SPECIFICATION AND VALIDATION OF Q.2931 ATM SIGNALING PROTOCOL USING ESTELLE

by
Dariusz Tasak

School of Computer Science
McGill University, Montreal

September 1997

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

0-612-37170-0

Abstract

ATM, which stands for Asynchronous Transfer Mode, is a networking technology widely considered to be the most promising and efficient method of transporting information through future telecommunication networks. An important part of its definition is a signaling mechanism, which is used to set up and release ATM connections. ITU-T defined a signaling protocol Q.2931, which is designated as an official standard for call control in both public and private versions of the ATM User Network Interface (UNI).

The main goal of this thesis is to specify formally the signaling protocol in Estelle — one of the Formal Description Techniques (FDT). A specification written in an FDT offers great advantages as compared to an informal one. It is unambiguous; it may be used for protocol simulation and validation; and it may also serve as a basis for an actual implementation.

In this study, we design and create a formal specification of the Q.2931 ATM signaling protocol. To demonstrate its signaling functionalities, we develop a simulation model representing the working environment of the protocol. We propose a validation methodology, which we use to show the conformance of our description to the requirements of the official protocol definition. Finally, we present observations and conclusions gathered as results of our experiments.

Résumé

Dans le domaine des télécommunications, on mise beaucoup sur l'efficacité des technologies appelées *Mode de Transfert Asynchrone* (MTA) pour les réseaux de demain. Le mécanisme de signalisation utilisé pour établir et terminer les connexions MTA en est un élément important. L'organisme international ITU-T a défini le protocole de signalisation Q.2931 qui sert de norme officielle pour le controle des appels tant pour les versions publiques que privées des interfaces réseau utilisateur MTA.

L'objectif principal de ma thèse est de définir formellement le protocole de signalisation avec Estelle — l'une des Techniques de Description Formelle (TDF). Une spécification formelle présente de multiples avantages par rapport à une autre non-formelle. La première est sans ambiguité; elle peut être utilisée à des fins de simulation et de contrôle de la validité; elle peut aussi servir de base à une implémentation.

Dans cette étude, nous concevons et créons une spécification formelle du protocole de signalisation Q.2931. Dans le but de démontrer les fonctionalités de signalisation, nous dévelopons un modèle simulant l'environnement d'opération du protocol. Nous proposons une méthodologie de validation que nous utilisons afin de démontrer la conformité de notre description avec les exigences de la définition officielle du protocole. Enfin, nous présentons nos observations et conclusion amalgamées avec les résultats de nos expériences.

To my Parents
and Monika

Moim Rodzicom
oraz Monice

Acknowledgments

Many people in many ways contributed to the creation of this thesis. Now, it is time to thank them all.

I would like to express my gratitude to Professor J.W. Atwood who supervised and guided me in this project. Ideas, experience, and support he provided made my work possible. From the first literature research to the final correction of this document, his availability for discussion and immediate answers to my questions never failed to amaze me. I want to thank him for being there, whenever I needed him.

I also thank my second supervisor Professor Gerald Ratzer. His suggestions and comments helped me to shape my thesis into its current form.

I would like to thank all my friends and colleagues in the School of Computer Science. Particularly, enlightening discussions with Dionis Hristov helped me to understand many issues. Marcia, Souad, Taha, Ioannis, Mark, Bora, Stefan, Max — to name just a few — created a great atmosphere during my two years at McGill.

I also want to thank my friends on both sides of the Ocean. Natalia and Jacek from Montréal made me — a complete stranger at that time — feel here like home. Sławek, Darek, Waldemar, Marcin, Iza, Ela, and all friends in the kayak club "Bystrze" from Kraków proved that 7 000 kilometers does not have to be far for friends. Gavin from Alberta showed me "the other side" of Canada.

Finally, I want to express my deep gratitude to Hans and Eugenia Jütting for their financial support. A fellowship they established made my visit to Canada and studies at McGill possible. I also thank Professor Atwood for the support through his research grant.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

During the last two decades, business, public and personal expectations towards the telecommunication industry became very demanding. Today's market requires increasingly sophisticated, bandwidth-intensive services, which often render existing telephone networks inapplicable. The traditional approach to telephony, *circuit switching*, was developed initially for voice transmission, but failed to deliver satisfactory results after the first computer networks started to exploit existing telephone resources. On the other hand, *packet switching*, used by computers, could not be easily applied for speech, due to the fundamental differences in transfer characteristics. Data transmission requires variable bandwidth and is extremely error-sensitive, but can tolerate delays and does not need any special timing synchronization between a sender and a receiver. Voice transmission needs relatively small and fixed bandwidth and can accept high rate of errors (quality of sound does not need to be perfect), but is very sensitive to delays. The advent of video technology and subsequent need for transport of video streams brought additional postulates: variable and large bandwidths, continuous, error-free transmission, and minimal delays [1].

Asynchronous Transfer Mode, ATM, is believed to address successfully all of the above problems. It integrates various services, satisfies diverse requirements, and homogeneously transports different kinds of traffic on the same lines. Over the years of its development, it has emerged as a leader technology on the market and gained wide recognition in the world of telecommunications.

One may expect that the technology, which aims to satisfy such a broad range of demands, is inherently complex. Indeed, despite quite simple principles, ATM is considered by some experts to be probably the most ambitious and complex undertaking in the history of networking [2]. From the start, it was designed as a global, worldwide solution to be a basis for future broadband telecommunication. As such, ATM must be developed and defined globally to accommodate different needs of all interested parties: governments, providers, vendors, software producers, and customers. Continuous standardization and a strict, meticulous process of defining all vital aspects of ATM allows avoiding many compatibility and heterogeneity problems, typical for early eras of networking. In future, when ATM will span organizations, telephone installations, countries, and continents, there will be no place for proprietary, standalone solutions and time consuming conversions between incompatible protocols or incongruous interfaces [3].

One of many areas of simultaneous research activities concentrates on the creation of a uniform and universal call control mechanism for all ATM users. Such a protocol, known as *higher layer signaling*, *access signaling*, or *layer 3 signaling* is necessary for establishment and termination of data transport connections across ATM. Even though the work is far from over, the first standard of ATM signaling, Q.2931 [4], has been already defined. It derives some basic principles from its narrowband predecessor Q.931 (Digital Subscriber Signaling System No.2)[5] used in the *Integrated Service Digital Network*, ISDN.

Unfortunately, official standard of Q.2931 protocol is defined exclusively in a traditional form, that is, narrative text describing messages and procedures to be implemented. The problem with natural language specifications is that they may contain hard-to-spot errors, ambiguities, and inconsistencies. In case of complex protocols, it may be also very difficult to analyze the protocol and visualize its behaviour directly from the text of the definition.

To depict protocol operations in a more precise and coherent way, one can create a so-called *formal specification* in one of the Formal Description Techniques, FDT.

FDTs are particular kinds of programming languages developed primarily for describing parallel, distributed systems and protocols. They are used to produce an accurate and complete description of system behaviour, while actual realization of this behaviour does not need to be specified (i.e., it is not necessary to solve implementation specific problems). By forcing the designer to follow a strict planning discipline, FDTs can give a better quality, well structured definition with clear separation of different abstraction levels. Moreover, a complete high-level formal description may be used as a starting point for development of real implementations. Since this form of the description is understood by the computer, automatic software tools can be used to assist in the process. FDTs make it possible to move from higher to lower levels of abstraction and gradually refine the initial model. In case of such application, subsequent formal descriptions are increasingly detailed; they embrace more and more implementation specific decisions up to the point where all aspects are accounted for and the implementation is ready [6].

Another motivation for using FDT is simplified validation of the description at each stage of the development. In particular, automatic tools may derive test cases and sequences to verify conformance of the formally defined system to the requirements of the original protocol specification. The designer of a formal definition has also the flexibility of building arbitrary environments to be used as testbeds for simulations and protocol behaviour analysis. Figure 1.1 depicts the idea of such systematic evolution.

1.2 Thesis Contributions

The major goal of this work is to specify formally the ATM higher layer signaling protocol, Q.2931, using the Estelle FDT. We design a formal counterpart of the definition presented in [4, 7] and create a simulation model for the protocol. For communication between the user and signaling ATM functions, we define and realize a simple Application Programmer Interface, API. Even though our model is meant to be a high level description of protocol behaviour, it is "implementation conscious", i.e., planned with possible future evolvement into a real implementation in mind. Finite

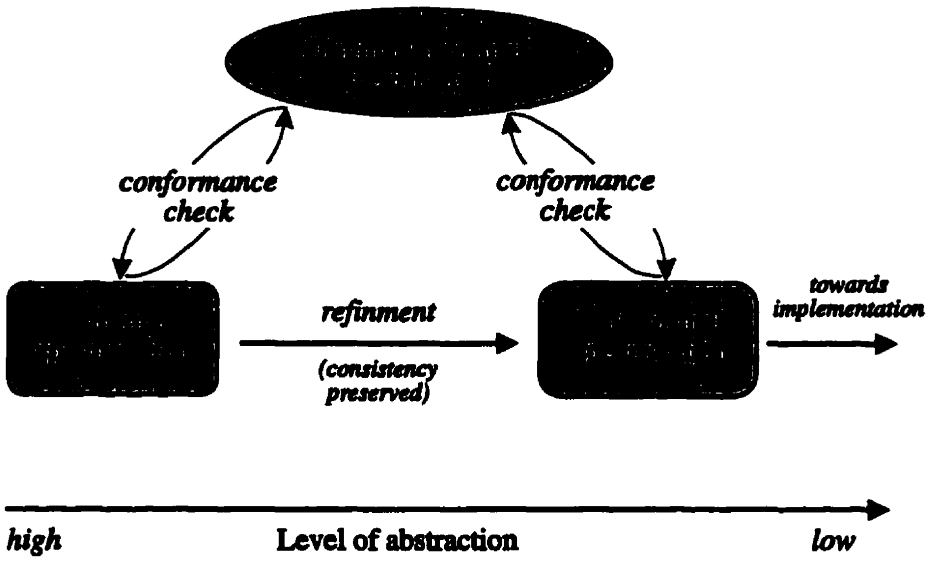


Figure 1.1: Iterative protocol development through subsequent refinements of formal description.

state machines, which represent protocol operation for a single call, are clearly separated from the general purpose and resource management functionalities (common to all connections), hence, our model is expected to accommodate potential changes in official standards.

The second major part of our work concentrates on validation of both our simulation model and the protocol definition itself. We plan the testing approach and methodology to support the claim that Estelle specification truly represents the informal definition of Q.2931. We derive test sequences, cases, and scenarios intended to examine functionalities of ATM signaling. We build test environments (also in Estelle) for integral elements of our protocol model — either each separately or all together. Finally, we conduct experiments and simulations to remove errors from the specification, gather evidence of correct protocol operation, and corroborate its conformance to the requirements of standards.

1.3 Thesis Layout

The remainder of this document is organized as follows:

Chapter 2 is intended for the reader with no background in ATM networks. It introduces basic ATM principles, explains some concepts, and defines the terminology used throughout the document. We want to stress that due to a vast range of important issues in ATM, it is not our goal to provide a complete overview of the technology. We summarize only these aspects which are absolutely necessary for understanding the following chapters. For additional information, the reader may refer to [1, 3, 8, 9, 10, 11].

Chapter 3 gives the reader an insight into ATM signaling mechanisms. We review some fundamental notions of call control in general, and move to explaining particular features of the Q.2931 protocol. All its aspects are categorized and briefly explained.

Chapter 4 introduces Estelle. It summarizes Estelle principles and highlights these elements that are vital for illustrating the features of our specification.

Chapter 5 concentrates on the simulation model created in this work. Presentation is organized in a top-down manner. First, we overview the high level design and explain our approach. Second, we define functions of two additional interfaces that we created to supplement the protocol and facilitate its validation. Third, we describe, in detail, elements that constitute the core of our project: signaling entities of Q.2931. Finally, we present implementation solutions for selected problems in the specification.

Chapter 6 brings the results of protocol validation. We outline general methodology, identify three major steps of the testing process, and accordingly schedule the verification of Q.2931 functions. For each of these steps, testing environment, set-up, and selected experimental outcome are presented. At the end of the chapter, we summarize the results obtained and share some observations gathered during this part of the project.

Chapter 7 puts our efforts in a "broader picture" of some related work.

Chapter 8 concludes this thesis and points out possible directions for further investigations.

Chapter 2

ATM Overview

2.1 Introduction

The ATM concept evolved from research on fast packet switching in early 1980s and was standardized for the first time in 1988 by ITU-T (former CCITT) as the target switching and multiplexing technology for *Broadband Integrated Service Digital Networks* (B-ISDN). The main reason for this designation is that ATM is able to carry all forms of information (images, computer data, voice, and video) in an integrated way. Since different kinds of traffic are treated in the same manner and conveyed transparently through the network, ATM can support not only existing and emerging services, but also the ones yet to come. It provides dynamic bandwidth allocation, thus maximizing resource utilization and lowering complexity of buffer management. Finally, by using short packets, reducing node processing time, and simplifying switching algorithms, ATM supports very high transmission speeds, which are essential for most of today's applications and services. Even though it is not completely free from disadvantages, such as the possibility of packet loss or variable delays, ATM's features make it a feasible and widely deployed technology in wide area, metropolitan, and local networking.

Although it was introduced almost ten years ago, the technology should still be regarded as an on-going project rather than *pret-a-porter* product. There is a substantial amount of research effort carried simultaneously by different groups, standard organizations, and industrial companies. The standard bodies involved in the definition of ATM and B-ISDN are: ITU-T (*International Telecommunication Union*) Study Group 13, ANSI (*American National Standards Institute*) T1S1 Technical Subcommittee, and ETSI (*European Telecommunications Standards Institute*) NA5 Committee. Another important group is the *ATM Forum*, an international consortium of hardware producers, their customers and service providers. The ATM Forum is not an official standards organization; nevertheless, it produces its own specifications, often to address issues not yet included in standards. Clearly, with so many participants and still evolving concepts, proposed solutions are not always completely coherent, even though there is a lot of effort to bring them together and achieve their full interoperability.

2.2 Basic Concepts

2.2.1 Information Transfer

The word *asynchronous* in ATM does not refer to the physical transmission (which may be, in fact, synchronous; for example in SONET/SDH based standards), but rather to the way the bandwidth is divided and distributed among connections. Unlike in some earlier solutions (e.g., time division multiplexing TDM), where the assignment of channels to their correspondent time slot in the frame is fixed, in ATM the user can take any available, empty slot, label it with the identifiers unique for its connection, and fill it with data. In other words, the network does not enforce or quantify in any way the speed of generation of the data by the user; theoretically, it can accept every bitstream as it comes.

ATM operates strictly in a *connection-oriented* mode. Before the user can send anything, the network must establish a connection and allocate necessary resources. Since one of the requirements of B-ISDN is the support of connectionless services, they must be emulated by higher layers of the ATM protocol stack. Both user and control information is transported in small, 53 byte packets called *cells*. Each cell consists of 5 bytes of *header* and 48 bytes of *payload* (information field). The main purpose of the cell header is to identify the cells belonging to the same connection, by means of Virtual Path and Virtual Channel Identifiers. Additionally, the header also includes fields necessary to: distinguish user-data cells from control cells, assign priorities in case some cells must be discarded, perform error control on the header (payload contents is not protected), and, in some cases, introduce flow control. Because there is no cell storage or retransmission in ATM networks, all cells in a given connection maintain their order and cannot be missequenced.

2.2.2 Virtual Connections

ATM has two types of transport connections: *Virtual Paths* and *Virtual Channels*, labeled by their respective identifiers: *VPI/VCI values*. Those values do not have any global meaning; instead, they are translated at each step of routing through the network. A basic element of an ATM connection is called a VC link. Each VC link connects two consecutive points in the network, where the VCI value is translated. A set of sequential VC links with end points and connecting points constitutes a Virtual Channel. A set of Virtual Channels traveling together through the network constitutes a Virtual Path.

Each connection may be unambiguously identified only by both VPI and VCI values, i.e., a given VCI value has a meaning only within its VPI. Because identifiers are translated during routing, which in ATM is rather called *switching*, there is no need for complex algorithms to select globally unique labels. Each routing node (*switch*) picks a locally unassigned pair of values, and, throughout the whole life of a connection, it translates VPI/VCI labels in all incoming cells into the new values (see Figure 2.1). After a connection is released, labels may be reused and translation information is removed from the switch's memory.

Each connection in ATM has *Quality of Service* (QoS) associated with it. It is a set of parameters specifying cell delays, delay variation, and cell loss rate for a given connection. Required QoS may be explicitly requested by the user or implicitly

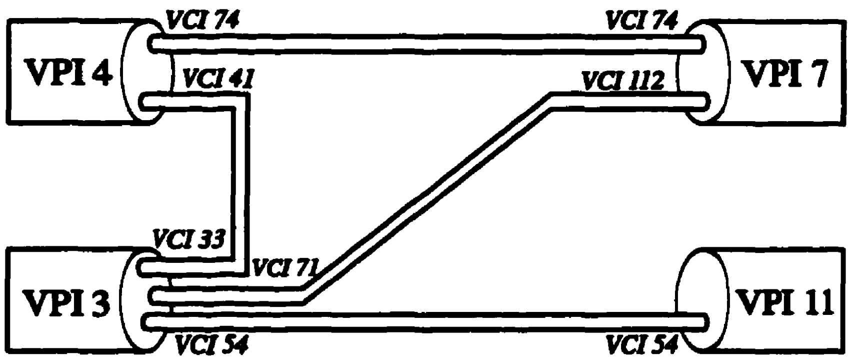


Figure 2.1: Switching of Virtual Paths and Connections.

associated with certain types of connection request. The network is responsible for maintaining the negotiated QoS throughout the duration of the connection.

2.2.3 Protocol Reference Model

The ATM reference model is shown in Figure 2.2. It consists of three main parts:

- *User plane*: transfer of user information,
- *Control plane*: connection control procedures (mainly signaling procedures),
- *Management plane*: network supervision.

The *user plane* consists of layers, as in the ISO OSI model, but their number is limited to 4. We will present them briefly in a top-down direction:

Higher layer of user plane provides support for user services, grouped by ITU-T into four classes: connection-oriented constant bit rate (Class A), connection-oriented variable bit rate with (Class B) or without (Class C) timing requirement, and connectionless variable bit rate services (class D).

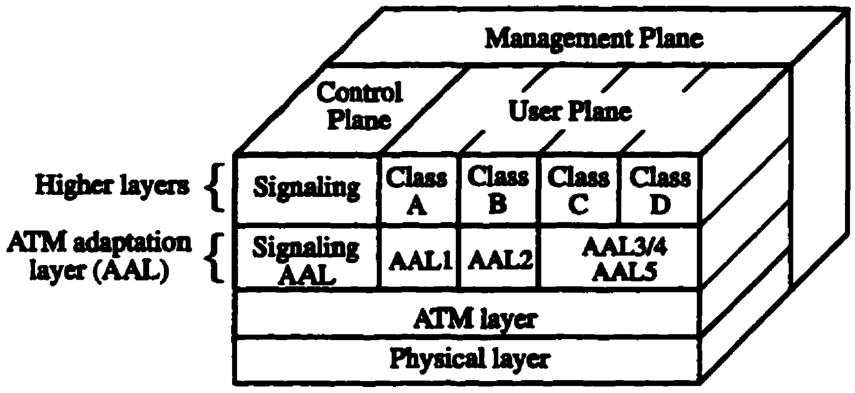


Figure 2.2: ATM protocol reference model.

The ATM Adaptation Layer (AAL) is a service specific layer, whose main purpose is to prepare user data, received from the higher layers, for the transport by the service independent ATM layer. Of course, it also conveys the data retrieved from the network in the opposite direction. AAL consists of two sublayers: CS (*convergence sublayer*) and SAR (*segmentation and reassembly*). CS performs various actions, depending on the requirements of a given class of service. For example, its functions may include error detection and recovery. SAR is responsible for converting the data from the variable length packets used by CS into ATM cells (outgoing data) and from cells back into CS packets (incoming data).

ATM layer is mainly responsible for end-to-end transport of cells. Cells being sent are furnished with headers (produced by ATM layer), while cells being received are stripped from their headers and processed according to the headers' contents. VPI/VCI values are translated here. This layer is completely service independent, that is, data in information field of cells has no meaning whatsoever; it is simply carried.

Physical layer is responsible for transmitting the information across the physical medium. Since ATM network may be implemented over various media (optical fibers, twisted pair), the physical layer will be different in each case.

