NOTICE

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

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

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

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

AVIS

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

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

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

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

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

Canada

A TOOL FOR SEMI-AUTOMATED PROTOTYPING OF COMMUNICATION PROTOCOLS

by

Charles O. Amalu

A thesis
presented to the University of Ottawa
in fulfillment of the
thesis requirement for the degree of
MASTER OF COMPUTER SCIENCE (M.C.S)
in

Department of Computer Science
University of Ottawa
Ottawa Ontario
Canada

Ottawa, Ontario, 1987

ISBN 0-315-40681-X

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The University of Ottawa requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

ABSTRACT

The development of protocol implementations is often a daunting task. This task is simplified when the development of the protocol implementations is based on a formal specification of the protocol. This thesis presents an semi-automated prototyping tool for the development of protocol implementations.

The semi-automated prototyping process generates C programming language source code from formal specification of communication protocols written in ESTELLE. This generated source code is highly portable.

The prototyping process was accomplished by using widely available software tools such as LEX and YACC. The tools were used to produce an ESTELLE compiler in the C programming language. The prototyping tool was developed on a VAX 11/750 running UNIX 4.2.

The thesis describes the methodology used to develop the semi-automated prototyping process based on the OSI Reference Model. The design and implementation issues related to the semi-automated development of protocols are also discussed. A major contribution of the thesis is an approach for handling target environment issues such as process management, inter-process communication.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. S. I. Omar and Dr. R. L. Probert for their contribution to the development of this methodology.

The author acknowledges the immense contribution from his wife. Her persistence and encouragement ensured the completion of the thesis.

CONTENTS

FIGURES

TABLES

Chapter I

INTRODUCTION

1.1 BACKGROUND

Over the past five years, the implementation of communication protocols from formal specifications has received increased attention from members of the software engineering community. This implementation process has been primarily manual to date. The need for formal specifications of communication protocols has led to the development of a wide variety of formal description methods [ANSA82a, ANSA83, ANDE84, BLUM82, CCITTa, ISO82a, ISO82b, ISO84, ISO85, ROCK82]. (Additional formal description methods are found in [AGGA84, AYAC81, BOCH78, BOCH80a, BOCH80b, BOCH85, BRIN84, BURK84, CCITTb, CHUNG84, DANT80, DIAZ83, EXEL82, HAAS85, LOGR84, SCHW81, SUNS82b, TENG79].) These methods are aimed at making the formal specifications of protocols more precise, more complete and easier to understand. To date, however, the use of the formal description methods is not widespread.

Confronted with this challenge, the communities of the International Standardization Organization (ISO) and CCITT embarked on the development and standardization of formal description techniques(FDTs). These description techniques are intended for Open System Interconnection (OSI) [DESJ81, IBM80a, ISO81, TANE81, ZIMM80]. On realizing the merits of formal description methods, the International Business Machines (IBM) Corporation has set out to describe completely, its Systems Network Architecture (SNA) [IBM80a, TANE81] using a meta-language called FAPL (i.e. Format And Protocol Language) [IBM80a]. In addition, there is on going work to use PDIL (i.e. Protocol Description and Implementation Language) [ANSA82a, ANSA83] to describe each layer of

the OSI Reference Model. These are only a few examples of the widespread activities in the area of formal description techniques.

Specifications written using these formal methods are intended to serve the following functions :

- as a standard reference model (or specification) which completely and unambiguously describes the particular network architecture.
- as an authoritative concise reference for verification and validation of protocol designs and implementations.
- as a formal protocol specification, used to generate a number of retargeted implementations of that protocol.

Given the technical capability to write precise and unambiguous protocol specifications using formal description methods, the next goal is to find cost-effective ways of implementing these specifications. The approach presented in this thesis, called semi-automated prototyping, is a cost-effective way of generating implementations of formal protocol specifications. Compared to manual implementation, semi-automated prototyping will produce reliable and portable implementations more efficiently [BOCH82a, BOCH82b, BOCH84a, BOCH84b, GERB83].

Our technique involves the use of parser generators, namely LEX and YACC, to generate a compiler for formal protocol specifications written in the FDT ESTELLE. The FDT ESTELLE was selected because it is a candidate international standard. The ESTELLE compiler is used to produce a C programming language source code for the prototype. The C programming language was chosen since it is widely used by the industry in telecommunication systems. The generated source code is compiled by the C language compiler to produce the executable machine code for the prototype.

1.2 *MOTIVATION FOR AUTOMATED PROTOTYPING*

The first motivation of this work is to provide consistency of distinct implementations of the same protocol. A network architecture such as the OSI Reference Model, is realized by many different implementations. These implementations are usually produced independently by different groups of people. These groups may be using the same formal specification. Each must accurately implement the OSI communication protocols. Some of the problems posed by this traditional approach are :

- considerable overhead from duplication of efforts. For example, in each level of a step-wise design of a formal communication protocol specification, each refinement of some part(s) of the specification would require a new prototype.
- also, a misinterpretation of a formal protocol specification by an implementor may result in an incorrect implementation which may be very difficult and expensive to uncover during testing.

These problems are usually magnified as the number of implementations of user applications increase.

Consequently, a large number of protocol designers are interested in the semi-automated implementation of formal protocol specifications. The trend now is towards the use of meta-languages such as, ESTELLE, FAPL, PDIL, LOTOS, SDL, RSPL, SL1, CHILL, SIMPL/1, and LC/1 [ANSA82a, ANSA83, AYAC81, BRIN85, CCITTa, EXEL82, IBM80a, IBM80b, ISO82b, ISO84, ISO85, LINN85, ROCK82, SCHI79, SCHW81, VALE84] for the formal specification of protocols. These languages contain some high-level programming language constructs and data types which are used for structured design of formal specifications.

1.3 *RELATED WORK*

1.3.1 Semi-automated Prototyping at the University of Montreal

There is on going work at the University of Montreal to automate the generation of prototypes of formal communication protocol specifications. The protocols are specified using an augmented version of the ISO subgroup B's FDT [ISO82b, ISO84].

An FDT compiler [GERB83] was used to produce prototype implementations in PASCAL. Given a formal communication protocol specification in FDT as input, the compiler produces PASCAL procedures. The procedures can be compiled separately. One PASCAL procedure is generated per module type in the specification. Instances of the module type are created by executing the PASCAL procedures for each protocol connection [ZIMM80]. An instance of the module type is used for each protocol connection.

Thus, the instance of a module transition is executed by invoking the corresponding PASCAL procedure with the enabling conditions [ISO82a, ISO82b, ISO84, ISO85]. The enabling conditions include the input interactions [ISO84, ISO85]. The input interactions (if any) are passed as arguments to the procedures at invocation. Included in the argument list are: the data for the running instance of the module and pointer to the data structures associated with the other modules to which the given module type is is connected.

A simple round-robin global scheduler is used to determine which of the modules should be executed. This scheduler is manually coded to suit the particular protocol system. The problem of having more than one transition enabled in a given module instance is handled by executing the transitions according to their relative position in the formal specification. Only the first transition enabled in the formal specification is executed.

For the channels between the modules, two kinds of mechanisms, a simplified rendez-vous and queues are used. For the rendez-vous, the sending module invokes the procedure corresponding to the receiving module. It is assumed that the receiving module is in a state where it can receive. In the case of queues, the sending module places the interaction on the receiver queue. The queue will in time be attended to by the scheduler.

The input/output operations are performed by calling the implementation-dependent input/output procedures.

Further details about this prototyping approach are available in [BOCH84a, GERB83].

The main drawback of this approach is the apparent poor utilization of existing Inter-Process Communication (IPC) facilities provided by the host Operating System. The use of IPC facilities is better than using shared (i.e. common) variables. Moreover, rendez-vous is no longer supported by ESTELLE.

By using a private scheduler, the approach does not fully utilize the scheduling capabilities of the host system. The use of private schedulers duplicate the services already provided by the host Operating System (OS), and can adversely affect the throughput of the host system thereby affecting the the performance of the communication protocol under development.

The work of the protocol implementor is further complicated by the scheduler complexity. Also, little consideration has been given to the portability of the generated source code. In addition, the system uses an FDT which is slightly different from the present day ESTELLE.

1.3.2 Semi-automated Prototyping at IBM Corporation

The aim of the semi-automated prototyping process at the IBM corporation is to compile formal specifications of SNA protocols [IBM80a, NASH83, POSM82, SCHZ80, TANE81] into intermediate high-level programming language programs which can then be compiled by existing compilers to produce machine codes. The intermediate languages include PL/I, PL/S and PL/DS [SHOR79, IBM74, IBM79].

The formal specification of SNA protocols are written using the meta-language, FAPL [IBM80a]. FAPL, a Format And Protocol Language, is a PL/I-like general purpose high-level language. In FAPL, Finite State Machines (FSMs) [BOCH78, BOCH82b] are represented by state-transition matrices. (Figure 1.1 below, shows a typical FAPL specification.)

State Names	RESET	PEND-DOC	QUIESCED
State Numbers	01	02	03
INPUTS			
R, RQ, EXP, QEC	-	>(r)	>(r)
S, -RSP, QEC	2	-	-
R, RQ, EXP, RELQ	-	-	-
S, -RSP, RELQ	-	1	1
R, RQ, NORM, CANCEL	-	-	>(s2)
S, RQ, NORM, ^CANCEL	-	>(s2)	>(s2)
S, RQ, NORM, ^CANCEL	-	-	>(s2)
R, RQ, NORM, QC	>(s1)	3	>(s1)
S, -RSP, QC	-	-	-
RESET /*From DFC-Reset */	-	1	1

OUTPUT CODE	FUNCTION
s1	SEND_CHECK_SENSE = X'0809'; /*Mode Inconsistent */
s2	SEND_CHECK_SENSE ='2006';
r	RECEIVE_CHECK_SENSE = X'0809';

Figure 1.1: A sample of a typical FAPL format.

Each row specifies an input condition that have to be satisfied, and the columns define the different states of the FSM. In the intersection of each row and column is the action-code to be performed when the input conditions are true and the FSM is in the state indicated by the column.

The action-code contains the next state of the FSM, and a label to the set of actions to be performed due to the state transition. A change of state is indicated by a new integer value while "-" indicates no state change. Error conditions are marked by ">" while "/" is used for impossible or "can not occur" events. The label identifier points to the output actions, if any, that have to be done. The output actions are defined by the FAPL statements in the matrix below the state transition matrix.

Given a formal protocol specification, the FAPL compiler or preprocessor generates a table-driven prototype in a high-level language program form of the selected intermediate language. The program or prototype is generated with respect to the target machine and the expected system environment in which the program would execute.

A dispatcher and scheduler are used to control the flow of messages in the protocol system. The flow of control within the system is achieved by a sequence of procedure calls and returns. For example, the FAPL "SEND" statement [IBM80a], transfers a message and execution control to another component of the SNA architecture by passing the appropriate information to the dispatcher. This introduces the concept of "pseudo-concurrency" since the invoked procedure will execute at some unknown time after.

Also, messages can be placed on queues which are associated with specific procedures. These queues are later serviced by the scheduler. When the dispatcher has no work to perform, control returns to the scheduler which then accepts input from any of the non-empty queues and transfers control to the corresponding procedure.

The scheduler is the core and the starting point of the generated prototype. Execution control starts and ends with the scheduler.

Each system dependent aspect of the prototype is implemented using macros. The macros are written in REX and have well-defined interfaces and functions. The body of these macros are tailored for each of the possible target environment. Consequently, the prototypes are highly portable.

The use of FAPL for protocol specification is cumbersome, and can be a drawback. In addition, the approach relies on a private scheduler.

1.3.3 Other Related Work

At the "Agence De L'informatique", project RHIN, researchers are working on semi-automated implementation of the OSI protocols. Formal protocol specifications are written using PDIL - Protocol Description and Implementation Language [ANSA83]. These specifications are read as input to the PDIL compiler which generates a prototype in PASCAL. Unlike the compilers used in the work at University of Montreal and IBM which are manually generated, the PDL compiler is produced via a semi-automated process.

At the University of Ottawa, a subgroup of the Protocol Research Group (PRG) is working on semi-automated generation of prototype implementation of formal specifications which are written using LOTOS - a temporal-logic based formal description technique under development by ISO [BRIN85]. One of the target implementation language is LISP [WILE84, WINS84]. Use of other languages such as PROLOG [CLOM81] is also widespread [BLUM83, LINN83a, LINN83b, PAVE84, POSM82, RAZO80, SCHZ80].

ESTELLE is also being used at Carleton University for the ARTT project [PEPP83]. In addition, a related work is being done by the National Bureau for Standards [NBS81a, NBS81b, NBS81c, NBS81d, NBS81e].

1.4 SCOPE OF THESIS

This thesis was based on a careful analysis of an earlier summer project at the University of Ottawa. The project started with the writing of a formal specification of Class 0 Transport protocol in ESTELLE, and the implementation of the specification in the C programming language. The implementation code was manually generated and targeted for the VAX-11/750 running UNIX 4.2 [BELL83a, BELL83b, BROW85, KERN78, KERN84, McGM83, WAIT84]. After analyzing the difficulties encountered in manually generating the executable prototype from the formal specification, we decided on investigating the feasibility of an semi-automated process by which executable prototypes of formal protocol specifications can be produced.

The class 0 transport protocol was chosen since we already have a specification and a manually generated executable prototype. The class 0 protocol is reasonably complex for our first attempt at semi-automated prototyping of formal protocol specifications. Furthermore, the semi-automated prototyping process can be used with some modifications, to develop even more complex protocols.

In addition to the Semi-automated Prototyping Process, the thesis discusses the System Dependent Aspects of semi-automated prototyping. These System Dependent Aspects are based on the use of ESTELLE as the formal specification language. A statistical report is presented which shows the amount of code produced by the ESTELLE compiler.

The adequacy of ESTELLE as a tool for specification of protocols is also assessed.

1.5 MAJOR CONTRIBUTIONS

This thesis main contribution is a technique for semi-automated prototyping of communication protocols. The technique involves the use of LEX and YACC to generate a compiler of formal protocol specifications written in ESTELLE. The ESTELLE compiler is used to produce a C programming language source code for the prototype. The generated

source code is compiled by the C language compiler to produce the executable machine code for the prototype. Distinct features of our approach includes: the C Language Prototype, the Extensions to the C Programming language, the Compiler (including the Lexical Analyzer and Parser), and the formal protocol specifications.

In addition to the semi-automated prototyping tools, the thesis discusses the System Dependent Aspects (SDAs) of the semi-automated prototyping of formal protocol specifications which are written in ESTELLE; the adequacy of ESTELLE for the specification of communication protocols used in semi-automated prototyping; and the design and implementation issues related to semi-automated prototyping of communication protocols.

The other contribution is the system interface issues of semi-automated generation of portable prototypes of communication protocols.

1.6 OUTLINE OF THESIS

Chapter 2 briefly presents the OSI Reference Model. A brief description of the layering concepts and communication between adjacent and peer entities of the OSI Reference Model are discussed.

Chapter 3 discusses the essential issues related to the development of protocols from the formal protocol specifications.

Chapter 4 presents the design strategy employed in the implementation of the executable prototypes. The overall configuration of the communication system is also discussed.

Chapter 5 discusses the semi-automated prototyping process. A complete step by step approach is employed in describing the process. Included in the chapter is how to use the existing software utilities such as YACC and LEX to produce the ESTELLE compiler.

Chapter 6 demonstrates the use of our approach on a formal specification of the Class-0 Transport Protocol.

Finally, in chapter 7 our impression of this approach is presented followed by a review of the goals and a statement of area for further endeavour.

1.7 ABBREVIATIONS

APP	Semi-automated Prototyping Process
ASP	Abstract Service Primitive
CCITT	Consultative Committee Institute for Telegram and Telegraph
CDP	Communication Daddy Process
CEP	Connection End-Point
FDT	Formal Description Technique
IPC	Inter-Process Communication
ISO	International Standard Organization
NSDU	Network Service Data Units
NSP	Network Service Provider
OSI	Open System Interconnection
PDU	Protocol Data Unit
PSP	Provider Service Primitives
SAP	Service Access Point
SP	Service Primitive
TCEP_ID	Transport Connection End-Point Identifier
TSDU	Transport Service Data Unit
USP	User Service Primitives
YACC	Yet Another Compiler-Compiler
LEX	A Lexical Analyzer Generator
Protocol	A set of rules or convention by which peer entities communicate.

Provider Primitives
Abstract Service Primitives that should only be used by the Provider entity in the OSI Reference-Model.

User Primitives

Abstract Service Primitives that should only be used by the User entity in the OSI Reference Model.

Chapter II

FORMAL SPECIFICATIONS OF PROTOCOLS USING ESTELLE

This chapter briefly presents the OSI Reference Model. A brief description of the layering concepts and communication between adjacent as well as between peer entities of the OSI Reference Model are discussed. In addition, the chapter presents the basic concepts related to the formal specification of communication protocols used in our automated prototyping process. The protocol specifications are written in ESTELLE [ISO85, ISO84].

2.1 THE OSI REFERENCE MODEL

The OSI Reference Model of Figure 2.1, consists of seven protocol layers namely: Application, Presentation, Session, Transport, Network, Data Link and Physical layers.

The Open System Interconnection (OSI) Reference Model [ISO81, ISO84, ISO85, ZIMM80] is designed to provide a common basis for coordination of the development of standards for each of the seven protocol layers. It is used for the evaluation of existing standards and provides means for improving these standards.

In the real world, distinct communication systems are developed by different manufacturers whose systems must cooperate in order to provide wide range of meaningful services to their users. To provide the various services, the communicating systems must be capable of understanding one another, and are usually inter-connected by physical media. Figure 2.2 shows four systems interconnected by physical links, such as cables, satellite, radio waves, etc.

The purpose of the OSI reference model is to facilitate the communication between the distinct systems by providing rules which all communication system manufacturers must abide by when communicating with one another.

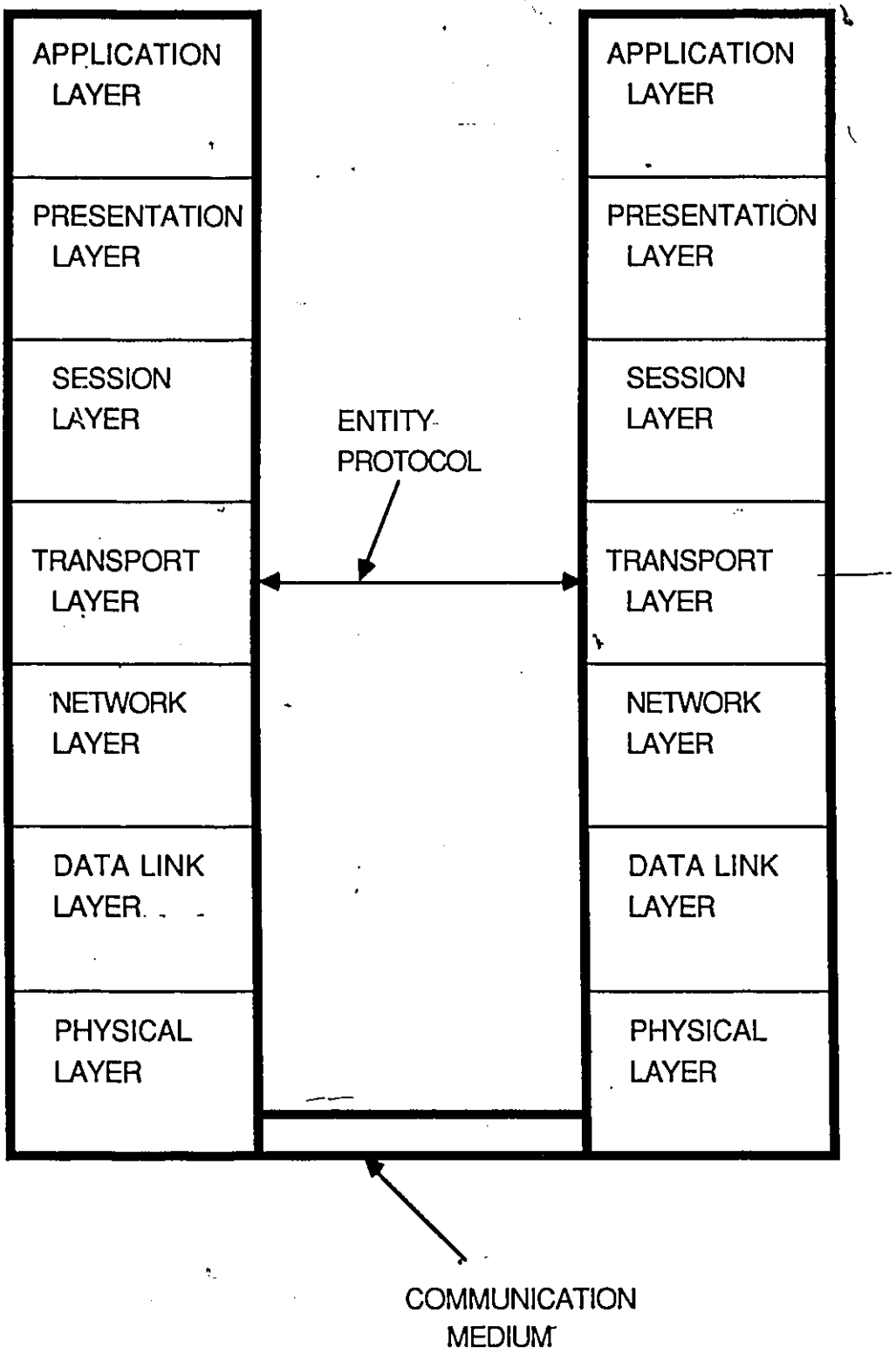


Figure 2.1: The OSI Reference Model

Furthermore, the OSI reference model encourages separate development of the various functional components of communication systems by partitioning of communication systems into functional layers. More details about the OSI model is provided below in the subsequent sections.

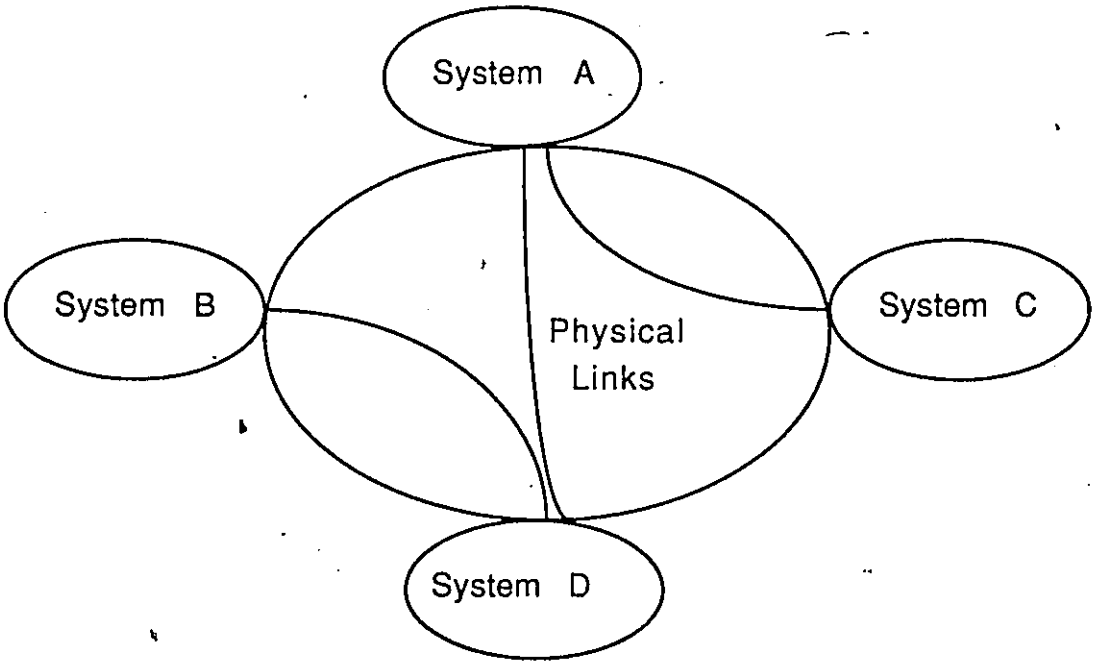


Figure 2.2: Systems Interconnected by Physical Links.

2.1.1 Reference Model Terminology

Protocol Entity (or Entity)

Each layer of the OSI reference model is implemented in an open system by an element called a protocol entity, or entity for short.

Peer Entities

Refers to protocol entities of the same protocol layer which are communicating by means of N-Layer protocol.

N-Layer Protocol

Refers to the rules of communication between peer N-protocol entities.

N-Layer Services (or N-Service)

Refers to the communication services provided to (N+1)-layer entities by the (N)-Layer entities.

Service Access Point (SAP)

This is the point at which two entities in the adjacent layers exchange information.

(N)-Service Access Point

Refers to the Service Access Point between a layer N entity and a layer N+1 entity.

Interface Data Unit

The unit of interface data (or information) exchanged between entities of adjacent layers.

Interface Data

Refers to the information exchanged between entities of adjacent layers.

(N)-Service Access Point Address (or (N)-Address)

Refers to the address associated with an (N)-Service Access Point.

(N)-Protocol Connection (or N-Connection)

Defines the protocol connection (or connection for short) between peer protocol entities in layer N.

Connection End Point

Each protocol connection has two connection end points: one at each end of the connection.

Connection End Point Identifier

This is the unique identifier associated with each connection end point.

2.1.2 The Layering Concepts

Layering is the basic structuring technique used in the OSI architecture [ISO81]. Each layer of the architecture performs a set of specific functions. The layers are represented for convenience in a vertical sequence as shown in Figure 2.3. The entities in the same layer on different systems are called peer entities. The entities in adjacent layers are called adjacent entities. See Figure 2.4 and Figure 2.5 for details. Except for the lowest layer (known as the physical layer), no direct physical link exists between the peer entities.

The highest layer utilizes the services of the lower layer which in turn is serviced by the layer below it. Layering shields the details of how such services are actually implemented. Layer N entities provide protocol services known as (N)-service to layer N+1 entities. Entities of Layer N are the providers and layer N+1 entities are the users of the (N)-service.

An entity may provide (N)-services to more than one (N+1)-layer entities. It may itself be serviced by more than

one (N-1)-entities. (See Figure 2.6.) An (N+1)-entity requests for (N)-service at an (N)-service-access-point. The (N+1)-layer and (N)-layer entities interact through the (N)-service-access-point using (N)-service primitives.

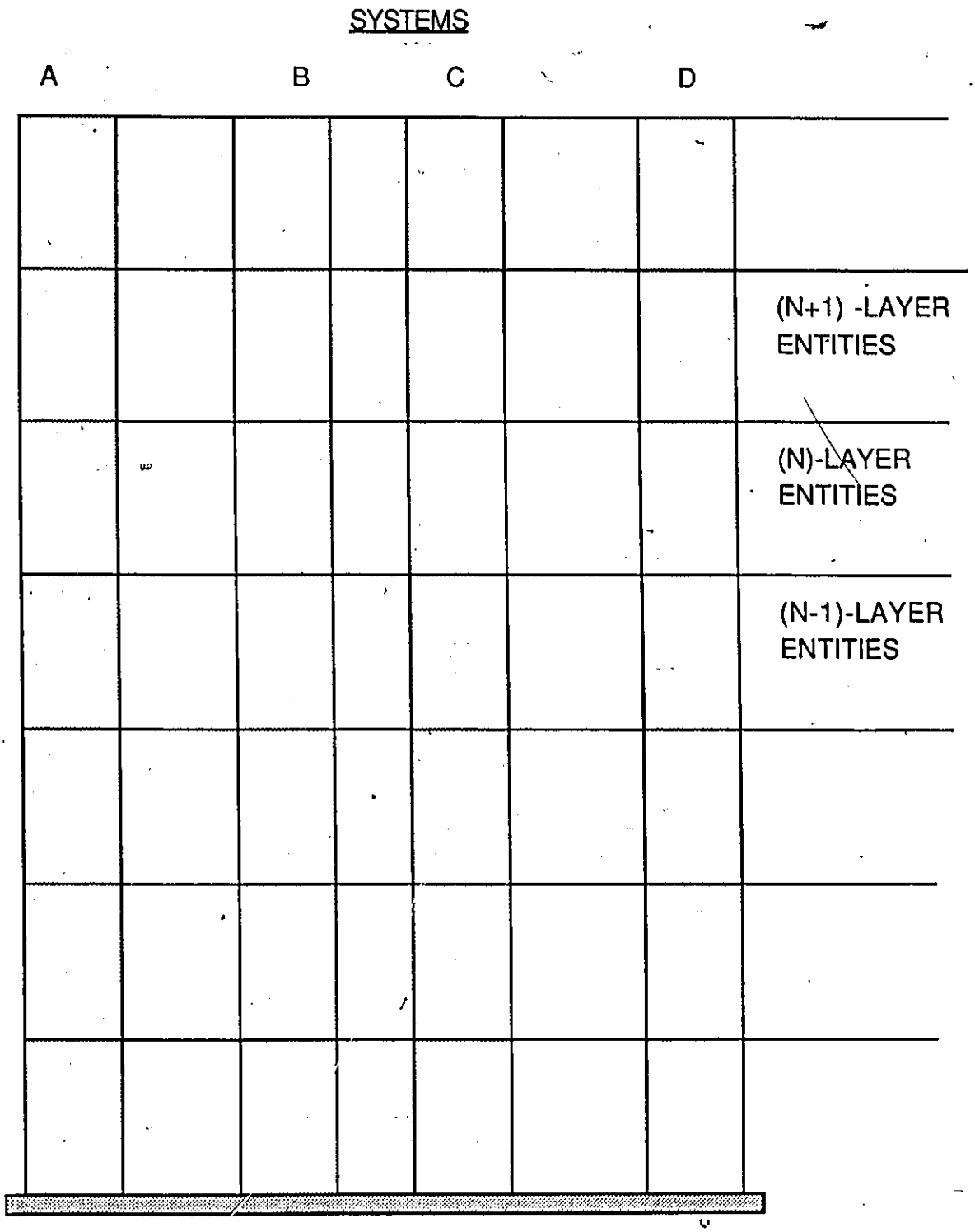


Figure 2.3: Layering in Communicating Open Systems.

The communication between the peer (N)-entities is conducted using the (N)-layer protocol. A connection is needed before entities can communicate. This connection is called an (N)-connection, and is established by means of a protocol. An (N)-connection is provided by the (N)-layer between two or more (N)-service-access-points. The point at which an (N)-connection is attached to an (N)-service-access-point is called an (N)-connection-end-point. The quality of the requested (N)-service is specified as one of the parameters of the (N)-service primitives used for connection establishment.

A (N)-service-access-point-address (or (N)-address for short), is associated with an (N)-service-access-point. An entity is associated with a (N)-address when an (N)-connection exists between the entity and another entity. Once such a connection is disconnected, the entity becomes disassociated with the (N)-address, and can no longer be reached via that (N)-address.

The unit of information exchanged between adjacent entities is called the (N)-interface-data-unit, and the unit of information exchanged by peer entities in layer N is called the (N)-protocol-data-unit. Data transfer between peer (N)-entities is carried through the (N-1) entities using the (N-1)-connection.

