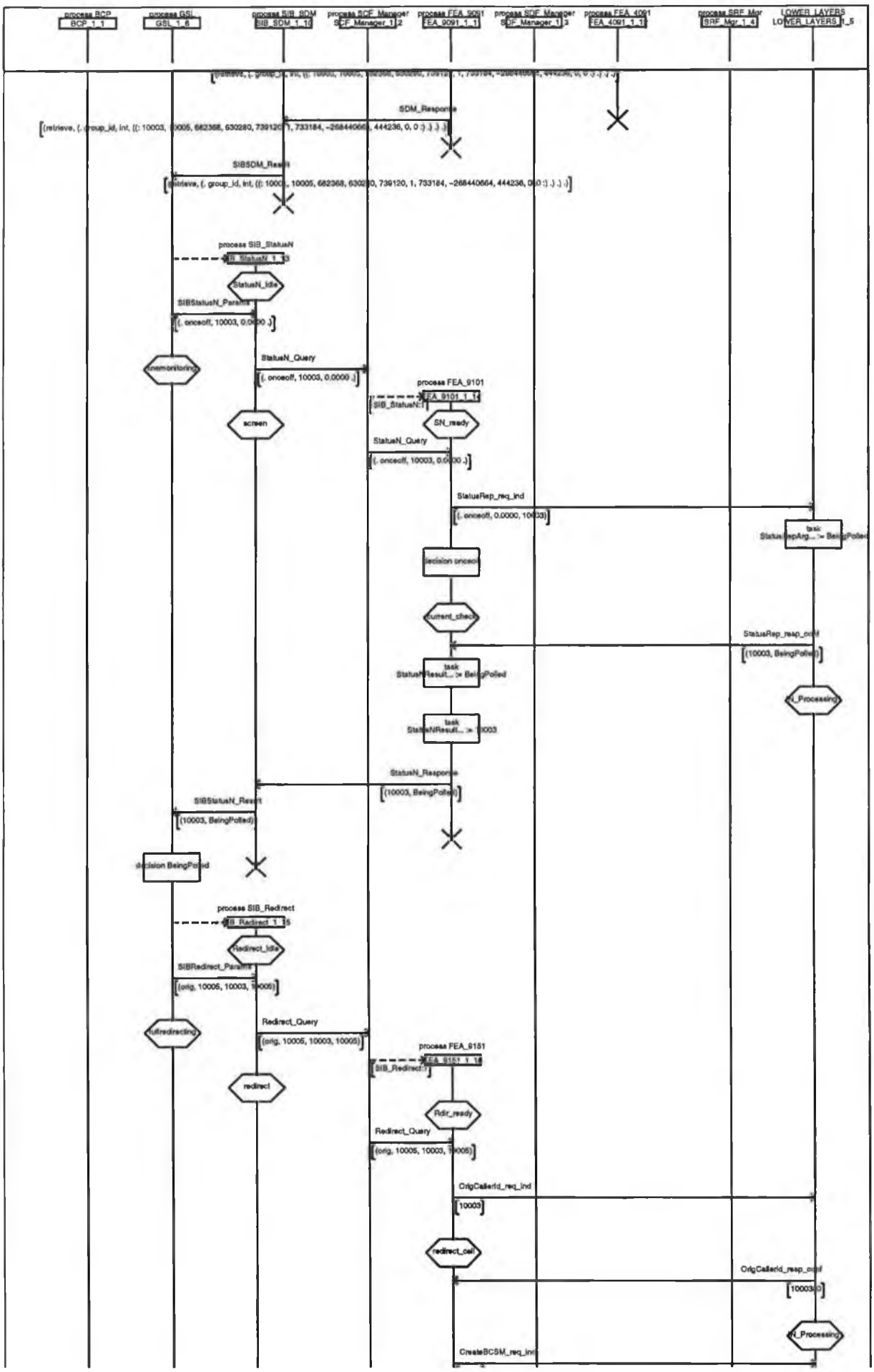
When the basic call state manager detects the digits 70 it knows that it is a service invocation and it sends an Armed Trigger to the SSF from where a detection point, InitialDP, with all the appropriate parameters, is sent to the Basic Call Process in the Global Functional Plane. The detection point can be seen arriving at the BCP in E.3.