It is important to note, that there is no direct and simple correspondence of ATM layers to the seven layers of ISO OSI model. A rigid structure of the ATM protocol stack allows layers to be independent from each other, thus achieving modularity and portability (e.g., in case of implementation over different physical media).

The *control plane* also consists of four layers. Since, as we already mentioned, the contents of payload does not matter for ATM layer (and below), both ATM and physical layers are able to carry control information in the same manner as user data and, therefore, they may be shared with the user plane. The control and user cells need to be distinguished no sooner than in higher layers, where the control cells are processed by signaling protocols. Two highest layers of the control plane stack: *signaling AAL* and *higher layer signaling* will be described in more detail in the next chapter.

The *management plane* performs monitoring and supervision of all ATM network operations. It is responsible for failure detection and recovery, collecting and reporting statistics of resource utilization (e.g., for the purpose of billing the users), configuration and maintenance of network elements, determining access privileges, and other security related issues. As shown in Figure 2.2, management plane covers the whole structure of ATM reference model, so its role will be different at each layer and will directly depend on the functionality of the layer.

2.3 Organization of ATM Networks

2.3.1 Classification

ATM network elements can be divided into two general groups:

user terminals : all kind of user equipment that is attached to a network (e.g., computers, IP routers, phone exchanges, phones, fax),

switching nodes : elements, whose primary purpose is to relay the information between nodes and convey it from one user terminal to another. Furthermore, switches may be classified as: private ATM equipment (owned by private organizations) and public switches (owned by service providers and carriers), which are part of the public B-ISDN infrastructure.

Figure 2.3 presents one possible layout of an ATM network organization. As we can see, network elements are interconnected using two different types of interfaces: *User-Network Interface* (UNI) and *Network-Node Interface* (NNI).

2.3.2 User-Network Interface (UNI)

The UNI is used to interconnect any kind of user terminal with any kind of ATM switch. Additionally, it is also used between private ATM networks and public switches.

Depending on the ownership of a switch, two distinct forms of ATM UNI are defined:

- *public UNI*, where the user terminal or private switch is connected to a public service provider network,
- *private UNI*, where the user terminal is connected to a corporate ATM switch, i.e., the organization responsible for the user terminal is also responsible for the switch.

The main difference is the physical configuration of the connection. In most of the cases, private UNI interfaces will be used in places where the ATM switch is located nearby (the same room, floor, building) as the terminals. This configuration does not require any sophisticated, long range physical layer technology. On the other hand, public UNI must meet much higher requirements — terminals may be many kilometers away from the switch. In this situation, it is understood that even though both UNIs share most of the ATM layer stack, they may use completely different physical media.

This thesis is entirely devoted to issues concerning the User-Network Interface. In all remaining chapters, whenever we use term "UNI", we will mean both private and public forms, unless explicitly specified otherwise. Whenever we use term "ATM user" we will mean: "anything connected to the network through UNI". It may be either a terminal, or a private ATM switch.

2.3.3 Network Node Interface (NNI)

The NNI, also known as the *Inter Switching System Interface* (ISSI), is used to interconnect switches within a particular ATM network. As with the UNI, the NNI may be classified, based on the ownership of the switches, as private or public NNI. The ATM user never communicates directly with the NNI, it perceives the network as being one carrier entity. Since the NNI is beyond the scope of this thesis, the interested reader should refer to relevant literature [1, 12, 9].

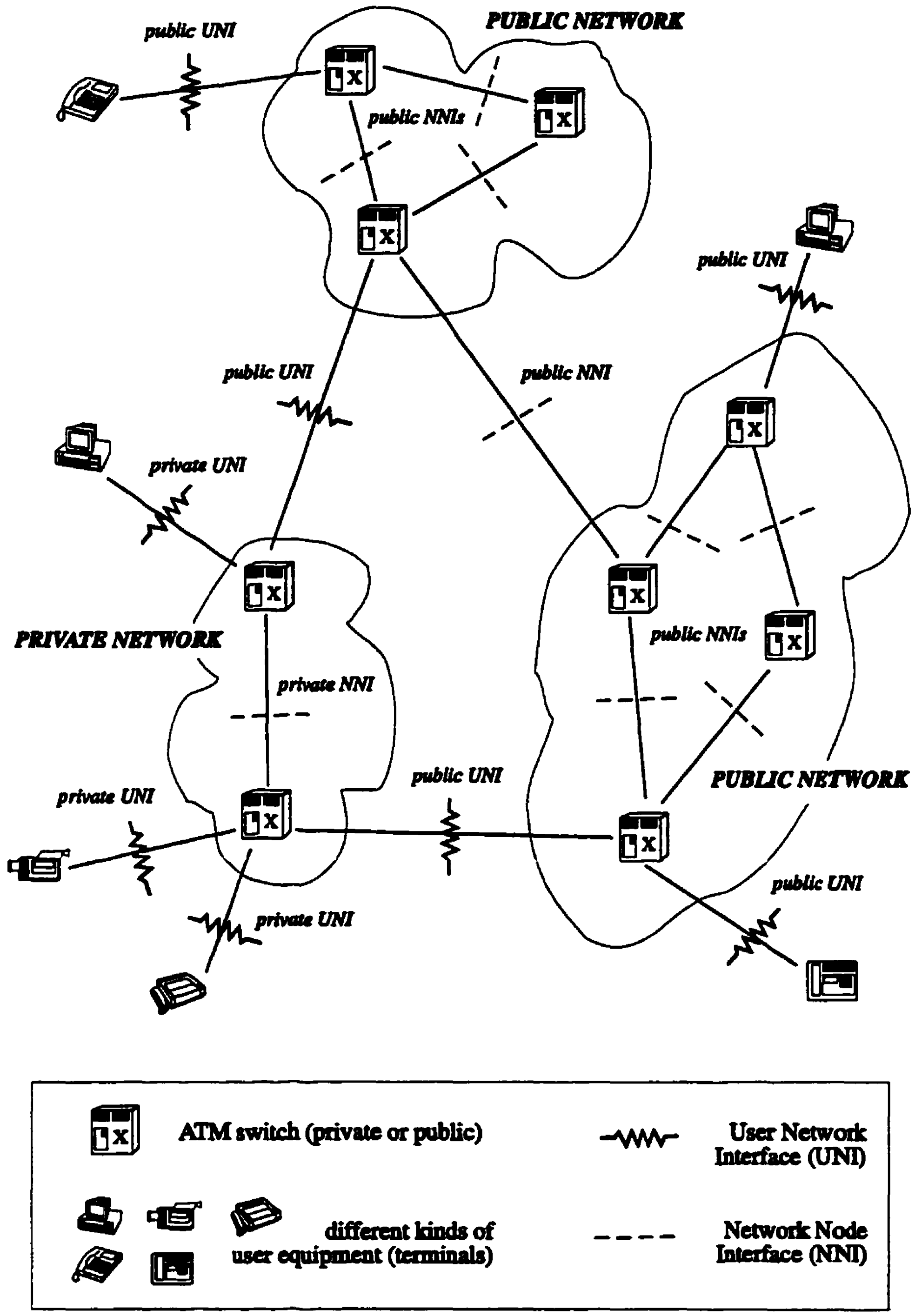


Figure 2.3: Organization of ATM networks.

Chapter 3

Signaling

3.1 Introduction

Signaling in ATM is defined as a set of messages, states and procedures, which allow the user to request from the network that the connection with the specified characteristics (e.g., Quality of Service) be created, maintained, and, finally, torn down after the information transfer is completed. These functions are performed in the two highest layers of the control plane: *Signaling AAL* and *higher layer signaling*, also known as *access signaling*. Moreover, there are two different versions of higher layer signaling: one for the UNI (ITU-T standard *Q.2931*[4]), and the second for the NNI (ITU-T standard *BISUP*[12]). This document deals mainly with higher layer UNI signaling (Q.2931), nevertheless, for clarity of the description, we will briefly present the Signaling AAL as well. It is important to stress that this chapter has only introductory character and covers the most basic issues. Signaling protocols are too complex to be reviewed thoroughly here; their specifications are hundreds of pages long.

Signaling peers in ATM are communicating by exchanging messages on a single, static, out-of-band *signaling virtual channel*. This channel cannot normally be used for any other purpose. Messages are carried through the ATM layer in the form of control cells, in the same way as user data cells. One of the responsibilities of the Signaling AAL layer is to retrieve control cells from the network, reconstruct the original signaling message, and carry it to the higher layer signaling entity for interpretation. Connections established as a result of signaling procedures are called *switched* virtual connections, to distinguish them from the *semi-permanent* virtual connections, which are handled by the network management plane.

In the remainder of this document, we will use terms "connection" and "call" exchangeably, as synonymous to "switched virtual connection" [1].

3.2 Signaling AAL (SAAL)

The SAAL structure is presented in Figure 3.1.

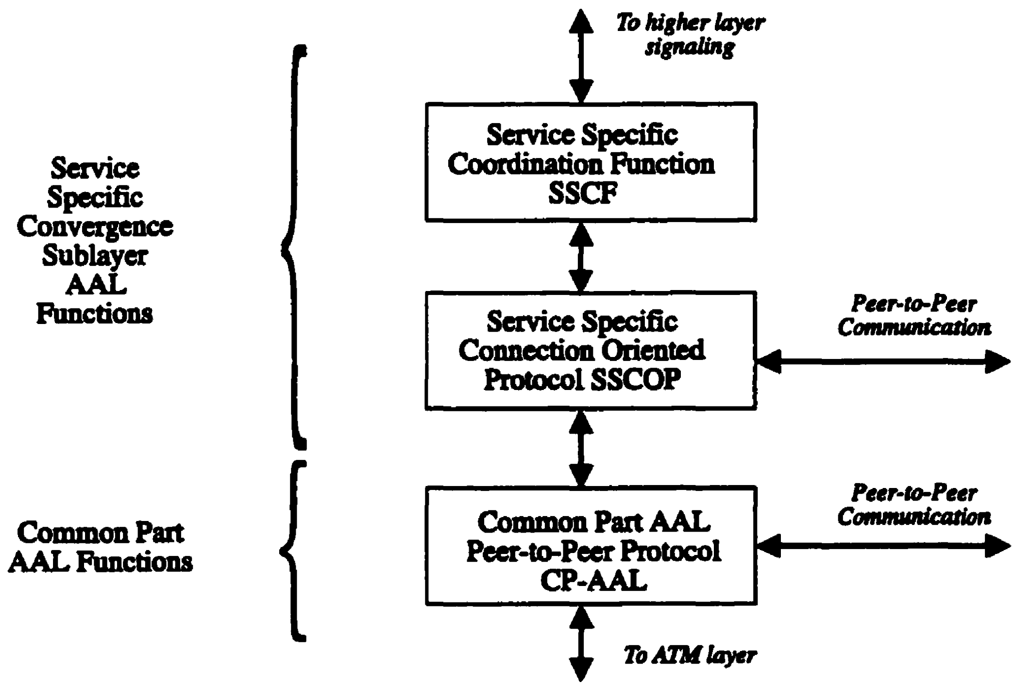

To higher layer signaling
Service Specific Convergence Sublayer AAL Functions
Service Specific Coordination Function SSCF
Service Specific Connection Oriented Protocol SSCOP
Peer-to-Peer Communication
Common Part AAL Functions
Common Part AAL Peer-to-Peer Protocol CP-AAL
Peer-to-Peer Communication
To ATM layer

Figure 3.1: Signaling AAL.

The upper layer of this structure, *Service Specific Coordination Function* (SSCF) [13], is exactly what its name implies: it simply coordinates transport of data from

[1] Even though it is not precise according to ITU-T terminology, it follows the conventions used in ATM Forum specification.

higher layer signaling to the SSCOP protocol below. SSCF does not add any information to the carried messages — it only maps services of SSCOP to the requirements of the higher layer.

Service Specific Connection-Oriented Protocol (SSCOP) [14] is the most important part of SAAL. It provides reliable data transfer between the signaling entities in the UNI (or the NNI), which communicate with SSCOP through their respective SSCFs. SSCOP supports message sequence integrity, error handling (detection, reporting to the management layer, and recovery by retransmission), flow control, connection control, as well as both assured and unassured (unacknowledged) modes of transport. In case of assured connection, which is used by the Q.2931 specification, signaling messages up to 4KB long are protected from loss, reordering and corruption.

3.3 Higher Layer UNI Signaling

3.3.1 Standards

Higher layer signaling in the UNI (access signaling) is specified in two versions: an official ITU-T standard Q.2931 [4] and the ATM Forum UNI specification [7]. Even though the latter is based on Q.2931 and designed to be compatible, some differences exist and the scopes of both documents are not the same. Some of the features supported by one document do not appear in the other and vice versa. For example, procedures for establishing point-to-multipoint connections are already defined by the ATM Forum and not yet by the ITU-T; on the other hand, ATM Forum does not support metasignaling, which is specified in Q.2931. Additionally, some of the messages and call states defined by ITU-T are not used by the ATM Forum [2].

When defining the area of our interest in this work, we decided to choose the common part of the two documents: point-to-point connection control with error handling, status enquiry and restart procedures. They will be described in the following sections. In cases where discrepancies between the two specifications exist, we

[2]The detailed list of differences is in [7].

are using the ATM Forum UNI Signaling 3.1 as a guideline for implementation. The terms: "specification", "Q.2931", and "UNI Signaling 3.1" from now on refer to the common part of specifications and are therefore synonymous.

3.3.2 Messages

Signaling messages are exchanged on a permanently established virtual channel connection, identified by VPI=0, VCI=5. They are sent to SAAL using an assured mode SAAL connection.

The specification defines ten variable length messages, which may be grouped into four classes:

- call setup messages: SETUP, CALL PROCEEDING, CONNECT, and CONNECT ACKNOWLEDGE,
- call tear-down messages: RELEASE and RELEASE COMPLETE,
- call or interface status reporting messages: STATUS ENQUIRY and STATUS,
- call or interface restart messages: RESTART and RESTART ACKNOWLEDGE.

Every message consists of the following parts (information elements):

1. **protocol discriminator**, always coded as: *Q.2931 user-network call control messages,*
2. **call reference** identifies call on the local interface, to which this message refers,
3. **message type** identifies type of the message,
4. **message length** indicates total length of the remainder of the message (in octets),

5. **variable length information elements** carry information pertinent to the call and specific to each message type. Some of these elements are mandatory and some optional.

The information carried in the messages may be of a global (end-to-end) importance, e.g., ATM traffic descriptor, or it may have only local (interface) significance, e.g., call reference identifier, VPI/VCI values. Most of the messages are always associated with the particular call; however, restart and status messages may refer to the whole interface. In such a case, their call reference element is coded as *Global Call Reference* value.

3.3.3 Connection Establishment and Release

In order to understand signaling in UNI, one must realize that on any given interface there must be two signaling entities: one to represent the user and the other one to represent the network. Figure 3.2 depicts a simple example of successful connection establishment and release.

Establishment is initiated when a Q.2931 entity on the user side transfers a SETUP message across the interface to the entity on the network side and enters Call Initiated state (U1). The SETUP message must contain all the information necessary for the network to process the call (e.g., called user address, ATM traffic descriptor, QoS, etc.) Upon the reception of SETUP, the network side of calling interface enters Call Initiated State (N1) and verifies that the resources for a call with required characteristics are available. One of the responsibilities of the network side, at this point, is to select unique VPI/VCI values for connection identification on this interface. If the Q.2931 entity is able to accept a new call, it forwards information through the network to the called user. Additionally, it can optionally notify the calling user about the progress by sending CALL PROCEEDING message (in such case, both sides enter Outgoing Call Proceeding state, N3 and U3 respectively).

After the ATM network delivers the establishment request information to the called interface (which is the responsibility of NNI signaling), the Q.2931 entity on the network side of this interface enters Call Present state (N6) and generates a

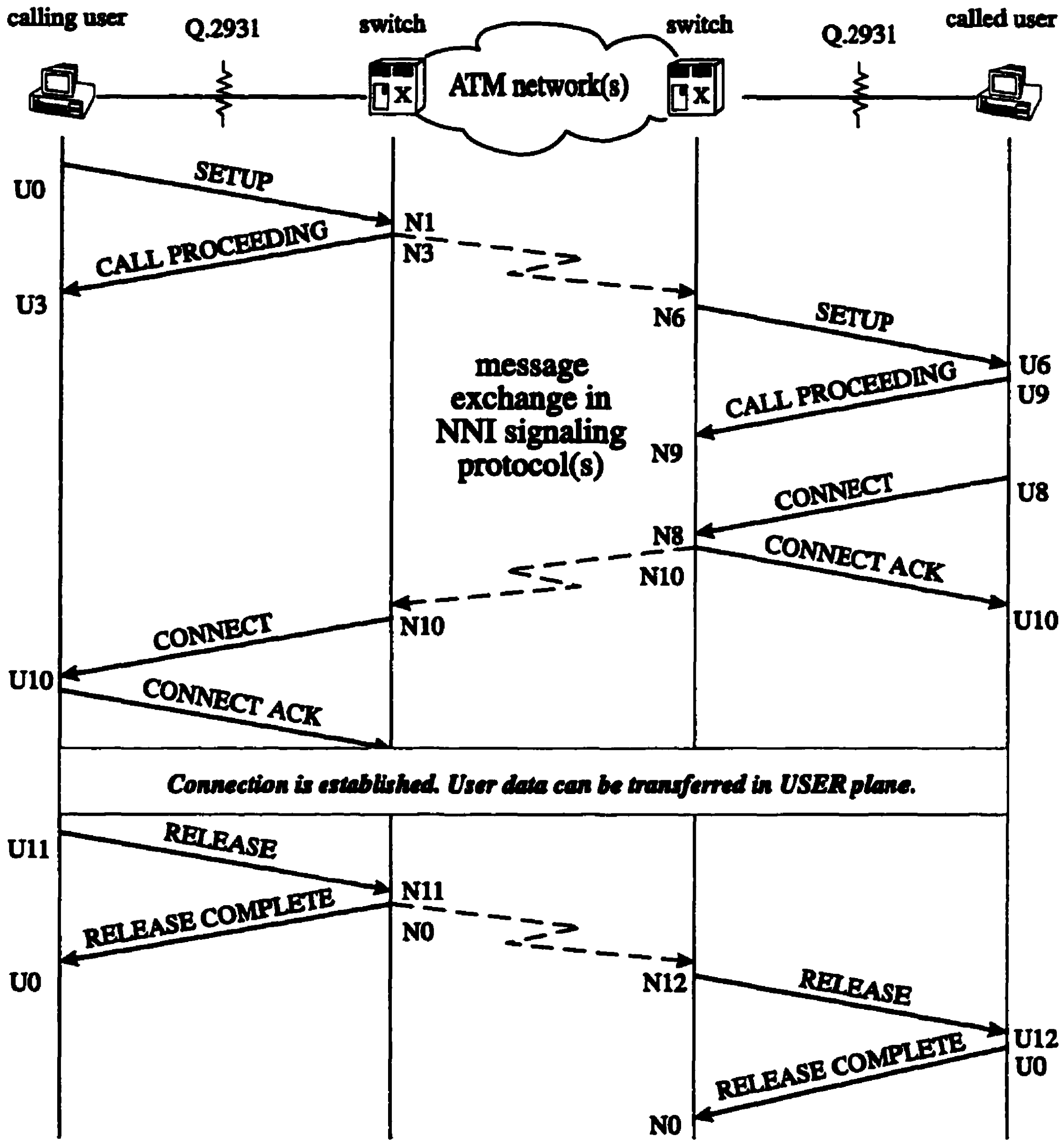


Figure 3.2: Connection establishment and release in Q.2931.

SETUP message across the UNI. It must include in SETUP the VPI/VCI [3] values

[3]Actually, for the purpose of identifying the virtual path, the specification uses Virtual Path Connection Identifier (VPCI) rather than VPI. Since VPI and VPCI are numerically equivalent, we will continue to use more common term VPI/VCI in this document, except in cases where we directly cite from [7].

selected for call identification on this interface. If the user wishes and is able to accept the call, it enters Call Present state (U6), and takes steps to accommodate a new connection. During this process, it has an option to notify the network side with CALL PROCEEDING message, which would result in entering Incoming Call Proceeding state (U9 and N9 respectively) on both sides of the UNI. When the user is ready, it sends a CONNECT message and enters Connect Request U8 state. Upon reception of CONNECT, the network side responds with CONNECT ACKNOWLEDGE and transfers its notification through the network to the calling interface. Both sides of the called interface are in Active state (N10, U10), which means that the connection at this end-point is considered established and ready for the transmission.