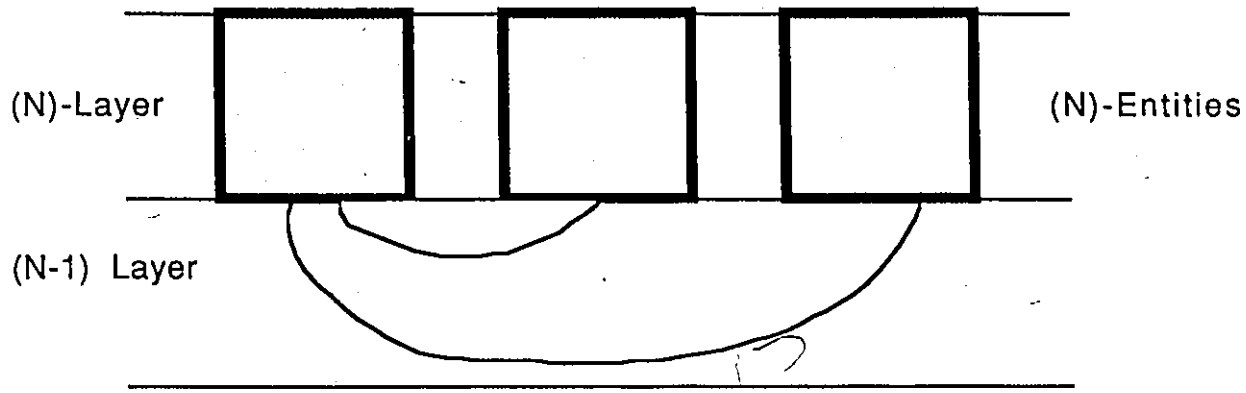


Figure 2.4: Peer Entities communicate through the lower layer

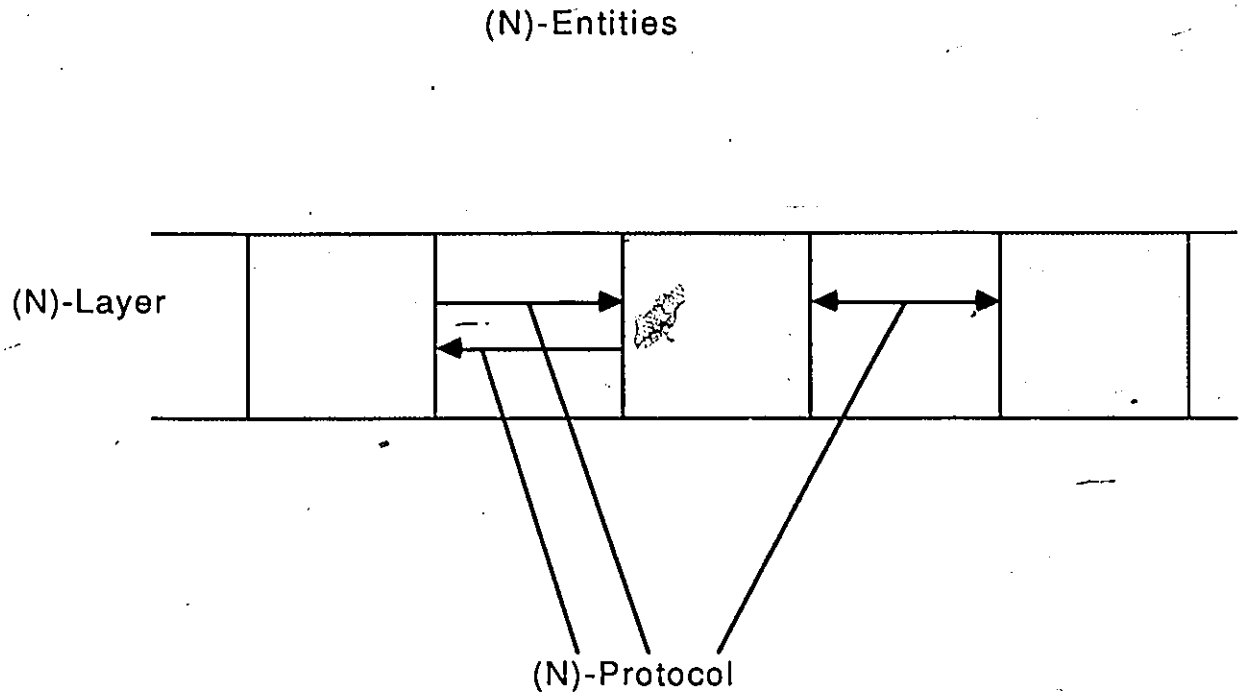


Figure 2.5: (N)-Layer Protocols.

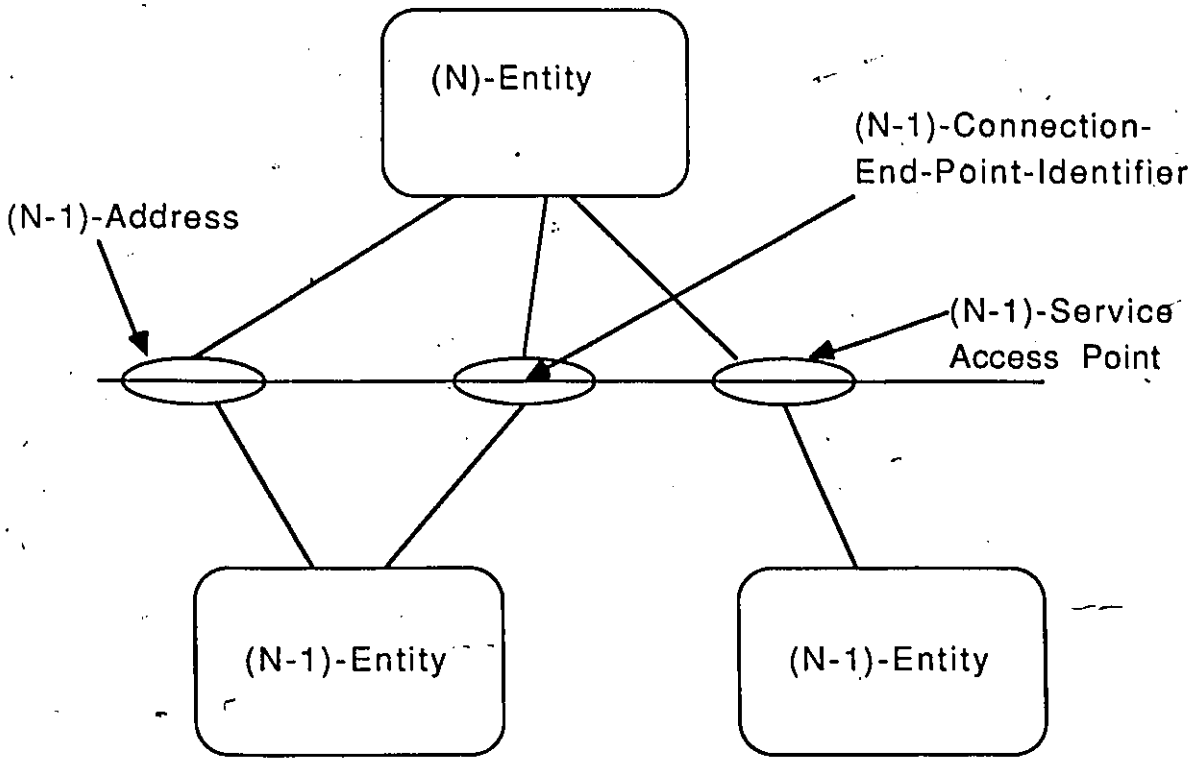


Figure 2.6: Entities, Service-Access-Points and Identifiers: The (N-1)-Address is the communication address of the (N-1)- Service Access Point.

2.1.3 The Model

This section provides a detailed functional description of each of the seven layers of the OSI Reference Model.

2.1.3.1 The Application Layer

The application layer consists mainly of user programs or processes. The layer is the user interface to a Network of communicating systems. It provides services directly to a user instead of a next higher layer. The peer entities of the application layer communicate using the application-layer protocol and the services of the presentation layer. Since the application layer interacts directly with a user, it is the apparent provider of services in a communication system.

2.1.3.2 The Presentation Layer

The presentation layer performs the functions that are commonly and frequently requested by users of a network architecture. These services normally relate to data transformations and automatic address look-ups.

The presentation layer represents information to communicating application entities in a form that is meaningful and syntactical correct. It provides the following services to the application entity:

- Data transformation
- Syntax selection
- Establishment of presentation-connection
- Error detection and correction

The Data transformation function involves mainly, code and character conversions such as data formatting, data encryption and data decryption. Data formatting deals with the modification of both the incoming and outgoing data such that they are meaningful to the users of the OSI architecture. The syntax selection facility allows the user to initially specify a syntax convention to be used in the presentation-connection. A presentation-entity establishes presentation-connections by sending a session-connection request to the session-layer entity. In addition, a presentation-entity may perform error detection or correction on a received data.

2.1.3.3 The Session Layer

The session layer provides the facilities for the communication between peer presentation-entities. The session layer organizes and manages the data exchanged between communicating presentation entities by establishing a session connection. The session connection is supported by a transport connection which is setup by the underlying transport layer. There is one-to-one mapping between a session connection and a transport connection.

To establish a session, the presentation-entity sends a request for a session-connection at a Session-Service-Access-Point (SSAP). The initiating presentation-entity is responsible for supplying the session-address of the destination presentation-entity. The session-connection exists only until it is released by one of the attached presentation-entities. A presentation-entity can only communicate with another presentation-entity by initiating a session-connection (or by accepting a session-connection request from another presentation-entity). Interaction between session-entities and presentation-entities are performed using session-service primitives. The data exchanged between the presentation-entity and a session-entity is called a session-service-data-unit (SSDU). The session entity maintains and monitors the conversation between the presentation-entities.

In summary, services provided by a session-entity include session-connection establishment, session-connection release, normal data exchange, expedited data exchange, session management, session-connection synchronization and exception reporting.

2.1.3.4 The Transport Layer

The transport layer deals with the fragmentation of data and merging of data from the session layer into small data units called Transport Protocol Data Units (TPDUs). It forwards these TPDUs to the network layer and then checks for their correct delivery at the destination peer entity.

The transport layer, in combination with the other lower three layers provide the transport service. The transport service is a transparent data transfer facility to the session layer. Since the network-service is responsible for the switching, routing, and relaying of messages, all protocols defined in the transport layer are end-to-end protocols. In fact, all protocols defined in the layers from and above the transport layer are end-to-end protocols. They are end-to-end since no intermediate peer entities are involved between the source and destination systems. The transport layer provides means for its users to establish, maintain and release transport-connections. It supports two-way simultaneous data path between a pair of transport-addresses. More than one transport-connection may exists between the same pair of transport-addresses. The transport-service user distinguishes between the Transport-Connection-End-Points (TCEP) by the use of Transport-Connection-End-Point-Identifiers (TCEP-ids). The negotiated quality of service specified during the establishment of transport-connection is maintained for the duration of the transport-connection.

While the transport connection exists, the transport-entity manages the connection; performing address mapping, multiplexing of other transport-connections to one or many network-connections, end-to-end error recovery, end-to-end data segmenting and blocking, and end-to-end flow control.

The transport layer provides up to five classes (Class 0, 1, 2, 3, and 4) of transport-services. The selected class is specified as a parameter of the Transport-Service-Primitives (TSPs). These classes of service are intended for the different parameter combinations such as throughput, transit delay, connection setup delay, residual errors and service availability. A transport-connection is possible only when both peer session-entities agree on these parameters. The unit of data exchanged between peer transport-entities is called the Transport-Service-Data-Unit (TPSDU). Two levels of data transfer services are provided, normal and expedited. When a session-entity requests for a session-disconnection, the transport-entity is responsible for terminating the transport-connection by informing the peer-entity of the intent to disconnect. A disconnection request may be initiated by the network-service-provider.

2.1.3.5 The Network Layer

The basic service of the network layer is to provide transparent transfer of all transport layer data. The network layer is responsible for establishing, maintaining and terminating the network-connections. A network-connection is the means of transfering data between transport-entities which are uniquely identified by their network-addresses. A network-connection is point-to-point, and more than one network connection may be required to join two network-addresses. Each network-address contains a Network-Connection-End-Point-Identifier (NCEP_id) which uniquely identifies the network connection.

Data transfer by the network layer is point-to-point and transparent to the transport entities. The unit of data transmission on a network-connection is called network-service-data-units (NSDU). Each NSDU has a distinct origin and destination. The integrity of these NSDUs are ensured by the relaying or intermediate network entities in the intermediary systems.

The network layer supports the selected quality of service for the duration of the network-connection. The quality of service parameters may specify the following:

- residual errors: occur from alteration, loss, duplication, disordering, improper delivery of NSDUs.
- service probability: probability that a requested network-connection will be established.
- reliability: average time between failures, and the mean elapsed time for recovering a network-connection.
- throughput: information transfer capacity.
- delay: include transit and network-connection establishment delays.

Unrecoverable errors are detected by the network layer and are reported to the transport-entity.

A network layer may provide a virtual or datagram service <ISO81, TANE81, ZIMM80> to a transport-entity. By virtual service, the network layer delivers the NSDUs to the transport entity in the order of their sequencing. The datagram service does not provide sequential delivery of the NSDUs. For virtual service, the actual routing of the NSDUs in the network may be datagram. The destination network entity must then buffer the NSDUs and deliver them in the proper sequence.

Flow control mechanisms are used by transport-entities to control the delivery of network-interface-data-units from the network layer. The network layer provides expedited network-service-data-unit transfer option. This is another means of reducing the transit delay of NSDUs.

Resetting network-connections is another service of the network layer. When a network-connection is reset, the integrity and proper delivery of any NSDU in transit is not guaranteed by the network service provider.

2.1.3.6 The Data Link Layer

The data link layer provides network layer with the capability to request for the assembly of data circuits within the physical layer. The unit of data exchanged over a Data-Link-Connection (DLC) is called the Data-Link-Service-Data-Units (DLSDU). In some networks, the DLSDUs are called packets in packet-switched networks and frames in other networks.

Data-link-connection-end-point-identifiers are used by network-entities to uniquely identify other network-entities. When requested, sequencing of the DLSDUs is provided by the data link layer. Detected unrecoverable errors are reported to the network layer.

Each network-entity can dynamically determine (up to the agreed maximum) the rate at which it receives data-link-service-data-units from the data-link-connection. This control may reflect the way in which a data link entity accepts data-link-service-data-units from the correspondent.

The data link layer also provides means of specifying the quality of service parameters such as mean time between unrecoverable errors, residual error rate, service availability, transit delay and throughput.

2.1.3.7 The Physical Layer

The physical layer is responsible for the transmission of data between entities of the data-link layer which are connected by a physical-connection. A physical-connection possibly consisting of a concatenation of data-circuits within the physical layer is offered to data-link-entities. Data-circuits are maintained by the physical layer for bit transmission between adjacent systems.

The unit of data transmitted between a Physical-Service-Data-Unit (PSDU) consists of one bit if a serial transmission is in use, and "n" bits if parallel transmission is in use. A Physical-Connection-End-Point-Identifier (PCEP_ID) uniquely identifys a data-link-entity in a multiplexed PCEP.

Bits are delivered by the physical layer in the order in which they were transmitted. On detection of error, the data-link-entity is notified of this error. The quality of service parameter may specify the error-rate, service availability, transmission rate and transit delay.

2.2 · PROTOCOL MODULES

Protocols are defined in ESTELLE using modules. The purpose of a module specification is to define the behaviour of the module as observable at its interaction points. Simply stated, interaction points of a module are the points at which the module interact with its environment.

The specification of a module can be given using any of the standard specification techniques available, including a single module, or substructured module definitions.

A single module definition implies that only one module is used in the formal protocol specification. A substructured definition of a module implies the module specification consists of a set of cooperating modules, namely; a parent module and one or more submodules.

The submodules interact with each other and the parent module to provide the expected functionality of the module under specification. In a substructured technique, the submodules may interact directly with the outside world. A submodule can itself be given as a substructured definition. Detail description of substructured definitions is provided in [ISO85]. In the subsequent discussions no attempt will be made to differentiate between modules and submodules since, submodules are modules themselves. Further, the terms "internal module" and "submodule" are used interchangeably in this document.

In ESTELLE, the behaviour of a module is defined by a state transition model. A state transition model consists of input and output interactions, states and transitions.

Since finite state diagrams or equivalent methods often lead to very complex representations, the finite state model is extended by the addition of variables to the states, parameters to the interactions, priorities and delay to the transitions. These extensions are introduced by programming language constructs such as data types and action statements which are used for manipulation of variables and parameters as well as for modules. This approach combines the simple concept of states and transitions with the power of a programming language (i.e. PASCAL).

The state space and transitions give the possible sequences of interactions of a module.

The state space of a module is specified by a set of variables. A possible state is characterized by the various possible values of each of these variables. The variable with the reserved name STATE, if it is defined, is the only one of these variables. The variable is sometimes referred to as the "major state" variable, and the state indicated by this variable can be considered to be the state of the FSM on which the module based.

Since the complete state of the module is specified by the "major state" variable and other variables, these other variables are often referred to as the "additional state variables".

For each transition in a module specification, the protocol designer would specify the state from which the transition is possible, the next state of the module and a list of actions to be performed. The action part normally includes an output interaction, and update of some additional state variables. Each transition is characterized by the following:

- present state:

  This is the state of the module before the transition is executed.

- next state:

  The state assumed by the module after the transition is executed.

- an enabling condition:

The enabling condition is a combination of one or more boolean expressions which must be satisfied before the transition can occur. Enabling conditions depend on the state variables, priority of the transition, and possibly an input interaction. Transitions with no input interactions are called **Spontaneous Transitions.**

- an action part:

  The action part is always executed as part of the transition. It may modify the values of the state variables, or output interactions to the environment. The execution of a transition within a module is atomic.

2.3 THE CONCEPTS OF MODULES AND INTER MODULE COMMUNICATION

The OSI Reference Model is viewed as a collection of inter-communicating modules. For the purpose of our discussion, the case of substructured definition of protocol entities is not required. Hence, each layer of the network architecture is assumed to be defined by a single module specification. In that case, an entity in any layer of the reference model is considered to be a module. The communication protocol between peer entities is defined by their respective module specification. As with the OSI reference model, the communication between peer modules (or protocol modules) are conducted through the intermediate modules in the lower layers of the network architecture. Each module can only communicate directly to the immediately adjacent modules through communication channels.

A channel is a bi-directional communication path as shown in Figure 2.7 and Figure 2.8. A channel supports a set of interaction commands called Service Primitives. For the reference model, a channel interconnects two directly adjacent modules, and defines the interface between the modules. In defining the channel between modules of the OSI Reference Model, an (N+1)-module plays the role of a USER, and a (N)-module plays the role of a PROVIDER. Depending on its role, the interconnected module is allowed to use a

subset of the Service Primitives provided by the channel. Currently, two roles are defined USER and PROVIDER.

The USER service primitives are initiated by the module which plays the role of the user of the channel. The PROVIDER service primitives are initiated by the module which plays the role of a provider in the channel.

The point at which a channel is attached to a module is called an Interaction Point. The collection of these interaction points and channels constitutes the Inter Module Communication.

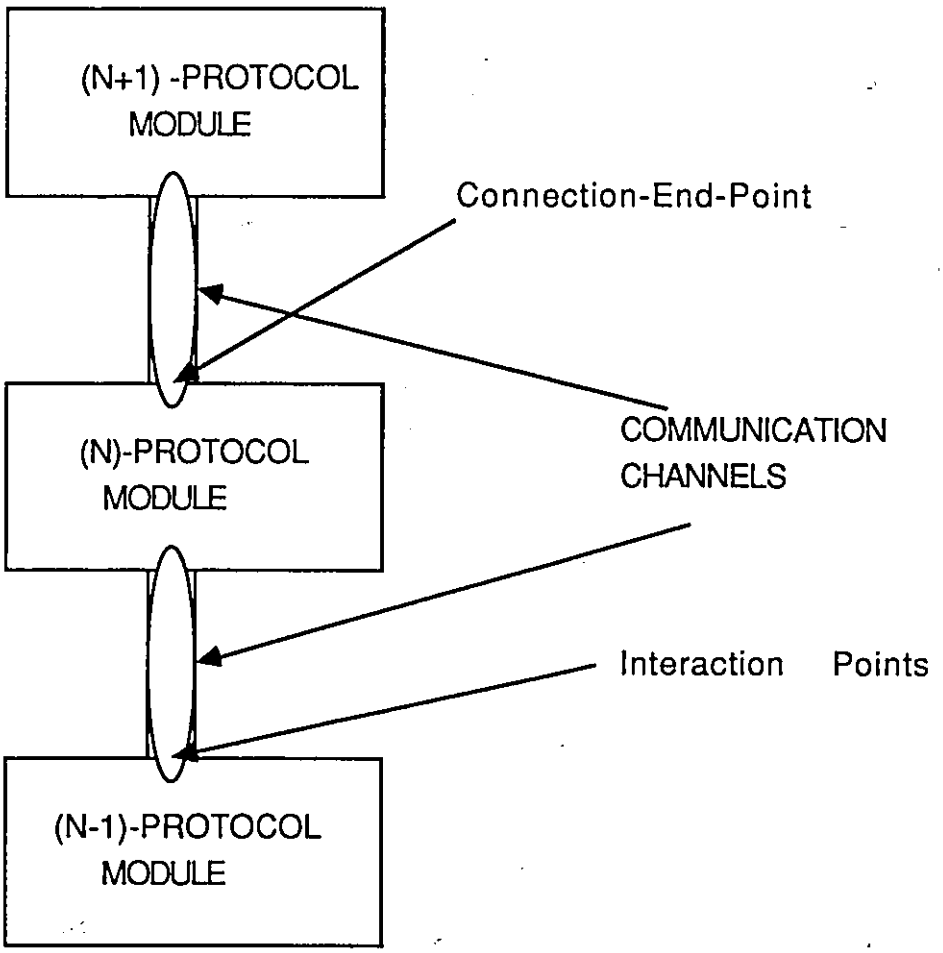


Figure 2.7: The Communication Channels between Modules.

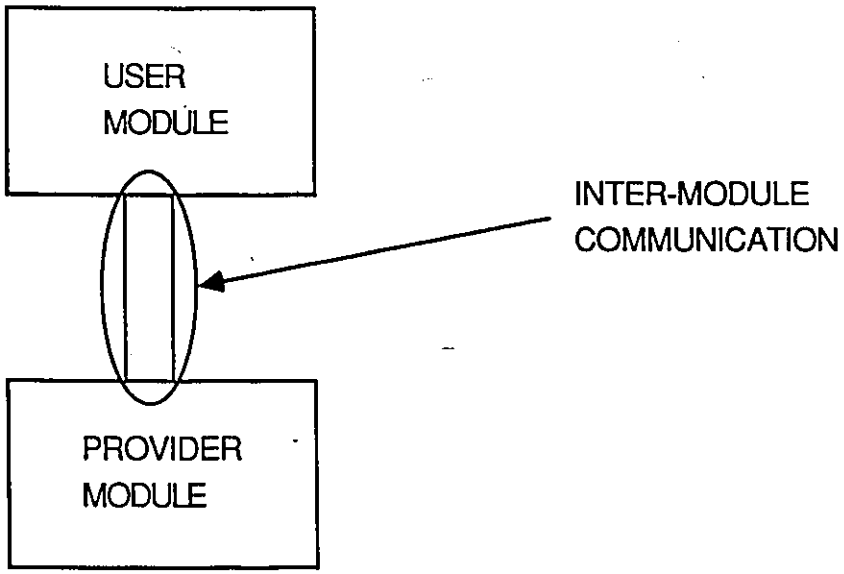


Figure 2.8: Inter Module Communication

2.4 SPECIFICATION OF PROTOCOL ENTITIES USING ESTELLE

As was previously stated, a module specification defines the behaviour of a protocol entity as observed at its interaction points from the environment. The environment of a module consists of the other modules interacting with the module. In this document, the specification of a protocol entity is presented in two sections: the specification of the channels through which a module interacts with its environment; and the specification of the module, describing the behaviour of the entity.

We will illustrate the general approach by referring to an example, namely the transport protocol. For this example, we employed the following ESTELLE language constructs which are supported by our ESTELLE compiler:

INITIALIZE clause:

used to specify the entry procedure.

CHANNEL clause:

used to specify the channel types, and roles of modules over the channel.

BY clause:

used to define the two sets of service primitives.

MODULE clause:

used to define the interaction points associated with the modules.

TYPE clause:

used for definition of data types.

CONST clause:

used for constant declarations.

TO clause:

used to specify the next major state of the underlying finite state machine

TRANS clause:

used for definition of a transition.

FROM clause:

used to specify the present major state of the underlying finite state machine. In other words, the value of the major state variable before the affected transition(s) can occur.

BEGIN and END clauses:

used to define the action part associated with a transition.

PROCEDURE clause:

used to define PASCAL procedures.

FUNCTION clause:

used to define PASCAL procedures.

SAME clause:

used to indicate no change in the major state required.

WHEN clause:

used for specifying the input interaction.

PROVIDED clause:

used to specify additional enabling conditions.

OTHERWISE clause:

used to specify the else part of a provided clause.

VAR clause:

used to define variables of various types.

OUT clause:

used to output interactions.

As well, we used the enabling conditions of the types:

- the present major state (FROM clause

- the input interaction (WHEN clause)
- the enabling condition (PROVIDED clause)
- the delay condition (DELAY clause)

The following enabling conditions were not used:

- the priority (PRIORITY clause)

In the action (or operation) part we used the following PASCAL language constructs:

- Simple Statements:

  assignment statement

  output statement

  nextstate statement

  goto statement

  procedure statement

  function statement

  return statement

  exit statement

  label statement

- Structured Statements:

  IF statement

  ELSE statement

  FOR statement

  DO statement

  WHILE statement

  REPEAT statement

  CASE statement

  BEGIN-END statement

2.4.1 Inter-Module Communication Specification

An inter-module communication specification consists of the channels accessible to the module, the Abstract Service Primitives, and the interaction points where the module is attached to this channel.

2.4.1.1 Specification of a Channel

In ESTELLE, a channel is defined as an abstract data type. The implementation strategy for the channel is left up to the implementor of the specification. The specification contains a list of the interaction primitives which must be supported by any implementation of the channel. These primitives are sometimes referred to as Abstract Service Primitives (ASPs) [ISO84, ISO85].

2.4.1.2 Abstract Service Primitives

The interaction primitives are said to be abstract because, no details on how such primitives are implemented is provided in the specification. The implementation of an ASP is dependent on the target environment.

A typical protocol specification in ESTELLE of both the User and Provider ASPs of a channel is shown in Table 2.1 below, describes the interface between the Session and Transport layer entities of the OSI Reference Model. Similarly, Table 2.2 shows the interface between the NSP and its users.

TS_access_point is defined as a channel which has two roles, USER and PROVIDER. This means any module accessing the channel, TS_access_point, can play either the role of the Provider or of the User, and not both. The list of interaction primitives and parameters a module can initiate over the channel is enumerated using the "by" clause. (Note, the case of more than two roles is still under study by ISO.)

As a syntactic and semantic requirement of ESTELLE, the types of the interaction parameters are defined before they are used. The new data types are declared in the type and constant sections that precede the channel declaration part.

Table 2.1: A sample specification of transport-session channel

```
channel
TS_ACCESS_POINT(USER,PROVIDER);
 by USER:
   T_CONNECT_request
           (TCEP_identifier     : TCEP_id_type;
            calling_T_address   : T_address_type;
            called_T_address    : T_address_type;
            QOTS_request        : quality_of_TS_type
            TS_connect_data     : TS_udata_type);
   T_CONNECT_response
           (TCEP_identifier     : TCEP_id_type;
            calling_T_address   : T_address_type;
            called_T_address    : T_address_type;
            QOTS_request        : quality_of_TS_type
            TS_connect_data     : TS_udata_type);
   T_DATA_request
           (TCEP_identifier : TCEP_id_type;
            TS_User_data    : TS_udata_type);
   T_DISCONNECT_request
           (TCEP_identifier : TCEP_id_type;
            TS_User_data    : TS_udata_type);

 by PROVIDER:
   T_CONNECT_indication
           (TCEP_identifier     : TCEP_id_type;
            calling_T_address   : T_address_type;
            called_T_address    : T_address_type;
            QOTS_request        : quality_of_TS_type
            TS_connect_data     : TS_udata_type);
   T_CONNECT_confirm
           (TCEP_identifier     : TCEP_id_type;
            calling_T_address   : T_address_type;
            called_T_address    : T_address_type;
            QOTS_request        : quality_of_TS_type
            TS_connect_data     : TS_udata_type);
   T_DATA_indication
           (TCEP_identifier : TCEP_id_type;
            TS_User_data    : TS_udata_type);
   T_DISCONNECT_confirm
           (TCEP_identifier : TCEP_id_type;
            TS_User_data    : TS_udata_type);
```

The same format is used for the specification of the channels between a module and

Table 2.2: A sample of channel declarations.

```
channel N_ACCESS_POINT ( USER, PROVIDER );
by USER:
 N_CONNECT_request (called_NSAP_id   : N_addr_type;
                    calling_NSAP_id  : N_addr_type;
                    NS_User_data     : TPDU_type;
                    quality_of_service: int_type);

 N_DATA_request     (TPDU : pdu_type);

 N_DISCONNECT_request
                   (NS_User_data : NS_u_data_type);

by PROVIDER:
 N_CONNECT_confirm (called_NSAP_id   : N_addr_type;
                    calling_NSAP_id  : N_addr_type;
                    NS_User_data     : TPDU_type;
                    quality_of_service: int_type);

 N_DATA_indication (TPDU : pdu_type );

 N_DISCONNECT_indication(NS_User_data : NS_u_data_type);
```

its internal modules. For example, the channel between a module and its internal timer module is specified as shown in Table 2.3.

Table 2.3: Transport Service Timer channel.

```
channel
   TIMER-ACCESS-POINT(USER,PROVIDER);
     by USER:
        TS_timer_start(period:time-type);
        TS_stop;
     by PROVIDER:
        TS_timeout;
```

2.4.1.3 Interaction Point Specification

The interaction points of any module are declared in the header of the module specification as shown in Table 2.4. The module header defines one or more interaction point identifiers which refer to a single channel or an array of channels. Each specified interaction point has associated with it, three attributes:

1. the channel type,
2. the role identifier (the User or the Provider), and
3. the queuing technique. The queuing options are: **COMMON QUEUE, INDIVIDUAL QUEUE** and **NOT QUEUED**. The **COMMON QUEUE** specifies a queue that is shared with other interaction points of the same channel type. The *INDIVIDUAL QUEUE* specifies a queue where each interaction point of the channel type has a separate queue. The *NOT QUEUED* option is the same as rendez-vous. Rendez-vous mechanism for ESTELLE has been left for further study by the ISO.

Table 2.4: A header for a Transport Entity for class 2 protocols

```
module TRANSPORT_ENTITY
      (U:array [1..20] of TS_access_point
            (Provider) common queue;
       N:NS_access_point(User) not queued);
```

In Table 2.4, the header for the Transport Entity is capable of supporting twenty TS_access_point interaction points through a shared queue. The module has a rendez-vous mechanism with the network layer.

2.4.2 Module Specification: an Extended Finite State Machine

The behaviour of protocol entities are modelled by the use of Extended Finite State Machines (EFSMs) [ISO82b, ISO84, ISO85, BOCH78, BOCH82b, CHUN84]. As described in "PROTOCOL MODULES" on page 31, the EFSM combines the simplicity of a finite state diagram and the processing power of a programming language to accurately specify protocol entities as modules in ESTELLE.

An instance of the EFSM represents a protocol entity. Further, one or more protocol connections can be supported by creating one or more instances of the EFSM (or module). An instance of a module specification is denoted by the collective instance of the state variables (i.e. the major state and additional state variables) and the interaction point variables used in the specification.

Instances of variables must not be confused with the range of values the variable can assume. By definition, a variable has many instances if within a program context, reference to the same variable identifier results in different storage addresses.

To support more than one protocol connection in one running instance of a module specification, a set of variables called **Context Variables** must be defined. These context variables are analogous to the Process Control Block (PCB) maintained by many Operating Systems for the various instances of a Computer Program called processes. For each protocol connection, the set of context variables contains of a major state variable and a subset of the available additional state variables.

To identify the correct state variables associated with a connection, the context variables are normally declared as an array of state variables, dereferenced using the connection identifier associated with each connection. Each element of the array maintains the relevant information pertaining to a protocol connection.

The usage of state variables (or context variables) again help minimize the problem of **State Space Explosion** which usually occurs when finite state machines are used to completely specify a real protocol such as the Transport protocol.

In summary, the state variables (or context variables) associated with each protocol connection must be unique in each protocol entity. These requirement may be satisfied usually by:

1. the creation of a process for each connection established by the entity. The process exists for the duration of the connection.
2. the creation of an array of records, called context record, with fields corresponding to the major state variable and additional state variables. Each element of the array is mapped to an instance of the connection. Table 2.5 shows a typical declaration of the major state and additional state variables used in a protocol specification.

Table 2.5: Sample Protocol system variables

```
var

TS: array [TCEP_id_type] of record
        remote_T_address : T_address_type;
        assigned_NC      : NCEP_id_type;
        .
        .
        .
        STATE            : major-state;
        .
        .
        .
    end;
```

3. Lastly, (1) and (2) above may be combined to produce a better and more complex structure.

The first technique is simple and practical, but expensive. It requires the operating system to schedule these separate individual processes. In addition, it requires the

operating system to dynamically, create these processes. Although this can easily be done on systems that run UNIX related operating systems, it is still expensive. Time is consumed when these processes are checkpointed [KERN84, BELL83a, BELL83b, McGM83] or swapped in memory. On UNIX, the system function, FORK [KERN84, BELL83a, BELL83b, McGM83], can be used to create these processes.

In the second case, the WITH clause [COOP83, GROG80, ISO84, ISO85] can be used to determine the scope and instance of the protocol module. However, the extra statements generated by the compiler for the WITH clauses is costly in real-time. The real-time required to resolve the variable accesses is minimized, if the protocol designer avoid using the WITH clause, and reference each field of the record explicitly.