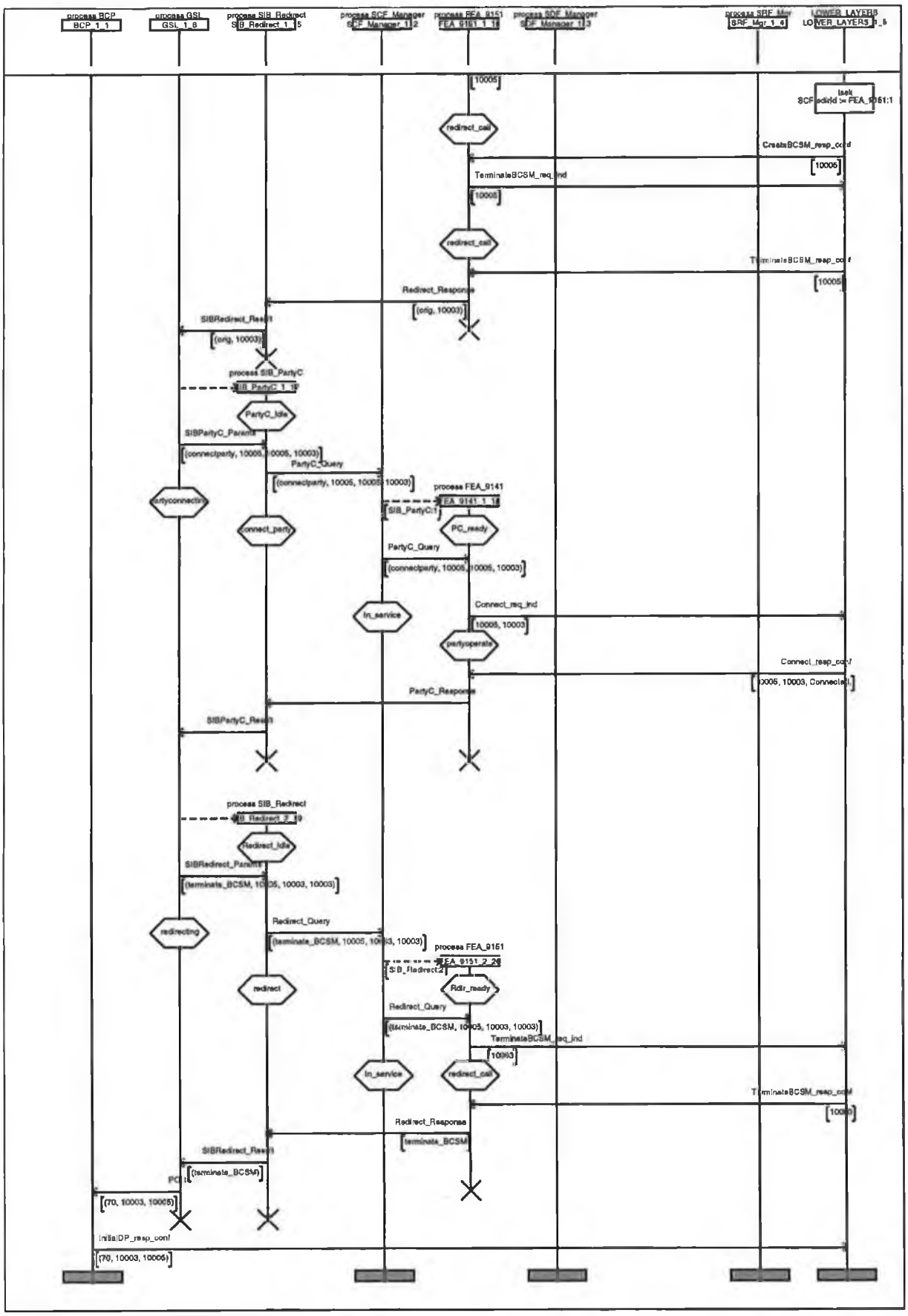

MSC E2 Answering a Group Call
env_0
BCP
BCP
process CCFMgr
CCFMgr_1_3
process SSF
SSF_1_4
process CCAF
CCAF_1_5
Idle
Idle
Idle
Idle
Off_Hook
[10002]
Off_Hook
[10002]
process Orig_BCSM
Orig_BCSM_1_7
[10002, normal]
W4OFFHOOK
Off_Hook
[10002]
Dialtone
[10002]
W4DIGITS
Dial_String
[70, 10002]
Dial_String
[70, 10002]
Dial_String
[70, 10002]
Trigger_Wa
[(. 70, 70, 10002, continue_wi
_existing, false, status_timer_expired .)]
W
INSTRUCTIONS
Armed_Trigger
[(. 70, 70, 10002, continue_with_exis
ng, false, status_timer_expired .)]
InitialDP
[(. 70, 70, 10002, continue_wi
_existing, false, status_timer_expired .)]

E.3 Group Call Pickup Service

This MSC shows the operation of the Group Call Pickup service. For the sake of clarity only the upper service layers are shown. Therefore all the lower level Distributed Functional Plane (DFP) functional entities are represented by the single process line; LOWER LAYERS.

