A SEMI-AUTOMATIC APPROACH TO PROTOCOL IMPLEMENTATION - THE ISO CLASS 2 TRANSPORT PROTOCOL AS AN EXAMPLE

by

Allen Chakming Lau

Technical Report 86-20

November, 1986

A SEMI-AUTOMATIC APPROACH TO PROTOCOL IMPLEMENTATION - THE ISO CLASS 2 TRANSPORT PROTOCOL AS AN EXAMPLE

By

ALLEN CHAKMING LAU

B.Sc, Simon Fraser University, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

[illegible]

[illegible]

THE UNIVERSITY OF BRITISH COLUMBIA

July 1986

Abstract

Formal Description Techniques (FDTs) for specifying communication protocols, and the adopted FDT standards such as *Estelle* have opened a new door for the possibility of automating the implementation of a complex communication protocol directly from its specification. After a brief overview of *Estelle* FDT, we present the basic ideas and the encountered problems in developing a C-written *Estelle* compiler, which accepts an *Estelle* specification of protocols and produces a protocol implementation in C. The practicality of this tool — the *Estelle* compiler — has been examined via a semi-automatic implementation of the ISO class 2 Transport Protocol using the tool. A manual implementation in C/UNIX 4.2bsd of this protocol is also performed and compared with the semi-automatic implementation. We find the semi-automatic approach to protocol implementation offers several advantages over the conventional manual one. These advantages include correctness and modularity in protocol implementation code and reduction in implementation development time. In this thesis, we discuss our experience on using the semi-automatic approach in implementing the ISO class 2 Transport Protocol.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my supervisor, Dr. Son Vuong, for his guidance throughout the course of this thesis and Dr. Harvey Abramson for his comments and careful reading of the thesis.

Many thanks are due to Susan Chan and Helen See for their helpful comments and their fine editing skills.

Finally, I wish to thank Frances Liu for her patience and love.

Chapter 1

Introduction

1.1 Motivations

Formal Description Techniques (FDTs) [Boch80] for specifying protocols and services have opened a new door for the possibility of automating the implementation of a complex communication protocol directly from its specification. These FDTs are advance enough that they are becoming standards such as [CCITT85], [Estelle85] and [Lotos84] and their compilers, [Ansart83], [Bria86], [Ford85], [Gerber83] and [Hans84], are also being developed to make themselves usable in the design and implementation of real-life protocols.

This new approach to protocol implementation is superior than the traditional approach in that communication protocols are implemented semi-automatically in a systematic manner rather than manually in an ad hoc manner. It avoids different interpretation of the specification and various implementation errors, hence, provides confidence in conformance to the specification. As a large portion of the protocol implementation is generated by the compiler in a standard target language, the implementation is highly portable. Furthermore, the generated code is well-constructed, and system-dependent features can be easily located in a few routines. Thus, the implementation is easier to maintain.

The motivation of this thesis is to verify the usefulness of the semi-automatic approach to protocol implementation. An Estelle compiler is chosen to implement a fairly complex ISO class 2 Transport Protocol [CCITT85,ISO82b]. A manual implementation of this protocol is also performed and compared with the semi-automatic implementation.

1.2 Scope and Contributions

The chosen compiler is developed by Daniel Ford in the language C on a VAX 11/750[1] running UNIX 4.2bsd[2]. The compiler accepts an Estelle specification for communication protocols and produces C code. The generated code is then incorporated with pre-written generic and implementation-dependent routines to implement the specified protocol.

The original C-written Estelle compiler[3] is erroneous and insufficiently tested. Its performance has been greatly enhanced by transforming BNF grammars into LALR grammars which best fit the YACC compiler [John75] for generating the parser of the C-Estelle compiler. The grammar rules were also rewritten so that the compiler supports complex data structures such as variant record and pointer which are commonly used in complex protocol specifications. Furthermore, the translation routines were modified to produce optimized and better-organized code.

The enhanced compiler was examined by using it to implement protocols such as hot potato, alternating bit, and ISO class 2 Transport Protocol. It was also ported to several SUN Workstations[4] and the protocol implementations are successfully running among the VAX 11/750 and SUN Workstations.

[1]VAX is a trademark of Digital Equipment Corporation

[2]UNIX is a trademark of AT&T Bell Laboratories.

[3]For brevity we shall often use the terms C-Estelle compiler in place of C-written Estelle compiler

[4]SUN Workstation is a trademark of Sun Microsystems.

1.3 Thesis Outline

After an overview of Estelle in Chapter 2, the development of the automatic tool, C-Estelle compiler is described. Chapter 3 explains the implementation strategy used in the tool, and Chapter 4 discusses the problems encountered. An extensive application of the tool is described in Chapter 5. The real-life ISO class 2 Transport protocol is implemented both semi-automatically by using the tool and manually in an ad hoc manner. After a presentation of their designs and implementations, experience learned from the implementations is discussed. The last chapter summarizes the thesis and offers suggestions for future work.

Since the implementations of the C-Estelle compiler and the protocol were written in the language C, all coding examples presented are C-like. In addition, implementations run on the UNIX 4.2bsd operating system. Thus, reader are assumed to have a basic understanding of the language C and the UNIX 4.2bsd operating system.

Chapter 2

Estelle

Estelle (*Extended State Transition Language*) is a formal description technique developed by the International Standard Organization (ISO) TC 97/SC 16/WG 1 — FDT, Subgroup B [Estelle85,ISO84]. Based upon an extended finite state transition model and the Pascal programming language, Estelle is used for the specification of communication protocols and services.

The framework of an Estelle specification is a set of co-operating entities, each described as a module, interacting with each other by exchanging information through channels. The actual behaviour of a module is specified as either an integrated behaviour of a set of interacting submodules or at the innermost level, an extended finite state automaton.

2.1 Channel and Interaction Primitive

A **channel** is a two-way simultaneous pipe which transmits information between two connected modules. A **channel-type** definition specifies a set of interaction primitives which is grouped under two different roles. These roles are used to distinguish the two sides of the channel, and hence, the two connected modules. Primitives grouped under one role can only be initiated by the module instance which plays that role in respect to the channel; and they are received by the module instance which plays the other role. Information is transmitted between module instances via the parameters of interaction primitives. As an example, figure 2.1 shows a definition of a channel-type TS_primitives. There are ten possible Transport

```
CHANNEL TS_primitives ( TS_user, TS_provider );
        BY TS_user :
           T_CONNECT_request          ( From_transport_addr : ADDR_TYPE;
                                        To_transport_addr   : ADDR_TYPE;
                                        Qual_of_service     : QOS_TYPE;
                                        TS_user_data        : DATA_TYPE );
           T_CONNECT_response         ( Qual_of_service     : QOS_TYPE;
                                        TS_user_data        : DATA_TYPE );
           T_DATA_request             ( TS_user_data        : DATA_TYPE );
           T_XPD_request              ( TS_user_data        : DATA_TYPE );
           T_DISCONNECT_request       ( TS_user_data        : DATA_TYPE );
        BY TS_provider :
           T_CONNECT_indication       ( From_transport_addr : ADDR_TYPE;
                                        To_transport_addr   : ADDR_TYPE;
                                        Qual_of_service     : QOS_TYPE;
                                        TS_user_data        : DATA_TYPE );
           T_CONNECT_confirm          ( Qual_of_service     : QOS_TYPE;
                                        TS_user_data        : DATA_TYPE );
           T_DATA_indication          ( TS_user_data        : DATA_TYPE );
           T_XPD_indication           ( TS_user_data        : DATA_TYPE );
           T_DISCONNECT_indication    ( Reason              : REASON_TYPE;
                                        TS_user_data        : DATA_TYPE );
END TS_primitives;
```

Figure 2.1: An Example of Channel Specification

service interaction primitives which can be used by a Transport service user to interact with the service provider. Five of them, namely T_CONNECT_request, T_CONNECT_response, T_DATA_request, T_XPD_request and T_DISCONNECT_request, can be initiated by a module instance which plays a role of TS_user in respect to the channel. The parameters of the interaction primitives, such as TS_user_data, carry the given information from a TS_user module instance to a receiving TS_provider module instance.

2.2 Module and Interaction Point

A **module** is the basic component of an Estelle specification and represents an entity of the specification. A **module-type** definition is a list of interaction points at which the module interacts with its environment. Each interaction point, (also called ***port***), is an abstract interface of a module used to interact with the connected modules. For each interaction point, a role of its associated channel-type is specified. An interaction is then identified by the name of the interaction point at which it occurs and the name of the interaction. In addition, the interaction has to be one of the defined interaction primitives in the corresponding channel-type definition.

The actual behaviour of a module is defined as either an integrated behaviour of a set of interacting submodules or an extended finite state automaton. For a given module-type, one or many module instances (i.e. protocol instances) can be obtained. An example of a module specification is given in Figure 2.2. All possible interactions of a Transport service user with a

```
MODULE TS_user_module;
          TSAP : TS_primitives ( TS_user );
END TS_user_module;
```

Figure 2.2: An Example of Module Specification

Transport service provider is then through an interaction point *TSAP*. The interaction point is associated with a TS_primitives channel, and the module plays a role of TS_user. Thus, at this interaction point, the module can initiate the interaction primitives T_CONNECT_request, T_CONNECT_response, T_DATA_request, T_XPD_request and T_DISCONNECT_request. It is also allowed to receive other interaction primitives defined only for the TS_primitives channel.

2.3 Refinement and Process

In Estelle, the actual behaviour of a module is specified either indirectly as a **Refinement** or directly as a **Process**. If a module is not a complete self-contained entity, it is decomposed into a set of co-operating submodules, each of which may be further decomposed. The behaviour of the module is the integrated behaviour of the submodules and hence it is called a refinement. A module can also be specified as a process which describes the corresponding finite state transition model of the module.

An Estelle refinement specification includes definitions of internal channel-types, submodule-types, and specifications of the corresponding processes and refinements. After the definition of the internal structures, module instances are created and connected accordingly. If necessary, interaction points of internal module-types may be replaced by those of their parent module-type.

A typical refinement of a Transport system is depicted in Figure 2.3. According to this refinement, a Transport_system module is refined as a Transport_ref refinement, which is decomposed into two TS_user modules, one ATP module, two RS modules, and four System modules. The corresponding Estelle specification is shown in Figure 2.4. After defining the internal structures, module instances are declared. Module instances are then connected provided that they play the different role of a channel through which they interact with each other. There are no replacement because Transport_system module is a closed system.

An Estelle process definition specifies the queueing discipline associated with each interac-

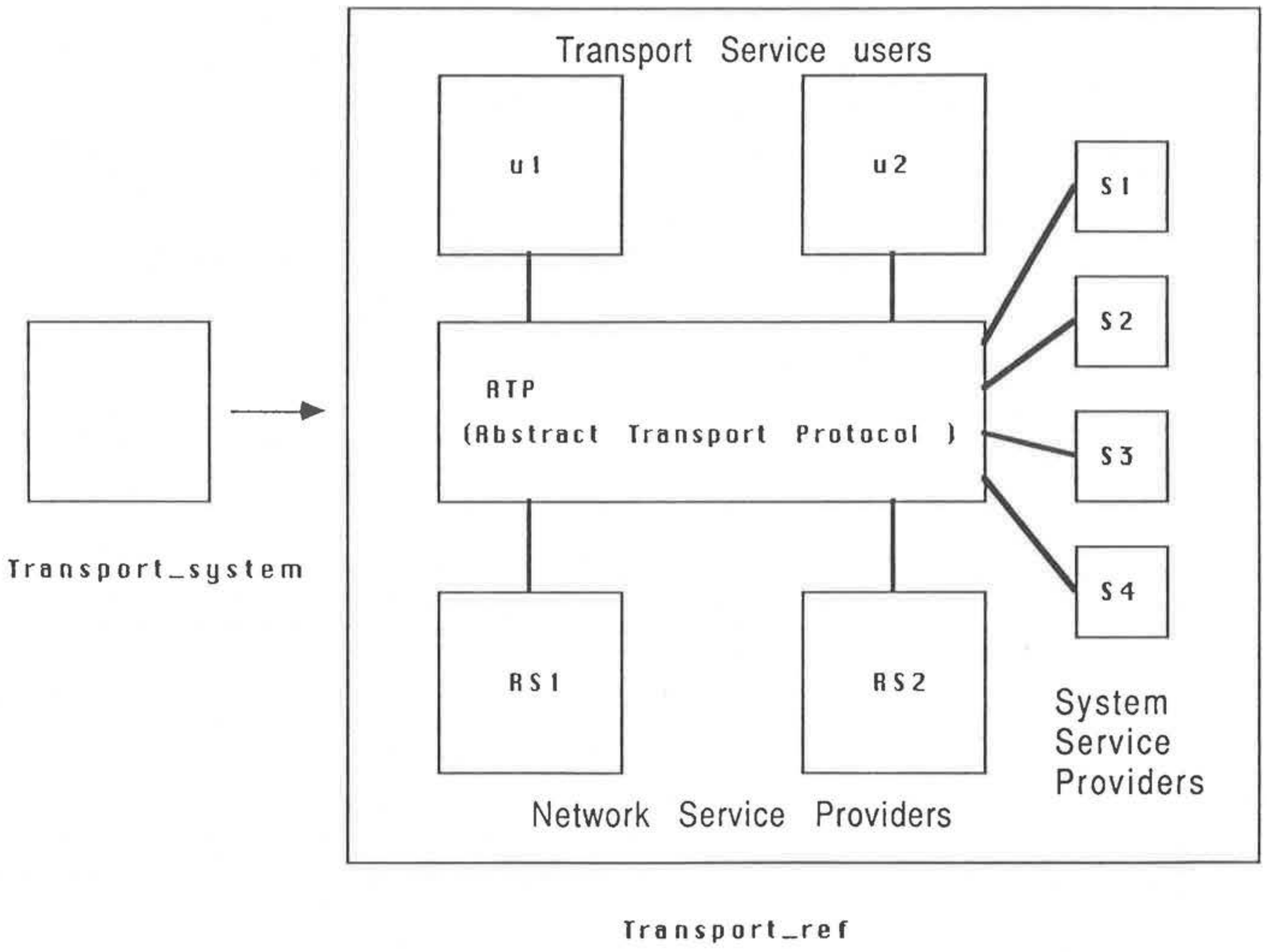


Figure 2.3: Typical Refinement of a Transport System

```
REFINEMENT Transport_ref FOR Transport_system;

(* Constant and Type Definitions *)
.....
(* Channel Definitions *)
.....
(* Module and Process/Refinement Declarations *)
.....
(* Module Instances *)

U1 :  TS_user_module WITH TS_user_process(1);
U2 :  TS_user_module WITH TS_user_process(2);

ATP : ATP_module     WITH ATP_process;

S1 :  System_module  WITH System_process(1);
S2 :  System_module  WITH System_process(2);
S3 :  System_module  WITH System_process(3);
S4 :  System_module  WITH System_process(4);

RS1 : RS_module      WITH RS_process(1);
RS1 : RS_module      WITH RS_process(2);

(* Connection Establishments *)

CONNECT

U1.TSAP      TO ATP.TCEP[1];
U2.TSAP      TO ATP.TCEP[2];

ATP.NSAP[1] TO RS1.NCEP;
ATP.NSAP[2] TO RS2.NCEP;

ATP.SAPT[1] TO S1.SEP;
ATP.SAPT[2] TO S2.SEP;
ATP.SAPN[1] TO S3.SEP;
ATP.SAPN[2] TO S4.SEP;

END Transport_ref;
```

Figure 2.4: An Example of Refinement Specification

tion point, the initial condition and all possible transitions of the corresponding extended finite state machine. For each interaction point of a module, an individual queue is reserved for the queueing of incoming interactions from the peer module before these interactions are considered as input by the module. These queues are on a first-come-first-serve basis and their lengths are either infinite or zero. If the queue length is zero, an output interaction is not queued but consumed immediately as an input by the rendezvous recipient module.