The final technique is a combination of both techniques. The combination is ideal in a multi-user environment where a user of the system is allowed to have multiple connection to different protocol peer entities. This is the technique used in our implementation of the Class-0 Transport protocol. A User application process is limited to an arbitrary number of connections. This number is a constant in the ESTELLE compiler. The value of the constant can easily be changed to suite specific implementation requirements.

It is through the use of these context variables that the problem of state space explosion minimized.

2.4.3 Module Structure

A module structure consists of the initialization part, the transitions part of the module, and it may contain one or more internal modules (or submodules), if desired.

Internal modules have been considered and analyzed in our work. It was observed not to be substantially different from the ordinary modules. Their incorporation into our design can be done with minimal effort.

2.4.3.1 Initialization

For each instance of a protocol module, the state variables are initialized. The initialization is done:

1. at the time of system initialization,
2. before a connection establishment request, or
3. before a connection indication is accepted.

The choice of either technique is left to the implementor of the protocol. However, in our implementation, the initialization is performed per connection instance basis.

Table 2.6 shows a simple initialization block in ESTELLE. ESTELLE requires the initialization block be indicated by the clause, "initialize" [ISO84, ISO85].

Table 2.6: A sample of module initialization

```
initialize ;
begin
     STATE   to   CLOSED ;
     L_addr              :=  addr_undefined;
     R_addr              :=  addr_undefined;
     src_ref             :=  ref_undefined;
     dst_ref             :=  ref_undefined;
     L_User_id           :=  id_undefined;
     R_User_id           :=  id_undefined;
     L_net_addr          :=  naddr_undefined;
     R_net_addr          :=  naddr_undefined;
     remote_credit       :=  0;
     send_seq            :=  seq_undefined;
     trans_tpdu_buff     :=  empty;
     owner               :=  false
end;
```

2.4.3.2 Transitions

Each state transition of a protocol module is composed of the enabling conditions and the operation part.

An enabling condition is a combination of boolean variables that must evaluate to true before a transition can be executed. These boolean variables include the major state variable, the additional state variables, the input interactions and their parameters. The FROM clause is used for specifying the present major state of the connection instance. The TO clause is used to specify the next major state of the underlying finite state machine. The WHEN clause is used to specify the enabling input interaction from the environment. The enabling predicate and the delay condition are specified using the PROVIDED and DELAY clauses, respectively. While the PRIORITY clause is used to indicate the priority of the transition. The use of these clauses are well documented in [ISO84, ISO85].

The action part of a transition usually results in changes to the major state variable and additional state variables. Transitions contain output interactions which are specified using the OUT clause in ESTELLE. The execution of a transition in a protocol system is assumed to be atomic, and cannot be interrupted by any interactions from the environment.

The execution of these transitions is non-deterministic. That is, at any given time, one or more transitions may be enabled. The choice of which transition to execute is randomly determined by the protocol system, or determined by the priority of the transitions, or by a deterministic algorithm provided by the protocol implementor which gurantees that all affected transitions will eventually be executed.

Chapter III

DESIGN ISSUES RELATED TO AUTOMATED PROTOTYPING

This chapter presents the design issues related to automated prototyping of formal communication protocol specifications. First, design decisions must be made regarding:

- the formal specification language.
- the target implementation language.
- Entity-related issues (or Protocol dependencies).
- The target environment.

Then detailed aspects of the actual semi-automated prototyping process can be addressed, including:

- subsetting requirements,
- conformance testing,
- performance evaluation, and
- portability considerations.

In addition, some justification for ESTELLE and C programming language are discussed in the chapter.

3.1 THE FORMAL SPECIFICATION LANGUAGE

A desired formal specification language is one that results in clear, concise, complete and unambiguous specifications [AGGA83b, BLUM83, BOCH80a, BOCH80b, BOCH84a, BOCH84b, DANT80, MERL79, SUNS82a, SUNS82b, SCHW81]. The language must enable the design of relatively simple interfaces between protocol entities in any environment.

3.1.1 Clear and Complete Specification

The advantages of a clear and complete specification are numerous. A clear specification language is easy to understand. The data structures, scope rules and language constructs are properly explained and documented. A concise specification language does not contain redundant or duplicate language constructs. Learning the syntax and semantics of the language should not be overwhelming.

Paramount to every designer of formal specification language is the completeness of the language. The language must for its application, completely provide facilities or constructs for manipulating the expected data. All parts of the language constructs must be defined completely without the language becoming ambiguous, imprecise, unclear and unconcise.

Languages based on Extended Finite State Machine (EFSM) model have been used for specification of communication protocols [ISO85, ISO84, CCITTa, SCHI79, EXEL82, ANSA82a, ANSA83, IBM80a, AYAC81]. These formal specification languages reduce the possibilities of over specification of communication protocols, and are machine environment independent.

3.1.2 Implementation Ease

When choosing a formal specification language emphasis should be placed on the characteristics of the language useful for the implementation. This would result in less time spent on the language and more time devoted to the implementation of the specifications. The specification language should also have high-level programming language constructs for which a compiler can be generated. The language constructs should be simple and relatively easy to implement.

3.1.3 Simple Interfaces

The implementation of the interfaces are environment dependent. The specification of the interfaces between the modules and their environment must be abstractly specified to provide maximum flexibility to the implementor. Abstract specification of the interfaces enhances the portability of any formal specification. Enhancements are easily achieved by the use of Abstract Service Primitives (ASPs). The techniques for implementing the ASPs in the different target environment should be left to the protocol implementor.

3.1.4 An Example : ESTELLE

ESTELLE is a formal specification language released by the Communities of ISO [ISO84, ISO85] and CCITT [CCITTa]. It benefits from the precision of a finite state machine model and the data manipulation power of a high-level programming language. Because of the high-level language constructs of ESTELLE, specifications written in ESTELLE are usually very precise and well structured. Consequently, protocol specifications are relatively easy to read and understand. The specifications are independent of the execution environment.

ESTELLE provides means for abstract specification of interfaces (or channels) between protocol entities, and interfaces between protocol entities and the Host system. It does not describe how the channels are implemented. Details of their implementation is left to the protocol implementor. This provides the much needed flexibility.

Furthermore, a specification is more likely to be correctly implemented if its correctly understood. An implementation is easier to validate during conformance testing if the specification is well understood and a structured design has been used in the specification. This is one of the reasons we chose ESTELLE over other languages for the specification of our protocols.

3.2 THE IMPLEMENTATION LANGUAGE

The factors of the implementation language that should be considered are the Syntax and Semantics, and the Operating System service primitives available via the language. Data handling flexibility, machine portability of source code, modularity of source code, and maintainability of code are extra issues to consider. The design and maintenance effort of the programs should also be assessed. The availability of tools such as Text Processors, Parsers, Lexical Analyzers, and compilers which support conditional compilation of source files are very useful.

3.2.1 Syntax and Semantics

The task of implementing a protocol specification is minimized if the implemention language shares common syntactic and semantic constructs with the formal specification language. For instance, similar scope rules, similar data structures, and similar language constructs, are very useful. During compilation (or translations) of formal specifications, the similar constructs are copied with little or no extra code generated. Also, less time is spent in debugging, translation, verification and validation of the protocol implementation.

This is why the C programming language [WAIT84, KERN78, FEUG84, BROW85] was chosen for our implementation. Although PASCAL [BROW85, COOP83, FEUG84, GROG80] is closer to ESTELLE than C, PASCAL's strong typing and poor flexibility is a limiting factor. For example, PASCAL provides no facility for performing bit manipulations. Moreover, dynamic type conversions are tedious. Other criticisms of PASCAL are well documented in [FEUG84]. On the contrary, C programming language is very flexible and is as strong typed as PASCAL. Type conversions are easily performed by using CAST [WAIT84, KERN78, FEUG84, BROW85]. CAST is a keyword in C Language which allows programmers to cast a previously defined variable as another data type. Furthermore, an implementor can use the system utility LINT, to perform type checking on C programs. Since ESTELLE originated from a well structured programming

language, such as PASCAL, it imposes PASCAL-style structure on the implementations of protocols in C. The C Programming language is said to provide the structured design of a high-level programming language but with the capablity of low-level programming language.

It has been a common occurance to refer to C as a systems programming language while PASCAL is a system design language.

3.2.2 Operating System Interface

The implementation language must provide facilities for requesting different OS services, such as timer services, Inter-Process Communication (IPC) services, and error handling services. Preferably, these OS services can be requested by simple function calls or macros. The C programming language provides numerous functions and macros which are used to request OS services.

3.2.3 Portability of Source Code

For the implementation language to be portable, it has to be host system independent. As a result, it is required that all the functional parts (or interface) of the language to be well defined. During execution, these functional parts (or interface) are substituted wherever they are required by the use of macros or system primitives. The details of the implementation of the interface must not be included as part of the language, since they are host system dependent.

The benefit of using C programming language is that input/output (I/O) functions are not part of the language as in PASCAL. The I/O functions are provided as standard routines in the C language library. The I/O operations are very efficient since they are tailored to the particular environment.

3.2.4 Modularity of Source Code

Modularity is an important characteristic of the source code. A modular implementation language allows software modules to be independently tested, debugged and validated by any of the available software engineering testing techniques. Errors are easily detected and isolated to a particular section of the implementation code. Modularity allows future replacement of portion(s) of the implementation code without having to regenerate the entire code. Software modules can also be compiled, tested, and debugged separately before they are combined with other software modules. Top-down or bottom-up testing techniques [ANSA82c, BILL85, DUNN84, FREN85, HUNT81, MYER79, PROB83, URAL83a, URAL83b, URAL86, WEST78], can also be applied because of the hierarchical structure of the OSI Reference Model [ZIMM80, TANE81, DESJ81].

3.2.5 Maintainability of Source Code

Modular programs are usually easier to maintain than non-modular programs. Maintainability is essential, since enhancements are normal as implementors find better and faster means of implementing formal protocol specifications. Furthermore, it is expected that the standardization communities will make modifications to the formal specification language. By using a modular implementation language, these modifications are easily adapted into the protocol implementations without overhauling the design of the implementation code.

3.2.6 Availability of Support Tools.

Developing a compiler for a formal specification language is very expensive and time consuming. A more reasonable approach is to translate a formal specification into an intermediate programming language. Such a language may be a high-level programming language which already has a compiler or an interpreter. The selected intermediate programming language should provide tools for requesting host system services. If these

tools are provided as standard utilities in the host system then the task of implementing a formal specification in the target language is highly reduced and less error prone.

Some of these tools may be in the form of text processors which perform functions such as macro expansion, text translation and interpretation, text compression, and formatting.

The C compiler has a preprocessor which allows macro definitions, text file inclusions and conditional compilations. PASCAL has no such preprocessor. The conditional compilations or subsetting (as it is often called), is the inclusion of a block of text only if it is needed by a program under compilation.

3.3 ENTITY RELATED ISSUES

To implement the abstract concepts of a formal specification language such as ESTELLE, the constructs: channels, transitions, non-determinism of the transitions, and Spontaneous transitions, should be implemented. [COUR85, HANS85, JARD85, VENK85]. This is just a few of the many entity related issues that should be considered.

3.3.1 Channels

Communication between two adjacent entities are performed by the use of Shared Memory, Mailboxes, or Message Passing. The concept of shared memory requires access to memory be restricted to one entity at a time. To maintain integrity and avoid deadlock, a handshaking technique is required. The use of semaphores, LOCKs and UNLOCKs, or FLAGS is necessary for a successful implementation of the channels using Common Memory. To use mailboxes, the communicating entities must know the location of each others mailbox. The OS is responsible for the delivery of messages to the mailboxes. The OS may decide to schedule an entity once a message has been delivered to the entity's mailbox. Mailboxes are special cases of common memories. Message passing is the ultimate technique in communication. The entities do not need any *Handshaking*

mechanisms. Messages are first sent to the message transfer part of the OS, which determines the destination of the message from the message header.

A channel specification contains the ASPs which are supported by the channel type. During implementation, the ASPs are used as message signals which are included in the header of the message. The format of messages between adjacent protocol entities is determined by the message signal. For each signal (or message) type, two standard procedures, SEND and RECEIVE, are provided.

In our automated prototyping process, the different SEND and RECEIVE procedures are coded manually, and are very efficient.

3.3.2 Transitions

In order to implement transitions in ESTELLE, the enabling conditions [ISO85, ISO84] and the action parts of the transitions must be implemented. To test the input interactions and other enabling conditions, "IF-ELSE" (in C language) or "IF_THEN_ELSE" (in PASCAL) can be used. The other option is the use of the "SWITCH-CASE" [WAIT84, KERN78, FEUG84, BROW85] statements. A combination of both approaches were used in our implementation. The "SWITCH-CASE" statements were used for the implementation of the "FROM and WHEN" clauses, while "IF-ELSE" statements were used for other conditions, such as the PRIORITY and DELAY clauses. The justification of this approach is not difficult. The identifiers of the major states of the module are enumerated constants, and are used as labels in "SWITCH_CASE" statements. Service primitives are also denoted by enumerated constant identifiers used in "SWITCH-CASE" statements. The structure of our transition implementation is illustrated in Table 3.1 and in Table 3.2.

Table 3.1: The Structure for the Transitions.

```
switch (state)
{
from CLOSED:
        switch (event)
        {
           when T_CON_request:
                .
                .
                break;
           when T_CON_indication:
                .
                .
                break:
           when T_DISCON_request:
                .
                .
                break;
        } break;
from OPEN:
        switch (event)
        {
           when T_DATA_request:
                .
                .
                break;
           when T_DATA_indication:
                .
                .
                break;
      }  break;
   .
   .
   .
   .
```

Table 3.2: The Structure for the Transitions (Continued).

```
    .
    .
    .
    .
from WFCC:
     switch (event)
     {
        when T_CON_confirm:
            .
            .
            break;
     } break;
from WFTRESP:
       switch (event)
       {
          when T_CON_response:
                .
                .
                break:
          when T_DATA_request:
                .
                .
                break;
          when T_DATA_indication:
                .
                .
                break;
     } break;
from WFNC:
}
```

3.3.3 Non-Deterministic Transitions

Non-determinism of transitions [ISO85, ISO84] means no prior knowledge of which transition is executed (i.e. fired) even when more than one transition is enabled (i.e. firable) [DIAZ83, AGGA83a, AZEMA]. The choice of transitions to execute is left to the implementor of the protocol. Protocol implementors are allowed to select the firable transitions based on random techniques, or by deterministic method which guarantees all transitions equal chance of occurring (or being fired).

3.3.4 Spontaneous Transitions

Spontaneous transitions are those transitions not enabled by external input interactions. Such transitions may be useful in such things as dynamic allocation of buffers, routing, etc. Our design strategy includes provision for later support for this type of transitions. The actual implementation of spontaneous transitions can easily be incorporated into our implementation.

3.4 THE ENVIRONMENT AND SYSTEM DEPENDENT ASPECTS

The target environment of the executable prototypes deserves thorough assessment. The factors affecting the behaviour of the prototypes by hindering their proper implementations are isolated and resolved. Those favourable factors are incorporated into the design of the prototype.

For example, the environment of a transport layer module consist of the User processes and Provider processes, and the Operating System. The User processes utilize the services provided by the transport module while the Provider processes provide Network services to the transport module. From Figure 3.1 and Figure 3.2, the user processes are the implementation of the session layer modules, while the provider processes are the implementation of the network layer modules. The Operating System provides the necessary communication facilities needed by the User and Provider processes. The environment contains the System Dependent Aspects (SDAs). The SDAs are partitioned into two groups: the Protocol dependent aspects and the Operating System dependent aspects.

3.4.1 User and Provider Processes

The protocol dependent aspects of protocol module are determined by the specifications of the channels and Interaction Points between the protocol module and its User and Provider modules. These dependencies are classified as user-initiated (upper layer) and provider-initiated (lower layer) dependencies.

The User-initiated protocol dependencies are the dependencies resulting from the implementation of ASPs of the channels and Interaction Points between a protocol module process and its User Process(es). Some examples of User-initiated protocol dependencies are:

- Connection Establishment. Not all classes of Transport protocol support multiple connection between session modules.
- Error Management. Not all classes of Transport protocol support error recovery.

Similarly, the Provider-initiated protocol dependencies are the dependencies resulting from the implementation of the ASPs of the channels and Interaction Points between a protocol module process and its service Provider process(es).

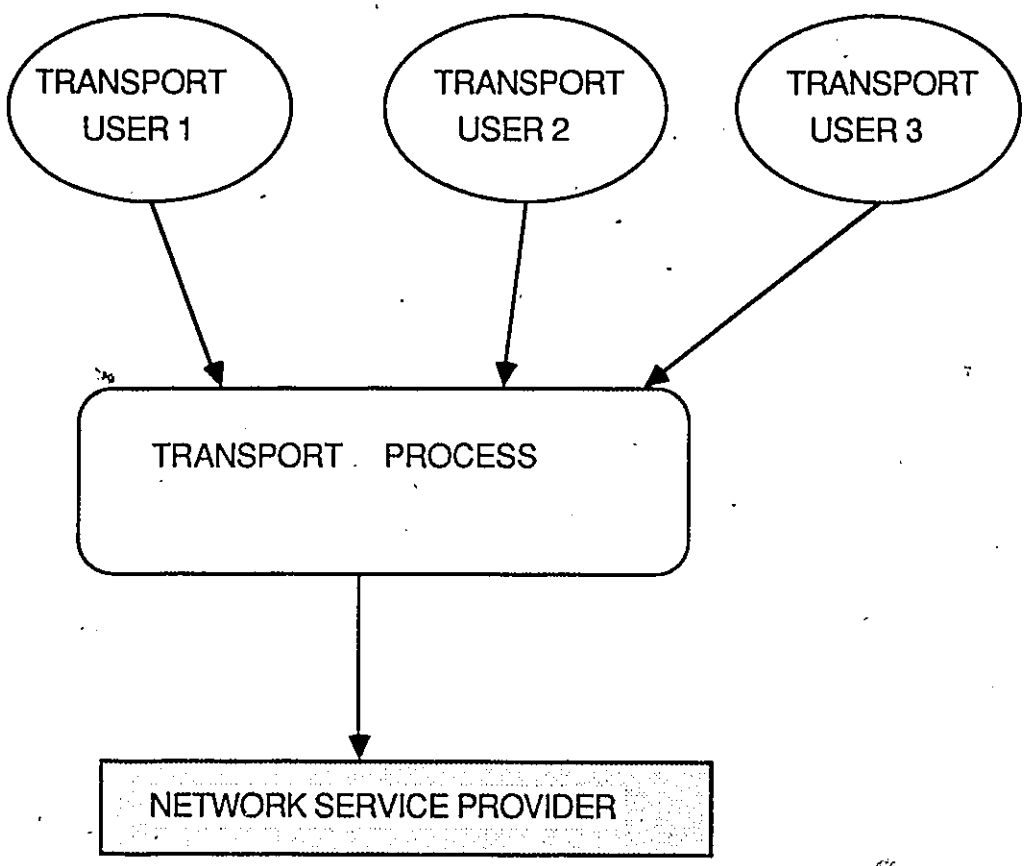


Figure 3.1: User Processes of a Transport Process.

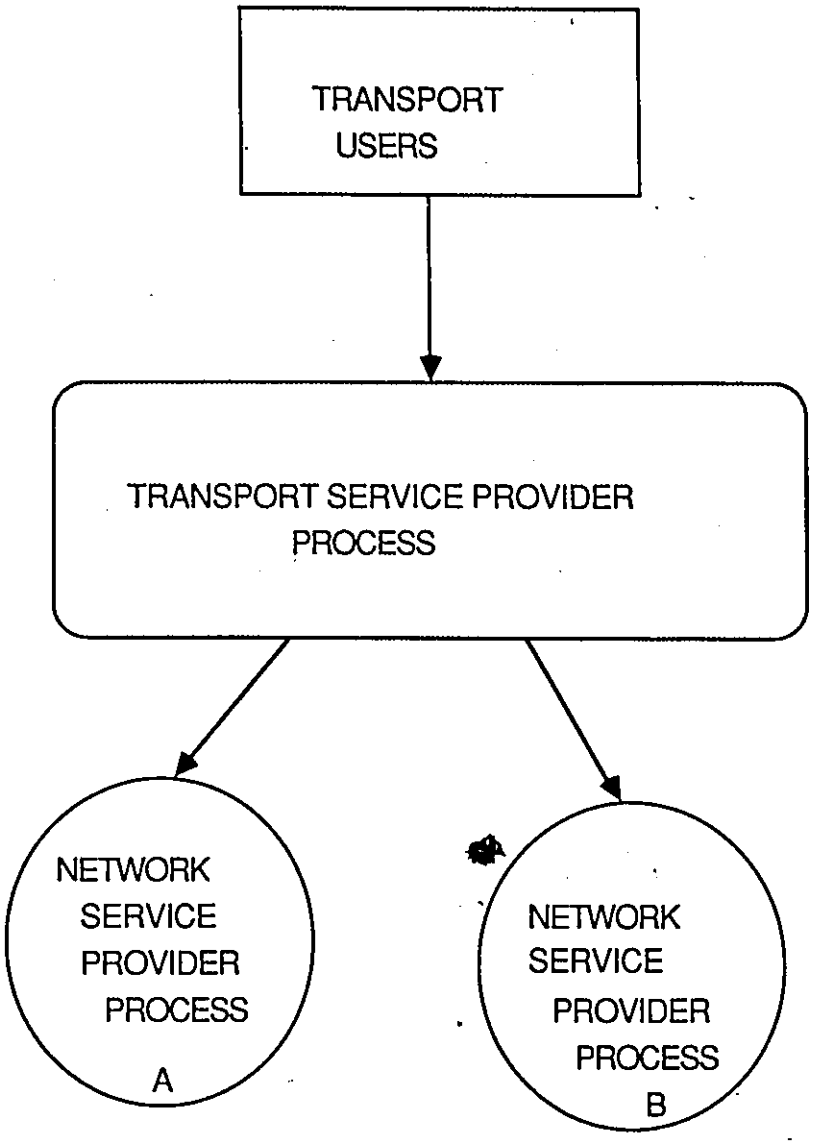


Figure 3.2: Provider Processes to a typical Transport Process.

3.4.2 The Operating System

In choosing an Operating system (OS), consideration is given to facilities provided by the OS, for Process Management, Inter_Process Communication, and Timing of individual process activities.

3.4.2.1 Process Management

The OS should provide simple and useful primitives for requesting the following process management services:

- Process creation
- Process scheduling and
- critical system error handling.

By definition, a process (a program under execution) is an instance of a program and consists of a set of instructions, data and control information. The instructions (or program) are executed by the Central Processing Unit (CPU) whenever the process is running. A Program is re-enterable when different processes can execute the same program. The instructions of the program are used to manipulate the data associated with each instance of the process. The control informations such as the process name, and Program Status Word (PSW) are used to manage execution of processes.

The requirements for the implementation of a process are OS dependent. The information that represents a process is stored in a Process Control Block (PCB). The PCB is an array containing most of the information shown in Figure 3.3. In some machines the required process information is too large to be stored in the PCB. Instead, the less frequently referenced information are stored in a secondary storage location called the Extended (or Secondary) Control Block.

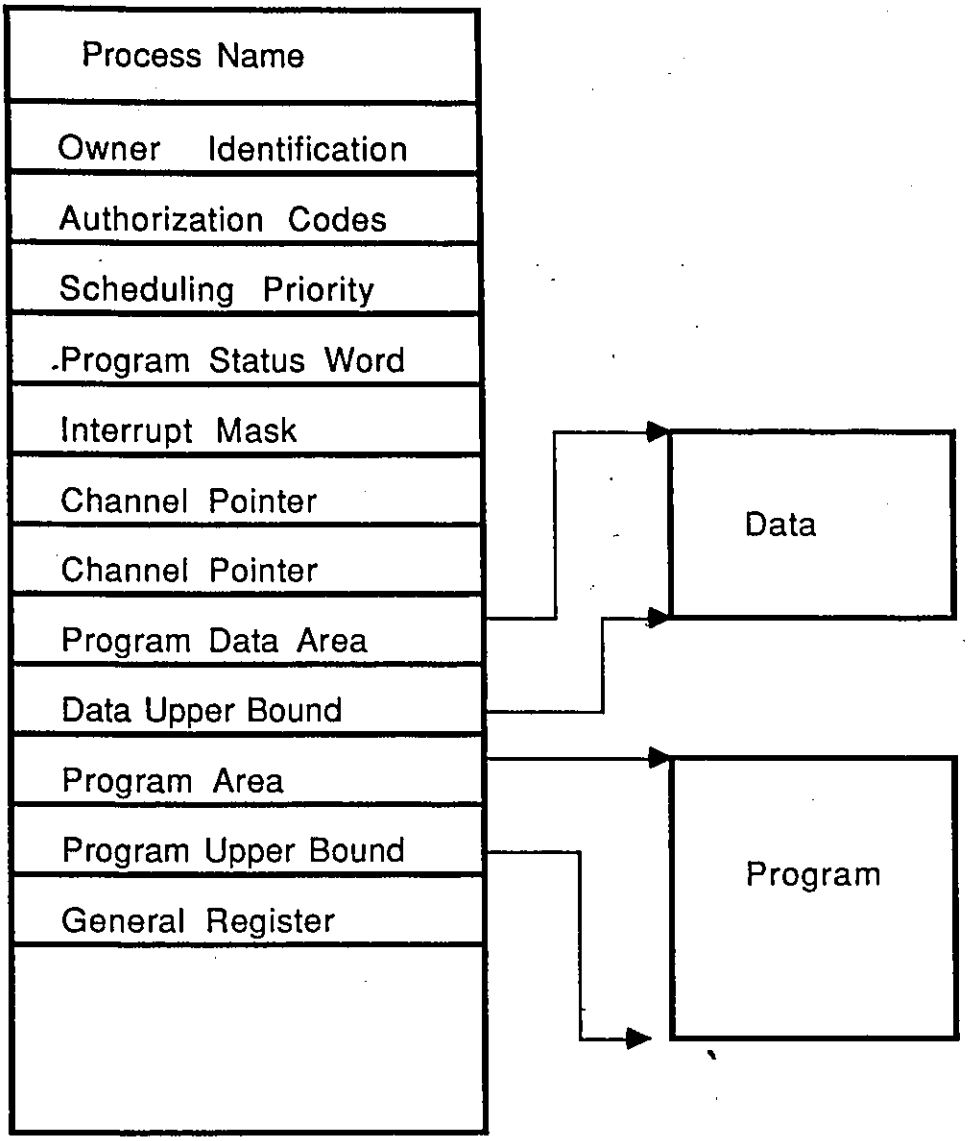


Figure 3.3: Process Control Block.

PROCESS CREATION

UNIX and XENIX provide facilities for dynamic creatation processes using simple function calls. These processes communicate by message passing. On UNIX, the function "FORK" is used to create processes. The communication medium between the processes are created by the use of "PIPES" and "SOCKETS" [KERN84, BELL83a, BELL83b, McGM83]. UNIX is responsible for scheduling and managing the channels between processes.

PROCESS SCHEDULING

In a typical computer system, many processes compete for processor time. An efficient scheduler is required to schedule these processes so that each process gets a fair share of the CPU time and resources. The OS ensures that processes executing the same *critical region (or section)*, are mutually and exclusively scheduled to enter and execute the critical region.

CRITICAL REGION (or SECTION)

A *Critical Section* of a program is a set of instructions in which the result of execution may vary unpredictably if some of the variables referenced in the section, are accessible to some other processes. To prevent inconsistency of the values assigned to these variables, any process in a critical section should never be preempted by another process which has a write access to any of the variables.

Mutual exclusion of processes is usually ensured by software programmers. Languages such as ADA [HABE84] have mutual exclusion capability builtin. In some applications, the OS handles mutual exclusions. In that case, the OS must mutually exclude parallel processes with critical sections from simultaneously entering the critical section. Mutual exclusion

of processes with respect to a given critical section means that no more than one process can be in the critical section at any time.

PROCESS SYNCHRONIZATION

The OS should synchronize the activities of the processes. Synchronization of a process ensures that the process does not proceed beyond a given point without an external stimulus. Such as, an explicit singal which the process can not generate by itself.

ROBUSTNESS

Robustness must be considered when selecting an OS. Two processes are deadlocked if neither can continue until the other continues. To eliminate race conditions and deadlocks, any process stopped outside a critical section should not impede the progress of other independent processes. Moreover, processes should not wait indefinitly for either the OS resources or signals. A system deadlock occurs if all processes in the system are deadlocked. Some techniques which could be used to avoid deadlocks include Preemption of processes which are waiting for resources or signals, and preemption of those processes that have exceeded their share of the resources.

Module Implementation Techniques:

The processes implementing protocol modules can be created as one or a combination of the following :

1. *A Procedure per Module:* In this approach, a procedure is generated per protocol module in the specification. In order to differentiate such procedures from the ordinary procedures or subprograms, I will refer to them as "Protocol Procedures". As a result of these procedures, a private scheduler, referred henceforth as a **Protocol scheduler,** is required to determine when any one of these protocol

procedures should be invoked. The collection of the protocol procedures and the private scheduler constitutes a **Protocol Machine (or System)** . Execution control normally starts and ends with the scheduler. Execution control is transferred to a protocol procedure and is returned to the scheduler once the protocol procedure is exited. The criteria for passing execution control from the scheduler to a protocol procedure are:

- the arrival of a message destined for the protocol module (implemented by the procedure), or
- the direct invocation from a protocol procedure in the case of rendez-vous channels.

In this approach, instances of the modules are stored as global variables which may be accessible to any module in the protocol system.

The disadvantage of this approach are:

- more complexity is introduced by the inclusion of a private scheduler.
- the privacy or independency of the modules may be compromised by making instances or context variables accessible to other modules.
- a higher probability of incorrect implementation of the underlying concept of the formal protocol specification.
- the portability of the prototype may be compromised depending on the design of the scheduler. This is the case when the scheduler has been fine-tuned to function optimally in a specific environment.
- the efficiency of such a scheduler is questionable when compared to the scheduler used by the host system.

2. *A process per Module:*

The alternative is to implement each module as a process. The processes communicate with each other using the available inter-process communication

facilities. For maximum utilization of the OS resources, the processes are created dynamically by the host system. Each process denoting a protocol module is used to implement many instances of protocol connections. Each connection instance is identified by the Connection_End_Point_identifier (CEP_id). The CEP_id is used to identify the instances of the state variables associated with the protocol connection.

3. *A Process per Connection:*

Another approach is to create a new process for each protocol connection instance. The pros of this approach is the ease of implementation. In reality, there is a limit to the number of processes that can simultaneously exist in any host system. Creating a process for each protocol connection may quickly exhaust the resources allocated for process creations. Worse, this may degrade the performance of the host system.

3.4.2.2 Inter-Process Communication

Essential to any multi-process Computer System is the mechanism to enable communication between processes. The inter-process facilities of the operating system are needed for :

- the creation of the communication channels between the protocol modules (or modules for short).
- the implementation of the Abstract Service Primitives (ASPs) associated with channels defined in the specification. The ASP implementations provide access to the host system messaging facilities.
- establishing the connection between the modules, and the connection between each module and the host system.

Inter-Process Communication (IPC) is carried out as an exchange of messages through a commonly accessible data base in accordance with some pre-arranged conventions among the communicating processes. The IPC consists of the following fundamental points:

- the IPC-setup. IPC-setup is the initial agreement among the communicating processes. Established externally to the universe, it specifies the shared data base and the conventions for accessing the data base.
- the basic mechanism. The basic mechanism is capable only of transmitting a single one-way inter-process message.

A useful general purpose IPC mechanism can be created by the expansion of the IPC fundamentals to the desirable degree. For example, instead of one-way channels that can only hold a single inter-process message, have one-way channels that support more than one single inter-process message. Also, it may be required that these channels ensure orderly delivery of the messages to the processes.

3.4.2.3 Timer Services

Most communication protocols are only useful when used with a timer service. The OS must provide timing services. For instance, most protocols are able to detect various error conditions from Timeout messages. Timeout messages are used to detect loss of messages, failure of a node to respond after being primed, the exact time to perform an event, and etc. For protocols that require timer services, a communication channel is defined between the OS and the protocol module. Table 2.3 shows a sample specification of the channel between a Transport protocol module and a timer module. The transport module will set its timers using the "TS_start_timer" primitive. The expiration of the timer is communicated to the module using the "TS_timeout" primitive. The module can cancel its timer request by using the "TS_stop_timer" primitive. A more sophisticated Timer process will allow multiple timers to be set by the same module.