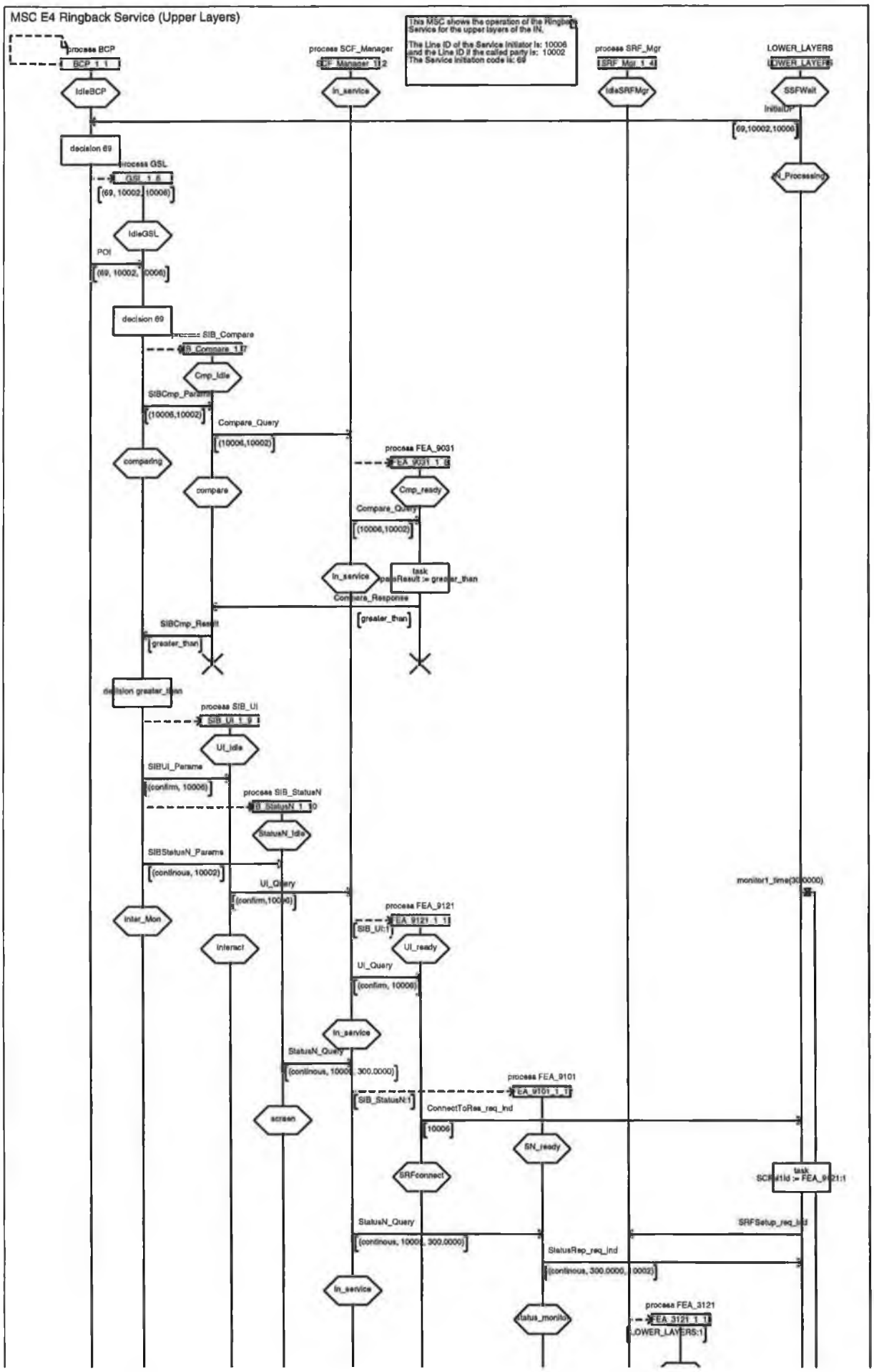

MSC E3 Group Call pickup (Upper Layers)
process BCP
BCP_1_1
This MSC show a simulation of the Group Call Pickup Service at the upper layers.
Line ID of service initiator: 10005
Line ID of Ringing Party: 10003
Service Initiation code: 70
process SCF_Manager
SCF_Manager_1_2
process SDF_Manager
SDF_Manager_1_3
process SRF_Mgr
SRF_Mgr_1_4
LOWER_LAYERS
LOWER_LAYERS_1_5
idleBCP
In_service
Idle
idleSRFMgr
SSFWait
InitialDP
(70, 10003, 10005)
N_Processing
process GSL
GSL_1_6
idleGSL
POI
process SIB_Screen
SIB_Screen_1_7
Scr_Idle
SIBScreen_Params
screening
Screen_Query
process FEA_9081
FEA_9081_1_8
SIB_Screen:1
acreen
Scr_ready
ScreenQ_req_ind
process FEA_4081
FEA_4081_1_9
FEA_9081:1
ScrDb_ready
task
GroupId := 3
Screen_Response
SIBScreen_Result
process SIB_SDM
SIB_SDM_1_10
SDM_Idle
SIBSDM_Params
dataoperation
SDM_Query
process FEA_9091
FEA_9091_1_11
SIB_SDM:1
Sdm_ready
SDMQ_req_ind
process FEA_4091
FEA_4091_1_12
FEA_9091:1
SDMdB_ready
task
groupid := 3