A process specification of a TS_user module is presented in Figure 2.5. The queueing discipline of its interaction point *TSAP*, local variables, primitive functions and procedures are first declared. The local variables are then initialized as the initial state of the corresponding extended finite state machine. The remaining specification is a list of transition definitions.

2.4 Extended Finite State Machine

The operation of a process is modeled as an extended finite state machine which is a finite state automaton extended with the addition of variables to the states, parameters to the interactions, time constraints and priorities to the transitions. The state space of a module is specified by a set of variables. One distinct variable, **state**, if defined, is used to represent the state of a finite state machine upon which the module is based. This major state variable, together with other context variables, determines a state of the module.

The general idea to express a transition, is that WHEN an interaction arrives, a transition has to be performed, FROM the current major state TO a new major state PROVIDED a condition is satisfied, through an action. The associated action of a transition is specified in terms of Pascal statements, and may include the initiation of output interactions with its peer modules.

```
PROCESS TS_user_process ( TS_index : integer ) FOR TS_user_module;

QUEUED TSAP;

(* Type and Variables Declarations *)

.....

(* Primitive function and procedure Declarations *)

.....

INITIALIZE
BEGIN
      user_id := TS_index;
      state := IDLE;

      for qkind := Q_NO_EXPEDITED_DATA to Q_EXTENDED_FORMAT do
         qual_of_service.misc[qkind] := FALSE;

      qual_of_service.class := CLASS_TWO;
      sndcnt := 0; xsndcnt := 0;
      rcvcnt := 0; xrcvcnt := 0;
END;

(* Transition Definitions *)

.....

END TS_user_process;
```

Figure 2.5: An Example of Process Specification

Transitions are classified into **input** and **spontaneous** transitions, depending on the presence of an input interaction (i.e. WHEN clause). An input transition occurs whenever there is an input interaction at a specified interaction point. A spontaneous transition lacks such a WHEN clause and may be executed regardless of any input interactions.

The Estelle state machine is **non-deterministic** in the sense that in a given major state and at a given time, several different transitions may occur. As mentioned in the ISO FDT document, an Estelle specification must not depend on non-deterministic choices. In order to handle the non-deterministic situation, an ANY clause is used to select a random value of the specified enumerated-type variable(s). Such an ANY clause can only be used in spontaneous transitions.

Figure 2.6 lists some transition types, which occur in a TS_user module. Transition one is an input interaction which is initiated by the Transport data arrival. The data arrival causes a cyclic transition from the major state **Alive** to itself, and an execution of procedure **Store_data** to store the data in a buffer pool. Transition two inherits the WHEN clause of transition one. When data arrives and the current major state is **Receiving**, counter *rcvcnt* is incremented and procedure **TS_output** is executed to notify the Transport service user the data arrival. The current major state is also changed into **Alive** as a result of the transition. Transition three is a spontaneous transition that is performed whenever the Transport service user has a request. Whenever the user wants to initiate a Transport connection and the present major state is **Idle**, it first sets up the parameters of the interaction primitive T_CONNECT_request. The request is then sent over the TS_primitives channel at interaction point TSAP and the major state of the module is changed to **Waiting**.

```
TRANS
    WHEN TSAP.T_DATA_indication
        FROM Alive TO Same                    (* Transition One *)
        BEGIN
            Store_data ( pool, TS_user_data )
        END;

        FROM Receiving TO Alive               (* Transition Two *)
        BEGIN
            rcvcnt := rcvcnt + 1;
            .....
            TS_output ( user_id, response );
        END;

TRANS
    PROVIDED TS_input ( user_id, request )    (* Transition Three *)
    BEGIN
        case request.kind of
            T_CONNECT :
                if state = Idle then begin
                  state := Waiting;
                  .....
                  OUT TSAP.T_CONNECT_request ( local_addr,
                                               remote_addr,
                                               qual_of_service,
                                               request.data )
                end;
                .....
    END;
```

Figure 2.6: An Example of Transition Specification

Chapter 3

The Implementation Strategy

In automatic implementation of protocols, a generic structure and organization of the implementation must be adopted. The implementation strategy adopted for our C-Estelle compiler is similar to the one used by G. Gerber in his Pascal-written Estelle compiler [Gerber83]. This approach makes use of data structures to represent module instances, interaction points, and interactions among module instances. A set of pre-written generic functions is used to allocate, initialize, and link data structures according to an Estelle specification. The pre-written functions also dispatch an output interaction to a recipient module, select the next available interaction, and make non-deterministic choice. Since different systems have different global environments and scheduling schemes, two special functions, namely **system_init** and **schedule** have to be tailored according to each specification. Figure 3.1 depicts the procedure of the semi-automatic implementation.

3.1 Data Structures

There are three major data structures which represent module instances, interaction points and interactions between module instances. When linked appropriately, these data structures can represent an arbitrarily complex Estelle specification in a simple manner.

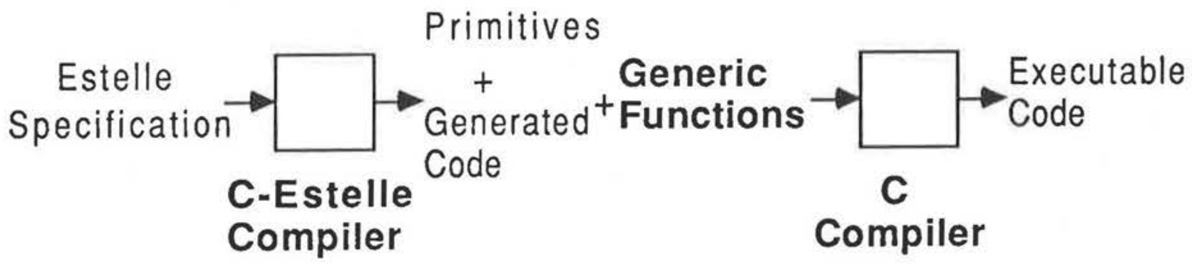


Figure 3.1: Procedure of the Semi-Automatic Implementation

In Figure 3.2, data structure **signal_block** represents an interaction (i.e a signal) and is

```
struct signal_block {
     int                 signal_id;
     struct signal_block *next;
     union {
     .....
     } lvars;
};
```

Figure 3.2: Data Structure of an Interaction

comprised of three attributes, namely *signal_id, next*, and *lvars*. For convenience, interaction primitives, specified in channel-type definitions, are numbered. These numbers are used in *signal_id* to identify an interaction. The attribute *next* links data structures to implement the queueing of incoming interactions at an interaction point. The values of the parameters of an interaction are stored as a single attribute *lvars* in the data structure. A simple scheme is applied to avoid the name conflict of having identical parameter names in different interaction primitives and identical interaction names in different channel-types. Interaction primitives under the same channel-type are grouped in a dummy structure which then appears as the only attribute of a variant of lvars. Similarly, parameters of an interaction primitive are grouped in a dummy structure which works as the only attribute of a variant of the interaction primitive.

Representing a module instance, data structure **process_block** (Figure 3.3) consists of

```
struct process_block {
     struct process_block *next;
     char                 p_ident[MAX_IDENT_LENGTH+1];
     struct channel_block *chan_list;
     struct process_block *refinement;
     int                  (*proc_ptr)();
     union {
     .....
     } lvars;
};
```

Figure 3.3: Data Structure of a Module Instance

six attributes, namely *next, p_ident, chan_list, refinement, proc_ptr,* and *lvars*. Similar to signal_block structure, a variant is added to attribute *lvars* of the structure in each module type definition. Local variables are grouped in a dummy structure as a single attribute in each variant. The attribute *proc_ptr* is an entry point to a transition function which implements the transition process of the corresponding protocol machine. The remaining attributes are used to identify the corresponding transition function, and to build and link various data structures modeling the specified system.

Representing an interaction point, data structure **channel_block** (Figure 3.4) contains the following attributes : *target_proc*, and *target_channel* are entry points to data structures which represent peer module instance and its corresponding interaction point; *signal_list* points to a list of incoming interaction; *queued* is a boolean flag that indicates the queueing discipline (queued

```
struct channel_block {
     struct channel_block *next;
     int                  *signal_list;
     int                  *target_proc;
     struct channel_block *target_channel;
     int                  queued;
     int                  c_id;
     int                  index_num;
};
```

Figure 3.4: Data Structure of an Interaction Point

or rendezvous) of the interaction point; *c_id* identifies the interaction point and additional *index_num* is used in case of multiplexing channel; finally *next* links all interaction points of a module-type.

3.2 Interactions

As mentioned in Chapter 2, interactions can be classified into **queued** and **rendezvous** types. Output queued interactions from a module are queued in the recipient module. They are considered by the global scheduler as input interactions to the recipient module in due time. On the other hand, output rendezvous interactions are sent to and consumed by the recipient module immediately. If the recipient module is not in a state which the incoming interaction can initiate a transition, the interaction is added to the awaiting incoming interaction queue and will be considered immediately for execution in due time by the global scheduler.

3.3 Transitions

In a given global system state, a number of different transitions belonging to different module instances is possible. The selection of the next available transition to be performed is made by a global scheduler, which is not part of the Estelle specification but part of the run-time support for the implementation. A simple round-robin scheduler is applied to choose the next available transition.

For a given input interaction and a given major state of a module instance, several different input transitions may occur. Similarly, several spontaneous transitions can exist for a given major state of a module instance. For simplicity, the first possible transition in the same order as defined in the specification is selected to be performed. Hence, for each cycle, in addition to which module instance, the global scheduler selects the next input transition only based on the interaction point and the input interaction, or just determines whether a spontaneous transition to be taken next.

3.4 System Interfaces

For each implementation, the protocol implementors will have to manually look after the system-dependent portion of the implementation, i.e. interactions between the specified protocol machine and its working environment. For instance, interactions with the operating system usually cause an undesirable blocking of the protocol machine and the solution to avoid such blocking varies largely on different machines and different operating systems. However, working environment such as the operating system is always known and its interfaces with the specified system can be well defined. This apriori knowledge can be used to simplify the system interactions. In our implementations, UNIX 4.2 socket primitive **select** is used to preview the socket so that the blocking is avoided when reading a socket. Thus, output to the environment can be implemented by invoking a set of system-dependent routines, while input from the environment by including spontaneous transitions which invoke the same set of routines. The global scheduler is fully aware of when and which spontaneous transition should be performed.

Chapter 4

The C-Estelle Compiler

In order to support the implementation strategy described in Chapter 3, a C-Estelle compiler was developed by D. Ford [Ford85] who rewrote G. Gerber's [Gerber83] Pascal-written Estelle compiler in the language C on a VAX 11/750 running UNIX 4.2bsd. The compiler was then modified by K. Chan, adding the capability of recognizing the additional scope of transition group. The previous version of the C-Estelle compiler was erroneous and insufficiently tested. In order to make it useful, the performance of the compiler has been greatly enhanced by transforming the BNF grammars into LALR grammars which best fit the YACC compiler for generating the parser of the C-Estelle compiler. The grammar rules were also rewritten so that the compiler supports complex data structures such as variant record and pointer which are commonly used in real-life protocol specifications. Furthermore, the translation routines were modified to produce optimized and better-organized code. During the test period, many minor problems, such as incorrect translation of Pascal **for** statement, have also been fixed. The enhanced compiler was later ported to several SUN Workstations and protocol implementations such as hot potato, alternating bit and ISO class 2 Transport Protocol are successfully running among the VAX 11/750 and SUN Workstations.

The enhanced C-Estelle compiler reads Estelle protocol specifications and produces C code. The generated C code is then incorporated with sets of system-dependent and pre-written generic routines into a C program which implements the specified communication protocol. This semi-automatic construction of protocol implementation is the main purpose of the development of the C-Estelle compiler.

4.1 The Structure

Similar to many other compilers [Aho78], the C-Estelle compiler is partitioned into several phases as shown in Figure 4.1. Both lexical analyzer and parser were generated by the UNIX standard utilities LEX [Lesk75] and YACC [John75] respectively. Error handling, table management and code generation were embedded in the YACC grammar input file. Currently, the compiler does not optimize the generated C code. It completes the translation in a single pass of the source specification.

A large number of semantic analysis is left untouched to the C compiler which compiles the generated C code into executable machine code. The C-Estelle compiler only verifies the semantic conditions that would not be detected by the subsequent C compilation. For instance, the C-Estelle compiler ensures, for each connection, that the two connected module instances play the different roles of the same channel-type. On the other hand, the C-Estelle compiler does not verify that arguments are of types which are legal for an application of an assignment.

4.2 Translation Issues

4.2.1 Pascal to C Problems

Since Estelle is a Pascal-based language, translating Pascal code into C code is a primary issue addressed during the implementation of the C-Estelle compiler. Although both Pascal and

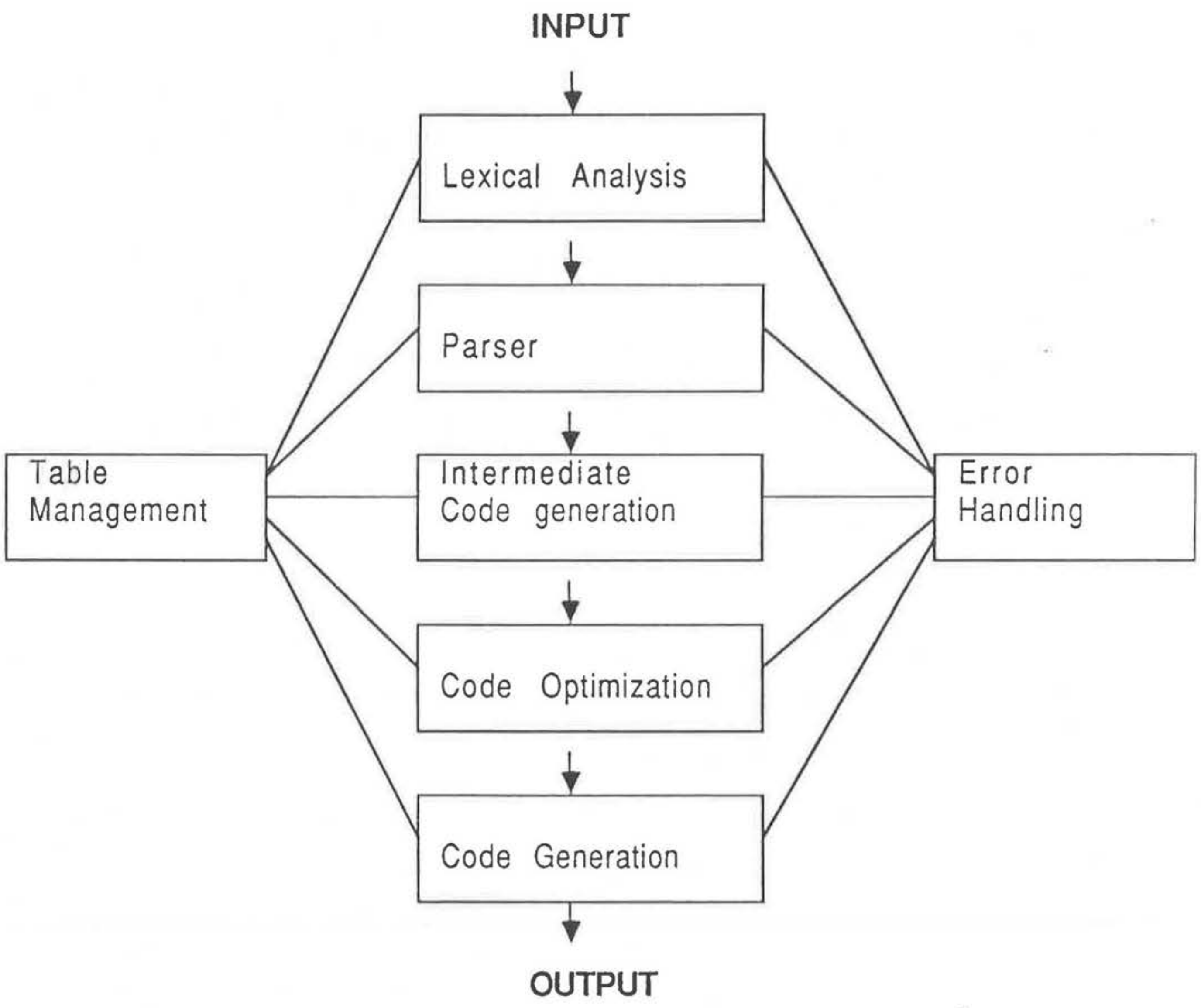


Figure 4.1: The Structure of the C-Estelle Compiler

C are high-level programming languages which have similar control flow constructions and basic data types, they have enough differences which makes the direct translation a very difficult task. The following discussion has a great impact on the performance and the use of the C-Estelle compiler.