In our experimentation with one of the classes of the OSI Transport Layer Protocol [BOCH82a, LOGR84, TEO84], namely, the Class 0 Transport Protocol, a limited use of the host system (i.e. UNIX 4.2) timer services was employed. The use of the UNIX timer facility was limited to the scheduling of the protocol processes and the polling of the

SOCKETs (for the arrival of messages). Also, the OS used the channels to communicate to each protocol process, the status of its OS resources.

3.5 SEMI-AUTOMATED PROTOTYPING

Semi-automated prototyping [ANSA83, AGGA84, GERB83, NASH83, BOCH84] is the process by which executable prototypes of formal protocol specifications are automatically generated either by the use of a compiler, a preprocessor or an interpreter.

The preprocessor or interpreter takes as input, a formal protocol specification, and produces an executable prototype or code. Due to the cost (both time and labour) of developing compilers, coupled with the fact that formal specification languages are always undergoing reviews and modifications, it may not be wise to develop a new compiler for such a language. As a result, many software engineers favour the use of a preprocessor which translates formal protocol specifications to an intermediate code. The intermediate code is usually a high-level programming language. From Figure 3.4, this intermediate language would already have a compiler to produce machine code from the produced source.

For the automated prototyping process to be flexible and portable, macro libraries are used. It is also necessary to provide a library of commonly requested functions and procedures. The library would contain customized input/output functions. The macro interfaces should be clear and well defined. The body of the macros are implementation dependent. It is the responsiblity of the compiler or preprocessor to include the body of each macro during the automated prototyping process. Also, the compiler provides some relevant implementation language details. In addition, the details of the services provided by the operating system are included from a library.

During compilation, some of the constructs of the formal specification which are similar to those of the target language are copied with few or no modifications. Others

may be expanded or discarded depending on their applicability to the target environment. Some specification constructs used for clarity, may be extremely difficult to implement in the target language. These constructs may be modified bearing in mind that such modifications should not affect the behaviour of the protocol module.

In the next few sections, the detailed aspects of the actual semi-automated prototyping process is presented.

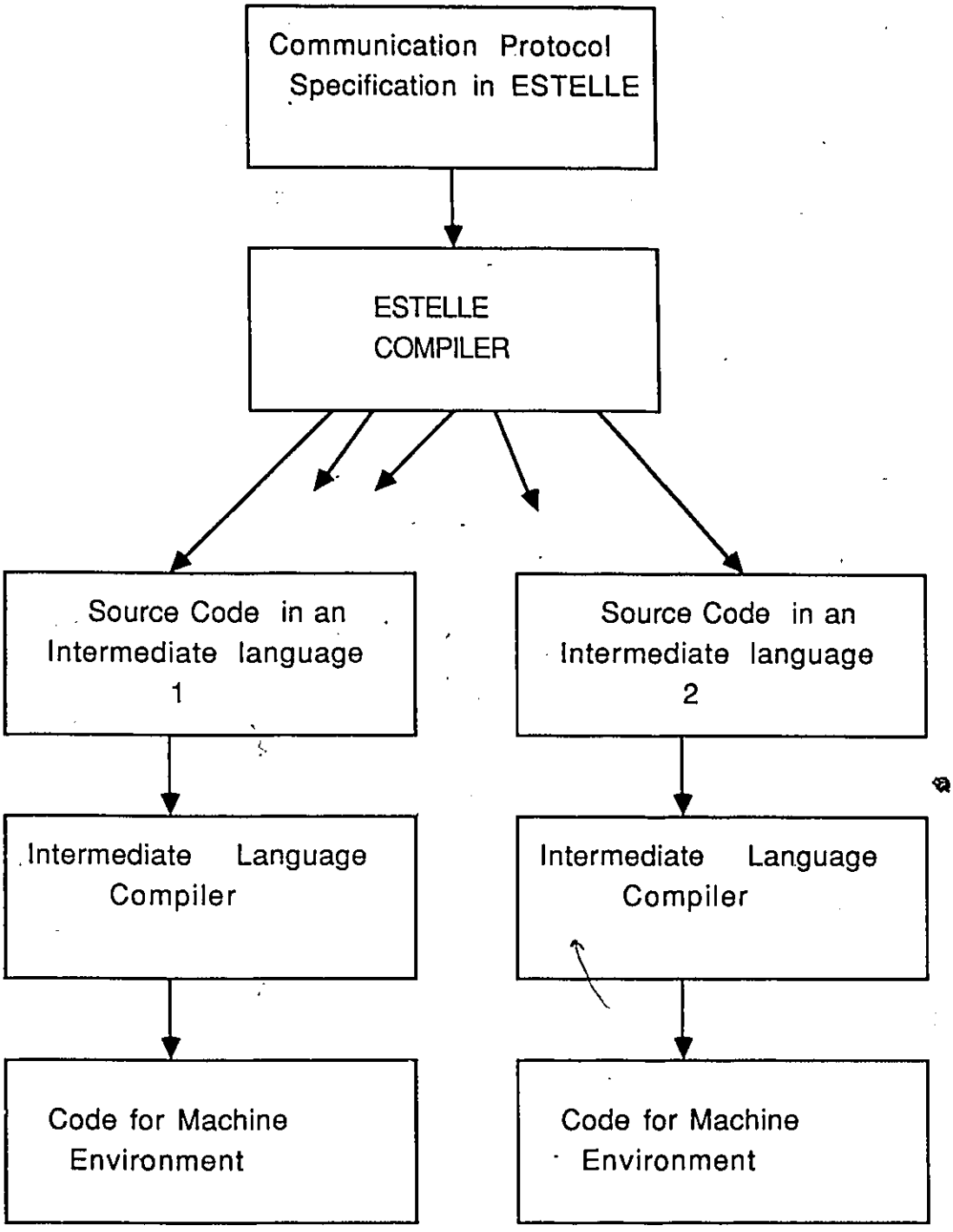


Figure 3.4: ESTELLE Intermediate Languages.

3.5.1 Subsetting

Subsetting is the ability to perform conditional compilation; to only include those macros and functions which are needed for a successful compilation of a generated prototype. Considering the fact that the macro and other utility libraries are available for compiling all protocols relevant to the seven layers of the OSI Reference Model, it is necessary that the formal specification language compiler (or preprocessor) have the capability to perform subsetting.

Our ESTELLE preprocessor and C language compiler have subsetting capability which further, enhances the qualities of ESTELLE as a specification language for semi-automated prototyping of communication protocols.

3.5.2 Implementation Conformance Evaluation

Every protocol implementation should be subjected to a conformance evaluation [BILL85, RAZO80, TENG79, WEST78]. The behaviour of the implementation must be consistent with that of the specification model. Automated test generation [PROB83, URAL83a, URAL83b, URAL86,] may be carried out to assess the performance of the implementation. The factors which are considered during performance analysis are efficiency of operation, maximum throughput of the system, delay encountered during high traffic and the memory requirement of such implementation.

3.5.3 Performance

Although the programming ease and correctness of automated prototyping should not be traded for space and/or time efficiency, every protocol implementation should include a performance specification that must be satisfied in order to be acceptable. Memory space and execution speed can not always be traded away. Fast memory is still quite expensive compared to machine instruction cycles, since some machine architectures are not

expandable to provide more memory. Further, any automated prototyping process that results in significant degradation of a system is not acceptable. This is why we have used existing utilities for our implementation. Consequently, space and time efficient codes are necessary to avoid the aforementioned pit falls.

3.5.4 Portability

With reference to "Portability of Source Code" on page 53, portability of protocol implementations is necessary in many industrial applications in order to justify the initial cost of a semi-automated prototyping process.

For instance, when considering a complete family of products based on the concepts of the OSI Reference Model [DESJ81, ISO81, ISO82a, TANE81, ZIMM80], many functions such as connection establishment, appear in all the products. Other functions such as: quality of service, data fragmentation and data segmentation, though not universal, appear in several classes of the products. It makes sense to implement these functions once and re-use the implementation for several other products.

Since such functions are specified using formal description languages like ESTELLE, the problem is reduced to that of portability. Portability is based on two requirements, language portability and execution environment portability.

The design choices for language portability are indicated in Figure 3.5. The first option is the generation of an intermediate text for the ESTELLE specification. The intermediate text can then be interpreted or compiled. Normally, interpretation will take more time and may not be acceptable.

The second option is to compile the text directly into machine specific codes. Unfortunately, experimentation into multi-target compilers seems to have its success only when restricted to machines with generally similar architectures and instruction sets.

The other alternative is to translate the ESTELLE specification into a programming language which already has a compiler in existence. Ideally, a single language is generated

for all the machine specific compilers. If that is not the case, then multi-language translators are required to produce specific languages for specific machines. The compiler on these machines can then be used to compile the generated programs.

Portability allows separate distinct developments of the various layers of the OSI reference model. These distinct implementations can then be integrated to form a communication system.

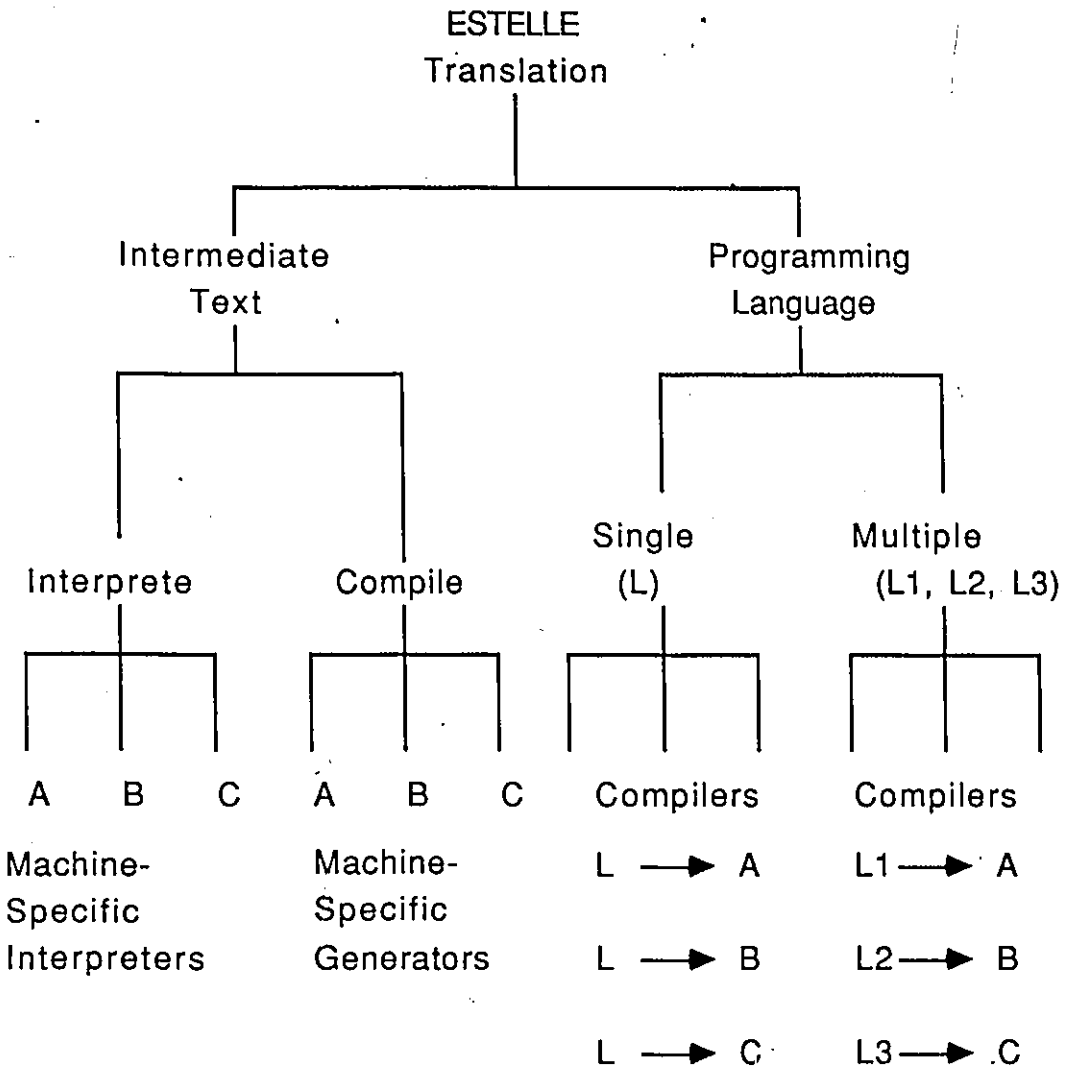


Figure 3.5: Language Portability: Appropriate translation steps for generating protocol implementations from ESTELLE specifications.

Chapter IV

THE DESIGN STRATEGY

This chapter discusses the configuration and the target Run-Time environment of prototypes of communication protocols developed by our automated prototyping process, as well as our design strategy. Our design approach is based on the OSI Reference Model discussed in "THE OSI REFERENCE MODEL" on page 14.

The typical *Run-Time system* is depicted as shown in Figure 4.1. The system consist of the Hardware, the Operating System, and the system Users (or Users for short).

Part of the hardware is used to physically interconnect many of these run-time systems. The Operating System is responsible for the orderly management of all the resources available in the system. One of those resources is access to the communication links connecting the system to other run-time systems. The collection of interconnected systems forms a network of communication systems.

The set of system users consists of a set of subsystems and the set of users of these subsystems. From Figure 4.2, the set of subsystems would include a Data Base subsystems, a Transaction subsystems, a Communication subsystems, etc.. In this thesis, we are interested in the design and implementation of a typical communication subsystem.

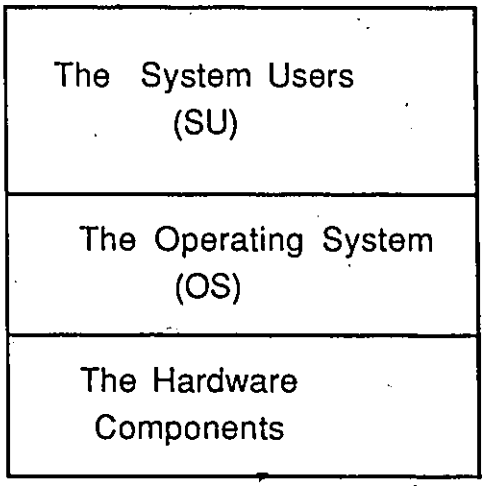


Figure 4.1: The Run-Time System: Comprises the System Users, the OS and Hardware.

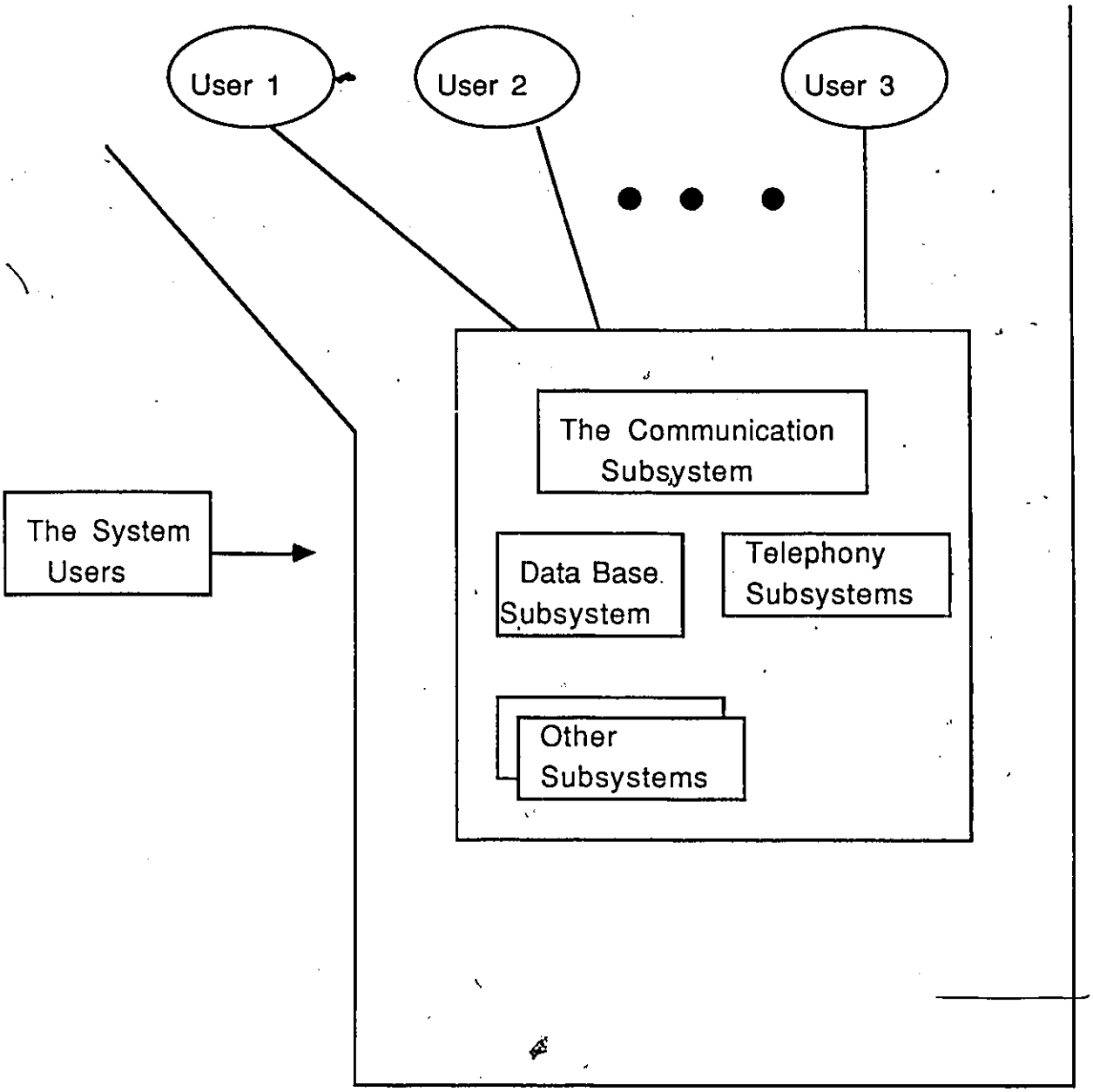


Figure 4.2: The System Users: Include such subsystems as: Data Base, Telephony, etc.

4.1 ASSUMPTIONS

To successfully complete this work on time, we made some reasonable assumptions, and at the same time, restricted ourselves to a subset of of the specification language. This subset is the minimum requirement for complete specification of a complex protocol such as the OSI Class 0 Transport Protocol. Here are the major assumptions we made:

- Single Module Specification:

  Our ESTELLE preprocessor require that no substructuring of module specifications are allowed. (See "PROTOCOL MODULES" on page 31 for detail description of substructured modules.) All protocol entity specifications must be in single module specifications using ESTELLE.

- Separately Generated Prototypes:

  Each module specification must be compiled (or preprocessed) individually by our ESTELLE Preprocessor. A source file containing N number of module specifications will be processed by the ESTELLE compiler into N prototypes corresponding to each of the modules.

- A Process per Module:

  Each protocol module is implemented as a process in the target environment.

- Loosely Coupled Modules:

  All modules are loosely coupled, and are not allowed to have common variables. This is justified by the fact that each prototype is executed as an independent process in the environment. The processes communicate with each other via messaging.

- System Configuration:

  We use an external process called, **The Communication Root Process (CRP):**

  to create each of the required processes.

  to establish the communication channels between the processes.

  to monitor the run-time activity of each process.

- Module Specification Structure:

  All module specifications must conform to the specification format specified by the International Standard Organization (ISO) [ISO85, ISO84].

- Protocol Errors:

  Incomplete specifications are serious errors, but cannot be treated fully in this thesis. The correctness of the formal specification is the responsibility of the protocol designer. We have plans to incorporate the ability to detect some protocol errors, such as deadlock, livelock, unspecified reception, and unexecutable transitions.

  However, some basic errors are currently detected by the ESTELLE preprocessor when generating the source for the executable prototype. These errors are compile errors related to the syntax and semantics of ESTELLE.

  More compile errors are detected by the C language compiler when generating the machine code from the generated source code for the specification prototype. Incompatibility between data types are also detected.

- Executable Prototype:

  Each generated source code from a given module specification must be compiled, linked and loaded separately.

- Manual Implementation Requirements:

  The following parts of our system were implemented manually:

  1. For speed of execution, the CRP is manually implemented so the required number of communicating processes are created and connected to each other.
  2. Entry (or Main) Code:

     The **Entry Code** is where execution begins when a program is loaded into memory.

     The entry code associated with each protocol process (or formal protocol specification) is manually implemented. In cases where the

modules are configured uniformly, the same entry code can be used in each of the processes.

Our ESTELLE preprocessor automatically includes a simple generic entry code. These entry code can be substituted for a designer-provided entry code. Detail description of the generic entry code is provided later in this chapter.

4.2 *TARGET RUN-TIME ENVIRONMENT*

Our design approach is based on the *Target Run-Time Environment* presented below. Note, our methodology (presented in "AUTOMATED GENERATION OF EXECUTABLE PROTOTYPES FROM ESTELLE" on page 97) is still valid for other possible target run-time environments.

Our Run-Time Environment is illustrated as shown in Figure 4.3. As was mentioned earlier, given a formal specification of a layer N protocol entity, we produce a source code in C programming language for the prototype which runs as a process in the target environment. (For clarity, we will at times call these generated processes, *Protocol Processes* .)

The run-time environment of each (N)-layer protocol process contain the following:

1. the Operating System,
2. the Communication Root Process (CRP),
3. the (N-1)-layer (or User) Protocol Process, and
4. the (N+1)-layer (or Provider) Protocol Process.

In addition to providing the facilities for:

- Process Management including process creation, process scheduling, etc.,
- Inter-Process Communications (IPC),
- Timer Services,

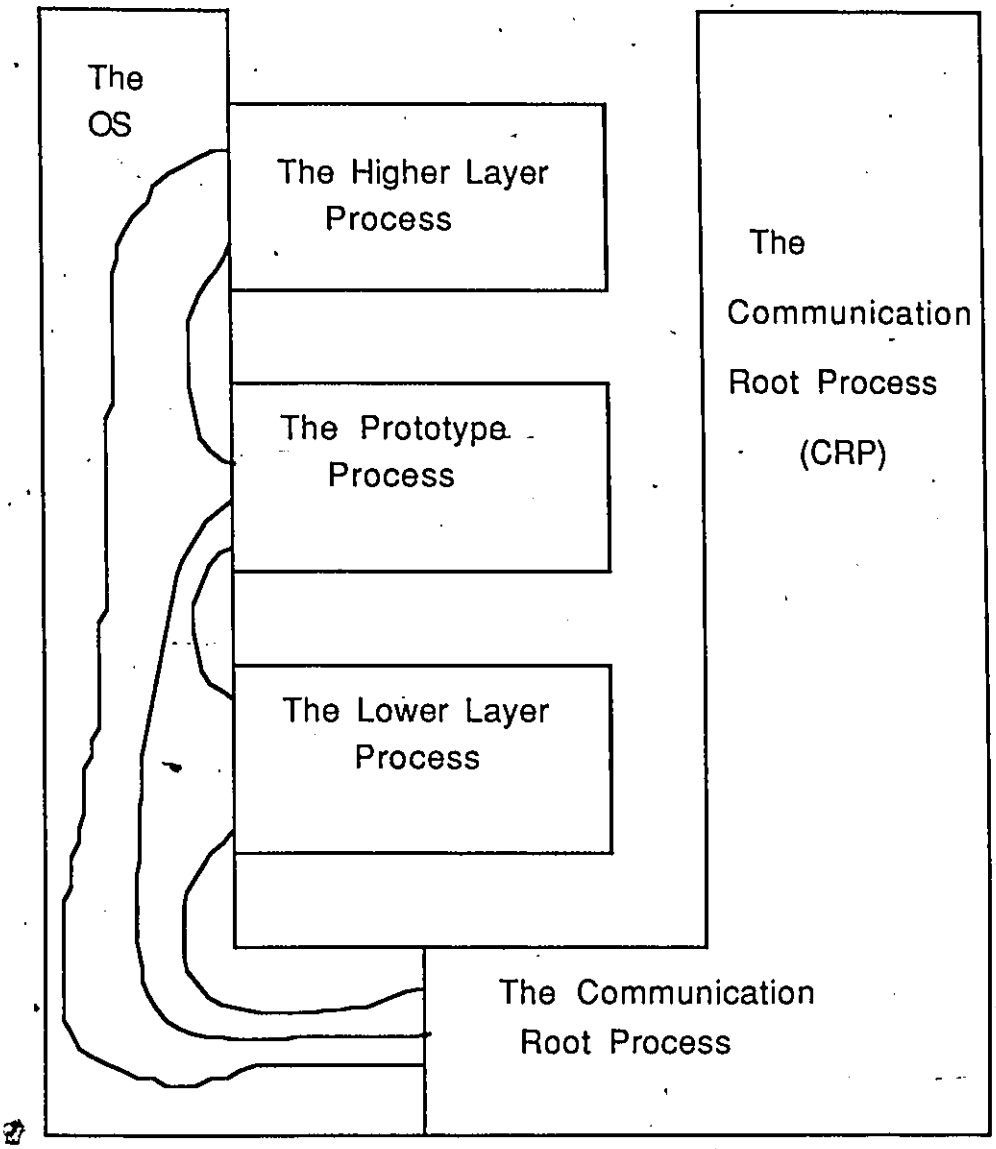


Figure 4.3: The Run-Time Environment of the Prototypes: consists of the OS, the CRP, the Higher and Lower layer processes.

the Operating System controls the hardware to provide the needed communication link with other communication systems. The Operating System is responsible for the creation of the CRP upon request. It is responsible also for the creation of all the protocol processes when requested by the CRP.

The Operating System is always called upon by the CRP to allocate communication channels used by the protocol processes in their interaction with one another, and their respectively restricted interaction with the CRP.

Always associated with each communication subsystem (or system) is a *Communication Root Process (CRP).* The CRP is an extension of the Operating System. It serves to request the creation of the various protocol processes which are needed for the configuration of the typical communication subsystem. It is also intended to configure the communication subsystem from the list of protocol processes it requested to be created. Each of these protocol processes provide a functionality associated with an (N)-Layer module of the OSI Reference Model (See "THE OSI REFERENCE MODEL" on page 14 for details on the reference model.) More details about the CRP is provided below in a later section.

The (N-1)-layer Protocol Process (if present) is the User process while the (N+1)-layer Protocol Process (if present) is the Provider protocol process. The environment associated with each of these protocol processes is the same as that of the (N)-layer protocol process being discussed.

In practice however, functions of an (N)-Layer entity can be provided by one or more processes. We refer to this as a multi-process configuration. The processes in the multi-process configuration cooperate in order to provide one interface to their environment. However, only one of these processes, *(the Parent Process)* is allowed to communicate directly with the environment. The process with this capability corresponds to the parent

module in the formal protocol specification, and its channels must be implemented as such if it is to conform to the ISO specification model. The other processes *(i.e. child or Internal Processes)* correspond to the submodules associated with the formal specification. This is how substructured modules (or module specifications with submodules) will be implemented in our approach (or methodology).

4.3 DESIGN APPROACH

For the purpose of our discussion, we will deliberate on a single process per protocol module. As a result, in the implementation a communication system, each protocol module, such as the Transport module, is implemented by a process as shown in Figure 4.4. From the OSI reference model, the communication subsystem will consist of seven processes corresponding to the seven layers of the model.

In practice, we could have implemented each of these protocol processes as a collection of one or more processes, but we did not.

Had such been the case, one of these processes would be responsible for removing interactions from the sockets, and enqueuing the interactions on the appropriate Connection_End_Point (CEP) queues. Another process would be assigned the task of writing the interactions to the sockets from the various outgoing interaction queues. However, the more processes employed implies a high consumption of host system resources.

An exception was made in the case of multi-connection establishment. In some instances (e.g. when all available Connection_End_Point_Identifiers are in use), we see a justification for spawning another protocol process instead of rejecting the request for connection establishment. The spawned process is merely another running instance of the same protocol process under consideration, and should not be confused with protocol processes for submodules.

However, the complexity introduced by the ability to spawn copies of a protocol process is high, and include such issues as channel access resolution. Where an arbitrator is required to resolve requests to the communication channels

since both processes share the same communication channels but not the same state variables.

So for the OSI reference model, each Layer N process communicates directly with the adjacent layer N-1 and layer N+1 processes via communication channels created by the CRP. These channels are bi-directional in that they allow the flow of data in both directions. They are also implemented as First-In-First-Out (FIFO) queues. In our target Host system, UNIX 4.2 running on VAX 750, each communication channel is composed of *a pair of SOCKETS*, or *a pair of PIPES*. (See details below.)

In this chapter and subsequent chapters, we will illustrate the design approach with the implementation of the Class 0 transport protocol specification in ESTELLE.

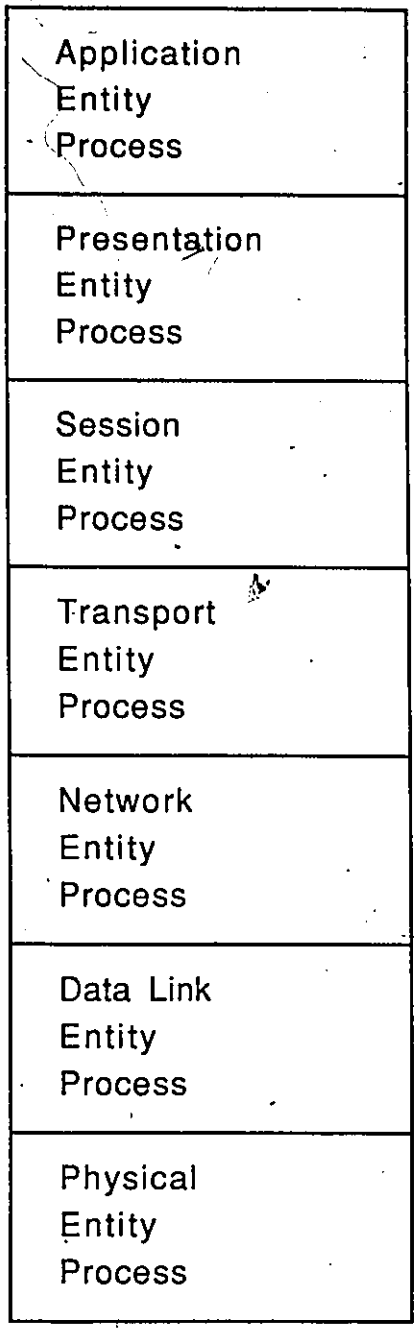


Figure 4.4: The Communication Subsystem: A communication based on the OSI Reference Model consists of the above processes.

4.3.1 The Communication Root Process

As was previously mentioned, the communication Root Process (CRP) (see Figure 4.3 and Figure 4.5) is an extension (or agent) of the Operating System, and is created by the Operating System.

The responsibilities of the CRP is best presented by a discussion of its usage in the design approach to semi-automated prototyping. The CRP creates all protocol processes used in the communication system, and their interconnecting channels. The CRP is used for the following:

- it allows the monitoring of the communication subsystem. During the execution of a communication system, the CRP receives messages from the host system informing it of the death of any of the protocol processes created by the CRP. This type of message we called *Death Message*.

  On the reception of a death message, the CRP could perform any one of (or a combination) of the following:

  deallocation of system resources allocated to the dead process.

  re-creation of the dead process.

  output of appropriate log (or software error) report indicating the possible cause(s) of death.

  in extreme cases, a deallocation of the entire communication system.

- it prevents the host system from sudden death. In many environments such as those provided by UNIX, when a process dies, all its child processes are automatically deallocated or killed. The resources allocated to the process and its offsprings are reclaimed by the Operating System.

Thus, by using the CRP instead of the Operating System, we reduce the chances of the Operating System dying due to a software trap (or exception). Moreover, if a

CRP dies, the OS can always spawn a new CRP. But if an OS dies, the entire system must undergo a system restart. In some cases, a system re-boot may be needed.

- it ensures the integrity and adds to the robustness of the entire system.

The CRP is invoked by the OS to configure the communication system as a result of:

1. a request from a user in the system for a protocol connection.
2. an Initial Program Load action (or restart).
3. some other reasons such as software exceptions, hardware exceptions occuring in one or more processes associated with the communication system.

At system initialization, the CRP will create all the inter-process communication channels before creating any of the protocol processes. Each created process implements a specific protocol module. The processes communicate via channels by means of "Message Passing". A channel consists of a "SOCKET-PAIR" [KERN84, BELL83a, BELL83b, McGM83]. A socket-pair is a collection of any two *Uni-directional SOCKETs*. A SOCKET is a uni-directional data flow facility that is available on UNIX 4.2. It allows data to be written (SEND) at one end and read (RECEIVE) from the other end. (See Figure 4.6 for details.)