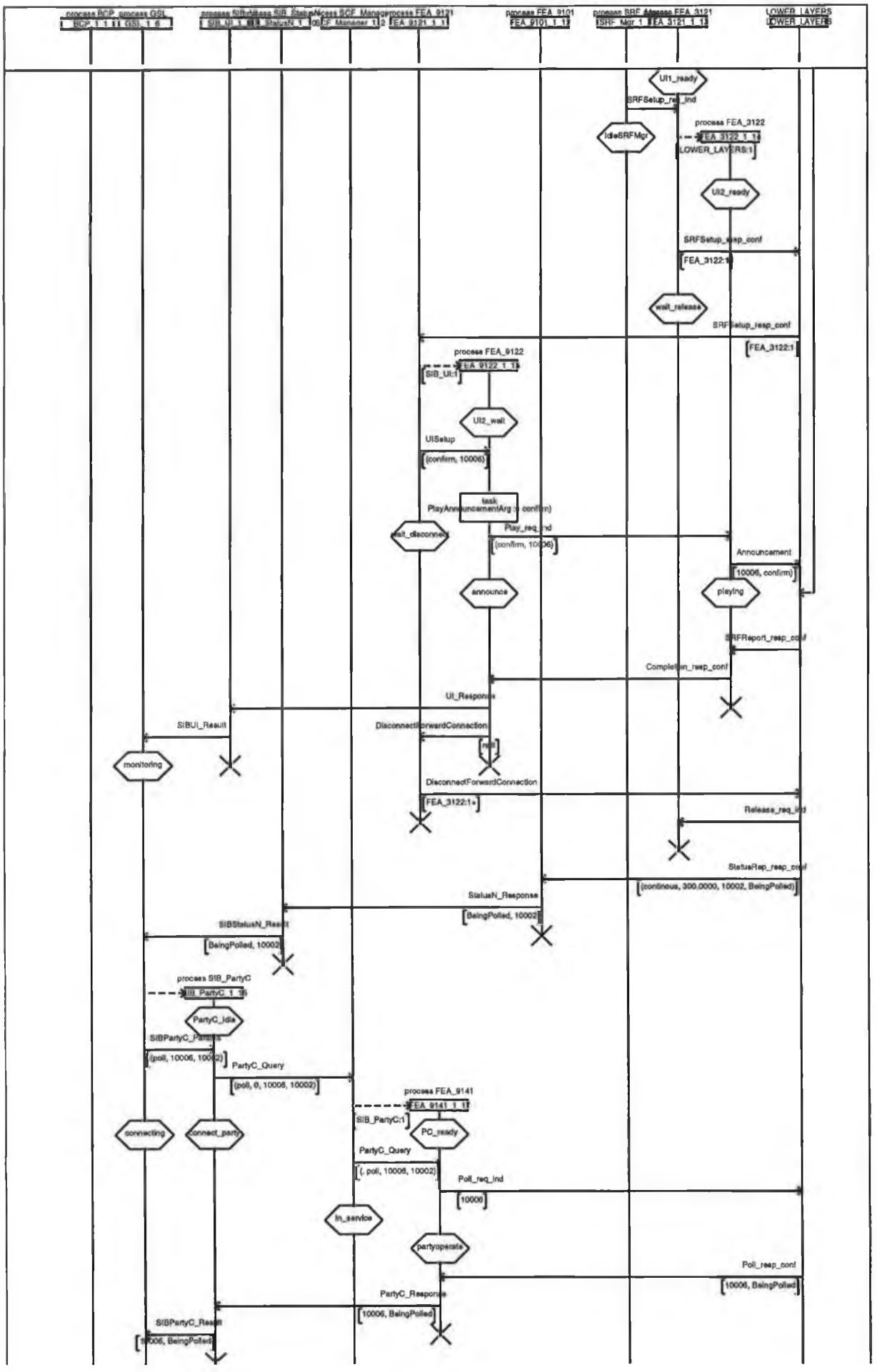

process BCP
BCP_1_1
process GSL
GSL_1_6
process SIB_SDM
SIB_SDM_1_10
process SCF_Manager
SCF_Manager_1_2
process FEA_9091
FEA_9091_1_1
process SDF_Manager
SDF_Manager_1_3
process FEA_4091
FEA_4091_1_1
process SRF_Mgr
SRF_Mgr_1_4
LOWER_LAYERS
LOWER_LAYERS_1_5
SDM_Response
SIBSDM_Result
process SIB_StatusN
StatusN_Idle
SIBStatusN_Params
StatusN_Query
process FEA_9101
screen
SN_ready
StatusN_Query
StatusRep_req_ind
task
Decision oncesoft
current_check
StatusRep_resp_cnf
task
task
StatusN_Response
SIBStatusN_Result
(10003, BeingPolled)
decision BeingPolled
process SIB_Redirect
Redirect_Idle
SIBRedirect_Params
(orig, 10005, 10003, 10005)
fullredirecting
Redirect_Query
(orig, 10005, 10003, 10005)
process FEA_9151
redirect
Rdir_ready
Redirect_Query
OrigCallerId_req_ind
10003
redirect_cell
OrigCallerId_resp_cnf
CreateBCSM_req_ind