After the ATM network delivers the response from the called to the calling interface, the Q.2931 entity on network side sends CONNECT to the user side and enters Active (N10) state. The calling user responds with CONNECT ACKNOWLEDGE, also enters Active state and from now on it perceives the connection to be ready for transmission.

It is important to stress that the actual data transmission takes place in a different plane of the protocol stack (user plane, not control plane) and, therefore, it does not involve signaling in any way.

Connection clearing is initiated when the user of any involved interface (not necessarily the one that established the call) sends a RELEASE message and enters Release Request state (U11). The message must contain a cause information element, which explains the reason for clearing. The network side also enters Release Request state (N11), then responds with RELEASE COMPLETE, forwards information about termination to the remote interface, and finally enters Null (N0). Upon receipt of RELEASE COMPLETE, the terminating user considers the connection to be cleared; no acknowledgment from the other interface is required.

After the network side of the remote interface is notified about clearing, it enters Release Indication state (N12) and sends RELEASE to the user side. Upon acknowledgement by the user with RELEASE COMPLETE, the connection at this end-point ceases to exist and its resources are freed for reuse.

3.3.4 Status Procedures

Status procedures and messages provide signaling entities with additional means to exchange information about the current state of a connection. Status procedures may be used in some error conditions to determine the situation on the interface before recovery actions are taken. For this purpose, two messages are defined: STATUS ENQUIRY, used for soliciting the call state in the peer, and STATUS, used for reporting the local call state to the peer. As a matter of fact, Q.2931 does not specify situations in which STATUS ENQUIRY should be sent. It does, however, define that in response to this message, the receiving entity should return STATUS with the current call state. Additionally, there are a number of situations in error handling procedures, when a STATUS message is sent unasked, to inform the peer about faulty but non-fatal conditions. Neither receiving nor sending of any of these messages can change the call state. Apart from reporting call states, STATUS message with Global Call Reference may be used to report the state of the whole interface, e.g., during the restart procedures.

3.3.5 Error Handling

As the reader might have noticed, our example of connection establishment and clearing presents just one, very simple sequence of events. In reality, setting up the call may be far more complicated. In each step of call processing, entities execute many actions (e.g., allocation of resources, analysis of parameters required by the user, checking for presence and validity of mandatory elements), which may not be successful. Entities often have a limited time to respond to messages; there are timers operating on both sides of the interface and they can sometimes expire. Messages transported across the UNI may get lost or arrive corrupted. Finally, the called user may be too busy, or simply may not wish to accept the connection.

To resolve all such problems, Q.2931 defines a set of actions, which should be taken in various error conditions. We will only present their brief summary here; as a matter of fact, error treatment constitutes a substantial part of the specification and its description takes more space than, for example, call establishment.

The first and sometimes very difficult task of error handling procedures is to recognize and correctly interpret a faulty situation. The next step depends highly on this interpretation. In general, we can distinguish two kinds of error conditions:

Fatal errors. They cannot be tolerated and require immediate termination of a connection. Some examples in this group are: first expiry of non-restartable timer, final expiry of restartable timer, reception of STATUS indicating incompatible state at peer or reception of a message that refers to a non-existing call, while the message type indicates that the call should be in progress. A special kind of fatal error exists when the entity is not able to clear the connection in a normal fashion — restart procedures must be then invoked.

Non-fatal errors. They do not require clearing and permit trying solving the problem. Two main forms of behaviour here are: discarding the message (upon the reception of badly corrupted, unrecognized or non-standard message) or sending the STATUS message (upon the reception of unexpected, out-of-sequence message, message referring to non-existing call, and message with mandatory elements missing or invalid). In the second case, STATUS must contain the cause element, which explains the problem. Sometimes, the entity may decide to process the erroneous message, if it is possible to retrieve all necessary information. It may also happen, that an attempt to recover from an initially non-fatal situation leads to more severe problems and the connection is cleared anyway.

More detailed information about error handling and checking may be found in Chapter 6, where we present simulation results.

3.3.6 Restart Procedures

Restart procedures define special kinds of actions, which are invoked usually as a last resort. They are used to return a virtual channel or all virtual channels on a given interface to the Null state. In general, Q.2931 recommends initiation of restart after any kind of local failure or certain maintenance actions. On the other hand, the documentation clearly specifies only one case when those procedures should be invoked: upon the failure of regular call clearing (final expiry of the timer associated with Call Request state). One must understand that restart is always executed locally on a given interface. It is not possible for UNI signaling to restart virtual channels on the remote interface.

An entity willing to return virtual channel(s) to the Null state, sends a RESTART message to the other side of the interface and starts a timer. Unlike other signaling messages, RESTART does not use call reference of its connection, instead, it is sent with Global Call Reference value. It also contains information about virtual channel(s) to be restarted. Upon reception of RESTART, the peer entity initiates releasing of the call(s) resources and starts its own timer. If the timer expires before calls are brought to Null, restart fails: connection is considered to be *out of order* and the maintenance entity should be informed. Q.2931 cannot use the call(s) or associated resources until some sort of repairs are made. If calls manage to enter Null before the timer expiry, the entity generates a RESTART ACKNOWLEDGE message (also with Global Call Reference) and considers restart a success.

When RESTART ACKNOWLEDGE is received by the initiating side of the interface, it can release resources of its connection(s). If the initiating side does not receive this acknowledgment before the first expiry of its timer, it is allowed to repeat an attempt and resend RESTART. If the timer expires for the last time, the call is considered *out of order*. More information about restart can be found in Chapter 6.

Chapter 4

Estelle FDT

4.1 Introduction

The Estelle language [15] is one of ISO standardized *Formal Description Techniques* (FDTs). It is mainly used, as all FDTs, to specify in a formal way the behaviour and structure of finite state machines, which represent distributed and parallel systems. One of its particular applications is networking protocols and services. A reason for creating formal descriptions is that the traditional form of the protocol definition (i.e., natural language text with state tables, graphs, etc.) usually contains ambiguities and different kind of deficiencies. Not only does the formal description technique enforce a methodological, careful and complete approach to protocol specification, but it also allows using tools to simulate the behaviour of the protocol and verify its conformity to the original requirements. Additionally, even though the unambiguous formal specification should not be mistaken with the actual implementation, Estelle is so close to natural programming languages that it allows the anticipation and solution of many implementation issues (usually not covered by standards).

Technically, the Estelle FDT is based on the syntax of the Pascal language and enhanced by a set of extensions to model finite state machines that are running in parallel and can communicate.

In this chapter, we will present a brief overview of Estelle principles. For exhaustive information, the reader should refer to [15, 16, 17].

4.2 Main Features of Estelle

4.2.1 Specification Structure

An Estelle specification of a protocol (or any other distributed system) is built of elements called *module instances* (briefly: *modules*). All entities, units, tasks or even the specification itself are defined as modules and every kind of activity must take place within one of those modules.

An Estelle module may contain:

1. Other modules. Estelle allows nesting, i.e., each module may have its children,
2. Data structures and Pascal functions/procedures to handle them. They are internal and private parts of the module; however, some data may be shared (but only with a parent),
3. Interaction points: external and internal. They are the communication ports for exchanging information with other modules (siblings, parents, and children),
4. Transitions. This part describes the finite state machine associated with the module and represents its internal behaviour.

Each system described in Estelle has a hierarchical structure, which may be presented as in Figure 4.1. The root of the tree, or the main enclosing box is always a module of the specification itself.

Modules in Estelle may be attributed in one of four ways: systemprocess, systemactivity, process or activity. Attributes divide the system into subtrees of modules called subsystems. A subsystem consists of a module attributed systemprocess or systemactivity (root) and all its descendants (children, grandchildren, etc.). Assignment

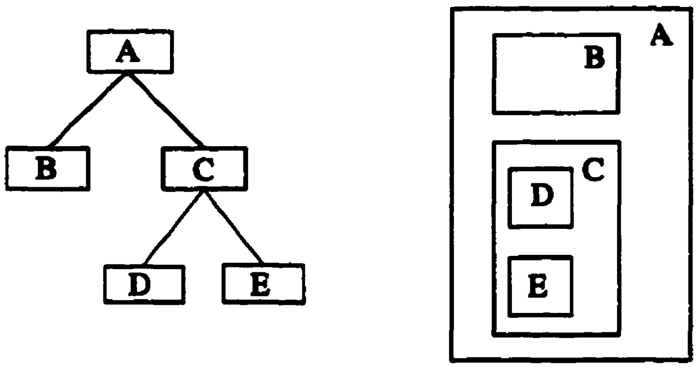


Figure 4.1: Hierarchy of modules in Estelle presented in two equivalent ways.

of the attributes strongly influences the behaviour of the whole system, and, in most cases, it is specific to the described protocol or service.

Apart from the tree-like hierarchy resulting from the parent/child relationship, modules may also be structured from the communication point of view. It will define information flow in the system, i.e., it will decide which modules can exchange messages. Only interaction points of modules may be linked and they can be either attached (if they are external points of the parent and the child) or connected (if both are external points of siblings, both are internal points of the same module, or one is an internal point of the parent and the other is an external point of the child). An example of communication structure is our model of the Q.2931 specification, shown in Chapter 5. All interaction points there are connected.

4.2.2 Communication

Estelle modules communicate most commonly by exchange of messages (called *interactions*). Messages are carried using *channels*: an abstract medium, which links interaction points. For each direction separately, a channel defines the set of messages that it is able to transmit. All messages received in the destination interaction point are inserted to an unbounded FIFO queue, where they wait for processing by transitions. It is the programmer's responsibility to ensure that the module can retrieve all types of interactions that may arrive at any given interaction point. If the message at the head of the queue is not processed, it will stall the interaction point. Only end-to-end communication is possible, which means that if more than two points are involved in the link, intermediate points can neither intercept nor send a message (if sent, it will be lost). Transmission is non-blocking, i.e., it is always possible to send an interaction. Data structures carried by messages are declared as their parameters.

Another communication means in Estelle is sharing of variables, but this method is very limited. Shared variables must be explicitly declared as exported by their module and they will be visible only to the parent. In case of simultaneous access to the variable by both parent and child, the parent's action has priority.

4.2.3 Simulation Process

As already explained, attributing splits the main system into subsystems. Each subsystem executes its own *computation steps* in its own computation time, which means that the behaviour of modules from different subsystems is completely asynchronous. In one computation step, each module in a subsystem chooses one of its transitions, which is ready for execution (*firing*). Then the subsystem executes in parallel these selected transitions and moves to the next computation step.

A transition is like a procedure, which can be activated when its firing condition is met. A transition consists of two parts: clause-group (which defines the situation in which the transition can be fired, i.e., firing condition) and transition-block (set of instructions to be executed).

Clause-group may contain any number of the following clauses:

from *s1, ..., sN*: ready-to-fire if the module is in one of *s1*,.., *sN* states,

when *ip.msg*: ready-to-fire if *msg* is the next interaction to be retrieved from interaction point *ip*,

provided *exp*: ready-to-fire if Boolean expression *exp* is true,

priority *n*: ready-to-fire if there is no other ready transition with a larger value of priority parameter *n*,

delay *(t1, t2)*: ready-to-fire if it stays ready for at least *t1* time and at most *t2* time.

Transitions are atomic; they cannot be interrupted and take no time to execute. After a single computation step is completed, firing conditions of all transitions within the module are reevaluated and new set of ready-to-fire transitions is selected.

4.3 Estelle Development Toolset (EDT)

EDT is a set of tools provided to allow and facilitate the development of implementations for systems specified in Estelle. The most important tools, from the perspective of our work, are: *Estelle Compiler* (Ec) and *Estelle Simulator/Debugger* (Edb).

Ec analyses an Estelle specification to check for any static errors and produces the corresponding source code in C language. Since Estelle is strongly typed language, many kinds of errors may be discovered during this compilation phase.

Edb is an interactive simulator, which allows the user to execute the specification and discover all the run-time and semantic errors. It is a powerful tool; not only does it enable the user to observe and trace the behaviour of the system, but also it allows to control and influence the execution.

Distributed systems are inherently hard to trace. Complexity and parallelism of their components make it difficult to present and visualize the course of action; an excess of trace data may prevent the user from locating the errors. Edb solves the problem by giving the user freedom to customize his scope of interests — both in terms of observed events and control of execution.

Simulation of an Estelle specification is completely interactive. The user can start, advance, break, finish and restart the simulation. He can choose a granularity of the execution by setting the number of transitions to be fired in one turn: from 1 (control is returned to the user after each transition) to infinity (execution advances on its own). He may use macro-commands and Edb scripting language to design his own simulation scenarios. He can also put some timing constraints on the execution, which do not exist in the Estelle language, i.e., he can define both execution time of transactions in a module and system management time (between transactions).

Nondeterminism of Estelle description is achieved by random selection of executable systems, modules, and transitions to be fired in each step of the simulation. However, the user has an option to take manual control of the selection process and, therefore, to enforce the future sequence of events.

Edb allows displaying and storing in trace-files various kinds of information about the on-going simulation. It defines, for this purpose, a large set of functions, which allow obtaining an insight concerning many aspects of the current execution status. Among other things, it is possible to: identify the last module where a transition was fired, identify the transition itself, display its input or output interactions (if any), see the contents of queues, check the state of a module and even the values of its internal variables.

Perhaps the most powerful tool of Edb is the so-called *observer*. It is a user-defined sequence of commands to be executed after each computation step. An observer has the form of a simple script and is capable of using most Edb features, including all predefined functions described above. The user may set observers to define events, messages, modules, and variables to be traced. Observers can also be used to break the simulation after occurrence of an event (either expected or undesired), which may be particularly interesting for the user. This feature is very helpful in tracing rarely occurring errors.

For further details about the tools, we refer the reader to the EDT documentation [18, 19].

Chapter 5

Simulation Model

5.1 High Level Design

A general structure of a simulation model, created as a result of this work, is depicted in Figure 5.1. The Q.2931 protocol is represented by a pair of signaling entities: *UserSignEnt* to control the user side and *NetSignEnt* to control the network side of the UNI.

Figure 5.1 shows two sibling instances of the protocol: on interface A and B respectively. By using names A and B, rather than terms "called" and "calling", we want to stress that those instances are functionally identical (they share the same definition of Estelle modules) and they are equally able to process both incoming and outgoing calls. Any of them can play either the role of the initiating or the remote interface [1].

All the remaining modules are not parts of the Q.2931 specification, but they reconstruct an environment in which the protocol is working and allow verification of its behaviour. *User* modules simulate the activity of users and *Network* represents NNI signaling protocols (e.g., BISUP).

[1]Or both, in case of simultaneous calls simulation; see Chapter 6.

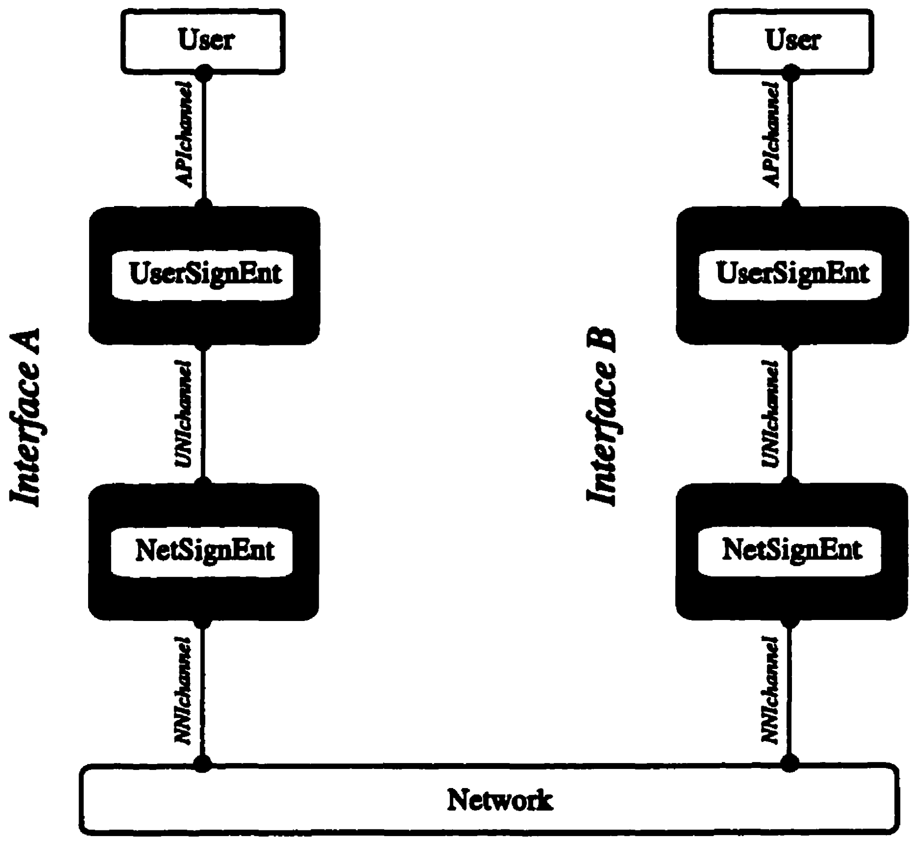


Figure 5.1: General structure of simulation model.

Modules in Figure 5.1 are interconnected by three types of channels: *APIchannel*, *UNIchannel*, and *NNIchannel*. Only UNIchannel is interesting from the perspective of UNI signaling. It is an abstract, bi-directional medium, which transports all ten signaling messages defined by Q.2931 between peer entities. APIchannel, which allows the user to access signaling, and NNIchannel, which simulates the bridge between UNI and NNI, were designed by us for use with this model and they do not represent any factual standards.

For better understanding of how our specification corresponds to ATM reality, in Figure 5.2 we projected one of the interfaces from Figure 5.1 onto the ATM protocol stack. Dark shaded elements are Q.2931 modules, and light shaded ones (User, BISUP) form the simulation environment. UNIchannel, even though it represents

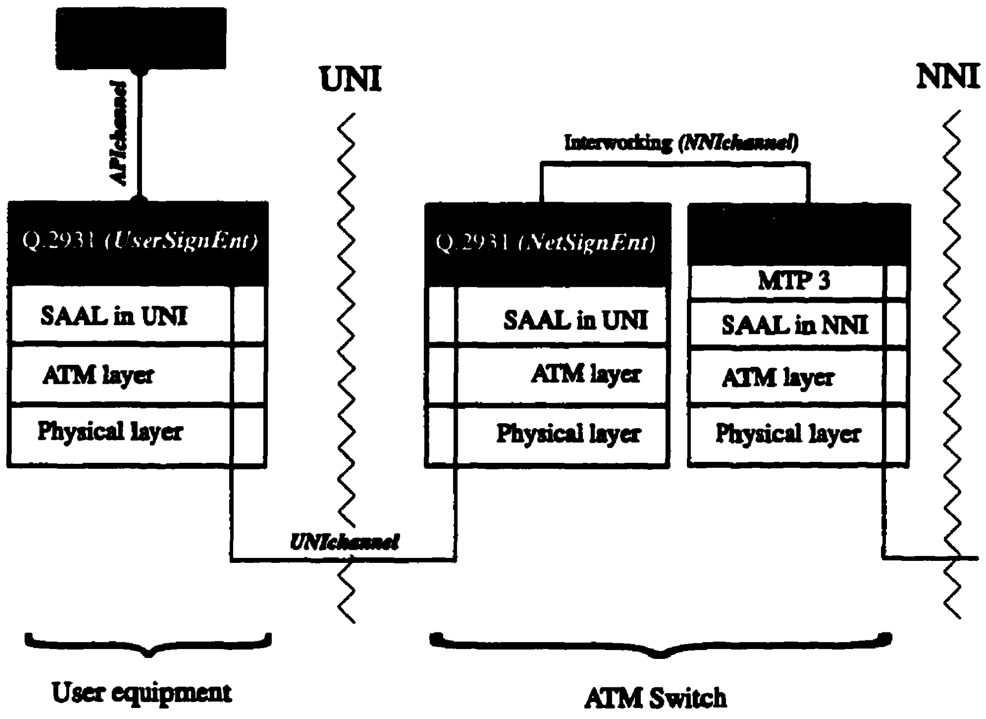


Figure 5.2: One interface from the simulation model placed in the ATM protocol stack.

peer-to-peer communication and directly connects signaling entities, is drawn through all the layers of ATM reference protocol model. It is intended to point out that the information generated by a Q.2931 entity in the form of a signaling message is, in reality, subsequently broken down by lower layers into layer-specific units (e.g., cells in ATM layer), transported over physical media as a bitstream, reassembled accordingly on the other side of the interface, and submitted as a signaling message to the peer.

In the remainder of this chapter, we will present the details of all our design concepts and elements.