In systems that do not currently support the use of SOCKETs, the channels can be implemented with the use of PIPES [KERN84]. PIPEs are very similar to SOCKETs. The major difference between PIPEs and SOCKETs is that SOCKETs

provide options for non-blocking of processes while PIPEs do not. A process is blocked (or temporarily suspended) when it attempts to read from an empty PIPE. The read operation on an empty PIPE will block the process that initiated the request until the pipe becomes non-empty. This is very dangerous and leads to *deadlock*. Two processes are when neither can continue execution until the other does. One means of avoiding such deadlocks is to poll any PIPE before requesting a read operation.

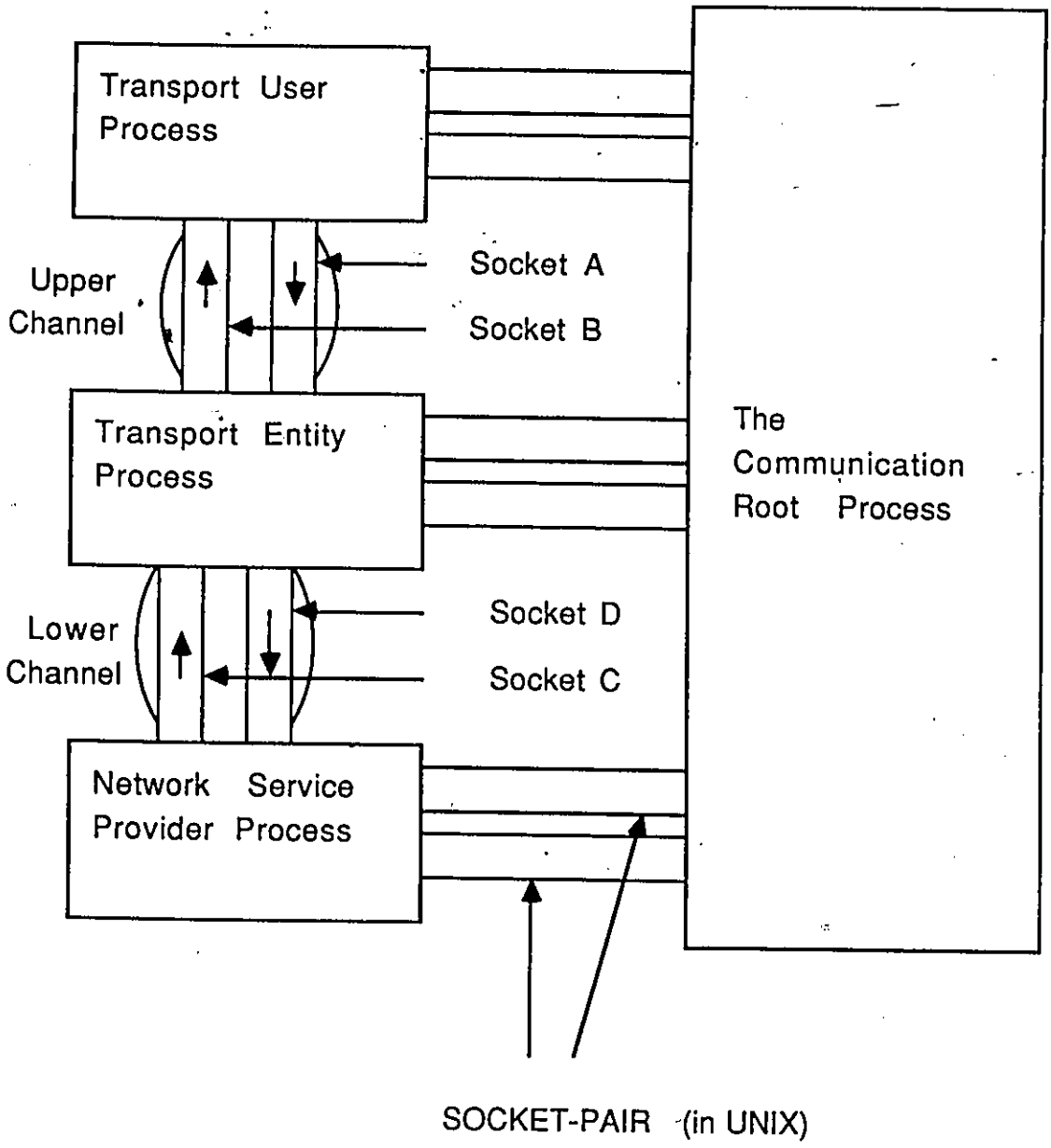


Figure 4.5: The Design and Implementation of a Transport Prototype.

To avoid conflicts over the channels, each protocol process has a capability assign to it for accessible socket. The capability is either for read or write. The read capability allows the protocol process read only access to the socket, while the write capability allows for a write only access.

For example, in our implementation of the Class 0 Transport Prototype, the CRP restricted the Session protocol process to only READ access to SOCKET A and WRITE access to SOCKET B. At the same time, the Transport process is allowed a WRITE access to SOCKET A and READ access to SOCKET B.

This is to prevent:

1. a process from reading its own message.
2. two communicating processes from trying to SEND messages at the same time to the same SOCKET.
3. two communicating processes from trying to RECEIVE messages at the same time from the same SOCKET.
4. deadlock.

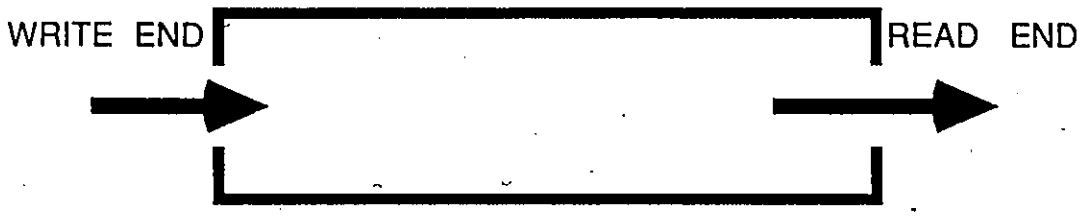


Figure 4.6: An example of a SOCKET: The above figure illustrates the read and write access allowed by a socket.

4.3.2 The Inter-Process Communication

In Figure 4.5 on page 90, the Transport Process receives all session layer messages through socket A of its UPPER channel and sends messages to the Session layer process through socket B of the channel. It, also receives messages from the Network Service Provider (NSP) through socket C, and sends its messages to the NSP through socket D of the LOWER channel.

Sockets A and C only recognize PROVIDER initiated Service Primitives while sockets B and D allow only USER initiated Service Primitives. Unrecognizeable Service Primitives are rejected and a LOG report is produced with a proper message stating the cause of the error.

Currently, all multiple connections are implemented with one channel. Each Service Primitive contains a Connection_End_Point_identifier (CEP_id). The CEP_id is used to inform the protocol process of the connection instance for which the Service Primitive is applicable.

The maximum number of multiple connections is specified in the formal protocol specification. If not specified, defaults to 1.

4.3.3 The Communication Processes

The generated C programming language source code for each protocol process is composed of two major parts, namely:

1. the ENTRY code, and
2. the TRANSITION or FINITE STATE MACHINE part

The detail description of the strategies employed in generating the C programming language source codes of each protocol module specification in ESTELLE ESTELLE is provided in the next chapter. However, we present a brief functional description of the two parts below.

4.3.3.1 The ENTRY Code

The ENTRY code (Figure 4.7) is a small program from which execution commences. For protocols based on the OSI reference model, the entry code is generic in that it performs the following functions:

1. Initialization of the protocol process.
2. Polling or scanning of the UPPER and LOWER channels of the protocol process for arrival of interactions. A simple round-robin technique is used with a pause time in between rounds.
3. When a interaction is received, it determines from the CEP_id in the Service Primitives which connection instance should be running.
4. It is responsible for initiating the execution of the Transition part with the correct CEP_id indicating the running connection instance.
5. At the end of each loop, it polls the channel connecting to the CRP for control messages. The control messages are used for graceful termination of the protocol process when needed.

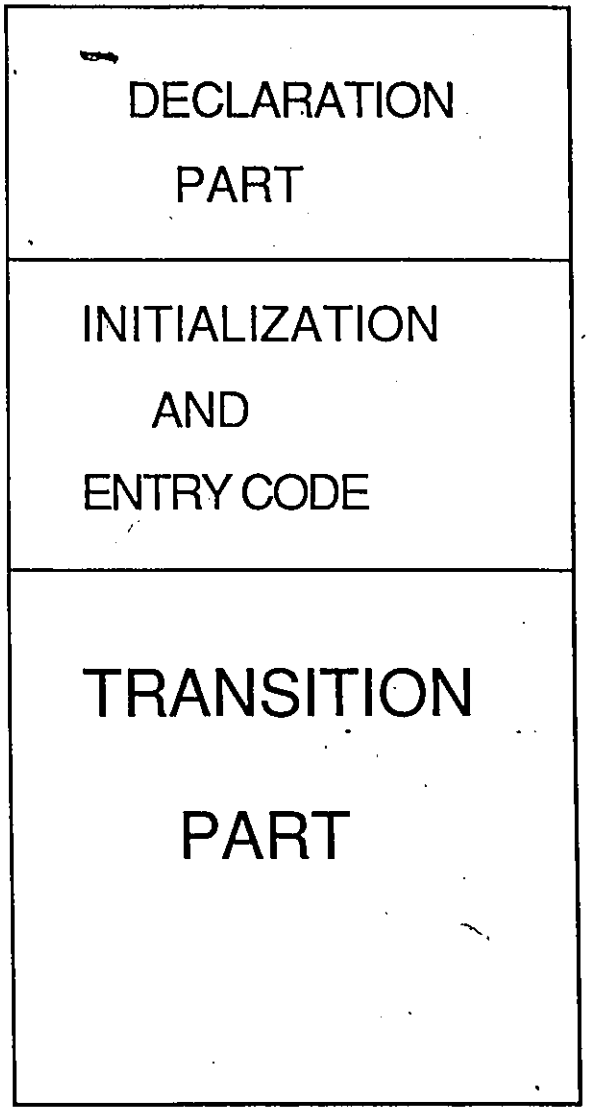


Figure 4.7: Structure of the Generated Prototype: The body of of the C language source generated for each prototype consist of the declaration part, the Initialization section, and last but not least, the Transition part.

4.3.3.2 The TRANSITION Part

The TRANSITION part is implemented as a procedure, invoked only within the ENTRY code. It accepts a CEP_id as a parameter, and manipulates the state variables associated with the supplied CEP_id.

Included in the state variables is a special variable called, *the interaction variable*. The interaction variable contains the received interaction. This variable is used among other variables to determine which transition to execute. That is, it identifies the the CASE statement label to jump to.

As was shown in "Transitions" on page 56, and illustrated in Table 3.1 on page 57 and Table 3.2 on page 58 the body of the transition part is implemented for speed and readability with CASE and IF-THEN-ELSE statements.

In addition, see Appendix H for an example of the Class-0 Transport prototype using the extended C language constructs.

Chapter V

AUTOMATED GENERATION OF EXECUTABLE PROTOTYPES FROM ESTELLE

As a part of the design strategy, this chapter presents the methodology for semi-automated generation of the executable prototypes from their formal specification. The methodology consists of the four steps depicted in Figure 5.1. In step one, the Abstract Service Primitives (ASPs) are developed. In step two, the ESTELLE Compiler is produced. In step three, the C programming language source code for the prototype was generated using the ESTELLE compiler. In step four, the generated source code for the prototype is tested and assessed for conformance to the given formal protocol specification.

Steps one and two of the automated prototyping process are performed once (except if there is a change in the syntax or semantics of the specification language). However, the identification of the ASPs must be done for all protocols concerned. Although, incremental approach (where ASPs are included as needed) is an option. Similarly, the macros, subprogram routines and the ESTELLE Compiler are produced once for all prototyping of communication protocols.

A point of emphasis, step three must be repeated for each formal specification of a Layer N protocol.

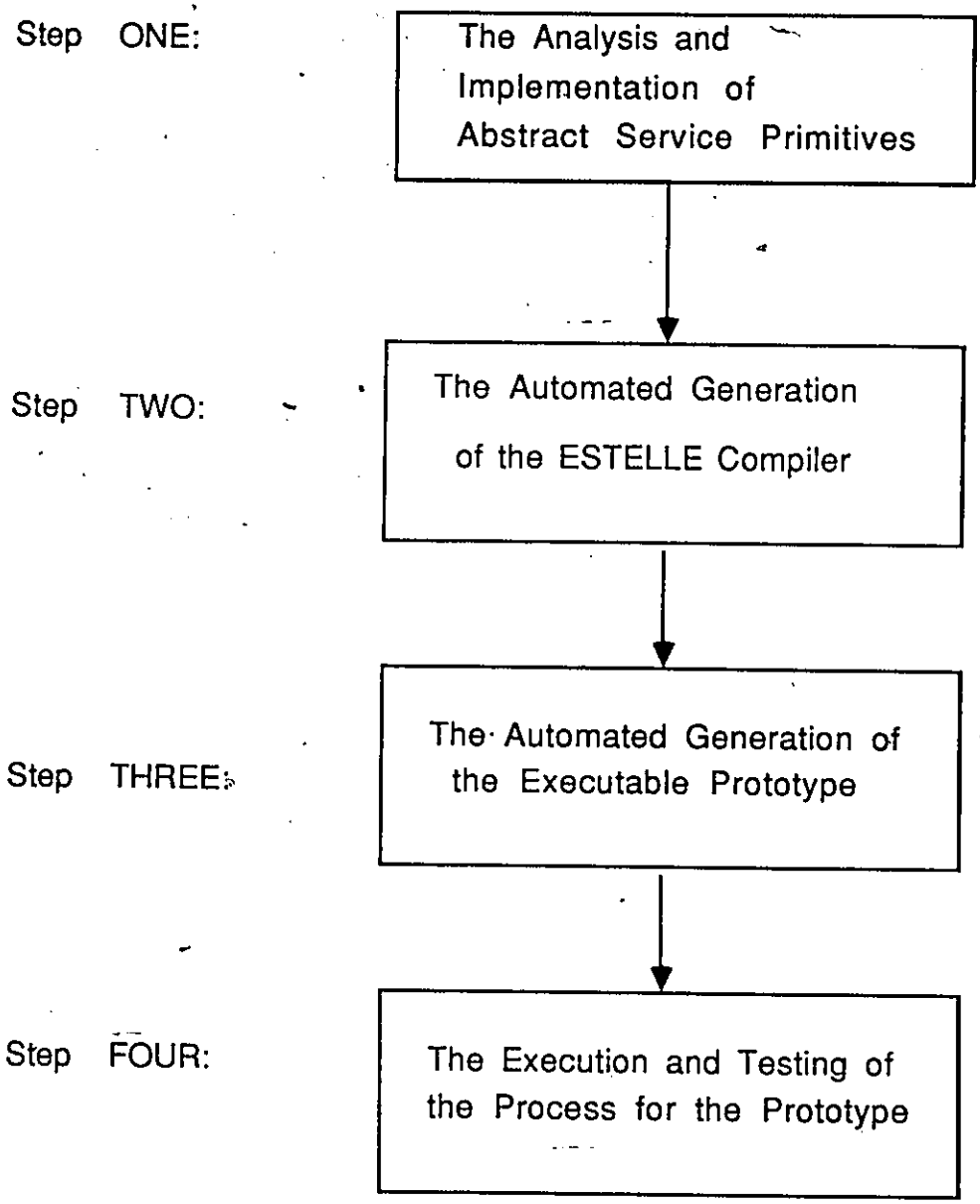


Figure 5.1: The Four steps of the Automated Prototyping Process.

5.1 *STEP ONE: MANUAL IMPLEMENTATION OF THE ASPs*

Step one includes the identification and implementation of the ASPs of the formal protocol specification. A decision on whether to implement the ASPs as procedural functions or as macros, is made. In addition, the OS services needed to implement each ASP is identified. The method of requesting such OS services is also resolved. Most of these OS services are requested by simple system calls (or primitives). In the case of the transport protocol, the ASPs include:

- T-CONNECT-request,
- T-CONNECT-indication,
- T-CONNECT-confirm,
- T-CONNECT-response,
- N-CONNECT-request,
- N-CONNECT-indication,
- N-CONNECT-confirm,
- N-CONNECT-response,
- etc.

The procedural functions and macros used for the implementation of the ASPs are manually written in the target language, in this case, is the C programming language. Each ASP implementation is compiled using the C language compiler. The produced machine code is tested to ascertain its correctness. During the testing of the ASPs, some routines are repeatedly modified in order to minimize the real-time requirement. The result of the ASP implementations is to produce clear, precise and well-defined interfaces between the modules and their environment.

In addition to developing the ASPs, step one deals with the implementation of some of the ESTELLE language constructs used in the formal protocol specification. (See Table 5.1 below for a partial listing of some ESTELLE constructs and their actual meaning.)

These constructs are implemented using the C programming language preprocessor macro utility. The implementation of these constructs are added as extensions to the C programming language. Given the formal specification of a protocol module, the ESTELLE compiler uses these ESTELLE constructs (or C Language extensions) to produce C programming language source code for the prototype of the protocol. As a result of the C extensions, the produced source code resembles the given specification. The resemblance between the generated source code and the formal specification makes the C source code legible, easy to verify and debugged. For clarity, we refer to the generated source code as being a C' source code. The C' indicating that is an extended C language source code.

We also added the keywords: RETURN and EXIT, for returning values from a function routine and leaving a statement block respectively.. A statement block is a compound statement which begins with a "BEGIN" and and terminates with an "END" statement.

Table 5.1: Some Extensions to the C Language Syntax

```
#define CHANNEL      typedef struct BEGIN
#define FROM         case
#define WHEN         case
#define PROVIDED     if
#define CASE         switch
#define OF
#define BEGIN        {
#define END          }
#define :=           =
#define RECORD       union {
#define TO           :=
#define PROCEDURE
#define FUNCTION
#define LABEL        define
#define CONST        define
#define TYPE         typedef
#define VAR
#define INITIALIZE   initialize()
#define (*           /*
#define *)           */
```

5.2 *STEP TWO: AUTOMATED GENERATION OF THE ESTELLE COMPILER*

Step two deals with the semi-automated production of the ESTELLE compiler. The parser for the ESTELLE compiler is generated by YACC - Yet Another Compiler Compiler [BELL83a, BELL83b, JOHN75, KERN84]. YACC is a general purpose utility used to generate parsers for compilers and text processors. (Figure 5.2 illustrates the process of producing the ESTELLE compiler.) YACC generates a procedure function called YYPARSE which is combined with other functions like the Lexical analyzer to produce the ESTELLE compiler. Given the ESTELLE grammar rules, YACC produces an ESTELLE parser. The parser calls the lexical analyzer to scan and recognize the tokens in the input source (or specification). The lexical analyzer used by the generated parser is called YYLEXX [BELL83a, BELL83b, LESM75, KERN84].

The lexical analyzer is generated by LEX. LEX is a general purpose utility used to generate lexical analyzers. LEX generates a function called YYLEXX which is combined with other functions like the YYPARSE to produce the compiler.

The outputs of YACC and LEX are combined and passed as input to the C language compiler. The C language preprocessor is responsible for expanding any macros which are used in the implementation of the ESTELLE compiler. The C preprocessor also includes the standard C library software and our set of libraries necessary for the successful compilation of the ESTELLE compiler source. The C compiler produces the executable machine code which is the ESTELLE Compiler (or preprocessor).

5.2.1 Overview of YACC

YACC can generate a parser either in the C programming language or RATFOR [BELL83a, BELL83b]. For example, the grammar rule for a statement in ESTELLE is given as shown in Table 5.2. The TOKENs (or terminals) are indicated as shown in upper case identifiers, while the non-terminals are in lower case.

Table 5.2: "STATEMENT" rule in YACC.

```
statement              : simple-statement
                       | compound statement
                       ;

simple_statement       : assignment_statement
                       | conditional_statement
                       | procedure_call
                       | function_call
                       ;

compound_statement     : BEGIN statement_part END
                       ;

statement_part         : statement
                       | statement_part SEMICOLON
                         statement
                       ;

assignment_statement   : IDENTIFIER  ASSIGN_TOKEN
                         expression
                       ;
```

The table defines a "statement" as a "simple" or "compound" statement. The compound statement is defined as a collection of one or more statements separated by semi-colons, contained inside a "BEGIN" and an "END". A YACC user can specify which target language is desired in the grammar rule with a simple YACC instruction. The default target is the C programming language. YACC provides facilities for associating meaning with the components of a grammar rule at the time of parsing (or compilation) called "Action statements". These action statements are executed by the parser whenever the corresponding production rule is matched.

The format of the input to YACC is precisely as shown in Table 5.3 below. The declaration section contains definitions of data types, variables, tokens and non-terminals. A partial listing of YACC rules are contained in Appendix B.

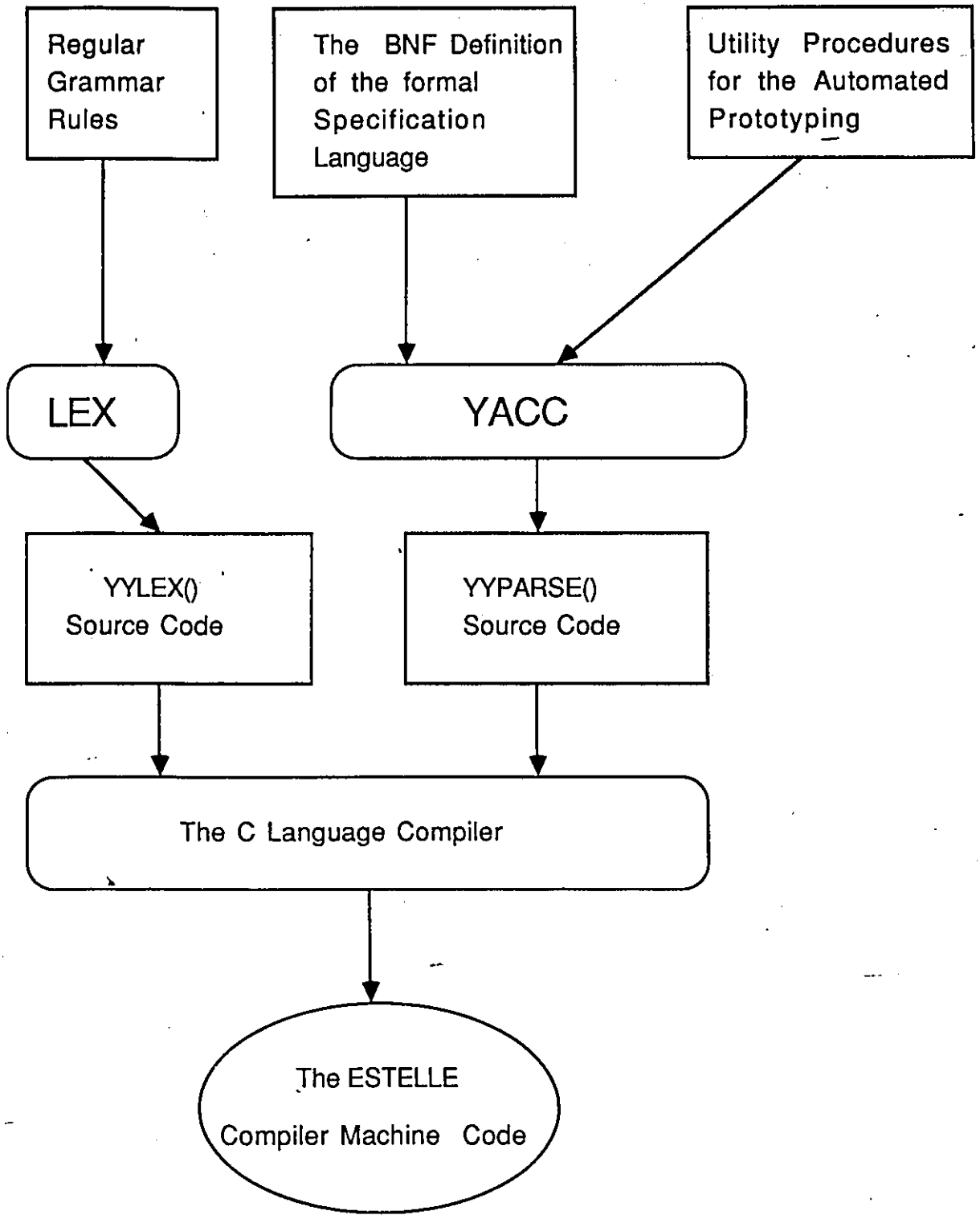


Figure 5.2: Automated Generation of the ESTELLE Compiler: First, the ESTELLE Compiler is generated by compiling the combined output of YACC and LEX (above).

The declaration section is demarcated from the Rule section by the token symbol "%%". The symbol also marks the beginning of the rule section.

The end of the rule section is also indicated by another "%%". This second appearance of the symbol also marks the beginning of the optional section. The optional section may be empty, or it may contain any C language statement including subprograms and data definitions as shown in the appendix. The subprograms are part of the compiler implementation.

Table 5.3: Structure of YACC rules.

```
Declarations

%%

YACC Rules coupled with the action parts.
%%

Programs
```

Table 5.4 provides more details on the various sections of input to YACC. The comments describe the different sections. In C programming language, comments begin with the token symbol "*" and are terminated by the symbol "*/" as shown in the table.

The format of the output produced by YACC is as shown in Table 5.5. The contents of the different sections of the input file are placed as shown in the table. Furthermore, additional parsing information are appended to the output file produced by YACC.

A partial listing of the output produced by YACC is given in Appendix C.

Table 5.4: More details on the format of the input to YACC.

The associated comment describes the functions of the various parts of the input.

```
%C          /* This specifies the target language */
            /* as the C Programming Language.     */
            /* "%R" may be used to specify        */
            /* RATFOR as the target language.     */

%{          /* This marks the beginning of the    */
            /* declaration part.  In this part,   */
            /* any C language construct may be    */
            /* used.  YACC does not evaluate the  */
            /* C constructs.  They are passed to  */
            /* the C compiler as.                 */

%}          /* Marks the end of the C language    */
            /* construct usage.                   */

            /* Zero or more YACC declarations     */
            /* may follow.                        */

%%          /* End Of Declarations indicator      */
            /* which is followed by the YACC      */
            /* rules for the Parser.              */

%%          /* Other C Language constructs may    */
            /* be added after here.               */
            /* The source code for the MAIN and   */
            /* Lexical analyzer may be added at   */
            /* this point.                        */

    MAIN ()     /* Main program */
    {
        .
        .
        .
        .
    }


    yylex()    /* the Lexical analyzer */
    {
        .
        .
        .
        .
    }
```

Table 5.5: ESTELLE Preprocessor layout.

```
Declarations

Programs

    MAIN ()    /* Main program */
    {
         .
         .
         .
         .
    }

    yylex()    /* the Lexical analyzer */
    {
         .
         .
         .
         .
    }

YYPARSE ()
{
      .
      .
      .
}
```

The main advantages of using YACC is that YACC generates small, efficient and correct parsers; though the YACC User is responsible for the correctness of the actions of the parser. Many of the tedious problems associated with writing compilers are taken care of automatically.

5.2.2 Overview of LEX

Given the regular grammar or lexical rules of any language, LEX will generate a lexical analyzer either in the C programming language or RATFOR. A LEX user can specify a target language. The default target is the C programming language. The lexical

rules contain simple regular expressions and fragments of C code as the action part. The action part is executed as in YACC, whenever the regular expression is matched or found in the input source.

The overall syntax organization for LEX is found in [BELL83a, BELL83b, LESM75, KERN84]. LEX shares similar syntactic structure as YACC. Appendix D contains the shortened input to LEX used for generating the Lexical analyzer for our ESTELLE compiler. YYLEX is the same name as the lexical analyzer invoked by YYPARSE. The format of the output of LEX is similar to that of YACC. Appendix E contains the abbreviated listing of the output from LEX. other functions that are needed by this routine can normally be found in the standard UNIX Library.

LEX is a powerful tool and provides the same advantages as YACC. Lexical analyzers generated by LEX are small, efficient and correct depending on the correctness of the action parts. Many of the tedious problems involved in recognizing strings are taken care of automatically. It also, allows for easy generation of lexical analyzers which can repeatedly be modified as the parser is changed, with little effort.

5.2.3 Developing the ESTELLE Compiler

The logical approach to developing the ESTELLE Compiler using LEX and and YACC is achieved in four steps. In the first step, the grammar rules for the language are passed as input to YACC. YACC flags all syntax errors, reports on conflicts, ambiguities, undefined terminals (or tokens) and non-terminals in the given set of rules. The generated report is used to produce precise, unambiguous and error free grammar rules.

In the second step, each production rule is augmented with an action part. The action part contains C language constructs. The C language constructs are executed whenever an instance of such a production rule is found in the source. The action part of any production rule is a block of C statements enclosed in both left and right braces [WAIT84, KERN78, FEUG84, BROW85]. The combination of the production rules and the action parts is used to describe the semantics of the language.

For the third step, the lexical analyzer is produced. The lexical rules are passed as input to LEX. LEX flags all syntax errors and reports conflicts and ambiguities. Once produced, the lexical analyzer is responsible for reading the source to be parsed and recognizing the tokens. The source is read once, a character at a time until the longest token is found. For example, "BEGIN" in Table 5.4 is a token. When the lexical analyzer recognizes the symbol "BEGIN", it informs the parser routine by returning the token BEGIN to the parser.

In the final step, a main routine is added to the combination of the lexical analyzer and the parser to form the ESTELLE Compiler. The main routine is used to invoke the parser for each source file to be compiled. The operation of the parser is to call repeatedly on the lexical analyzer for tokens, recognize the grammatical (syntactic) structure in the input, and perform the semantic actions as each production rule is recognized.

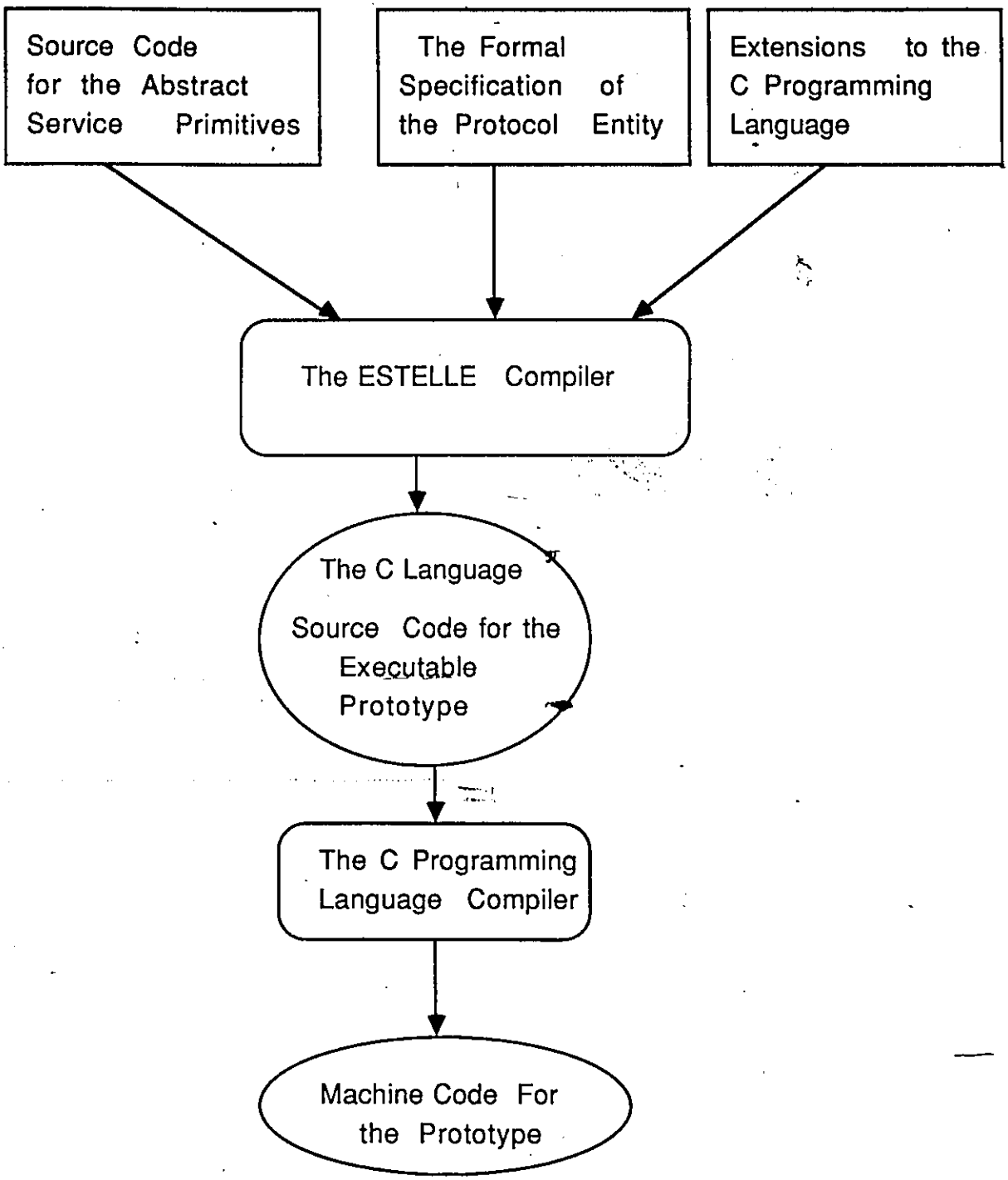


Figure 5.3: Automated Generation of the Executable Prototype: The generated ESTELLE compiler accepts a formal protocol specification in ESTELLE; producing a compilable C language source for the prototype.