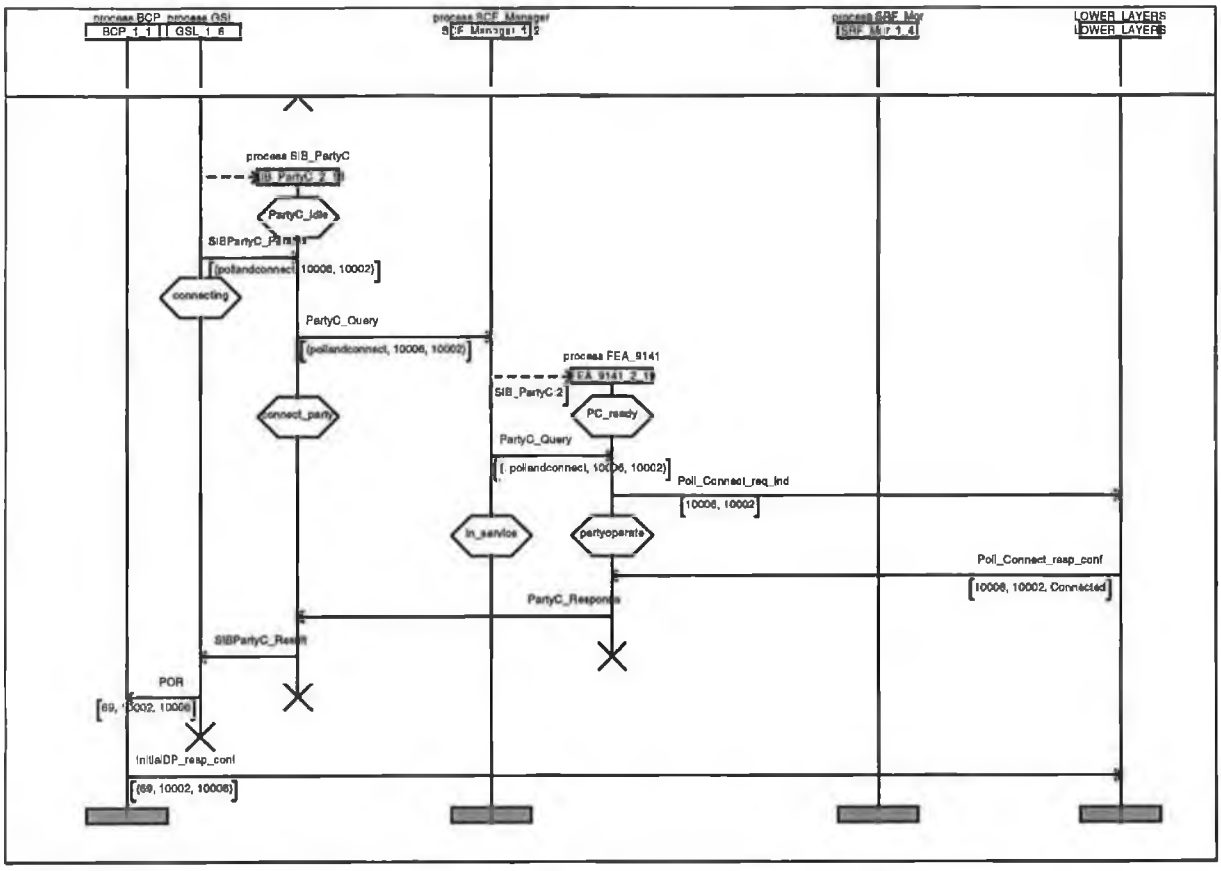

process BCP
BCP_1_1
process GSL
GSL_1_8
process SIB_Redirect
SIB_Redirect_1_5
process SCF_Manager
SCF_Manager_1_2
process FEA_9151
FEA_9151_1_16
process SDF_Manager
SDF_Manager_1_3
process SRF_Mgr
SRF_Mgr_1_4
LOWER_LAYER3
LOWER_LAYER3_1_5
[10005]
redirect_call
CreateBCSM_resp_conf
[10005]
TerminateBCSM_req_ind
[10005]
redirect_call
[10005]
Redirect_Response
[orig, 10003]
SIBRedirect_Result
[orig, 10003]
process SIB_PartyC
SIB_PartyC_1_19
PartyC_Idle
SIBPartyC_Parent
[connectparty, 10005, 10005, 10003]
PartyC_Query
[connectparty, 10005, 10005, 10003]
process FEA_9141
FEA_9141_1_16
SIB_PartyC1
PC_ready
connect_party
PartyC_Query
[connectparty, 10005, 10005, 10003]
In_service
Connect_req_ind
[10005, 10003]
partyoperate
Connect_resp_conf
[10005, 10003, Connected]
PartyC_Response
SIBPartyC_Result
process SIB_Redirect
SIB_Redirect_2_19
Redirect_Idle
SIBRedirect_Parent
[terminate_BCSM, 10005, 10003, 10003]
redirecting
Redirect_Query
[terminate_BCSM, 10005, 10003, 10003]
process FEA_9151
FEA_9151_2_26
SIB_Redirect2
redirect
Rdir_ready
Redirect_Query
[terminate_BCSM, 10005, 10003, 10003]
TerminateBCSM_req_ind
[10003]
In_service
redirect_call
TerminateBCSM_resp_conf
[10003]
Redirect_Response
[terminate_BCSM]
SIBRedirect_Result
[terminate_BCSM]
[70, 10003, 10005]
InitialDP_resp_conf
[70, 10003, 10005]

E.4 Ringback Service

This MSC shows the operation of the Ringback service. For the sake of clarity only the upper service layers are shown. Therefore all the lower level Distributed Functional Plane (DFP) functional entities are represented by the single process line; LOWER LAYERS.


MSC E4 Ringback Service (Upper Layers)
This MSC shows the operation of the Ringback Service for the upper layers of the IN.
The Line ID of the Service Initiator is: 10006 and the Line ID if the called party is: 10002
The Service Initiaton code is: 69
process BCP
BCP_1_1
process SCF_Manager
SCF_Manager_1_2
process SRF_Mgr
SRF_Mgr_1_4
LOWER_LAYERS
LOWER_LAYERS
IdleBCP
In_service
IdleSRFMgr
SSFWait
InitialDP
69,10002,10006
decision 69
process GSL
GSL_1_6
(69, 10002, 10006)
IdleGSL
POI
(69, 10002, 10006)
decision 69
SIB_Compare
SIB_Compare_1_7
Cmp_Idle
SIBCmp_Params
(10006,10002)
Compare_Query
(10006,10002)
process FEA_9031
FEA_9031_1_8
comparing
compare
Cmp_ready
Compare_Query
(10006,10002)
In_service
task
CompResult := greater_than
Compare_Response
(greater_than)
SIBCmp_Result
(greater_than)
decision greater_than
process SIB_UI
SIB_UI_1_9
UI_Idle
SIBUI_Params
(confirm, 10006)
process SIB_StatusN
SIB_StatusN_1_10
StatusN_Idle
SIBStatusN_Params
(continous, 10002)
UI_Query
(confirm,10006)
monitor1_time(3000000)
Inter_Mon
process FEA_9121
FEA_9121_1_11
SIB_UI:1
interact
UI_ready
UI_Query
(confirm, 10006)
In_service
StatusN_Query
(continous, 10002, 300.0000)
process FEA_9101
FEA_9101_1_12
SIB_StatusN:1
screen
ConnectToRes_req_Ind
(10006)
SN_ready
SRFconnect
task
SCIId1Id := FEA_9121:1
StatusN_Query
(continous, 10002, 300.0000)
SRFSetup_req_Ind
StatusRep_req_Ind
(continous, 300.0000, 10002)
In_service
process FEA_3121
FEA_3121_1_13
status_monito
LOWER_LAYERS:1