First of all, both languages have very different approaches in defining the scope of objects. In Pascal, procedures and functions can be nested, and identifiers have no storage class attributes. The scope of an identifier is the block in which it is declared and every sub-block in which the identifier is not declared again. Whereas in C, only external functions are supported; function nesting is not allowed, and identifiers have a special storage class attribute. The scope of an identifier within a source file is basically the same as the one in Pascal. In addition, identifier which is not declared in any block, can be accessed within any blocks that is lexically after its declaration. Furthermore, the scope of externals, identifiers whose storage class are **extern**, may be defined in another source file. Two proposed solutions are to use multiple output files and to make all identifiers distinct and external. Both solutions are not straight-forward and very cumbersome to implement. For simplicity, the use of Pascal's scoping rules and nested routines is disallowed. Thus, when using the C-Estelle compiler, both global variables and nested routines are not allowed.

Secondly, self-referential data structures are declared in different sequences. Due to the syntax of Pascal type declaration, self-referential data structure is defined in a way that a self-referential pointer to an object can be exceptionally defined before the object is defined. C does not have this syntax problem and an object must be defined before its reference pointer is defined. Hence, direct translation is not possible. The solution employed in the C-Estelle compiler is to define all objects first and then pointers.

Thirdly, the formats of input/output statements are very different. Directly translation is so difficult that only Pascal's output statements, i.e. **write** and **writeln** statements, are supported and translated into equivalent C **printf** statements. Other forms of input/output statements can be embedded in primitive routines.

Furthermore, Pascal's unique **WITH** statements and **SET** operations cannot be translated directly into any equivalent C statements. Additional statements and pre-written functions are required to make the translation. These Pascal features are currently not supported.

4.2.2 Estelle to C Considerations

In addition to the above-mentioned difficulties of translating Pascal into C, there are certain aspects of Estelle which are very hard to handle. These are the additional Estelle scoping rules introduced by the enabling conditions of a transition type and the additional variables used by the run-time supporting routines. Some restrictions have been imposed in order to overcome these problems.

First of all, the parameters of an input interaction, which are declared in the corresponding channel-type definition, are accessible within the scope of a WHEN clause. To avoid the name conflict, the parameter names cannot be used for local variables for any module-types which the interaction may occur. Secondly, if the interaction point identifier in a WHEN clause is indexed, the index identifier(s) must be declared as local variable(s) of the corresponding module-type. Thirdly, since an ANY clause introduces additional variable(s) within the scope of the clause, a block is used to hide the new variable(s) from other transitions. The value of the variable is randomly selected from its specified domain by a pre-written function. Furthermore, additional identifiers are generated by the C-Estelle compiler and used by the run-time supporting functions. These identifiers should never be in conflict with other identifiers of the specification which are still present in the generated C code.

Chapter 5

Implementation Example - The ISO Transport Protocol

In order to evaluate the usefulness of the C-Estelle compiler, a fairly complex ISO class 2 Transport Protocol has been implemented both semi-automatically by using the C-Estelle compiler and manually in an ad hoc manner. Both implementations run on a VAX 11/750 and several SUN Workstations under the UNIX 4.2bsd operating system. After an overview of the protocol, the design of its implementation is presented. The two implementation approaches and the experience learned from the implementations are discussed, followed by a tentative comparison of these implementations.

The state diagram of the protocol is depicted in Appendix A and the Estelle specification of the protocol in Appendix B. The system initializer and scheduler of the semi-automatic implementation is listed in Appendix C and those of the manual one in Appendix D.

5.1 Overview of The ISO Class 2 Transport Protocol

The ISO Transport Protocol [CCITT85,ISO82b] is a connection-oriented, end-to-end protocol, providing a reliable and efficient mechanism for the exchange of data between processes in different computer systems. The class 2 protocol assumes a highly reliable network service, such as X.25, and has the ability to multiplex multiple Transport connections onto a single network connection. It also uses a credit allocation scheme to provide an explicit flow control because a single network connection flow control is insufficient to handle individual flow control of multiplexed Transport connections.

Since Transport layer provides end-to-end data transfer independent of the nature of the underlying network, the Transport service is the same for all classes. The ten Transport service primitives have been listed in Figure 2.1 and Figure 5.1 displays the sequence in which these primitives are used. In order to communicate over a Transport connection, nine types of Transport protocol data units (**TPDUs**) are used. These TPDUs, shown in Figure 5.2, carry parameters which play an important role in the protocol mechanism.

Each TPDU conveys a destination reference which uniquely identifies the Transport connection within the receiving Transport entity. Thus, multiplexing is allowed. After a Transport connection is established by exchanging CR/CC TPDUs, each data TPDU (DT/ED TPDU) is sequentially numbered. This sequence number is used for the flow control. A Transport connection is released whenever the Transport entity has sent or received a DR TPDU. The entity will then ignore any incoming TPDUs except DC/DR TPDUs. This explicit termination mechanism allows that a Transport connection is released independently of the underlying network connection.

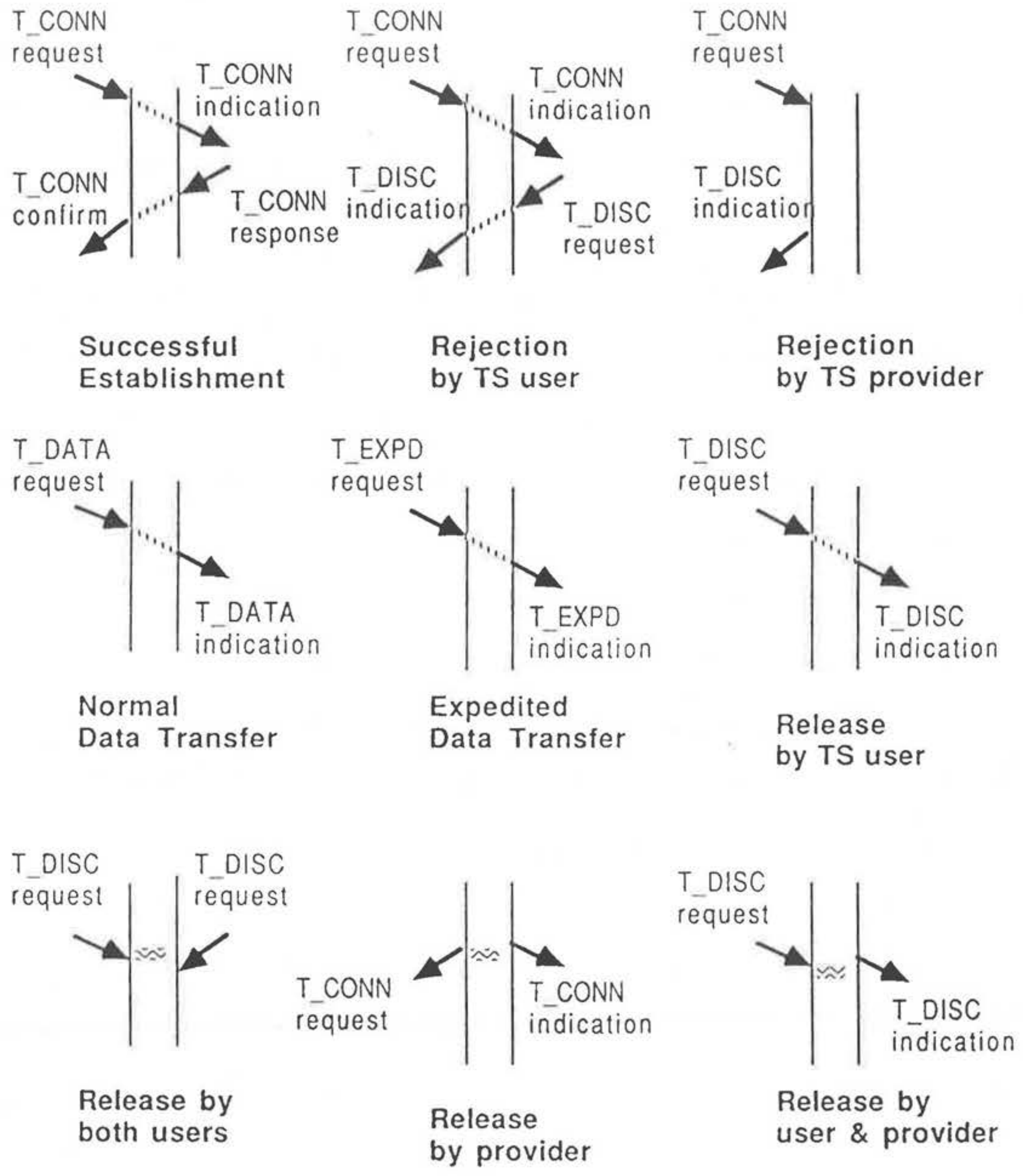


Figure 5.1: Transport Service — Primitive Sequence

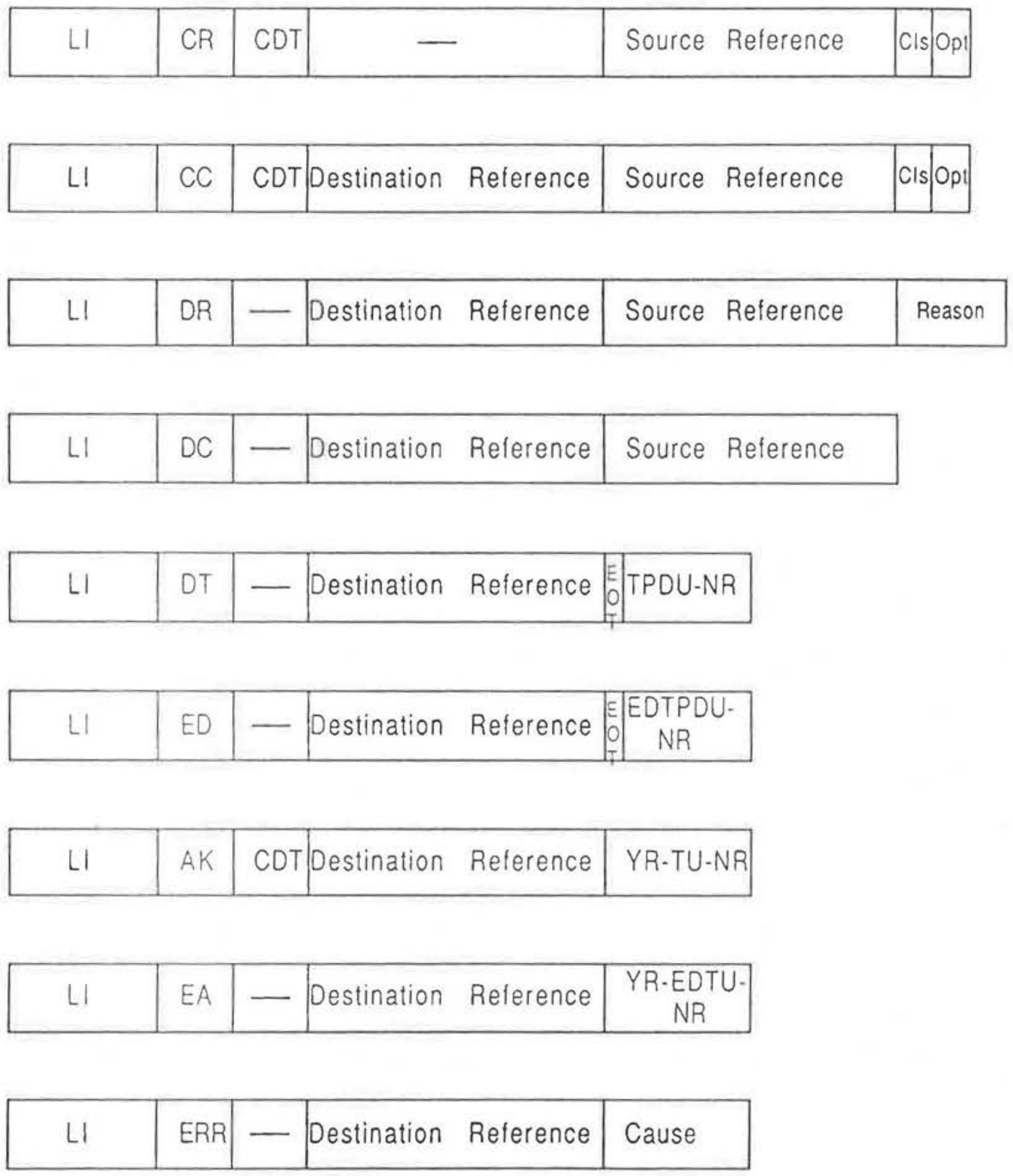

LI | CR | CDT | — | Source Reference | Cls | Opt
LI | CC | CDT | Destination Reference | Source Reference | Cls | Opt
LI | DR | — | Destination Reference | Source Reference | Reason
LI | DC | — | Destination Reference | Source Reference
LI | DT | — | Destination Reference | EOT | TPDU-NR
LI | ED | — | Destination Reference | EOT | EDTPDU-NR
LI | AK | CDT | Destination Reference | YR-TU-NR
LI | EA | — | Destination Reference | YR-EDTU-NR
LI | ERR | — | Destination Reference | Cause

Figure 5.2: Transport Protocol Data Unit Fixed Header Formats

5.2 Design of the Implementation

5.2.1 Structure

The overall structure of an Estelle specified Transport entity has already given in Figure 2.3. There are four different module types : TS_user, ATP, System and RS. Module instances of these four module types are incorporated with each other to represent a Transport entity.

A TS_user module is a sub-layer which converts a Transport service user request into a well-defined Transport service primitive or changes the module state according to the request. A user task in the working environment can bind with one or more than one TS_user modules, and hence one or more than one Transport connections. An ATP module is an abstract Transport entity that establishes Transport connections, transfers data, and releases connections. A System module simulates a system timer for an incoming network connection or the flow control of a Transport connection. Finally, a RS module converts the network service primitives into system calls. It also sets flag and stores data whenever an incoming network event occurs.

5.2.2 Implementation Issues

Since there are many unspecified properties in the protocol specification, these properties have to be determined for each particular implementation such that the resulting implementation best fits the working environment. Unspecified properties can be classified into **implementation-defined** and **implementation-dependent**.

Implementation-defined properties are left unspecified and their definitions can vary from one implementation to another. For instance, in the TS_primitives channel definition, data type ADDR_TYPE is implementation-defined. Type ADDR_TYPE represents Transport address which may be implemented differently by different implementors. Similarly, the buffer management and data exchanged by TS_users and a TS_provider are all implementation-defined. Their definitions and implementations are left untouched to the implementor.

On the other hand, some properties are defined in the specification but their implementation is left unspecified. Examples of such properties are functions constructing Transport protocol data units. The format of a Transport protocol data unit is specified but how to construct such a TPDU is unspecified.