5.3 STEP THREE : GENERATION OF EXECUTABLE PROTOTYPE

At this step, we are concerned with generating a C language source code for the prototype of the formal protocol specifications in ESTELLE. The compilation is achieved by the translation of the constants, data types, channel types, module interaction points and control structures into the equivalent C programming Language constructs. In the following sections, we present the details of the strategy used for the translation in our implementation.

5.3.1 Translation Consideration from ESTELLE to C

The translation of formal specifications in ESTELLE to their C programming language prototype, involves the mapping of ESTELLE constructs into the equivalent C programming language constructs. In order to understand fully the translation of ESTELLE constructs to the C programming language constructs, it is important to keep in mind the following important differences between Pascal and the C programming language. The designers of these languages have used different design philosophies [BROW85, FEUG84].

For example, input/output is considered a part of PASCAL and not considered part of the C programming language. Instead, interface primitives are provided in C by the underlying I/O system for requesting I/O operations.

DEFINITION: A language is said to be strongly typed, if:

1. every object in the language belongs to one type only.
2. type conversion in the language can only occur by converting one type to another. In other words, type conversion does not stem from viewing the data representation as a different type.

The difference between the C programming language and PASCAL is more apparent in their data types. Although C is a typed language, it is not as strongly typed [FEUG84] as PASCAL. From the above, we define representation viewing as a mechanism to subconvent the strongly typed ability of a language.

However, from the above definition, PASCAL is not a strongly typed language, since a data type can be viewed in different ways by using variant records. But it is very close. Strongly typed languages increase the clarity, correctness and reliability of programs.

In system programming, the need for strong typing and representation viewing is necessary and C provides it in abundance. A character variable can be treated as an integer variable with no problem at all in C. While this is not the case in PASCAL.

The other difference between C and PASCAL is the way implicit conversions are handled. Pascal allows implicit conversions only in cases where no loss of information is involved. For example, the conversion from subranges to the parent range and the conversion from integers to reals (although in some machines the later is not correct). PASCAL allows the use of explicit functions in conversion of other types. In the contrary, the C programming language allows conversions between any of its basic types and pointers.

Because of its strong typing characteristics, PASCAL programs generate more errors than C programs. The type handling method of PASCAL is sometimes frustrating, as in the case of arrays. Since PASCAL considers the size of the array as one of the type characteristics, a programmer can not write a sort routine for arrays of any size. This at times poses a serious constraint to programmers. The use of C language omits such constraints. Furthermore, LINT [KERN84, BELL83a, BELL83b, McGM83] a utility program which analyzes C programs have been produced to help enforce some type checking of C programs.

PASCAL, being restrictive, emphasizes program reliability and readability, whereas C with its absence of restrictions, emphasizes programmer's flexibility.

The C preprocessor [WAIT84, KERN78, FEUG84, BROW85] provides three main facilities: string substitution, text inclusion and conditional compilations. The preprocessor command lines contain "#" in the first column, and may appear anywhere in a C text

file. The string substitution is used for constant declarations including string constants. Text inclusion commands are used to include source codes from other files into the current file compilation.

Extensions to C were achieved by using the preprocessor substitution instructions. They were necessary in order to produce executable prototypes that are legible and similar, syntactically to the given protocol specification in ESTELLE.

A partial listing of the extensions to C is provided in Table 5.1 on page 101.

5.3.2 Using the ESTELLE Compiler

To produce a prototype from a protocol specification, the user would enter the following command from a keyboard:

```
PROTOTYPE  <file1, file2, ... file3>
           or
PROTOTYPE  +C control_file
```

This command invokes the ESTELLE preprocessor. "file1", "file2", etc., are the names of the files containing module specifications. Note, a module specification may span two or more files. In this case, the subsequent files are treated as containing the continuation of the module specification. Such continuation files cannot be compiled separately, and must be passed in their order of compilation.

The order of compilation is such that when the files are appended together, they form a valid ESTELLE specification.

The option "+C" tells the informs the preprocessor that the given file is a control file. A control file contains names of ESTELLE files to be compiled. One file name per line.

The default file type for all ESTELLE files is "ESTEL". Assuming the file *TRANSPORT.ESTEL* exists, a user can produce the corresponding prototype by typing:

```
PROTOTYPE transport
          or
PROTOTYPE transport.ESTEL
```

Similarly, the default file type for the control file is "ESTEL".

When multiple module specifications are contained in the input file(s), the ESTELLE preprocessor produces a separate source file for each of the modules in the specification. Files of the type, *MODULE*, are produced for each module in the specification. The produced source can be compiled and executed separately with a minor modification.

Further, a listing file of the type *LISTING* is produced for each input file in the command line.

Figure 5.3, illustrates what happens when the user invokes the compiler. Depending on the module and channel types being defined, the compiler retrieves the appropriate implementation of the ASPs from the service primitive library. It also, retrieves the C extensions from the extension library which we named, *C' Library*.

The compiler uses these libraries to produce the appropriate C' code. (The C' indicating an extension of the C programming language.)

To execute the produced prototype source, the user must provide a matching CRP for the prototypes. The CRP is manually implemented, should take into account the number of processes are to be created, and the interconnecting channels.

5.4 STEP FOUR: EVALUATION OF THE PROTOTYPE

Each generated prototype must be subjected to at least a basic operational test in order to determine that the code is executable.

The basic operational test is best achieved by an incremental approach where each protocol process is tested in isolation. A Test Driver is used to inject interactions at the Service_Access_Points associated with the protocol process. The behaviour of the process is monitored at the Service Access Points (SAPs). If the output interaction of the process fails to match the expected output interaction, the test case is marked as having failed. At the successful end of the isolation test, a basic integration test is required.

During the integration testing, two or more protocol processes are allowed to interact and their behaviour monitored by means of a Test Driver [PEPP83, URAL83b, URAL86]. For expediency, the number of protocol processes tested should be incremented by one at the end of each integration test.

The integration test is not complete until all protocol processes have been added and tested together.

In the next chapter, we present an application of our methodology to a Transport layer protocol.

Chapter VI

A SAMPLE OF AN EXECUTABLE CLASS 0 TRANSPORT PROTOTYPE

6.1 THE TRANSPORT PROTOTYPE

This chapter discusses the systematic development of a Class-0 Transport Prototype using the steps described in Chapter V.

For the implementation, we started with a formal specification of the Class-0 Transport Protocol written in ESTELLE. A formal specification listing is included in Appendix A. However, the listing has been shortened due to space limitation. The formal specification of the Class-0 Transport Protocol contains two channel type declarations: one channel describes the channels between the Transport layer module and a Session layer module, while the other declaration is for the channels between the Transport module and a Network layer module. The channel type between the Transport and Session modules is called *U_ACCESS_POINT*, and that between the Transport and Network modules is called *N_ACCESS_POINT*. The Abstract Service Primitives (ASPs) allowed in each channel type are as shown in the listing. (See [ISO84] for the detail syntax and semantics of ESTELLE.)

6.2 IMPLEMENTATION OF ABSTRACT SERVICE PRIMITIVES

Adhering to the description of step one in Chapter V, we identified all the Abstract Service Primitives (ASPs) used in the specification of the protocol. The primitives were manually coded using the C programming language. The source code for the primitives were then saved in a Library. The primitives implemented for the U_ACCESS_POINT channels are:

T_CONNECT_request

T_CONNECT_indication

T_CONNECT_response

T_CONNECT_confirm

T_DATA_request

T_DATA_indication

T_DISCONNECT_request

T_DISCONNECT_indication

The primitives implemented for the N_ACCESS_POINT channel types include the following:

N_CONNECT_request

N_CONNECT_indication

N_CONNECT_response

N_CONNECT_confirm

N_DATA_request

N_DATA_indication

N_DISCONNECT_request

N_DISCONNECT_indication

N_RESET_request

6.3 *GENERATING THE COMPILER*

The ESTELLE compiler was generated as discussed in step two of Chapter V. The YACC rules for ESTELLE were written and read by YACC to produce the parser. Appendix B contains the shortened listing of the rule given to YACC. Also, an abbreviated listing of the generated parser is included in Appendix C.

Similarly, the grammar rule used for LEX is shown in Appendix D. The grammar rule is read by LEX to produce the lexical analyzer[1] used by the parser. A shortened listing of the generated lexical analyzer is shown in Appendix E.

The generated listings of both the parser and the lexical analyzer were combined with the routines in one of the libraries to produce the ESTELLE compiler. (See Appendix F for illustrations.) Appendix G also discusses the translation strategy employed in going from ESTELLE to C.

6.4 *THE TRANSPORT PROTOTYPE*

The specification of the Class-0 Transport Protocol is compiled using our new ESTELLE compiler to produce a C language source code for the prototype. The shortened output of the ESTELLE compiler is as shown in Appendix H. Included in the source code of the prototype is the code for the ASPs used in the formal specification.

The C language source code for the Class-0 Transport Prototype is again compiled using the C Language compiler to produce a *Machine Executable Prototype.*

[1] N.B. the parser and the lexical analyzer are generated once for producing prototypes of formal communication protocol specifications.

6.5 *EVALUATION OF THE EXECUTABLE PROTOTYPE*

From Step Four of Chapter V, the basic conformance evaluation of the produced protocol process was conducted using a simple Black-Box testing system [DUNN84, LINN83a, LINN83b, MYER79, PROB83, URAL83a, URAL83b, URAL86]. The testing system consists of a Test Driver, and the protocol process providing the functions of the Class 0 Transport Protocol. The Test Driver reads a set of test cases from a file or terminal then formats the the message with the appropriate header, and sends it to the Prototype Under Test (PUT) through one of the available channels.

A detail design description is presented in the next section.

6.5.1 The Testing System

The Black-box Testing System is as shown in Figure 6.1. It consists of a stand-alone Test Driver. The Test Driver monitors the behaviour of the PUT. The Test Driver consists of two test beds :

1. an Upper Test Bed (UTB), and
2. a Lower Test Bed (LTB).

From the UTB, the Test Driver can inject Session module messages into the PUT, and intercept all Session module messages from the PUT. Similarly, it injects NSP messages into the PUT and intercept the NSP messages from the PUT at the LTB. The injected session messages are sent through socket A of the upper channel while the responses from the PUT are received through socket B of the upper channel. The intercepted messages are written to a file or console for later viewing.

6.5.2 Testing Expectations and Experience

For basic operational test, we expected the following:

1. the proper creation of the prototype as a process in the environment.

2. the correct configuration of the three communication channels connecting the prototype to:
   a. the upper layer process,
   b. the lower layer process, and
   c. the Communication Root Process.
3. last but not least, the behaviour of the generated executable prototype must comply with those of the given formal specification (which in this case, is the Class 0 Transport layer protocol). For each external input interaction, the output interaction(s) must be as given in the specification. The prototype is forced to timeout by preventing the Test Driver from inserting an acknowledgment (or reply).

Although the testing has not been thoroughly performed by the time of the writing of this thesis, we found the result to be impressive. The responsibility for thorough testing of the prototype has been assigned to another group of students at the University. In limited tests, the executable prototype was found to conform to the intended behaviour of the original protocol specification.

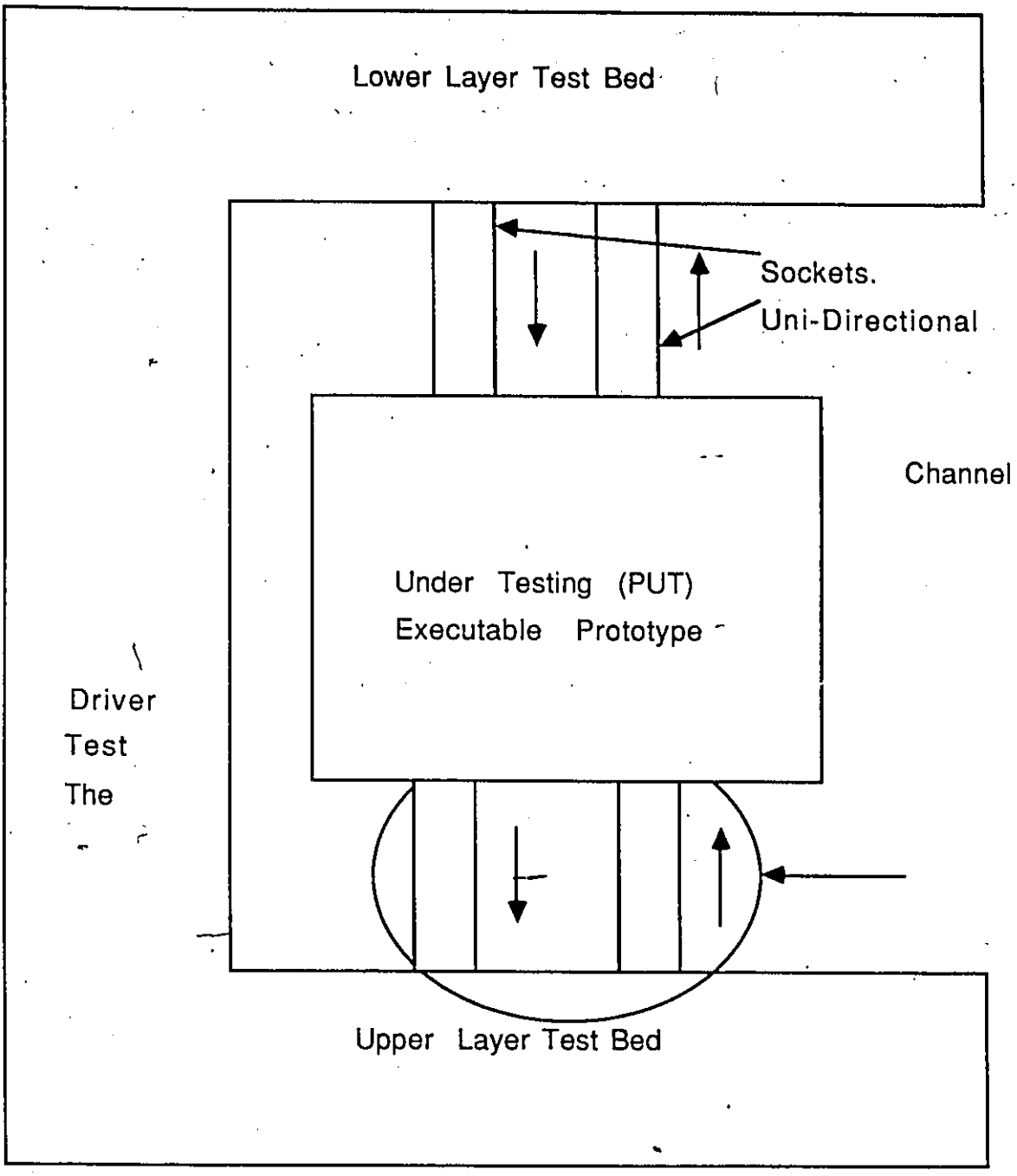


Figure 6.1: Architecture for testing the Transport Prototype.

Chapter VII

CONCLUSIONS AND SUGGESTIONS FOR FURTHER RESEARCH

Given formal specifications of an OSI protocols in ESTELLE, the author has demonstrated a semi-automated methodology for producing executable prototypes of these protocols. The methodology involved the use of a preprocessor. The preprocessor accepts single module specifications of the OSI protocols in ESTELLE as input, and produces the appropriate C programming language source files. Each source file is compiled separately by the C language compiler to produce the machine executable code. The produced machine codes are then integrated (or combined together) by a Communication Root Process (CRP) to produce the Communication System. The parser and lexical analyzer for the ESTELLE preprocessor are generated semi-automatically by the aid of YACC and LEX respectively.

7.1 ACCESSING THE PROTOTYPING METHODOLOGY

To properly access the semi-automated methodology for prototyping of communication protocols, we first present our experience with a manual implementation of the same class 0 Transport protocol discussed above, and the problems encountered.

7.1.1 Experience With a Manual Implementation

The semi-automated prototyping methodology presented in this thesis came as a result of work on manual implementations of the Class 0 and 2 Transport layer protocols. Our experience came from the actual manual implementations of Class 0 transport protocol and an attempt at a manual implementation of the Class 2 version of the same protocol.

For the manual implementations, we started with a formal specification of the protocols in ESTELLE. In fact, it is exactly the same formal specification used in chapter VI. From the formal specification, we manually implemented each ESTELLE construct into the equivalent C programming language construct(s).

The architecture of the manual implementation is identical to those of the semi-automated prototyping process provided in chapter V. This was not unexpected since the semi-automated technique was based on our experience with the manual implementations.

Further, there was a desire to minimize the size of the new software required for the automated prototyping. Instead, we are to make extensive use of the existing software including those from the manual implementations. The advantages of using existing software are numerous, including the fact that they have been tested and proven to work correctly; they are economically cheaper; and their use should encourage the industry to allocate more resources to the enhancement of this methodology (or investigate other semi-automated methodologies).

Thus, for the semi-automated prototyping of the Class 0 transport protocol in chapter VI, the software for the implementation of the channels and ASPs came directly from our library of service primitives used in the manual implementation of the Class 0 transport protocol. The ability to use a disjointly produced software further, illustrates the flexibility and practicality of our work.

As should be expected, the software for the manual implementation was significantly less than those of the semi-automated prototyping, and also executed faster.

But, the manual implementation was particularly demanding, mentally and physically very tedious and time-consuming. At each step of the implementation, decisions had to be made on the best technique for implementing various constructs. Inconsistency in the choice of techniques for implementing the non-distinct ESTELLE language constructs was also evident.

Furthermore, our attempt to produce a manual implementation of the Class 2 transport protocol from our Class 0 implementation was not easy. It meant re-coding most of the work we did for the Class 0 protocol. Cloning of some parts of the manually written software was necessary.

Based on the Class 2 experience, it is almost impossible to make use of any software produced for the Class 0 protocol for a different protocol such as the Network Layer Protocol. Our observation is that when it can be done, it is usually a daunting task and much less efficient.

7.1.2 Evaluating the Prototype Process

From the experience above, we were convinced that a tool for semi-automated prototyping is in order and can be justified. The tool will be used to implement the daunting aspects of manual protocol implementations. In particular, the aspect of the manual implementation which deals with the translation of ESTELLE constructs to the target implementation language constructs.

However, all is not well with semi-automated prototyping. There were considerable number of problems encountered when using a semi-automated approach to protocol development. The problems can be grouped into three classes:

1. code generation problems,
2. protocol semantic problems, and
3. Operating System related problems.

Some of these problems have been discussed in Chapter III.

Although most of these problems were solved by generating extra control statements, by code generation problems we mean those problems encountered when translating ESTELLE language constructs into their C language equivalent. Included in such list of problems is the translation of variant record types. The C programming language does support variant record types but not to the same level of complexity as in ESTELLE.

Variant types are defined in the C language using UNIONs, and UNION data types do not support tags. As a result, our ESTELLE compiler had to generate unique identifiers for variant type components. In addition to the variant types, SET data types also required special consideration. SET data types have been implemented as FIELD structures in the C language. Each element occupies one bit in the FIELD.

For debugging purposes, we have incorporated Input/Output routines which can be used in the formal specification. For simplicity, these input routines have the same syntax as that of the C language.

The protocol semantic problems include such concepts as Interaction Points, Queuing mechanisms, and channel implementations. The Operating System related problems include Inter-process communications, Input/Output operations, process scheduling and process management. Although many of these problems were successfully addressed, in some of the solutions, efficiency has been compromised for the ease of implementation.

For the Class-0 Transport Protocols tested, we observed that given 500 statements in ESTELLE our compiler produced 581 statements (excluding macro definitions) in the C language. This is approximately, 16 percent statements more than the given formal specification. This number will of course vary from protocol to protocol. It was observed that this percentage will increase as the number of SET and SUBRANGE variables increase. Thus, protocol designers are advised to avoid sets wherever possible. Due to the small number of statements generated, the generated source code can visually be compared with the given formal specifications.

Given that the preprocessor is produced once (except in cases of testing or product enhancements), we observed that the total effort required to implement the abstract service primitives and interaction points are the same for the semi_automated prototyping as in the case of the manual implementation. This observation is complete and valid for all layered protocols such as the OSI protocols. This is atested to by the fact that we used the same software in our manual implementation for our semi-automated implementation.

Where automated prototyping is beneficial is in maintaining consistency of implementation for all applications. This consistency is a key to product improvement and cost reduction. Errors found in testing one instance of protocol are immediately fixed in the preprocessor, and all other implementations benefit from this added value.

Reduction in man-power requirement is a major benefit. The labour required to translate the EFSM to the target implementation language can be expended on other meaningful projects.

7.1.3 Limitations

Although the preprocessor will recognize the following ESTELLE constructs, they are not yet implemented. Below is the list of the constructs and estimate on their implementation complexity:

- prioritized transitions:

  This is easily implemented in our approach.

- spontaneous transitions:

  This is easily implemented in our approach.

- WITH statements:

  This can be implemented but not easily. Any construct with the WITH statement will be treated as a macro. All references to elements of the record structure in the WITH statement will be expanded to their full representation. In short, the WITH statement will be treated as if it were a macro.

  Another possible technique is to use pointers. This is currently used in many programming languages. In these languages, instead of using the WITH statement, one uses a BIND statement. The BIND statement allows the programmer to associate a variable with an identifier over a certain scope. So, to reference an field in the variable, the designer must either use the bound identifier as a prefix, or use the explicit reference to the field.

- ANY clauses:

  This is easily implemented in our approach.

The *rendezvous* concept cannot be supported by approach without a major overhaul of the entire design philosophy. Thus, no rendezvous protocol will be supported. However, protocol designers can still utilize our tool to translate the transition part (or EFSM) of their formal protocol specifications into C language. In which case, they must edit the output of the preprocessor associated channel implementations.

Finally, as discussed in section 6.5.2, only very limited testing has been carried out. Further testing is recommended.

7.2 AREAS OF FURTHER RESERACH

In future endeavours, we will work on incorporating the following constructs into our work:

- prioritized transitions.
- spontaneous transitions.
- WITH statements.
- ANY clauses.

In addition, a Targeter will be added to our system. The compiler will be modified so it produces an intermediate code which is independent of any existing programming language. This language independent output can then be read by the targeter to produce specific source code in many programming languages for the different execution environments.

A Protocol Debuger is also slated for the future. The debuger facilities will be closely coupled to the compiler and will be used to enhance testing of generated prototypes. The debuger utility will be provided as an option during compilation. The debuger will also collect run-time informations which inform designers of problems such as, Unspecified Reception, Deadlock, Unexecutable Reception, and Live-Lock.

7.3 CONCLUSION

In conclusion, it is our opinion that the main foundation has been laid on which more sophisticated protocols can be produced. We have met the objectives of this work; development of an automated prototyping tool; an approach for determining and isolating all operating system protocol dependent aspects; and we have also demonstrated support for the adequacy of ESTELLE specifications for semi-automated implementation of communication protocols.

Appendix A

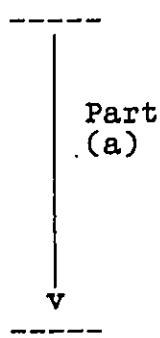
EXAMPLE OF THE SPECIFICATION OF A TRANSPORT PROTOCOL IN ESTELLE

Below, is a partial listing of the formal specification of the Class-0 Transport protocol. The formal specification consists of the Channel and Module specification sections.

The channel specification section is made of parts (a), (b) and (c). Where part (a), (b) and (c) are the Constant, Type and Channel declaration parts respectively.

The Module specification consists of the parts (d), (e), (f) and the Transition part. Where parts (d), (e) and (f) are the declarations of the module Variables, Subprograms (or routines) and the Initialization procedure respectively.

```
(*********************************)
(* CONSTANT DECLARATION          *)
(*********************************)
 const
    null              = NULL;
    buffsize          = BUFFSIZE;
    timel             = TS1;
    maxseq            = 8  ;
     :
     :
     :
     :
```



```
(*********************************)
(* TYPE     DECLARATION          *)
(*********************************)
type
   seq_type      =  (. 1 .. maxseq .);
   int_type      =  integer;
   ref_type = integer;
   N_addr_type     = integer;
   T_addr_type     = integer;
   u_data_type = array (.1 .. UDATA_SIZE .) of char;
   boolean         = ( TRUE,
                       FALSE
                     );
   reason_type     = ( RESET,
                       INVALID_TPDU,
                       UNEXPECTED_CC,
                       UNACCEPTABLE_REQUEST,
                       PROTOCOL_ERROR,
                       :
                       :
                       :
                       TS_USER_UNKNOWN,
                       CALLED_TS_USER_UNKNOWN
                     );
   pdu_name        = ( NO_NAME,
                       AK,
                       CR,
                       CC,
                       DR,
                       DT,
                       EA,
                       ER
                     );
   class_type      = ( CLASS_0 );
   frame_type      = ( BASIC, EXTENDED );
   mark_type       = ( not_TSDU, TSDU);
   tpdu_type       =
       RECORD  (*includes all possible fields in a *)
               (*class 0 protocol *)
            li               : int_type;
            name             : pdu_name;
            cdt              : seq_type;
            dst_ref          : ref_type;
            src_ref          : ref_type;
            class            : class_type;
            option           : frame_option;
            calling_TSAP_id  : T_addr_type;
            called_TSAP_id   : T_addr_type;
            size             : int_type;
            EOT              : boolean;
            tpdu_nr          : int_type;
            user_reason      : reason_type; (* user *)
                               (* provided reason *)
```

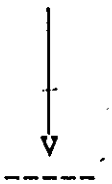
Part (b)

```
        prot_reason     : reason_type;
                          (*TS_provider reason  *)
        user_data       : u_data_type;
    END;
major_state   =
                ( CLOSED,
                  OPEN,
                  WFCC,
                  WFNC,
                  WFTRESP
                );
```

```
(*********************************)
(* CHANNEL  DECLARATION          *)
(*********************************)
channel U_ACCESS_POINT ( USER, PROVIDER );
   by USER :
         T_CONNECT_request (
               called_TSAP_id   : T_addr_type;
               calling_TSAP_id : T_addr_type;
               exp_data_option : int_type;
               quality_of_service: int_type
                           );
         T_CONNECT_response (
               quality_of_service: int_type;
               responding_TSAP_id: T_addr_type;
               exp_data_option    : int_type
                           );
         T_DATA_request      (
               TS_user_data : u_data_type
                           );
         T_DISCONNECT_request (
               TS_user_data : u_data_type
                           );
   by PROVIDER:
         T_CONNECT_indication (
               called_TSAP_id   : T_addr_type;
               calling_TSAP_id : T_addr_type;
               exp_data_option : int_type;
               quality_of_service: int_type
                           );
         T_CONNECT_confirm  (
               quality_of_service: int_type;
               responding_TSAP_id: T_addr_type;
               exp_data_option    : int_type
                           );
         T_DATA_indication  (
               user_data : u_data_type
                           );
         T_DISCONNECT_indication (
               TS_user_data      : u_data_type
                           );
channel N_ACCESS_POINT ( USER, PROVIDER );
   by USER :
         N_CONNECT_request (
               called_NSAP_id   : N_addr_type;
               calling_NSAP_id : N_addr_type;
               NS_user_data       : TPDU_type
               quality_of_service: int_type
         N_DATA_request (TPDU : pdu_type);
         N_DISCONNECT_request (
               NS_user_data : NS_u_data_type);
   by PROVIDER:
         N_CONNECT_confirm     (
               called_NSAP_id   : N_addr_type;
```

Part (c)

```
        calling_NSAP_id : N_addr_type;
        NS_user_data    : TPDU_type
        quality_of_service: int_type);
N_DISCONNECT_Indication (
        NS_user_data : NS_u_data_type
```



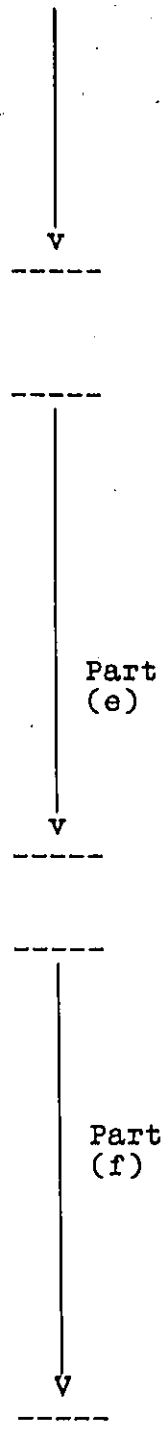
```
(*********************************)
(* MODULE SPECIFICATION PART     *)
(*********************************)
MODULE TRANSPORT_ENTITY
         ( U : U_ACCESS_POINT ( PROVIDER )  queued;
           N : N_ACCESS_POINT (USER)  not queued

(*********************************)
(* VARIABLE DECLARATION          *)
(*********************************)
VAR
   L_net_addr ,     (* local network  address *)
   R_net_addr ,     (* remote network address *)
   called_NSAP_id,
   calling_NSAP_id : N_addr_type;
   L_addr ,       (* LOCAL TRANSPORT ADDR *)
   R_addr ,       (* REMOTE TRANSPORT ADDR *)
   called_TSAP_id ,
   calling_TSAP_id ,
   responding_TSAP_id : T_addr_type;
   src_ref ,      (* source ref number  *)
   dst_ref ,      (* destination ref number *)

   (* It has been assumed that the address scheme    *)
   (* used for the transport addr is a concatenation *)
   (* of the network addr the user identifier number *)

   L_user_id,                (* a user ref number *)
   R_user_id  : ref_type; (* identifies a particular *)
                        (* user of a transport entity *)

   send_seq ,           (* sequence of TPDUs sent *)
   send_next_seq,
   cdt   ,
   tpdu_nr ,
   recv_seq   : seq_type;

   reason     ,   (* reason for disconnection *)
   user_reason ,  (* reason specified by user *)
   more_info  :reason_type;
   max_TPDU_size ,
   TPDU_size     ,
   udata_size    ,
   max_data_size ,
   header_size   :  int_type;
   owner,
   CC_sent,
   CR_sent,
   REQ_sent,
   CC_recved,
   CR_recved,
   REQ_recved ;
   EOT             :  boolean;
```