5.1.1 User Modules

A set of identical *user* modules represents all kinds of clients (protocols, applications, systems, etc.,) that are using UNI signaling to manipulate their ATM connections.

At any time during the simulation, the user may request the establishment of a connection, up to the total limit of simultaneous calls allowed on the interface. The user is responsible for providing all the information necessary for processing such a request. After being notified about the acceptance, the user may keep the connection active for as long as desired, and then the request for termination will be sent. In our test cases, we were running the scenarios in which either just one user or both of them were granted permission to terminate the connection.

After the signaling entity on the user side of the interface sends an indication that the remote user wishes to set up a new call, the local user has a choice to accept or reject this incoming connection. It is implemented by simultaneous activation of "accepting" and "rejecting" transitions, and the decision is left to the Estelle simulator. Depending on the choice, acknowledgment or request for termination is sent in response. Again, in different scenarios, one or both users could reject or accept the call.

Additionally, a user can receive information about the failed restart procedure, which results in a channel being *out of order*. The user respects the consequences, i.e., does not place requests for the broken channel, but has no means to repair it.

Since there is no officially standardized communication between the user and Q.2931, the documentation does not specify any protection from the faulty behaviour of the user [2]. Therefore, our user modules are implemented to behave in a friendly way: they do not try to cause any protocol malfunction by generating out-of-sequence messages or by not responding.

Messages exchanged between the user and the Q.2931 protocol are defined in Section 5.2.1.

5.1.2 Network Module

The *Network module* is the simplest of all elements in our model, even though it simulates very complex activity. It represents the NNI signaling protocol together

[2]It would be the responsibility of the *Application Programming Interface (API)* definition.

with switching and inter-network routing mechanisms. This part of ATM protocols is normally responsible for the difficult task of finding the route through the network, setting the signaling connections between the switches (within the networks and between them — see Figure 2.3), and delivering information to the remote interface.

In our simulation, the Network module merely carries messages between the Q.2931 entities. The module is equipped with an array of interaction points. Each of the interaction points is associated with the corresponding network interface. Their number is controlled by constant MAXPARTIES, and, in this specification, it is currently set to 2 (point-to-point connection). Network module is able to relay messages of the NNIchannel — described in Section 5.2.2 — between any of the two points, depending on the addressing information provided by the network sides of interfaces. With the current specification supporting only point-to-point connections, the features of this module are not fully exploited. However, they were designed with future point-to-multipoint call support in mind.

5.1.3 Signaling Entities

A pair of Q.2931 entities, *UserSignEnt* and *NetSignEnt*, implement the behaviour of the entire ATM Forum UNI signaling 3.1, in a scope discussed in Chapter 3. A formal specification of these entities constitutes the main goal of this work. The rationale behind splitting the functionality of a protocol into separate modules is that, depending on their location on a physical interface, they have slightly different responsibilities and require different mechanisms to communicate with their neighbors: user modules and network module. Additionally, in a real implementation, NetSignEnt would be a part of a switch and UserSignEnt would reside in the user's equipment; therefore, they could be supplied by two different software vendors.

The signaling entities support multiple simultaneous calls in both directions: outgoing and incoming. The detailed design of the modules is presented in Section 5.3.

5.2 API and NNI Interface Definitions

5.2.1 API Interface

As we already mentioned, the Q.2931 specification does not define what interface should be used between the user and the protocol. It is left as an implementation dependent issue, to be defined by the ATM users according to their needs. Commercial vendors of signaling software packages offer their own versions of such *Application Programming Interfaces*, either general or designed for particular customers (e.g., telecommunication companies).

In order to test the behaviour of our specification, we need such an interface as well. In our case, however, it is not necessary to solve any practical problems that are typical for real applications. We only need a communication scheme that would allow the translation of basic intentions regarding the connection (establishment, acceptance, rejection and termination) into the actions understood by both involved parties: the protocol and the user.

Our design of APIchannel is based on the following messages:

1. messages sent by the user to the Q.2931 protocol:

   callREQUEST — initiation of call establishment signaling procedures. It carries two parameters: *ID*, which identifies the connection, and *data*, which contains call characteristics required by the user: QoS, ATM traffic descriptor, etc. *ID* has only a local meaning for the user and should not be mistaken with the call reference value. *Data* does not need to be coded as *"Q.2931 user-network call control"*, but it uses the same data structure and the call characteristics must be in a form of valid information elements [3]. Additionally, the message length element in *data* must be correct to assure proper processing by the protocol;

[3]Information elements will be copied directly to Q.2931 messages, so if they are not coded properly, signaling error handling will reject them.

callTerminRQST — initiation of call termination. It may be generated for an active connection, which means that the data transfer is over, or as a response to an incoming call indication, which means that the user wishes to reject the call. In either case, apart from the call *ID*, the user shall include *data* parameter with the cause information element indicating the reason for termination;

callStatusRQST — initiation of status enquiry procedures from the user side. In fact, the user's behaviour is not influenced by the result of status enquiry procedures, but the message allows testing of this part of the protocol. This message carries call *ID* only;

2. messages received by the user from the Q.2931 protocol:

callINDICATION — indication that the remote user requested the establishment of a new connection. This message includes call *ID* and *data*, which contains a complete SETUP message received from the network. In the current implementation this information is not used, but in some future work the user may analyze it to make a decision whether to accept the call;

callTerminIND — indication that the connection has been cleared. This message may be received as a response to callTerminRQST (as an acknowledgment) or may arrive unasked (when the protocol or the remote user cleared the call). Apart from *ID*, it includes also *data* parameter with the cause information element provided by the protocol;

callOUTofORD — indication that the call identified by *ID* is *out of order* and shall not be used;

callStatusIND — indication of the result of status enquiry procedure. It includes *ID* and *data*, which contains a complete STATUS message;

3. message sent and received by both the user and the protocol:

callACK — acknowledgment of the connection. If sent from the user to the protocol, this message indicates that the user accepts the call. If sent in the opposite direction, it acknowledges that the call establishment (outgoing or incoming) has been completed and the call is in active state. It contains *ID* and *data*.

In the case of an outgoing call, user initiates the connection by sending callREQUEST. After the Q.2931 completes its establishment procedures and is ready to award the call, it generates a callACK message to the user. At this point, the connection is already in the Active state. If the call cannot be established (procedure failed or the remote user rejected it), callTerminIND is returned by the protocol.

In the case of an incoming call, the user is notified by callINDICATION. It may respond with either callACK or callTerminRQST. If the user sends callACK, the Q.2931 completes connection establishment and also responds with callACK. If the user sends callTerminRQST, the protocol clears the connection and responds with callTerminIND.

When the user wants to clear an already existing connection, it generates callTerminRQST. The protocol starts release procedures and, after they are completed, responds with callTerminIND.

When Q.2931 initiates clearing, it sends callTerminIND. The user shall consider connection released right away and there is no need for an acknowledgment.

5.2.2 NNI Channel and Routing Simulation

NNIchannel, which is meant to simulate the information transfer between UNI and NNI signaling entities, consists of only three messages:

- **NetSETUP**, used to notify the called interface about the connection,
- **NetCONNCT**, used to inform the calling interface that the call has been accepted,
- **NetRLEASE**, used to inform the calling interface that the call has been rejected, or to notify any of the interfaces about call clearing.

All three messages carry two parameters: *data* and *NNITag*.

Data contains information necessary for further connection processing, and it is always a copy of corresponding signaling message (SETUP, CONNCT or RLEASE).

NNITag is an additional parameter, which allows the Network module to find the destination of the message. It is necessary, together with certain bookkeeping mechanism, because — except for SETUP — signaling messages do not contain an address of the called interface. Moreover, the call reference value, which identifies call on the interface, has only local meaning. Without the additional information carried by NNITag, only SETUP message would find its way to the destination, however, it would be impossible to convey a response. Even if there is no problem with the identification of the destination interface [4], received response (e.g., NetCONNCT) must still be assigned to the particular connection, but it contains only the remote user's call reference, which has no meaning to the receiver.

In our implementation, this problem is solved by including the NNITag with the following elements:

- *srcAddr*, an ATM address of the sending interface,
- *srcCallRef*, call reference value used locally by the sender,
- *dstAddr*, an ATM address of the destination interface,
- *dstCallRef*, call reference value used locally by the destination.

When the signaling entity on the network side of the interface sends NetSETUP, it fills three of four fields of NNITag: *srcAddr* (its own address), *dstAddr* (retrieved from mandatory calling party number information element), and *srcCallRef* (local). It is, obviously, not able to fill *dstCallRef* since this value has not yet been assigned (the remote interface does not even know about the call yet). The receiving interface copies all this information into the entry in the so-called *Correspondence Array* and appends its own call reference value allocated for the new call. Correspondence Array

[4]Since in point-to-point signaling there only two interfaces, for each of them "the destination" means simply "the other one".

is used during the remainder of the call as a translation table between call references on the two interfaces; NNItags of sent messages are filled with this data. Initiating interface has a Correspondence Array as well, and it creates its own entry upon the reception of complete NNITag in the first response to NetSETUP [5]. In reality, similar translation mechanism would be executed by NNI signaling, at each step of switching.

ATM addresses used in our specification are coded according to ISO E.164 Integrated Services Digital Network standard and are meaningful for the Network module. As a matter of fact, the Network module does not analyze the whole address, but only one octet of End System Identifier which corresponds to identification of interfaces within the domain [6].

Each interface has a number which is encoded in the 14th byte of an ATM address, and this number is used by the Network module for relaying messages between appropriate interaction points. Since one octet of the address is used, the number of interfaces which can be connected to the Network module is currently limited to 256 [7]. In reality, these issues would be addressed by routing algorithms.

5.3 Signaling Entity Modules Description

The structure of both signaling entities: *UserSignEnt* and *NetSignEnt* is presented in Figure 5.3. Since their overall design is almost the same, we will refer to a single *SignEnt* in the description and, where necessary, we will point out the differences.

SignEnt contains a set of its children modules called Call Control Units or, briefly, CC units. Each of the identical CC units takes care of one single call, from the time it was requested until it is released. Finite state machines for connection processing are implemented in these units.

As seen in Figure 5.3, SignEnt modules communicate using only one UNIchannel which spans their external interaction points: UNI_USignIP and UNI_NSignIP.

[5]Actually, it is not necessary in case of only two interfaces. However, if there are more than two, such information will be required due to the lack of calling party number in RLEASE message.

[6]More about E.164 can be found in Appendix A.

[7]Which seems to be enough for any simulation purposes. If not, it may be easily increased to accommodate up to 6 octets long identifiers.

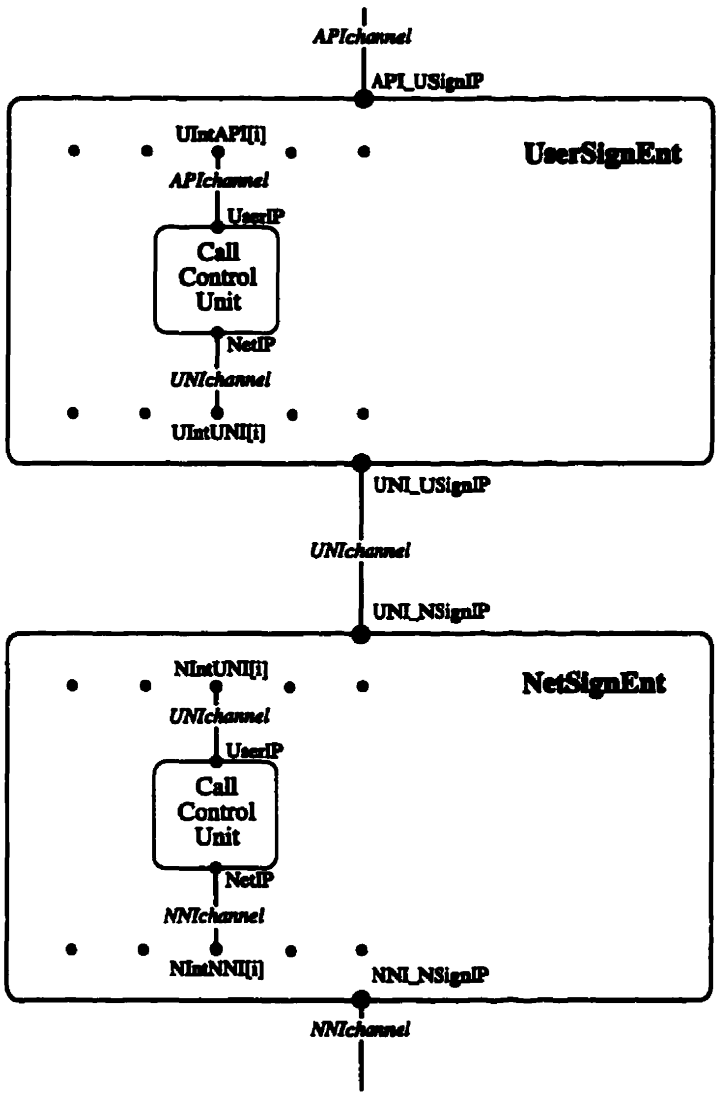


Figure 5.3: Signaling entity modules.

It corresponds to a single signaling virtual connection used by the Q.2931 protocol (VCI=0, VPI=5). Messages exchanged on the UNIchannel have almost the same names as defined in the specification, except that they may be shortened: STATUS_ENQ, RESTART_ACK, or slightly modified: CONNCT, RLEASE [8]. SignEnt

[8]We need to modify them, because both *release* and *connect* are reserved keywords in Estelle. From now on, we will use these modified names in the remainder of the document.

module contains also an array of internal interaction points, used for communication with its CC units.

The main functions performed by SignEnt module are:

1. managing all CC units: dynamic creation of units upon the reception of call requests and their destruction after they are done with call processing,
2. administration and allocation of resources for new calls: assigning available call reference values, selecting VPI/VCI values for a call (NetSignEnt only), book-keeping of connection status (used, empty, restarted, out of order) and maintaining Correspondence Array for the purpose of routing (NetSignEnt only),
3. dispatching messages from the external interaction point to their respective CC units, as well as conveying messages from the units outside,
4. verifying the structure and contents of all messages received through the UNIchannel,
5. conducting most of error recovery procedures associated with format and message contents errors,
6. handling all restart procedures.

The following are short explanations for for the above functions:

Function 1, 2: The first step taken by UserSignEnt after the reception of callREQUEST from the user is an allocation of the call reference value, to be used by all messages relating to this new connection [9]. UserSignEnt inserts this value in a slot of its internal administration database *RefArray*, marks the call as "used" and initializes a new CC unit. Links between external interaction points of CC unit and internal points of SignEnt are created and callREQUEST is passed to the unit for processing. After CC generates SETUP, UserSignEnt conveys it to NetSignEnt. On the other side of the interface, NetSignEnt also needs to allocate the resources. It does not involve call reference (since call already received this value from UserSignEnt) but it

[9]See Section 5.5.5 for further details.

includes allocation of empty slot in RefArray and VPI/VCI values for the connection. If it is successful, NetSignEnt initializes CC unit, links appropriate interaction points and passes SETUP to the unit for processing. A message generated by CC is sent to UserSignEnt. If this is CALL_PROCEEDING or CONNCT, it contains mandatory VPI/VCI values, so that UserSignEnt may insert them into its internal database.

In case of incoming call, the procedure is a little bit different. Since here NetSignEnt is the side which initiates the establishment, it must allocate both call reference and VPI/VCI [10]. Therefore, SETUP received by UserSignEnt contains full information about the connection and no later updates in RefArray are necessary.

When the connection is terminated and CC unit is no longer needed, SignEnt destroys the unit and removes from its databases all the information about the call.

Function 3: Throughout the whole course of connection processing, SignEnt modules pick messages generated by CC units and simply relay them from internal interaction points to the external point (UNI_USignIP or UNI_NSignIP respectively). In the other direction, SignEnt picks a message from its external interaction point, retrieves the call reference information element, locates in RefArray the CC unit associated with this call reference and passes the message.

Function 4, 5: All Q.2931 messages that cross the UNIchannel are subject to the detailed examination described in Section 5.5.3. Messages are tested by the SignEnt module right after they are received. If they fail the verification, SignEnt triggers error recovery. Most of the error handling procedures, except for unexpected message detection, are located in this module.

Function 6: Restart messages, used to return call or calls to Null state, are never forwarded to CC units and are entirely processed by SignEnt. This module is responsible for running restart timers, updating information about the states associated with Global Call Reference (Null, Restart Request, and Restart), blocking forbidden simultaneous restarts, resending RESTART upon timer expiry, responding with acknowledgments, etc. If restart is successful, SignEnt destroys CC unit(s) of involved call(s) and it is the only case when it makes this decision on its own, without waiting

[10]The specification requires that VPI/VCI are always assigned by the network. User side may only reject proposed values, but cannot suggest anything on its own.

for permission from the unit. If restart failed, SignEnt marks call as *out of order* in RefArray and it should notify the maintenance entity. Since we decided, for sake of clarity, not to include maintenance module in our model [11], we substituted this notification with callOUTofORD message sent to the user by UserSignEnt. Restart being local, it does not involve the remote interface, so NetSignEnt does not send anything in this situation.

5.4 Call Control Unit Description

CC unit module is a formal description of a finite state machine for call control procedures defined in ATM UNI signaling documentation. It represents a single ATM connection on a given side of the interface. Units from network and user sides of UNI realize the same functionality, but they do not share the same Estelle definition.

A module communicates through its two external interaction points: UserIP and NetIP. How those points are linked by channels depends on where the module is located (see Figure 5.3). Each CC unit has three exported variables that are shared with SignEnt:

- *CCNr*: identifier of the unit in SignEnt; it is also used as a position of the corresponding entry in RefArray,
- *CCState*: current state of the unit; even though internal call state does not concern SignEnt in general, it is necessary in some error recovery actions (e.g., upon reception of a message with missing elements, SignEnt must respond with STATUS which contains call state),
- Done: Boolean value used to inform SignEnt that call control is completed and the unit may be destroyed.

The main functions of Call Control Unit are:

[11] If included, this module would not do anything anyway, since repairs to be done are not specified.

1. implementing call establishment and release procedures,
2. implementing status procedures,
3. error recovery associated with incorrect behaviour of connection: unexpected messages and non-responding calls,
4. initiating restart upon a failure of release procedures.

Function 1: When the CC unit is notified about a new call, it creates a SETUP message with all the information provided in callREQUEST or NetSETUP. Connection specific data necessary for composing the message (e.g., call reference value, VPI/VCI values) are passed by SignEnt during initialization as a parameter called *context*. In fact, *context* is simply a copy of a corresponding entry from RefArray. After SETUP is sent and call establishment initiated, the CC unit processes incoming messages, generates its responses and changes states in a manner explained in Section 3.3.3. Since messages received by this module successfully passed the test in SignEnt, the unit assumes they are free of errors. As well, all the messages produced here have valid form and contents.

Function 2: Upon reception of callStatusRQST, CC unit on the user side of the interface generates STATUS_ENQ message, without changing its state. Since the specification does not define any kind of "remote status enquiry" (soliciting the status of the remote interface), there is no way for the network side to send such a message. Nevertheless, when STATUS_ENQ is received, both sides are obliged to reply with STATUS. This response does not change the call state. When the STATUS message is received, the CC unit analyses its content. If cause element indicates that it is a response to status enquiry, the user is notified by callStatusIND. If cause element indicates that this STATUS was sent by error recovery procedures, unit initiates call clearing.

Additionally, if cause element does not report any problem, but call state element indicates that the call state on the other side is Null, the CC unit also goes to Null immediately and sets its Done variable to true, in order to let SignEnt destroy the unit.

Function 3: CC unit module provides a set of traps: transitions intended to intercept all unexpected (out-of-sequence) messages. A trap is activated if a message is received that should never arrive in a given state. Traps do not change a call state, but they generate a STATUS message with the cause element coded as *"message incompatible with the call state"*.

Call Control Unit is responsible for running the timers associated with the call (except restart timers), resending the messages upon the timer expiry (if allowed), and terminating non-responding connections.

CC module is also able to detect lack of mandatory cause element in RLEASE. Such messages are processed normally as if they were correct, but RLEASE_COMPLETE sent in response contains the cause element coded as *"mandatory element missing"*. This is the only situation when handling of the message contents error is done by Call Control Unit; normally, such procedures are located in SignEnt modules.

Function 4: In cases when call clearing fails, i.e., the timer associated with release procedures expires for the last time, CC unit module generates RESTART. It does not carry on restart procedures, however. They are conducted by SignEnt.