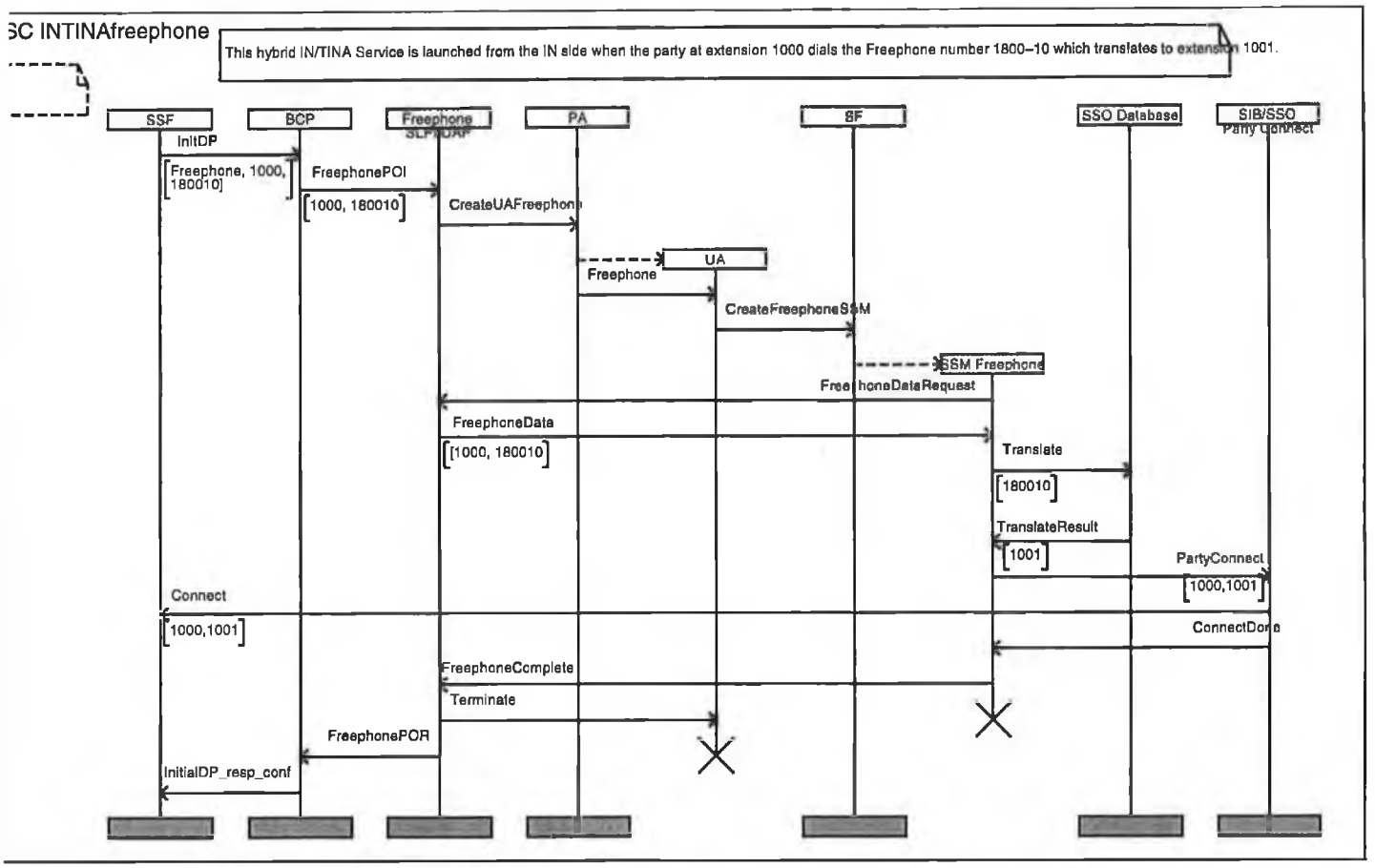

process BCP_1
process GSL_1_6
process SIB_UI_1
process StatusN_1
process SCF_Manager_1_2
process FEA_9121_1_1
process FEA_9101_1_1
process SRF_Manager_1
process FEA_3121_1_1
LOWER_LAYERS
UI1_ready
SRFSetup_req_ind
IdleSRFMgr
process FEA_3122
FEA_3122_1_1
LOWER_LAYERS:1
UI2_ready
SRFSetup_resp_conf
FEA_3122:1
wait_release
SRFSetup_resp_conf
FEA_3122:1
process FEA_9122
FEA_9122_1_1
SIB_UI:1
UI2_wait
UISetup
(confirm, 10006)
task
PlayAnnouncementArg (confirm)
wait_disconnect
Play_req_ind
(confirm, 10006)
Announcement
10006, confirm)
playing
announce
SRFReport_resp_conf
Completion_resp_conf
UI_Response
SIBUI_Result
DisconnectForwardConnection
right
monitoring
DisconnectForwardConnection
FEA_3122:1*
Release_req_ind
StatusRep_resp_conf
(continous, 300.0000, 10002, BeingPolled)
StatusN_Response
BeingPolled, 10002
SIBStatusN_Result
BeingPolled, 10002
process SIB_PartyC
SIB_PartyC_1_35
PartyC_Idle
SIBPartyC_Params
(poll, 10006, 10002)
PartyC_Query
(poll, 0, 10006, 10002)
process FEA_9141
FEA_9141_1_17
SIB_PartyC:1
connecting
connect_party
PC_ready
PartyC_Query
(, poll, 10006, 10002)
Poll_req_ind
10006
In_service
partyoperate
Poll_resp_conf
10006, BeingPolled
PartyC_Response
10006, BeingPolled
SIBPartyC_Result
10006, BeingPolled


process BCP_process GSI
BCP_1_1
GSL_1_6
process SCF_Manager
SCF_Manager_1_2
process SRF_Mgr
SRF_Mgr_1_4
LOWER_LAYERS
LOWER_LAYERS
process SIB_PartyC
SIB_PartyC_2_1
PartyC_Idle
SIBPartyC_Param
[pollandconnect, 10006, 10002)]
connecting
PartyC_Query
[pollandconnect, 10006, 10002)]
process FEA_9141
FEA_9141_2_1
SIB_PartyC_2
PC_ready
connect_party
PartyC_Query
[pollandconnect, 10006, 10002)]
Poll_Connect_req_Ind
[10006, 10002]
in_service
partyoperate
Poll_Connect_resp_conf
[10006, 10002, Connected]
PartyC_Response
SIBPartyC_Result
POR
[69, 10002, 10006]
InitialDP_resp_conf
[69, 10002, 10006]

E.5 Hybrid IN/TINA Freephone Service

This MSC shows the operation of the hybrid IN/TINA Freephone service. This MSC was created using the SDT MSC Editor. It describes how the IN entities and the TINA Computational Objects interact with each other to realise the hybrid service.