Part (d)

```
    quality_of_service,
    option      :  frame_type;
    class           : class_type;
    trans_tpdu_buff,
    recv_tpdu_buff    :  tpdu_buffer;
    user_data    :  u_data_type
    out_TPDU     :  @tpdu_type;
    STATE        :  major_state;          --
```

```
(*********************************)
(* PROCEDURE DECLARATION         *)
(*********************************)
procedure   transmit ( seq : seq_type);
BEGIN
     DATA := retrieve ( Tx_buffer,
                        seq
                      );
     out N.N_DATA_request(DATA);
     RELEASE ( Tx_buffer,
               seq
             );
 END;
     :
     :
     :
```

```
(*********************************)
(* INITIALIZATION CODE           *)
(*********************************)
initialize ;
begin
     STATE   to   CLOSED ;
     remote_credit      :=  0;
     user_reason        :=  NORMAL;
     reason             :=  NORMAL;
     owner              :=  false;
     :
          :
          :
          :
end;
```



```
(*********************************)
(* TRANSITIONS                   *)
(*********************************)

(* from  CLOSED STATE *)
trans
     from CLOSED
        when N.N_DATA indication
           provided  (( N.in_TPDU.name = AK )
                          or
                     ( N.in_TPDU.name = EA )
                          or
                     ( N.in_TPDU.name = ER )
                          or
                     ( N.in_TPDU.name = DC )
                          or
                     ( N.in_TPDU.name = DT ))
              to   SAME
                begin
                end;
trans
     from CLOSED
        when N.N_DATA indication
           provided  (( N.in_TPDU.name = CC )
           to   SAME
           begin
              dst_ref := N.in_TPDU.src_ref;
              src_ref := 0;
              reason  := UNEXPECTED_CC;
              out_TPDU := pduf ( DR,
                                 null,
                                 dst_ref,
                                 src_ref,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 reason,
                                 null
                               );
              out N.DATA_request (TPDU);
           end;
```

```
(* from  WAIT_FOR_NETWORK_CONNECTION STATE *)
trans
     from WFNC
        when N.N_CON_confirm
         to   WFCC
           begin
               src_ref := alloc_ref;  (* allocates a reference *)
                                      (* number to the pending *)
                                      (* transport connection  *)
               dst_ref := 0;
               cdt     := 0;
               class   := CLASS_0;
               option  := BASIC;
               out_TPDU    := pduf ( CR,
                                     cdt ,
                                     dst_ref,
                                     src_ref,
                                     class,
                                     option,
                                     calling_TSAP_id,
                                     called_TSAP_id,
                                     TPDU_size,
                                     null,
                                     null,
                                     null,
                                     null,
                                     null

                                   );

               out N.N_DATA_request ( out_TPDU );
           end;
```

```
(* from  WAIT_FOR_CONNECTION_CONFIRM STATE *)
trans
     from WFCC
        when N.N_DISC_indication
         to   CLOSED
           begin
               reason := NETWORK_DISCONNECTION;
               out U.T_DISC_indication ( reason );
               release_all;
           end;
trans
     from WFCC
        when N.N_DATA_indication
           provided  (( N.in_TPDU.name = CC )
                          and
                     ( unacceptable_CC ( N.in_TPDU )))
           to   CLOSED
           begin
               dst_ref := N.in_TPDU.src_ref;
               reason  := NEGOTIATION_FAILED;
               out U.T_DISC_indication ( reason );
               out N.N_DISC_request;
               release_all;
           end;
trans
     from WFCC
        when N.N_DATA_indication
           provided  (( N.in_TPDU.name = CC )
                          and
                     ( not unacceptable_CC ( N.in_TPDU )))
           to   OPEN
           begin
              TPDU_size := N.in_TPDU.size;
              remote_CRedit := N.in_TPDU.cdt;
              responding_TSAP_id := called_TSAP_id;
              out U.T_CON_confirm ( quality_of_service,
                                    responding_TSAP_id,
                                    null
                                  );
           end;
```

```
(* from  WAIT_FOR_T_CON_RESPONSE STATE *)
trans
     from WFTRESP
        when U.T_CON_response
        to   OPEN
        begin
            out_TPDU     := pduf ( CC,
                                   cdt ,
                                   dst_ref,
                                   src_ref,
                                   class,
                                   option,
                                   calling_TSAP_id,
                                   called_TSAP_id,
                                   TPDU_size,
                                   null,
                                   null,
                                   null,
                                   null,
                                   null
                                 );
            out N.N_DATA_request ( out_TPDU );
        end;   ( * trans * )
trans
     from WFTRESP
        when U.T_DISC_request
        to CLOSED
        begin
            reason := USER_INITIATED;
            more_info := U.disc_reason;
            out_TPDU := pduf ( DR,
                               null,
                               dst_ref,
                               src_ref,
                               null,
                               null,
                               null,
                               null,
                               null,
                               null,
                               null,
                               more_info,
                               reason,
                               null
                             );
            out N.N_DATA_request ( out_TPDU );
        end;
```

```
(* from  OPEN STATE *)
trans
     from OPEN
        when N.N_DATA_indication
           provided  (( N.in_TPDU.name = DT )
                          and
                      ( not invalid_tpdu ( N.in_TPDU )))
           to   SAME
           begin
              recv_seq := recv_seq + 1;
              merge ( recv_tpdu_buff,
                      N.in_TPDU.user_data,
                      recv_seq
                    );
              if ( N.in_TPDU.EOT ) then
                 mark ( recv_tpdu_buff,
                        recv_seq,
                        tsdu
                      );
              if    TSDU_in ( recv_tpdu_buff ) then
              begin
                 user_data := extract_TSDU ( recv_tpdu_buff );
                 out U.T_DATA_request ( user_data );
              end
           end;
```

```
trans
     from OPEN
        when U.T_DATA_request
        to   SAME
        begin
           tpdu_nr := 0; (* THE SOURCE REF *)
           while ( length ( U.user_data ) >
                   ( max_TPDU_size  - header_size )) do
           begin
              user_data := extract ( U.user_data,
                                     max_TPDU_size -
                                     header_size
                                   );
              EOT    := false;
              tpdu_nr := 0;
              out_TPDU := pduf ( DT,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 null,
                                 EOT,
                                 tpdu_nr ,
                                 null,
                                 null,
                                 user_data
                               );
              store ( trans_tpdu_buff,
                      send_seq,
                      out_TPDU
                    );
              send_seq := send_seq + 1;
           end;
           user_data = extract ( U.user_data,
                                 max_TPDU_size -
                                 header_size
                               );
           EOT    := false;
           tpdu_nr := 0;
           out_TPDU := pduf ( DT,
                              null,
                              null,
                              null,
                              null,
                              null,
                              null,
                              null,
                              null,
                              EOT,
                              tpdu_nr ,
```

```
                                null,
                                null,
                                user_data
                              );
        store ( trans_tpdu_buff,
                send_seq,
                out_TPDU
              );
        send_seq := send_seq + 1;
        send_next_seq := 0;
        while ( send_next_seq < send_seq ) do
        begin
             transmit ( send_next_seq );
             send_next_seq := send_next_seq + 1;
        end;
    end;  (* trans *)
```

Appendix B

EXAMPLES OF YACC RULES

Below is a partial listing of some of the rules used to generate the parser for the Preprocessor of the formal protocol specification.

The listing contains the declaration of the tokens or terminal symbols and non-terminal symbols. More information on the format of the rules can be found in [BELL83]. We have represented the terminal symbols as upper case letters while the non-terminal symbols are encoded in lower case letters. Shown in the listing is the rules for ESTELLE statements. For example, a statement is one of the following:

- simple statements
- state-initialization statements
- output statements
- repetitive statements
- conditional statements
- compound statements
- labelled statements

Where for instance a simple statement is defined as:

1. an assignment statement or
2. a goto statement or
3. a procedure call statement.

```
%
#include  "yydef"
%
%start  system
%union begin
          int ival;
          char cval [MAXLEN];
       end
%token   <cval>   ALL
%token   <cval>   AND
%token   <cval>   ANY
%token   <cval>   ARRAY
%token   <cval>   ASSIGN
%token   <cval>   BEGINN
%token   <cval>   BY
%token   <cval>   CASE
%token   <cval>   INTEGER
%token   <cval>   USER
%token   <cval>   VAR
%token   <cval>   WHEN
%token   <cval>   WHILE
%token   <cval>   WITH

      /* precedence information about operators */
%left PLUS MINUS OR
%left MULT DIVIDE AND
%left UMINUS   /* precedence for unary minus */
```

```
%%
action_part        : BEGINN
        {
          if (first_trans == 1)
          {
             fprintf(yyout,"\n/*********************************/");
             fprintf(yyout,"\n/**                             **/");
             fprintf(yyout,"\n/**                             **/");
             fprintf(yyout,"\n/**    TRANSITION  PART         **/");
             fprintf(yyout,"\n/**                             **/");
             fprintf(yyout,"\n/**                             **/");
             fprintf(yyout,"\n/*********************************/");
             fprintf(yyout,"\n\n\n");
             fprintf(yyout,"\nSWITCH ( context[CEP_id].state )");
             fprintf(yyout,"\nBEGIN\n");
          }
          if ((strcmp(from_stat,old_from_stat) != 0) &&
              (from_flag == 1))
          {
             if (first_trans != 1)
             {
                fprintf(yyout,
                   "\n          END  /* end of EVENT */");
                fprintf(yyout,"\n          BREAK;\n");
                fprintf(yyout,"\n    END  /* end of STATE */");
                fprintf(yyout,"\n    BREAK; \n");
             }
```

```
        fprintf(yyout,"\n   FROM %s :\n",from_stat);
        fprintf(yyout,"\n   BEGIN\n");
        fprintf(yyout,"\n      SWITCH ");
        fprintf(yyout," ([context[CEP_id].event )");
        fprintf(yyout,"\n      BEGIN\n");
        strcpy(old_when_stat,"");  /*reset when state */
        strcpy(old_from_stat,from_stat);
        first_trans = 1;
     }
     if ((strcmp(when_stat,old_when_stat) != 0) &&
         (when_flag == 1))
     {
        if (first_trans != 1 )
        {
           fprintf(yyout,
                "\n          END  /* end of EVENT */");
           fprintf(yyout,"\n          BREAK;\n\n");
        }
        fprintf(yyout,"\n          WHEN %s :\n",when_stat);
        fprintf(yyout,"\n          BEGIN");
        strcpy(old_when_stat,when_stat);
     }
     fprintf(yyout,"\n\n             BEGIN  /* beginning");
     fprintf(yyout," of transition */");
     indent = 4;
     if (provided_flag == 1)
     {
        INDENT;
        fprintf(yyout,"PROVIDED ( %s )",provided_stat);
     }
     INDENT;
     fprintf(yyout,"BEGIN");
     indent = 5;
     if (to_flag == 1)
     {
       INDENT;
       fprintf(yyout,"context[CEP_id].STATE = %s;",to_stat);
     }
  }
statement_seq END
  {
     INDENT;
     fprintf(yyout,"yyenter = YYES;");
     indent -4;
     INDENT;
     fprintf(yyout,"END");
     fprintf(yyout,"\n\n             END ");
     fprintf(yyout,"  /* end of transition */");
     fprintf(yyout,"\n             if (yyenter == YYES)");
     fprintf(yyout,"\n             BREAK;");
     first_trans = 0;
     from_flag = 0;
     when_flag = 0;
     to_flag = 0;
```

```
            provided_flag =0;
          }
      ;

statement_seq     : statement
          {
             NEXTSTAT;
          }
      | statement_seq SEMICOLON statement
          {
             NEXTSTAT;
          }
      ;
assign_stat       : variable_access ASSIGN expression
          {
             INDENT;
             fprintf(yyout,"%s %s %s",$1,$2,$3);
          }
      ;
case_statement    : CASE  expression OF
          {
             INDENT;
             fprintf(yyout,"%s (%s)",$1, $2,$3);
             INDENT;
             fprintf(yyout,"BEGIN\n");
             indent++;
          }
       case_list_part END
          {
             indent--;
             INDENT;
             fprintf(yyout,"END\n");
          }
    ;
compound_stat     : BEGINN
          {
              INDENT;
              fprintf(yyout,"%s\n",$1);
              indent++;
          }
       statement_seq END
          {
              indent--;
              INDENT;
              fprintf(yyout,"%s\n",$4);
          }
    ;
condition_stat    : if_statement
                  | case_statement
    ;
```

```
else_part        : ELSE
        {
            INDENT;
            fprintf(yyout,"%s\n");
            indent++;
        }
      statement
        {
            indent--;
        }
    ;
for_statement    : FOR IDENTIFIER ASSIGN expression to_downto
                   expression DO statement
    ;
goto_stat        : GOTO label
    ;
if_statement     : IF expression THEN
        {
            INDENT;
            fprintf(yyout,"%s (%s) %s",$1,$2,$3);
            indent++;
        }
     statement
        {
            indent--;
        }
   | if_statement else_part
    ;
initialization_part : INITIALIZE BEGINN
        {
            fprintf(yyout,"\n\n%s()",$1);
            fprintf(yyout,"\n%s",$2);
            indent = 1;
        }
      statement_seq END  SEMICOLON
        {
            fprintf(yyout,"\n%s",$5);
            indent = 1;
        }
     ;
```

```
label           : UNSIGNED_NUMBER ;
output_stat     : OUT IDENTIFIER DOT IDENTIFIER
      {
         INDENT;
         fprintf(yyout,"%s()",$4);
      }
    | OUT IDENTIFIER DOT IDENTIFIER  LEFT  RIGHT
      {
         INDENT;
         fprintf(yyout,"%s()",$4);
      }
    | OUT IDENTIFIER DOT IDENTIFIER  LEFT argument_list RIGHT
      {
         INDENT;
         fprintf(yyout,"%s(%s)",$4,$6);
      }
  ;
repeat_stat     : REPEAT
      {
          labelyy++;
          INDENT;
          fprintf(yyout,"%d : %s\n",labelyy,$1);
          indent++;
      }
     statement_seq UNTIL expression
      {
          indent--;
          labelyy++;
          INDENT;
          fprintf(yyout,"%s ( %s) GOTO %d\n",$4,$5,labelyy);
      }
   ;
repetitive_stat : repeat_stat | while_statement | for_statement;
simple_stat     : assign_stat | goto_stat | proc_call;
proc_call       : IDENTIFIER LEFT argument_list RIGHT
      {
         strcpy($$,$1);
         strcat($$,$2);
         strcat($$,$3);
         strcat($$,$4);
      }
     | IDENTIFIER LEFT  RIGHT
      {
         strcpy($$,$1);
         strcat($$,$2);
         strcat($$,$3);
      }
   ;
```

```
statement       : simple_stat
                | state_init
                | output_stat
                | repetitive_stat
                | condition_stat
                | compound_stat
                | label
                  COLON
                    {
                          fprintf(yyout,"\n%s :\n ",$1);
                    }
                  statement
    ;
to_downto       : TO
                {
                    strcpy($$,"++");
                }
                | DOWNTO
                {
                    strcpy($$,"--");
                }
    ;
while_statement   : WHILE expression DO
                  {
                    INDENT;
                   fprintf(yyout,"%s  %s  %s",$1,$2,$3);
                    NEXTLINE;
                    indent++;
                  }
          statement
                  {
                    indent--;
                  }
    ;
%%
# include "lex.yy.c"
# include "lib.y"
```

Appendix C

PARTIAL YYPARSE LISTING

Below is the corresponding listing generated by YACC for the parser. The listings have been shortened because of space.

```
# line 7 "types.y"
#include  "yydef"
# line 12 "types.y"
typedef union  {
     int ival;
     char cval [MAXLEN];
 } YYSTYPE;
# define ALL 257
# define AND 258
# define ANY 259
# define ARRAY 260
# define ASSIGN 261
# define BEGINN 262
# define BY 263
# define CASE 264
  :
  :
  :
```

```
  :
  :
  :
# define INTEGER 337
# define USER 338
# define VAR 339
# define WHEN 340
# define WHILE 341
# define WITH 342
# define UMINUS 343
#define yyclearin yychar = -1
#define yyerrok yyerrflag = 0
extern int yychar;
extern short yyerrflag;
#ifndef YYMAXDEPTH
#define YYMAXDEPTH 150
#endif


YYSTYPE yylval, yyval;
# define YYERRCODE 256
# line 2246 "types.y"
# include "lex.yy.c"
# include "lib.y"
short yyexca[] {
-1, 1,
  0, -1,
  -2, 0,
-1, 136,
  :
  :
  :
  :
  :
-1, 396,
  328, 20,
  -2, 12,
  };
```

```
# define YYNPROD 258
# define YYLAST 666
short yyact[]={
 186, 188, 414, 416, 135,  35, 334, 198, 309, 216,
  14, 223, 174, 143, 217, 118, 278, 118,  36, 179,
 :
 :
 :
 :
 :
-1000,-253,-1000,-1000,-1000,-237,-1000,-1000 };
 :
 :
 :
 :
 :
#ifndef lint
static char yaccpar_sccsid[] =
                "@(#)yaccpar     4.1     (Berkeley)2/11/83";
#endif not lint
#


# define YYFLAG -1000
# define YYERROR goto yyerrlab
# define YYACCEPT return(0)
# define YYABORT return(1)
/*parser for yacc output  */
#ifdef YYDEBUG
int yydebug = 0; /* 1 for debugging */
#endif
YYSTYPE yyv[YYMAXDEPTH]; /* where the values are stored */
int yychar = -1; /* current input token number */
int yynerrs = 0;  /* number of errors */
short yyerrflag = 0;  /* error recovery flag */
yyparse()
  short yys[YYMAXDEPTH];
  short yyj, yym;
  register YYSTYPE *yypvt;
  register short yystate, *yyps, yyn;
  register YYSTYPE *yypv;
  register short *yyxi;
```

```
  yystate = 0;
  yychar = -1;
  yynerrs = 0;
  yyerrflag = 0;
  yyps= &yys[-1];
  yypv= &yyv[-1];
 yystack:    /* put a state and value onto the stack */
#ifdef YYDEBUG
  :
  :
  :
  :
  :

case 1:
# line 180 "types.y"
{
                strcpy(funct_head,yypvt[-1].cval);
              } break;
case 2:
# line 184 "types.y"
{
                lscope = 0;
                procfun = 0;
                strcpy(scope,"global");
              } break;
  :
  :
  :
  :
  :
case 203:
# line 2023 "types.y"
{
                  NEXTSTAT;
               } break;
case 204:
# line 2027 "types.y"
{
                  NEXTSTAT;
               } break;


case 205:
# line 2034 "types.y"
{
                  INDENT;
                  fprintf(yyout,"%s %s %s",
                  yypvt[-2].cval,yypvt[-1].cval,yypvt[-0].cval);
               } break;
```

```
case 214:
# line 2064 "types.y"
{
                INDENT;
                fprintf(yyout,"%s %s %s",yypvt[-2].cval,
                yypvt[-1].cval,yypvt[-0].cval);
} break;
case 217:
# line 2076 "types.y"
{
  INDENT;
  fprintf(yyout,"%s (%s)",yypvt[-2].cval,
  yypvt[-1].cval,yypvt[-0].cval);
  INDENT;
  fprintf(yyout,"BEGIN\n");
  indent++;
} break;
case 218:
# line 2085 "types.y"
{
  indent--;
  INDENT;
  fprintf(yyout,"END\n");
} break;
case 219:
# line 2092 "types.y"
{


   INDENT;
   fprintf(yyout,"%s\n",yypvt[-0].cval);
   indent++;
 } break;
case 220:
# line 2099 "types.y"
{
   indent--;
   INDENT;
   fprintf(yyout,"%s\n",yypvt[-0].cval);
 } break;
case 223:
# line 2109 "types.y"
{
   INDENT;
   fprintf(yyout,"%s\n");
   indent++;
 } break;
case 224:
# line 2115 "types.y"
{
   indent--;
} break;
```

```
case 227:
# line 2131 "types.y"
{
  INDENT;
  fprintf(yyout,"%s (%s) %s",yypvt[-2].cval,yypvt[-1].cval,yypvt[-
  indent++;
 } break;
case 228:
# line 2137 "types.y"
{
  indent--;
 } break;
case 230:
# line 2144 "types.y"
{
  fprintf(yyout,"\n\n%s()",yypvt[-1].cval);
  fprintf(yyout,"\n%s",yypvt[-0].cval);
  indent = 1;
 break;
case 231:
# line 2151 "types.y"
{
  fprintf(yyout,"\n%s",yypvt[-1].cval);
  indent = 1;
 } break;
case 233:
# line 2159 "types.y"
{
  INDENT;
  fprintf(yyout,"%s()",yypvt[-0].cval);
} break;


case 234:
# line 2164 "types.y"
{
  INDENT;
  fprintf(yyout,"%s()",yypvt[-2].cval);
} break;
case 235:
# line 2169 "types.y"
{
  INDENT;
  fprintf(yyout,"%s(%s)",yypvt[-3].cval,yypvt[-1].cval);
} break;
case 236:
# line 2175 "types.y"
{
   labelyy++;
   INDENT;
   fprintf(yyout,"%d : %s\n",labelyy,yypvt[-0].cval);
   indent++;
} break;
```

```
case 237:
# line 2183 "types.y"
{
   indent--;
   labelyy++;
   INDENT;
   fprintf(yyout,"%s ( %s) GOTO %d\n",
   yypvt[-1].cval,yypvt[-0].cval,labelyy);
} break;
case 244:
# line 2199 "types.y"
{
  strcpy(yyval.cval,yypvt[-3].cval);
  strcat(yyval.cval,yypvt[-2].cval);
  strcat(yyval.cval,yypvt[-1].cval);
  strcat(yyval.cval,yypvt[-0].cval);
 } break;
case 245:
# line 2206 "types.y"
{
  strcpy(yyval.cval,yypvt[-2].cval);
  strcat(yyval.cval,yypvt[-1].cval);
  strcat(yyval.cval,yypvt[-0].cval);
 } break;
case 252:
# line 2220 "types.y"
{
 fprintf(yyout,"\n%s :\n ",yypvt[-1].cval);
 } break;
case 254:
# line 2226 "types.y"


{
   strcpy(yyval.cval,"++");
 } break;
case 255:
# line 2230 "types.y"
{
   strcpy(yyval.cval,"--");
 break;
case 256:
# line 2235 "types.y"
{
   INDENT;
  fprintf(yyout,"%s  %s  %s",yypvt[-2].cval,
  yypvt[-1].cval,yypvt[-0].cval);
   NEXTLINE;
   indent++;
} break;
```

```
case 257:
# line 2242 "types.y"
{
   indent--;
} break;
 :
 :
 :
 :
 :
```

Appendix D

EXAMPLES OF LEX RULES

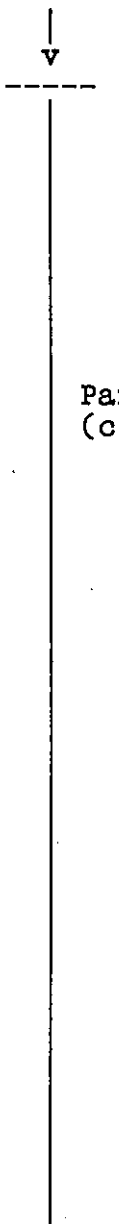
This appendix shows the listing of the source file that was read by LEX to produce the appropriate lexical analyzer called YYLEX() for the ESTELLE compiler. The first line requests a C language source code for the lexical analyzer. Part (a) contains the declaration of the global variables used by the lexical analyzer. Part (b) contains the declaration of the macros used in the writing of the rules. Part (c) contains the lexical rules and their action parts. Part (d) normally will contain the procedures used by the analyzer but in this example none is included.

```
%C
%{
int numb_file ;
int in_argc;
char *In_argv[10];
FILE *sysin , *sysout;
%}
blank_line ¬[ \t]*$
letter     [a-zA-Z]
digit[0-9]
number     [0-9]+
alphanum   [a-zA-Z0-9]
sign [-+]
identifier [a-zA-Z][a-zA-Z0-9_]*
```

Part (a)

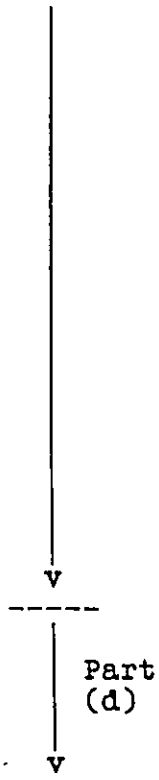
Part (b)

```
string      \"[¬"]*\"
comment     [ \t]*"(*"
%%
all {
          strcpy(yylval.cval,"ALL");
          return(ALL);
    }
and {
          strcpy(yylval.cval,"AND");
          return(AND);
    }
any {
          strcpy(yylval.cval,"ANY");
          return(ANY);
    }
array  {
          strcpy(yylval.cval,"ARRAY");
          return(ARRAY);
    }
begin {
          strcpy(yylval.cval,"BEGIN");
         return(BEGINN);
    }
by {
          strcpy(yylval.cval,"BY");
         return(BY);
    }
case {
          strcpy(yylval.cval,"CASE");
         return(CASE);
    }
channel {
          strcpy(yylval.cval,"CHANNEL");
         return(CHANNEL);
    }
common  {
          strcpy(yylval.cval,"COMMON");
         return(COMMON);
    }
```



Part (c)

```
const  {
          strcpy(yylval.cval,"CONST");
         return(CONST);
       }
delay {
          strcpy(yylval.cval,"DELAY");
         return(DELAY);
       }
       :
       :
       :
       :
       :
¬[ \t]+  {
          strcpy(yylval.cval,yytext);
       }
[\n]+  {
          stat_line ++;
       }
%%
       :
       :
       :
       :
       :
```



Appendix E

PARTIAL YYLEX LISTING

Below is the generated listing by LEX in C language. Part (a) contain some of the variables manipulated by the lexical analyzer. Part (b) is the code generated for YYLEX().

```
# include "stdio.h"
# define U(x) x
# define NLSTATE yyprevious=YYNEWLINE
# define BEGIN yybgin = yysvec + 1 +
# define INITIAL 0
# define YYLERR yysvec
# define YYSTATE (yyestate-yysvec-1)
# define YYOPTIM 1
# define YYLMAX 200
# define output(c) putc(c,yyout)
```

```
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr){
getc(yyin))==10?(yylineno++,yytchar)}
yytchar)==EOF?0:yytchar)
# define unput(c) {yytchar= (c);
if(yytchar==\n)yylineno--;*yysptr++=yytchar;}
# define yymore() (yymorfg=1)
# define ECHO fprintf(yyout, "%s",yytext)
# define REJECT { nstr = yyreject(); goto yyfussy;}
int yyleng; extern char yytext[];
int yymorfg;
extern char *yysptr, yysbuf[];
int yytchar;
FILE *yyin ={stdin}, *yyout ={stdout};
extern int yylineno;
struct yysvf {
  struct yywork *yystoff;
  struct yysvf *yyother;
  int *yystops;};
struct yysvf *yyestate;
extern struct yysvf yysvec[], *yybgin;
int numb_file ;
int in_argc;
char *In_argv[10];
FILE *sysin , *sysout;
# define YYNEWLINE 10

yylex(){
int nstr; extern int yyprevious;
```

Part (a)

```
while((nstr = yylook()) >= 0)
yyfussy: switch(nstr)-
case 0:
if(yywrap()) return(0); break;
case 1:
{
        strcpy(yylval.cval,"ALL");
        return(ALL);
    }
break;
case 2:
{
        strcpy(yylval.cval,"AND");
        return(AND);
    }
break;
case 3:
{
        strcpy(yylval.cval,"ANY");
        return(ANY);
    }
break;
case 4:
{
        strcpy(yylval.cval,"ARRAY");
        return(ARRAY);
    }
break;
case 5:
{
        strcpy(yylval.cval,"BEGIN");
       return(BEGINN);
    }
break;
case 6:
{
        strcpy(yylval.cval,"BY");
       return(BY);
    }
break;
case 7:
{
        strcpy(yylval.cval,"CASE");
       return(CASE);
    }
break;
```

Part (b)

```
case 8:
{
         strcpy(yylval.cval,"CHANNEL");
        return(CHANNEL);
    }
break;
case 9:
{
         strcpy(yylval.cval,"COMMON");
        return(COMMON);
    }
break;
case 10:
{
         strcpy(yylval.cval,"CONST");
        return(CONST);
    }
break;
case 11:
{
         strcpy(yylval.cval,"DELAY");
        return(DELAY);
    }
break;
    :
    :
    :
    :
    :
break;
case 87:
{
         strcpy(yylval.cval,yytext);
    }
break;


case 88:
{
         stat_line ++;
    }
break;
case -1:
break;
default{
fprintf(yyout,"bad switch yylook %d",nstr);
} return(0); }
/* end of yylex */
    :
    :
    :
    :
    :
```

Appendix F

PARTIAL PREPROCESSOR LISTING

The preprocessor listing is a combination of the lexical analyzer, YYLEX(); the parser, YYPARSE(); and other miscellaneous subprograms, such as the main program of the preprocessor etc. Below, is a description of the format of the preprocessor.

Table F.1: ESTELLE Preprocessor layout.

```
Constant Declarations
Type Declarations
Variable Declarations

Programs

   MAIN ()      /* Main program */
        .
        .
        .
        .

   yylex()     /* the Lexical analyzer */
        .
        .
        .
        .

YYPARSE ()
        .
        .
        .
        .

OTHER UTILITY FUNCTIONS
        .
        .
        .
        .
```

Appendix G

TRANSLATION STRATEGY

The automated translation process generates for each data type defined in the specification, a C language equivalent. A sample of a generated source code in C language for a class 0 protocol is listed in Appendix H. Because of the numerous choices of data structure constructs in ESTELLE compared to the handful available in C language, the reader will notice that some of the different data types in the specification have been translated into similar structures in C language.

C has three basic data types; integer, character and reals, while ESTELLE supports four basic data types: integer, character, reals and booleans. However, both languages provide facilities for user defined data types, though not to the same degree.

The use of some data types in ESTELLE are very difficult to support in C because of the strong typing which ESTELLE supports. In the current implementation of the ESTELLE preprocessor, it is the responsibility of the protocol implementor to ensure that the data types are properly referenced in the formal specification.

The following sections show how ESTELLE code are compiled into executable C code.

G.1 IMPLEMENTING DECLARATIONS

The declaration block of ESTELLE consists of constant, type, variable, array, record, and subroutine declarations. In this section, we will show how these declarations are encoded in C by the ESTELLE translations.

G.1.1 Symbolic Constants

Symbolic constants in ESTELLE are specified by the syntax:

```
CONST name1 = value;
```

Where the value could be an integer, real, a value from an enumerated scaler type in string constant, but not an expression. (See Table G.1)

Table G.1: A Sample Of Constant declaration in ESTELLE.

```
CONST

     MAXLEN      = 100;
     PDUNAME     = 'TPDU';
     QUIT_COUNT  =  250.0;
```

In C, constants are implemented by string substitution using the C preprocessor instruction #define.

Thus, given an ESTELLE program with the constant block statement defined as shown in Table , the ESTELLE compiler will generate the code in Table G.2.

Table G.2: A Sample Of Constant declaration in C.

```
CONST

     # define ‹constant identifier› ‹constant value›

E.G.:
     # define MAXLEN 100
     # define PDUNAME 'TPDU'
     # define QUIT_COUNT 250.0
```

From the output, CONST is an extension to C. The other statements request that the identifier be replaced by the string token, wherever the identifier appears. (Note, no semicolon terminates the definitions since they are not C statements.)

G.1.2 Data Types

In ESTELLE, type specification has the format:

```
type    name   =  data type;
```

In C, the format is:

```
typedef data_type name1;
```

G.1.2.1 Basic Types

A type block in ESTELLE, such as,

```
Type  quick  =   real;
      index  =   integer;
      flag   =   char;
      signal =   boolean;
```

When translated to C by the compiler gives:

```
TYPE
      typedef real    quick;
      typedef integer index;
      typedef char    flag;
      typedef boolean signal;
```

Note, the predefined types of real, integer and boolean are extensions to C. They are included from the library and are defined as:

```
typedef int     integer;
typedef float   real;
typedef enum    {false, true}  boolean;
```

G.1.2.2 Enumerated Type

Conversion of the data type:

```
type
    day = (sun,mon,tues,wed,thurs,friday,saturday);
```

into C equivalent is as shown below,

```
typedef Enum { sun,mon,tues,wed,thurs,friday,saturday} day;
```

G.1.2.3 Subrange Type

To convert a subrange type, there are two methods: one is to enumerate the elements of the subrange, and the other is to denote subrange as a structure type.

```
For case one:
    type
        index = 1..10 is translated to C as:
        typedef Enum { 1,2,3,4,5,6,7,8,9,10} index;
```

The advantage of enumerating is the ability to use the ENUM Construct, and the C language compiler is responsible for type checking.

In the other case,

```
type
    index = 1..10 ;
```

is translated to the basic structure type in C language as follows:

```
typedef  SUBRANGE  index;
```

where SUBRANGE is an extension of C language.

G.1.2.4 Records

Records in ESTELLE are used for structured data types. C has an equivalent structure called STRUCT which are used for declaration of record variables. For example, the record:

```
tpdu_type = RECORD
              li     :  int_type;
              name   :  pdu_name;
              cdt    :  seq_type;
                :
                :
                :
              user_data:u_data_type
            END;
```

is equivalent to the following C language declaration:

```
typedef struct  {
        int_type            li;
        pdu_name            name;
        seq_type            cdt;
          :
          :
          :
        u_data_type         user_data
            }  tpdu_type.
```

During translation, the preprocessor will generate the following statements for the record structure above:

```
typedef record
          int_type     li;
          pdu_name     name;
          seq_type     cdt;
            :
            :
            :
          u_data_type user_data;
    end    tpdu_type;
```

The symbols "RECORD" and "END" are predefined as "STRUCT { " and "}" respectively.

C.1.2.5 Variant Record

Variant records in ESTELLE are implemented as UNION structures in C language with some minor constraints. The preprocessor requires that the field list of a variant part contain only a single variable. This variable may be of any type (i.e. simple or structured type). Consider the declaration below:

```
type shape = (point, line, circle);
     figure= record
             case tag : shape of
                 point:
                       (position : coordinate);
                 line:
                       (xcoeff, ycoeff, con : real);
                 circle:
                       (center : coordinate;
                        radius : real)
                 end;
```

This is a correct declaration of a record structure in ESTELLE, but its not acceptable to the preprocessor. The preprocessor will generate an error statement because if the number of fields declared for the case when the record contains a line or circle.