5.5 Implementation Issues

5.5.1 Resolution of Deficiencies in Specification

One of the main advantages of formal specification is that it requires the designer to define the protocol in an unambiguous way, without deficiencies and uncertainties typical for the traditional forms of description. Even though, in our opinion, Q.2931 is a carefully and quite coherently defined protocol, we needed to resolve some of the issues not (or not clearly enough) addressed by the documentation. Examples of such problems and their solutions are given below. Sections and pages cited here refer to the ATM Forum UNI Specification version 3.1 [7].

Deficiency: It is defined that "... when both sides of the interface initiate simultaneous restart requests, they shall be handled independently. In the case when the same virtual channel(s) are specified, they shall not be considered free for reuse until all the relevant restart procedures are completed." (Section 5.5.5, page 251)

When there are parallel incoming and outgoing restarts for the same channel, one of the independent procedures might be successful and the other not. What should be done by successfully completed incoming restart in case when failed outgoing restart finished first and already marked the channel as *out of order*? Should this decision be changed? Should RESTART_ACK be sent?

Solution: We decided, that the intention of *out of order* link marking is to require an external intervention of maintenance entity and that Q.2931 cannot in any way impact this process. Therefore, a channel once considered broken cannot be reused even if the incoming restart procedure succeeded in bringing it to Null. No acknowledgment is sent (we do not want the other side to consider this channel to be restarted).

Deficiency: It is not specified what should be done after the arrival of RESTART specifying a channel that is marked as *out of order*.

Solution: For the same reasons as above, we decided not to send an acknowledgment. After several failed attempts of restarting the channel, the other side will have no choice but to mark this channel as *out of order*, too.

Deficiency: After the reception of RESTART specifying that "all channels controlled by the layer 3 entity" shall be restarted, should acknowledgment be sent if some of the channels were previously *out of order* and, therefore, were not restarted with all others?

Solution: We chose not to consider broken channels as included in restart request (since it is beyond the means of the protocol). RESTART_ACK will be sent if all working channels at the time of arrival were successfully restarted, even if it does not mean that all channels controlled by SignEnt are operational.

Deficiency: It is defined, that when the destination interface is notified about the call establishment " ... [SETUP message to the called user] is sent by the network only if resources for the call are available; otherwise, the call is cleared toward the calling user" (Section 5.5.2.1, page 246). Cause for this clearing is not specified.

Solution: We chose to use cause #47: "Resources unavailable, unspecified."

Deficiency: It is specified, that the timer T322 (associated with sending STATUS_ENQ message) shall be stopped upon the reception of any clearing message (page 259). It is not clear, what should be done if the clearing message is not received, but *sent* by the same entity which generated STATUS_ENQ. We may have such a situation if there is another timer running simultaneously on this side (e.g., T310 associated with Outgoing Call Proceeding state) and this timer expires first.

Solution: We decided to stop the T322 timer in any case of clearing: either outgoing or incoming. Otherwise, we would have second RLEASE sent after T322 expires, while the call is already in Release Request state.

Deficiency: It is specified, that both RESTART and RESTART_ACK messages shall be used with Global Call Reference values. The specification does not define any error handling actions, if one of these messages is received indicating different call reference.

Solution: If the message is otherwise correct and contains all necessary information, we decided to ignore the call reference value and processs the message as if it arrived with Global Call Reference.

5.5.2 Message Structure

The general message structure is the same for all messages used in our implementation and is presented in Figure 5.4. It has been designed to reflect very closely the layout of the original signaling messages.

The basic elements of the message are *octets*, which are grouped together to contain information elements specified by the Q.2931 coding standards. Each signaling message has four mandatory, fixed length elements: *Protocol discriminator*, *Call reference* (CallRefL is length and CallRefV..CallRefV3 contain value), *Message type* and

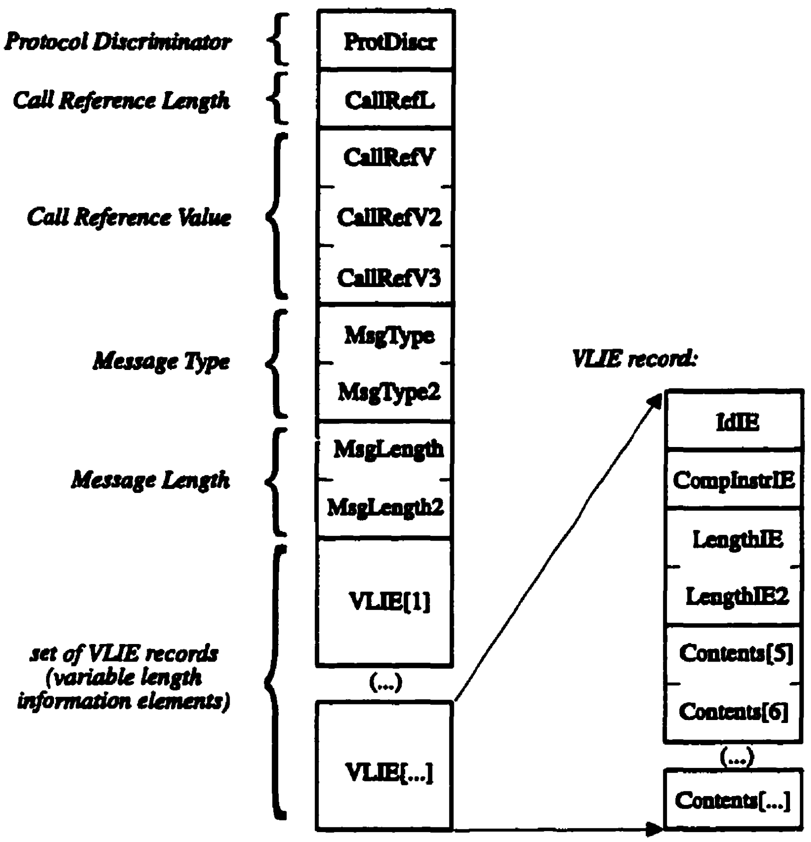


Figure 5.4: General message structure.

Message length (in octets). When an element occupies more than one octet, the first octet in the group is the most significant, e.g., CallRefV, MsgType, and the last octet is the least significant, e.g., CallRefV3, MsgLength2. It is important to note, that the octets used for these four fixed elements are *not* included in the value of message length.

The remaining part of the message is an array of so called *VLIEs*: variable length information element records. They are used to encode all remaining information carried by the message. Each VLIE record consists of:

1. **IdIE** (mandatory, 1 octet) - Identifier of Information Element,
2. **CompInstrIE** (mandatory, 1 octet) - Compatibility Instruction Information Element. It contains Coding Standard Field and IE Instruction Field. In this

implementation, both are coded to all 0s, which has a meaning: *"ITU-T standardized coding standard with IE instruction field not significant (regular error handling applies),"*

3. **LengthIE, LengthIE2** (mandatory, 2 octets) - length of the information element. As it is with message length, element length also do not include octets used by the *header*: IdIE, CompInstrIE, and IE length itself,

4. **Contents** (optional, variable length) - an array of octets. It contains the remainder of this information element. Its range starts from 5, in order to preserve numbering of octets (*header* takes 4) and a compatibility with the documentation.

Constant MaxIELength, which controls the size of the Contents array, is set to 34 and constant MaxNrIEs, which controls the size of the VLIE array, is set to 18, in order to accommodate the largest possible signaling message.

The same data structure is also used for communication on APIchannel and NNIchannel, but, in these cases, contents of the first 3 fixed elements in the message does not matter. The message Length element is still important for correct detection of the end.

5.5.3 Message Check

Message checking (and consequent error recovery, if necessary) is a very important part of the SignEnt module. It is performed every time the signaling message arrives through UNIchannel, and it is broken into two phases:

Testing — checking for presence and validity of all mandatory elements. Number of messages and variety of elements in different parameter configurations make it a tedious and fairly complex task, executed by large function msgCheck,

Reacting — i.e., following one of possible patterns of processing, depending on the result of the testing phase. This part is implemented by two transitions: one for correct and the other one for incorrect messages.

An outline of the Estelle solution for this mechanism is presented in Figure 5.5.

```
trans
   when UNI_USignIP.CONNCT_ACK
     provided msgCheck(data, CONNCT_ACKtype, RefArray)=OK    {— SUCCESS —}
         name MessageProcessing:
         begin
          cRef:=getCallRefAll(data);
          cRef:=alterCRFlag(cRef);
          position:=findCallRef(cRef, RefArray);
          output UIntUNI[position].CONNCT_ACK(data);
         end;
     provided otherwise                                      {— FAILURE —}
         name ErrorHandling:
         begin
          testResult:=msgCheck(data, CONNCT_ACKtype, RefArray);
          cRef:=getCallRefAll(data);                      { get info about the call}
          cRef:=alterCRFlag(cRef);
          if testResult=Discard then
                begin
                   { ignore message - do not do anything}
                end
          else if testResult=IncorrectUseGlobalCR then
                begin
                   (...) { send STATUS message with Global CR state}
                end
          else if testResult=NonExistingCall then
                begin
                   (...) { send Release Complete - call ref not recognized as relating to a call}
                end
          else if testResult=MandatoryIEMissing then
                begin
                   (...) { send status with cause #96 - mandatory element is missing}
                end
          else if testResult=MandatoryIEBad then
                begin
                   (...) {send status with cause #100 - invalid information element content}
                end
         end;
```

Figure 5.5: Message checking and error handling for CONNCT_ACK.

The first transition (marked SUCCESS) defines actions to be taken, when the message passed the test successfully, i.e., msgCheck returned OK. Most of the messages will be simply sent to their respective Call Control unit modules [12].

The second transition (marked FAILURE) is fired if msgCheck returns anything other than OK. First the transition must invoke msgCheck once again to repeat the testing, since it is not possible to use assignments within the conditional expressions in Estelle. When the exact diagnosis of a problem is known, appropriate actions are taken.

Careful analysis of error handling procedures in Q.2931 shows that all errors related to the message contents may be grouped into five distinct classes, each associated with its own pattern of processing. They correspond to the following test results:

- *Discard*: for all types of severe errors, which do not allow reading of some or all parts of the message header; also for some call reference errors in SETUP or RLEASE_COMPLETE,
- *NonExistingCall*: for call reference values indicating connections which do not have their Call Control units while the type of message requires CC [13],
- *IncorrectUseGlobalCR*: for Global Call Reference values found in messages which are not allowed to use them,
- *MandatoryIEMissing*: for messages without one or more variable length information elements, which are considered mandatory for this message type,
- *MandatoryIEBad*: for messages with one or more variable length information elements, which are considered mandatory and have invalid contents.

Actions assigned to these cases depend on the message type - Figure 5.5. shows their examples for CONNCT_ACK message.

[12]Depending on the type of message, some variations to this scheme may happen (e.g., additional check for Global Call Reference, if message is allowed to use it).

[13]Examples of messages which do not require the presence of CC on their arrival are: SETUP and STATUS with Global Call Reference.

5.5.4 Timers

When describing the Estelle language, we mentioned an important type of transitions with so called *delay clauses* (Section 4.2.3). These transitions introduce a notion of time into Estelle and allow us to implement timers.

The signaling specification defines timers in different states of call processing to limit the maximum response time of the other side of the interface or, in some cases, the response time of the remote interface. SignEnt and CC unit modules must be able to start the timer, detect its expiry, restart it and finally stop it at any given moment. Certain actions must be performed upon each timer expiry; moreover, they can be different every time the timer expires.

As the reader may recall, all transitions in Estelle are atomic and take no time to execute. It should be clear then, that there is no possibility to implement the timer in the form of a single transition; instead, we must define the whole set. We are also constrained by the fact that we cannot define separate timer states (e.g., stop, running, restart), since module states in our implementation are reserved for call states. Additionally, we would like to have one universal timing mechanism, which could be easily copied in many different situations and states without any major changes.

An outline of the solution which satisfies all the above conditions is presented in Figure 5.6. Timer T303 is used as an example here, but all other ones in our specification are based on the same concept.

Among the 4 transitions involved in timer T303 operations, only those named RUN1 and RUN2 are purely designed for timing. Transitions START and STOP, which turn the timer on or off, have also other duties, for example: processing and sending the messages.

Initially, the value of T303tryLimit is 2 and T303run is 0, therefore RUN1 and RUN2 are disabled (their firing conditions are false). When the transition START is executed, T303run changes to 1 and RUN1 becomes enabled. Since it is a delayed transition, it may fire only when it stays enabled for T303timeout period of time. If during that time STOP does not fire and T303run is still 1, the action defined for

```
from U0 to U1                   { START turns the timer on upon sending of SETUP}
   when UserIP.callREQUEST
      name START:
      begin
                {create SETUP message and copy all global info from what user sent}
       createMsgQ2931(stpdata, SETUPtype, context.callRef);
       copyGlobalIEs(stpdata, data);
       output NetIP.SETUP(stpdata);
                {start T303 timer}
       T303run:=1;
      end;

from U1 to same                 { RUN1 — timing for the first time}
   provided (T303run > 0) and (T303run < T303tryLimit)
     delay(T303timeout)
      name RUN1:
      begin
       output NetIP.SETUP(stpdata);  {resend SETUP after expiry}
       T303run:=T303run+1;           {increment the control variable of the timer}
      end;

from U1 to U0                   { RUN2 - timing for the last time}
   provided (T303run=T303tryLimit)
     delay(T303timeout)
      name RUN2:
      begin
       Done:=True;                   {destroy this module - internal clearing}
       T303run:=0;                   {stop the timer}
      end;

from U1 to U3                   { STOP turns the timer off}
   when NetIP.CALL_PROCEEDING
      name STOP:
      begin
       T303run:=0;                   { stop the timer T303}
       T310run:=1;                   { start the timer T310}
      end;
```

Figure 5.6: Implementation of example timer T303.

RUN1 is executed (here: SETUP is resent) and the value of T303run is increased by 1. Since it is now 2, RUN1 is disabled and RUN2 becomes enabled. Transition RUN2 controls the last run of T303. Again, if it succeeds to stay enabled for T303timeout, it executes actions defined for the final timer expiry (here: call clearing) and, eventually, turns itself off, i.e., T303run is set to 0. At any time, if an event defined for transition STOP satisfies its firing condition (here: arrival of CALL_PROCEEDING), STOP fires and turns the timer off.

5.5.5 Call Reference Allocation

Call reference is mandatory for each message, and it consists of *call reference value* and *flag*.

Value is coded on 23 bits, which corresponds to the decimal range 0..8388607. However, neither end of this range can be used for call identification, since they are both reserved by ATM Forum: 0 (binary all 0s) is used for Global Call Reference and 8388607 (called "dummy", binary all 1s) is reserved for future purposes.

In our implementation, each subsequent allocation produces a unique value in a valid range, which is always larger than the one allocated recently. The last constraint follows a suggestion in the documentation, that implementations should not reuse the call references right away after they are released [14]. Allocation is cyclic, i.e., after reaching the upper limit (8388606), it starts again from 1.

As we have already explained, a call reference value may be assigned by either side of the interface: always the one which initiates the call. It poses a problem, that the same value may be allocated simultaneously by user and network sides to two different calls, hence, it might fail to unambiguously identify the connection on the interface. UNI signaling resolves this issue by using the *call reference flag*.

Flag is located in 24th, the most significant bit of the call reference element. Its purpose is to identify the side of the interface, that allocated the call reference value. Flag is always set to 0 by the originating side and always set to 1 by the destination. Even if the same value is assigned by two sides, on each of the sides it will have a different flag and it may be properly distinguished.

[14]Even though it is formulated as a mere "suggestion", non-complying may result in serious malfunctions in the protocol (see the next chapter).

In our implementation, we simply negate the flag bit every time it crosses the UNIchannel, before it is used to find a corresponding CC unit in RefArray database. In this way, all entries in RefArray with flag=0 describe calls originated here, and all entries with flag=1 describe calls initiated on the other side. Since the flag is stored together with call reference value (as one integer), the signaling entity does not even need to consider the origins of the call in order to correctly dispatch its messages [15].

[15]Note, that whatever the value of a flag on a given interface is, after being sent back and forth (hence negated twice), it will be eventually the same as in the beginning.

Chapter 6

Testing and Validation

6.1 Introduction

As the final step in the development process, we needed to design some mechanisms for validation of our simulation model. Such mechanisms provide a means for systematic testing of the formal specification, fixing errors, and convincing us that our model behaves as we intended. In general, we may highlight two major goals of the validation:

1. It must show that the formal protocol specification conforms to the informal definition, that is, the behaviour of the proposed solution is identical to the requirements defined in the standard. This part concentrates on the implementation; it verifies accuracy of designer's own ideas, algorithms, procedures and data representation,

2. It must show that the protocol itself conforms to the intentions of its creators and achieves its goals, e.g., the protocol can set up and tear down the connection, recover from all kinds of faulty conditions and ensure continuous service for the user (no deadlocks, correct resource management, etc.). This part concentrates on both the implementation and the specification; it verifies that the protocol can actually deal with real-life situations.

We described validation as the final step of formal protocol specification because major testing efforts are taken after the code is written. It does not mean, however, that the validation begins at this point. In fact, it starts at a very early stage, together with the first outlines of the design. In order to be able to test the specification efficiently and thoroughly, we needed to think of its testing at each step of the development. Many issues had to be resolved not only to have the most efficient code, but also the one which could be easily tested. As a matter of fact, in some particular cases, the validation methodology was the most important factor; for example, APIchannel was designed to test the protocol rather than to provide any real life functionality. Validating such a "testing conscious" implementation gives us a very important advantage in comparison with testing of a commercially available signaling package: without an additional code instrumentation we are able to observe all internal actions of the protocol. It facilitates the validation process — which is complicated enough — and reduces the number of testing cases and messages. For example: instead of invoking status enquiry procedures to check the call state after each message, we can simply examine the internal CC module state; instead of waiting for time-outs to see if the timer is on, we can check the firing conditions of the timer transitions.

After we coded the protocol in Estelle, we compiled it using the Estelle Compiler, Ec. Since Estelle is based on very strict Pascal syntax and its own extensions push this formalism even further [1], Ec is able to discover many problems that are normally not detected in case of other programming languages. Not only did it allow us to remove many syntax errors, but also forced us to review some of the semantic problems and design faults that became apparent during the compilation.

After the above errors were corrected, we moved to the most difficult and tedious task of the protocol validation process. We conducted a series of simulations executed in different environments and for different parts of the protocol. The main purpose was to find all semantic and run-time errors, and to collect convincing evidence of correct protocol operation. We built several testing environments, prepared appropriate test cases, and ran them using the Estelle Debugger/Simulator, Edb. Depending on the testing scenarios and verified functionalities, simulations could be executed

[1] For example, Estelle requires that all Pascal functions must be *pure*, i.e., they do not have any side effects.

manually (step-by-step control, with thorough analysis of each fired transition), or they could be fully automatic (non-interactive experiments, running several hours and controlled entirely by Edb or custom defined scripts). Edb observers were used to trace the protocol activity and report all results.

The remainder of this chapter gives the details of our testing methodology, simulation environments and obtained results.

6.2 Methodology

The literature provides us with the broad spectrum of validation methods: from exhaustive, complete approaches, which verify all possible cases, to more traditional testing, which takes into account only some carefully selected subset of important (from the user's point of view) scenarios [20]. The discussion of advantages and disadvantages of particular methods is far beyond the scope of our work. For the purpose of this document, it is only important to note that, overall, it is difficult to find the best validation method for a real, complex communication protocol. Each approach has a certain cost associated with it, both in terms of effort and time necessary to obtain the results. Exhaustive testing is trustworthy, but is just too expensive. Less formal, cheaper tests do not prove correctness. For a fairly complex protocol such as Q.2931, combinations of states, messages, and carried parameters may cause "state explosion", which means that the set of possible cases and testing scenarios grows beyond a manageable size. Obviously, some trade-offs are necessary: the validation process has to be carefully designed to minimize the cost, but it must also cover all areas of protocol operation and provide enough evidence of correct behaviour.

We drafted our approach based on the ideas used for validation of the XTP protocol specification in [21, 22, 23].

We have broken the validation into two major steps:

Unit testing. Each distinct element of the signaling protocol (i.e., Call Control Unit on both sides of the interface, UserSignEnt and NetSignEnt), is extracted from the original model, placed into its own simulation environment and validated separately. This allows us to make sure that all these elements exhibit expected behaviour both in ideal conditions and in the presence of errors. By splitting this step into separate testing of Call Control and SignEnt modules, we can concentrate at any given moment only on functions that are realized within the particular module. It introduces a natural order of testing scenarios and greatly simplifies locating of errors. Since both sides of the interface communicate using different interfaces, we need separate testing environments for each side.