The service is launched from the IN side of the Adaptation Unit when a user goes off hook and dials a 1800 freephone number. The Freephone service uses a TINA database (SSO Database) to convert the 1800 number into an extension number. The IN SIB Party Connect is then used to connect the service initiator to the called party.

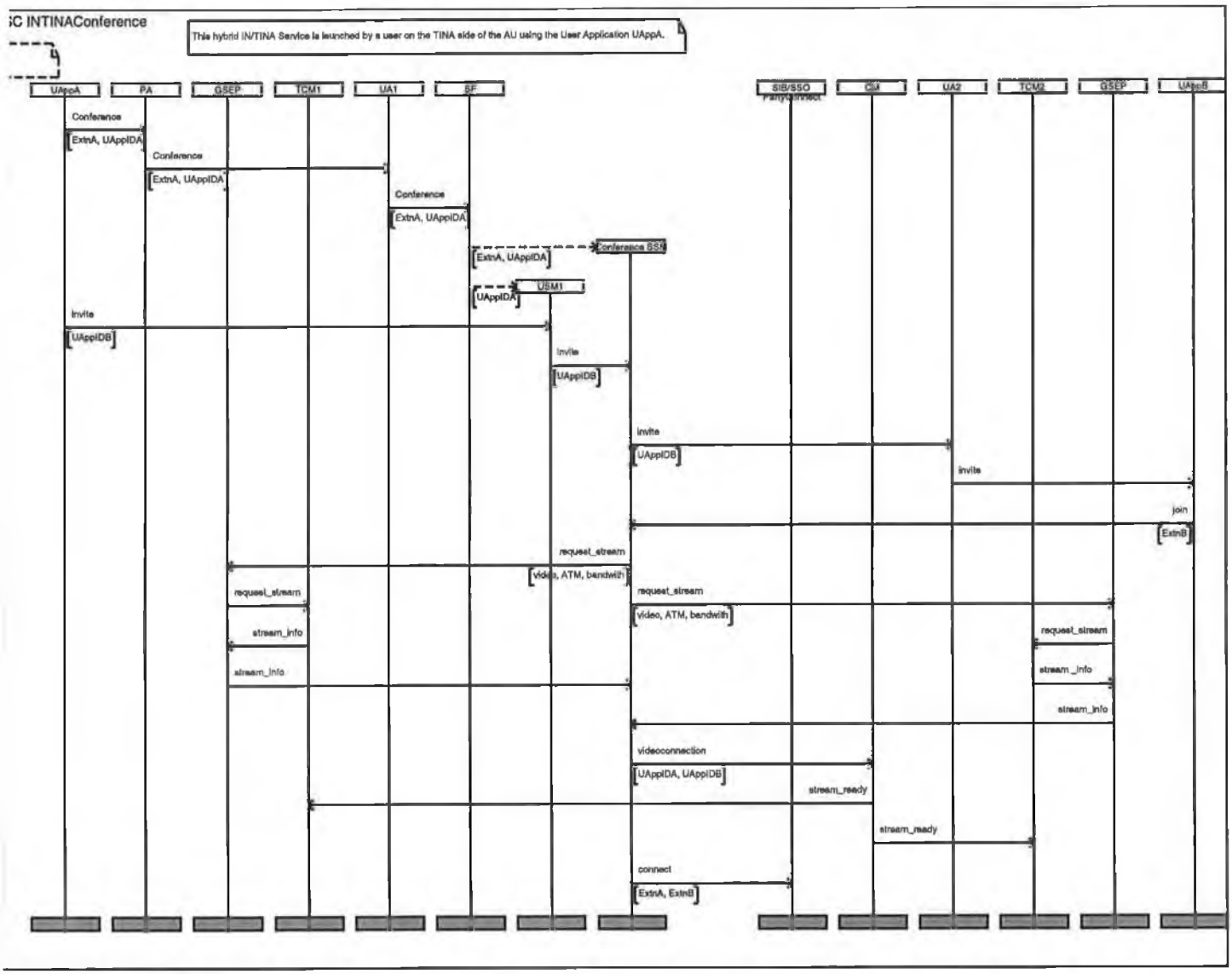

SC INTINAfreephone
This hybrid IN/TINA Service is launched from the IN side when the party at extension 1000 dials the Freephone number 1800–10 which translates to extension 1001.
SSF
BCP
Freephone SLPSOAP
PA
SF
SSO Database
SIB/SSO Party Connect
InitDP
[Freephone, 1000, 180010]
FreephonePOI
[1000, 180010]
CreateUAFreephone
UA
Freephone
CreateFreephoneSSM
SSM Freephone
FreephoneDataRequest
FreephoneData
[1000, 180010]
Translate
[180010]
TranslateResult
[1001]
PartyConnect
[1000,1001]
Connect
[1000,1001]
ConnectDone
FreephoneComplete
Terminate
FreephonePOR
InitialDP_resp_conf

E.6 Hybrid IN/TINA Audio-Video Conferencing Service

This MSC shows the operation of the hybrid IN/TINA Audio-Video Conferencing service. The Service is launched by a user on the TINA side of the Adaptation Unit via a terminal - their User Application, represented by UApp.