5.2.3 Scheduler Design

A simple round-robin scheduler is employed to select the next available input interaction. This scheduler scans queues associated with each interaction point of module instances for the existence of any input interactions. The first available interaction is chosen and passed together with the information of the associated interaction point to the module instance which executes a transition.

As mentioned in Section 3.3, for a given input interaction and a given module state, a number of transitions may be possible. Which possible transition is chosen to execute depends on the priority and the order it is defined in the specification. Generally, the chosen transition is the one has the highest priority and the first one which enabling condition is satisfied.

At a regular time interval, a module instance which has spontaneous transitions is attempted to execute one of its spontaneous transitions. The first possible spontaneous transition which enabling condition is satisfied will be performed. This simple scheme works fine provided that the enabling conditions of the spontaneous transitions are all distinct, and spontaneous transitions are defined in a well-defined order.

The above consideration of spontaneous transitions does not work satisfactorily for those initiated by the working environment. A module instance require to execute such a transition immediately whenever the working environment notifies the module an external event occurred. The global scheduler is fully aware of the external events, and invokes the module instance to perform an action immediately whenever such event comes up.

5.3 Semi-Automatic Implementation

The protocol was first specified in Estelle from the description in the ISO document [CCITT85,ISO82b] and by adapting many other specification attempts [ISO84,NBS83]. The Estelle specification was then compiled by the C-Estelle compiler to generate parts of the protocol implementation. After this automatic process, the generated code was incorporated with the pre-written generic routines and the system-dependent functions into a C program to implement the protocol in question.

5.3.1 The Generated Code

The generated code can be classified into three types. The first type is the **deftype** and **structure** declarations which represent module instance, interaction, type and variable definitions. These definitions are required by the run-time executives to store the state information of the protocol machines. The second type is a set of functions which creates, initializes and constructs data structures in the specified fashion. The last type is another set of functions which implements the transition processes of the protocol machines.

Most data structures are self-explanatory and the special data structures have been discussed in Chapter 3. They are the wheels of the protocol machines which are initialized and constructed by the generated functions to implement the specified protocol.

Initialization functions can be further subdivided into two types, depending on their corresponding Estelle specifications. A function which corresponds to an Estelle **Process** definition creates and initializes a **process_block** data structure. This process_block represents one of the protocol machine instances in the specified system. Other type function corresponds to an Estelle **Refinement** definition. It creates the sub-module instances and links the instances according to the Estelle CONNECT and REPLACE definitions. Both type functions use a set of pre-written generic function to perform the creation, initialization, and integration of the specified system components.

Transition functions are simply a series of conditional expressions and statement blocks. Expressions evaluate the enabling conditions of a possible transition type and block performs the associated action. Unless priority is set, input transition types are always generated ahead of spontaneous transition types. Only the first transition type, which enabling condition is satisfied, will be performed at a given time.

Each transition type is generated in the same pattern. For an input transition, the operation is preceded by tests on the identity (*signal_id*) of the received interaction and those (*c_id* and *index_num*) of the interaction point at which it came. Additional tests, which correspond to PROVIDED clause and/or TO clause, may also preceded the operation. At the end of each transition type, a *goto dispose* statement passes control to the **signal** data structure dispose code. For a spontaneous transition, the pattern is the same except that no tests on the identities of the input interaction and the interaction point. For an ANY clause, which requires to make a non-deterministic choice, a sub-block is created. The specified variable(s) is declared within the sub-block and its value is randomly selected from its defined domain by the pre-written function **random_select**.

Creation and destruction of **signal** structures which represent interactions between module instances are implemented completely within the generated transition functions. The output statement OUT is implemented as follows. First, a signal structure is created and initialized with the given parameters. The signal structure is then passed to a generic function **out** together with the information of the interaction point at which the module instance interacts with the peer. If the interaction is a queued type, the signal structure is placed in the reception queue of the peer module instance. Control returns to the initiating module instance immediately. If the interaction is a rendezvous type, the transition function corresponding to the peer module is invoked directly at this point. The destruction of the signal structure is handled by the recipient module instance.

5.3.2 Integration Process

For convenience, deftype and structure definitions of the generated code were first extracted into a well-known header file **defs.h**. Two run-time supporting functions, **system_init** and **schedule**, was then modified to suite the specified system. Finally, the generated code was incorporated with the system-dependent primitives and the run-time supporting functions into a C program to implement the protocol in question.

Besides **defs.h**, there is another global header file **listdefs.h** included in all files. File **listdefs.h** contains macro definitions and specification-independent **channel_block** structure declaration. This structure is used to represent an interaction point of a module. Another important header file **fdtglobal.h**, which is required to be modified for every different specification, contains the declaration of all global variables and external functions. This **fdtglobal.h** file is included only in the main routine file. There are two key global variables : **p_block** and **signal_pending**. During execution, pointer **p_block** is an entry to the current machine instance, and **signal_pending** is a counter of interactions which have been initiated and are waiting for execution.

To execute, function **system_init** first builds and interconnects the specified machine instances. The working environment is also set up so that the upcoming scheduler can be fully aware of any interested external events. Function **schedule** is then invoked to repeatedly scan all interaction queues associated with channels and to activate the module instances. Module instances which contain spontaneous transitions are tried at a regular time interval. Furthermore, whenever an external event occurs, the scheduler will activate a proper module instance to perform a special-designed spontaneous transition.

5.4 Manual Implementation

Based on the same specification and the semi-automatic implementation, the protocol was re-implemented manually in an ad hoc manner. Most principles discussed in Chapter 3 and previous Section 5.2 were followed. The overall structure is similar to that of the semi-automatic implementation. The Transport entity is implemented as a single task in the operating system. It communicates with user tasks and the network service provider through operating system primitives (i.e. system calls). The major difference to the semi-automatic approach is the implementation of scheduling interactions which are initiated either by a module instance or the working environment.

Instead of using a single data structure **process_block**, three different data structures, TS_MACHINE, TP_MACHINE and NP_MACHINE, are designed to store the state information of a Transport service user, a Transport connection and a network service provider respectively. Three global variables, **tslist, tplist** and **nplist**, are declared as head pointers of the three different control queues.

The interactions between the Transport entity task and the working environment, user tasks and the network service provider, are based on the inter-process communication primitives provided by the operating system, i.e. UNIX 4.2bsd socket primitives. Spontaneous transitions initiated by the working environment were handled in an ad hoc manner similar to that in the semi-automatic implementation. Whenever an external event occurs, the corresponding module instance is activated to perform a proper transition. A series of input transitions, initiated after this spontaneous transition, is then performed until all module instances are in a steady state. As a result of this transformation, the global scheduler is simply a loop which performs the processing for the incoming external events one after the other.

5.5 Results

The size of different parts of the resulting implementations are shown in Table 5.1. Both implementations used the same INET primitives to interact with the network service provider. This network service provider is usually a daemon process in the operating system. INET primitives provide an uniform access scheme which can be easily modified to suite different network service access schemes in different systems. Similarly, TSP primitives were used for the interactions between Transport service user tasks and the Transport entity task.

Both implementations spent a large amount of code in TPDU encoding/decoding and buffer management. However, they were not very difficult to implement because of the powerfulness of the C language. The encoding/decoding of TPDUs were implemented almost the same in both implementations. Both implementations shared the same header file **pdu.h** and differed only in the passing parameters when decoding a TPDU. Since the buffer management was implemented intermixed with other code in the manual implementation, no separate entry for its code is in the table.

PART OF PROGRAM	Number of Functions and Macros (A)	(B)	Number of Source Lines (A)	(B)	Program size (in bytes) (A)	(B)
INET PRIMITIVES	9		509		10969	
TSP PRIMITIVES	12		741		17073	
ESTELLE SPECIFICATION	—	20	—	1910	—	46351
GENERATED CODE		20		1447		91421
RUN-TIME SUPPORTING ROUTINES	76	16	3420	770	78821	21054
PRIMITIVE ROUTINES		82		3049		71340

(A) --- Manual Implementation

(B) --- Semi-Auotmatic Implementation

Table 5.1: Sizes of Different Parts of Implementations

Forty two additional functions were used in the semi-automatic implementation. Sixteen of them were pre-written run-time supporting functions and the rest were specially designed for the global scheduler to activate the specific modules.

During the semi-automatic implementation, the most difficult task was integrating the generated code with the working environment. Both the implementation scheme using by the C-Estelle compiler and the behaviour of the working environment must be thoroughly understood in order to design the specific spontaneous transitions and to modify the two special run-time supporting functions : **system_init** and **schedule**.

The weakness of Estelle forced the static allocation of data structure **process_block** which represents a module instance. The number of Transport service users and network connections must be pre-defined in the specification. The pre-definition was then used by the C-Estelle compiler to generate code that the corresponding **process_block** structures must be allocated in the global initialization phase. To execute, a pre-defined number of Transport service user tasks must be executed so that the implemented system went through the global initialization stage.

The advantages of the semi-automatic approach came from the well-constructed generated code. Since the code was generated directly from a formal specification, the conformation was almost guaranteed. The well-constructed code also localized hazards and system dependent properties in a few routines, and hence, maintenance was much easier.

On the other hand, the most difficult task of the manual implementation was to design the interfaces with the operating system for interactions with the user tasks and the network service provider. Interactions initiated by the working environment intermixed with other input interactions. The layer structure was less clear in the resulting code. A longer debugging period was spent and more exceptional cases were required to be handled.

Although the manual implementation was based on the same specification, no restriction on static allocation was imposed in the global initialization phase. Any number of Transport service user tasks can interact with the Transport entity. The Transport entity required no static connections to go through its initialization phase. Furthermore, any number of network connections can be established during the execution.

The manual implementation is tied closer with the working environment. An interaction was implemented as simply a function call. It was always faster than the semi-automatic implementation because of the reduction of a large amount of generated code which had additional swapping overhead for module interactions.

It took approximately one year to study and implement the ISO class 2 Transport Protocol manually in an ad hoc manner without an Estelle specification. The protocol was subsequently specified in Estelle, and re-implemented semi-automatically in about two months. After this exercise, we gained a profound experience on protocol implementation and a good insight to the ISO class 2 Transport Protocol. Therefore, in our last attempt, it took us only one month to re-implement the protocol manually. From our experience, we think it saves protocol development times and it is good practice to start with the semi-automatic approach to protocol implementation, assuming one is familiar with the FDT compiler. The code produced this way is well structured and easy to maintain. Even if the code is not efficient enough, we can always attempt a manual implementation subsequently. Protocol implementations generally require a lot of time on the development of the interfaces with the working environment. The manual approach required additional time to implement module interactions. It also required more debugging time than the semi-automatic approach.

Chapter 6

Conclusions

6.1 Thesis Summary

This thesis has discussed a semi-automatic approach to implement a protocol. The protocol is first specified in the Estelle FDT, and translated into C code by using an automatic tool, C-Estelle compiler. The generated code is then incorporated with system-dependent primitives and run-time supporting functions into a C program which implements the protocol in question.

Despite the fact that the semi-automatic implementation tends to be slow and an initial effort is required to learn the Estelle FDT and the automatic tool C-Estelle compiler, the new approach has the following benefits :

1. Easy maintenance because the generated code was constructed in a simple and easy-to-read pattern.
2. Good conformance because the specification was directly (automatically) translated into C code.
3. High portability because large amount of code was generated in standard C language and system-dependent properties were easily located and modified.
4. Less development time because large amount of code was translated directly from the specification.

Experience on implementing the ISO class 2 Transport Protocol has verified the usefulness of the C-Estelle compiler and the semi-automatic approach to protocol implementation. From our experience, it is a good practice to approach a protocol implementation in the following sequence :

1. Implement the protocol semi-automatically using the C-Estelle compiler.
2. Optimize the semi-automatic implementation, especially the generated code.
3. Re-implement the protocol manually (if high performance is required.)

6.2 Future Work

Further study on the semi-automatic implementation would be useful, in that a protocol can be implemented by two completely independent teams, one using the traditional ad hoc approach and the other, the new semi-automatic approach. This way, more concrete and objective comparisons can be made on the performance and usefulness of the new approach.

Further testing of the C-Estelle compiler on complex protocols such as ISO class 4 Transport Protocol is a natural extension of our thesis. Such experiment would further demonstrate the usefulness of the compiler. Several enhancements to this technique and the compiler are under consideration.

In order to enhance the C-Estelle compiler, some of the high-level code for interactions of the specified system with its working environment should be generated by the compiler. Dynamic structure, such as *Process* allocation should be supported by the Estelle, and hence the compiler. Since global variables, WITH statements and SET operations are very useful features, the compiler is also required to support them.

To be realistic, the compiler should be modified to support a general multi-process structure instead of the procedure-oriented structure. Since UNIX 4.2bsd is a procedure-oriented operating system, a better working environment, such as V-system and Team Shoshin which are process-oriented , may be chosen.

To overcome the Pascal-to-C problem, a C-oriented FDT would be desirable for protocol implementors who are working in C/UNIX oriented environment. However, the apparently irreversible decision by the ISO standard committee (ISO TC 97/SC 16/WG 1 - FDT Subgroup B) has been made to keep Estelle Pascal-oriented. Whenever the final Estelle standard becomes available, the compiler will have to be adapted to that (our implementation of the C-Estelle compiler is based on [ISO84], not the latest [Estelle85]).

As the last comment, the compiler can be well used as a simulation tool, and could be incorporated with some validation, testing and performance evaluation facilities so that we can have a complete automatic system for the design, validation, implementation, testing and performance evaluation of the communication system.

Bibliography

[Aho78] Aho, A. and Ullman, J., "Principles of Compiler Design," Addison–Wesley, 1978.

[Ansart83] Ansart, J.P., Chari, V. and Simon, D., "From formal description to automated implementation using PDIL," *Protocol Specification, Testing and Verification,* (IFIP/WG 6.1), H. Rudin and C. H. West, eds, North Holland (1983).

[Blum82] Blumer, T.P. and Tenny, R., "A formal specification technique and implementation method for protocols," *Computer Networks,* 6 (3), June 1982, pp. 201–217.

[Boch80] Bochmann, G.v. and Sunshine, C., "Formal Methods in communication Protocol Design," *IEEE Trans. on communications,* COM–28 (2), April 1980, pp. 624–631.

[Boch84] Bochmann, G.v., Gerber, G. and Serre, J.M., "Semi-automatic Implementation of Communication Protocols," TR 518, d'IRO,Universite de Montreal, December 1984.

[Bria86] Briand, J.P., Fehri, M.C., Logrippo, L. and Obaid, A., "Structure and Use of a LOTUS Interpreter," *SIGCOMM '86,* Symposium, Vermont, 1986.

[Brin85] Brinksma, E., "A Tutotial on LOTUS," *Protocol Specification, Testing and Verification V,* (IFIP/WG 6.1), M. Diaz, eds, North Holland (1985).

[CCITT85] CCITT, Recommendations X.200 to X.250, Red Book, Geneva, 1985.

[Estelle85] ISO TC 7/SC 21/WG 1 – FDT, Subgroup B, "Estelle — a formal description technique based on an extended state transition model," Feb. 1985.

[Ford85] Ford, D.A., "Semi–Automatic Implementation of Network Protocols," Master Thesis, University of British Columbia, March 1985.

[Gerber83] Gerber, G.W., "Une Methode D'Implantation Automatisq de Systemes Specifies Formellement," Master Thesis, University of Montreal, 1983.

[Grog80] Grogono, P., "Programming in Pascal," Rev. ed., Addison–Wesley, 1980.

[Hans84] Hansson, H., "Aspie, A system for Automatic Implementation of Communication Protocols," *Uptec 8486R*, Uppsala Institute of Technology, Uppsala, 1984.

[ISO82a] ISO TC 97/SC 16, DP 8073, "Transport Protocol specification," June 1982.

[ISO82b] ISO TC 97/SC 16, DP 8072, "Transport Service Definition," June 1982.