All the user has to do is to declare one variable per case label above. This may mean, that new record types have to be defined which will consist of all the fields in a label case. The above can be re-written as the following:

```
type coordinate = record
                  xcoor, ycoor:real
                  end;
     shape      = (point, line, circle);
     line_rec   = record
                  xcoeff, ycoeff, con:real
                  end;
     circle_rec= record
                  center:coordinate
                  radius:real
                  end;
     figure     = record case tag:shape of
                  point:
                       (position:coordinate);
                  line:
                       (line_str:line_rec);
                  circle:
                       (circle_str:circle_rec);
                  end;
```

This restriction, though an inconvenience, is still acceptable, and is as a result of the syntax of UNION which is used to implement the variant records.

So given the re-written declarations, the preprocessor will generate the following statements:

```
TYPE
    typedef record
            real, xcoor, ycorr;
            end coordinate;
    typedef enum begin
            point,
            line,
            circle
            and shape;
    typedef record
            real, xcoeff, ycoeff, con;
            end line_rec;
    typedef record
            coordinate center;
            real       radius;
            end circle_rec;
    typedef record
            shape  tage;
            union
                begin
                    coordinate   position;
                    line_rec     line_str;
                    circle_rec   circle_str;
                end;
            end
```

G.1.2.6 Procedures and Functions

Since C language only supports functions, all procedures are implemented as functions. A typical translation of a procedure and function declaration is shown below:

```
Procedure  transmit (index integer) ;
begin
   :
   :
   :
end;

Function xsend (index integer; num : retrials):bool;
var i : int;
begin
   :
   :
   return(true);
   :
   :
   return (false);
end;
```

are translated into the C language equivalent as:

```
PROCEDURE transmit (index)
Integer   index;
begin
   :
   :
   :
end;

FUNCTION  bool xsend (index)
integer  index;
retrials num;
Var integer i;
begin
   :
   :
   return(true);
   :
   :
   return(false);
end;
```

Appendix H

PARTIAL LISTING OF THE PROTOTYPE OF THE TRANSPORT PROTOCOL

This appendix contains the listing of the source code in C language generated for the formal protocol specification given in Appendix A. The listing contains sections which correspond to the sections of the specification. The different constants, types, variables, etc. have been generated as labelled below.

Each section of the generated source code corresponds to a similar section in the formal protocol specification. More details on the implementation of the data types could be found in a soon to be published paper. For instance, the preprocessor generates extra code for the channel types which are produced in the file named "CHANNEL.IMP". The content of this file replaces the

```
#include  CHANNEL.IMP
```

line during the compilation of the prototype source code. Included in this file are the channel events that are associated with the given protocol.

```
CONST

#define NULL 0
#define null NULL
#define maxseq 8

TYPE

typedef integer   seq_type;
#define www_SET (1 << 1)
#define rrr_SET (1 << 2)
#define tttt_SET (1 << 3)
typedef set_type   sawww ;
```

```
typedef integer   int_type ;
typedef integer   ref_type ;
typedef integer   N_addr_type ;
typedef integer   T_addr_type ;
typedef char   u_data_type [UDATA_SIZE][34];
typedef  enum begin
                        RESET,INVALID_TPDU,UNEXPECTED_CC,
                        UNACCEPTABLE_REQUEST,
                        PROTOCOL_ERROR,PROVIDER_INITIATED,
                        USER_INITIATED,
                        NEGOTIATION_FAILURE
              end reason;
typedef  enum begin
                      NO_NAME,AK,CR,CC,DR,DT,EA,ER
                    end  pdu_name ;
typedef  enum begin
                      CLASS_0
                     end  class_type ;
typedef  enum begin BASIC,EXTENDED end    frame_type ;
typedef  enum begin not_TSDU,TSDU end    mark_type ;
 /* here are the events */
typedef  enum begin
                      T_CONNECT_request,T_CONNECT_response,
                      T_DATA_request,T_DISCONNECT_request
                      T_CONNECT_indication,
                      T_CONNECT_confirm,T_DATA_INDICATION
            end event_type;
# include  channel.module

VAR

     N_addr_type  calling_NSAP_id,
                        called_NSAP_id, R_net_addr, L_net_addr;
     T_addr_type  responding_TSAP_id,
                        calling_TSAP_id, called_TSAP_id, R_addr,
                        L_addr;
     ref_type     R_user_id, L_user_id, dst_ref, src_ref;
     seq_type     recv_seq, tpdu_nr, cdt,
                        send_next_seq, send_seq;
     reason_type  more_info, user_reason, reason;
     int_type     header_size, max_data_size,
                        udata_size, TPDU_size,
                        max_TPDU_size;

     boolean      EOT, REQ_recved, CR_recved, CC_recved,
                        REQ_sent, CR_sent,CC_sent, owner;
     frame_type  option, quality_of_service;
     class_type  class;
     tpdu_buffer  recv_tpdu_buff, trans_tpdu_buff;
     u_data_type  user_data;

PROCEDURE  transmit(seq)
seq_type  seq;
BEGIN
```

```
        DATA = retrieve(Tx_buffer,seq);
        N_DATA_request(DATA);
    END

    INITIALIZE()
    BEGIN
        STATE TO CLOSED;
        L_addr = addr_undefined;
        R_addr = addr_undefined;
        owner  = false;
    END
    U_ACCESS_POINT  U;
    N_ACCESS_POINT  N; /* interaction point definition */
    MODULE tr(CEP_id)
    interact_end_type CEP_id;
    BEGIN

    /*****************************/
    /**                         **/
    /**                         **/
    /**    TRANSITION  PART     **/
    /**                         **/
    /**                         **/
    /*****************************/
    SWITCH ( context[CEP_id].state )
    BEGIN
        FROM CLOSED :
        BEGIN
            SWITCH  ([context[CEP_id].event )
            BEGIN
                WHEN N_DATA_indication :
                BEGIN
                    BEGIN  /* beginning of transition */
                        PROVIDED ( ((N.in_TPDU.name == AK)
                                    OR (N.in_TPDU.name == EA)
                                    OR (N.in_TPDU.name == ER)
                                    OR (N.in_TPDU.name == DC)
                        BEGIN
                            context[CEP_id].STATE = SAME;
                            xxxx = 12;
                            yyenter = YYES;
                        END
                    END   /* end of transition */
                    if (yyenter == YYES)
                    BREAK;
                    BEGIN  /* beginning of transition */
                        PROVIDED ( (N.in_TPDU.name == CC) )
                        BEGIN
                            context[CEP_id].STATE = SAME;
                            dst_ref = N.in_TPDU.src_ref;
                            src_ref = 0;
                            reason = UNEXPECTED_CC;
                            out_TPDU =
                                pduf(DR,null,dst_ref,src_ref,
```

```
                                    null,null,null,null,null,null,
                                    null,null,reason,null );
                            DATA_request(TPDU);
                            yyenter = YYES;
                        END
                    END   /* end of transition */
                    if (yyenter == YYES)
                        BREAK;
                END  /* end of EVENT */
                BREAK;
END  /* end of STATE */
BREAK;
FROM WFCC :
BEGIN
    SWITCH  ([context[CEP_id].event )
    BEGIN
        WHEN N_DISC_indication :
        BEGIN
            BEGIN  /* beginning of transition */
                BEGIN
                    context[CEP_id].STATE = CLOSED;
                    reason = NETWORK_DISCONNECTION;
                    T_DISC_indication(reason);
                    yyenter = YYES;
                END
            END   /* end of transition */
            if (yyenter == YYES)
            BREAK;
        END  /* end of EVENT */
        BREAK;
        WHEN N_DATA_indication :
        BEGIN
            BEGIN  /* beginning of transition */
                PROVIDED ( ((N.in_TPDU.name == CC) AND
                            (unacceptable_CC(N.in_TPDU))) )
                BEGIN
                    context[CEP_id].STATE = CLOSED;
                    dst_ref = N.in_TPDU.src_ref;
                    reason = NEGOTIATION_FAILED;
                    T_DISC_indication(reason);
                    N_DISC_request();
                    yyenter = YYES;
                END
            END   /* end of transition */
            if (yyenter == YYES)
            BREAK;
        END  /* end of EVENT */
        BREAK;
END  /* end of STATE */
BREAK;
FROM WFTRESP :
BEGIN
    SWITCH  ([context[CEP_id].event )
    BEGIN
```

```
        WHEN T_CON_response :
        BEGIN
            BEGIN  /* beginning of transition */
                BEGIN
                    context[CEP_id].STATE = OPEN;
                    out_TPDU =
                            pduf(CC,cdt,dst_ref,src_ref,
                              class,option,calling_TSAP_id,
                              called_TSAP_id,TPDU_size,null,
                              null,null,null);
                    N_DATA_request(out_TPDU);
                    yyenter = YYES;
                END
            END   /* end of transition */
            if (yyenter == YYES)
            BREAK;
        END  /* end of EVENT */
        BREAK;
        WHEN T_DISC_request :
        BEGIN
            BEGIN  /* beginning of transition */
                BEGIN
                    context[CEP_id].STATE = CLOSED;
                    reason = USER_INITIATED;
                    more_info = U.disc_reason;
                    out_TPDU =
                          pduf(DR,null,dst_ref,src_ref,null,
                                null,null,null,null,null,null,
                                more_info,reason,null);
                    N_DATA_request(out_TPDU);
                    yyenter = YYES;
                END
            END   /* end of transition */
            if (yyenter == YYES)
                BREAK;
        END  /* end of EVENT */
        BREAK;
END  /* end of STATE */
BREAK;
FROM OPEN :
BEGIN
    SWITCH  ([context[CEP_id].event )
    BEGIN
        WHEN N_DATA_indication :
        BEGIN
            BEGIN  /* beginning of transition */
                PROVIDED ( ((N.in_TPDU.name == DT) AND (NOT
                                invalid_tpdu(N.in_TPDU))) )
                BEGIN
                    context[CEP_id].STATE = SAME;
                    recv_seq = recv_seq + 1;
                    IF ((N.in_TPDU.EOT)) THEN
                        IF (TSDU_in(recv_tpdu_buff)) THEN
                          BEGIN
```

```
                                        user_data = extract_TSDU(recv_tpdu_buff);
                                        T_DATA_request(user_data);
                                    END
                                yyenter = YYES;
                            END
                        END   /* end of transition */
                        if (yyenter == YYES)
                            BREAK;
    END   /* SWITCH on EVENTS */
END   /* SWITCH on STATES */
END     /*   tr.module  */
```

REFERENCES

AGGA83a S. Aggarwal, R. P. Kurshan, and K. K. Sabnani, *A CALCULUS FOR PROTOCOL SPECIFICATION AND VALIDATION,* Protocol Specification Testing and Verification, III, (H. Rudin and C. H. West eds.), North-Holland, 1983. pp. 19-34.

AGGA83b S. Aggarwal, R. P. Kurshan, and D. K. Sharnia, *A LANGUAGE FOR THE SPECIFICATION AND ANALYSIS OF PROTOCOLS,* Protocol Specification Testing and Verification, III, (H. Rudin and C. H. West eds.), North-Holland, 1983. pp.35-50

AGGA84 S. Aggarwal, and R. P. Kurshan, *AUTOMATED IMPLEMENTATION FROM FORMAL SPECIFICATION,* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.127-136.

ANDE84 D. P. Anderson and L. H. Landweber, *PROTOCOL SPECIFICATION BY REAL-TIME ATTRIBUTE GRAMMARS (EXTENDED ABSTRACT)* University of Wisconsin - Madison, April 13, 1984.

ANSA82a J. P. Ansart, O. Rafiq, V. Chari, *PDIL - PROTOCOL DESCRIPTION AND IMPLEMENTATION LANGUAGE,* Proc. IFIP WG.G.1 2nd Int. Workshop on Protocol Specification, Testing and Verification - Idylwild-California, May 1982, pp.101-112.

ANSA82c J. P. Ansart, *A PROTOCOL INDEPENDENT SYSTEM FOR TESTING PROTOCOL IMPLEMENTATION,* Protocol Specification, Testing and Verification, C. Sunshine(ed.), North-Holland Publishing Company, IFIP, 1982.

ANSA83 J. P. Ansart, V. Chari, D. Simon, *FROM FORMAL DESCRIPTION TO AUTOMATED IMPLEMENTATION USING PDIL (PROTOCOL DESCRIPTION AND IMPLEMENTATION LANGUAGE),* Protocol Specification, Testing and Verification, III H. Rudin and C. H. West(eds.), IFIP, 1983. pp.381-390.

AYAC81 J. M. Ayache, J. P. Courtiat, *LC/1, A SPECIFICATION AND IMPLEMENTATION LANGUAGE FOR PROTOCOLS,* Protocol Specification, Testing and Verification, III H. Rudin and C. H. West(eds.), IFIP, 1983. pp.333-346.

AZEMA P. Azema, G. Juandle, E. Sanchis, M. Montbernard, *SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS USING PROLOG INTERPRETED PETRI NETS.* Laboratoire dAutomatique et dAnalyse des Systemes du CNRS., 7 Avenue Colonel Roche, 31400 Toulouse, France.

BELL83a Bell Laboratories, *VOLUME 1: UNIX PROGRAMMERS MANUAL REVISED AND EXPANDED VERSION,* Bell Laboratories, 1983.

BELL83b Bell Laboratories, *VOLUME 2: UNIX PROGRAMMERS MANUAL REVISED AND EXPANDED VERSION,* Bell Laboratories, 1983.

BILL85 J. Billington, M. C. Wilbur-Ham, M. Y. Bearman, *AUTOMATED PROTOCOL VERIFICATION,* Protocol Specification Testing and Verification, V_r (M. Diaz eds.), North-Holland, 1985. pp.59-70.

BLUM82 T. P. Blumer, R. L Tenney, *A FORMAL SPECIFICATION TECHNIQUE AND IMPLEMENTATION METHOD FOR PROTOCOLS,* Computer Networks 6.3, July 1982. pp. 201-217.

BLUM83 T. P. Blumer and D. P. Sidhu, *EXPERIENCE WITH AN AUTOMATED PROTOCOL DEVELOPMENT SYSTEM,* Protocol Specification, Testing and Verification, III H. Rudin and C. H. West(eds.), IFIP, 1983. pp.369-380.

BOCH78 G. V. Bochmann, *FINITE STATE DESCRIPTION OF COMMUNICATION PROTOCOLS,* Computer Networks, Vol. 2, 1978, pp. 361-372.

BOCH80a Gregor V. Bochman, *A GENERAL TRANSITION MODEL FOR PROTOCOL AND COMMUNICATION SERVICES,* IEEE Transaction on Communications, Vol. Com-28, No.4, April 1980, pp. 643-650.

BOCH80b G. V. Bochmann, C. Sunshine, *FORMAL METHODS IN COMMUNICATION PROTOCOL DESIGN,* IEEE Transaction on Communications, vol COM-28, No.4 1980. pp. 624-631.

BOCH82a G. V. Bochmann, *EXAMPLE OF A TRANSPORT PROTOCOL SPECIFICATION,* ISO/TC97/SC16/WG1/FDT CAT-13 Catania, Nov. 1982.

BOCH82b G. V. Bochmann, *EXPERIENCE WITH FORMAL SPECIFICATIONS USING AN EXTENDED STATE TRANSITION MODEL,* IEEE Trans. Commun., Vol. COM-30, pp.2506-2513, Dec. 1982.

BOCH84a G. V. Bochmann, G. Gerber, J. M. Serre, *SEMI-AUTOMATIC IMPLEMENTATION OF COMMUNICATION PROTOCOLS,* Publication #518, Faculty of Arts and Sciences, University of Montreal. 1984.

BOCH84b G. V. Bochman, E. Cerny, G. Gerber, R. Dissouli, M. maksud, B. H. Phan, B. Sarikaya, and J. M. Serre, *USE OF FORMAL SPECIFICATIONS FOR PROTOCOL DESIGN, IMPLEMENTATION AND TESTING,* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.137-144.

BOCH85 G. V. Bochman, R. Dssouli, W. Lopes De Souza, B. Sarikaya, H. Ural, *USE OF PROLOG FOR BUILDING PROTOCOL DESIGN TOOLS,* Protocol Specification Testing and Verification, IV, (M. Diaz eds.), North-Holland, 1985. pp.59-70.

BRIN84 E. Brinksma, and G. Karjoth *A SPECIFICATION OF THE OSI TRANSPORT SERVICE IN LOTOS* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.227-252.

BRIN85 E. Brinksma, *A TUTORIAL ON LOTOS,* Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.171-194.

BROW85 D. L. Brown, *FROM PASCAL TO C : AN INTRODUCTION TO THE C PROGRAMMING LANGUAGE,* Wadsworth Publishing Company, Belmont California, 1985.

BURK84 H. J. Burkhardt, H. Eckert, and R. Prinoth *MODELLING OF OSI-COMMUNICATION SERVICES AND PROTOCOLS USING PREDICATE/TRANSITION NETS,* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.165-192.

CCITTa CCITT, *FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE, SDL,* CCITT Working Party X-1/3-1, Temp. Doc. 35E, 1976.

CCITTb CCITT *DRAFT RECOMMENDATIONS Z.200: PROPOSAL FOR RECOMMENDATION FOR A CCITT HIGH LEVEL PROGRAMMING LANGUAGE, (CHILL),* Com XI-no.379, Feb. 1980.

CHUN84 Richard Chung, *A METHODOLOGY FOR PROTOCOL DESIGN AND SPECIFICATION BASED ON AN EXTENDED STATE TRANSITION MODEL,* ACM 1984 pp 34-41.

CLOM81 W. F. Clocksin and C. S. Mellish, *PROGRAMMING IN PROLOG,* Springer-Verlag Berlin, Heidelberg, New York.

COOP83 D. Cooper, *STANDARD PASCAL USER REFERENCE MANUAL,* W. W. Norton and Company, 1983.

COUR85 J. P. Courtiat, A. Pedroza, J. M. Ayache, *A SIMULATION ENVIRONMENT FOR PROTOCOL SPECIFICATIONS DESCRIBED IN ESTELLE,* Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.297-312.

DANT80 Andre' A. S. Danthine, *PROTOCOL REPRESENTATION WITH FINITE-STATE MODELS,* IEEE Transaction on Communications, Vol. Com-28, No.4, April 1980, pp. 632-643.

DESJ81 Richard DesJardins, *OVERVIEW AND STATUS OF THE ISO REFERENCE MODEL OF OPEN SYSTEMS INTERCONNECTION,* Computer Networks 5 (1981) pp 77-80.

DIAZ83 M. Diaz, G. G. Da Silveira, *SPECIFICATION AND VALIDATION OF PROTOCOLS BY TEMPORAL LOGIC AND NETS,* Proc. IFIP Congress Paris, Sept. 1983.

DUNN84 Robert H. Dunn, *SOFTWARE DEFECT REMOVAL* McGraw-Hill Book Company, New York, 1984.

EXEL82 Matija Exel, Branislav T. Popvic, Francel Prijatelj, *SL1 LANGUAGE - A SPECIFICATION AND DESIGN TOOL FOR SWITCHING SYSTEMS SOFTWARE DEVELOPMENT,* IEEE Transactions on Communications, Vol. Com-30, No.6, June 1982. pp. 1356-1362.

FEUG84 A. Feuer and N. Gehani, *COMPARING AND ASSESSING PROGRAMING LANGUAGES: ADA, C AND PASCAL*, Prentice-Hall (1984).

FREN85 Karen A. Frenkel, *TOWARD AUTOMATING THE SOFTWARE-DEVELOPMENT CYCLE*, Communications of the ACM, Vol. 28, No. 6, June 1985.

GERB83 G. W. Gerber, *UNE METHOD DIMPLANATATION AUTOMATISEE DE SYSTEMS SPECIFIES FORMELLEMENT*, Thesis Report, Department of Computer Science and Operational Research, University of Montreal, Montreal Quebec, Canada. August 1983.

GROG80 P. Grogono, *PROGRAMMING IN PASCAL REVISED EDITION*, Addison-Wesley Publishing Company, Inc.

HAAS85 O. Haas, *FORMAL PROTOCOL SPECIFICATION BASED ON ATTRIBUTE GRAMMARS*, Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.39-48.

HABE84 A. Nico Habermann, Dewayne E. Perry, *ADA FOR EXPERIENCED PROGRAMMERS* Addison Wesley Publishing Company 1982.

HANS85 H. A. Hansson, *AUTOMATIC IMPLEMENTATION OF FORMAL DESCRIPTIONS OF COMMUNICATION PROTOCOLS*, Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.259-270.

HUNT81 R. Hunter, *THE DESIGN AND CONSTRUCTION OF COMPILERS*, Wiley series in computing.

IBM74 IBM Corp., *GUIDE TO PL/S II*, IBM Form No. GC28-6794-0, 1974.

IBM79 IBM Corp., *PROGRAMMING LANGUAGE FOR DISTRIBUTED SYSTEMS REFERENCE (PL/DS)*, IBM Form No. SC27-0446-0, 1979.

IBM80a IBM, *SYSTEMS NETWORK ARCHITECTURE FORMAT AND PROTOCOL REFERENCE MANUAL: ARCHITECTURE LOGIC*, IBM Form No. SC30-3112-2, 1980.

IBM80b IBM, *SIMPL/I (SIMULATION LANGUAGE BASED ON PL/I) ARCHITECTURE LOGIC*, IBM Form No. SC30-3112-2, 1980.

ISO81 ISO/TC97/SC16, *DATA PROCESSING OPEN SYSTEMS INTERCONNECTION BASIC REFERENCE MODEL*, American National Standards Institute, 1430 Broadway, New York, NY 10036, USA. Computer Networks 5 (1981) pp 81-118.

ISO82a ISO/TC97/SC16/WG1/FDT subgroup A, *CONCEPTS FOR DESCRIBING THE OS ARCHITECTURE*, Working document, July 1982.

ISO82b ISO/TC97/SC16/WG1/FDT subgroup B, *A FDT BASED ON AN EXTENDED STATE TRANSITION MODEL*, Working document, July 1982.

ISO84 ISO TC 97/SC16/WG1 - FDT, Subgroup B (Q 15), *A FORMAL DESCRIPTION TECHNIQUE BASED ON EXTENDED STATE TRANSITION MODEL.* International Organization for Standardization, ISO/TC 97/SC 16, Open Systems Interconnection, March 1984.

ISO85 ISO TC 97/SC 21/N 422 - FDT, Subgroup B (DP 9074), *A FORMAL DESCRIPTION TECHNIQUE BASED ON EXTENDED STATE TRANSITION MODEL.* International Organization for Standardization, ISO/TC 97/SC 21, Open Systems Interconnection, June 1985.

JARD85 C. Jard, J. F. Monin, R. Groz, *EXPERIENCE IN IMPLEMENTING ESTELLE-X.250 (A CCITT SUBSET OF ESTELLE) IN VEDA,* Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.315-332.

JOHN75 S. C. Johnson, *YACC- YET ANOTHER COMPILER COMPILER,* volume 2: unix programmers manual revised and expanded version, Bell Laboratories, 1983, pp.353-387.

KERN78 B. W. Kernighan and D. M. Ritchie, *THE C PROGRAMMING LANGUAGE,* Prentice-Hall Software Series (1978).

KERN84 B. W. Kernighan and R. Pike, *THE UNIX PROGRAMMING ENVIRONMENT,* Prentice-Hall Software Series (1984).

LESM75 M. E. Lesk, *LEX - A LEXICAL ANALYZER GENERATOR* volume 2: unix programmers manual revised and expanded version, Bell Laboratories, 1983, pp.388-400.

LINN83a R. J. Linn, W. H. McCoy, *PRODUCING TESTS FOR IMPLEMENTATIONS OF OSI PROTOCOLS,* Protocol Specification Testing and Verification, III, (H. Rudin and C. H. West eds.), North-Holland, 1983. pp. 505-520.

LINN83b R. J. Linn, J. S. Nightingale, *SOME EXPERIENCE WITH TESTING TOOLS FOR OSI PROTOCOL IMPLEMENTATIONS,* Protocol Specification Testing and Verification, III, (H. Rudin and C. H. West eds.), North-Holland, 1983. pp. 521-531.

LINN85 R. J. Linn, *THE FEATURES AND FACILITIES OF ESTELLE : A FORMAL DESCRIPTION TECHNIQUE BASED UPON AN EXTENDED FINITE STATE MACHINE MODEL,* Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.271-296.

LOGR84 L. Logrippo, D. Simon and H. Ural, *EXECUTABLE DESCRIPTION OF THE OSI TRANSPORT SERVICE IN PROLOG,* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.279-294.

McGM83 H. McGilton and R. Morgan, *INTRODUCING THE UNIX SYSTEM,* McGraw-Hill Software series for computer for computer professionals, 1983.

MERL79 P. M. Merlin, *SPECIFICATION AND VALIDATION OF PROTOCOLS,* IEEE Trans. Commun., Nov. 1979. pp. 1671-1680.

MYER79 G. J. Myers, *THE ART OF SOFTWARE TESTING*, Wiley-Interscience.

NASH83 S. C. NASH, *AUTOMATED IMPLEMENTATION OF SNA COMMUNICATION PROTOCOLS.* IEEE Transaction on communications, pp. 1316-1322, 1983.

NBS81a NBS, *SPECIFICATION OF THE TRANSPORT PROTOCOL, VOLUME 1: OVERVIEW AND SERVICES*, Report ICST/HLNP-81-11, National Bureau of Standards, Washington, D.C., 1981.

NBS81b NBS, *SPECIFICATION OF THE TRANSPORT PROTOCOL, VOLUME 2: BASIC CLASS PROTOCOLS*, Report ICST/HLNP-81-12, National Bureau of Standards, Washington, D.C., 1981.

NBS81c NBS, *SPECIFICATION OF THE TRANSPORT PROTOCOL, VOLUME 3: EXTENDED CLASS PROTOCOL*, Report ICST/HLNP-81-13, National Bureau of Standards, Washington, D.C., 1981.

NBS81d NBS, *SPECIFICATION OF THE TRANSPORT PROTOCOL, VOLUME 4: NETWORK INTERFACES*, Report ICST/HLNP-81-14, National Bureau of Standards, Washington, D.C., 1981.

NBS81e NBS, *A FORMAL SPECIFICATION TECHNIQUE AND IMPLEMENTATION METHOD FOR PROTOCOLS*, Report ICST/HLNP-81-15, National Bureau of Standards, Washington, D.C., 1981.

PAVE84 J. R. Pavel, and D. J. Dwyer, *SOME EXPERIENCES TESTING PROTOCOL IMPLEMENTATIONS*, Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.657-655.

PEpp83 A. S. PEppLE, *LOGICAL DESIGN AND TESTING OF OSI SOFTWARE FOR UNIX.* A Thesis Report: Dept. of Systems and Computer engineering, Faculty of Engineering , Carleton University, Ottawa, Ontario, December 1983.

POSM82 D. P. POZEFSKY, F. D. SMITH, *A META-IMPLEMENTATION FOR SYSTEMS NETWORK ARCHITECTURE.* IEEE Transaction on communications, Vol. COM-30, No. 6, pp. 1348-1355, June 1982.

PROB83 R. L. Probert, H. Ural, *REQUIREMENTS FOR A TEST SPECIFICATION LANGUAGE FOR PROTOCOL IMPLEMENTATION TESTING.* Protocol Specification, Testing and Verification, III H. Rudin and C. H. West(eds.), IFIP, 1983. pp.437-444.

RAZO80 R. R. Razouk, G. Estrin, *MODELLING AND VERIFICATION OF COMMUNICATION PROTOCOLS IN SARA: THE X.21 INTERFACE*, IEEE Trans. on Computers Vol. C-29, No.12, Dec. 1980.

ROCK82 Anders Rockstrom, Roberto Saracco, *SDL - CCITT SPECIFICATION AND DESCRIPTION LANGUAGE*, IEEE Transaction on Communications, Vol. Com-30, No.6, June 1982, pp. 1310-1318.

SCHI79 S. Schindler and H. Marxen, *THE OSA PROJECT : AUTOMATIC TRANSLATION OF RPSL SPECIFICATIONS*, TU Berlin Fachbereich Informatik (20), Bericht, 79-19, September 1979.

SCHW81 R. L. Schwartz, P. M. Melliar-Smith, *TEMPORAL LOGIC SPECIFICATION OF DISTRIBUTED SYSTEMS*, Proc. 2nd Int. Conf. on Distributed Computing Systems, Paris, April 1981.

SCHZ80 G. D. Schultz, D. B. Rose, C. H. West and J. P. Gray, *EXECUTABLE DESCRIPTION AND VALIDATION OF SNA*, IEEE Trans. Commun., vol. com-28, pp. 661-677, April 1980.

SHOR79 J. Short, T. C. Wilson, *SECOND EDITION PROBLEM SOLVING AND THE COMPUTER: A STRUCTURED CONCEPT WITH PL/1 (PL/C)*, Addison Wesley Publishing Company, 1979.

SUNC75 C. Sunshine, *INTERPROCESS COMMUNICATION PROTOCOLS FOR COMPUTER NETWORKS*, Tech. Report No. 105, Digital Systems Lab., Stanford University, 1975.

SUNS82a C. A. Sunshine, D. A. Smalltarg, *AUTOMATED PROTOCOL VERIFICATION*, ISI/USC RR-83-110, Oct. 1982.

SUNS82b C. A. Sunshine, D. H. Thompson, R. W. Erikson, S. L. Gerhart, D. Swabe, *SPECIFICATION AND VERIFICATION OF COMMUNICATION PROTOCOLS IN AFFIRM USING STATE TRANSITION MODELS*, IEEE Trans. Software Engineering, Vol. SE-8, No.5, pp. 460-489, Sept. 1982.

TANE81 A. S. Tanenbaum, *COMPUTER NETWORKS*, Prentice-Hall (1981).

TENG79 A. Y. Teng, T. L. Ming, *A FORMAL MODEL FOR AUTOMATIC IMPLEMENTATION AND VALIDATION OF NETWORK COMMUNICATION PROTOCOL*, Proc. NBS Computer Networking Symposium Gaithersburg, Maryland, Dec. 1978, pp. 114-123.

TEO84 E. H. TEO, *A TRANSPORT DESIGN FOR LOCAL AREA NETWORKS.* A Thesis Report: Dept. of Electrical Engineering, Faculty of Science and Engineering, University of Ottawa, Ottawa, Ontario, 1984.

URAL83a H. Ural, P. L. Probert, *TECHNICAL REPORT TR-83-13: ARCHITECTURES FOR TESTING PROTOCOL IMPLEMENTATIONS*, Protocols Research group, Computer Science Department, University of Ottawa, Ottawa, Ontario Canada, K1N 9B4.

URAL83b H. Ural, P. L. Probert, *USER-GUIDED TEST SEQUENCE GENERATION*, Protocol Specification, Testing and Verification, III H. Rudin and C. H. West(eds.), IFIP, 1983. pp.421-436.

URAL86 H. Ural, P. L. Probert, *STEP-WISE VALIDATION OF COMMUNICATIONS PROTOCOLS AND SERVICES*, Protocols Research group, Computer Science Department, University of Ottawa, Ottawa, Ontario Canada, K1N 9B4.

VALE84 I. Valet and D. Rerat, *FORMAL DESCRIPTION OF THE NADIR TRANSPORT PROTOCOL USING THE PDIL LANGUAGE,* Protocol Specification Testing and Verification, IV, (Y. Yemini, R. Strom, and S. Yemini eds.), North-Holland, 1984. pp.147-164.

VENK85 R. C. Venkatraman, T. F. Piatkowski, *A FORMAL COMPARISON OF FORMAL PROTOCOL SPECIFICATION TECHNIQUES, ESTELLE,* Protocol Specification Testing and Verification, V, (M. Diaz eds.), North-Holland, 1985. pp.401-420.

WAIT84 M. Waite, S. Prata, D. Martin, *C PRIMER PLUS,* Howard W. Sams, and Inc. 1984.

WEST78 C. H. West, *GENERAL TECHNIQUE FOR COMMUNICATIONS PROTOCOL VALIDATION,* IBM Journal Res. Develop., vol. 22, pp. 393-404,july 1978.

WILE84 Robert Wilensky, *LISPCRAFT,* University of California, Berkley, W. W. Norton & Company, New York, 1984.

WINS84 P. H. Winston, B. K. P. Horn, *LISP (SECOND EDITION),* Addison-Wesley Publishing Company, 1984.

ZIMM80 H. Zimmermann, *OSI REFERENCE MODEL - THE ISO MODEL OF ARCHITECTURE FOR OPEN SYSTEMS INTERCONNECTION,* IEEE Trans. Commun., Vol. COM-28, pp.425-432, April 1980.