Interoperability testing. Validated and corrected basic elements of the protocol are put together into one model and tested in regard to their cooperation and mutual communication. It is necessary to check that individually correct actions of all parts constitute an equally good entirety, and that they do not lead the whole simulation into faults (such as deadlocks or contradictory activities of signaling entities). Also, only at this point, can some globally meaningful or implementation dependent issues can be tested, together with the definition of the protocol itself (e.g., usefulness of error recovery procedures). Simulations are conducted with many simultaneous calls, both in an ideal, "friendly" environment, where messages are conveyed without any problems, as well as a "hostile" one — messages can be delayed, corrupted, and lost.

Before we can move to the first phase of validation, it is necessary to decompose the formal specification into the set of separate, basic functions; assign them to their respective Estelle modules (in which these functions are implemented); and schedule them to the appropriate testing steps. Table 6.1 presents the validation schedule for all major functions described in Sections 5.3 and 5.4.

6.3 Call Control Unit Testing

6.3.1 Set-up

Two environments used for Call Control unit testing are depicted in Figure 6.1.

Tested function	**Location**
Format and contents of generated messages	Call
Message sequence for call establishment/release	
Timers associated with establishment/release	Control
Response to unexpected messages	
Status enquiry procedures	Unit
Spontaneous (unrequested) STATUS handling	
Call reference and connection identifier selection	
Dynamic allocation/release of CC units	Signaling
Message dispatching	
Message check and error recovery	Entity
Restart procedures	
Complete establishment/release procedures	
Managing simultaneous calls	Complete
Resource allocation (continued)	
Error recovery (continued)	Model
Deadlocks and other abnormalities	

Table 6.1: Verification schedule.

Other Side Test module is a generic module, which always plays the role of the opposite, currently not tested, side of the interface. It can accept and generate any message standardized by the Q.2931 specification. It does not contain any algorithm, however, so it cannot process received messages (they are discarded) or send anything out on its own initiative (all sent messages must be manually requested by the user). User Test and Network Test have similar tasks, but they simulate User and Network modules and use the messages defined for their respective interfaces: APIchannel and NNIchannel.

We are able to force each of these Test modules to send any type of message we want and observe the reaction of the Call Control Unit. The simulation is executed step-by-step and the outcome is constantly examined. Every time when one of the Test modules (User, Network, Other Side) should respond, we have a choice of providing either a desired or an unexpected message.

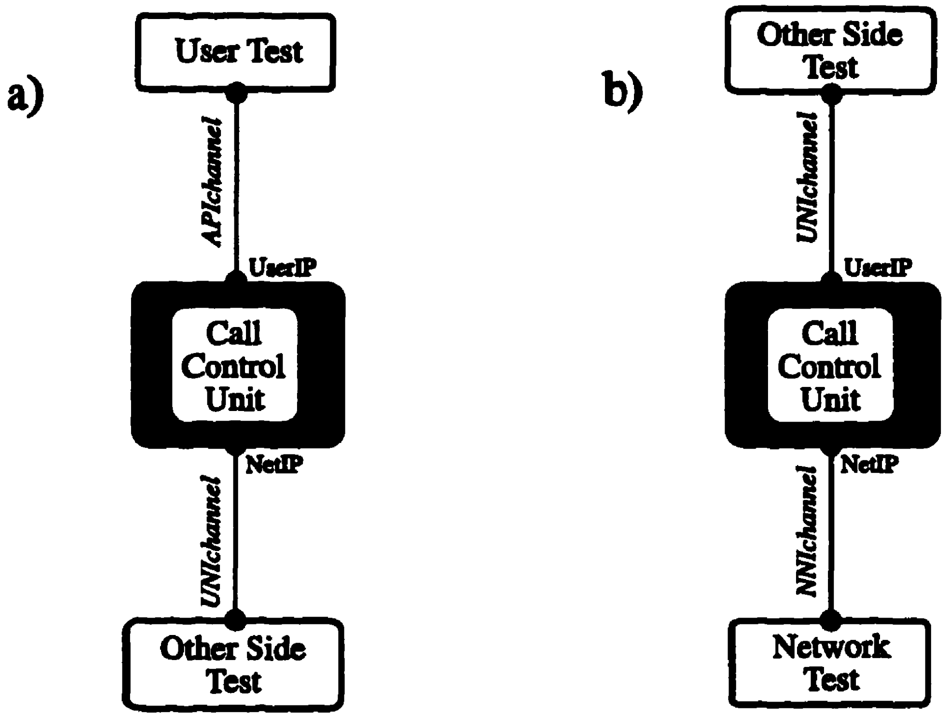


Figure 6.1: Test environments for Call Control unit on the user (a) and the network (b) side.

Technically, generation of messages by the Test modules is based on conditional transitions with initially false firing conditions. When we wish to send any message, we use one of the debugging features of Edb: an ability to modify internal module variables. We can access a Boolean variable controlling the condition of the appropriate transition, change it to true, and force this transition to fire. The last action executed by the transition is always returning its control variable to false, so that there are no further messages sent, unless we request them again. Messages used in this phase have correct form and contents, since message check is not tested in CC unit [2].

Because of the large number of possible test cases, they will not be presented here. Instead, in Figures 6.2 and 6.3 we have shown some example scenarios of message exchange in fault-free situations. Some test cases for erroneous conditions,

[2]Except for the lack of cause element in RLEASE.

such as out-of-sequence messages, may also be easily derived from these figures by replacing correct responses with all possible unexpected ones. Please note, however, that unexpected SETUP is a special case, which is always processed in SignEnt, so it must be tested there as well.

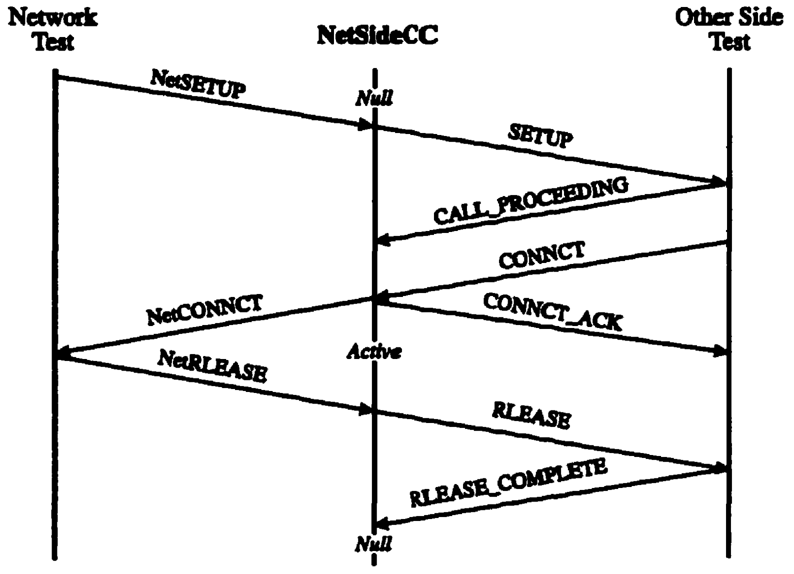


Figure 6.2: Sequence of messages for the incoming call on the network side.

6.3.2 Results

Connection establishment on the user side. It has been verified, that

for outgoing call:

- messages generated by the module have proper format and contents,
- SETUP sent after the reception of callREQUEST contains all the information supplied by the user. In particular, it does not contain connection identification (VPI/VCI), even if it was mistakenly sent by the user,

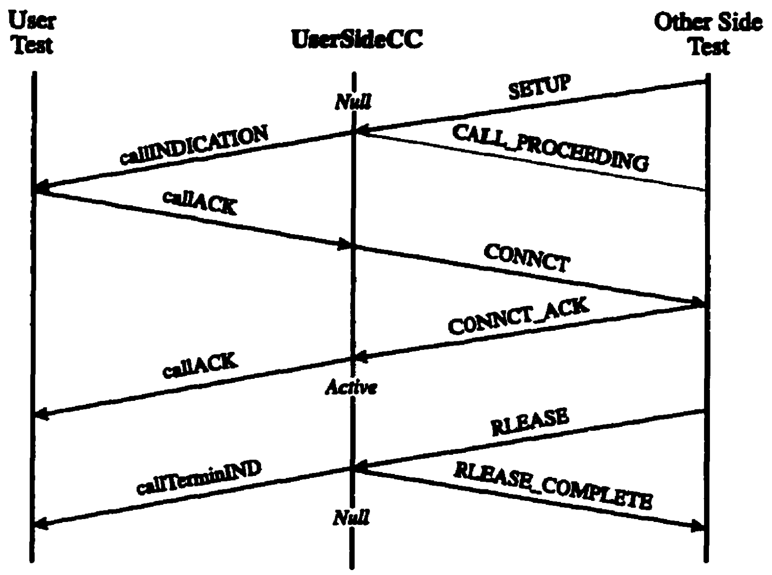


Figure 6.3: Incoming call from Figure 6.2 as seen on the network side.

- if the other side does not respond to SETUP with CALL_PROCEEDING or CONNCT (expiry of timer T303), SETUP is resent and if there is still no answer, the call is eventually internally cleared,
- if the other side responds with CALL PROCEEDING but it does not send CONNCT before timer T310 expires, clearing procedures with cause #102 "recovery on timer expiry" are invoked,
- upon reception of CONNCT, module enters Active state, returns CONNCT_ACK and notifies the user (by callACK).

for incoming call:

- upon the reception of SETUP with acceptable parameters (e.g., correct VPI/VCI values), the user is notified with callINDICATION. Then the module may choose to send CALL_PROCEEDING or not,
- after receiving callACK from the user, CC unit generates CONNCT and enters Active state.

Connection establishment on the network side. It has been verified, that

for outgoing call:

- upon the reception of SETUP, CC unit checks that VPI/VCI are available and sends NetSETUP with the contents supplied by the user. Then the module randomly chooses to send or not to send CALL_PROCEEDING to the calling user. If CALL_PROCEEDING is sent, it contains mandatory connection identification element with VPI/VCI values assigned to the call,
- after NetCONNCT is received, the module generates CONNCT message and enters Active state. If the module chose not to send CALL_PROCEEDING before, CONNCT contains connection identification element,
- if CONNCT_ACK message is received in Active state, it is ignored.

for incoming call:

- upon the reception of NetSETUP, CC unit checks that VPI/VCI are available and generates SETUP with all globally meaningful information retrieved from NetSETUP. Information elements that have local meaning and were included in NetSETUP (e.g., connection identification on the remote interface) are not transferred. Additionally, connection identification element with local VPI/VCI values is also appended,
- if the other side does not respond with CALL_PROCEEDING or CONNCT before the final expiry of respective timers, call clearing procedures are initiated,
- upon reception of CONNCT, the module generates CONNCT_ACK and NetCONNCT.

Connection clearing on the user side. It has been verified, that

for outgoing clearing:

- RLEASE is sent and timer T308 started; the message contains the cause element supplied by the user,
- upon reception of RLEASE_COMPLETE, all timers are stopped, callTerminIND is sent to the user, and the Done variable is set to true. It does not matter if RLEASE_COMPLETE contains a cause element,
- if RLEASE_COMPLETE is not received before expiry of T308, RLEASE is resent and timer restarted. Upon final expiry of T308, RESTART is sent,

for incoming clearing:

- upon reception of RLEASE, the unit responds with RLEASE_COMPLETE, sends callTerminIND to the user and sets Done to true. If RLEASE contains cause element, it is copied into RLEASE_COMPLETE, otherwise cause #96 "mandatory element missing" is used.

Connection clearing on the network side. It has been verified, that the behaviour of this module during clearing procedures corresponds to the behaviour of the CC unit on the user side with the following modifications:

for outgoing clearing:

- procedure is initiated upon reception of NetRLEASE message, instead of callTerminRQST,
- upon reception of RLEASE_COMPLETE, the module is returned to Null, but no confirmation is sent to the remote user (i.e., there is no equivalent of callTerminIND on the network side of interface).

for incoming clearing:

- instead of callTerminIND message, NetRLEASE is used to inform the remote interface about termination.

Error handling on the user side. It has been verified that:

- timers used for both outgoing (T303, T310) and incoming (T313) call, as well as clearing timer T308 and status enquiry timer T322 are properly started, stopped and restarted according to guidelines in specification,
- actions upon the expiry of all above timers are correct,
- clearing procedures invoked as a result of errors are performed in the same way as user initiated call clearing, with the exception that the cause element contains the value corresponding to the error that caused clearing (instead of the value supplied by the user),
- in case of incoming call, an attempt to assign incorrect VPI/VCI values generates RLEASE_COMPLETE with cause #35 "requested VPCI/VCI values unavailable",
- when unexpected RLEASE_COMPLETE is received, the unit is returned to Null state and callTerminIND is sent to the user,
- when unexpected RLEASE is received, the unit is returned to Null state, CallTerminIND is sent to the user and RLEASE_COMPLETE is returned,
- when any other unexpected message is received, the module generates STATUS message with the current state of the call and cause value #101 "message not compatible with call state",
- when STATUS is received indicating Null state, while the module is any other state, the module is returned to Null and the callTerminIND is sent,
- when STATUS is received indicating a state other than Null, but with one of the inappropriate cause values (96, 97, 99, 100, 101), outgoing call clearing is initiated,
- when STATUS is received with contents that qualifies for clearing of the call, but the call is already being cleared, no action is taken and the release process continues.

Error handling on the network side. It has been verified, that the behaviour of the unit in error conditions is very similar to the behaviour of the unit on the user side, except for the following:

- T322 and T313 timers are not used,
- in addition to regular actions after the final expiry of timers T303 and T310, NetRLEASE with cause #18 "no user responding" is sent,
- if incoming call is initiated when it is not possible to accommodate a new connection, NetRLEASE is returned with cause #41 "temporary failure",
- instead of callTerminIND in error conditions, NetRLEASE is generated with the cause provided by the user side or #111 "protocol error, unspecified", if nothing is supplied,
- upon reception of CALL_PROCEEDING or CONNCT which contains connection identification element indicating different values from the ones included in SETUP, outgoing call clearing is triggered with cause #36 "VPCI/VCI assignment failure" and NetRLEASE with cause #41 "temporary failure" is generated.

Status enquiry on the user side. It has been verified that:

- when STATUS_ENQ is received in any state, STATUS message is sent with the current state of the unit and cause #30 "response to status enquiry". No change in the call state occurs,
- STATUS with contents that does not trigger error recovery (described above) is disregarded,
- upon reception of callStatusRQST, STATUS_ENQ is generated and timer T322 started. Until this enquiry is completed with STATUS received from the other side, all subsequent callStatusRQST messages from the user are ignored (only one outstanding enquiry in a given moment exists),

- when STATUS is received as a response to STATUS_ENQ, it is processed in the same way as the one that arrived spontaneously, with the exception that it is additionally forwarded to the user as callStatusIND.

Status enquiry on the network side. It has been verified that status procedures here are the same as on the user side with the exception that there is no equivalent of callStateRQST and callStateIND messages from the network. As a consequence, sending of STATUS_ENQ by the module does not take place and timer T322 is not used. Actions following reception of STATUS_ENQ and STATUS messages sent by the other side are identical to those described above.

6.4 Signaling Entity Testing

6.4.1 Set-up

The testbed used in this phase is basically the same as in Figure 6.1. Module CC unit has been replaced with respective SignEnt module and Other Side Test module has been slightly modified. Instead of generating only correct messages, Other Side must be able to produce all kinds of faults in message formats and contents. Messages may be sent with incorrect protocol discriminator, call reference values, message types, missing or invalid information elements, etc. Most of it is achieved by simple manual intervention in the message contents during the simulation, because source code changes would require lengthy recompilations. Again, the simulation is step-by-step and completely under our control. Apart from checking the new functions assigned to this step in Table 6.1, we also need to repeat some of the test cases for CC unit. This time, however, we do not verify the responses of Call Control, but rather the correctness of SignEnt transitions responsible for dispatching the messages.

Since functions performed by signaling entities on both network and user sides of the interface are almost the same and their testing is very similar, the results will be discussed together.

6.4.2 Results

Call control unit administration, resource and message management. It has been verified that:

- Call reference values selected by the entity are unique, in valid range and generated in a progressive manner,
- VPI/VCI selection assigns values that are in valid range and unique on a given interface,
- upon reception of correct SETUP, a local *context* (set of parameters) for the connection is created, Call Control unit is dynamically allocated and duly initialized. Throughout the whole life of connection, all newly obtained information (e.g. connection identification) is properly updated in internal structures. When the call is cleared, its local context is destroyed, resources returned for reuse and the CC unit is released,
- incoming messages are correctly dispatched to their respective CC units, outgoing messages are conveyed to the external interaction point and sent outside,
- on the network side, the additional "routing" database (Correspondence Array) is properly created, updated with new data, and finally disposed of upon clearing of the network connection.

Error conditions. It has been verified that the msgCheck function returns appropriate evaluation of the message state, which results in the following actions of the signaling entity:

- if the message should be discarded, no action is taken on the message and no state change occurs, as if it never arrived,
- if any message, except SETUP, RLEASE_COMPLETE, STATUS_ENQ and STATUS, relates to a non-existing call, RLEASE_COMPLETE is returned with cause #81 "invalid call reference value",

- if any message, except RESTART, RESTART_ACK and STATUS, uses Global Call Reference, STATUS is returned with cause #81 "invalid call reference value",

- if any message, except SETUP, does not have an element which is mandatory for this message, STATUS is returned with cause #96 "mandatory element missing". In case of SETUP, RLEASE_COMPLETE is returned,

- if any message, except SETUP, contains a mandatory element with invalid contents, STATUS is returned with cause #100 "invalid information element contents". In case of SETUP, RLEASE_COMPLETE is returned,

- on the user side, if CALL_PROCEEDING or CONNCT (when it is the first response to SETUP) arrives without or with invalid VPI/VCI, it is treated as a message with mandatory element missing or invalid, respectively,

- optional information elements are simply transported and not checked either for presence or for validity. Since they are not mandatory, connection may proceed even if they are corrupted or missing.

Restart procedures. It has been verified that:

- RESTART and RESTART_ACK generated by the entity have proper format and contents,

- upon the initiation of outgoing restart (expiry of timer T308), timer T316 is started and no other outgoing restart is permitted. If another Call Control unit generates RESTART while T316 is running, this message is not processed until the ongoing restart is completed,

- the first expiry of T316 resends RESTART; when it is expired for the last time, connection is considered to be unusable. Call control unit is not destroyed, virtual channel is not released and associated resources (e.g., call reference value, VPI/VCI values) are left in the local call context and marked as "OutOfOrder". On the user side, callOutOfOrder is additionally generated for the user,

- when RESTART_ACK arrives before T316 expires and the channel indicated in the message corresponds to the one being restarted, call control unit and all resources are released (local context is wiped). Additionally, on the user side, call termination is reported to the user, while on the network side of the interface the remote connection is cleared (i.e., NetRLEASE is sent),
- when RESTART_ACK arrives and T316 is running, but the channel indicated in the message is different from the one being restarted, the message is discarded; when RESTART_ACK arrives and T316 is not running, STATUS with cause #101 "message not compatible with call state" is sent,
- when incoming restart is initiated upon reception of correct RESTART message, timer T317 is started and simulation of internal "clearing" is launched,
- if internal clearing finishes before T317 expires, Call Control unit and all resources are released. Additionally, either user is notified of termination (user side) or remote connection is cleared (network side),
- if T317 expires, the connection is considered unusable. On the the user side, callOutOfOrder is sent,
- if RESTART arrives when T317 is already running, STATUS is generated with cause #101 "message not compatible with call state",
- incoming and outgoing restart procedures are carried in parallel and are independent. If the same channel is being restarted simultaneously, the final result depends on the procedure that terminates first — the second one has no way to overrule a decision already taken (success or failure).

6.5 Complete Model Testing

6.5.1 Set-up

The testing environment used in this step is based on the original simulation model from Chapter 5. It is presented in Figure 6.4. The only modifications are two additional instances of the module SAAL Test, which simulate the SAAL layer on both interfaces.