[ISO84] ISO TC 97/SC 16/WG 1 — FDT, Subgroup B, "A Formal Description Technique based on an extended state transition model," Working Document, March 1984.

[John75] John, S.C., "YACC : Yet Another Compiler-Compiler," CS TR 32, Bell Laboratories, NJ, 1975.

[Kern78] Kernighan, B.W. and Ritchie, D.M., "The C Programming Language," Prentice–Hall, 1978.

[Lesk75] Lesk, M.K., "Lex—A Lexical Analysis Generator," CS TR 39, Bell Laboratories, NJ, 1975.

[Lotos84] ISO TC 7/SC 16/WG 1 – FDT, Subgroup C, N 299, "Definition of the Temporal Ordering Specification Language," May 1984.

[NBS83] National Bureau of Standards, "Specification of a Transport Protocol for Computer Communication," ICST/HLNP 83–2, Feb. 1983.

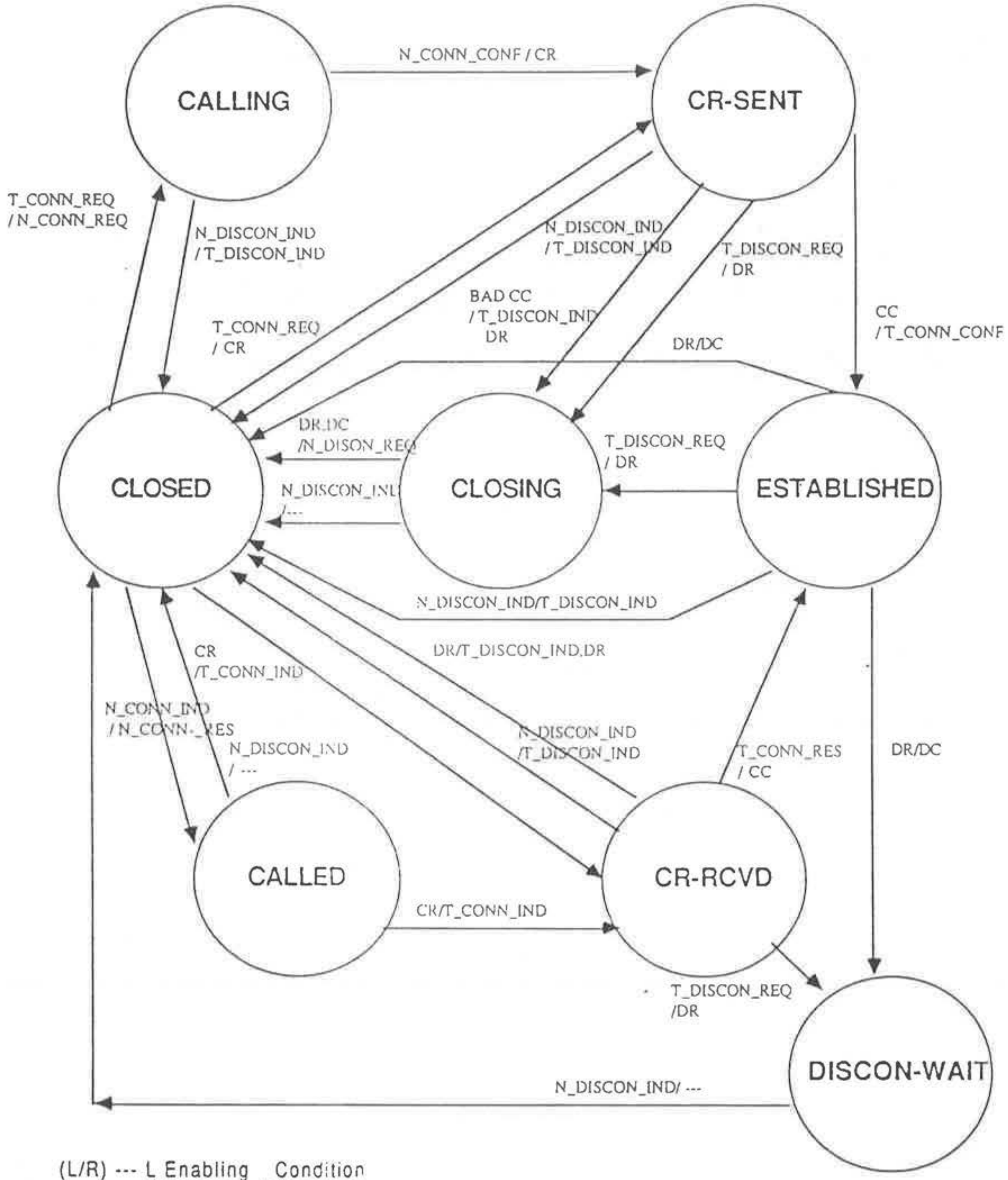
[Rit78] Ritchie, D.M, and Thompson, K., "The UNIX time-sharing system," Bell Sys. Tech., 57(6), July 1978, pp. 1905–1929.

[Tanen81] Tanenbaum, A.S., "Computer Networks," Prentice–Hall, 1981.

[Vuong86] Vuong, S.T., and Ford, D.A., "An Automatic Approach to Protocol Implementation," TR draft, Dept. of Comp. Sci., University of British Columbia, 1986.