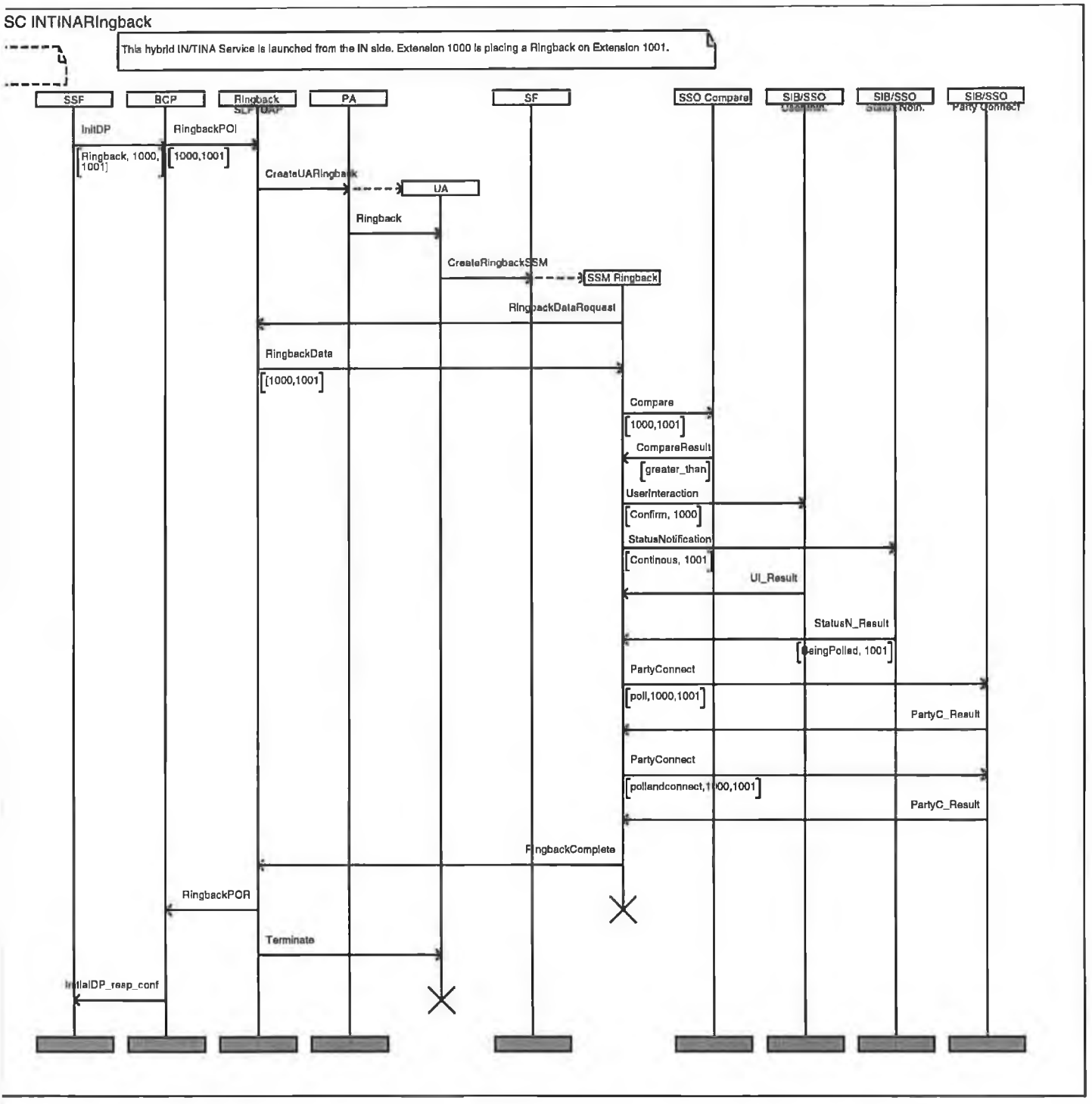
When the called party accepts the conference invitation the Connection Manager is used to set up the video connection between the users terminals.

The IN SIB Party Connect is then used to connect the users in an audio call via their telephone extensions. The Party Connect SIB establishes this connection by communicating with the IN SSF.


C INTINAConference
This hybrid IN/TINA Service is launched by a user on the TINA side of the AU using the User Application UAppA.
UAppA
PA
GSEP
TCM1
UA1
SF
SIB/SSO
CM
UA2
TCM2
GSEP
UAppB
Conference
ExtnA, UApplDA
Conference
ExtnA, UApplDA
Conference
ExtnA, UApplDA
Conference SSM
ExtnA, UApplDA
USM1
UApplDA
invite
UApplDB
invite
UApplDB
invite
UApplDB
invite
join
ExtnB
request_stream
video, ATM, bandwith
request_stream
request_stream
video, ATM, bandwith
request_stream
stream_info
stream_info
stream_info
stream_info
videoconnection
UApplDA, UApplDB
stream_ready
stream_ready
connect
ExtnA, ExtnB

E.7 Hybrid IN/TINA Ringback Service

This MSC shows the operation of the hybrid IN/TINA Ringback service. The Service is launched from the IN side of the Adaptation Unit in the same way as a normal IN Ringback Service is launched. However in this scenario the SLPI launched is for the hybrid Ringback service.


SC INTINARIngback
This hybrid IN/TINA Service is launched from the IN side. Extension 1000 is placing a Ringback on Extension 1001.
SSF
BCP
Ringback
SLP TCAP
PA
SF
SSO Compare
SIB/SSO
SIB/SSO
SIB/SSO
Party Connect
InitDP
RingbackPOI
[Ringback, 1000, 1001]
[1000,1001]
CreateUARingback
UA
Ringback
CreateRingbackSSM
SSM Ringback
RingbackDataRequest
RingbackData
[1000,1001]
Compare
[1000,1001]
CompareResult
[greater_than]
UserInteraction
[Confirm, 1000]
StatusNotification
[Continous, 1001]
UI_Result
StatusN_Result
[BeingPolled, 1001]
PartyConnect
[poll,1000,1001]
PartyC_Result
PartyConnect
[pollandconnect,1000,1001]
PartyC_Result
RingbackComplete
RingbackPOR
Terminate
InitialDP_resp_conf