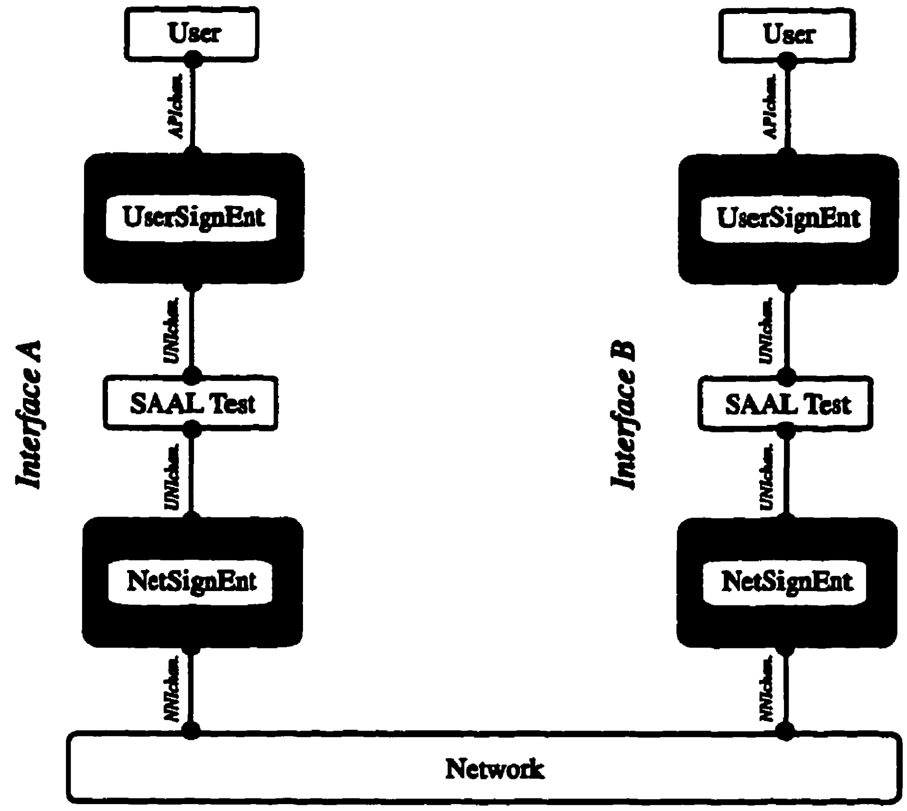


Figure 6.4: Test environment for complete model.

SAAL Test modules are necessary to introduce a "hostile" environment. They intercept messages sent between protocol entities, delay these messages for a randomly chosen period of time, and, finally, forward them to the destination. Since the capacities of SAAL Test modules are limited, it is possible that an arriving message cannot be stored and must be dropped. As well, messages do not have to be sent in the order they were received (there is no FIFO queue there), so that missequencing of messages belonging to the same connection can occur [3]. Additionally, we also use

[3]It may look like an overkill, since SAAL normally guarantees delivery and sequencing. Please note, however, that the protocol must also deal with situations where the other signaling entity is faulty. By introducing all kinds of hazards between the entities, we avoid the necessity of explicitly simulating incorrect behaviour on the other side.

Edb observers to remove messages from interaction points and corrupt them in transport, thus directly triggering recovery procedures for otherwise difficult-to-provoke situations.

Simulations are automatic; they are partially controlled by scripts, but mostly by a non-deterministic selection of execution paths. User modules on both interfaces randomly generate multiple requests for outgoing connections, accept or reject the incoming calls, and terminate the ones in progress. Characteristic of the user behaviour is basically defined by the algorithm coded in the module, but may be also tuned by setting some global variables (e.g., each user may not be allowed to terminate, initiate, or reject the call). Experiments are conducted in both "friendly" (no external disruptions) and "hostile" environments (with SAAL Test modules and destructive observers). The level of "hostilities" can also be adjusted by changing transmission delays, manipulating the capacities of internal SAAL modules storage and defining appropriate observers.

6.5.2 Results

This final step of testing turned out to be the most difficult to plan and summarize. During Call Control and SignEnt validation, the test scenarios are directly derived from the functions performed by the modules. In the testing of a complete model, we do not really know what kind of problems we are looking for — our goal is to find all abnormalities which might lead to any kind of incorrect behaviour. First, we need to identify faulty situations (and it is not trivial in case of four signaling entities working in parallel), then we have to trace back the sequence of events that caused this error to occur, understand its reasons, and, finally, find a solution. It is always possible, that in the process of fixing one particular problem, we interfered with previously tested elements of specification, so that the whole testing must be relaunched. Additionally, however long and exhaustive the simulations may be, they are still merely based on a statistical assumption that sooner or later the errors will be revealed. Obviously, there is no guarantee that all problems can be found; random simulation may never exploit some scenarios.

It is not possible to directly present the exact results of this phase of the validation.

The reader should realize that each of the simulations produced hundreds of kilobytes of trace files, which kept track of all events in the system. We had to analyze these files step-by-step, visualize the protocol operation and decide whether it complies with the requirements of the documentation. It was definitely the most tedious and tiresome stage of the entire project. As a result of these experiments, we introduced many changes to the original draft of Estelle specification. Particularly, error recovery mechanisms in SignEnt modules were almost completely redesigned to receive their final shape as described in Section 5.5.3.

Our implementation of the protocol has been shown to be able to establish, sustain and tear down multiple point-to-point connections. It can correctly recognize and recover from erroneous situations. Error handling procedures are sensitive enough; even if they are unnecessarily triggered in correct situations [4], they do not cause any malfunction. Simultaneous connections do not interfere, i.e., states or messages belonging to one call have no influence on another. Allocation of resources is correct: entities do not assign more than they have, all values which have to be unique do not appear more than once, released resources are reused. The protocol does not deadlock or stall.

Additionally, these simulations contributed to even deeper understanding of some subtleties of Q.2931, clarified the intentions of the protocol designers and revealed interesting situations that may happen in reality. Some of them are briefly described in the next section.

6.6 Observations and Conclusions

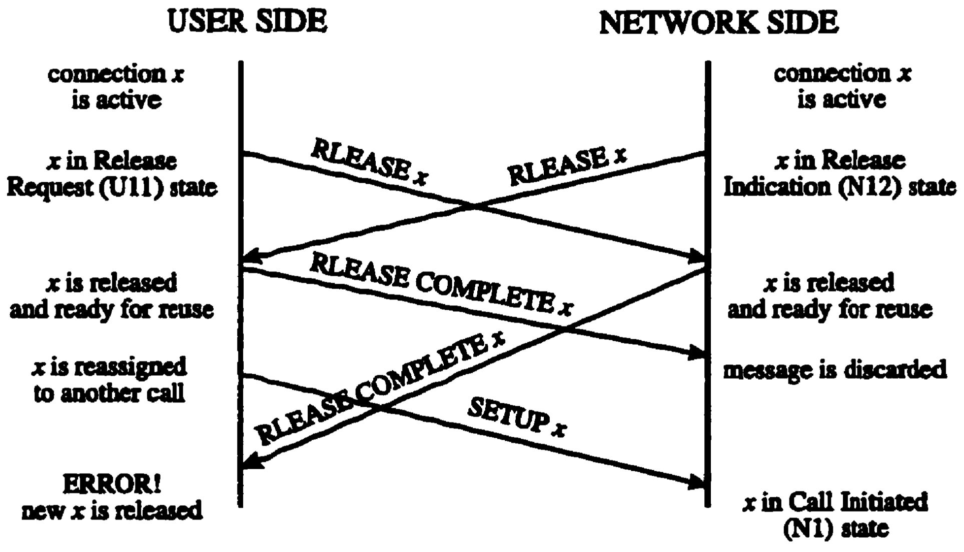
Like most specifications, Q.2931 defines actions to be implemented, but it does not bother to explain or justify their purposes. Sometimes, it is difficult to understand some of the solutions only on the basis of their definition. Random simulations of the protocol may be of great help here, since they tend to wander into sequences of events that are not easily anticipated. We will present some selected issues discovered during

[4]It is not an error, but may happen if two entities at the same time try to do something with the connection (e.g., release it). See the next section for details.

simulations, give additional explanations, and back them with specific examples. It will allow us to understand an importance of certain design solutions, which otherwise might not get enough attention, even though they can have a large impact on the correct protocol operation.

Why should the call reference value not be reused immediately?

According to the specification: "... it is suggested that implementors avoid immediate reuse of the call reference values after they are released." As a matter of fact, it should not be just a suggestion, since disregarding this rule may lead to serious problems on the interface. Figure 6.5 presents an example of scenario which illustrates the danger.



USER SIDE
NETWORK SIDE
connection x is active
connection x is active
x in Release Request (U11) state
RLEASE x
RLEASE x
x in Release Indication (N12) state
x is released and ready for reuse
RLEASE COMPLETE x
x is released and ready for reuse
x is reassigned to another call
RLEASE COMPLETE x
message is discarded
SETUP x
ERROR! new x is released
x in Call Initiated (N1) state

Figure 6.5: Immediate reuse of the call reference value.

Let us assume, that both sides simultaneously decide to clear the connection identified by the call reference x. After RLEASE messages for x are sent, UserSignEnt enters Release Request state (U11) and NetSignEnt enters Release Indication state (N12). They both await for RLEASE_COMPLETE, but instead they receive RLEASE messages, so error recovery procedures are triggered. According to the definition, both sides must return RLEASE_COMPLETE, enter Null state and release all resources. If the "suggestion" from the documentation were not followed, the call reference value could be reused for any other connection now.

In our example, the user side assigns x to a new outgoing call and includes it in SETUP sent to the other side. On the network side, reception of RLEASE_COMPLETE does not cause any problems, since RLEASE_COMPLETE referring to an unknown call (x is already released) is simply discarded. Upon arrival of SETUP, new call establishment is started. Unfortunately, reception of RLEASE_COMPLETE on the user side has disastrous consequences. It is treated as a response to SETUP and the newly initiated call is terminated without any reason.

How can we have unexpected messages in a fault-free environment?

Examples of unexpected reception of RLEASE and RLEASE_COMPLETE messages, even though everything works correctly, are presented above. It happens due to the simultaneous actions of both signaling entities rather than real errors. Another example, this time for CONNCT_ACK is shown in Figure 6.6.

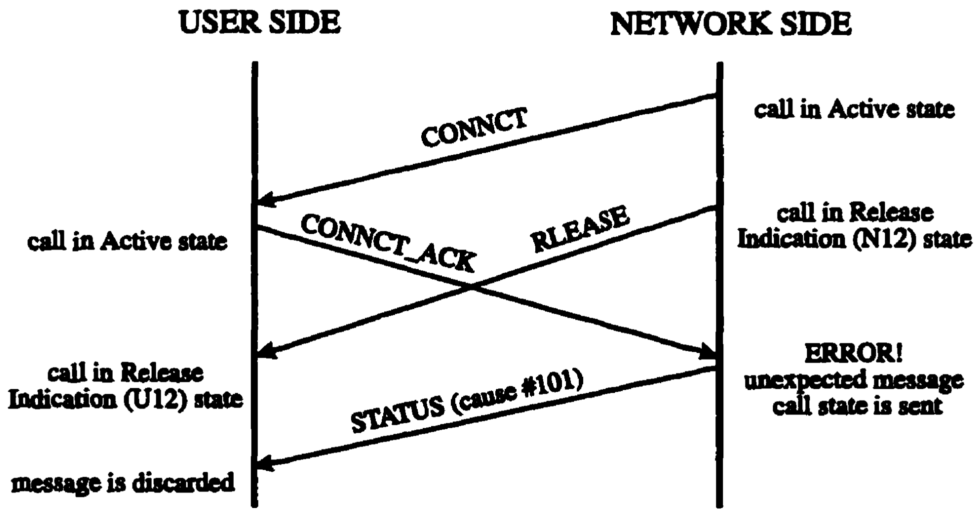


Figure 6.6: Unexpected CONNCT_ACK in error-free message exchange.

After NetSignEnt transfers CONNCT message across the interface, it enters an Active state immediately, without waiting for acknowledgment from UserSignEnt. This acknowledgment will be sent anyway, but should be discarded by the network.

It is possible though, that before UserSignEnt responds with CONNCT_ACK, the network side already starts clearing: it sends RLEASE and enters N12. Arrival of CONNCT_ACK in this situation is treated as an error condition and STATUS with cause #101 "message not compatible with call state" is generated. Fortunately, this message does not escalate the confusion; although usually it signals a fatal error, when received in Release Request, Indication or Null state, it is only discarded.

Why should the user ever reject the VPI/VCI values allocated by the network?

The specification states that the user can reject the connection identification (VPI/VCI) assigned by the network by sending RLEASE_COMPLETE with cause #35 "VPCI/VCI not available". On the other hand, allocation of these values is entirely and always the responsibility of NetSigEnt. Therefore, it might not be clear, why UserSignEnt has a right to question the assignment received from the network. Figure 6.7 describes a situation, in which it is justified.

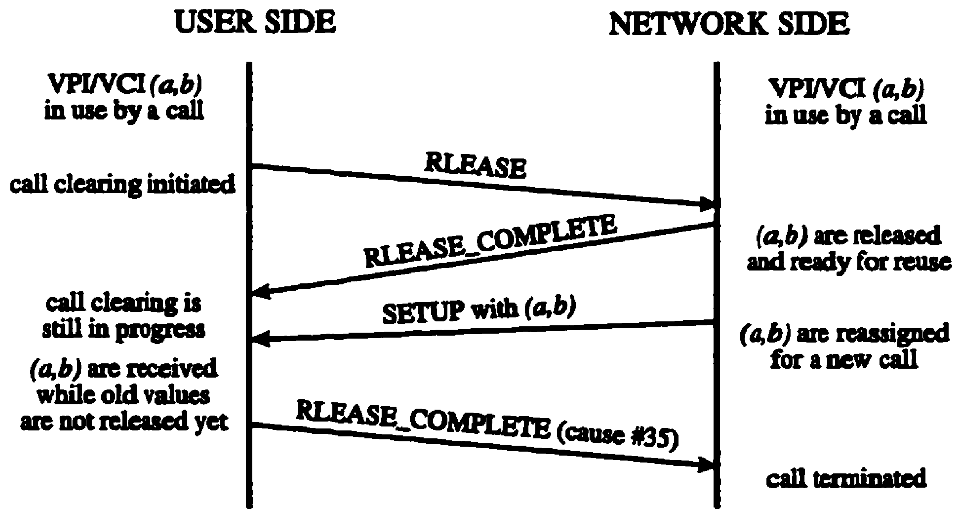

USER SIDE
NETWORK SIDE
VPI/VCI (a,b) in use by a call
VPI/VCI (a,b) in use by a call
call clearing initiated
RLEASE
RLEASE_COMPLETE
(a,b) are released and ready for reuse
call clearing is still in progress
SETUP with (a,b)
(a,b) are reassigned for a new call
(a,b) are received while old values are not released yet
RLEASE_COMPLETE (cause #35)
call terminated

Figure 6.7: VPI/VCI values rejection by UserSignEnt.

Let us assume, that there is a connection with VPI/VCI values (a, b). UserSignEnt wishes to clear the call and sends RLEASE. (a, b) will not be released on the user side until all clearing procedures are finished. After NetSignEnt receives the clearing message, it releases all resources and responds with RLEASE_COMPLETE.

At this point, (a, b) may be reused by NetSignEnt, since the call is already terminated and there are no other constraints (as in case of call reference). Hence, when another establishment is initiated, NetSignEnt assigns (a, b) to the new connection and includes them in SETUP (could be also CALL_PROCEEDING or CONNCT). Please note, that even though the sequence is preserved and SETUP arrives later than RLEASE_COMPLETE, it belongs to another call. Different calls are controlled by separate finite state machines, which are neither synchronized nor correlated in any way. It is possible, that SETUP will start to be processed when the processing of RLEASE_COMPLETE is not finished and resources of the old call are still not free. In such a situation (a, b) received in SETUP cannot be accepted for the new connection and RLEASE_COMPLETE with #35 "VPCI/VCI not available" will be returned.

In conclusion of the verification process, we believe that our formal specification truthfully represents the Q.2931 protocol in all these aspects, which were intended for implementation in this work. We do not claim it may be formally proven, since validation through simulation cannot guarantee the correctness, but we trust that both our approach to testing and our obtained results are credible. Nonetheless, there are certainly other ways to validate the protocol and other scenarios to explore. In particular, restart procedures, which do not follow the regular pattern of message processing in Call Control units, may be potentially error-prone. Since they can be launched in unforeseen and undefined circumstances, it is extremely difficult to construct any exhaustive methodology for testing this kind of behaviour. Unquestionably, more work can be done in this field, but the development of complete and universal validation suites for all aspects of ATM signaling was not meant to be a part of this project.

Chapter 7

Related Work

7.1 Research on ATM Signaling Protocols

Due to the common recognition of ATM as a technology of the future, it is impossible to even list universities, research centers, governmental institutions, commercial vendors and telecommunication companies working in all areas of ATM networking [1]. In a brief summary below, we will only mention some of the publicly accessible projects, the most recent and the most correlated with the topic of this document.

In Concordia University, Montréal, Morteza Ghodrat, under the supervision of Professor J.W. Atwood, formally specified and validated in Estelle the major part of the ATM lower layer signaling: Service Specific Connection Oriented Protocol (1995). This project [25] provided direct motivation for our own work and was the first step in an on-going effort to create a complete ATM UNI signaling mechanism in Estelle.

In the University of Ottawa, students supervised by Professor Luigi Logrippo specified part of the ATM UNI signaling using another Formal Description Technique — LOTOS (1995) [26]. Both SSCOP and Q.2931 are taken into account. Due to the mathematical and strict nature of this particular description technique, the LOTOS

[1]ATM Forum alone, since it foundation in 1991, grew from 4 to more than 750 member organizations worldwide [24].

specification concentrates more on a theoretical analysis and formal correctness of the protocol, as opposed to the implementation-oriented solutions typical for Estelle.

Simultaneously with our research, in the Université de Montréal, students under the supervision of Professor Gregor v. Bochman are working on yet another ATM signaling project — this time in Specification and Description Language, SDL. Their efforts include formal specification of Q.2931 point-to-point connection control, as well as the development of test suites for automatic validation of the protocol. Results of this project are expected to be available in 1997.

Harri Hansen from Helsinki University of Technology deals in [27] with signaling issues in case of a wireless access to the ATM network (1996). He proposes a set of mobility specific functions to be implemented over conventional ATM switches, in order to create a Wireless ATM (WATM) environment and integrate it with stationary public ATM networks. Even though most of the work concentrates on solving purely mobility-related problems, connection management functions identify also signaling protocols and handover mechanisms necessary for integration with fixed telecommunication networks. From the point of view of both mobile terminal (MBT) and wired ATM switch, call control is based on the Q.2931 specification. To support mobility of terminals, an additional signaling protocol, called W-EXT (Wireless Extension), is used between the MBT and the switch. W-EXT is transparent for the Q.2931 connection and is used for locating the terminal. security call authentication, and handover. The author suggests that the W-EXT signaling connection is established "on demand", i.e., in situations where Q.2931 generates SETUP and initiates its own procedures. Unfortunately, he does not elaborate in more detail on any practical ways of doing it, in particular, on the issue of possible time-outs. Timers in Q.2931 are designed for very reliable physical media and do not leave too much time for any extra activity. For example, the first response to SETUP is expected in just 4 seconds. Apart from usual signaling delays (resource allocation, waiting for a response from the user, etc.), in WATM this period must additionally include paging the MBT, security functions (checking the databases), access to the medium, and two way radio communication (possible faults and retransmissions). Since increasing time-out values is not a solution (being connected to the public or private UNI, WATM must work with standard settings), a successful connection establishment seems to be difficult to achieve.

An interesting case of education-oriented work on Q.2931 can be found on the Internet. David Hudek created demonstration package for UNI 3.1 signaling [28], written entirely in Java. It is not a working implementation or specification of the protocol, but it may serve as a useful educational tool for understanding of signaling principles. After call establishment or release is interactively requested, the Java applet visualizes step-by-step the processing of call control procedures. Apart from a simple animation representing information flow in the network, the tool records and displays an exact sequence and contents of all exchanged messages.

As we already mentioned before, various software and hardware vendors offer their own signaling packages. These solutions may be either direct implementations of official specification or independently defined protocols, which preserve partial or full compatibility with standards. The first category is represented, for example, by CELL-UNI 3.1&3.0 [29] package from Cellware; the second group includes Simple Protocol for ATM Network Signaling, SPANS [30] designed for Fore switches. Obviously, only general information and documentation for these products can be accessed; their implementation details and the code are not freely distributed.

Even though Q.2931 is an international standard, it is not the only existing ATM signaling protocol. Grenville Armitage in University of Melbourne proposes gNET [31] — signaling protocol for ATM local area networks (1994). This work was initiated when international standards did not exist and it was specifically designed to support multimedia terminals. gNET provides basic service for connection establishment and termination, but with the set of additional constraints resulting from its intended applications. Virtual connections can only be unidirectional, they are able to perform point-to-multipoint unidirectional multicasting, and they are admitted with a limited set of traffic parameters. The protocol defines its own addressing mechanism (both for interfaces and AAL users) and also supports "shared medium" links, where different nodes are connected to the common bus and have access to each other's cells. A distinctive feature of this solution is that signaling messages are restricted to only one cell. This simplifies greatly the management of connections over a shared medium, but, at the same time, limits to a minimum a variety of services provided by the protocol.