Appendix A

The ISO Class 2 Transport Protocol — State Diagram



(L/R) --- L Enabling Condition
R Output Interaction

Figure A.1: Transport Protocol State Diagram

Appendix B

The ISO Class 2 Transport Protocol — Estelle Specification

```
MODULE Transport_system;
END Transport_system;

REFINEMENT Transport_ref FOR Transport_system;

(*********************************************************************
 *
 *      Transport Protocol machine Module
 *
 *********************************************************************)

(* Constant and Type Definitions *)

.....

(* Channel Definitions *)

CHANNEL TS_primitives ( TS_user, TS_provider ) ;

        BY TS_user :

                T_CONNECT_request      ( From_transport_add:  : ADDR_TYPE;
                                         To_transport_addr     : ADDR_TYPE;
                                         Qual_of_service       : QOS_TYPE;
                                         TS_user_data          : DATA_TYPE );

                T_CONNECT_response     ( Qual_of_service       : QOS_TYPE;
                                         TS_user_data          : DATA_TYPE );

                T_DATA_request         ( TS_user_data          : DATA_TYPE );

                T_XPD_request          ( TS_user_data          : DATA_TYPE );

                T_DISCONNECT_request   ( TS_user_data          : DATA_TYPE );

        BY TS_provider :

                T_CONNECT_indication      ( From_transport_addr   : ADDR_TYPE;
                                            To_transport_addr     : ADDR_TYPE;
                                            Qual_of_service       : QOS_TYPE;
                                            TS_user_data          : DATA_TYPE );

                T_CONNECT_confirm         ( Qual_of_service       : QOS_TYPE;
                                            TS_user_data          : DATA_TYPE );

                T_DATA_indication         ( TS_user_data          : DATA_TYPE );

                T_XPD_indication          ( TS_user_data          : DATA_TYPE );

                T_DISCONNECT_indication ( Reason                  : REASON_TYPE;
                                            TS_user_data          : DATA_TYPE );

END TS_primitives;

CHANNEL NS_primitives ( NS_user, NS_provider );

        BY NS_user :

                N_CONNECT_request         ( From_network_addr     : NADDR_TYPE;
                                            To_network_addr       : NADDR_TYPE;
                                            QOS                   : NQOS_TYPE );

                N_CONNECT_response;

                N_DATA_request            ( NS_user_data          : NDATA_TYPE );

                N_DISCONNECT_request;

        BY NS_provider :

                N_CONNECT_indication      ( From_network_addr     : NADDR_TYPE;
                                            To_network_addr       : NADDR_TYPE;
                                            QOS                   : NQOS_TYPE );
                N_CONNECT_confirm;

                N_DATA_indication        ( NS_user_data   : NDATA_TYPE );

                N_DISCONNECT_indication ( Reason          : REASON_TYPE );

END NS_primitives;

CHANNEL System_primitives ( S_user, S_provider ) ;

        BY S_user :

                Timer_request     ( Name  :          TIMER_TYPE;
                                    Time  :          integer;
                                    Seqno :          SEQUENCE_TYPE    );

                Timer_cancel      ( Name  :          TIMER_TYPE;
                                    Seqno :          SEQUENCE_TYPE;
                                    Allseq:          boolean          );

        BY S_provider :

                Timer_response    ( Name  :          TIMER_TYPE;
                                    Seqno :          SEQUENCE_TYPE    );

END System_primitives;

(*******************************************************)

MODULE TS_user_module;

        TSAP : TS_primitives ( TS_user );

END TS_user_module;

PROCESS TS_user_process( TS_index : integer ) FOR TS_user_module;

.....

END TS_user_process;

(*******************************************************)

MODULE System_module;

        SEP : System_primitives ( S_provider );

END System_module;

PROCESS System_process ( Sys_index : integer ) FOR System_module;

....

END System_process;

(*******************************************************)

MODULE ATP_module;

        TCEP : ARRAY[TSAP_TYPE] OF TS_primitives ( TS_provider );
        NSAP : ARRAY[NCEP_TYPE] OF NS_primitives ( NS_user );
        SAPT : ARRAY[TSAP_TYPE] OF System_primitives ( S_user );
        SAPN : ARRAY[NCEP_TYPE] OF System_primitives ( S_user );

END ATP_module;

PROCESS ATP_process FOR ATP_module;

QUEUED  TCEP, NSAP;

(* Variable declarations *)

VAR
        tc        : TP_TABLE;
        nc        : NS_TABLE;
        data, temp         : DATA_TYPE;
        ndata              : NDATA_TYPE;
        pdu                : TPDU_TYPE;
        tid                : TSAP_ID_TYPE;
        nid                : NCEP_ID_TYPE;
        qkind              : Q_MISC_KIND;
        n                  : SEQUENCE_TYPE;
        reason             : REASON_TYPE;
        nsdu_len           : integer;

(* Primitive functions and procedures *)


PROCEDURE Add_request( VAR tc    : TP_MACHINE;
                           data  : DATA_TYPE );                    PRIMITIVE;

FUNCTION  Alloc_ref : REFERENCE_TYPE;                              PRIMITIVE;

PROCEDURE Concatenate_2_NSDU ( VAR nc    : NS_MACHINE;
                                   data  : DATA_TYPE       );      PRIMITIVE;

PROCEDURE Construct_AK( VAR packet      : DATA_TYPE;
                            cdt         : SEQUENCE_TYPE;
                            dref        : REFERENCE_TYPE;
                            seqno       : SEQUENCE_TYPE;
                            extenced    : boolean          );      PRIMITIVE;

PROCEDURE Construct_CC( VAR packet      : DATA_TYPE;
                            buf_m       : SEQUENCE_TYPE;
                            sref        : REFERENCE_TYPE;
                            dref        : REFERENCE_TYPE;
                            lsuf        : SUFFIX_TYPE;
                            fsuf        : SUFFIX_TYPE;
                            maxsz       : integer;
                            qos         : QOS_TYPE;
                            data        : DATA_TYPE        );      PRIMITIVE;

PROCEDURE Construct_CR( VAR packet      : DATA_TYPE;
                            buf_m       : SEQUENCE_TYPE;
                            sref        : REFERENCE_TYPE;
                            lsuf        : SUFFIX_TYPE;
                            fsuf        : SUFFIX_TYPE;
                            maxsz       : integer;
                            qos         : QOS_TYPE;
                            data        : DATA_TYPE        );      PRIMITIVE;

PROCEDURE Construct_DC( VAR packet      : DATA_TYPE;
                            dref        : REFERENCE_TYPE;
                            sref        : REFERENCE_TYPE  ) ;      PRIMITIVE;

PROCEDURE Construct_DR( VAR packet      : DATA_TYPE;
                            dref        : REFERENCE_TYPE;
                            sref        : REFERENCE_TYPE;
                            reason      : REASON_TYPE;
                            data        : DATA_TYPE        );      PRIMITIVE;

PROCEDURE Construct_DT( VAR packet      : DATA_TYPE;
                            dref        : REFERENCE_TYPE;
                            eflag       : boolean;
                            seqno       : SEQUENCE_TYPE;
                            extended    : boolean;
                            data        : DATA_TYPE        );      PRIMITIVE;

PROCEDURE Construct_ERR( VAR packet     : DATA_TYPE;
                             dref       : REFERENCE_TYPE;
                             reason     : REASON_TYPE;
                             data       : DATA_TYPE        );      PRIMITIVE;

PROCEDURE Construct_XAK( VAR packet     : DATA_TYPE;
                             dref       : REFERENCE_TYPE;
                             xseqno     : SEQUENCE_TYPE;
                             extended   : boolean          ).      PRIMITIVE;

PROCEDURE Construct_XPD( VAR packet     : DATA_TYPE;
                               dref         : REFERENCE_TYPE;
                               xseqno       : SEQUENCE_TYPE;
                               extended     : boolean;
                               data         : DATA_TYPE      );     PRIMITIVE;

FUNCTION Determine_TC( tc         : TP_TABLE;
                       data       : DATA_TYPE;
                       VAR pdu    : TPDU_TYPE ) : TSAP_ID_TYPE;     PRIMITIVE;

PROCEDURE Extract_NSDU( VAR nbuffer : NBUFFER_PTR;
                        VAR ndata   : NDATA_TYPE          );        PRIMITIVE;

PROCEDURE Extract_TPDU(     nsdata : NDATA_TYPE;
                        VAR tpdata : DATA_TYPE;
                        VAR nslen  : integer     );                 PRIMITIVE;

PROCEDURE Extract_TSDU(     buffer : BUFFER_TYPE;
                        VAR tsdu   : DATA_TYPE;
                        VAR count  : SEQUENCE_TYPE        );        PRIMITIVE;

PROCEDURE Get_net_addr( VAR naddr : NADDR_TYPE;
                            taddr : ADDR_TYPE ) ;                   PRIMITIVE;

PROCEDURE Merge( VAR buffer     : BUFFER_PTR;
                     pdu        : TPDU_TYPE     );                  PRIMITIVE;

PROCEDURE Release( VAR buffer     : BUFFER_PTR;
                       seqno      : SEQUENCE_TYPE;
                       kind       : PDU_KIND;
                       allseq     : boolean         );              PRIMITIVE;

PROCEDURE Release_all( VAR tc     : TP_MACHINE;
                       VAR nc     : NS_MACHINE );                   PRIMITIVE;

PROCEDURE Resume_data( VAR tc     : TP_MACHINE;
                       VAR nc     : NS_MACHINE );                   PRIMITIVE;

PROCEDURE Resume_xdata( VAR tc    : TP_MACHINE;
                        VAR nc    : NS_MACHINE );                   PRIMITIVE;

PROCEDURE Retrieve( buffer      : BUFFER_PTR;
                    seqno       : SEQUENCE_TYPE;
                    kind        : PDU_KIND;
                    VAR data    : DATA_TYPE      );                 PRIMITIVE;

FUNCTION  Same_naddr( naddr1, naddr2 : NADDR_TYPE ) : boolean;      PRIMITIVE;

PROCEDURE Store( VAR buffer     : BUFFER_PTR;
                     data       : DATA_TYPE;
                     seqno      : SEQUENCE_TYPE;
                     kind       : PDU_KIND        );                PRIMITIVE;

FUNCTION  SEQ_ADD  ( seq1, seq2  : SEQUENCE_TYPE;
                    extended     : boolean ) : SEQUENCE_TYPE;       PRIMITIVE;

FUNCTION  SEQ_MINUS  ( seq1, seq2 : SEQUENCE_TYPE;
                      extended    : boolean ) : SEQUENCE_TYPE;      PRIMITIVE;

PROCEDURE Uncode  ( VAR pdu      : TPDU_TYPE;
                        ndata    : NDATA_TYPE;
                        extended : boolean );                       PRIMITIVE;

(*****************************************************************************)

FUNCTION Acceptable_CC( qos      : QOS_TYPE;
                        sref     : REFERENCE_TYPE;
                        pdu      : TPDU_TYPE      ) : boolean;
BEGIN

  Acceptable_CC := TRUE;

  if ( pdu.version       <> VERSION )      or
     ( pdu.data.dlen     >  MAX_CRCC_SZ ) or
     ( pdu.maxsz         =  0     )
   then
```

```
        Acceptable_CC    := FALSE.

  if ( pdu.qos.class <> qos.class ) and ( pdu.ocls <> qos.class )
    then
        Acceptable_CC := FALSE;

        if ( NOT qos.misc[Q_NO_EXPEDITED_DATA]   and
             pdu.qos.misc[Q_NO_EXPEDITED_DATA]   )         or
           ( NOT qos.misc[Q_CHECKSUM_IN_USE]      and
             pdu.qos.misc[Q_CHECKSUM_IN_USE]      )         or
           ( NOT qos.misc[Q_NO_FLOW_CONTROL]      and
             pdu.qos.misc[Q_NO_FLOW_CONTROL]      )         or
           ( NOT qos.misc[Q_EXTENDED_FORMAT]      and
             pdu.qos.misc[Q_EXTENDED_FORMAT]      )
          then
            Acceptable_CC := FALSE;

  if pdu.dref <> sref then Acceptable_CC := FALSE

END;

(*****************************************************************)

FUNCTION Acceptable_CR( qos : QOS_TYPE;
                        pdu : TPDU_TYPE ) : boolean;

VAR
        qkind : Q_MISC_KIND;

BEGIN

  Acceptable_CR := TRUE;

  if ( pdu.version       <> VERSION )      or
     ( pdu.data_dlen     >  MAX_CRCC_SZ )  or
     ( pdu.maxsz         =  0      )
    then
        Acceptable_CR    := FALSE;

  if ( pdu.qos.class <> qos.class ) and ( pdu.ccls  <> qos.class )
    then
        Acceptable_CR := FALSE;

        if ( NOT qos.misc[Q_NO_EXPEDITED_DATA]  and
             pdu.qos.misc[Q_NO_EXPEDITED_DATA]  )         or
           ( NOT qos.misc[Q_CHECKSUM_IN_USE]     and
             pdu.qos.misc[Q_CHECKSUM_IN_USE]     )         or
           ( NOT qos.misc[Q_NO_FLOW_CONTROL]     and
             pdu.qos.misc[Q_NO_FLOW_CONTROL]     )         or
           ( NOT qos.misc[Q_EXTENDED_FORMAT]     and
             pdu.qos.misc[Q_EXTENDED_FORMAT]     )
          then
            Acceptable_CR := FALSE
END;

(*****************************************************************)

FUNCTION Choose_class( qos : QOS_TYPE ) : Q_CLASS_TYPE;
BEGIN
        Choose_class := qos.class
END;

(*****************************************************************)

PROCEDURE Construct_addr( VAR transport_addr : ADDR_TYPE;
                              suffix         : SUFFIX_TYPE;
                              net_addr       : NADDR_TYPE);
BEGIN
        transport_addr.suffix := suffix;
        transport_addr.prefix := net_addr
END;

(*****************************************************************)

FUNCTION Get_ncep( nc                   : NS_TABLE;
                   l_naddr, f_naddr : NADDR_TYPE ) : NCEP_ID_TYPE;
VAR
        nid     : NCEP_ID_TYPE;
        notdone : boolean;
BEGIN

        notdone := TRUE;
        nid     := 1;

        while notdone and ( nid <= MAX_NCEP_ID ) do
          begin
            if Same_naddr( f_naddr, nc[nid].f_net_addr )
            then begin
              notdone   := FALSE;
              Get_ncep := nid
            end
            else
              nid := nid + 1
          end;

        nid := 1;

        (* A new network connection is required *)

        while notdone and ( nid <= MAX_NCEP_ID ) do
          begin
            if nc[nid].state = NIDLE
              then begin
                notdone   := FALSE;
                Get_ncep := nid
              end
              else
                nid := nid + 1
          end;

        if notdone then Get_ncep := 0
END;

(*****************************************************************)

FUNCTION Get_suffix( transport_addr : ADDR_TYPE ) : SUFFIX_TYPE;
BEGIN

        Get_suffix := transport_addr.suffix
END;

(*****************************************************************)

FUNCTION Min( m, n : integer ) : integer;
BEGIN

        if m > n
          then Min := n
          else Min := m
END;

(*****************************************************************)

FUNCTION Nc_multiplexed( np  : NS_MACHINE ) : boolean;
BEGIN
        if np.link > 1
          then Nc_multiplexed := TRUE
          else Nc_multiplexed := FALSE
END;

(*****************************************************************)

FUNCTION New_nc_required( nc              : NS_TABLE;
                          laddr, faddr    : ADDR_TYPE ) : boolean;
VAR
        nid     : NCEP_ID_TYPE;
        notdone : boolean;
BEGIN
        notdone := TRUE;
        nid       := 1;

        while notdone and ( nid <= MAX_NCEP_ID ) do
          begin
            if ( nc[nid].state <> NIDLE ) and
               Same_naddr( nc[nid].f_net_addr, taddr.prefix )
              then begin
                notdone          := FALSE;
                New_nc_required  := FALSE
              end
              else
                nid := nid + 1
          end;

        if notdone then New_nc_required := TRUE
END;

(*****************************************************************)

FUNCTION  Size( data : DATA_TYPE ) : integer;
BEGIN
        Size := data.dlen
END;


(* Initialization *)

INITIALIZE
BEGIN
        for tid := 1 to MAX_TSAP_ID do
          begin
            tc[tid].state          := CLOSED;
            tc[tid].ncep_id        := 0;
            tc[tid].src_ref        := UNDEFINED_REFERENCE;
            tc[tid].dst_ref        := UNDEFINED_REFERENCE;

            tc[tid].snd_upper_edge         := 0;
            tc[tid].snd_seq                := 0;
            tc[tid].snd_una                := 0;
            tc[tid].snd_nxt                := 0;

            tc[tid].rcv_nxt                := 0;
            tc[tid].rcv_upper_edge         := DEF_BUFFER_M;

            tc[tid].x_seq                  := 0;
            tc[tid].x_nxt                  := 0;
            tc[tid].xsnd_nxt               := 0;
            tc[tid].x_una                  := 0;

            for qkind := Q_NO_EXPEDITED_DATA to Q_EXTENDED_FORMAT do
               tc[tid].qual_of_service.misc[qkind]        := FALSE;

            tc[tid].qual_of_service.class          := CLASS_TWO;

            tc[tid].reason                 := NORMAL;

            tc[tid].max_TPDU_size          := DEF_TPDU_SZ;
            tc[tid].DT_maxlen              := DEF_TPDU_SZ - NOR_DT_HEADER_SZ;

            tc[tid].sbuf          := NIL;
            tc[tid].rbuf          := NIL;
            tc[tid].xbuf          := NIL
          end;

        for nid := 1 to MAX_NCEP_ID do
          begin
            nc[nid].state         := NIDLE;
            nc[nid].link          := 0;
            nc[nid].nqos          := CLASS_TWO;
            nc[nid].sbuf          := NIL;
            nc[nid].rbuf          := NIL
          end

END;    (* Initialization *)
```

```
(* Transitions *)

TRANS
        WHEN TCEP[tid].T_CONNECT_request                    (* Transition 1 *)
          PROVIDED ( ( tc[tid].state = CLOSED ) and
                     ( New_nc_required( nc, From_transport_addr,
                                        To_transport_addr ) )          and
                     ( Choose_class( Qual_of_service ) = CLASS_TWO ) and
                     ( Size( TS_user_data ) <= MAX_CRCC_SZ        ) )
        BEGIN
          tc[tid].state := CALLING;

          tc[tid].local_addr     := From_transport_addr;
          tc[tid].remote_addr    := To_transport_addr;

          tc[tid].l_suffix       := Get_suffix( From_transport_addr );
          tc[tid].f_suffix       := Get_suffix( To_transport_addr );

          Get_net_addr( tc[tid].l_net_addr, From_transport_addr );
          Get_net_addr( tc[tid].f_net_addr, To_transport_addr );

          nid    := Get_ncep( nc, tc[tid].l_net_addr, tc[tid].f_net_addr );

          tc[tid].ncep_id        := nid;

          tc[tid].qual_of_service := Qual_of_service;

          Store(tc[tid].sbuf, TS_user_data, 0, 0);
(**)
          nc[nid].state          := NWAITING;
          nc[nid].l_net_addr     := tc[tid].l_net_addr;
          nc[nid].f_net_addr     := tc[tid].f_net_addr;
          nc[nid].link           := 1;
          nc[nid].nqos           := Qual_of_service.class;

          OUT NSAP[nid].N_CONNECT_request ( nc[nid].l_net_addr,
                                            nc[nid].f_net_addr,
                                            nc[nid].nqos )
        END;

TRANS
        WHEN NSAP[nid].N_CONNECT_confirm
          PROVIDED nc[nid].state = NWAITING
        BEGIN
          nc[nid].state := NOPEN;

          for tid := 1 to MAX_TSAP_ID do
            begin
              if (tc[tid].ncep_id = nid) and
                 (tc[tid].state = CALLING)                 (* Transition 2 *)
                then begin
                  tc[tid].state          := CR_SENT;

                  tc[tid].src_ref        := Alloc_ref;
                  Retrieve(tc[tid].sbuf, 0, 0, temp);
                  Release(tc[tid].sbuf, 0, 0, TRUE);

                  Construct_CR( data,    tc[tid].rcv_upper_edge,
                                         tc[tid].src_ref,
                                         tc[tid].l_suffix,
                                         tc[tid].f_suffix,
                                         tc[tid].max_TPDU_size,
                                         tc[tid].qual_of_service,
                                         temp );

                  Concatenate_2_NSDU ( nc[nid], data )
                end      (* if CALLING *)
            end          (* for loop *)
        END;

TRANS
        WHEN TCEP[tid].T_CONNECT_request                    (* Transition 3 *)
          PROVIDED ( ( tc[tid].state = CLOSED )                          and
                     ( NOT New_nc_required( nc, From_transport_addr,
                                            To_transport_addr ) )        and
                ( Choose_class( Qual_of_service ) = CLASS_TWO )        and
                ( Size( TS_user_data ) <= MAX_CRCC_SZ        ) )
        BEGIN
          tc[tid].state := CR_SENT;

          tc[tid].local_addr    := From_transport_addr;
          tc[tid].remote_addr   := To_transport_addr;

          tc[tid].l_suffix        := Get_suffix( From_transport_addr );
          tc[tid].f_suffix        := Get_suffix( To_transport_addr );

          Get_net_addr( tc[tid].l_net_addr, From_transport_addr );
          Get_net_addr( tc[tid].f_net_addr, To_transport_addr );

          nid := Get_ncep( nc, tc[tid].l_net_addr, tc[tid].f_net_addr );

          tc[tid].ncep_id          := nid;
(**)
          nc[nid].link             := nc[nid].link + 1;
(**)
          tc[tid].qual_of_service := Qual_of_service;

          tc[tid].src_ref          := Alloc_ref;

          Construct_CR( data,      tc[tid].rcv_upper_edge,
                                   tc[tid].src_ref,
                                   tc[tid].l_suffix,
                                   tc[tid].f_suffix,
                                   tc[tid].max_TPDU_size,
                                   tc[tid].qual_of_service,
                                   TS_user_data    );

          Concatenate_2_NSDU ( nc[nid], data )
        END;

TRANS
        WHEN NSAP[nid].N_DATA_indication
          PROVIDED nc[nid].state = NOPEN
        BEGIN
          nsdu_len := 0;

          while ( nsdu_len < NS_user_data.dlen ) do
          begin
            Extract_TPDU( NS_user_data, data, nsdu_len );
            tid := Determine_TC( tc, data, pdu );

            if tid <> 0
            then begin
              Uncode(pdu,data,tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT]);

              if pdu.kind = CR   (* CR TPDU *)
              then begin
                if tc[tid].state = CLOSED
                then begin                                  (* transition 4 *)
                  if Acceptable_CR( tc[tid].qual_of_service, pdu )
                  then begin
                    tc[tid].state          := CR_RCVD;

                    OUT SAPN[nid].Timer_cancel( INCOMING_NC, 0, TRUE );

                    tc[tid].f_suffix     := pdu.lsuf;
                    tc[tid].l_suffix     := pdu.fsuf;

                    tc[tid].f_net_addr  := nc[nid].f_net_addr;
                    tc[tid].l_net_addr  := nc[nid].l_net_addr;
                    tc[tid].ncep_id      := nid;

                    Construct_addr( tc[tid].local_addr,
                                    tc[tid].l_suffix,
                                    tc[tid].l_net_addr   );

                    Construct_addr( tc[tid].remote_addr,
                                    tc[tid].f_suffix,
                                    tc[tid].f_net_addr   );
          tc[tid].qual_of_service       := pdu.qos;
          tc[tid].max_TPDU_size         := Min( pdu.maxsz,
                                                tc[tid].max_TPDU_size);

          tc[tid].dst_ref               := pdu.sref;
          tc[tid].snd_upper_edge        := pdu.cdt;

          OUT TCEP[tid].T_CONNECT_indication( tc[tid].remote_addr,
                                              tc[tid].local_addr,
                                              pdu.qos,
                                              pdu.data           )

        end (* Acceptable CR *)
        else begin                            (* transition 5 *)
          tc[tid].dst_ref      := pdu.sref;
          tc[tid].reason       := NEGOTIATION_FAILED;

          Empty_data( temp );

          Construct_DR( data, tc[tid].dst_ref, 0,
                        tc[tid].reason, temp);

          Concatenate_2_NSDU( nc[nid], data )
        end    (* NOT Acceptable CR *)
      end      (* CLOSED *)
    end;       (* CR TPDU *)

    if pdu.kind = CC (* CC TPDU *)
    then begin
      if tc[tid].state = CR_SENT                  (* Transition 6 *)
      then begin
        if Acceptable_CC( tc[tid].qual_of_service,
                          tc[tid].src_ref,
                          pdu )
        then begin
          tc[tid].state         := ESTABLISHED;

          tc[tid].dst_ref               := pdu.sref;
          tc[tid].snd_upper_edge        := pdu.cdt;
          tc[tid].qual_of_service       := pdu.qos;
          tc[tid].max_TPDU_size         := pdu.maxsz;

          OUT TCEP[tid].T_CONNECT_confirm( pdu.qos, pdu.data )
        end                                   (* Acceptable CC *)
        else begin                            (* Transition 7 *)
          tc[tid].state        := CLOSING;

          tc[tid].dst_ref      := pdu.sref;

          tc[tid].reason       := NEGOTIATION_FAILED;

          Empty_data( temp );

          OUT TCEP[tid].T_DISCONNECT_indication( tc[tid].reason,
                                                 temp);

          Construct_DR( data, tc[tid].dst_ref, tc[tid].src_ref,
                        tc[tid].reason, temp);

          Concatenate_2_NSDU( nc[nid], data )
        end    (* NOT Acceptable CC *)
      end      (* CR_SENT *)
    end;       (* CC TPDU *)

    if pdu.kind = DT  (* DT TPDU *)
    then begin
      if tc[tid].state = ESTABLISHED              (* Transition 8 *)
      then begin
        if  ( pdu.seqno = tc[tid].rcv_nxt ) and
            ( pdu.seqno < tc[tid].rcv_upper_edge )
        then begin
(**)      Merge( tc[tid].rbuf, pdu );

          tc[tid].rcv_nxt := SEQ_ADD( tc[tid].rcv_nxt, 1,
              tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );
        if pdu.eflag          (* a complete TSDU in the buffer *)
        then begin
          Extract_TSDU( tc[tid].rbuf, data, n );
          Release( tc[tid].rbuf, tc[tid].rcv_nxt, DT, FALSE );

          (* update the upper edge of the receiving window *)

 tc[tid].rcv_upper_edge := SEQ_ADD(tc[tid].rcv_upper_edge, n,
           tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT]   );

          OUT TCEP[tid].T_DATA_indication( data );

          (* compute the current buffer space *)
          n := SEQ_MINUS( tc[tid].rcv_upper_edge, tc[tid].rcv_nxt,
                 tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

          Construct_AK( data, n, tc[tid].dst_ref, tc[tid].rcv_nxt,
            tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT]   );

          Concatenate_2_NSDU( nc[nid], data );

          if n = 0 then
            OUT SAPT[tid].Timer_request( WINDOW, WN_SYNC, 0 )
          else
            OUT SAPT[tid].Timer_cancel( WINDOW, 0, TRUE )
        end                   (* pdu.eflag *)
      end                     (* receivable DT *)
      else begin              (* Transition 9  *)
        tc[tid].reason := INVALID_TPDU;

        Construct_ERR( data, tc[tid].dst_ref,
                             tc[tid].reason,
                             pdu.data       );

        Concatenate_2_NSDU( nc[nid], data )
      end    (* NOT receivable DT *)
    end      (* ESTABLISHED *)
  end;       (* DT TPDU *)

  if pdu.kind = AK
  then begin
    if (( tc[tid].state = ESTABLISHED ) or
        ( tc[tid].state = CLOSING))      and (* Transition 10 *)
       ( pdu.seqno >= tc[tid].snd_una )
    then begin

      tc[tid].snd_una        := pdu.seqno;
      tc[tid].snd_upper_edge := SEQ_ADD( tc[tid].snd_upper_edge,
        pdu.cdt, tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

      Release( tc[tid].sbuf, tc[tid].snd_una, DT, FALSE );

      Resume_data( tc[tid], nc[nid] )
    end       (* ESTABLISHED and AK_ok *)
  end;        (* AK TPDU *)

  if pdu.kind = XPD
  then begin
    if tc[tid].state = ESTABLISHED
    then begin
      if pdu.seqno = tc[tid].x_nxt
      then begin                             (* Transition 11 *)
        OUT TCEP[tid].T_XPD_indication( pdu.data );

        Construct_XAK( data, tc[tid].dst_ref, tc[tid].x_nxt,
            tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

        Concatenate_2_NSDU( nc[nid], data );

        tc[tid].x_nxt := SEQ_ADD( tc[tid].x_nxt, 1,
              tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] )
      end
      else begin
        tc[tid].reason := INVALID_TPDU;
        Construct_ERR( data, tc[tid].dst_ref, tc[tid].reason,
                              pdu.data );
        Concatenate_2_NSDU( nc[nid], data )
      end
    end       (* ESTABLISHED and OK *)
  end;

  if pdu.kind = XAK
  then begin
    if (( tc[tid].state = ESTABLISHED ) or
        ( tc[tid].state = CLOSING))       and
       ( pdu.seqno = tc[tid].x_una )
    then begin                                      (* Transition 12 *)
      tc[tid].x_una  := tc[tid].xsnd_nxt;

      Resume_xdata( tc[tid], nc[nid] );
      Resume_data( tc[tid], nc[nid] )
    end
  end;

  if pdu.kind = ERR
  then begin                                        (* Transition 13 *)
    if ( ( tc[tid].state = CALLING ) or
         ( tc[tid].state = CR_SENT ) or
         ( tc[tid].state = CR_RCVD ) or
         ( tc[tid].state = ESTABLISHED ) or
         ( tc[tid].state = CLOSING ) )
    then begin
      tc[tid].state := CLOSING;

      Empty_data( temp );
      reason := PROTOCOL_ERROR;

      OUT TCEP[tid].T_DISCONNECT_indication( reason, temp );

      OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE );

      Construct_DR( data, tc[tid].dst_ref, tc[tid].src_ref,
                       reason, temp );

      Concatenate_2_NSDU( nc[nid], data )
    end       (* active connection *)
  end;        (* ERR TPDU       *)

  if pdu.kind = DR
  then begin
    if ( ( tc[tid].state = CR_RCVD ) or      (* Transition 14 *)
         ( tc[tid].state = CR_SENT ) or
         ( tc[tid].state = ESTABLISHED ) )
    then begin
      OUT TCEP[tid].T_DISCONNECT_indication( pdu.reason,
                                             pdu.data );

      Construct_DC( data, tc[tid].dst_ref, tc[tid].src_ref );

      Concatenate_2_NSDU( nc[nid], data );

      OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE );

      if Nc_multiplexed( nc[nid] )
      then begin
        tc[tid].state := CLOSED;

        Release_all( tc[tid], nc[nid] )
      end                      (* Nc_multiplexed *)
      else
        tc[tid].state         := DISCON_WAIT

    end      (* CR_SENT, CR_RCVD, ESTABLISHED *)
  end;       (* DR TPDU *)

  if tc[tid].state = CLOSING
  then begin
    if ( ( pdu.kind = DR ) or ( pdu.kind = DC ) )
```

```
          then begin                                         (* Transition 15 *)
            tc[tid].state := CLOSED;

            OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE );

            if NOT Nc_multiplexed( nc[nid] ) then
               OUT NSAP[nid].N_DISCONNECT_request;

            Release_all( tc[tid], nc[nid] )
          end       (* DR TPDU, DC TPDU *)
        end         (* CLOSING *)
      end           (* tid <> 0 *)
      else begin
        if (pdu.sref <> UNDEFINED_REFERENCE) and
           (pdu.kind <> DR) and (pdu.kind <> DC)
        then begin
          Empty_data( temp );

          Construct_DR( data, pdu.sref, 0, pdu.reason, temp);
          Concatenate_2_NSDU( nc[nid], data )
        end
      end
    end             (* while loop  *)
  END;

TRANS
  WHEN NSAP[nid].N_CONNECT_indication                 (* Transition 16 *)
    PROVIDED nc[nid].state = NIDLE
  BEGIN
    nc[nid].state := NOPEN;

    nc[nid].l_net_addr      := To_network_addr;
    nc[nid].f_net_addr      := From_network_addr;
    nc[nid].link            := 0;

    OUT NSAP[nid].N_CONNECT_response;
    OUT SAPN[nid].Timer_request( INCOMING_NC, NET_WAIT, 0 )
  END;

TRANS
  WHEN NSAP[nid].N_DISCONNECT_indication
    PROVIDED ( nc[nid].state = NOPEN )
  BEGIN
    nc[nid].state := NIDLE;                           (* Transition 17 *)

    if nc[nid].link > 0
    then begin
      for tid := 1 to MAX_TSAP_ID do
      begin
        if tc[tid].ncep_id = nid
        then begin                                    (* Transition 18 *)
          if ( ( tc[tid].state = CR_RCVD ) or
               ( tc[tid].state = CR_SENT ) or
               ( tc[tid].state = ESTABLISHED ) )
          then begin
            tc[tid].state          := CLOSED;

            Empty_data( data );

            OUT TCEP[tid].T_DISCONNECT_indication(
                                      LOSS_OF_NETWORK_CONNECTION,
                                      data );
            OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE );
            Release_all( tc[tid], nc[nid] )
          end;     (* CR_RCVD, CR_SENT, ESTABLISHED *)

          if tc[tid].state = CALLING                  (* Transition 19 *)
          then begin
            tc[tid].state          := CLOSED;

            Empty_data( data );

            OUT TCEP[tid].T_DISCONNECT_indication(
                                      NETWORK_CONNECT_FAILED,
                                      data );
```

```
                  OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE );
                  Release_all( tc[tid], nc[nid] )
                end;     (* CALLING *)

                if tc[tid].state = CLOSING                     (* Transition 20 *)
                then begin
                  tc[tid].state           := CLOSED;

                  if Reason <> NORMAL
                  then begin
                     Empty_data( data );
                     OUT TCEP[tid].T_DISCONNECT_indication(
                                                LOSS_OF_NETWORK_CONNECTION,
                                                data )
                  end;
                  Release_all( tc[tid], nc[nid] )
                end;     (* CLOSING *)

                if tc[tid].state = DISCON_WAIT                 (* Transition 21 *)
                then begin
                  tc[tid].state           := CLOSED;

                  Release_all( tc[tid], nc[nid] )
                end      (* DISCON_WAIT *)
              end        (* matching transport connection *)
            end (* for loop *)
          end   (* link > 0 *)
          else
            OUT SAPN[nid].Timer_cancel( INCOMING_NC, 0, TRUE )
        END;

TRANS
        WHEN TCEP[tid].T_CONNECT_response                      (* Transition 22 *)
          PROVIDED ( ( tc[tid].state = CR_RCVD ) and
                     ( Choose_class( Qual_of_service ) = CLASS_TWO ) and
                     ( Size( TS_user_data ) <= MAX_CRCC_SZ        ) )
        BEGIN
          tc[tid].state := ESTABLISHED;

          tc[tid].src_ref                 := Alloc_ref;
          tc[tid].qual_of_service         := Qual_of_service;

          Construct_CC( data,     tc[tid].rcv_upper_edge,
                                  tc[tid].src_ref,
                                  tc[tid].dst_ref,
                                  tc[tid].l_suffix,
                                  tc[tid].f_suffix,
                                  tc[tid].max_TPDU_size,
                                  tc[tid].qual_of_service,
                                  TS_user_data    );

          nid    := tc[tid].ncep_id;
          Concatenate_2_NSDU ( nc[nid], data )
        END;

TRANS
        WHEN TCEP[tid].T_DISCONNECT_request                    (* Transition 23 *)
          PROVIDED tc[tid].state = CR_RCVD
        BEGIN
          tc[tid].state := DISCON_WAIT;

          tc[tid].src_ref         := Alloc_ref;
          tc[tid].reason          := CONN_REJECT;

          Construct_DR( data,    tc[tid].dst_ref,
                                 tc[tid].src_ref,
                                 tc[tid].reason,
                                 TS_user_data     );

          nid    := tc[tid].ncep_id;
          Concatenate_2_NSDU ( nc[nid], data );

          Release_all( tc[tid], nc[nid] )
        END;

          PROVIDED ( ( tc[tid].state = CR_SENT ) or        (* Transition 24 *)
                     ( tc[tid].state = ESTABLISHED ) )
        BEGIN
            tc[tid].state  := CLOSING;

            tc[tid].reason := NORMAL;

            Construct_DR( data,    tc[tid].dst_ref,
                                   tc[tid].src_ref,
                                   tc[tid].reason,
                                   TS_user_data  );

            Store( tc[tid].sbuf, data, 0, DR );

            nid := tc[tid].ncep_id;
            Resume_data( tc[tid], nc[nid] );
            OUT SAPT[tid].Timer_cancel( ALL_TIMER, 0, TRUE )

        END;

TRANS
        WHEN TCEP[tid].T_DATA_request                      (* Transition 25 *)
          PROVIDED tc[tid].state = ESTABLISHED
        BEGIN
          Add_request( tc[tid], TS_user_data );

          nid   := tc[tid].ncep_id;
          Resume_data( tc[tid], nc[nid] )
        END;

TRANS
        WHEN TCEP[tid].T_XPD_request                       (* Transition 26 *)
          PROVIDED ( tc[tid].state = ESTABLISHED ) and
                   ( Size( TS_user_data ) <= MAX_XTSDU_32 )
        BEGIN
          Construct_XPD( data, tc[tid].dst_ref, tc[tid].x_seq,
                         tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT],
                         TS_user_data );

          tc[tid].x_seq := SEQ_ADD( tc[tid].x_seq, 1,
                         tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

          Store( tc[tid].xbuf, data, tc[tid].xsnd_nxt, XPD );

          nid := tc[tid].ncep_id;
          Resume_xdata( tc[tid], nc[nid] )
        END;

TRANS
        WHEN SAPN[nid].Timer_response                      (* Transition 27 *)
          PROVIDED ( Name = INCOMING_NC ) and
                   ( nc[nid].state = NOPEN )
        BEGIN
          nc[nid].state := NIDLE;

          OUT NSAP[nid].N_DISCONNECT_request
        END;

TRANS
        WHEN SAPT[tid].Timer_response                      (* Transition 28 *)
          PROVIDED ( Name = WINDOW ) and
                   ( tc[tid].state = ESTABLISHED )
        BEGIN
          n := SEQ_MINUS( tc[tid].rcv_upper_edge, tc[tid].rcv_nxt,
                          tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

          if n > 0 then
          begin
            Construct_AK( data, n, tc[tid].dst_ref, tc[tid].rcv_nxt,
                        tc[tid].qual_of_service.misc[Q_EXTENDED_FORMAT] );

            nid := tc[tid].ncep_id;
            Concatenate_2_NSDU( nc[nid], data );

            OUT SAPT[tid].Timer_cancel( WINDOW, 0, TRUE )
          end
          else
            OUT SAPT[tid].Timer_request( WINDOW, WN_SYNC, 0 )
        END;

(* spontaneous transition -- Send the network data anyway *)

TRANS
        PROVIDED TRUE
        BEGIN
          for nid := 1 to MAX_NCEP_ID do
          begin
            if nc[nid].sbuf <> NIL
            then begin
              Extract_NSDU( nc[nid].sbuf, ndata );
              OUT NSAP[nid].N_DATA_request( ndata )
            end
          end
        END;

END ATP_process;

(************************************************************)

MODULE RS_module;

        NCEP : NS_primitives ( NS_provider );

END RS_module;

PROCESS RS_process ( RS_index : integer ) FOR RS_module;

QUEUED  NCEP;

VAR
        rs_id              : integer;
        local, remote      : NADDR_TYPE;
        qos                : NQOS_TYPE;
        reason             : REASON_TYPE;
        data               : NDATA_TYPE;

(* Primitive functions and procedures   *)

FUNCTION  Net_accept(      rs_id           : integer;
                     VAR local, remote : NADDR_TYPE;
                     VAR qos           : NQOS_TYPE ) : boolean;PRIMITIVE;

PROCEDURE Net_close( rs_id        : integer );                     PRIMITIVE;

FUNCTION  Net_confirm(  rs_id   : integer ) : boolean;              PRIMITIVE;

PROCEDURE Net_connect(  rs_id   : integer;
                        local   : NADDR_TYPE;
                        remote  : NADDR_TYPE;
                        qos     : NQOS_TYPE );                      PRIMITIVE;

FUNCTION  Net_disconnect(      rs_id  : integer;
                           VAR reason : REASON_TYPE ) : boolean; PRIMITIVE;

FUNCTION  Net_recv(       rs_id   : integer;
                      VAR data    : NDATA_TYPE ) : boolean;         PRIMITIVE;

PROCEDURE Net_send( rs_id        : integer;
                    data         : NDATA_TYPE );                    PRIMITIVE;

INITIALIZE
BEGIN
        rs_id   := RS_index
END;    (* Initialization *)


TRANS
```

```
        WHEN NCEP.N_CONNECT_request
        BEGIN
          local  := From_network_addr;
          remote := To_network_addr;
          qos    := QOS;

          Net_connect( rs_id, local, remote, qos )
        END;

TRANS
        WHEN NCEP.N_CONNECT_response
        BEGIN
                (* Do nothing : the Network primitives handle it *)
        END;

TRANS
        WHEN NCEP.N_DATA_request
        BEGIN
          data := NS_user_data;
          Net_send( rs_id, data )
        END;

TRANS
        WHEN NCEP.N_DISCONNECT_request
        BEGIN
          Net_close( rs_id )
        END;

TRANS
        PROVIDED Net_accept( rs_id, local, remote, qos )
        BEGIN
          OUT NCEP.N_CONNECT_indication( remote, local, qos )
        END;

TRANS
        PROVIDED Net_confirm( rs_id )
        BEGIN
          OUT NCEP.N_CONNECT_confirm
        END;
TRANS
        PROVIDED Net_recv( rs_id, data )
        BEGIN
          OUT NCEP.N_DATA_indication( data )
        END;

TRANS
        PROVIDED Net_disconnect( rs_id, reason )
        BEGIN
          OUT NCEP.N_DISCONNECT_indication( reason )
        END;

END RS_process;

(**********************************************************)

U1:  TS_user_module with TS_user_process(1);
U2:  TS_user_module with TS_user_process(2);

ATP: ATP_module with ATP_process;

S1: System_module with System_process(1);
S2: System_module with System_process(2);
S3: System_module with System_process(3);
S4: System_module with System_process(4);

RS1:  RS_module with RS_process(1);
RS2:  RS_module with RS_process(2);

(**********************************************************)

CONNECT

U1.TSAP TO ATP.TCEP[1];
U2.TSAP TO ATP.TCEP[2];
```

```
ATP.NSAP[1] TO RS1.NCEP;
ATP.NSAP[2] TO RS2.NCEP;

ATP.SAPT[1] TO S1.SEP;
ATP.SAPT[2] TO S2.SEP;
ATP.SAPN[1] TO S3.SEP;
ATP.SAPN[2] TO S4.SEP;

END Transport_ref;
```