Another example of a signaling protocol is the Generic Universal Line Protocol, GULP, presented by See-Mong Tan in [32]. The protocol is used to control communication between active objects in an object-oriented Architecture for Call Processing, Archos, developed in University of Illinois at Urbana Champlain. GULP is a simple signaling and supervision protocol, which realizes "process-per-call" (also known as "thread-per-connection") model of ATM call processing. Apart from setting up and tearing down connections, it is also used to synchronize communicating objects. See-Mong Tan is also a co-author (with Roy Campbell) of a project on x-ATM: A Portable ATM Protocol Toolkit [33]. It is an environment for experiments in implementation of ATM protocols, located in various layers of protocol reference model: AAL, signaling, IP over ATM, etc. One of the most important features of the toolkit is its signaling suite. It is based on a generic SIG protocol, which represents the finite state machine for processing abstract signaling messages. Upon reception of initiating message, SIG creates a separate and autonomous thread for a new connection, and from now on, all the remaining messages are processed by this thread [2]. Interesting property of this protocol is that its behaviour may be easily "translated" into any other linear signaling protocol (like Q.2931, SPANS, or GULP). In the author's own words: "In object- oriented terminology, the superclass SIG implements abstract signalling while subclasses such as SPANS and Q.2931 specialize SIG for their own particular protocols." The finite state machine implemented in SIG is independent and invariable in all cases, only messages must be translated from an abstract format used by SIG into the factual signaling formats used by a given protocol.

7.2 Estelle Specifications

The Estelle FDT is used all around the world for formal specification of protocols, services, and systems. One Estelle software toolkit, EDT [18, 19] alone is officially licensed to more than 30 universities and research centers in 11 countries. It was also used in some industrial applications.

The French Institut National des Télécommunications, INT (where EDT is being developed) is one of the most active promoters of Estelle. Researchers from INT are

[2]Please note similarity to our own solution with SignEnt and Call Control Unit.

particularly involved in work on the Xpress Transfer Protocol, XTP [34]. XTP is a high performance protocol designed for modern distributed, real-time, and multimedia systems. It defines functionalities within transport and, partially, network ISO OSI layers (e.g., support of routing). Subsequent versions of XTP were specified and validated using Estelle. Validation methods designed for specification of XTP version 4.0 [21, 22, 23] were a basis for drafting our own testing techniques for Q.2931. Xpress Transport Protocol is also under extensive study in the High Speed Protocols Laboratory in Concordia University, Montréal (often in collaboration with INT) [35, 36].

La Trobe University in Melbourne, Australia, carries research on formal description of ISO standardized ROSE protocol [37] and applying Numerical Petri Net approach for verification of Estelle specifications [38]. Introduction of analysis facilities offered by high-level Petri Nets into Estelle definitions is also studied in the Technical University of Ilmenau, Germany [39].

The University of Delaware, Newark and the U.S. Army cooperate in using Estelle for designing, testing and performance evaluation of military communication protocols [40, 41]. They also created formal description for ISO defined Virtual Terminal Protocol [42] and Network Time Protocol [43]. Additionally, University of Delaware hosts an ftp site [44] with publicly accessible complete Estelle specifications. Apart from many protocols mentioned above, interested reader may find there, among other definitions, Distributed Queue Dual Bus (DQDB) standard for Metropolitan Area Network (ISO 802.6) [45] and ITU-T Recommendation Q.921 [46].

Chapter 8

Conclusions and Future Work

ITU-T Recommendation Q.2931 and ATM Forum specification UNI 3.1 define the signaling protocol to be used in ATM networks for both private and public versions of the User Network Interface. The protocol provides a uniform means for establishment and release of switched virtual connections.

8.1 Conclusions

In this thesis, we designed and created a formal specification of the Q.2931 protocol in the Estelle FDT. Since the protocol operation on the user and the network sides of the interface is not symmetric, Q.2931 is represented formally by two corresponding signaling entities: UserSignEnt and NetSignEnt. In order to demonstrate and validate signaling functionalities, we developed a simulation model, which corresponds to the working environment of the protocol. Apart from the signaling entities, it includes modules acting as ATM users and NNI signaling protocol of ATM networks. For the purpose of this model, we also needed to fill existing specification gaps with our own solutions for the Application Programming Interface, API, and the interworking procedures between UNI and NNI protocols.

During the experimental phase of the project, we ran numerous simulations to verify that our Estelle description can be considered a formal counterpart of the original specification. We followed a carefully designed, systematic, bottom-up validation path. In the first place, we tested the behaviour of the basic module representing a single finite state machine. Then, we concentrated on validation of the signaling entities on both sides of the interface independently. Finally, in a series of extensive simulations, we tested interoperability of all elements of the model. For each step, we developed a separate testing environment. This methodology allowed us to locate and remove many errors, which resulted in redesigning and rewriting some parts of the specification. As a consequence of the validation process, we believe that the final version of our formal description corresponds to the definition of Q.2931 protocol in all the aspects that we decided to handle in this work.

We also conclude that Q.2931 itself is an example of a well and carefully defined protocol. In all cases covered by our simulations, even with very "hostile" behaviour of the environment, the protocol turned out to be successful. However, in this context, it does not necessarily mean that requested connection is actually established. From the signaling point of view, success may also mean that if the call cannot be placed, the protocol gracefully recovers and terminates the establishment procedure. The crucial point for protocol definition is to make sure that, within a reasonable period of time, all involved parties perceive the call status in the same way (established or released). Q.2931 has shown to possess this feature.

Unfortunately, the ability to deal with numerous unexpected situations comes at a price of complex message verification and fault recovery. Error handling procedures contain many exceptions from general rules and provisions for treating particular events. In real implementations, it translates into time-consuming, processor-intensive algorithms, and it may incline a programmer to consider simplifying, or even discarding, certain — seemingly unnecessary — procedures. In this thesis, however, we discovered and presented selected examples of situations that justify the solutions proposed in the definition.

8.2 Future Work

We would like to suggest several possible directions for continuing this project. We do not intend to explore all opportunities, but rather wish to indicate the areas that we find particularly interesting.

The specification may be extended to include functionalities not handled in this thesis, such as point-to-multipoint connection procedures (defined by ATM Forum in [7]) or metasignaling (defined by ITU-T in [4]).

The official definition is not a final product yet; it is still in the development phase, so there will be a constant need for updating our formal specification to accommodate new elements and changes. For example, the ATM Forum has already announced ATM UNI 4.0, and ITU-T is working on their version of multi-party connections.

It would be interesting to combine our Q.2931 definition with the already existing specification of Signaling AAL protocol, SSCOP [25]. After the missing joint, Service Specific Coordination Function, is defined, we could simulate a complete ATM UNI signaling stack.

Validation methods used in this work are based on simulations, and, as we explained, they do not guarantee correctness. We believe that it would be very useful to design and conduct more formal verification procedures. One opportunities in this area is to look at some work on applying Petri Nets to Estelle specifications [38, 39], which we mentioned in Section 7.2.

There are many implementation specific or undefined issues in the documentation, and they should be subjects of further studies. For example, it is not known which states shall be considered incompatible by status procedures, and what kind of recovery actions shall be taken. As well, in some cases, protocol creators leave developers an option to either design their own error handling procedures or use the default connection clearing. These and other akin problems may be addressed in subsequent refinements of our description, so that, eventually, an implementation of the protocol can be built. We think that it would be also worthwhile to port such an implementation onto the real network environment.

Bibliography

[1] Chen, T.M., Liu, S.S., "ATM switching systems", Artech House, 1995.

[2] Alles, A., "ATM Internetworking", Cisco Systems Inc. Publication, 1995.

[3] Wajda, K., "Sieci szerokopasmowe" (Broadband Networks - in Polish), *Wydawnictwo FPT*, Krakow, 1994.

[4] ITU-T Recommendation Q.2931 "B-ISDN User-Network Interface layer 3 specification for basic call/bearer control", 1994.

[5] ITU-T Recommendation Q.931 "Digital Subscriber Signalling System No.1 (DSS 1) — ISDN User-Network Interface layer 3 specification for basic call control", 1993.

[6] ISO Document DTR 10167 "Guidelines for the Application of Estelle, LOTOS, and SDL", 1990.

[7] ATM Forum, "ATM User-Network Interface Specification, version 3.1", 1994.

[8] Zahir Ebrahim, "A brief tutorial on ATM", *http://dallas.ucd.ie/~ndowney/atm_intro.html*

[9] Le Boudec, J.Y., "Welcome to the LRC Tutorial Pages", *http://lrcwww.epfl.ch/PS_files/~tutorial.html*

[10] Reddivalam, S., "ATM Module", *http://cne.gmu.edu/~sreddiva/atm_module.html*

[11] Xylan Corporation, "The switching book",
http://www.xylan.com/sb/start.html

[12] ITU-T Recommendations Q.2761 "BISUP—Functional description", Q.2762 "BISUP—General functions of messages and signals", Q.2763 "BISUP—Formats and codes", Q.2764 "BISUP—Basic call procedures", 1993.

[13] ITU-T Recommendation Q.2130 "B-ISDN Signalling ATM Adaptation Layer — Service Specific Coordination Function for support of signalling at the User-to-Network interface (SSCF at UNI)", 1993.

[14] ITU-T Recommendation Q.2110 "B-ISDN ATM Adaptation Layer Service Specific Connection Oriented Protocol (SSCOP)", 1993.

[15] "Estelle: A Formal Description Technique based on Extended State Transition Model", International Standard ISO 9074: 1989 (E) (1989-07-15).

[16] Budkowski, S., Dembinski, P., "An introduction to Estelle: A specification language for distributed systems", *Computer Networks and ISDN Systems Journal*, vol.14, No.1, 1988.

[17] ISO 9074:1989/Amendement 1, Annex D, "Estelle Tutorial", 1989.

[18] Estelle Development Toolset (EDT), version 4.0, "General information and Estelle-to-C compiler (Ec)", *User Reference Manual*, INT Evry, France, 1996.

[19] Estelle Development Toolset (EDT), version 4.0, "Estelle Simulator/Debugger (Edb) and Universal Test Drivers Generator (Utdg)", *User Reference Manual*, INT Evry, France, 1996.

[20] Holzmann, G.J., "Design and validation of computer protocols", Prentice-Hall, 1991.

[21] Alkhechi, B., Benalycherif, M.L., Budkowski, S., Dembinski, P., Gardie, M., Lallet, E., Mouchel La Fosse, J.P., Octavian, C., Souissi, Y., "Formal specification, validation, and performance evaluation of the Xpress Transfer Protocol (XTP)", *Research report No. 931004*, Institut National des Télécommunications, Évry, France, 1993.

[22] Catrina, O., Lallet, E., "Contributions to the specification and validation of the Xpress Transfer Protocol", *Research report No. 931005*, Institut National des Télécommunications, Évry, France, 1993.

[23] Catrina, O., "Protocols for telecommunication networks: Development of complex communication protocols using Estelle FDT", *Extended abstract of the Ph.D. Thesis*, Politehnica University Bucharest, 1996.

[24] ATM Forum, *http://www.atmforum.com/* , *ftp.atmforum.com/pub*

[25] Ghodrat, M., "Specification and verification of the Service Specific Connection Oriented Protocol", *M.Sc. Thesis*, Concordia University, Montreal, 1995.

[26] Bihan-Faou, P., Mahamad, E., "Rewriting (part of) the ATM specification in LOTOS", *LOTOS Research Group report*, University of Ottawa, Canada, 1995.

[27] Hansen, H., "Connection management functions of a private wireless ATM network", *M.Sc. Thesis*, Helsinki University of Technology, 1996.

[28] Hudek, D., "Demo of (Java powered) UNI 3.1 Signaling Package", *http://www.ultranet.com/~dhudek/junidemo1.shtml*

[29] Cellware Broadband, "CELL_EXPRESS introduction", *http://www.cellware.de/software/q2931.html*

[30] Fore, "SPANS Protocol Specification Version 2.0", Fore Systems Inc. Publication, 1993.

[31] Armitage, G.J., "gNET: An ATM LAN signalling protocol", *Technical Report*, University of Melbourne, 1994.

[32] Tan, S.M., "An architecture for call processing", *M.Sc. Thesis*, University of Illinois at Urbana Champaign, 1993.

[33] Tan, S.M., Campbell, R.H., "x–ATM: A Portable ATM Protocol Toolkit", *http://choices.cs.uiuc.edu/latex.docs/suite/suite.html*

[34] XTP Forum, "Xpress Transport Protocol Specification, Revision 4.0", 1995.

[35] Cheung, J., "An Estelle Specification and Partial Validation of the Xpress Transfer Protocol", *M.Sc. Thesis*, Concordia University, 1990.

[36] Soumas, N., "An XTP Router in the Internet Addressing Domain", *M.Sc. Project Report*, McGill University, 1994.

[37] Jirachiefpattana, A., Lai, R.,"Verification Results for the ISO ROSE Protocol Specified in Estelle". In: *Protocol Specification, Testing and Verification*, XIV, eds. S.T. Vuong and S.T. Chanson, Chapman & Hall, IFIP, 1995.

[38] Jirachiefpattana, A., Lai, R., "Verifying Estelle Specifications: Numerical Petri Nets Approach". In: *Proceedings of the 1993 International Conference on Network Protocols*, IEEE Computer Society Press, 1993.

[39] Nuetzel, J., "Analysis and Verification of High-Level-Nets in Combination with Formal Estelle Specification", Petri Nets applied to Protocols, *Workshop of the 16th International Conference on Application and Theory of Petri Nets*, Torino, Italy, 1995.

[40] Amer, P., Burch, G., Sethi, A., Zhu, D., Dzik, T., Menell, R., McMahon, M., "Estelle specification of MIL-STD 188-220A data link layer", In: *Proceedings of MILCOM '96*, McLean, VA, 1996.

[41] Amer, P., Sethi, A., Fecko, M., Uyar, M., "Formal design and testing of army communication protocols based on Estelle", In: *Proceedings of 1st ARL/ATIRP Conference*, College Park, 1997.

[42] Amer, P., Çeçeli, F., Juanole, G., "Formal Specification of ISO Virtual Terminal in Estelle", In: *Proc. IEEE INFOCOM'88*, IEEE, New Orleans, 1988.

[43] Mills, D., "Network Time Protocol (Version 2) Specification and Implementation", *Technical Report*, University of Delaware, 1989.

[44] University of Delaware, *ftp.udel.edu/pub/grope/estelle-specs/*

[45] ISO/IEC Standard DIS 8802-6, "Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements – Part 6: Distributed Queue Dual Bus (DQDB) access method and physical layer specifications (Formerly DAM 1)".

[46] ITU-T Recommendation Q.921bis "Abstract test suite for LAPD conformance testing", 1993.

Good starting points for ATM research on the Internet:

[47] Batsell, S., "ORNL Network Research Navigator - ATM Page", *http://www.epm.ornl.gov/~batsell/atm.html*

[48] Robel, A., Dent, C., "The cell relay retreat", *http://cell-relay.indiana.edu/cell-relay/*

[49] Koester, D., "Asynchronous Transfer Mode (ATM) Technology Web Knowledgebase", *http://www.npac.syr.edu/users/dpk/ATM_Knowledgebase/ATM-technology.html*

Appendix A

E.164 Addressing Format

An ATM address uniquely identifies the ATM endsystem in the network(s). The ATM Forum specification uses three different formats of the address: *E.164 numbering* defined by ITU-T Recommendation E.164, *Data Country Code* (DCC), and *International Code Designator* (ICD). The specification recommends the support of all three formats for private networks and either E.164 or all three formats for public ATM. In our simulation model, only E.164 is currently supported. The general structure of the E.164 ATM format is presented in Figure A.1.

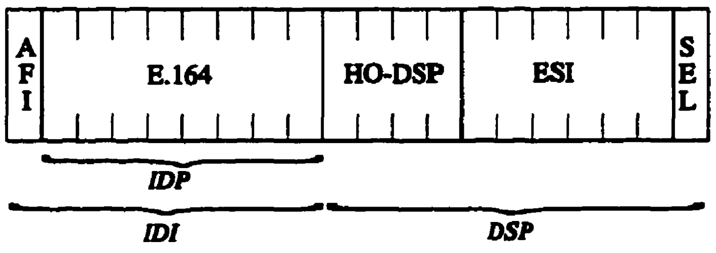


Figure A.1: E.164 ATM address format.

An ATM address is always 20 octets. It consists of two main parts: *Initial Domain Part* (IDP) and *Domain Specific Part* (DSP).

The first octet of IDP is *Authority and Format Identifier*, AFI. It identifies the administrative authority that allocated the number and the format of the remaining part. For E.164 numbering the value of AFI is coded to 45. The next 8 octets, *Initial Domain Identification* (IDI), specify the ISDN telephone numbers in their international form. They can be up to 15 digits long and they are coded in Binary Coded Decimal, BCD, syntax (i.e., one digit takes one semi-octet, two digits form one octet). If the telephone number is less than 15 digits long, it is padded with leading 0000 semi octet to obtain the maximum length. The address is ended with the 1111 semi octet after the last digit to obtain an integral number of octets (note that 15 digits takes only 7.5 octets and the IDI is 8 octets).

Domain Specific Part, DSP, consists of *High Order DSP* (HO-DSP), *End System Identifier* (ESI), and *Selector*. HO-DSP is used by the authority identified in IDP to divide the domain into separate subdomains. It defines the hierarchical structure of the authority's networking resources. ESI identifies an end system within the subdomain created by HO-DSP. It takes 6 octets and must be unique within the particular value of the IDP+HO-DSP. The last octet, Selector, is not used by ATM routing but may be used by endsystems.

In our implementation of the specification, ATM addresses are produced by the procedure `produceE164Addr(phLikeNr, HoDsp, IntNumber, VAR atmAddr)`.

phLikeNr ("phone-like number") is a string of 15 digits forming the telephone number [1]. **HoDsp** is a string of 8 digits, and it identifies the domain within phLikeNr number. Both phLikeNr and HoDsp are inserted accordingly into the ATM address, but they are actually not used by the Network module for routing; so their values do not have any meaning for the current implementation.

For the purpose of identifying the interface in our simulation model, we use **IntNumber** value, which is encoded in the first octet of ESI (i.e., 14th octet of the whole address). Currently, only this octet is used by Network module for routing, so the number of interfaces participating in the simulation is limited to 256. Remaining five octets of ESI and Selector are coded to all 0s.

The resulting 20 octets of the complete ATM address are inserted into the **atmAddr** output parameter.

[1]phLikeNr supplied to the procedure must have exactly 15 digits — in case of shorter numbers, the programmer is responsible for providing leading 0s.

Appendix B

List of Acronyms

A	
AAL	ATM Adaptation Layer
AFI	Authority and Format Identifier
ANSI	American National Standards Institute
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
B	
BCD	Binary Coded Decimal
B-ISDN	Broadband Integrated Service Digital Network
BISUP	Broadband ISDN User Part
C	
CC	Call Control
CP-AAL	Common Part AAL
CS	Convergence Sublayer

D	
DCC	Data Country Code
DSP	Domain Specific Part
E	
ESI	End System Identifier
ETSI	European Telecommunications Standards Institute
F	
FDT	Formal Description Technique
FIFO	First-In-First-Out
H	
HO-DSP	High Order DSP
I	
ICD	International Code Designator
IDI	Initial Domain Identification
IDP	Initial Domain Part
IE	Information Element
IP	Interaction Point
ISDN	Integrated Service Digital Network
ISSI	Inter Switching System Interface
ITU-T	International Telecommunication Union
N	
NNI	Network Node Interface (also known as Network Network Interface)
Q	
QoS	Quality of Service

S	
SAAL	Signaling ATM Adaptation Layer
SAR	Segmentation And Reassembly
SDH	Synchronous Digital Hierarchy
SONET	Synchronous Optical Network
SSCF	Service Specific Coordination Function
SSCOP	Service Specific Connection Oriented Protocol
T	
TDM	Time Division Multiplexing
V	
VCI	Virtual Connection Identifier
VLIE	Variable Length Information Element
VPCI	Virtual Path Connection Identifier (equivalent to VPI)
VPI	Virtual Path Identifier

IMAGE EVALUATION
TEST TARGET (QA-3)