Appendix C

System Initialization and Scheduler — For Semi-Automatic Implementation

```
/*-----------------------------------------------------------------------
 *  fdtutil.c - system_init, schedule
 *-----------------------------------------------------------------------
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <sys/timeb.h>
#include <sys/time.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <strings.h>

#include "../inet/inet.h"
#include "listdefs.h"
#include "defs.h"
#include "tpdefs.h"
#include "../tsp/tsp.h"

/* Define the outermost refinement name */
#define REF_NAME ioTransport_ref

extern int signal_pending;
extern struct process_block *p_block;
extern NCONN    conn[];

/*-----------------------------------------------------------------------*
  This routine causes the generation of the data structure and then checks
  for dangling channel connections.
 *-----------------------------------------------------------------------*/

struct process_block *system_init()
{
    struct process_block *ptr, *process_list, *remove_header();
    struct channel_block *c_ptr;
    struct itimerval        value;
    int     i, j;

    /* user included dcl */
    struct process_block    *REF_NAME();

    process_list = remove_header(REF_NAME(NULL));

    for (ptr = process_list; ptr != NULL; ptr = ptr->next)
      {
        if (strcmp(ptr->p_ident, "TS_user_process") == 0)
          {
            i = ptr->lvars.s_TS_user_process.user_id - 1;
            uprocess[i] = ptr;
          }

        if (strcmp(ptr->p_ident, "System_process") == 0)
          {
            i = ptr->lvars.s_System_process.sys_id - 1;
            sprocess[i] = ptr;
          }

        if (strcmp(ptr->p_ident, "RS_process") == 0)
          {
            i = ptr->lvars.s_RS_process.rs_id - 1;
            rprocess[i] = ptr;
          }

        for (c_ptr = ptr->chan_list; c_ptr != NULL; c_ptr = c_ptr->next)
          {
            if (c_ptr->target_channel == NULL)
              {
                /* oops a dangling connection */

                fprintf(stderr,"\nSYSTEM INITIALIZATION ERROR: dangling");
                fprintf(stderr," channel in an instance of \"%s\". ",
                                                   ptr->p_ident);
                fprintf(stderr,"channel number %d, ind %d\n",c_ptr->c_id,
                                                          c_ptr->index_num);
             }
          }
      }

     /* join the ends of the process list into a loop */

     for (ptr = process_list; ptr->next != NULL; ptr = ptr->next);

     ptr->next =         process_list;

/*-------------------------------------------------------------------------
 * Establishing connection to the system environment
 *-------------------------------------------------------------------------
 */
    /* fire up a clock */
    value.it_interval.tv_sec  = value.it_value.tv_sec  = 1l;
    value.it_interval.tv_usec = value.it_value.tv_usec = 0l;

    setitimer(ITIMER_VIRTUAL, &value, (struct itimerval *)0);
    signal(SIGVTALRM, clock);

    /* set up timer lists */
    for (i = 0; i < NTIMER; i++)          timerlist[i] = NULL;

    /* open a network listener */
    if ((i = N_open(&conn[0], NSNAME)) != NET_OK)
      {
        fprintf(stderr,">>> N_open problem %d\n",i);
        exit(1);
      }
    netmask    = (1 << conn[0]->socket);
    nc_inuse = 0;

    for (i = 0; i < MAX_NCEP_ID; i++)
      {
        netpool[i].fill = FALSE;
        netreason[i]     = NET_OK;
        net_status[i]    = NET_NORMAL;
      }

    /* open a UNIX listen socket */
    if ((usock[0] = TS_open(TSNAME)) < 0)
      {
        fprintf(stderr,">>> TS_open problem %d\n",usock[0]);
        exit(1);
      }

    /* establish the inter-process connections */
    for (usermask = 0,i = 1; i <= MAX_TSAP_ID; i++)
      {
        struct sockaddr_un from;
        int len = sizeof(struct sockaddr_un);

        j = i - 1;
        userpool[j].fill = FALSE;
        errclose[j]       = FALSE;

        usock[i] = accept(usock[0], (struct sockaddr *)&from, &len);
        if (usock[i] < 0)
          {
            perror("UNIX domain : accept");
            exit(1);
          }
        usermask |= (1 << usock[i]);

      }
    close(usock[0]);     /* close the listener */

    return(process_list);
}
```

```
/*-----------------------------------------------------------------*
  This is the main driving routine.
 *-----------------------------------------------------------------*/

schedule(process_list)
struct process_block *process_list;
{
    extern int signal_pending;
    extern struct process_block *p_block;

    struct channel_block *c_ptr;
    struct signal_block  *s_ptr, *get_signal();
    struct process_block *p_ptr, *p_ptr2;
    struct timer_block   *tptr1, *tptr2;
    struct timeval        timeout;
    int                   i, j, n, mask, notdone;

    p_ptr  = p_ptr2 = process_list;
    c_ptr  = p_ptr->chan_list;
    signal_pending = 0;

    while ((usock[1] != -1) || (usock[2] != -1)) /* while there is a user */
      {
        if (signal_pending > 0)
          {
            s_ptr = get_signal(&c_ptr, &p_ptr);
            signal_pending--;

            /* call the transition routine */
            p_block = p_ptr;
            (*(p_ptr->proc_ptr))(c_ptr,s_ptr);

            /* move on to the next channel for the start of the next search */
            if (c_ptr->next != NULL)
                c_ptr = c_ptr->next;
            else
              {
                p_ptr = p_ptr->next;
                c_ptr = p_ptr->chan_list;
              }
          }     /* internal input signal pending */

    /* spontaneous transitions are handled below          */

    /* TS_user_process  */
    mask = usermask;
    timeout.tv_sec  = 0;
    timeout.tv_usec = 501;

    if (select(16, &mask, 0, 0, &timeout) < 0)
      {
        perror("UNIX domain : select");
        exit(2);
      }

    for (i = 1;(mask > 0) && (i <= MAX_TSAP_ID); i++)
      {
        if ((usock[i] != -1) &&
            (mask & (1 << usock[i]))) /* Incoming request */
          {
            j = i - 1;
            if ((n = TD_input(usock[i], userpool[j].datum, TS_MAX_LENGTH+2))
                        >= sizeof(struct data_hdr))
              {
                userpool[j].fill = TRUE;
                userpool[j].len  = n;
              }
            else
              {
                errclose[j] = TRUE;
              }
            p_block = uprocess[j];
            (*(p_block->proc_ptr))(NULL, NULL);
          }
  }

/* RS_process          */
mask = netmask;
timeout.tv_sec   = 0;
timeout.tv_usec  = 501;

if (select(16, &mask, 0, 0, &timeout) < 0)
  {
    perror("INET domain : select");
    exit(2);
  }

if ((mask > 0) && (nc_inuse < MAX_NCEP_ID)) /* network channel available */
  {
    if (mask & (1 << conn[0]->socket))
      {
        for (notdone = TRUE, i = MAX_NCEP_ID; notdone && (i > 0); i--)
          {
            if (conn[i] == NULL)
              {
                notdone = FALSE;
                j = i - 1;
                if (N_accept(&conn[i], conn[0]->socket) == NET_OK)
                  {
                    net_status[j] = NET_NEWCOMER;
                    nc_inuse++;
                    netmask |= (1 << (conn[i]->socket));
                    p_block = rprocess[j];
                    (*(p_block->proc_ptr))(NULL, NULL);
                  } /* if acceptance is OK */
              }     /* if conn is available */
          }         /* for loop */
      }             /* new comer arrives */
  }                 /* nc_inuse */

for (i = 1; i <= MAX_NCEP_ID; i++)
  {
    j = i - 1;
    if ((conn[i] != NULL) &&              /* the network channel is inuse */
        (mask & (1 << conn[i]->socket))) /* Incoming request */
      {
        if ((n = N_receive(conn[i], netpool[j].datum, NET_DATA_SIZE))
               >= NET_OK)
          {
            netpool[j].fill = TRUE;
            netpool[j].len  = n;
          }
        else
          {
            netreason[j] = n;
          }

        p_block = rprocess[j];
        (*(p_block->proc_ptr))(NULL, NULL);
      }

    if ((netreason[j] != NET_OK) || (net_status[j] == NET_CONFIRM))
      {
        p_block = rprocess[j];
        (*(p_block->proc_ptr))(NULL, NULL);
      }
  } /* for i-loop */

/* ATP process */
if (strcmp(p_ptr2->p_ident,"ATP_process") == 0)
  {
    p_block = p_ptr2;
    (*(p_ptr2->proc_ptr))(NULL,NULL);
  }

p_ptr2 = p_ptr2->next;

/* System_process */
for (i = 0; i < NTIMER; i++)
```

```
         {
           for (tptr1 = timerlist[i]; tptr1 != NULL; tptr1 = tptr1->next)
             {
               for (tptr2 = tptr1; tptr2 != NULL; tptr2 = tptr2->next)
                 {
                   if (tptr2->time [illegible] 0) /* timer expired  */
                     {
                       p_block = [illegible];
                       (*(p_block->proc_ptr))(NULL, NULL);
                     } /* time out */
                 } /* each timer    */
             } /* each timer list  */
         } /* each system process  */
      }   /* forever loop          */
}
```

Appendix D

System Initialization and Scheduler — For Manual Implementation

```
/*--------------------------------------------------------------------------
 *                    TS_initsys           (private)
 *                    TS_schedule          (private)
 *--------------------------------------------------------------------------
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <sys/time.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include "../inet/inet.h"
#include "tpdefs.h"
#include "tp.h"
#include "../tsp/tsp.h"
#include "tpvar.h"

/*--------------------------------------------------------------------------
 *      TS_initsys
 *      Transport Station initialization ( GLOBAL )
 *
 *      1. Handling the SIGINT & SIGCHLD signals
 *      2. Initialize TS, TP and NP queues.
 *      3. Open TS listener.
 *      4. Open NP listener.
 *--------------------------------------------------------------------------
 */

static
TS_initsys()
{
    struct itimerval     time;
    int       n;

    time.it_interval.tv_sec  = time.it_value.tv_sec  = 1l;
    time.it_interval.tv_usec = time.it_value.tv_usec = 0l;

    setitimer(ITIMER_VIRTUAL, &time, (struct itimerval *)0);
    signal(SIGVTALRM, TM_clock);

    /*
     *  Open a NSNAME server listening to the NP providers.
     */

    if ((n = N_open(&(nplist.nconn), NSNAME)) != NET_OK)
      {
        fprintf(stderr,">>> N_open problem %d\n",n);
        exit(1);
      }

    /*
     *  Open a TSNAME server listening to the TS users.
     */

    if ((tslist.tsap = TS_open(TSNAME)) < 0)
      {
        fprintf(stderr,">>> TS_open problem %d\n",tslist.tsap);
        exit(1);
      }

    /*
     *  Initialize the global queues.
     */

    tslist.prev = tslist.next = &tslist;
    tplist.prev = tplist.next = &tplist;
    nplist.prev = nplist.next = &nplist;

}
```

```
/*---------------------------------------------------------------------------
 *      TS_schedule      (private)
 *      This is the scheduler of the interactions
 *---------------------------------------------------------------------------
 */

static
TS_schedule()
  {
    struct timeval time;
    struct sockaddr_un  from;
    int     n, flen, mask, sock;
    char    datum[TS_MAX_LENGTH+2], ndatum[NET_DATA_SIZE];

    TSCONN tsp, tspnext;
    TPCONN tp, tpnext;
    NPCONN np, npnext;
    NDATA_PTR nptr;
    NCONN  nconn;

    for (;;)
      {
    /*  Transport service users */
        mask = TS_buildmask();
        time.tv_sec  = 0;
        time.tv_usec = 5001;

        if ((n = select(16, &mask, 0, 0, &time)) < 0)
            if (errno != EINTR)     TS_errorshutdown();

        if (n > 0)
          {
            if (mask & (1 << (tslist.tsap)))
              {
                flen = sizeof(struct sockaddr_un);
                if ((sock = accept((tslist.tsap), (struct sockaddr *)&from,
                                        &flen)) < 0)
                    TS_errorshutdown();
                else
                  {
                    if (TS_newuser(sock) == NULL)
                      {
                        shutdown(sock, 2);
                        close(sock);
                      }
                  }
              }    /* new TS user */

            for (tsp = tslist.next; tsp != &tslist; tsp = tspnext)
              {
                tspnext = tsp->next;
                if (mask & (1 << tsp->tsap))
                  {
                    if ((n = TD_input(tsp->tsap, datum, TS_MAX_LENGTH+2))
                            < sizeof(struct data_hdr))
                        TS_disconnect(tsp, UNKNOWN_ERROR);
                    else
                        (void)TS_input(tsp, datum, n);
                  }
              }    /* for tslist */
          }     /* n > 0 */

    /* Send the filled network outgoing buffers */
        for (np = nplist.next; np != &nplist; np = npnext)
          {
            npnext = np->next;
            if (np->sbuf != NULL)
              {
                nptr = np->sbuf->data;

                if (N_send(np->nconn, nptr->datum, nptr->dlen) != NET_OK)
                    NP_close(np);
                else
                    NP_release(&(np->sbuf), FALSE);
            }
        }
  /* network provider */
      mask = NP_buildmask();
      time.tv_sec   = 0;
      time.tv_usec = 5001;

      if ((n = select(16, &mask, 0, 0, &time)) < 0)
           if (errno != EINTR)     TS_errorshutdown();
      if (n > 0)
        {
          if (mask & (1 << (nplist.nconn->socket)))
            {
              if (N_accept(&nconn, nplist.nconn->socket) != NET_OK)
                   TS_errorshutdown();
              else
                {
                  if (NP_accept(nconn) != NET_OK)
                       N_close(nconn);
                }
            }     /* new TS user */

          for (np = nplist.next; np != &nplist; np = npnext)
            {
              npnext = np->next;
              if (mask & (1 << np->nconn->socket))
                {
                  if ((n = N_receive(np->nconn, ndatum, NET_DATA_SIZE))
                          < NET_OK)
                       NP_close(np);
                  else
                       NP_input(np, ndatum, n);
                }
            }     /* for nplist */
        }     /* n > 0 */

  /* timers */
      for (tp = tplist.next; tp != &tplist; tp = tpnext)
        {
          tpnext    = tp->next;

          if ((tp->timp != NULL) && (tp->timp->time == 0))
               TP_expired(tp);
        }

      for (np = nplist.next; np != &nplist; np = npnext)
        {
          npnext = np->next;

          if ((np->timp != NULL) && (np->timp->time == 0))
               NP_expired(np);
        }
    }     /* forever loop */
}
```